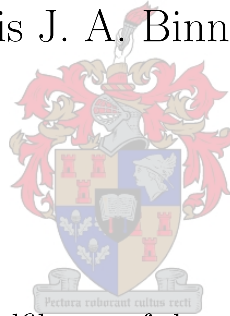
Network Reliability as a result of Redundant Connectivity

Francois J. A. Binneman



Thesis presented in partial fulfilment of the requirements for the degree
Master of Science
in the inter-departmental programme of Operational Analysis
at the University of Stellenbosch, South Africa

Supervisor: Prof JH van Vuuren

March 2007

Stellenbosch University http://scholar.sun.ac.za

Stellenbosch University http://scholar.sun.ac.za

Declaration

I, the undersigned, hereby declare that the work contained in this thesis is my own original work and that I have not previously in its entirety or in part submitted it at any university for a degree.

Signature: ______________________ Date: ______________________

Stellenbosch University http://scholar.sun.ac.za

Abstract

There exists, for any connected graph G, a minimum set of vertices that, when removed, disconnects G. Such a set of vertices is known as a *minimum cut-set*, the cardinality of which is known as the *connectivity number* $\kappa(G)$ of G. A *connectivity preserving* [*connectivity reducing*, respectively] spanning subgraph $G' \subseteq G$ may be constructed by removing certain edges of G in such a way that $\kappa(G') = \kappa(G)$ [$\kappa(G') < \kappa(G)$, respectively]. The problem of constructing such a connectivity preserving or reducing spanning subgraph of minimum weight is known to be *NP*–complete.

This thesis contains a summary of the most recent results (as in 2006) from a comprehensive survey of literature on topics related to the connectivity of graphs.

Secondly, the computational problems of constructing a minimum weight connectivity preserving or connectivity reducing spanning subgraph for a given graph G are considered in this thesis. In particular, three algorithms are developed for constructing such spanning subgraphs. The theoretical basis for each algorithm is established and discussed in detail. The practicality of the algorithms are compared in terms of their worst-case running times as well as their solution qualities. The fastest of these three algorithms has a worst-case running time that compares favourably with the fastest algorithm in the literature.

Finally, a computerised decision support system, called *Connectivity Algorithms*, is developed which is capable of implementing the three algorithms described above for a user-specified input graph.

Opsomming

Daar bestaan, vir enige samehangende grafiek G, 'n kleinste versameling punte wat, wanneer dit verwyder word, G in komponente ontbind. So 'n versameling punte staan as 'n *minimum snit-versameling* bekend, en die kardinaliteit daarvan staan as die *samehangendheidsgetal* $\kappa(G)$ van G bekend. 'n *Samehangendheids-behoudende* [*samehangendheids-reduserende*, repektiewelik] spangrafiek $G' \subseteq G$ kan gekonstrueer word deur sekere lyne uit G op so 'n manier te verwyder dat $\kappa(G') = \kappa(G)$ [$\kappa(G') < \kappa(G)$, repektiewelik]. Die probleem om so 'n samehangendheids-behoudende of -reduserende spangrafiek van minimum gewig te konstrueer, is *NP*–volledig.

Hierdie tesis bevat 'n opsomming van die mees onlangse resultate (soos in 2006) van 'n omvangryke oorsig van literatuur oor onderwerpe verwant aan die samehangendheid van grafieke.

Tweedens word die berekeningsprobleme om 'n minimum-gewig samehangendheids-behoudende of -reduserende spangrafiek van 'n gegewe grafiek G te konstrueer, in hierdie tesis beskou. Daar word in die besonder drie algoritmes vir die konstruksie van sulke spangrafieke ontwikkel. Die teoretiese grondslag vir elke algoritme word daargestel en breedvoerig bespreek. Die praktiese nut van die algoritmes word onderling vergelyk in terme van hul slegste–geval looptyd asook die kwaliteit van oplossings met hul verkry. Die vinnigste van hierdie drie algoritmes het 'n slegste–geval looptyd wat gunstig met dié van die vinnigste algoritme in die literatuur vergelyk.

Laastens word 'n besluitneming steunstelsel, genaamd *Connectivity Algorithms*, ontwikkel, wat die vermoë het om die drie algoritmes soos bo beskryf, rekenaarmatig te implementeer vir 'n gebruiker-gespesifiseerde toevoergrafiek.

Stellenbosch University http://scholar.sun.ac.za

Acknowledgements

The author hereby wishes to express his gratitude towards the following for their support and guidance during the writing of this thesis:

- My Heavenly Father, for His guidance and love, making this thesis possible. *Soli Deo Gloria!*
- Professor Jan van Vuuren for his guidance, dedication and time, and his valuable feedback throughout this thesis.
- My family and friends for their loyal support during the good and the hard times.

Stellenbosch University http://scholar.sun.ac.za

Table of Contents

List of Figures xi

List of Tables xv

List of Algorithms xvii

Glossary xix

Reserved Symbols xxv

1 Introduction 1

1.1 Introduction to the problem . . . 1

1.2 Informal problem description . . . 1

1.3 Objectives of this thesis . . . 2

1.4 Thesis Layout . . . 2

2 Basic Concepts in Graph and Complexity Theory 3

2.1 Basic Graph Theoretic Concepts . . . 3

2.1.1 Walks, trails, paths and fans . . . 4

2.1.2 Neighbourhoods . . . 4

2.1.3 Isomorphisms and Subgraphs . . . 5

2.1.4 Adjacency matrices and edge weights . . . 6

2.1.5 Graph unions and joins . . . 7

2.1.6 Special Graphs . . . 7

2.1.7 Connectedness . . . 10

2.1.8 Vertex Splitting . . . 11

2.1.9 Independence number . . . 11

2.2 Basic Concepts in Complexity Theory . . . 12

2.2.1 Algorithmic complexity . . . 12

2.2.2 The classes P, NP and co-NP . . . 13

2.2.3 Polynomial time reducibility, NP-hardness and NP-completeness . . . 14

2.2.4 Computation problems . . . 15

2.3 Chapter Summary . . . 16

3 Literature Survey 17

3.1 Connectivity and edge-connectivity . . . 17

3.2 Menger's Theorem . . . 17

3.3 Computing $\kappa(G)$ and $\lambda(G)$ . . . 19

3.4 Computing $\lambda(G)$ with high probability . . . 19

3.5 k-Connected and k-Edge-Connected Graphs . . . 20

3.6 Construction of k-connected and k-edge connected graphs . . . 22

3.6.1 Construction from basic graphs . . . 23

3.6.2 Construction by adding edges . . . 23

3.6.3 Expansion of a k-connected graph . . . 23

3.7 Minimally and critically connected graphs . . . 24

3.8 Disconnecting a graph into more than two components . . . 25

3.9 The average connectivity number of a graph . . . 26

3.10 Uniformly connected graphs . . . 28

3.11 Construction using approximation algorithms . . . 29

3.12 Chapter Summary . . . 29

4 Spanning Subgraphs with Connectivity Number $\leq k$ 31

4.1 Finding the connectivity number of a graph . . . 31

4.2 Finding a minimum cut-set of a graph . . . 32

4.2.1 Working of Algorithm *Cut-Vertex Set* . . . 32

4.2.2 Time Complexity of Algorithm *Cut-Vertex Set* . . . 35

4.3 Finding disjoint paths in a graph . . . 36

4.3.1 Ford's Algorithm . . . 36

4.3.2 Shortest Augmenting Path Algorithm . . . 38

4.3.3 Converting a graph to a directed graph . . . 41

4.3.4 Constructing internally disjoint paths for a directed graph . . . 42

4.3.5 Putting it all together . . . 46

4.4 Implementation of Whitney's Theorem . . . 49

4.5 Removing the most expensive edge first . . . 51

4.5.1 Construction of a spanning subgraph G' of G, with $\kappa(G') = \kappa(G)$ . . . 51

4.5.2 Construction of a spanning subgraph G' of G, with $\kappa(G') < \kappa(G)$ . . . 52

4.6 Constructing spanning subgraphs by means of $F(x, U)$ fans . . . 55

4.6.1 Working of Algorithm *Fan* . . . 55

4.6.2 Construction of a spanning subgraph G' of G, with $\kappa(G') \leq \kappa(G)$ . . . 60

4.7 Comparison of Algorithms . . . 64

4.8 Chapter Summary . . . 65

Stellenbosch University http://scholar.sun.ac.za

5 Decision Support System 67

5.1 Technical aspects & limitations of *Connectivity Algorithms* 67

5.2 Introduction to the system components 68

5.3 A worked example 69

5.3.1 Implementation of Algorithm *Whitney* 72

5.3.2 Implementation of Algorithm *MEEF* 76

5.3.3 Implementation of Algorithm *Fan* 79

5.3.4 Summary of results obtained 83

5.4 Case study: The connectivity of a Spider's Web 84

5.5 Chapter Summary 91

6 Conclusion 93

6.1 Thesis Summary 93

6.2 Future Work 94

References 95

A How to use the CD 99

B Source Code for the Program *Connectivity Algorithms* 101

Index 131

Stellenbosch University http://scholar.sun.ac.za

List of Figures

2.1 Graphical representation of an undirected and directed graph . . . 3
2.2 Graphical representation of an $F(x, U)$ fan . . . 4
2.3 Isomorphism and equality in graphs . . . 5
2.4 A subgraph, spanning subgraph and induced subgraph of a graph . . . 6
2.5 A subdigraph, spanning subdigraph and induced subdigraph of a digraph . . . 6
2.6 The deletion of a vertex and an edge subset . . . 7
2.7 The adjacency matrix of a graph . . . 7
2.8 The union and join of two graphs . . . 8
2.9 Graphical representation of a path and a cycle . . . 8
2.10 Illustration of a regular graph and a perfect matching . . . 8
2.11 Graphical representation of a null graph of order 10 . . . 9
2.12 A complete graph . . . 9
2.13 Graphical representation of multipartite and bipartite graphs . . . 9
2.14 Illustration of a tree . . . 10
2.15 Illustration of the notion of a wheel . . . 10
2.16 Graphical representation of a pseudograph. . . . 10
2.17 Components in a graph. . . . 11
2.18 A bridge and cut-vertex in a connected graph. . . . 11
2.19 Illustration of a 3-vertex splitting . . . 12
2.20 The independence number of a graph and the notion of a clique. . . . 12
2.21 The classes P, NP and co-NP. . . . 13
2.22 The classes P, NP, co-NP and NP-complete. . . . 14

3.1 The graph G_{17} . . . 18
3.2 The graph G_{18} . . . 18
3.3 The cycle C_4 . . . 20
3.4 The 3-connected hypercube Q_3 . . . 21
3.5 The graph G_{19} accompanied with an orthogonal representation of the graph . . . 22
3.6 Graphical representation of graphs with an edge-connectivity number of 1 . . . 25
3.7 The graphs G_{21} and G_{22} . . . 27

3.8 Graphical representation of graphs of order and size 5. 27

4.1 The graphs G_{25} and G'_{25} 33
4.2 Progress of the calculation of the distance labels for the graph G_{26} 38
4.3 Maximum flow operations on the graph G_{27} 40
4.4 Conversion of an undirected graph to a directed graph 42
4.5 The adjacency matrices of the graphs G_{28} and G'_{28} 42
4.6 The graphs G_{29} and G'_{29} and path augmentation steps of Algorithm 6 44
4.7 The symmetric traversal matrix *arcCounter* 46
4.8 The graph G_{30} and steps taken by Algorithm 9. 47
4.9 Obtaining internally disjoint paths in the graph G_{30} 48
4.10 Graphical representations of the graphs G_{31} and G'_{31} 50
4.11 The graph G_{32} 52
4.12 Graphical representations of the cases considered in Theorem 4.1 54
4.13 Graphical representations of the graphs relevant to Example 4.11 55
4.14 Graphical representations of the cases considered in Theorem 4.3 57
4.15 Tree representation of the cases considered in Theorem 4.3 58
4.16 The graph G_{34} and spanning subgraph G'_{34} 60
4.17 Graphical representation of the different steps of Algorithm 15 62
4.18 Different versions of the graph G''_{34}, obtained as output from Algorithm 15 63

5.1 Main window of the program *Connectivity Algorithms* 68
5.2 The window, *Load New Graph* 69
5.3 The graph G_{34} 70
5.4 The constructed adjacency matrix for the graph G_{34} 70
5.5 The coordinates for the vertices of the graph G_{34} 70
5.6 Visual representation of the loading process of the file *testgraph.xls* 71
5.7 The graph G_{34} 71
5.8 The graph G_{34} (with a minimum cut-set) 72
5.9 The window *Enter the desired connectivity number* 72
5.10 Notification window stating that a cheaper spanning subgraph could not be constructed 72
5.11 The file *testgraphOutput.xls*, displaying an adjacency matrix 73
5.12 The 4-connected spanning subgraph of G_{34} constructed using Whitney's Algorithm . . . 73
5.13 The 3-connected spanning subgraph of G_{34} constructed using Algorithm Whitney 74
5.14 The 2-connected spanning subgraph of G_{34} constructed using Algorithm *Whitney* 74
5.15 The 1-connected spanning subgraph of G_{34} constructed using Algorithm *Whitney* 75
5.16 The 0-connected spanning subgraph of G_{34} constructed using Algorithm *Whitney* 75
5.17 The 5-connected spanning subgraph of G_{34} constructed using Algorithm *MEEF* 76
5.18 The 4-connected spanning subgraph of G_{34} constructed using Algorithm *MEEF* 76

5.19 The 3-connected spanning subgraph of G_{34} constructed using Algorithm *MEEF* 77
5.20 The 2-connected spanning subgraph of G_{34} constructed using Algorithm *MEEF* 77
5.21 The 1-connected spanning subgraph of G_{34} constructed using Algorithm *MEEF* 78
5.22 The 0-connected spanning subgraph of G_{34} constructed using Algorithm *MEEF* 78
5.23 The 5-connected spanning subgraph of G_{34} constructed using Algorithm *Fan* 79
5.24 The 4-connected spanning subgraph of G_{34} constructed using Algorithm *Fan* 79
5.25 The 3-connected spanning subgraph of G_{34} constructed using Algorithm *Fan* 80
5.26 The 2-connected spanning subgraph of G_{34} constructed using Algorithm *Fan* 80
5.27 The 1-connected spanning subgraph of G_{34} constructed using Algorithm *Fan* 81
5.28 The 0-connected spanning subgraph of G_{34} constructed using Algorithm *Fan* 81
5.29 Graphical representation of a graph that does not have a complete cut-set 82
5.30 A message notifying the user that no complete cut-set exists in the input graph 82
5.31 The *Output* listing all minimum cut-sets in the input graph 82
5.32 Graphical representation of the k-connectivity level vs. the weight improvement 83
5.33 Graphical representation of the connectivity number vs. the weight improvement 84
5.34 Case Study: Graphical representation of the file *Spider21.xls* depicting a spider's web . . 84
5.35 Case Study: The 2-connected spanning subgraph constructed using Algorithm *MEEF* . 85
5.36 Case Study: The 1-connected spanning subgraph constructed using Algorithm *MEEF* . 86
5.37 Case Study: The 0-connected spanning subgraph constructed using Algorithm *MEEF* . 86
5.38 Case Study: The 2-connected spanning subgraph constructed using Algorithm *Fan* . . . 87
5.39 Case Study: The 1-connected spanning subgraph constructed using Algorithm *Fan* . . . 87
5.40 Case Study: The 3-connected spanning subgraph constructed using Algorithm *MEEF* . 88
5.41 Case Study: The 2-connected spanning subgraph constructed using Algorithm *MEEF* . 88
5.42 Case Study: The 1-connected spanning subgraph constructed using Algorithm *MEEF* . 89
5.43 Case Study: The 0-connected spanning subgraph constructed using Algorithm *MEEF* . 89
5.44 Case Study: The 2-connected spanning subgraph constructed using Algorithm *Fan* . . . 90
5.45 Case Study: The 1-connected spanning subgraph constructed using Algorithm *Fan* . . . 90

Stellenbosch University http://scholar.sun.ac.za

List of Tables

3.1 Bounds of $(p+k-2)/2$ on $\delta(G)$ in Theorem 3.10 22

3.2 Exact values of $q = q_{n,3}(p)$ for $\ell = 3$, $n \in \{1, \ldots, 5\}$ as well as bounds on $q_{6,3}(p)$ 26

4.1 The minimum cut-sets of the graph G_7 in Example 4.1 34

4.2 Benchmark tests for Algorithm *Cut-Vertex Set* on complete graphs 35

4.3 The list of paths used in the construction of the graph G'_{31} 50

4.4 The list of paths constructed using Algorithm 14 and inserted into the graph G'_{34} 61

4.5 The list of paths used to construct the graph G''_{34} 62

4.6 Summary of the worst-case running times of Algorithms *Whitney*, *MEEF* and *Fan* . . . 64

5.1 The weight improvement obtained for each k-connected graph 83

5.2 The weight improvement obtained for each subgraph G constructed, such that $\kappa(G) = k$ 84

A.1 Description of a list of graphs included in the folder *Graphs* 99

Stellenbosch University http://scholar.sun.ac.za

List of Algorithms

1 $D_{\text{clique}}(G, k)$ . . . 14

2 $C_{\text{clique}}(G)$ . . . 15

3 Computing $\kappa(G)$ of a graph G . . . 31

4 Cut-Vertex Set . . . 33

5 Ford's Algorithm . . . 37

6 Shortest Augmenting Path Algorithm . . . 39

7 Undirected Graph to Directed Graph . . . 42

8 Constructing Internally Disjoint Paths in a Directed Graph . . . 45

9 Finding Internally Disjoint Paths in an Undirected Graph . . . 46

10 Directed Paths to Undirected Paths . . . 48

11 Whitney's Algorithm . . . 50

12 MEEF: Connectivity Preserving . . . 52

13 MEEF: Connectivity Reducing . . . 54

14 Fan: Connectivity Preserving . . . 56

15 Fan: Connectivity Reducing . . . 61

Stellenbosch University http://scholar.sun.ac.za

Stellenbosch University http://scholar.sun.ac.za

Glossary

Acyclic: A *graph* G is called *acyclic* if it does not contain any *cycles*.

Adjacency Matrix: Let G be a *graph* whose *vertices* have been (arbitrarily) ordered $V = (v_1, v_2, \ldots, v_p)$. The *adjacency matrix* $A = [a_{ij}]$ of G is a $p \times p$ matrix with entries $a_{ij} = 0$ if $v_i v_j \notin E$, else $a_{ij} = w_{ij}$, where w_{ij} is the *weight* of the *edge joining* the *vertices* v_i and v_j.

Adjacent: Two *vertices* of a *graph* G are said to be *adjacent* if there exists an *edge* of G *joining* the two *vertices*.

Algorithmic Complexity: *Algorithmic complexity* is a measure of the number of basic operations performed, and the memory expended by an algorithm. If a problem cannot (with current knowledge) be solved by a polynomial time algorithm, it is referred to as an intractable or hard problem, otherwise it is called a tractable problem.

Arc: An *arc* $e = \{xy\}$ is considered to be a *directed edge* from *vertex* x to *vertex* y; *vertex* y is called the *head* and *vertex* x is called the *tail* of the *edge*.

Arc Set: The set $A(D)$, comprised of all the *arcs* of a *digraph* D, is called the *arc set* of D.

Average Connectivity Number: The *average connectivity number* $\overline{\kappa}(G)$ of a *graph* G is the expected number of *vertices* that have to be removed in order to disconnect an arbitrary pair of non-adjacent *vertices* in G. Hence, $\overline{\kappa}(G) = \frac{\sum_{u,v} \kappa(u,v)}{\binom{p}{2}}$, for all *vertices* u and v in $V(G)$.

Bipartite: An *n-partite graph* is called *bipartite* if $n = 2$.

Bridge: An *edge* e is called a *bridge* of a *graph* G if the *graph* $G - e$ has more *components* than does G.

Cardinality: The number of elements in a set is called its *cardinality*.

Circuit: A *circuit* is a closed *trail*.

Clique: A *clique* is a *complete subgraph* of a *graph* G that is not an *induced subgraph* of any other *complete subgraph* of G.

Clique Number: The maximum *order* of a *clique* in a *graph* G is called the *clique number* of G, denoted $\omega(G)$.

Closed Neighbourhood: The *closed neighbourhood* of a *vertex* v in a *graph* G is the set of all *vertices adjacent* to v in G, as well as v itself, and is denoted $N_G[v]$. The *closed neighbourhood* of a *vertex set* S in G is defined as $N_G[S] = \{N_G[v] : v \in S\}$.

Closed Walk: A *closed walk* is a walk in which the start and end *vertices* are the same.

Complement: The *complement* $\overline{G}$ of a *graph* G is the *graph* for which $V(\overline{G}) = V(G)$ and $e \in E(\overline{G})$ if and only if $e \notin E(G)$.

Complete Graph: A *complete graph* of *order* n, denoted by K_n, is a *graph* in which every pair of *vertices* are *adjacent*.

Component: A *subgraph* H of a *graph* G is called a *component* of G if H is a maximally *connected subgraph* of G.

Connected: For *vertices* u and v of a *graph* G, u is said to be *connected* to v if G contains a $u-v$ *path*. The *graph* G is called a *connected graph* if every pair of its *vertices* u and v are *connected*.

Connectivity Number: The *Connectivity number* $\kappa(G)$ of a *graph* G is the minimum *cardinality* of a set S of *vertices* for which $G \setminus S$ is disconnected or is the *trivial graph*.

Connectivity Sequence: The *connectivity sequence* $\boldsymbol{s}$ of a *graph* G of *order* p is the sequence $\boldsymbol{s}$: $\kappa_2(G), \kappa_3(G), \ldots, \kappa_p(G)$, where $\kappa_i(G)$ denotes the minimum *cardinality* of a set of *vertices* whose removal from G produces a *graph* with at least i *components* or a *graph* with fewer than i vertices.

Critically k-connected: A *graph* G is said to be *critically k-connected* if $\kappa(G) \geq k$ and $\kappa(G-v) < k$ for every *vertex* $v \in V(G)$, where $\kappa(G)$ denotes the connectivity number of the graph G.

Critically k-edge-connected: A *graph* G is said to be *critically k-edge-connected* if $\lambda(G) \geq k$ and $\lambda(G-v) < k$ for every *vertex* $v \in V(G)$, where $\lambda(G)$ denotes the *edge-connectivity number* of the *graph* G.

Cut-set: A *cut-set* is a set of *vertices* whose removal disconnects a *connected graph*.

Cycle: A *cycle* is a *walk* of length $n \geq 3$ in which the begin- and end-*vertices* are the same, but in which no other *vertices* repeat. A *graph* consisting of a single *cycle* of length n is so called and denoted C_n.

Degree: The *degree* of a *vertex* v of a *graph* G is the *cardinality* of the *open neighbourhood* of v in G, and is denoted $\deg_G v$.

Degree Sequence: A sequence d_1, $d_2, \ldots,$ d_n of nonnegative integers is called a *degree sequence* of a *graph* G if the *vertices* of G can be labelled v_1, $v_2, \ldots,$ v_n such that $\deg\ v_i = d_i$.

Deletion: The *deletion* of a non-empty *vertex* subset $S \subseteq V(G)$ from a *graph* G is the *subgraph* with *vertex set* $V(G)\backslash S$ and *edge set* $\{uv \in E(G) :\ u, v \notin S\}$. Such a *subgraph* is written as $G \setminus S$. If a single *vertex* v is removed from the set $V(G)$, the resulting *subgraph* may be written as $G-v$. For any *edge* subset $J \subseteq E(G)$ the *deletion* of the *edge set* J from G, denoted by $G \setminus J$, is the *spanning subgraph* of G with *edge set* $E(G)\backslash J$. If a single *edge* e is removed from the set $E(G)$, the resulting *subgraph* may be written as $G-e$.

Digraph: Short for *directed graph*.

Directed Graph: A *directed graph* is an ordered pair (V, A), where V is a set of *vertices* and A is a set of ordered pairs of *vertices* called *arcs*.

Disconnected: A *graph* that is not *connected* is said to be *disconnected*.

Edge: An *edge* is a 2-element subset of the *vertex set* of a *graph*. *Edges* are indicated by inter-connecting lines between *vertices* in graphical representations of a *graph*.

Edge-connectivity Number: The *edge-connectivity number* $\lambda(G)$ of a *graph* G is the minimum *cardinality* of a set S_J of *edges* for which $G \setminus S_J$ is *disconnected* or is the *trivial graph*.

Edge Set: The set $E(G)$, comprised of all the *edges* of a *graph* G, is called the *edge set* of G.

Equal: Two *graphs* G and H are said to be *equal*, written as $G = H$, if $V(G) = V(H)$ and $E(G) = E(H)$.

End-vertex: If the *degree* of a *vertex* is 1, then it is called an *end-vertex*.

Forest: A *graph* that is *acyclic*, is called a *forest*, and consists of a number of *disconnected trees*.

Graph: A *graph* is a finite, nonempty set of elements, called *vertices*, together with a (possibly empty) set of 2-element subsets of the *vertex set* called *edges*. A *graph* may be represented graphically as a set of points with inter-connecting lines.

 Stellenbosch University http://scholar.sun.ac.za

Incident: A *vertex* v and *edge* e of a *graph* G is said to be *incident*, if e *joins* v to another *vertex* in G.

Independent Set: A set of *vertices* in a *graph*, none of which are connected by an *edge*.

Independence Number: The maximum *cardinality* over all *maximal independent sets* of a *graph* G is called the *independence number* of G and is denoted $\beta(G)$.

Induced Subdigraph: For a non-empty subset $S \subseteq V(D)$ of a *digraph* D the so-called *induced subdigraph* of S in D, denoted $\langle S\rangle_D$, is the *subdigraph* of D with *vertex set* $V(\langle S\rangle_D) = S$ and *arc set* $A(\langle S\rangle_D) = \{uv \in A(D) : u, v \in S\}$.

Induced Subgraph: For a non-empty subset $S \subseteq V(G)$ of a *graph* G the so-called *induced subgraph* of S in G, denoted $\langle S\rangle_G$, is the *subgraph* of G with *vertex set* $V(\langle S\rangle_G) = S$ and *edge set* $E(\langle S\rangle_G) = \{uv \in E(G) : u, v \in S\}$.

Isomorphic: Two *graphs* G and H are said to be *isomorphic*, written as $G \cong H$, if there exists a one-to-one mapping $\phi : V(G) \to V(H)$ such that $uv \in E(G)$ if and only if $\phi(u)\phi(v) \in E(H)$.

Join: The *join* of two *graphs* G_1 and G_2, written as $G_1 + G_2$, is defined as the *union* of G_1 and G_2 together with all *edges* uv for which $u \in V(G_1)$ and $v \in V(G_2)$. Two *vertices* of a *graph* G are said to be *joined* in G if the *edge* uv is contained in the *edge set* of G.

k-connected: A *connected graph* G is said to be *k-connected* (for some $k \geq 1$) if the removal of fewer than k *vertices* always produces a nontrivial *connected graph*.

k-edge-connected: A *connected graph* G is said to be *k-edge-connected* (for some $k \geq 1$) if the removal of fewer than k *edges* always produces a nontrivial *connected graph*.

Loop: A *loop* is an *edge* that *joins* a *vertex* to itself.

ℓ-connectivity Number: For an integer $\ell \geq 2$ and a *graph* G of *order* $p \geq \ell$, the *ℓ-connectivity number* $\kappa_\ell(G)$ is the minimum *cardinality* of a set U of *vertices* whose removal from G produces a *graph* with at least ℓ components or a *graph* with fewer than ℓ *vertices*.

ℓ-edge-connectivity Number: For an integer $\ell \geq 2$ and a *graph* G of *order* $p \geq \ell$, the *ℓ-edge-connectivity number* $\lambda_\ell(G)$ is the minimum *cardinality* of a set S of *edges* whose removal from G produces a *graph* with at least ℓ components.

ℓ-way Cut: For an integer $\ell \geq 2$, a *ℓ-way cut* of a *graph* G is a partition of $V(G)$ into ℓ non-empty disjoint subsets $\{V_1, V_2, \ldots, V_\ell\}$.

Maximal Independent Set: An *independent set* S of *vertices* in a *graph* G is called a *maximal independent* set if S is not a proper subset of any other *independent set* of G.

Minimally k-connected: A *graph* G is said to be *minimally k-connected* if $\kappa(G) \geq k$ and $\kappa(G-e) < k$ for every *edge* $e \in E(G)$.

Minimally k-edge-connected: A *graph* G is said to be *minimally k-edge-connected* if $\lambda(G) \geq k$ and $\lambda(G-e) < k$ for every *edge* $e \in E(G)$.

Minimum ℓ-way Cut: A *minimum ℓ-way cut* is a ℓ-way cut that minimizes the weight sum of the *edges* between the non-empty disjoint subsets $\{V_1, V_2, \ldots, V_\ell\}$ into which $V(G)$ is divided.

Multipartite: An *n-partite graph* is called *multipartite* if $n > 2$.

n-partite: A *graph* G is called *n-partite*, for some $n \geq 2$, if the *vertex* set of G may be partitioned into n subsets, such that no *edge* of G joins *vertices* in the same subset.

Null Graph: A *null graph* is a *graph* with no *edges*.

Open Neighbourhood: The *open neighbourhood* of a *vertex* v in a *graph* G is the set of all *vertices adjacent* to v in G, and is denoted $N_G(v)$. The *open neighbourhood* of a set S is defined as $N_G(S) = \{N_G(v) : v \in S\}$.

Open Walk: An *open walk* is a *trail* in which the start and end *vertices* differ.

Order: The *cardinality* of the *vertex set* of a *graph* G is called the *order* of G.

Path: A *walk* in which no *vertex* is repeated is called a *path.* A *graph* solely consisting of a *path* of *order* n is so called and denoted P_n.

Pseudograph: A *pseudograph* G is a *graph* in which both multiple edges and *loops* are permitted.

Regular: A *graph* G is called r*-regular* if each *vertex* of G has *degree* r, for some $r \in \mathbb{N}_0$. A *graph* is also referred to as *regular* if it is r*-regular* for some $r \in \mathbb{N}_0$.

Sequence of Strong Connectivity Numbers: The *sequence of strong connectivity numbers* $\boldsymbol{s}$ for a *digraph* D of *order* p is the sequence $\boldsymbol{s}$: $\kappa_2(D), \kappa_3(D), \ldots, \kappa_p(D)$, where $\kappa_i(D)$ denotes the minimum *cardinality* of a set of *vertices* whose removal from D produces a *digraph* with at least i strong *components* or a *digraph* with at most $i-1$ vertices.

Size: The *cardinality* of the *edge set* of a *graph* G is called the *size* of G.

Spanning Subdigraph: A *digraph* H is called a *spanning subdigraph* of D if $V(H) = V(D)$ and $A(H) \subseteq A(D)$.

Spanning Subgraph: A *graph* H is called a *spanning subgraph* of G if $V(H) = V(G)$ and $E(H) \subseteq E(G)$.

Spanning Forest: A *spanning forest* of a *graph* G is a *spanning graph* of G for which every *component* is a *tree.*

Spanning Tree: A *spanning tree* of a *graph* G is a *connected*, *acyclic subgraph* containing all the *vertices* of G.

Star: The *bipartite graph* $K_{1,n} \cong K_{n,1}$ is a often called an n*-star* for some $n \in \mathbb{N}$.

Strong Component: A *strong component* of a *digraph* D is a maximal *induced subdigraph* of D which is *strongly connected.*

Strong Independence Number: The *strong independence number* $\beta_S(D)$ of a *digraph* D is the maximum *cardinality* of a set S of *vertices* of D for which the *subdigraph* $\langle S \rangle_D$ is *acyclic.*

Strong ℓ-connectivity Number: The *strong* ℓ*-connectivity number* $\kappa_\ell(D)$ of a *digraph* D denotes the minimum *cardinality* of a a set of of *vertices* whose deletion from D produces a *digraph* with at least ℓ *strong components* or a *digraph* with at most $\ell - 1$ *vertices.*

Strong ℓ-arc-connectivity Number: The *strong* ℓ*-arc-connectivity number* $\lambda_\ell(D)$ of a *digraph* D as the minimum number of *arcs* whose deletion from D produces a *digraph* with at least ℓ *strong components* or a *digraph* with at most $\ell - 1$ *vertices.*

Strongly Connected: A *digraph* D is *strongly connected* if, for every pair of *vertices* u and v of D, there is a directed *path* from u to v.

Strongly (n,ℓ)-connected: A *digraph* D is said to be *strongly* (n,ℓ)*-connected* if $\kappa_\ell(D) \geq n$ for some $n \geq 0$.

Subdigraph: A *digraph* H is called a *subdigraph* of a *digraph* if $V(H) \subseteq V(D)$ and $A(H) \subseteq \{uv \in A(D) : u, v \in V(H)\}$.

Subgraph: A *graph* H is called a *subgraph* of G if $V(H) \subseteq V(G)$ and $E(H) \subseteq \{uv \in E(G) : u, v \in V(H)\}$.

Trail: A *walk* in which no *edge* is repeated is called a *trail.*

Tree: A *tree* is an *acyclic connected graph.*

Trivial Graph: A *graph* with only one *vertex* is called a *trivial graph.*

Trivial Tree: A *tree* is *trivial* if it consists of only one *vertex.*

 Stellenbosch University http://scholar.sun.ac.za

Undirected Graph: A *graph* with *undirected edges*.

Union: The *union* of two *graphs* G_1 and G_2, written as $G_1 \cup G_2$, is the *graph* G with *vertex set* $V(G) = V(G_1) \cup V(G_2)$ and *edge set* $E(G) = E(G_1) \cup E(G_2)$.

Vertex: A *vertex* is a combinatorial element in terms of which a *graph* is defined. *Vertices* are indicated by points in a graphical representation of a *graph*.

Vertex Connectivity Number: The *(vertex) connectivity number* $\kappa(G)$ of a *graph* G is the minimum *cardinality* of a set U of *vertices* for which $G - U$ is *disconnected* or is the *trivial graph*.

Vertex Set: The set comprised of all *vertices* of a *graph* G, is called the *vertex set* of G.

Walk: A *walk* in a *graph* G is an alternating sequence of *incident vertices* and *edges*. The number of *edges* in the *walk* defines its length, while the number of *vertices* defines its *order*.

Weight: The *weight* of an *edge* is a number w_{ij} associated with the *edge* ij of a *graph* G.

Wheel: A *wheel* W_n of *order* n may be defined as the *join* of a *cycle* of *order* n with another *vertex*, sometimes referred to as the hub of the *wheel*.

Stellenbosch University http://scholar.sun.ac.za

Stellenbosch University http://scholar.sun.ac.za

Reserved Symbols

$A(G)$	Adjacency matrix of a graph G.
$\beta(G)$	The independence number of a graph G.
$\beta_S(D)$	The strong independence number of a digraph D.
C_n	A cycle of order n.
$\deg_G v$	The degree of a vertex v in a graph G.
$\Delta(G)$	The maximum vertex degree of a graph G.
$\delta(G)$	The minimum vertex degree of a graph G.
$E(G)$	The edge set of a graph G.
G	A graph $G = (V, E)$, with vertex set V and edge set E.
$\overline{G}$	The complement of a graph G.
$k(G)$	The number of components of a graph G.
$\kappa(u, v)$	The maximum number of internally disjoint u-v paths of a given graph.
$\kappa(G)$	The (vertex) connectivity number of a graph G.
$\overline{\kappa}(G)$	The average (vertex) connectivity number of a graph G.
$\kappa_\ell(D)$	The strong ℓ-connectivity number of a digraph D.
$\kappa_\ell(G)$	The ℓ-connectivity number of a graph G.
K_n	A complete graph of order n.
$K_{p_1,p_2,\ldots,p_t}$	A complete multipartite graph, with partite set cardinalities $p_1 \leq p_2 \leq \cdots \leq p_t$, $t \in \mathbb{N}$.
$\lambda(G)$	The edge connectivity number of a graph G.
$\lambda_\ell(D)$	The strong ℓ-arc-connectivity number of a digraph D.
$\lambda\ell(G)$	The ℓ-edge-connectivity number of a graph G.
$\mu_G(p, q)$	The maximum average connectivity number of a graph G.
$N_G(v)$	The open neighbourhood of a vertex v in a graph G.
$N_G[v]$	The closed neighbourhood of a vertex v in a graph G.
$N_G(S)$	The open neighbourhood of a set $S \subseteq V(G)$ in a graph G.
$N_G[S]$	The closed neighbourhood of a set $S \subseteq V(G)$ in a graph G.
$\omega(G)$	The clique number of a graph G.
p	The order of a given graph — the graph will be clear from context.
P_n	A path of order n.
q	The size of a given graph — the graph will be clear from context.
$V(G)$	The vertex set of a graph G.

Stellenbosch University http://scholar.sun.ac.za

Stellenbosch University http://scholar.sun.ac.za

Chapter 1

Introduction

1.1 Introduction to the problem

It is common practice to incorporate some level of redundancy as fail-safe measure when designing networks for a variety of applications. When incorporating such redundancy into a network there are usually two conflicting objectives: (i) to build in enough redundancy so as to guarantee a certain minimum threshold of fail-safeness in the network, and (ii) to limit the level of redundancy so as to achieve cost-effectiveness. An optimal level of redundancy is therefore a trade-off between achieving these objectives.

Consider, for instance, the road network in a large city. Roadways should be designed so that one can drive from any part of the city to any other part, without making too much of a detour. Also, the planning of which parts of the city to join together by means of roads is very important in terms of efficient traffic flow. Furthermore, if the shortest or most suitable road to one's destination has been damaged or rendered inaccessible due to traffic congestion, one would expect that there should be at least one other road that one is able to take to one's destination — hence a level of redundancy should be present in the road network. Another important aspect of road networks, is that they must be constructed cost effectively. Building such a network requires considerable amounts of capital; hence care should be taken not to incorporate excessive levels of redundancy.

Another example of this conflicting bi-objective phenomenon, but on a larger scale, is present in the design of a national electricity grid. In such a grid cities and power stations have to be interconnected by means of high-voltage power lines. In this application the aim of the national service provider is typically to interconnect the cities and power stations in such a way that, should some number of power lines or power stations (not exceeding a planning threshold) fail simultaneously, all cities still receive electricity from power stations via some other (functional) infrastructure components. However, at the same time the service provider typically aims to minimise the total length of high voltage power lines for the required level of fail-safeness (due to the high cost per unit length of such power lines).

Attempts at achieving a suitable trade-off between the above-mentioned conflicting network design objectives typically involve an in-depth analysis of those parts of the network that are most vulnerable in the sense that if such network parts fail, then the network is disconnected into a number of disjoint components.

1.2 Informal problem description

Networks, such as the ones described above, may be modelled mathematically by means of graphs in which infrastructure hub components, such as street intersections, cities or power stations, are represented by vertices, and where interconnecting network components, such as roads or power lines, are represented by means of edges. In such graphs the edges are typically weighted, indicating the cost in some sense of the corresponding interconnecting network components.

Stellenbosch University http://scholar.sun.ac.za

The problems considered in this thesis are (i) to develop a methodology capable of determining which edges in the model graph should be kept operational at all costs so as to retain a given level of connectivity, and (ii) to answer the question as to which edges should be removed (from the point of view of cost-efficiency in terms of the graph weights) during the constructing of a subgraph with a predetermined, smaller level of connectivity than that of an original graph. It is known that the problem of finding a shortest subgraph with a predetermined, smaller level of connectivity than that of an original graph is *NP*-complete (see Kortsarz & Nutov [34]).

1.3 Objectives of this thesis

Seven objectives are pursued in this thesis:

Objective I: To introduce a framework within which the above mentioned connectivity problems may be studied and solved, and to provide the reader with precise definitions of the various concepts that are required in this field of study.

Objective II: To provide the reader with a thorough survey of literature on topics related to the connectivity of graphs, highlighting the latest results, algorithms and avenues of investigation (as in 2006).

Objective III: To develop the prerequisite theory underlying any algorithms that form the basis of constructing subgraphs of a specified connectivity level from a given graph.

Objective IV: To develop and implement an algorithm or algorithms capable of constructing a lower weighted subgraph of a given graph, without reducing the level of connectivity.

Objective V: To extend Objective III, by developing and implementing an algorithm or algorithms capable of constructing a lower weighted subgraph with a smaller, user-defined level of connectivity than that of a given graph.

Objective VI: To draw a comparison between the algorithms in Objectives IV and V in terms of their worst-case running times and solution qualities.

Objective VII: To develop a decision support system capable of implementing the algorithms in Objectives IV and V for user-specified input graphs.

1.4 Thesis Layout

This thesis consists of five chapters, in addition to this introductory chapter. In Chapter 2 an overview of basic graph and complexity theoretic concepts used throughout the remainder of this thesis is given. A literature review of topics related to graph connectivity is given in Chapter 3. The main contributions of this thesis may be found in Chapters 4 and 5. Three algorithms capable of constructing a lower weighted subgraph of a given graph with a predefined level of connectivity are developed in Chapter 4. The prerequisite theory underlying each algorithm is also presented in Chapter 4. A newly designed decision support system (DSS) is presented in Chapter 5. This DSS is based on the algorithms of Chapter 4. Finally, the thesis closes with a summary of the contributions made as well as an indication of possible future avenues of investigation in Chapter 6.

Stellenbosch University http://scholar.sun.ac.za

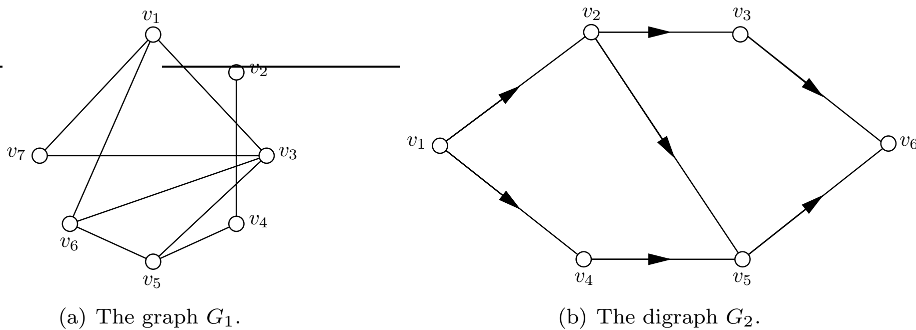
Chapter 2

Basic Concepts in Graph and Complexity Theory

The graph theoretic definitions required for this thesis are introduced in §2.1, and an overview of basic concepts in complexity theory is given in §2.2.

2.1 Basic Graph Theoretic Concepts

A *graph* $G = (V, E)$ is a finite, nonempty set $V(G)$, together with a (possibly empty) set $E(G)$ of 2-element subsets of $V(G)$. The elements of V are called *vertices*, while those of E are called *edges*. The number of vertices in a graph G is called the *order* of G, denoted by $p(G) = |V(G)|$, while the number of edges in G is called the *size* of G, denoted by $q(G) = |E(G)|$. If no ambiguity exists, the order and size of a graph are simply referred to as p and q respectively. A graph of order p and size q is often referred to as a (p, q)-graph. A graph with only one vertex (a $(1, 0)$-graph) is called a *trivial* graph. If the unordered pair $e = \{u, v\}$ is an edge of the graph G, informally written as $e = uv$, it is said that the vertices u and v are *adjacent* in G, that the edge e *joins* u and v, and that e is *incident* with the vertices u and v. A graph having no loops, or multiple edges between the same pair of vertices is referred to as a *simple graph*. A graphical representation of an order 7 graph G_1 of size 9 is shown in Figure 2.1 (a). The vertex set is $V(G_1) = \{v_1, v_2, v_3, v_4, v_5, v_6, v_7\}$ and the edge set is $E(G_1) = \{v_1v_6, v_1v_3, v_1v_7, v_2v_4, v_3v_5, v_3v_6, v_3v_7, v_4v_5, v_5v_6\}$. The vertices v_1 and v_6 are adjacent in G_1, while v_1 and v_2 are not. A *directed graph* is a nonempty set of vertices $V(G)$ and a possibly empty set $E(G)$ of ordered pairs from $V(G)$. Directed graphs are often called *digraphs* for short. A graphical representation of a digraph G_2 consisting of 6 vertices and 7 directed edges is shown in Figure 2.1 (b).



(a) The graph G_1. (b) The digraph G_2.

Figure 2.1: *Graphical representation of an undirected (G_1) and directed (G_2) graph.*

Stellenbosch University http://scholar.sun.ac.za

2.1.1 Walks, trails, paths and fans

A *walk* in a graph G is an alternating sequence of vertices and edges

$$v_0, e_1, v_1, e_2, v_2, \ldots, v_{i-1}, e_i, v_i, \ldots, v_{n-1}, e_n, v_n,$$

also called a v_0-v_n walk, such that $e_i = v_{i-1}v_i$ for $i = 1, 2, \ldots, n$. The number of edges in the walk defines its *length*, while the number of vertices defines its *order*. Non-endpoint vertices of a walk are known as *internal vertices*. When referring to a walk, the edges are often omitted where ambiguity is impossible. An example of a walk in the graph G_1 in Figure 2.1(a) is v_5, v_6, v_3, v_5, v_4. A u-v walk is referred to as an *open walk* if $u \neq v$ or as a *closed walk* if $u = v$. A walk in which no edge is repeated is called a *trail*. A closed trail is referred to as a *circuit*. A walk in which no vertex is repeated is called a *path*. A *cycle* is a walk of length $n \geq 3$ in which the begin- and end-vertices, v_0 and v_n, are the same, but in which no other vertices repeat. Considering the graph G_1 in Figure 2.1(a), the (open) walk v_1, v_7, v_3 is a path of order 3 and length 2, while $v_1, v_7, v_3, v_5, v_6, v_3, v_1$ is a circuit. The vertices v_5, v_6, v_3, v_5 form a cycle of length 3.

A set of paths is said to be *internally disjoint* if none of the paths contain internal vertices of any of the other paths. Consider the directed graph G_2 shown in Figure 2.1 (b). Three v_1-v_6 paths exist, namely $P^{(1)} = v_1,\ v_2,\ v_3,\ v_6$, $P^{(2)} = v_1,\ v_2,\ v_5,\ v_6$ and $P^{(3)} = v_1,\ v_4,\ v_5,\ v_6$. Of these three, a maximum set of internally disjoint paths consists of the two paths $P^{(1)}$ and $P^{(3)}$.

Let U be a subset of the vertices of a graph G and let $x \in V(G) \setminus U$. An $F(x, U)$ *fan* of G is a set of $|U|$ paths starting from x and ending in different vertices of the set U, of which any two paths are internally disjoint. An $F(x, U)$ fan is illustrated in Figure 2.2. Notice that none of the paths from x to the vertices in U share internal vertices.

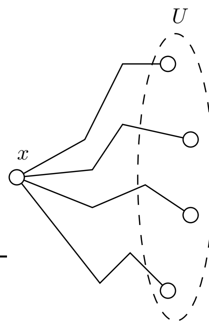


Figure 2.2: *Graphical representation of an $F(x, U)$ fan.*

2.1.2 Neighbourhoods

The *open neighbourhood* of a vertex v in a graph G is defined as the set

$$N_G(v) = \{u \in V(G) :\ uv \in E(G)\},$$

while the *closed neighbourhood* of v in G is defined as

$$N_G[v] = N_G(v) \cup \{v\}.$$

The closed neighbourhood of a set $S \subseteq V(G)$ is defined as $N[S] = \{N[v] :\ v \in S\}$, while the open neighbourhood of a set S is defined as $N(S) = N[S] \setminus S$. For any vertex v in a graph G, the number of vertices adjacent to v, i.e. $|N_G(v)|$, is called the *degree* of v in G, denoted by $\deg_G v$. Note that if the reference to a graph G is clear from the context, the subscript is often omitted, hence written as $\deg\ v$ only. For a directed graph, neigbours of a vertex v are classified as either an *out-neighbour* or *in-neighbour*. A neighbour u of a vertex v in a directed graph G is an out-neighbour if the arc $vu \in E(G)$. Similarly, the vertex u is an in-neighbour if $uv \in E(G)$. If both arcs uv and vu are present, then the vertex u may be referred to as both an in- and an out-neighbour of the vertex v. If the degree of a vertex

is 0, it is called an *isolated vertex*, while if the degree is 1, it is called an end-vertex. The minimum degree of the vertices in G is denoted by $\delta(G)$, while the maximum degree of these vertices is denoted by $\Delta(G)$. The *degree sequence* of a graph G is a sequence of nonnegative integers $d_1, d_2, \ldots, d_n$ such that the vertices of G can be labelled $v_1, v_2, \ldots, v_n$ in such a way that deg $v_i = d_i$. Referring to the graph G_1 in Figure 2.1, the open neighbourhood of the vertex v_5 is $N_{G_1}(v_5) = \{v_3, v_4, v_6\}$, while its closed neighbourhood is $N_{G_1}[v_5] = \{v_3, v_4, v_5, v_6\}$. The graph has no isolated vertices, but v_2 is an end-vertex. The minimum degree of G_1 is therefore $\delta(G_1) = 1$, while the maximum degree is $\Delta(G_1) = 4$. The degree sequence of G_1 is 1, 2, 2, 2, 3, 3, 4 with respective vertex sequence $v_2, v_1, v_4, v_7, v_5, v_6, v_3$.

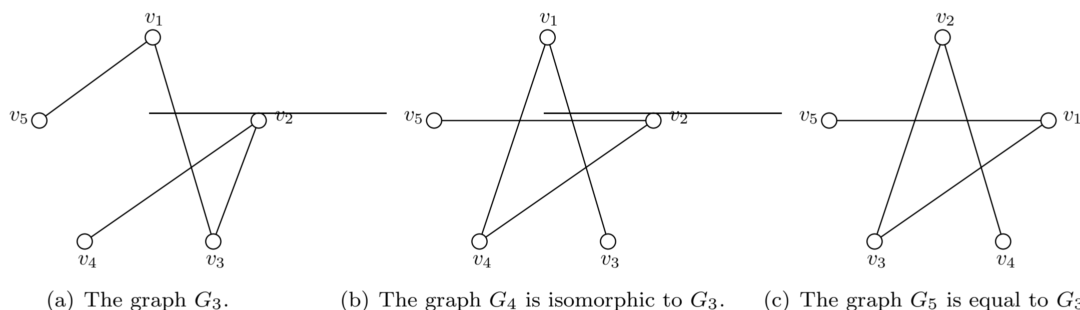
The following theorem, often referred to as the *Fundamental Theorem of Graph Theory*, is probably one of the most well-known results in the discipline and relates the sum total of the degrees and the size of any graph.

Theorem 2.1 *Let G be a (p, q)-graph, with $V(G) = \{v_1, v_2, \ldots, v_p\}$. Then* $\sum_{i=1}^{p} \deg_G v_i = 2q$.

Proof: When the degrees of all the vertices are summed, each edge is counted twice, once for each of the vertices that it joins. ■

2.1.3 Isomorphisms and Subgraphs

Two graphs G and H are called *isomorphic*, written as $G \cong H$, if there exists a one-to-one mapping $\phi : V(G) \to V(H)$ such that $uv \in E(G)$ if and only if $\phi(u)\phi(v) \in E(H)$. The function ϕ is called an *isomorphism*. If ϕ maps G onto itself, it is called an *automorphism*. Two graphs G and H are said to be *equal* if $V(G) = V(H)$ and $E(G) = E(H)$. Therefore, equal graphs are isomorphic, but the converse is not true. The graph G_4 shown in Figure 2.3(b) is isomorphic (but not equal) to G_3, shown in Figure 2.3(a), while G_5, shown in Figure 2.3(c), is both equal and isomorphic to G_3.



(a) The graph G_3. (b) The graph G_4 is isomorphic to G_3. (c) The graph G_5 is equal to G_3.

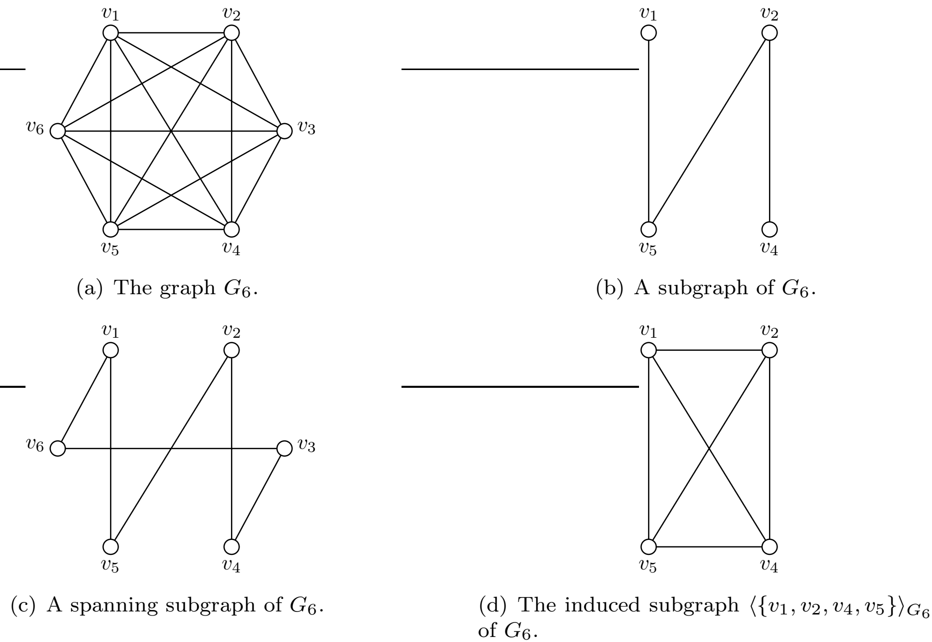
Figure 2.3: *Illustration of isomorphism and equality in graphs.*

A graph H is called a *subgraph* of G if $V(H) \subseteq V(G)$ and $E(H) \subseteq \{uv \in E(G) : u, v \in V(H)\}$, and is called a *spanning subgraph* of G if $V(H) = V(G)$ and $E(H) \subseteq E(G)$. For a non-empty vertex subset $S \subseteq V(G)$ of a graph G the so-called *induced subgraph* of S in G, denoted by $\langle S \rangle_G$, is the subgraph of G with vertex set $V(\langle S \rangle_G) = S$ and edge set $E(\langle S \rangle_G) = \{uv \in E(G) : u, v \in S\}$. The graph shown in Figure 2.4(b) is an example of a subgraph of G_6, shown in Figure 2.4(a), while the graph in Figure 2.4(c) is a spanning subgraph of G_6. Moreover, the induced subgraph $\langle \{v_1, v_2, v_4, v_5\} \rangle_{G_6}$ is illustrated in Figure 2.4(d).

Similar concepts exist for digraphs. A digraph H is called a *subdigraph* of D if $V(H) \subseteq V(D)$ and $A(H) \subseteq \{uv \in A(D) : u, v \in V(H)\}$, and is called a *spanning subdigraph* of D if $V(H) = V(D)$ and $A(H) \subseteq A(D)$. For a non-empty vertex subset $S \subseteq V(D)$ of a digraph D the so-called *induced subdigraph* of S in D, denoted by $\langle S \rangle_D$, is the subgraph of D with vertex set $V(\langle S \rangle_D) = S$ and arc set $A(\langle S \rangle_D) = \{uv \in A(D) : u, v \in S\}$. The digraph shown in Figure 2.5(b) is an example of a subdigraph of D_6, shown in Figure 2.5(a), while the digraph in Figure 2.5(c) is a spanning subdigraph of D_6. Moreover, the induced subdigraph $\langle \{v_1, v_2, v_4, v_5\} \rangle_{D_6}$ is illustrated in Figure 2.5(d).

The *deletion* of a non-empty vertex subset $S \subseteq V(G)$ from a graph G is the subgraph with vertex set $V(G) \backslash S$ and edge set $\{uv \in E(G) : u, v \notin S\}$. Such a subgraph is denoted by $G - S$. For any edge subset

Stellenbosch University http://scholar.sun.ac.za



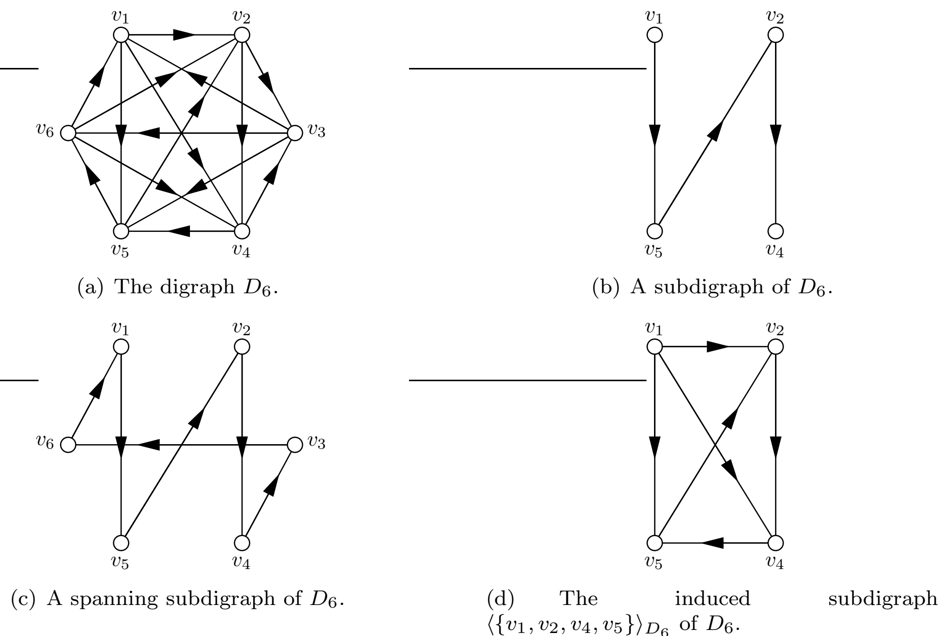
(a) The graph G_6.

(b) A subgraph of G_6.

(c) A spanning subgraph of G_6.

(d) The induced subgraph $\langle\{v_1, v_2, v_4, v_5\}\rangle_{G_6}$ of G_6.

Figure 2.4: *Illustration of a subgraph, spanning subgraph and induced subgraph of graph.*



(a) The digraph D_6.

(b) A subdigraph of D_6.

(c) A spanning subdigraph of D_6.

(d) The induced subdigraph $\langle\{v_1, v_2, v_4, v_5\}\rangle_{D_6}$ of D_6.

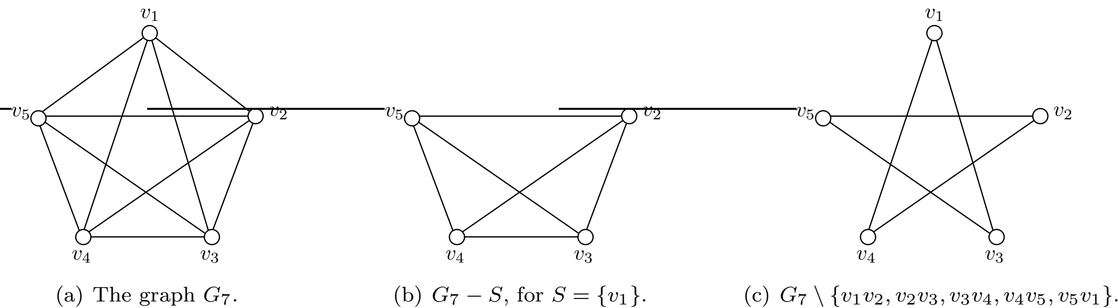
Figure 2.5: *Illustration of a subdigraph, spanning subdigraph and induced subdigraph of digraph.*

$J \subseteq E(G)$ the deletion of the edge set J, denoted by $G - J$, is the *spanning subgraph* of G with edge set $E(G)\backslash J$. If only one edge e or vertex v is removed from a graph, the subgraph may also be denoted by $G - e$ or $G - v$ respectively. Considering the graph G_7 in Figure 2.6(a), with vertex subset $S = \{v_1\}$ and edge subset $J = \{v_1v_2, v_2v_3, v_3v_4, v_4v_5, v_5v_1\}$, the subgraph $G_7 - S$ (or equivalently $G_7 - v_1$) is shown in Figure 2.6(b), while $G_7 - J$ is shown in Figure 2.6(c).

2.1.4 Adjacency matrices and edge weights

Each edge e of a graph G may be assigned an *edge weight*, denoted by $w(e)$. The weight can be seen as some value lost or gained when moving from one vertex to the other joined by the edge e. For example,

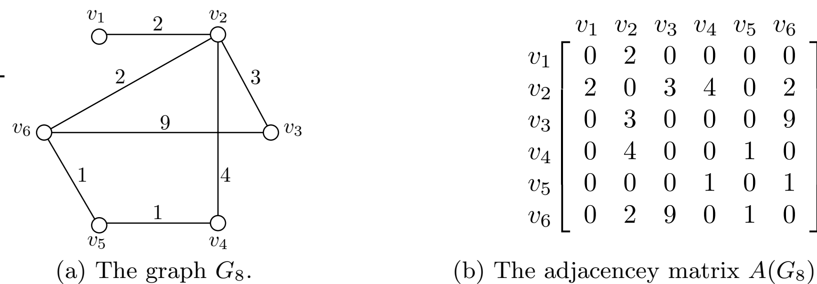
Stellenbosch University http://scholar.sun.ac.za



(a) The graph G_7. (b) $G_7 - S$, for $S = \{v_1\}$. (c) $G_7 \setminus \{v_1v_2, v_2v_3, v_3v_4, v_4v_5, v_5v_1\}$.

Figure 2.6: *Illustration of the deletion of a vertex and an edge subset.*

in a map that is represented by a graph, with cities as vertices and roads as edges, the weight of an edge might represent the distance between the cities joined by e. Figure 2.7(a) shows a graph G_8 with edge weights displayed on the graph. The weights associated with the edge set of a graph G may be represented on a computer in the form of an *adjacency matrix*, $A(G)$, in which the entry in row i and column j corresponds to the weight of edge v_iv_j. If no edge exists, then the entry may either be taken as zero or infinity, depending on how the adjacency matrix is to be used. The adjacency matrix for G_8 is given in Figure 2.7(b). Notice that the adjacency matrix is symmetric for undirected graphs.



	v_1	v_2	v_3	v_4	v_5	v_6
v_1	0	2	0	0	0	0
v_2	2	0	3	4	0	2
v_3	0	3	0	0	0	9
v_4	0	4	0	0	1	0
v_5	0	0	0	1	0	1
v_6	0	2	9	0	1	0

(a) The graph G_8. (b) The adjacencey matrix $A(G_8)$ of G_8.

Figure 2.7: *Illustration of the adjacency matrix of a graph.*

2.1.5 Graph unions and joins

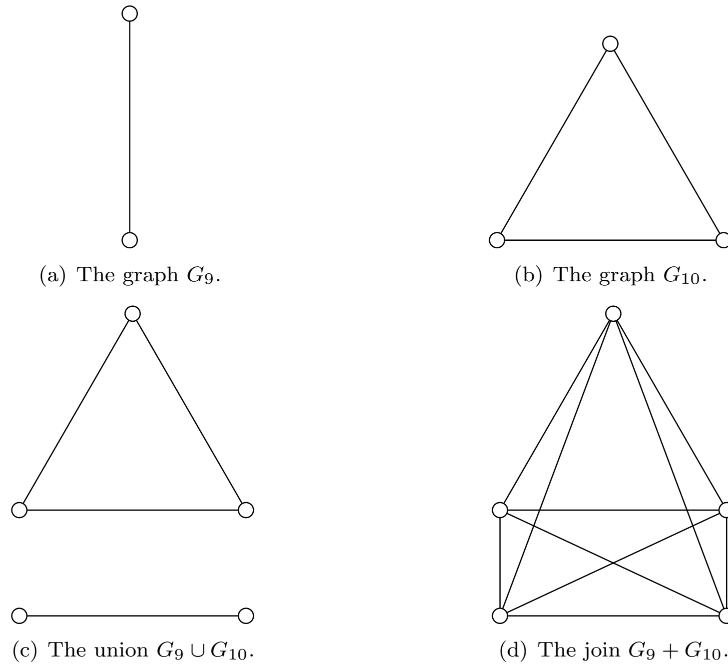
Graphs may be produced from other graphs in several ways. The *union* of two graphs F and H, denoted by $F \cup H$, is the graph G with vertex set $V(G) = V(F) \cup V(H)$ and edge set $E(G) = E(F) \cup E(H)$. The *join* of two graphs F and H is denoted by $F + H$ which is the union of F and H, including all uv edges for which $u \in V(F)$ and $v \in V(H)$. From the symmetry in the definition it follows that $H_1 \cup H_2 \cong H_2 \cup H_1$, $H_1 + H_2 \cong H_2 + H_1$. As an illustration, the union and join operations between the two graphs G_9 and G_{10} depicted in Figures 2.8(a) and (b) are shown in Figures 2.8(c) and (d) respectively.

2.1.6 Special Graphs

A graph solely consisting of a *path* of order p is so called and denoted by P_p. A path that starts at some vertex x and ends at a vertex y is sometimes referred to as an *x-y path*. Similarly, a graph consisting of a single *cycle* of length p is so called and denoted by C_p. Paths and cycles are referred to as *odd* (or *even*) if they have odd (or even) lengths. Graphical representations of an (odd) path P_8 and (even) cycle C_6 are shown in Figure 2.9.

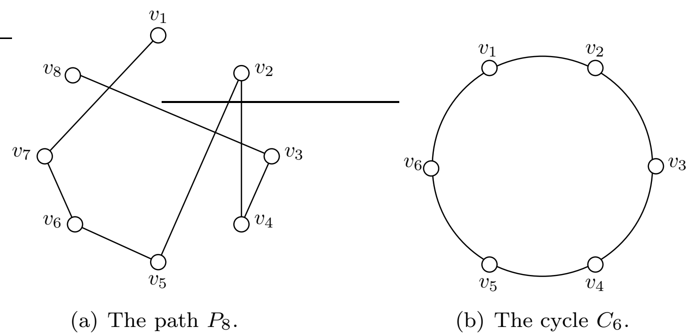
A graph G is called *r-regular* if each vertex of G has degree r. A graph is referred to as *regular* if it is r-regular for some $r \in \mathbb{N}_0$. Any 1-regular subgraph of G is called a *matching* of G. A matching of G with the maximum number of vertices is called a *maximum matching* of G, while the *matching number* $\nu(G)$ denotes the number of edges in a maximum matching of G. A *perfect matching* of G, if it exists, is a matching of G containing all the vertices of G. The 3-regular graph G_{12} in Figure 2.10(a) possesses a perfect matching, shown in Figure 2.10(b).

Stellenbosch University http://scholar.sun.ac.za



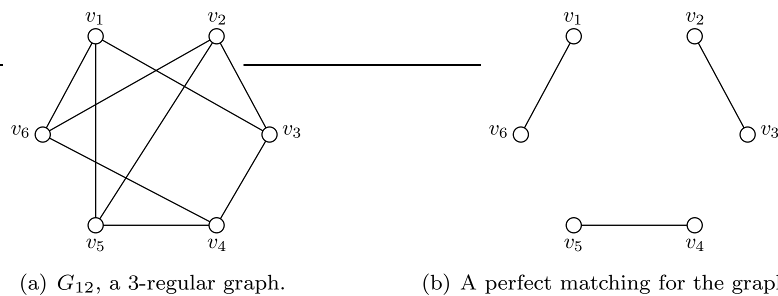
(a) The graph G_9.
(b) The graph G_{10}.
(c) The union $G_9 \cup G_{10}$.
(d) The join $G_9 + G_{10}$.

Figure 2.8: *Illustration of the union and the join of two graphs.*



(a) The path P_8.
(b) The cycle C_6.

Figure 2.9: *Graphical representation of a path and a cycle.*



(a) G_{12}, a 3-regular graph.
(b) A perfect matching for the graph G_{12}.

Figure 2.10: *Illustration of a regular graph and a perfect matching.*

A *null graph* is a graph with no edges. The null graph G_{13} depicted in Figure 2.11 has 10 vertices, but no edges.

A *complete graph* of order p, denoted by K_p, is a graph in which every distinct pair of vertices are adjacent. As an illustration of the concept, the complete graphs K_5 and K_6 are shown graphically in Figure 2.12.

Stellenbosch University http://scholar.sun.ac.za

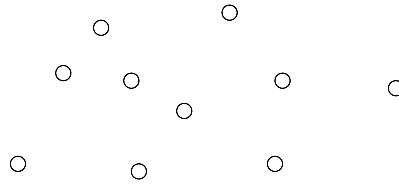
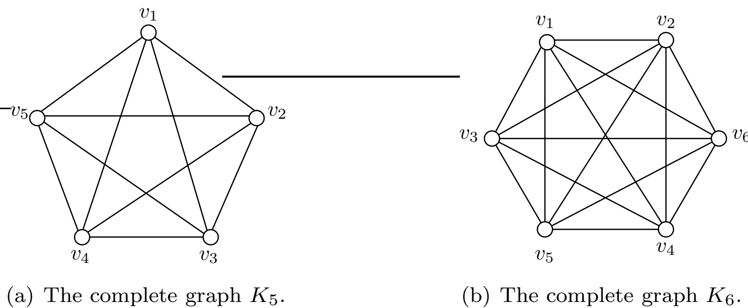


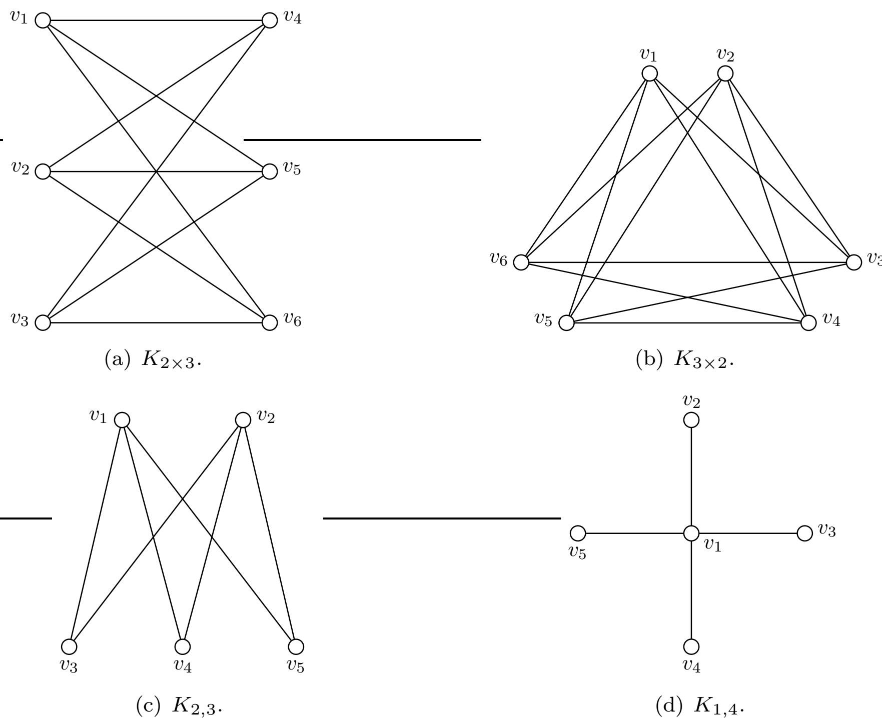
Figure 2.11: *Graphical representation of the null graph G_{13} of order 10.*



(a) The complete graph K_5. (b) The complete graph K_6.

Figure 2.12: *Illustration of the concept of a complete graph.*

A graph G is called *n-partite*, $n \geq 2$, if its vertex set may be partitioned into n subsets, such that no edge of G joins vertices from the same subset. For $n = 2$, G is sometimes called *bipartite*, otherwise it is sometimes called *multipartite*. If a vertex in a partition set V_i of a multipartite graph G is adjacent to every vertex in the other sets $\{V_j : j \neq i\}$ for any vertex in G, then G is called *complete n-partite*. Such a graph G with $|V_i| = p_i$, $i = 1, 2, \ldots, n$, is denoted by $K_{p_1,p_2,\ldots,p_n}$. If $p_1 = p_2 = \ldots = p_n = p$, say, then G is called a *complete, balanced n-partite* graph and denoted by $K_{n\times p}$. Also, the bipartite graph $K_{1,n}$ is a popular graph, called an *n-star*. The vertex adjacent to all other vertices of the star is called the *centre*. Illustrations of multipartite and bipartite graphs are shown in Figure 2.13.



(a) $K_{2\times 3}$. (b) $K_{3\times 2}$.

(c) $K_{2,3}$. (d) $K_{1,4}$.

Figure 2.13: *Graphical representation of multipartite and bipartite graphs.*

The simplest connected graph structure is known as a *tree*, which is an acyclic connected graph. A graph which is acyclic, is called a *forest*, and consists of a number of disconneced trees. A *leaf* of a tree T is an end-vertex of T. Similar to the trivial graph, a *trivial tree* is a graph consisting of only one vertex and

Stellenbosch University http://scholar.sun.ac.za

no edges. An example of a tree of order 10 is shown in Figure 2.14, in which the 5 leaves are indicated as black vertices.

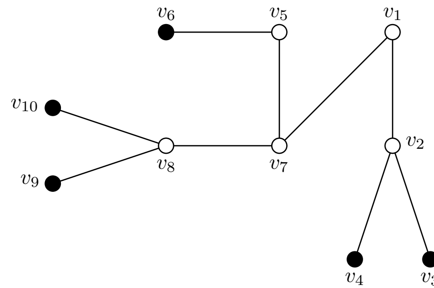
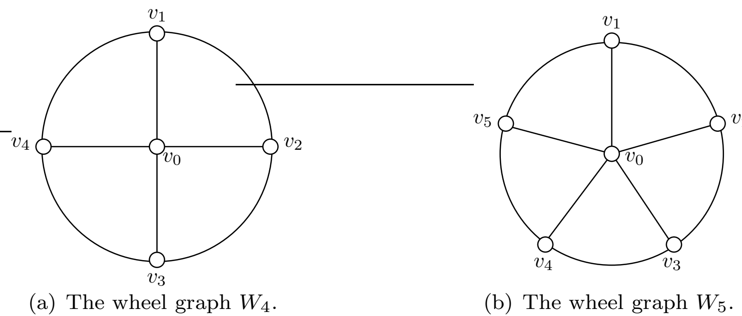


Figure 2.14: *Illustrations of a tree, with leaves indicated as black vertices.*

Consider a cycle of length $n \geq 3$, $C_n: \ v_1v_2\cdots v_n$, and another vertex, v_0 say. A *wheel* W_n of order n is defined as the graph join $C_n + \langle v_0 \rangle$, with the vertex v_0 sometimes referred to as the hub of the wheel. The edges connecting the hub to the rest of the graph are often referred to as spokes. The wheel graphs W_4 and W_5 are shown in Figure 2.15 as examples.



(a) The wheel graph W_4. (b) The wheel graph W_5.

Figure 2.15: *Illustration of the notion of a wheel.*

A *pseudograph* is a graph in which both multiple edges and loops are permitted (a *loop* is an edge that joins a vertex to itself). A graphical representation of such a graph is shown in Figure 2.16.

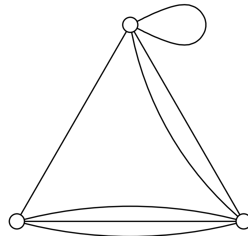


Figure 2.16: *Graphical representation of a pseudograph.*

2.1.7 Connectedness

For vertices u and v of a graph G, u is said to be *connected* to v if G contains a u-v path. The graph G is called a *connected graph* if the vertices u and v are connected for any pair $u, v \in V(G)$. A graph that is not connected is said to be *disconnected*. A subgraph H of G is called a *component* of G if H is a maximally connected subgraph of G. The number of components of a graph G is denoted by $k(G)$. Hence, a graph is connected if $k(G) = 1$. An example of a graph G_{14} for which $k(G_{14}) = 3$ is shown in Figure 2.17.

An edge e is called a *bridge* of a graph G if the graph $G - e$ has more components than G, and a vertex v is called a *cut-vertex* of G if the graph $G - v$ has more components than G. Therefore, an edge e in a connected graph G is a bridge if $G - e$ is disconnected and a vertex v in a connected graph G is a cut-vertex if $G - v$ is disconnected. The graph G_{15} shown in Figure 2.18(a) has the edge v_3v_6 as a bridge,

 Stellenbosch University http://scholar.sun.ac.za

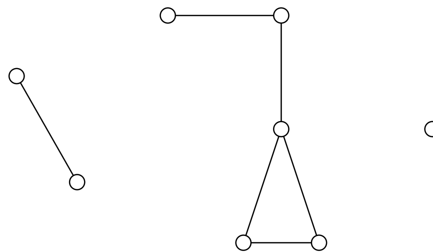
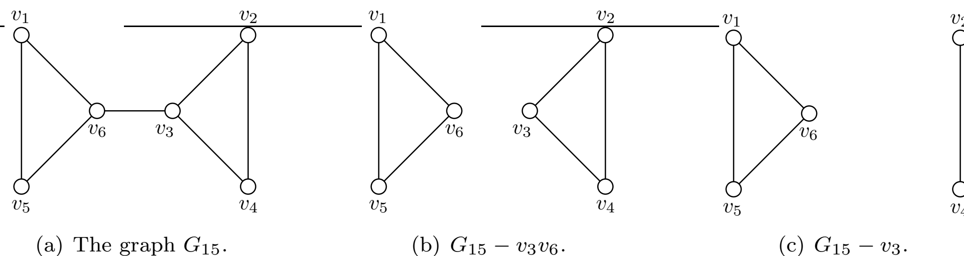


Figure 2.17: *An example of the graph G_{14} for which $k(G_{14}) = 3$.*

while v_3 is a cut-vertex of G_{15}. The graphs $G_{15} - v_3v_6$ and $G_{15} - v_3$ are depicted in Figures 2.18(b) and (c) respectively. The following theorem provides a characterisation for when an edge is a bridge. A proof of this theorem may be found in Chartrand & Oellerman [11, p. 22].

Theorem 2.2 *An edge e of a connected graph G is a bridge of G if and only if e does not lie on a cycle of G.* ■



(a) The graph G_{15}. (b) $G_{15} - v_3v_6$. (c) $G_{15} - v_3$.

Figure 2.18: *Illustration of a bridge and cut-vertex in the connected graph G_{15} in (a). (b) The edge v_3v_6 is a bridge, since $G_{15} - v_3v_6$ is disconnected. (c) The vertex v_3 is a cut-vertex, since $G_{15} - v_3$ is disconnected.*

2.1.8 Vertex Splitting

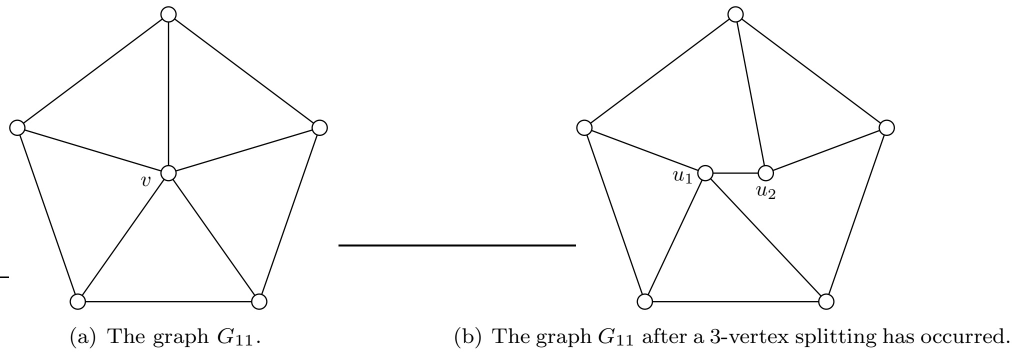
Let G be a connected graph and let k be the minimum cardinality of a set U of vertices for which $G \setminus U$ is disconnected or is the trivial graph. Furthermore, suppose there exists a vertex $v \in V(G)$ such that $\deg v \geq 2k - 2$. Let $N(v) = N_1 \cup N_2$, with $|N_1| \geq k - 1$ and $|N_2| \geq k - 1$ and $N_1 \cap N_2 = \emptyset$. Construct the graph G' from G by replacing vertex v with two adjacent vertices u_1 and u_2, and joining vertex u_i to every vertex in N_i, $i = 1, 2$. Such a construction of the graph G' from the graph G is referred to as a *k-vertex splitting*. Consider the graph G_{11} depicted in Figure 2.19(a), where $k = 3$, as at least 3 vertices must be removed in order to disconnect the graph (for instance, $G_{11} \setminus \{v_1, v_6, v_3\}$ is disconnected). A 3-vertex splitting may be applied to the graph G_{11} at vertex v_6, producing the graph depicted in Figure 2.19(b).

Figure 2.19(b) depicts the graph G_{11}, shown in Figure 2.19(a), after a 3-vertex splitting has occured at vertex v.

2.1.9 Independence number

A vertex subset $S \subseteq V(G)$ of G is called *independent* if no two vertices in S are adjacent in G. An independent set S of vertices in a graph G is called a *maximal independent* set if S is not a proper subset of any other independent set of G. The maximum cardinality of such maximal independent sets S is called the *independence number* of G and is denoted by $\beta(G)$. For the bipartite graph $K_{2,3}$, shown in Figure 2.20(a), both partite sets $\{v_1, v_2\}$ and $\{v_3, v_4, v_5\}$ are maximal independent sets of $K_{2,3}$. Since the independent set $\{v_3, v_4, v_5\}$, indicated as dark vertices in Figure 2.20(a), is the largest maximal independent set, it follows that $\beta(K_{2,3}) = 3$. Opposite to the notion of independence is the notion

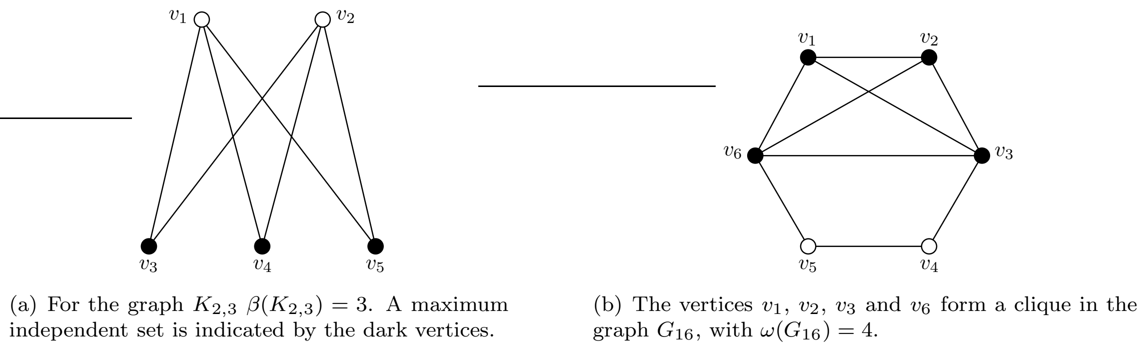
Stellenbosch University http://scholar.sun.ac.za



(a) The graph G_{11}. (b) The graph G_{11} after a 3-vertex splitting has occurred.

Figure 2.19: *Illustration of a 3-vertex splitting.*

of a *clique*, which is a complete subgraph of a graph G that is not an induced subgraph of any other complete subgraph of G. A clique of a graph G is thus a maximal complete subgraph of G. The order of a maximum clique is defined as the *clique number* $\omega(G)$ of the graph G. A graphical representation of a clique consisting of the vertices v_1, v_2, v_3 and v_6 in a graph G_{16} is shown in Figure 2.20 (hence $\omega(G) = 4$).



(a) For the graph $K_{2,3}$ $\beta(K_{2,3}) = 3$. A maximum independent set is indicated by the dark vertices. (b) The vertices v_1, v_2, v_3 and v_6 form a clique in the graph G_{16}, with $\omega(G_{16}) = 4$.

Figure 2.20: *Graphical representation of the independence number of a graph and the notion of a clique.*

2.2 Basic Concepts in Complexity Theory

An overview of basic concepts in complexity theory is now given.

2.2.1 Algorithmic complexity

An *algorithm* is a specific set of instructions for carrying out a procedure or solving a problem, usually with the requirement that the procedure terminate at some point. All instructions can be broken down into a sequence of operations that cannot be broken down any further. These operations are known as *basic operations*. *Algorithmic complexity* is measured by a *time complexity* variable and a *space complexity* variable, usually expressed in terms of the input size n of the algorithm in question. These variables measure respectively the number of basic operations performed and the memory required by the algorithm. It is not always easy to measure the time complexity and space complexity of an algorithm. By knowing these quantities, different algorithms for performing the same task can be compared to determine which algorithm is more efficient. A well-known measurment used for obtaining a bound on the running time of an algorithm is by observing the *order of magnitude* of an algorithm. The order of magnitude, denoted by means of the symbol O, of the algorithmic complexity is defined as follows: Let f and g be two real-valued functions. Then $f(n) = O(g(n))$ if there exist a $c \in \mathbb{R}^+$ and an $n_0 \in \mathbb{N}$ such that $0 \leq f(n) \leq cg(n)$ for all $n \geq n_0$. The function g is said to be an asymptotic upper bound for f. An algorithm for which the order of magnitude of its worst-case time complexity is of the form $O(n^k)$, for some $k \in \mathbb{R}^+$ in terms of its input size n, is called a *polynomial time* algorithm. If a problem cannot (with current knowledge) be solved by a polynomial time algorithm, it is referred to as an *intractable* or

Stellenbosch University http://scholar.sun.ac.za

hard problem, otherwise it is called a *tractable* problem. While the term complexity usually refers to the time complexity of an algorithm, the importance of the space complexity should not be disregarded in practical algorithm implementations.

2.2.2 The classes P, NP and co-NP

Decision theory is the branch of complexity theory where problems to be solved are interpreted as binary questions, that may be answered "true" or "false". Since any computational problem may be reduced to a *decision problem*, it is possible, without loss of generality, to consider decision theory only in the theoretical analysis of complexity issues. The class *P* is defined as the set of decision problems that can be solved by way of a polynomial time algorithm. The class *NP* constitutes the set of decision problems that may be answered "true" by a polynomial time algorithm, given additional information (known as a *certificate* with respect to the specific instance of the decision problem). Similarly, the class *co-NP* is the set of all decision problems that may be answered "false" by a polynomial time algorithm, given additional information (also called a certificate). Note that although a certificate with respect to a decision problem instance may exist, finding this certificate may be difficult. The various classes described above into which a problem may be classified, are shown graphically in Figure 2.21.

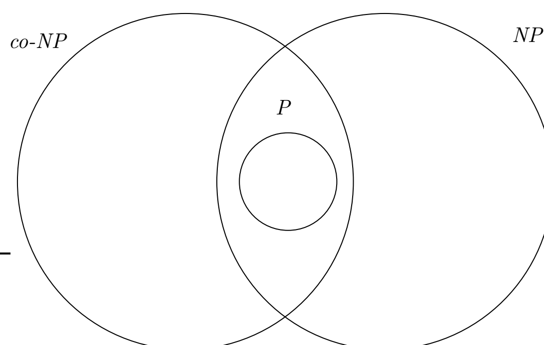


Figure 2.21: *The classes P, NP and co-NP.*

As an example, consider the following decision problem.

$D_{clique}(G, k)$
INSTANCE: A graph G and $k \in \mathbb{N}$.
QUESTION: Does G have clique number $\omega(G) \geq k$?

The following proposition shows that the decision problem $D_{clique}(G, k)$ belongs to the class *NP*, by using a clique of G of order k, say $\langle v_1, v_2, \ldots, v_k \rangle_G$, as certificate.

Proposition 2.1 $D_{clique}(G, k) \in NP$

Proof: Algorithm 1 verifies whether the induced graph $s = \langle v_1, v_2, \ldots, v_k \rangle_G$ is a clique in G, a graph of order p, say. Let q_s be a counter for summing the degrees for each vertex in $\langle s \rangle_G$. For $\langle s \rangle_G$ to be a clique, it must contain $\frac{1}{2}k(k-1)$ edges.

The for loop on lines 2 to 4 sums the degree of each vertex in $\langle s \rangle_G$. It follows from the Fundamental Theorem of Graph Theory (Theorem 2.1) that if $q_s = k(k-1)$, then $\langle s \rangle_G$ is a clique (the algorithm actually counts each edge twice). Note that calculating the degree of a vertex in the proposed clique takes $O(p)$ time, as a row (or column) in the adjacency matrix of the graph G needs to be traversed. As there are p elements in a row (or column), traversal has a worst-case running time of $O(p)$. Hence it follows that the for loop on lines 2 to 4, and consequently also the whole of Algorithm 1 has a worst-case running time of $O(kp)$. It is therefore concluded that the algorithm will produce an output in polynomial time. As the decision $D_{\text{clique}}(G, k)$ may be answered "true" with the aid of a certificate was required, it follows that $D_{clique}(G, k) \in NP$. ∎

Stellenbosch University http://scholar.sun.ac.za

Algorithm 1 $D_{clique}(G, k)$

Input: A graph G and vertex set $S = \{v_1, v_2, \ldots, v_k\}$.
Output: TRUE, if $\langle S \rangle_G$ is a clique, otherwise FALSE.

1: $q_s \leftarrow 0$
2: **for** $i = 1$ to k **do**
3: $\quad q_s \leftarrow q_s + \deg v_i$
4: **end for**
5: **if** $q_s = k(k-1)$ **then**
6: $\quad$ Return *true*
7: **else**
8: $\quad$ Return *false*
9: **end if**

The question of whether a graph G does not have a clique number $\omega(G) \geq k$ poses to be a harder probem. If a certificate exists such that this problem may be answered by a polynomial time algorithm, it follows that this problem is of the class *co-NP*. The questions of whether $P = NP$, $NP = co\text{-}NP$ or $P = NP \cap co\text{-}NP$ remain unanswered to this day. The following result (see Cormen *et al.* [12, pp.981–982]) relates the subsets P, NP and *co-NP*.

Theorem 2.3 (Cormen *et al.* [12]) *$P \subseteq NP$ and $P \subseteq co\text{-}NP$.* ■

2.2.3 Polynomial time reducibility, NP-hardness and NP-completeness

Let D_1 and D_2 be two decision problems. The problem D_1 is *polynomial time reducible* to D_2, denoted by $D_1 \preceq D_2$, if there exists an algorithm A_1 that can solve all instances of D_1 which contains as a subroutine an algorithm A_2 that can solve all instances of D_2, such that A_1 is a polynomial time algorithm if A_2 is a polynomial time algorithm. Informally stated, D_2 is therefore computationally at least as hard to solve as D_1.

A decision problem D is *NP-hard* if $D_1 \preceq D$ for all $D_1 \in NP$. Furthermore, a decision problem D is said to be *NP-complete* if $D \in NP$ and D is *NP*-hard. An *NP*-hard problem may differ from an *NP*-complete problem in the sense that it may belong to a different class of computation problems altogether. For instance, an *NP*-hard problem may belong to a class of problems that can only be solved in exponential time. *NP*-complete problems may be seen as computationally the most difficult problems to solve in *NP*, since they are computationally at least as hard to solve as any other problem in *NP*. Although it is not currently known whether the classes P and NP differ, it has been proven that, if a decision problem D exists for which $D \in NP$-complete and $D \in P$, then $P = NP$ (see Sipser [57, pp.247–253]). The relation between the classes P, NP and *NP*-complete are shown in Figure 2.22.

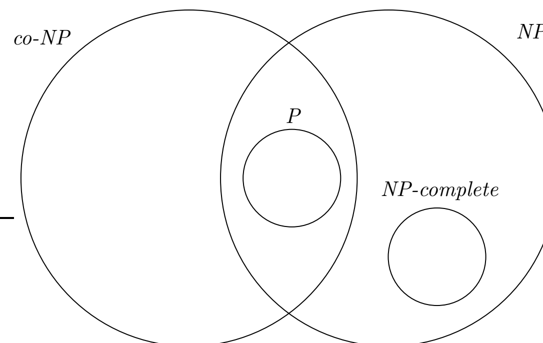


Figure 2.22: *The classes P, NP, co-NP and NP-complete.*

A set of *NP*-complete problems can be constructed if there exists a method by which a problem of an unknown class can be compared to a specific *NP*-complete problem, thereby determining whether the

Stellenbosch University http://scholar.sun.ac.za

problem of unknown class is also *NP*-complete. The following result (see Sipser [57, p. 253] for a proof) illustrates such a method of comparison.

Theorem 2.4 (Sipser [57]) *Let D_1 and D_2 be two decision problems. If $D_1 \in \mathrm{NP}$, D_2 is NP-complete and $D_2 \preceq D_1$, then D_1 is NP-complete.* ■

The decision problem D_2 thus acts as a starting point for building up a set of *NP*-complete problems. An example of such a decision problem that may be used as a starting point is the decision problem $D_{clique}(G, k)$ according to the following theorem (see Cormen *et al.* [12, pp.1003–1005] for the proof), which states that $D_{clique}(G, k)$ is *NP*-complete.

Theorem 2.5 (Cormen *et al.* [12]) *$D_{clique}(G, k)$ is NP-complete.* ■

2.2.4 Computation problems

A *computation problem* differs from a decision problem in that its solution may be a real value, not necessarily a mere binary value. Decision problems may therefore be seen as special cases of computation problems. Some computation problems may be solved efficiently in terms of algorithmic procedures by implementing the solution to related decision problems. This again provides a means for classifying computation problems into various classes such as *N*, *NP* and *NP*-complete.

Let $C_{clique}(G)$ denote the computation problem of finding the clique number $\omega(G)$ of a given graph G. The clique number of a graph may be calculated by means of a so-called *interval halving scheme* in terms of the *NP*-complete decision problem $D_{clique}(G, k)$. This decision problem is called the underlying decision problem of the computation problem $C_{clique}(G)$. Consider Algorithm 2 for solving $C_{clique}(G)$ with underlying decision problem $D_{clique}(G, k)$.

Algorithm 2 $C_{clique}(G)$

Input: A graph G of order p.
Output: The clique number $\omega(G)$.

1: **if** $D_{clique}(G, n) = true$ **then**
2:   print p, stop.
3: **else**
4:   $\ell \leftarrow 1, r \leftarrow p$
5:   $m \leftarrow \left\lfloor \frac{\ell+r}{2} \right\rfloor$
6:   **if** $D_{clique}(G, m) = true$ **then**
7:     $\ell \leftarrow m$
8:   **else**
9:     $r \leftarrow m$
10:   **end if**
11:   **if** $r - \ell = 1$ **then**
12:     print ℓ, stop.
13:   **else**
14:     go to Step 5
15:   **end if**
16: **end if**

The rationale behind Algorithm 2 is that it maintains, at each iteration, an interval on the real line with left and right endpoints the integers ℓ and r respectively, such that $D_{clique}(G, \ell) = true$ and $D_{clique}(G, r) = false$, implying that $\ell \leq \omega(G) < r$. During each interval of the algorithm, this interval is halved (hence the term interval halving scheme) by determining an integer m, such that m is the largest integer not exceeding the midpoint $(\ell+r)/2$ of the interval $[\ell, r)$. If $r - \ell = 1$, then the clique number of G has been found, since $\ell \leq \omega(G) < \ell + 1$ implies that $\omega(G) = \ell$, at which point the algorithm terminates.

Stellenbosch University http://scholar.sun.ac.za

If this condition is not satisfied, the interval is halved. The algorithmic search then continues in the right half interval if $D_{clique}(G, m) = true$ (there exists a clique of at least m vertices), otherwise the algorithmic search will continue in the left half interval (there is no clique in the graph G consisting of m or more vertices). This process ensures that $D_{clique}(G, \ell) = true$ and $D_{clique}(G, r) = false$ for each new half interval.

Let ℓ_i and r_i denote the values of ℓ and r respectively during iteration i of Algorithm 2. It follows by the interval halving property that the interval width during iteration i satisfies the inequality $r_i - \ell_i \leq (r_{i-1} - \ell_{i-1} + 1)/2$. By iteratively applying this inequality j times, it follows that

$$r_i - \ell_i \leq \frac{r_{i-j} - \ell_{i-j} + 2^j - 1}{2^j}.$$

Hence, for $j = i$, the bound

$$r_i - \ell_i \leq \frac{r_0 - \ell_0 + 2^j - 1}{2^j} = \frac{n - 2 + 2^j}{2^j}$$

is obtained. Now the process of interval halving will continue as long as

$$2 \leq r_i - \ell_i \leq \frac{n - 2 + 2^i}{2^i},$$

that is, as long as $i \leq \log_2(n-2)$. This implies that Algorithm 2 will apply the interval halving method a worst-case number of $O(\log n)$ times. If the notion of polynomial time reducibility is generalised to encompass not only decision problems, but also computation problems, it follows that $C_{clique}(G) \preceq D_{clique}(G, k)$. As $D_{clique}(G) \in NP$-complete, it further follows that $C_{clique}(G) \in NP$-complete.

The above example serves to demonstrate that the classification scheme of categorizing problems into the various classes such as N and NP and the notions of polynomial time reducibility and NP-completeness are not only applicable to decision problems, but may also be generalised to accommodate computation problems.

2.3 Chapter Summary

In this chapter, the basic concepts of graph theory and complexity theory, relevant to this thesis, were introduced for the benefit of the reader. The appropriate graph theoretic concepts were discussed in §2.1-2.1.7. The last section, §2.2, familiarized the reader with the basic concepts in complexity theory. This chapter does not serve as a comprehensive study in the fields of graph theory and complexity theory. Extensive research has already been done in both of these fields for which vast amounts of literature exist. Only those parts of these fields that pertain to the theory described in this thesis have been discussed.

Stellenbosch University http://scholar.sun.ac.za

Chapter 3

Literature Survey

This chapter contains a survey of work published in the field of graph connectivity. Some basic definitions are given first, after which the cornerstone theorems of Menger and Whitney are reviewed. This is followed by a survey of other research done on graph connectivity. Some of the results mentioned have a broader scope than that of this thesis, but are included for the sake of completeness.

3.1 Connectivity and edge-connectivity

Let G be a connected graph. Recall that if G contains an edge e such that $G - e$ is disconnected, then e is called a *bridge* of G. Also, if G has a vertex v such that $G - v$ is disconnected, then v is called a *cut-vertex* of G. These definitions may be expanded to cases where more than one edge or vertex are removed. A subset S of the edges of a connected graph G is said to be an *edge cut-set* of G if $G \setminus S$ is disconnected. Similarly, if U is a subset of the vertices of G such that $G \setminus U$ is disconnected, then U is called a *vertex cut-set* of G.

Definition 3.1 *The* edge-connectivity number $\lambda(G)$ *of a graph G is the minimum cardinality of a set S of edges for which $G \setminus S$ is disconnected or is the trivial graph.* ■

Definition 3.2 *The* (vertex) connectivity number $\kappa(G)$ *of a graph G is the minimum cardinality of a set U of vertices for which $G \setminus U$ is disconnected or is the trivial graph.* ■

The only connected graph that cannot be disconnected by the removal of edges is the trivial graph K_1 Hence $\lambda(K_1) = 0$. Furthermore, no complete graph K_p can be disconnected by the removal of any subset of vertices, but removing $p - 1$ of the vertices produces the trivial graph. Hence $\kappa(K_p) = p - 1$. Apart from these two cases, every non-trivial connected graph may be disconnected by the removal of edges and every non-complete graph may be disconnected by the removal of vertices. Clearly, for a disconnected graph G, $\lambda(G) = \kappa(G) = 0$. Whitney [64] related the quantities $\lambda(G)$, $\kappa(G)$ and $\delta(G)$ for any graph G.

Theorem 3.1 (Whitney [64]) $\kappa(G) \leq \lambda(G) \leq \delta(G)$ *for any graph G.* ■

As an example, consider the graph G_{17} in Figure 3.1. In this case $\delta(G_{17}) = 3$. The edge-connectivity number of G_{17} is $\lambda(G_{17}) = 3$, as the removal of three edges (say v_1v_2, v_1v_3 and v_1v_4) disconnects the graph, but the removal of no set of two egdes disconnects G_{17}. The connectivity number of G_{17} is $\kappa(G_{17}) = 1$, as the removal of the vertex v_4 disconnects the graph.

3.2 Menger's Theorem

Many of the results dealing with the two parameters $\kappa(G)$ and $\lambda(G)$ of a graph G hinge on the 1927 theorem of Menger [46], which was proved in the course of research he conducted on curve theory in

Stellenbosch University http://scholar.sun.ac.za

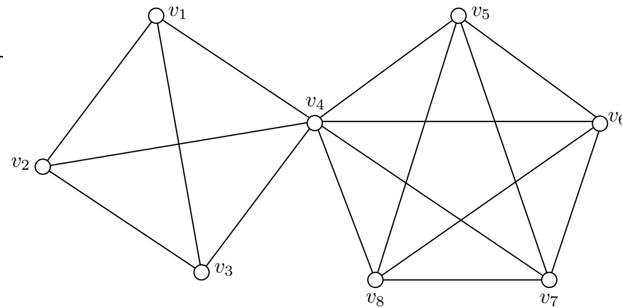


Figure 3.1: *The graph G_{17}.*

point set topology. However, the original proof contained a nontrivial gap. The first flawless proof of Menger's theorem, due to Noebling, appeared in a publication of Menger's (Menger [47]).

Before Menger's theorem may be stated, a few more definitions are required. Two u-v paths $P^{(1)}$ and $P^{(2)}$ are *edge disjoint* if $P^{(1)}$ and $P^{(2)}$ have no edges in common. Two u-v paths $P^{(3)}$ and $P^{(4)}$ are said to be *internally disjoint* if they have only the start and end vertices in common, thus $V(P^{(3)}) \cap V(P^{(4)}) = \{u, v\}$. It is clear that if two paths are internally disjoint, then they are also edge disjoint. As an example, consider the paths $P^{(1)} = v_1, v_2, v_5, v_8, v_6, v_4, v_{10}$, $P^{(2)} = v_1, v_7, v_5, v_3, v_6, v_9, v_{10}$, $P^{(3)} = v_1, v_2, v_3, v_4, v_{10}$ and $P^{(4)} = v_1, v_7, v_8, v_9, v_{10}$ in the graph G_{18} shown in Figure 3.2. The paths $P^{(1)}$ and $P^{(2)}$ are edge-disjoint and the paths $P^{(3)}$ and $P^{(4)}$ are internally disjoint.

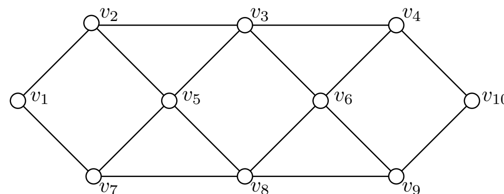


Figure 3.2: *The graph G_{18}.*

A set U of vertices in G not containing two specified vertices u and v is said to be a *u-v vertex-separator* if u and v belong to distinct components of $G \setminus U$ (U *separates* u and v). For nonadjacent vertices u and v let $\kappa'(u, v)$ denote the minimum cardinality of a u-v vertex-separator. Similarly, a set S of edges in G is called a *u-v edge-separator* if these vertices lie in distinct components of $G \setminus S$ (S *separates* u and v). Let $\lambda'(u, v)$ be the minimum cardinality of a u-v edge-separator. Furthermore, for a pair of distinct vertices u and v in a graph G, let $\kappa(u, v)$ [$\lambda(u, v)$, respectively] denote the maximum number of internally disjoint [edge-disjoint, respectively] u-v paths in G. Menger's theorem may now be stated.

Theorem 3.2 (Menger [46]) *For distinct nonadjacent vertices u and v of a graph G, $\kappa(u, v) = \kappa'(u, v)$.* ■

The edge analogue of Menger's theorem (originally proven by Elias *et al.* [16] and Ford & Fulkerson [20]) follows.

Theorem 3.3 *For distinct nonadjacent vertices u and v of a graph G, $\lambda(u, v) = \lambda'(u, v)$.* ■

Again consider the graph G_{17} in Figure 3.1. For the vertices v_1 and v_7, $\kappa'(v_1, v_7) = \kappa(v_1, v_7) = 1$ (the vertex v_4 may be removed to disconnect the graph). For the same two vertices, $\lambda'(v_1, v_7) = \lambda(v_1, v_7) = 3$ as the cardinality of the edge set $E = \{v_1v_2,\ v_1v_3,\ v_1v_4\}$ and the cardinality of any set of edge-disjoint v_1-v_7 paths both equal 3.

Both Menger's theorem and its edge analogue may be deduced from the max-flow min-cut theorem of Ford & Fulkerson [20]. A minimum s-t-cut is a set X of edges having a minimum combined weight such that no flow can be constructed from the vertex s to the vertex t in the graph $G \setminus X$. The max-flow min-cut theorem of Ford & Fulkerson [20] (see Bollobás [5, pp.50–53] for a proof) is now stated.

Stellenbosch University http://scholar.sun.ac.za

Theorem 3.4 (Bollobás [5]) *The maximal flow value from a vertex s to a vertex t is equal to a minimum s-t-cut.* ∎

Different methods of proof of Menger's theorem may be found in Bollobás [4, pp.8–9], Diestel [15, pp.50–54], Hajós [28], König [33] and West [63, pp.149–150].

3.3 Computing $\kappa(G)$ and $\lambda(G)$

Most of the algorithms that compute $\kappa(G)$ and $\lambda(G)$ make use of the max-flow min-cut theorem of Ford & Fulkerson [20]. Both of these parameters may be computed in polynomial time, where the polynomial is a function of the order p and size q of a graph. The complexity of computing the parameters depends on the number of maximum flow problems that have to be solved.

There exist $\binom{p}{2} - q$ pairs of non-adjacent vertices in a graph G of order p and size q. Hence this is the worst-case number of maximum flow problems that have to be solved in order to compute the vertex connectivity number of a graph G. Esfahanian & Hakimi [17] improved this bound by developing an algorithm that computes $\kappa(G)$ by solving $p-\delta(G)-1+\frac{1}{2}\kappa(G)(2\delta(G)-\kappa(G)-3)$ maximum flow problems. This is the best bound known. Picard & Queyranne [54] developed a method by which all minimum cut-sets of a graph G may be represented. The fastest algorithm based on these principles is due to Gabow [22], with a worst-case running time of $O\left(pq\log\left(p^2/q\right)\right)$. A method has been developed that calculates $\lambda(G)$ and returns a minimum edge-cut-set with very high probability. This method is discussed in the next section.

The edge-connectivity number of a graph may be computed by solving $p(p-1)$ maximum flow problems (see Esfahanian & Hakimi [17]). However, algorithms have been developed that are able to compute $\lambda(G)$ without the requirement of solving maximum flow problems. The best procedure known in this class is an algorithm developed by Nagamochi & Ibaraki [50], which finds $\lambda(G)$ and a minimum edge cut-set in $O(pq)$ time. Their algorithm is based on a decomposition of the graph G into spanning forests.

An important question to ask is what the maximum connectivity number (edge-connectivity number) may be for a graph constructed on p vertices by means of q edges. Harary [29] proved the following useful result.

Theorem 3.5 (Harary [29]) *Among all graphs with p vertices and q edges, the maximum connectivity number (edge-connectivity number) is 0 when $q < p-1$ and $\lfloor \frac{2q}{p} \rfloor$ if $q \geq p-1$.* ∎

3.4 Computing $\lambda(G)$ with high probability

Mitzenmacher & Upfal [49, pp.12–14] developed a probabilistic method for calculating the edge-connectivity number as well as a minimum edge-cut-set of a graph with very high probability. Multiple edges between vertices are allowed, but no loops are allowed. An *edge contraction* is defined as the process of merging two adjacent vertices u and v. All edges that join u and v are eliminated, while all other edges are retained. The algorithm functions by selecting an edge $uv \in E(G)$ uniformly at random and then contracts that edge. This operation is repeated $p-2$ times. The end result is a graph with two vertices joined by a certain number of edges. The following result was proved.

Theorem 3.6 (Mitzenmacher & Upfal [49]) *The algorithm described above outputs a minimum edge-cut-set with probability at least $\frac{2}{p}(p-1)$.* ∎

At first sight the above theorem does not seem a very good result. However, if the algorithm is repeated $p(p-1)\ln p$ times, then the probability of not finding a minimum edge-cut-set is less than $\frac{1}{p^2}$. Thus, if this algorithm is applied to a graph of order 10, the probability of finding a minimum edge-cut-set is greater than 99%, rendering this simplistic algorithm very powerful.

Stellenbosch University http://scholar.sun.ac.za

Unfortunately, it does not seem that this concept can be adapted to calculate the vertex-connectivity number of a graph. The reason for this is that the removal of an edge-cut-set always splits a connected graph into two components, whereas the removal of a cut-set disconnects the graph into two or more components. This is, in fact, why the algorithm terminates after $p-2$ iterations when calculating the edge-connectivity number. It is not clear how many iterations an adapted version of this algorithm should perform before it is terminated in order to calculate the connectivity number.

3.5 k-Connected and k-Edge-Connected Graphs

It is often more valuable to know that a graph cannot be disconnected or reduced to the trivial graph by the removal of $k-1$ edges or vertices, rather than to know its actual edge-connectivity number or connectivity number.

Definition 3.3 *A connected graph G is said to be* k-connected ($k \geq 1$) *if the removal of fewer than k vertices always produces a nontrivial connected graph.* ■

Definition 3.4 *A connected graph G is said to be* k-edge-connected ($k \geq 1$) *if the removal of fewer than k edges always produces a nontrivial connected graph.* ■

Consider the cycle C_4 depicted in Figure 3.3. As $\kappa(C_4) = 2$, C_4 is 1-connected and 2-connected, but not 3-connected.

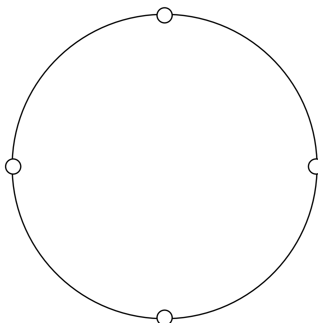


Figure 3.3: *The cycle C_4.*

The connectivity number of a graph is a sharp upper bound on the value of $k \in \mathbb{N}$ for which the graph is k-connected, as stated in the following lemma.

Lemma 3.1 *Let G be a graph for which $\kappa(G) = k$. Then G is ℓ-connected, $\ell \in \{1, \ldots, k\}$.*
Proof Suppose that G is an ℓ-connected graph for which $\kappa(G) = k$. As there exists a set of k vertices whose removal disconnects the graph G, it follows that k is an upper bound for ℓ. The removal of any number of vertices less than k will not disconnect the graph. Hence the result is proven. ■

Similarly the edge-connectivity number is a sharp upper bound for the k-edge-connectivity number of a graph.

Some characterisations of k-connectivity and k-edge-connectivity are now discussed. The first characterisation of k-connectivity and its edge-analogue was given by Whitney [64].

Theorem 3.7 (Whitney [64]) *A graph G is k-connected [k-edge-connected, respectively] if and only if for each pair of vertices $\{u, v\}$ there are at least k internally disjoint [k edge-disjoint, respectively] u-v paths in G.* ■

Whitney proved this result without making use of Menger's theorem, although it may be proved more elegantly by making use of it. Dirac (see Bollobás [4, p.10]) proved the following useful result.

Stellenbosch University http://scholar.sun.ac.za

Theorem 3.8 (Dirac) *A graph G of order p is k-connected if and only if $p \geq k+1$ and for any k-set $U \subset V(G)$ and vertex $x \in V(G) \setminus U$, there is an x-U fan in G.* ■

Another characterisation of k-connected graphs was conjectured independently by Frank [21] and Maurer (see [44, p.73]) and proved independently by Györi [27] and Lovász [36].

Theorem 3.9 *Let G be a graph of order p such that $p \geq k+1$. Then G is k-connected if and only if for any distinct vertices $v_1, v_2, \ldots, v_k$ of G and for any partition of p into positive integers $m_1, m_2, \ldots, m_k$ there exists a partition $V_1, V_2, \ldots, V_k$ of $V(G)$ such that for each $1 \leq i \leq k$, $v_i \in V_i$, $|V_i| = m_i$ and $\langle V_i \rangle_G$ is connected.* ■

The following example illustrates how Theorem 3.9 may be applied.

Example 3.1

Consider the 3-connected hypercube Q_3 shown in Figure 3.4.

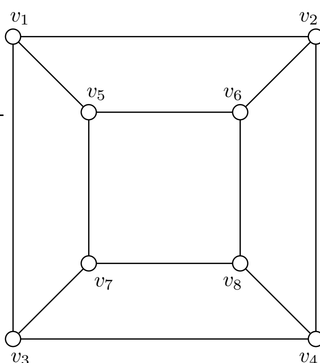


Figure 3.4: *The 3-connected hypercube Q_3.*

Now $m_1 = 4$, $m_2 = 2$ and $m_3 = 2$ is a partition of the 8 vertices. Furthermore, $V_1 = \{v_1, v_2, v_5, v_7\}$, $V_2 = \{v_6, v_8\}$ and $V_3 = \{v_3, v_4\}$ is a partition of the vertex set of Q_3 with the desired properties. ■

Some well-known results exist that may be used to test whether a graph is k-connected. The advantage of these results is that they are easily applied and tested in linear time. However, a drawback is that they do not cover all k-connected graphs — some graphs that are k-connected do not necessarily satsify the conditions required by the results. One such a result (see Chartrand & Harary [8, pp.61–63]) gives a sufficient condition for a graph to be k-connected.

Theorem 3.10 *Let G be a graph of order p, and let k be an integer satisfying $1 \leq k \leq p-1$. If $\delta(G) \geq (p+k-2)/2$, then G is k-connected.* ■

An advantage of this theorem is that it is not required to know beforehand whether the graph being tested is connected or not. If the requirements of the theorem hold, then connectedness is guaranteed. However, a disadvantage of this theorem is that it does not provide tight bounds on $\delta(G)$, as may be seen in Table 3.1. The bound on $\delta(G)$ in Theorem 3.10 is typically larger than is required for k-connectivity for high p and k values. For instance, for a graph to be 2-connected, the minimum degree of a connected graph should be 2 or greater. If $p = 20$, then $(p+k-2)/2 = 10$, which is much larger than 2. Therefore the approach of not being required beforehand to test whether the graph is connected does not seem to be a real advantage, due to the poor bounds on $\delta(G)$. To determine whether or not a graph is connected may be calculated in $O(p^3)$ time using Floyd's shortest paths algorithm, which seems a better option, typically yielding better bounds on $\delta(G)$ so as to guarantee k-connectivity.

A stronger result that may be used to test whether a graph is k-connected is due to Bondy [7].

Stellenbosch University http://scholar.sun.ac.za

p	k	$(p+k-2)/2$
1	1	0.0
2	1	0.5
3	1	1.0
4	1	1.5
5	1	2.0
6	1	2.5
7	1	3.0
8	1	3.5
9	1	4.0
10	1	4.5
11	1	5.0
12	1	5.5
13	1	6.0
14	1	6.5
15	1	7.0
16	1	7.5
17	1	8.0
18	1	8.5
19	1	9.0
20	1	9.5

p	k	$(p+k-2)/2$
1	2	0.5
2	2	1.0
3	2	1.5
4	2	2.0
5	2	2.5
6	2	3.0
7	2	3.5
8	2	4.0
9	2	4.5
10	2	5.0
11	2	5.5
12	2	6.0
13	2	6.5
14	2	7.0
15	2	7.5
16	2	8.0
17	2	8.5
18	2	9.0
19	2	9.5
20	2	10.0

p	k	$(p+k-2)/2$
1	3	1.0
2	3	1.5
3	3	2.0
4	3	2.5
5	3	3.0
6	3	3.5
7	3	4.0
8	3	4.5
9	3	5.0
10	3	5.5
11	3	6.0
12	3	6.5
13	3	7.0
14	3	7.5
15	3	8.0
16	3	8.5
17	3	9.0
18	3	9.5
19	3	10.0
20	3	10.5

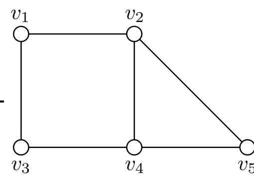
Table 3.1: *Bounds of $(p+k-2)/2$ on $\delta(G)$ in Theorem 3.10, given values of p and k.*

Theorem 3.11 (Bondy) *Let G be a graph of order $p \geq 2$, let $d_1 \leq d_2 \leq \ldots \leq d_p$ be the degree sequence of G and let k be an integer such that $1 \leq k \leq p-1$. If $d_n \leq n+k-2$, then G is k-connected.* ■

An interesting geometric characterisation of k-connected graphs was established by Lovász *et al.* [37].

Theorem 3.12 *A graph G of order p is k-connected if and only if its vertices may be represented by vectors of $\mathbb{R}^{p-k}$ such that (a) nonadjacent vertices are represented by orthogonal vectors, and (b) the vectors representing the vertices of G spans the vector space $\mathbb{R}^{p-k}$.* ■

As an example of such a representation, consider the 2-connected graph G_{19} depicted in Figure 3.5(a). A possible orthogonal representation of the graph G_{19} is shown in Figure 3.5(b).



(a) The graph G_{19}.

$$\begin{array}{ccccc} v_1 & v_2 & v_3 & v_4 & v_5 \end{array}$$
$$\begin{bmatrix} 1 & 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 1 & 0 \\ 0 & 1 & 0 & 0 & 0 \end{bmatrix}$$

(b) Orthogonal representation of the graph G_{19}.

Figure 3.5: *The graph G_{19} accompanied with an orthogonal representation of the graph.*

3.6 Construction of k-connected and k-edge connected graphs

The purpose of this section is to review theory and algorithms for generating k-connected or k-edge connected graphs. Methods for generating a k-connected (k-edge connected) graph from a given set of basic graphs are discussed in §3.6.1. In §3.6.2 methods for increasing the connectivity of a graph by the addition of edges is reviewed. A result on how to add vertices to a k-connected graph whilst maintaining k-connectivity is briefly discussed in §3.6.3.

Stellenbosch University http://scholar.sun.ac.za

3.6.1 Construction from basic graphs

The first recursive procedure to obtain a new k-connected graph from a given k-connected graph is due to Tutte [60]. Tutte established the following characterization of 3-connected graphs.

Theorem 3.13 (Tutte) *A graph is 3-connected if and only if it is a wheel or can be obtained from a wheel by repeated applications of the following two operations: (1) edge additions and (2) 3-vertex splittings.* ■

Slater [58] subsequently developed a characterization of 4-connected graphs by means of the application of a sequence of operations from a set of five different types, starting from K_5. The problem of constructing all k-connected graphs, with $k \geq 5$, is still unsolved.

The problem of constructing k-edge-connected pseudographs has been solved for all k. Lovász [38, p.45, pp.286–287] solved this problem for all $2m$-edge connected pseudographs, while Mader [43] solved the problem for all $(2m+1)$-edge connected pseudographs. Both methods are based on a series of operations consisting of subdivision of edges (inserting a vertex into an edge) and insertion of new vertices into a graph that are joined to sets of specific vertices.

3.6.2 Construction by adding edges

Graphs with a higher connectivity number may be constructed by the addition of edges to a graph with a lower connectivity number. The problem of finding a smallest cardinality set of edges whose inclusion increases the edge-connectivity number of a graph by some specified increment was solved by Frank [21] and by Noar *et al.* [53]. Finding a set of edges whose inclusion increase the connectivity number of a graph by a specified quantity proved to be a harder problem to solve.

Slater [59] determined the smallest number of edges that have to be added to a tree so as to produce a 2-connected graph. A more general result that determines the smallest number of edges that have to be added to a graph in order to produce a 2-connected graph was independently discovered by Plesník [55] and Rosenthal & Goldner [56]. Rosenthal & Goldner [56] developed an $O(p + q)$ algorithm that finds such a minimum cardinality set of edges for an arbitrary graph. Eswaran & Tarjan [18] also developed an $O(p + q)$ algorithm that finds the minimum cardinality set of edges that have to be added to an unweighted graph to produce a 2-edge-connected graph.

Watanebe & Nakamura [62] solved the problem of finding the smallest cardinality set of edges whose inclusion in a 2-connected graph produces a 3-connected graph. Hsu & Ramachandran [30] developed a linear time algorithm that finds such a set of edges.

The problem of finding a smallest cardinality set of edges whose inclusion in a 3-connected graph produces a 4-connected graph was solved by Hsu [31]. Such a set of edges can also be found in linear time.

The problem of adding a smallest cardinality set of edges to a 4-connected graph in order to obtain a 5-connected graph remains open. However, Jordán [32] considered the problem of finding a minimum cardinality set of edges whose inclusion in a k-connected graph renders it $k + 1$ connected. Sharp lower and upper bounds on this minimum were established and it was shown that the gap between these bounds is at most $k - 2$.

3.6.3 Expansion of a k-connected graph

The following result on how to expand a k-connected graph by the addition of vertices whilst maintaining k-connectivity was established by West [63, p.145]. This result will be used later in this thesis to construct a spanning subgraph of a given graph with the same connectivity number as that of G.

Theorem 3.14 (West [63]) *If G is a k-connected graph and G' is obtained from G by adding a new vertex y adjacent to at least k vertices of G, then G' is k-connected.* ■

3.7 Minimally and critically connected graphs

Section 3.6.2 was devoted to a review of how the connectivity number of a graph may be increased by the addition of edges. Another interesting question that arises is how many edges or vertices can be removed before the connectivity number of a graph is reduced. A graph G is said to be *minimally k-connected* (*minimally k-edge-connected*) if $\kappa(G) \geq k$ (or $\lambda(G) \geq k$ respectively) and $\kappa(G-e) < k$ (or $\lambda(G-e) < k$) for every edge $e \in E(G)$. A similar definition exists for the removal of vertices. A graph G is said to be *critically k-connected* (*critically k-edge-connected*) if $\kappa(G) \geq k$ (or $\lambda(G) \geq k$ respectively) and $\kappa(G-v) < k$ (or $\lambda(G-v) < k$) for every vertex $v \in V(G)$. Many nontrivial results have been established for the above mentioned concepts. Results regarding minimally k-connected and k-edge-connected graphs will be discussed next, followed by results regarding critically k-connected and critically k-edge-connected graphs.

It was found by Lick [35] that a minimally k-edge-connected graph will contain at least one vertex of degree k. Lick's result was generalised by Mader [40]. Let V_k be the number of vertices of degree k.

Theorem 3.15 (Mader [40]) *For every minimally k-edge-connected graph G, $V_k \geq k+1$.* ■

A similar result was established by Mader [41] for minimally k-connected graphs.

Theorem 3.16 (Mader [41]) *For every minimally k-connected graph G, $V_k \geq k+1$.* ■

The following theorem implicitly provides a good lower bound for V_k in a minimally k-connected graph.

Theorem 3.17 (Mader [41]) *Every circuit in a minimally k-connected graph contains a vertex of degree k.* ■

The problem of determining the number of vertices of degree k in a minimally k-connected graph is nearly solved. The following result gives bounds for this number.

Theorem 3.18 (Mader [45]) *Let $|V_k(G)|$ denote the number of vertices of degree k in a graph G and let $F(V_k) = \min\{V_k(G) \mid G$ is a minimally k-connected graph of order $p\}$. Then for all integers $p > 2k$,*

$$\left\{\frac{k-1}{2k-1}p + \frac{2k}{2k-1}\right\} \leq F(V_k) \leq \left\{\frac{k-1}{2k-1}p + \frac{2k}{2k-1}\right\} + 1.$$ ■

It has been shown that the lower bound given in Theorem 3.18 is best possible for $p \equiv 1, 3, 5, \ldots, 2n-1 \pmod{2n-1}$ and for $p \equiv 2 \pmod{2n-1}$. Furthermore, for $p \equiv 4 \pmod{2n-1}$ and for $n \geq 3$, the upper bound in Theorem 3.18 is sharp. The cases $p \equiv 6, 8, \ldots, 2n-1 \pmod{2n-1}$ remain undecided.

The problem of determining the number of vertices of degree k in a minimally k-edge-connected graph has also not been determined thus far. It was shown by Mader [42] that for $k \neq 1, 3$ there is a constant $c_k > 0$ such that for all minimally k-edge-connected graphs G of order p, $V_k \geq c_k p$. The best known lower bound for V_k was established by Bollobás *et al.* [6].

Theorem 3.19 (Bollobás *et al.* [6]) *If G is a minimal k-edge-connected graph on p vertices, then $V_k \geq k$, and, if $k \neq 1, 3$, then V_k is bounded from below as follows:*

$$V_k \geq \begin{cases} \left\lfloor \frac{p}{k+1} \right\rfloor + k & (k \text{ odd}), \\ \left\lfloor \frac{p-1}{2k+1} \right\rfloor + k + 1 & (k \text{ even}). \end{cases}$$

The following result, by Chartrand *et al.* [9], gives a bound on the degree of a specific vertex in a critically k-connected graph. It was found that this bound is the best possible.

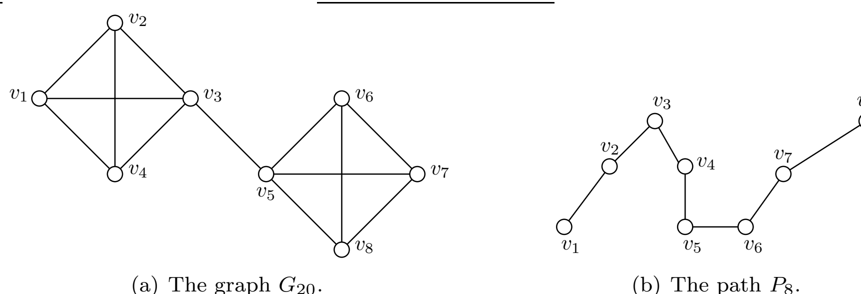
Theorem 3.20 (Chartrand *et al.* [9]) *Every critically k-connected graph G contains a vertex x such that $deg_G\ x \leq \frac{3p}{2} - 1$.* ■

Stellenbosch University http://scholar.sun.ac.za

3.8 Disconnecting a graph into more than two components

Recall that the connectivity number of a graph is the number of vertices that, when removed, disconnects the graph. The connectivity number itself does not provide any information on how many components remain in a graph G after the removal of $\kappa(G)$ vertices. A graph left with more components than another graph with the same connectivity number after the removal of $\kappa(G)$ vertices may be thought of as a more *vulnerable* graph. Consider the star $K_{1,n}$ and the path P_{n+1}, $(n \geq 3)$. Both graphs have a connectivity number of 1, but the deletion of a cut-vertex from $K_{1,n}$ produces n components, while the deletion of a cut-vertex from P_{n+1} produces only two components. The graph $K_{1,n}$ is thus more vulnerable than the graph P_{n+1}.

A connected graph G will have exactly two components after the removal of $\lambda(G)$ edges. In some graphs, considerably more edges have to be removed to produce more than two components. As an example, consider the two graphs depicted in Figure 3.6.



(a) The graph G_{20}. (b) The path P_8.

Figure 3.6: *Graphical representation of two graphs with an edge-connectivity number of 1.*

Both graphs have an edge-connectivity number of 1. However, four edges have to be removed from the graph G_{20} in Figure 3.6(a) (eg. v_1v_2, v_1v_3, v_1v_4 and v_3v_5), whilst only two edges have to be removed from the path P_8 in Figure 3.6(b) (*i.e.* v_4v_5 and v_6v_7) to produce a graph with three components.

A generalisation of the connectivity number and edge-connectivity number of a graph is now given.

Definition 3.5 *For an integer $\ell \geq 2$ and a graph G of order $p \geq \ell$, the* ℓ-connectivity number $\kappa_\ell(G)$ *is the minimum cardinality of a set U of vertices whose removal from G produces a graph with at least ℓ components or a graph with fewer than ℓ vertices.*

Definition 3.6 *For an integer $\ell \geq 2$ and a graph G of order $p \geq \ell$, the* ℓ-edge-connectivity number $\lambda_\ell(G)$ *is the minimum cardinality of a set S of edges whose removal from G produces a graph with at least ℓ components.*

From the above definitions it is clear that, $\kappa_2(G) = \kappa(G)$ and $\lambda_2(G) = \lambda(G)$. The concept of ℓ-edge-connectivity was first introduced by Boesch & Chen [3]. This concept was further studied by Goldsmith [24, 26] and by Goldsmith *et al.* [25]. In all these papers, it became aparent that $\lambda_\ell(G)$ is a difficult parameter to compute. This led to the development of heuristics and bounds to approximate the parameter.

For an integer $\ell \geq 2$, an *ℓ-way cut* of a graph G is a partition of $V(G)$ into ℓ non-empty disjoint subsets $\{V_1, V_2, \ldots, V_\ell\}$. A *minimum ℓ-way cut* is an ℓ-way cut that minimizes the weight sum of the edges between the different subsets. If a minimum ℓ-way cut of a graph can be deterimined, it may be used to calculate $\lambda_\ell(G)$. Nagamochi & Ibaraki [51] developed an algorithm that computes minimum 3-way and 4-way cuts for any edge-weighted graph. The algorithm runs in $O(qp^\ell \log(p^2/q))$ time and $O(p^2)$ space for $\ell = 3, 4$ and uses the maximum flow algorithm developed by Goldberg & Tarjan [23].

Nagamochi *et al.* [52] developed a similar algorithm to compute minimum 5-way and 6-way cuts, with the same worst-case running time as that of the algorithm used for 3-way and 4-way cuts. To the best knowledge of the author, no algoritm exists that can calculate minimum ℓ-way cuts, $\ell \geq 7$.

Stellenbosch University http://scholar.sun.ac.za

Chartrand *et al.* [10] introduced the concept of ℓ-connectivity. They also developed the notion of a *connectivity sequence* s for a graph G of order p, which is the sequence $s : \kappa_2(G), \kappa_3(G), \ldots, \kappa_p(G)$. A characterisation of ℓ-connectivity sequences was also given in the same paper, as is stated in the following theorem.

Theorem 3.21 (Chartrand *et al.* [10]) *A sequence $k_2, k_3, \ldots, k_p$ of nonnegative integers is realised as the connectivity sequence $\kappa_2(G), \kappa_3(G), \ldots, \kappa_p(G)$ of a graph G of order p if and only if there exists an integer n such that $k_2 \leq k_3 \leq \ldots \leq k_n \leq k_{n+1}$ and $k_{n+i} = p - (n+i) + 1$ for $i = 1, 2, \ldots, p-n$. Moreover, $n = \beta(G)$.* ■

It is interesting to note that the sequence s starts to decrease where $n = \beta(G)$. Similar results exist for digraphs. The *strong independence number* $\beta_S(D)$ of a digraph D is the maximum cardinality of a set S of vertices of D such that the subdigraph $\langle S \rangle_D$ is acyclic. A digraph D is said to be *strongly connected* if, for every pair of vertices u and v of D, there is a directed path from u to v. Furthermore, a *strong component* of a digraph D is a maximal induced subdigraph of D which is strongly connected. In their article, Day *et al.* [13] defined the *strong ℓ-connectivity number* $\kappa_\ell(D)$ (*strong ℓ-arc-connectivity number* $\lambda_\ell(D)$) of a digraph D as the minimum number of vertices (arcs) whose deletion from D produces a digraph with at least ℓ strong components or a digraph with at most $\ell - 1$ vertices. Furthermore, for an integer $n \geq 0$, a digraph D is said to be *strongly (n, ℓ)-connected* if $\kappa_\ell(D) \geq n$. Several sufficient conditions for a digraph to be strongly (n, ℓ)-connected were established in their paper. A characterization of the sequence of numbers $s : \kappa_2(D), \kappa_3(D), \ldots, \kappa_p(D)$ for a given digraph D of order p, defined as the *sequence of strong connectivity numbers*, was also established.

Let $n, \ell, p \in \mathbb{N}$ with $\ell \geq 2$ and $p \geq \ell + n$. The smallest integer q for which there exists a graph G of given order p such that $\kappa_\ell(G) = n$ is denoted by $q_{n,\ell}(p)$. Bounds on the size of graphs of given order and ℓ-connectivity are given in a paper by Day *et al.* [14]. More specifically, the following result was proved.

Theorem 3.22 (Day *et al.* [14]) $q_{n,\ell}(p) < q_{n,\ell-1}(p)$. ■

Among the results in their paper, exact values of $q = q_{n,\ell}(p)$ for $\ell = 2, 3$ and $n \in \{1, \ldots, 5\}$ as well as bounds on $q_{6,3}(p)$ were provided. Results for $\ell = 3$ are reproduced in Table 3.2.

n	p	$q = q_{n,3}(p)$
1	≥ 4	$p - 2$
2	≥ 5	$p - 1$
3	≥ 6	p
4	7 or 8	$p + 3$
4	≥ 9	$\lceil \frac{5p}{4} \rceil$
5	8, 9 or 10	$p + 6$
5	≥ 11	$\lceil \frac{3p}{2} \rceil$
6	9	18
6	≥ 10	$\lceil \frac{7p}{4} \rceil \leq q \leq 2p$

Table 3.2: *Exact values of $q = q_{n,3}(p)$ for $\ell = 3$, $n \in \{1, \ldots, 5\}$ as well as bounds on $q_{6,3}(p)$.*

3.9 The average connectivity number of a graph

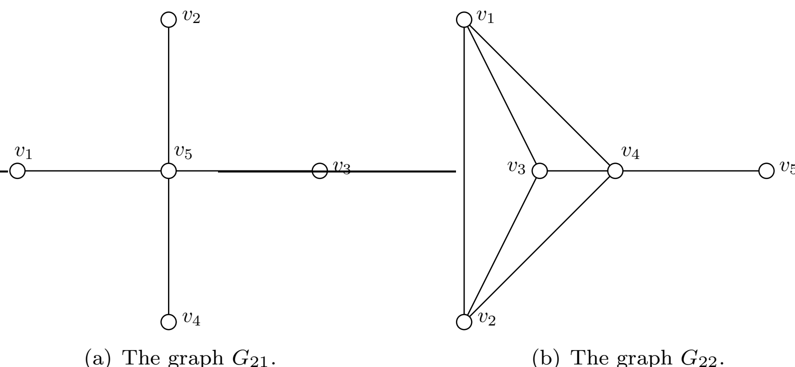
The *average connectivity number* of a graph G is defined as

$$\overline{\kappa}(G) = \frac{\sum_{u,v} \kappa(u,v)}{\binom{p}{2}}, \qquad u, v \in V(G).$$

In contrast to the connectivity number of a graph, which is the smallest number of vertices whose deletion from a connected graph disconnects the graph, the average connectivity number is the expected number of

Stellenbosch University http://scholar.sun.ac.za

vertices that must be removed in order to disconnect an arbitrary pair of non-adjacent vertices. Consider the graphs depicted in Figure 3.7. Both graphs are of order $p = 5$ and have a vertex connectivity number of 1. However $\overline{\kappa}(G_{21}) = 1$ and $\overline{\kappa}(G_{22}) = 2.2$. The graph G_{22} may thus be seen as a safer construction, as it has the higher average connectivity number.



(a) The graph G_{21}. (b) The graph G_{22}.

Figure 3.7: *Graphical representation of two graphs with a connectivity number of 1.* $\overline{\kappa}(G_{21}) = 1$ *and* $\overline{\kappa}(G_{22}) = 2.2$.

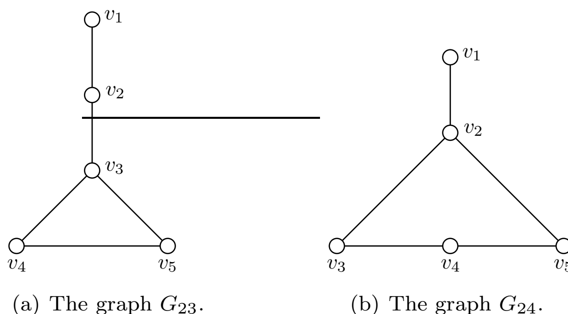
It should be clear from the definition of the average connectivity number of a graph G that $\kappa(G) \leq \overline{\kappa}(G)$. Beineke *et al.* [2] developed many results on the average connectivity number of a graph. Some of their results are listed here.

1. $\overline{\kappa}(G) = 0$ if and only if the graph G is a null graph.

2. $\overline{\kappa}(G) = 1$ if and only if the graph G is a non-trivial tree.

3. If a graph G has order p, then $\overline{\kappa}(G) \leq p - 1$, with equality if and only if G is the complete graph.

It is interesting to note that it is not necessary for a graph to have more edges in order to have a higher average connectivity number. Consider the graphs depicted in Figure 3.8. Both graphs have the same order, size and degree sequence. However $\overline{\kappa}(G_{23}) = 1.3$ and $\overline{\kappa}(G_{24}) = 1.6$.



(a) The graph G_{23}. (b) The graph G_{24}.

Figure 3.8: *Illustration of two graphs both of order and size 5.*

Beineke *et al.* [2] established upper bounds for the parameter $\overline{\kappa}(G)$.

Theorem 3.23 (Beineke *et al.* [2]) *Let G be a graph of order p with independence number $\beta(G)$. Then*

$$\overline{\kappa}(G) \leq \left[(p-1)\binom{p-\beta(G)}{2} + (p-\beta(G))\binom{\beta(G)}{2} + (p-\beta(G))^2\beta(G)\right] \Big/ \binom{p}{2} \qquad \blacksquare$$

The graph generated by the join $K_{p-\beta} + \overline{K}_\beta$ attains the bound for the average connectivity number given in Theorem 3.23. In the same paper, it was noted that bounds on the average connectivity number of a graph are more readily established if the order and size of a graph are known.

Stellenbosch University http://scholar.sun.ac.za

Theorem 3.24 (Beineke *et al.* [2]) *Let G be a graph of order p, size q and degree sequence $d_1 \geq d_2 \geq \ldots \geq d_p$. Then*

$$\overline{\kappa}(G) \leq \frac{2q}{p} - \sum_{i=1}^{p} \frac{(p-2i+1)}{p(p-1)} d_i.$$

■

The following result shows that the average connectivity number is bounded by the average degree of a graph, just as the connectivity number is bounded by the minimum degree.

Theorem 3.25 (Beineke *et al.* [2]) *Let G be a graph of order p and size q, with $q \geq p$, and let $r = 2q - p\lfloor 2q/p \rfloor$. Then*

$$\overline{\kappa}(G) \leq \frac{2q}{p} - \frac{r(p-r)}{p(p-1)}.$$

■

It has been shown that the bound given in Theorem 3.25 is sharp. Beineke *et al.* [2] also investigated what the *maximum average connectivity number* $\mu_G(p,q)$ for a graph with p vertices and q edges may be.

Theorem 3.26 (Beineke *et al.* [2]) *The maximum average connectivity number in a graph G with p vertices and q edges is, with $r = 2q - p\lfloor 2q/p \rfloor$,*

$$\mu_G(p,q) = \begin{cases} 0 & \text{if } q = 0, \\ \frac{2}{p(p-1)} & \text{if } q = 1, \\ \frac{6}{p(p-1)} & \text{if } q = 2, \\ \frac{2q(q-1)}{p(p-1)} & \text{if } 3 \leq q \leq p-1, \\ \frac{2q(p-1)-r(p-r)}{p(p-1)} & \text{if } q \geq p \geq 4. \end{cases}$$

■

3.10 Uniformly connected graphs

In their paper, Beineke *et al.* [2] also defined a graph G to be *uniformly k-connected* if $\kappa(u,v) = k$ for all pairs of vertices u and v. This implies that $\kappa(G) = \overline{\kappa}(G) = k$. It is easy to see that a graph G is uniformly

1. 0-connected if and only if it has no edges,
2. 1-connected if and only if it is a tree,
3. 2-connected if and only if it is a cycle,
4. $(p-2)$-connected if and only it is the result of removing $\lfloor \frac{p}{2} \rfloor$ edges from K_p and
5. $(p-1)$-connected if and only if G is complete.

The following interesting result relates the notions of minimally and critically connectedness to uniformly connectedness.

Theorem 3.27 (Beineke *et al.* [2]) *Let G be a uniformly k-connected graph.*
(a) If $k \geq 1$, then G is minimally k-connected.
(b) If $k \geq 2$, then G is critically k-connected.

■

Stellenbosch University http://scholar.sun.ac.za

3.11 Construction using approximation algorithms

The focus of this thesis is on the construction of spanning subgraphs with specified connectivity number. To simplify the writing, the following two definitions are introduced.

Definition 3.7 *A* connectivity preserving *spanning subgraph G' constructed from a graph G is a spanning subgraph such that $\kappa(G') = \kappa(G)$.* ■

Definition 3.8 *A* connectivity reducing *spanning subgraph G' constructed from a graph G is a spanning subgraph such that $\kappa(G') < \kappa(G)$.* ■

In a recent article Kortsarz & Nutov [34] explained how a connectivity preserving spanning subgraph may be constructed by making use of an *approximation algorithm* that they developed. An approximation algorithm returns an output that differs by at most a fixed factor more than the optimal solution for the problem at hand. This *fixed factor* is known as an *approximation ratio*. In this case, an optimal solution is a spanning subgraph of minimum cost. The approximation ratio for the algorithm of Kortsarz & Nutov [34] is $O\left(\ln k \cdot \min\left\{\sqrt{k}, p/(p-k)\ln k\right\}\right)$. It has a worst-case running time of $O(k^2pq^2)$. This is indeed a very powerful result and has the lowest approximation ratio of all existing approximation algorithms for the construction of connectivity preserving spanning subgraphs.

3.12 Chapter Summary

In this chapter, basic definitions were discussed concerning graph connectivity, followed by a survey of articles relating to this concept. The most recent results concerning the bounds on certain connectivity parameters were also stated, each followed with a short discussion. The reader should now have a firm understanding of the various graph connectivity concepts that will further be discussed or built upon in the remainder of this thesis.

Stellenbosch University http://scholar.sun.ac.za

Stellenbosch University http://scholar.sun.ac.za

Chapter 4

Spanning Subgraphs with Connectivity Number $\leq k$

In this chapter, algorithms for constructing connectivity preserving and reducing spanning subgraphs are discussed.

Algorithms for returning information about a graph, such as the connectivity number and a minimum cut-set are introduced in sections §4.1 and §4.2. These algorithms are utilised by some of the algorithms described in the following sections.

4.1 Finding the connectivity number of a graph

As mentioned in §3.3, the best result known for computing the connectivity number of a graph G was established by Esfahanian & Hakimi [17]. Define $N(u,v)$ for a given graph G as follows: If $uv \in E(G)$, then $N(u,v) = p-1$, else $N(u,v) = \kappa(u,v)$. The approach of Esfahanian & Hakimi [17] is presented here in pseudocode as Algorithm 3.

Algorithm 3 *Computing $\kappa(G)$ of a graph G*

Input: A graph G.
Output: The connectivity number, $\kappa(G)$.

1: $i \leftarrow 1$, $N_{\min}(u,v) \leftarrow p-1$, select a vertex $u \in V(G)$ of minimum degree and let $A(u) = \{u_1, u_2, \ldots, u_{\delta(G)}\}$ be the neighbours of vertex u.
2: **for** each $v \in V(G) \setminus (A(u) \cup u)$ **do**
3:     compute $N(u,v)$ and set $N_{\min} \leftarrow \min\{N_{\min}, N(u,v)\}$.
4: **end for**
5: **for** $j = i+1$ to $\delta(G)-1$ **do**
6:     **if** $i \geq \delta(G)-2$ **or** $i \geq N_{\min}$ **then**
7:         go to Step 14
8:     **end if**
9:     **if** $u_i u_j \notin E(G)$ **then**
10:         compute $N(u_i,u_j)$, and set $N_{\min} \leftarrow \min\{N_{\min}, N(u_i,u_j)\}$.
11:     **end if**
12: **end for**
13: $i \leftarrow i+1$, go to Step 5.
14: $\kappa(G) \leftarrow N_{\min}$; stop.

Algorithm 3 starts by finding a vertex $u \in G$ of minimum degree and strores the neighbours of u in the set $A(u)$ (line 1). The maximum number of internally disjoint u-v paths, with $v \in V(G) \setminus (A(u) \cup u)$, are then calculated in the for loop spanning lines 2 to 4. The algorithm then computes the maximum number

Stellenbosch University http://scholar.sun.ac.za

of internally disjoint paths between all pairs of non-adjacent vertices in the set $A(u)$ (lines 5 to 13). The minimum number of internally disjoint paths that were calculated for any pair of vertices is then taken as the connectivity number for the graph G (line 14). In their paper Esfahanian & Hakimi [17] showed that Algorithm 3 makes $p-\delta(G)-1+\frac{1}{2}\kappa(G)(2\delta(G)-\kappa(G)-3)$ calls to $N(u,v)$. If the max-flow min-cut theorem of Ford & Fulkerson [20], with worst-case running time $O\left(pq^2\right)$, is used to compute $N(u,v)$, it follows that the worst-case running time of Algorithm 3 is $O\left(pq^2\left[p-\delta(G)-1+\frac{1}{2}\kappa(G)(2\delta(G)-\kappa(G)-3\right]\right) = O\left(p^3q^2\right)$.

The advantage of Algorithm 3 is that it is the fastest known algorithm for computing $\kappa(G)$. However, a disadvantage of this algorithm, is that it does not return a minimum cut-set of a graph, which is a requirement for some of the algorithms considered in this chapter.

4.2 Finding a minimum cut-set of a graph

The next algorithm, Algorithm *Cut-Vertex Set*, finds a minimum cut-set for a given graph. This algorithm has a relatively low running time. The algorithm employs a *bit-vector* to mark those vertices to be removed from the vertex set $V(G)$. The size k of the set of vertices to be removed is varied from 1 to $p-1$, while all possible combinations for each set of k vertices are tested. Once a cut-set has been found, the algorithm terminates.

Algorithm *Cut-Vertex Set* takes the adjacency matrix N of a graph as input. A vector with two elements is returned as output. The first element indicates the cardinality of the cut-set found, while the second element stores the actual vertices comprising the cut-set.

4.2.1 Working of Algorithm *Cut-Vertex Set*

The *Cut-Vertex Set* algorithm is presented in two parts. Procedure 1 is contained or called within Algorithm 4 on line 22. Algorithm *Cut-Vertex Set* finds a minimum cut-set by performing a brute force search through all possible combinations of the vertices of G. A minimum cut-set is found by employing the bit-vector, *nodes*, in which a position is stored for every vertex in G. A one in position i of *nodes* indicates that vertex $v_i \in G$ is being tested for forming part of a cut-set. The set of vertices chosen as a cut-set is then stored as a binary string.

Lines 1 to 3 test whether the input graph is connected. If the graph is disconnected, $CVS(1)$ is initialised to 0 and the algorithm terminates. The next part of Algorithm 4 is responsible for testing all combinations of vertices, each set forming a possible cut-set. The variable m stores the cardinality of the set of vertices being tested as a cut-set. Its value is incremented from 1 to $p-1$ in the for loop spanning lines 6 to 25. To test for all $\binom{p}{m}$ combinations of m cut vertices, the value of the binary string is initialised with the first m vertices chosen as the cut-set. The value of the binary string is then incremented in every iteration of the for loop spanning lines 13 to 24. The function dec2bin(i) on line 14 converts the value of the counter i to a binary string of length p and stores it in the variable *nodes* (all positions not storing a one is filled with a zero). If *nodes* contains more than m ones, the for loop immediately moves to the next iteration. The for loop eventually terminates after the last m vertices have been tested for inclusion in a cut-set. The initial (variable *start*) and terminal (variable *stop*) values for the counter of this for loop is computed on lines 9 to 12.

Once a possible cut-set has been identified by Algorithm 4, control is passed to Procedure 1. Here, a new adjacency matrix *Ntmp* is constructed by removing those rows and columns corresponding to cut vertices (lines 1 to 14). *Ntmp* is then tested for connectivity by means of Floyd's shortest path algorithm (line 15). If the graph is disconnected by the removal of the vertices stored in the cut-set, then the adjacency matrix contains at least one column and row of which the entries have infinite values, indicating that the graph specified by the adjacency matrix *Ntmp* is disconnected (lines 16 to 22). This indicates that a valid cut-set has been found. The cardinality of the cut-set is stored in $CVS(1)$ (line 20) while the bit-vector for the cut-set is stored in $CVS(2)$ (line 21). The algorithm then immediately terminates. If no cut-set of cardinality m could be found, control returns to Algorithm 4. A special case exists in the event where all possible combinations of vertices have been tested in search of a cut-set, but none could

Stellenbosch University http://scholar.sun.ac.za

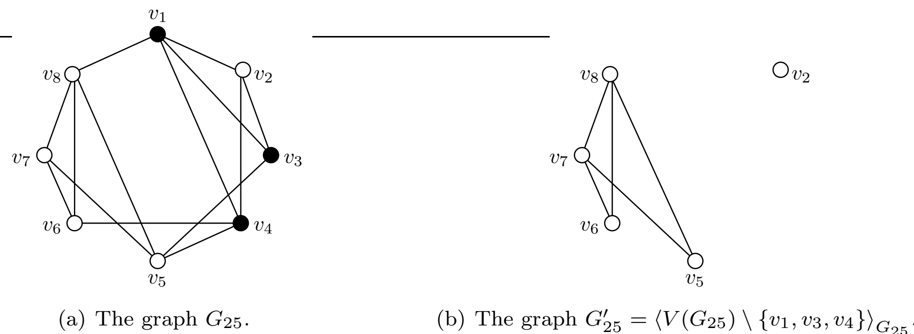
Algorithm 4 *Cut-Vertex Set: Main*

Input: The adjacency matrix N of a graph for which a minimum cut-set is sought.
Output: *CVS*, a two-element array containing a minimum cut-set cardinality and vertices comprising a minimum cut-set.

```
1:  if (N = zero matrix) or (N contains a row and column of zeros) then
2:      CVS(1) ← 0.
3:      stop.
4:  else
5:      CVSFound ← false.
6:      for m ← 1 to p − 1 do
7:          start ← 0.
8:          stop ← 0.
9:          for i ← 1 to m do
10:             start ← start + 2^(i−1).
11:             stop ← stop + 2^(p−i).
12:         end for
13:         for i ← start to stop do
14:             nodes ← dec2bin(i).
15:             count ← 0.
16:             for j ← 1 to p do
17:                 if nodes(j) = 1 then
18:                     count ← count + 1.
19:                 end if
20:             end for
21:             if count = m then
22:                 Call TestCVS.
23:             end if
24:         end for
25:     end for
26: end if
```

be found. In this case, the first entry in *CVSList* is initialised to ∞. This is done to indicate to the algorithm calling Algorithm *Cut-Vertex Set* that the graph tested was a complete graph and does not contain a cut-set.

Example 4.1
Consider the graph G_{25} depicted in Figure 4.1(a). The cardinality of a minimum cut-set returned by Algorithm *Cut-Vertex Set* is three, with corresponding bit-vector 00001101. Thus, the minimum cut-set returned by the algorithm consists of the vertices v_1, v_3 and v_4, coloured black in Figure 4.1. If this cut-set is removed, the graph G'_{25}, shown in Figure 4.1(b), is obtained.



(a) The graph G_{25}. (b) The graph $G'_{25} = \langle V(G_{25}) \setminus \{v_1, v_3, v_4\}\rangle_{G_{25}}$.

Figure 4.1: *The graphs G_{25} and G'_{25}. The set of vertices $\{v_1, v_3, v_4\}$ comprising a possible minimum cut-set in G_{25} are coloured black.*

Note that this is just one of eight distinct minimum cut-sets. Table 4.1 lists all possible minimum cut-sets for the graph G_{25}. ■

Stellenbosch University http://scholar.sun.ac.za

Procedure 1 *Cut-Vertex Set: TestCVS*

Output: Returns a minimum cut-set if the given set of vertices comprises a minimum cut-set.

```
1: V ← ∅.
2: pos ← 1.
3: for j ← 1 to p do
4:   if nodes(p + 1 − j) = 0 then
5:     V(pos) ← j.
6:     pos ← pos + 1.
7:   end if
8: end for
9: Ntmp ← zeros(length(V)).
10: for j ← 1 to length(V) do
11:   for k ← 1 to length(V) do
12:     Ntmp(j, k) ← N(V(j), V(k)).
13:   end for
14: end for
15: NShortestPaths ← Floyd(Ntmp, length(V)).
16: val ← max(NShortestPaths).
17: maxVal ← max(val).
18: if maxVal = inf then
19:   CVSFound ← true.
20:   CVS(1) ← m.
21:   CVS(2) ← i.
22:   stop.
23: else
24:   if (i = stop) and (m = p − 1) and (CVSFound = false) then
25:     CVS(1) ← inf.
26:   end if
27: end if
```

Bit-Vector	CVS
00001101	$\{v_1, v_3, v_4\}$
00010011	$\{v_1, v_2, v_5\}$
00011001	$\{v_1, v_4, v_5\}$
00110001	$\{v_1, v_5, v_6\}$
10001100	$\{v_3, v_4, v_8\}$
10011000	$\{v_4, v_5, v_8\}$
10110000	$\{v_5, v_6, v_8\}$
11001000	$\{v_4, v_7, v_8\}$

Table 4.1: *The minimum cut-sets of the graph G_7 in Example 4.1.*

It should be noted here that Algorithm *Cut-Vertex Set* may easily be adapted to find all possible cut-sets of a graph. This may be achieved by changing the variable *CVS* to contain rows, each representing a different cut-set, with cardinalities ranging from the lowest to the highest. Only one algorithm discussed in this thesis is required to search all minimum cut-sets of the graph, as it needs to find a cut-set that is also a clique. This special case has the same worst-case running time as that of Algorithm *Cut-Vertex Set*. This can be seen from the fact that, if the cardinality of a minimum cut-set is k, then the worst-case running time for Algorithm *Cut-Vertex Set* occurs if all $\binom{p}{k}$ combinations of vertices are tested to locate a minimum cut-set. This is the same as constructing a list of all minimum cut-sets (a possible $\binom{p}{k}$ cut-sets exist). Searching for all possible cut-sets drastically increases the average running time of the algorithm. It is clear that the worst-case running time of the algorithm occurs when a complete graph is given as input. Table 4.2 shows benchmark tests on the running time of Algorithm *Cut-Vertex Set* that have been performed on a 3GHz Intel PC with 512MB RAM. The running time of complete graphs from K_1 up to K_{22} were investigated.

Stellenbosch University http://scholar.sun.ac.za

	Running time
i	**Time**
1	0
2	0.009
3	0.007
4	0.011
5	0.021
6	0.032
7	0.056
8	0.114
9	0.253
10	0.575
11	1.322
12	3.037
13	6.912
14	15.554
15	34.603
16	76.881
17	168.928
18	371.370
19	813.542
20	1773.985
21	3858.282
22	8331.452

Table 4.2: *Benchmark tests for Algorithm* Cut-Vertex Set *on complete graphs. Testing was performed on a 3GHz Intel PC with 512MB RAM. Complete graphs were used as input ranging from K_1 up to K_{22}. The running time is given in seconds.*

An input graph will, on average, comprise considerably fewer edges than a complete graph of the same order, ensuring a much faster computation time than listed in Table 4.2. However, these tests confirm the importance of terminating the algorithm once a cut-set of minimum cardinality has been discovered.

4.2.2 Time Complexity of Algorithm *Cut-Vertex Set*

The for loop spanning lines 6 to 25 of Algorithm 4 contributes $O(p)$ to the worst-case running time of Algorithm *Cut-Vertex Set*. The nested for loop spanning lines 9 to 11 has a running time of $O(k)$ when $m = k$. Although the counter of the for loop on lines 13 to 24 runs from *start* to *stop* (decimal values of the binary strings of the first and last cut-set of cardinality m respectively), Procedure 1 is called only $\binom{p}{k}$ times on line 22 (for cut-sets of cardinality m). The function dec2bin(i) has a worst-case running time of $O(\log i)$. The other $\left(stop - start - \binom{p}{k}\right)$ iterations of the for loop (starting on line 13) have a linear ($O(p)$) running time due to the nested for loop spanning lines 16 to 20.

The nested for-loops spanning lines 10 to 14 of Procedure 1 contribute $O(p^2)$ to the worst-case running time of this algorithm. The worst-case running time of Floyd's shortest path algorithm (line 15) is $O(p^3)$. The remaining part of Procedure 1 either has a linear (the for loop spanning lines 3 to 8) or constant (line 16) running time. Procedure 1 thus contributes $O(p^3)$ to the running time of Algorithm *Cut-Vertex Set*.

As a result, the for loop spanning lines 14 to 29 of Algorithm 4 has a worst-case running time of $O\left(\left(stop - start - \binom{p}{m}\right) \times p + O\left(\binom{p}{k} \times \left(p + p^3\right)\right)\right) = O\left(\binom{p}{k}p^3\right) = O\left(p^k p^3\right) = O\left(p^{k+3}\right)$.

Since k may be as large as $p - 1$, this algorithm is most suitable for graphs where k is relatively small compared to p.

Stellenbosch University http://scholar.sun.ac.za

4.3 Finding disjoint paths in a graph

In this section a method is described for finding disjoint paths starting from a certain vertex, referred to as a *source*, and ending in another vertex or set of vertices, referred to as a *sink*. Disjoint paths may be found in polynomial time if the Shortest Augmenting Path Algorithm is used in conjunction with Ford's Algorithm, both described in this section.

4.3.1 Ford's Algorithm

The algorithm described in this section was developed by Ford [19] and is used to find the shortest path from a specified vertex s to all other vertices in a graph. In addition, vertices are labelled with the shortest distance from the vertex s using a *breadth-first search* approach. A breadth-first search algorithm is an algorithm that first searches for the neighbours of all the vertices that it has already searched for. In this case, the neighbours of vertex s are searched for and located. The algorithm then continues by searching for the neighbours of all discovered vertices and continues to iterate in this manner until all vertices in the graph are found. Ford's Algorithm has a worst-case running time of $O(pq)$, where p and q are the order and size of a given graph respectively. The algorithm is presented in pseudocode in Algorithm 5.

An advantage of Ford's Algorithm is that it always terminates, even if there are negative cycles (a cycle with a negative weight) present in the graph. However, the only aspect of Ford's Algorithm that is relevant to this thesis, is its labelling of vertices using a breadth-first search approach. The labelled vertices are used later by the Shortest Augmenting Path algorithm. The labelling process in Algorithm 5 is now described. The sets $\mathcal{S}$ and $\overline{\mathcal{S}}$ are initialised as empty sets (lines 1 and 2), after which the distance label of each vertex v of the graph G is initialised as $d(v) = \infty$ (line 3). The vertex s is added to the set $\overline{\mathcal{S}}$ and its distance label is set to zero (lines 4 and 5). The while loop spanning lines 8 to 33 then iterates until all vertices have been labelled, which is achieved when all vertices are included in the set $\mathcal{S}$. In line 9, a vertex in $\overline{\mathcal{S}}$ with the minimum distance label in the set $\overline{\mathcal{S}}$ is assigned to the vertex u. For the first iteration of this while loop, only the vertex s is included in $\overline{\mathcal{S}}$ and hence is assigned to vertex u. The chosen vertex u is then removed from the set $\overline{\mathcal{S}}$ and inserted in the set $\mathcal{S}$ (lines 10 to 11). The variable *labelCount* is used to test for negative cycles in the graph. If a negative cycle is found, the algorithm terminates (lines 12 to 15).

All neighbours of vertex u are now traversed. The for loop spanning lines 16 to 29 selects a neigbour v and determines whether $d(v) > d(u)+c_{uv}$, where c_{uv} is the weight of the edge joining the vertices u and v (line 17). If the condition evaluates to true, the distance label of vertex v is adjusted to $d(v) = d(u)+c_{uv}$ (line 18). The vertex v is inserted into the set $\overline{\mathcal{S}}$, if it was not already included as an element of the set (lines 21 to 26). The for loop spanning lines 16 to 29 terminates once all neighbours of vertex u have been traversed. If, after this for loop, $\overline{\mathcal{S}} = \emptyset$, all vertices have been inserted into the set $\mathcal{S}$ and the algorithm terminates (lines 30 to 32). If this is not the case, the while loop spanning lines 16 to 29 is repeated.

A breadth-first search technique is achieved by the way in which vertices are transferred between the sets $\mathcal{S}$ and $\overline{\mathcal{S}}$. The following example demonstrates this technique.

Example 4.2
Consider the directed, weighted graph G_{26} depicted in Figure 4.2(a), with arc weights as indicated. Ford's algorithm is implemented on this graph, starting from the vertex s. Vertex s is labelled first with distance $d(s) = 0$ (see Figure 4.2(b). The vertex s is now removed from the set $\overline{\mathcal{S}}$ and inserted into the set $\mathcal{S}$ (lines 10 to 11 in Algorithm 5). All the out-neigbours of vertex s are now relabelled as the condition on line 18 evaluates to true (see Figure 4.2(c)). The sets $\mathcal{S}$ and $\overline{\mathcal{S}}$ are updated such that $\mathcal{S} = \{s\}$ and $\overline{\mathcal{S}} = \{v_1, v_3, v_4\}$ (lines 22 and 23).

The while loop is repeated, with vertex u initialised as vertex v_3 (line 9). Vertex v_3 is removed from the set $\overline{\mathcal{S}}$ and inserted into the set $\mathcal{S}$ (lines 10 and 11), hence $\mathcal{S} = \{s, v_3\}$ and $\overline{\mathcal{S}} = \{v_1, v_4\}$. Control is passed to the for loop spanning lines 16 to 29, where the distance labels of the out-neighbours of vertex v_3 are tested for possible relabelling. As $d(v_2) = \infty$, the distance label of v_2 is relabelled as

Stellenbosch University http://scholar.sun.ac.za

Algorithm 5 *Ford's Algorithm*

Input: A graph G with edge weights and the vertex s from which shortest paths should be found.
Output: Shortest paths from vertex s to all other vertices in G.

1: $\mathcal{S} \leftarrow \emptyset$.
2: $\overline{\mathcal{S}} \leftarrow \emptyset$
3: **for** every vertex $v_i \in V(G)$ do $d(v_i) \leftarrow \infty$ **end for**
4: $\overline{\mathcal{S}} \leftarrow \overline{\mathcal{S}} \cup s$
5: $d(s) \leftarrow 0$
6: $p(s) \leftarrow 0$
7: **for** every vertex $v_i \in V(G)$ do $labelCount(v_i) \leftarrow 0$ **end for**
8: **while** $|\mathcal{S}| \leq p$ **do**
9: Let $u \leftarrow \overline{\mathcal{S}}$ such that $d(u) = \min\left\{d(v_j) : v_j \in \overline{\mathcal{S}}\right\}$
10: $\mathcal{S} \leftarrow \mathcal{S} \cup \{u\}$
11: $\overline{\mathcal{S}} \leftarrow \overline{\mathcal{S}} - \{u\}$
12: $labelCount(u) \leftarrow labelCount(u) + 1$
13: **if** $labelCount(u) = p$ **then**
14: **stop** [negative cycle has been found]
15: **end if**
16: **for** every vertex $v \in V(G)$ **do**
17: **if** $uv \in E(G)$ **then**
18: **if** $d(v) > d(u) + c_{uv}$ **then**
19: $d(v) \leftarrow d(u) + c_{uv}$
20: $p(v) \leftarrow u$
21: **if** $v \in \mathcal{S}$ **then**
22: $\mathcal{S} \leftarrow \mathcal{S} - \{v\}$
23: $\overline{\mathcal{S}} \leftarrow \overline{\mathcal{S}} \cup \{v\}$
24: **else if** $v \notin \overline{\mathcal{S}}$ **and** $v \notin \mathcal{S}$ **then**
25: $\overline{\mathcal{S}} \leftarrow \overline{\mathcal{S}} \cup \{v\}$
26: **end if**
27: **end if**
28: **end if**
29: **end for**
30: **if** $\overline{\mathcal{S}} = \emptyset$ **then**
31: **stop** [no temporary vertices to set permanent]
32: **end if**
33: **end while**

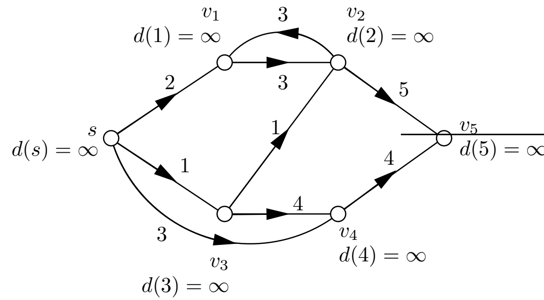
$d(v_2) = d(v_3) + c_{v_3v_2} = 2$ and the vertex v_2 is inserted into the set $\overline{\mathcal{S}}$, so that $\overline{\mathcal{S}} = \{v_1, v_4, v_2\}$ (lines 18 to 23). The distance label of vertex v_4 is less than $d(v_3) + c_{v_3v_4} = 1 + 4 = 5$; hence its distance label remains unchanged (see Figure 4.2(d)). Currently, $\mathcal{S} = \{s, v_3\}$ and $\overline{\mathcal{S}} = \{v_1, v_4, v_2\}$.

The while loop is repeated, this time with u initialised as vertex v_2; hence vertex v_2 is removed from the set $\overline{\mathcal{S}}$ and inserted into the set $\mathcal{S}$ ($d(v_2) = d(v_1) = 2$ and vertex v_2 was arbitrarily selected as the vertex with the lowest distance label). The only neighbour of vertex v_2 whose distance label can be updated is vertex v_5, with a relabelled distance of $d(v_5) = 7$ (see Figure 4.2(e)). The sets $\mathcal{S}$ and $\overline{\mathcal{S}}$ are again updated so that $\mathcal{S} = \{s, v_3, v_2\}$ and $\overline{\mathcal{S}} = \{v_1, v_4, v_5\}$ (lines 22 and 23).

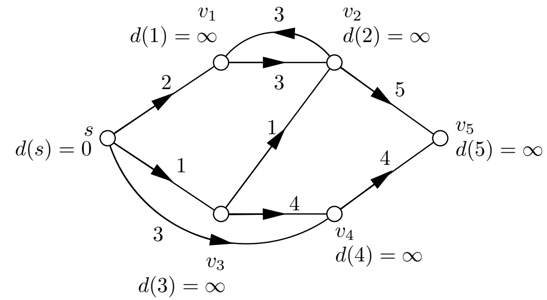
The while loop is repeated, with vertex u initialised as vertex v_1; hence vertex v_1 is removed from the set $\overline{\mathcal{S}}$ and inserted into the set $\mathcal{S}$. As no distance labels can be updated, vertex v_1 is again removed from the set $\overline{\mathcal{S}}$ and inserted into the set $\mathcal{S}$. The while loop is repeated in a similar fasion for the remaining vertices in $\overline{\mathcal{S}}$ until the set is empty. The final result is depicted in Figure 4.2(f). ■

Thus far, nothing has been said about the variable $p(v)$ mentioned on lines 6 and 20. This variable may be used to trace back the shortest path from any vertex to the source s. However, this information is not of importance for this thesis.

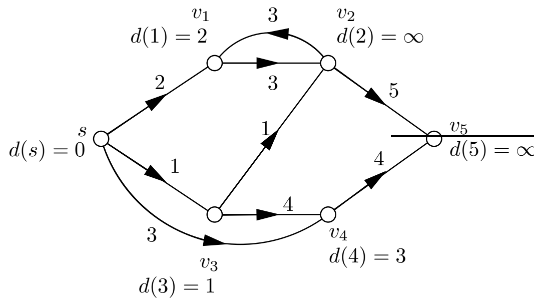
Stellenbosch University http://scholar.sun.ac.za



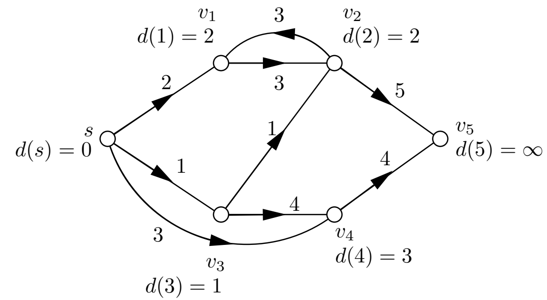
(a) The digraph G_{26}.



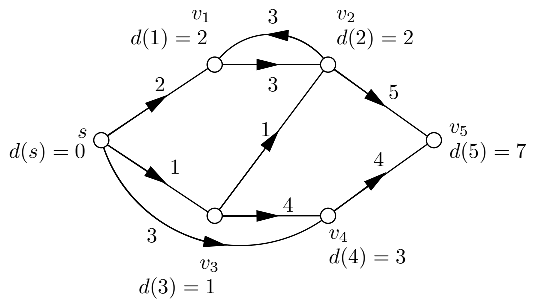
(b) The digraph G_{26} before first iteration of while loop. $\mathcal{S} = \overline{\mathcal{S}} = \emptyset$.



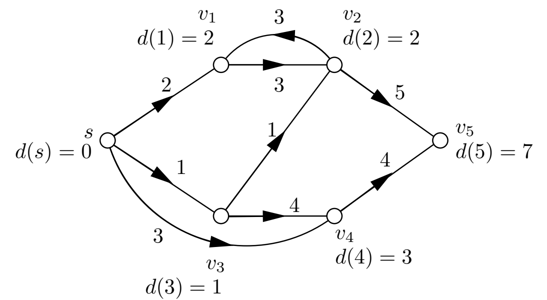
(c) First iteration of while loop. $\mathcal{S} = \{s\}$, $\overline{\mathcal{S}} = \{v_1, v_3, v_4\}$.



(d) Second iteration of while loop. $\mathcal{S} = \{s, v_3\}$, $\overline{\mathcal{S}} = \{v_1, v_4, v_2\}$.



(e) Third iteration of while loop. $\mathcal{S} = \{s, v_3, v_2\}$, $\overline{\mathcal{S}} = \{v_1, v_4, v_5\}$.



(f) Sixth iteration of while loop. $\mathcal{S} = \{s, v_3, v_2, v_1, v_4, v_7\}$, $\overline{\mathcal{S}} = \emptyset$.

Figure 4.2: *Graphical representations of the progress of Algorithm 5 during the calculation of the distance labels of every vertex of the graph* G_{26}.

4.3.2 Shortest Augmenting Path Algorithm

In many applications, the edges joining vertices in a graph model have certain limitations on the volume of some substance that may be sent across a link from one station to the other, in some infrastructure network. For instance, a water pipe may have a limit on the volume of water that is allowed to pass through it per unit of time. This may be due to physical limitations of the water pipe itself, or that the station to which the pipe is connected is only able to handle a specified amount of water during a given time period. The objects that travel or flow through a network are called *flow units* or *units* for short. A *maximum flow* from a vertex u to a vertex v is the maximum number of units that can be sent from vertex u to vertex v. For instance, if two stations in a network of water pipes are joined by two pipelines, of which the first pipeline can transmit $5m^3/s$ of water and the second $3m^3/s$ of water, with both flows in the same direction then the maximum flow from the one station to the other is $8m^3/s$. The Shortest Augmenting Path Algorithm calculates the maximum flow in a directed graph from a source vertex s to a sink vertex t. It also returns the paths along which the maximum flow from vertex s to vertex t takes place. The name of the algorithm derives from its mechanism. It always searches for the shortest paths in the network, using Ford's Algorithm, to augment the flow from vertex s to vertex t. The Shortest Augmenting Path Algorithm has a worst-case running time of $O(pq^2)$, where p and q are the order and size of a given graph respectively. The algorithm is presented in pseudocode in Algorithm 6.

Stellenbosch University http://scholar.sun.ac.za

Algorithm 6 *Shortest Augmenting Path Algorithm*

Input: A directed graph G with a unit flow limit on each arc. A source vertex s and a sink vertex t.
Output: The maximum flow possible through G from vertex s to vertex t.

```
 1: E'(G) ← the reverse of all arcs in E(G)
 2: Determine the distance labels for all vertices from vertex t using Ford's algorithm and E'(G)
 3: r_uv ← upper bound on flow allowed for arc uv
 4: u ← s
 5: while d(s) ≤ p do
 6:   Let v be the neighbour of u with the lowest distance label.
 7:   if r_uv > 0 and d(u) = d(v) + 1 then
 8:     p(v) ← u
 9:     u ← v
10:     if v = t then
11:       Use p(v) to determine the path P from vertex s to t
12:       Let δ ← min {r_uv : uv ∈ P}
13:       Augment δ units of flow along path P
14:       u ← s
15:     end if
16:   else
17:     d(u) ← d(u) + 1
18:     if u ≠ s then
19:       u ← p(u)
20:     end if
21:   end if
22: end while
```

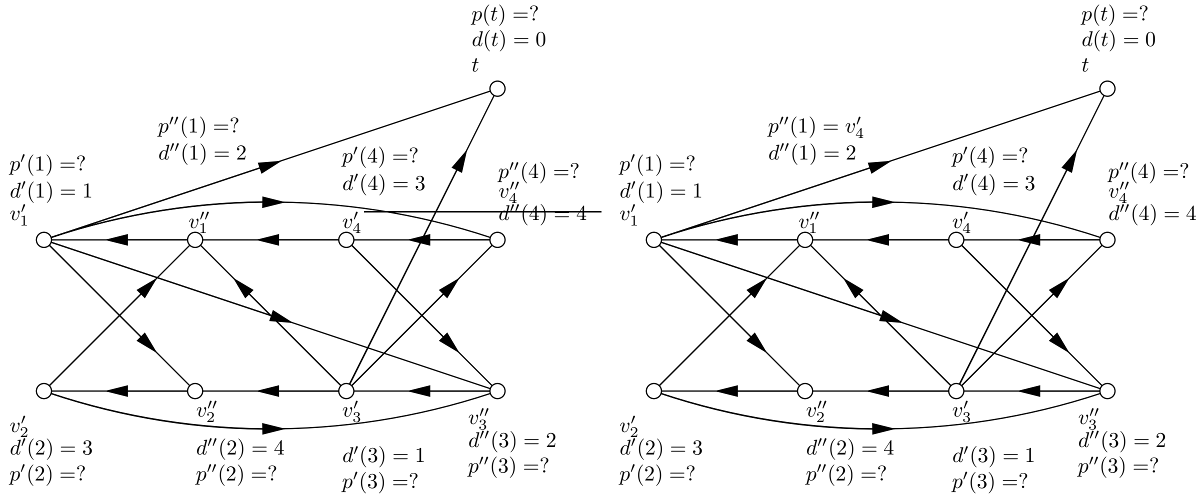
The working of Algorithm 6 is now described. The algorithm starts by reversing all arcs in the graph and stores the reversed set of arcs in the set $E'(G)$ (line 1). This is necessary to compute the correct distance labels from the vertex sink t to every other vertex (line 2). For the rest of the algorithm, the original set of arcs, $E(G)$, is used. The variable r_{uv} is defined as the maximum flow allowed along the arc uv from vertex u to vertex v. It is also known as the *residual capacity* of the arc uv. The residual capacity for each arc uv is assigned to r_{uv} on line 3. The sink s is assigned to vertex u on line 4, after which control is passed to the while loop spanning lines 5 to 22. The out-neighbour of vertex u with the minimum distance is assigned to vertex v (line 6). The if statement on line 7 evaluates to true if both conditions $r_{uv} > 0$ and $d(u) = d(v) + 1$ are satisfied. The variable $p(v)$ stores the predecessor of the vertex v along the path traversed thus far, namely vertex u. If $v = t$ (see line 10), then a valid path P from vertex s to vertex t has been found. The variable $p(v)$ may now be used to reconstruct the path P. The minimum residual capacity of all arcs along the path P is assigned to the variable δ on line 11. This is used to augment δ units of flow along the path P on line 13. This means that, for every edge uv along the path P, $r_{uv} = r_{uv} - \delta$ and $r_{vu} = r_{vu} + \delta$. The vertex u is again initialised as vertex s, as the end of the if statement spanning lines 10 to 15 has been reached and control is again passed to the while loop starting on line 5. If the if statement on line 7 evaluates to false, then control is passed to line 17, where the distance label of vertex u is increased by one. Furthermore, if $u \neq s$, then vertex u is assigned the value of its predecessor, $p(u)$ (line 19). If the while condition on line 5 evaluates to true, then the while loop is repeated, otherwise the algorithm terminates.

Maximum flow algorithms have the advantage that they may easily be adapted to incorporate graphs with more than one sink and/or source vertex. For this thesis, flow to more than one sink vertex is of importance. This may be achieved by adding another vertex to the graph, called a *supersink*. All vertices that act as sinks are joined to the supersink with infinite residual capacities for each of these arcs. Only graphs with residual capacities of one or zero are of importance for this thesis. In this case, the arcs from the sinks to the supersink have their residual capacities set to one and not infinity. A residual capacity of $r_{uv} = 1$ implies that an arc exists that joins the vertex u to the vertex v. If $r_{uv} = 0$, no arc exists between the vertices u and v. These concepts are illustrated in the following example. In the graphical representations of graphs that follow, the residual capacities of each arc are not indicated so as to render the graphs less cluttered. The existance of an arc in a graph indicates a residual capacity in the direction of the arc.

Stellenbosch University http://scholar.sun.ac.za

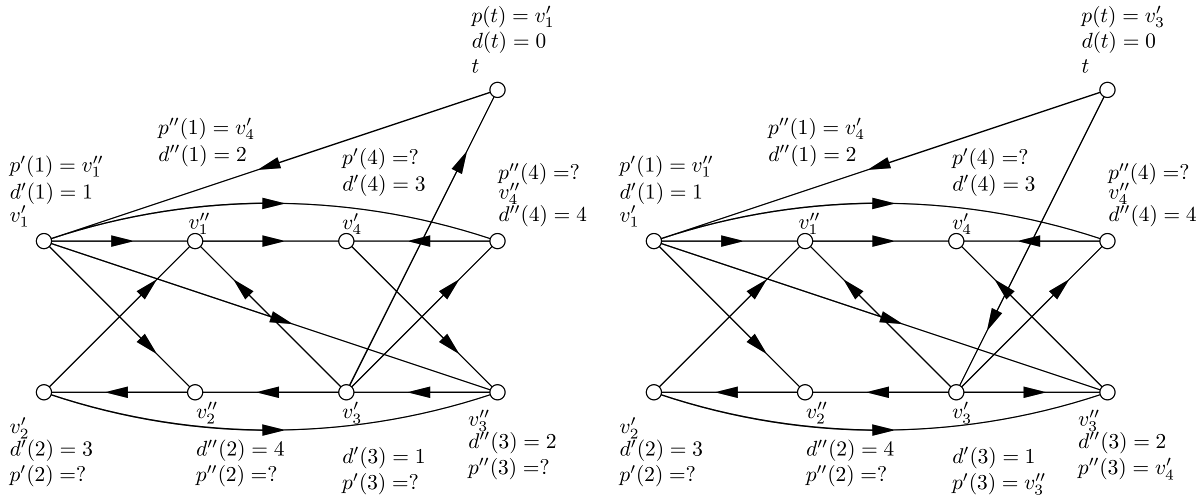
Example 4.3

Consider the graph G_{27} depicted in Figure 4.3(a). The vertices v_1' and v_3' are sinks that are joined to the supersink vertex t. The distance labels for each vertex are indicated on the graph. The predecessor variable for each vertex is currently unknown (indicated by a question mark). Say, for instance, the maximum flow from vertex v_4' (acting as the source s) to the supersink t is sought. In Algorithm 6 vertex u is initialised as vertex v_4' (line 4) after which control is passed to the while loop spanning lines 5 to 22.



(a) The digraph G_{27} before the first iteration.

(b) The digraph G_{27} after the first iteration of the while loop.



(c) The digraph G_{27} after the third iteration of the while loop.

(d) The digraph G_{27} after the sixth iteration of the while loop.

Figure 4.3: *Graphical representations of the steps taken by algorithm 6 to calculate the maximum flow from the source vertex v_4' to the two sink vertices v_1' and v_3' in the graph G_{27}.*

As $d(s) \leq p$, the while loop is executed. Vertex v_1'' is arbitrarily chosen as vertex v, as $d(v_1'') = d(v_3'') = 2$ (line 6). The if statement on line 7 evaluates to true; hence $p(v) = p(v_1'')$ is set to vertex v_4' and vertex u is set to $v = v_1''$ (lines 8 and 9). The changes are reflected in Figure 4.3(b). The if statement on line 10 is not executed, as $v = v_1'' \neq t$. Consequently, control is passed back to the start of the while loop spanning lines 5 to 22.

As $d(s) \leq p$, the while loop is executed. The only out-neighbour of vertex v_1'' is vertex v_1'; hence vertex v is initialised as vertex v_1'. The if statement on line 7 evaluates to true and the predecessor of vertex v_1' is set to vertex v_1'' (line 8). Vertex u is also set to vertex v_1''. The if statement on line 10 again evaluates to false and the while loop is repeated, this time extending the path P found to vertex t. Vertex v is thus set to vertex t, $p(t)$ is set to vertex v_1' and vertex u is set to v. As $v = t$, the if statement on line 10

Stellenbosch University http://scholar.sun.ac.za

evaluates to true. The path P from vertex t may now be traced back to vertex v'_4 (line 11) and the path is augmented (line 12). The vertex u is also reset as vertex s and control is passed back to the while loop spanning lines 5 to 22. These changes to the graph and the mentioned variables are reflected in Figure 4.3(c). Note that the arcs on the path found are turned around due to the augmentation of the path on lines 12 and 13.

The while loop spanning lines 5 to 22 now enters its fourth iteration. In a similar fashion to the steps described above, the path $P = v'_4, v''_3, v'_3, t$ is discovered through iterations 4 to 6 of the while loop spanning lines 5 to 22. The changes to the graph are reflected in Figure 4.3(d).

The while loop spanning lines 5 to 22 now enters its seventh iteration. As the vertex v'_4 has no out-neighbours, the if statement on line 7 evaluates to false and control is passed to line 17, where the distance label of vertex v'_4 is increased by one. Since $u = s$, the if statement spanning lines 18 to 20 is not executed. This process is repeated until $d(s) = d(v'_4) = p$, which is achieved in the thirteenth iteration of the while loop spanning lines 5 to 22, after which the algorithm terminates.

Two shortest paths v'_4, v''_1, v'_1, t and v'_4, v''_3, v'_3, t are thus found by the algorithm. ■

4.3.3 Converting a graph to a directed graph

Maximum flow algorithms, such as the Shortest Augmenting Path algorithm, require a directed graph as input. A special conversion is required to construct a directed graph G' from a graph G in such a way that it can be used later to compute internally disjoint paths in the graph G.

An undirected, weighted graph G of order p and size q may be converted to a directed graph G' as follows. Every vertex $v \in V(G)$ is replaced by two vertices v' and v'' in the vertex set $V(G')$, and is joined by an arc from vertex v' to vertex v'' that is included in the arc set $E(G')$ of the directed graph G'. This step inserts p arcs into the arc set $E(G')$. The vertex v' is referred to as the *in-vertex* and the vertex v'' is referred to as the *out-vertex* associated with $v \in V(G)$. When using Algorithm 6 to compute the maximum flow through a graph, this part of the construction of G' causes flow to occur in only one direction between the vertices v' and v'' (either from v' to v'', or from v'' to v'). This property is used by later algorithms to construct internally disjoint paths in the graph G'. For every other vertex $u \in V(G)$, if $uv \in E(G)$, then the arcs $u''v'$ and $v''u'$ are inserted into the arc set $E(G')$ of the directed graph G'. As each edge $uv \in E(G)$ is replaced by two arcs in the graph G', this step inserts $2q$ arcs into the arc set $E(G')$. Hence, the directed graph G' has order $2p$ and size $2q + p$.

It should be noted here that any path in the graph G' comprises an alternating sequence between in- and out-vertices. It is impossible to construct a path where two in- or out-vertices are adjacent, as no arc exists in a graph G' that joins two in- or two out-vertices. A further construction property of the graph G' is that it retains the property of being a simple graph. These concepts are used later to revert paths that are constructed in the graph G' to internally disjoint paths in the graph G.

The pseudocode for converting an undirected graph G to a directed graph G' is presented in Algorithm 7.

For the purpose of ℓ-connected subgraph generation, all arcs in the directed graph G' are required to have a weight of 1. It is clear that the worst-case running time of Algorithm 7 is $O(p^2)$, due to the two nested for loops spanning lines 4 to 11. The working of the algorithm is explained in the following example.

Example 4.4

Consider the undirected graph G_{28} depicted in Figure 4.4(a) with edge weights as indicated. Each vertex of the graph G_{28} is replaced by two adjacent vertices (an in- and an out-vertex) in the graph G'_{28} (lines 1 to 3 of Algorithm 7). For instance, vertex v_1 is replaced by the vertices v'_1 and v''_1 which are joined by the arc $v'_1v''_1$. Furthermore, if $uv \in E(G_{28})$, then the arcs $u''v'$ and $v''u'$ are inserted into the arc set $E(G'_{28})$ (lines 4 to 11 of Algorithm 7). For instance, the edge v_1v_4 in the graph G_{28} is replaced by the arcs $v''_1v'_4$ and $v''_4v'_1$ in the graph G'_{28}. The directed graph G'_{28}, obtained as output of Algorithm 7, is depicted in Figure 4.4(b). The adjacency matrices for the graphs G_{28} and G'_{28} are depicted in Figures 4.5(a) and (b) respectively. ■

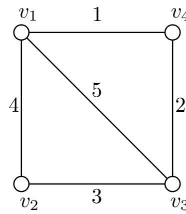
Stellenbosch University http://scholar.sun.ac.za

Algorithm 7 *Undirected Graph to Directed Graph*

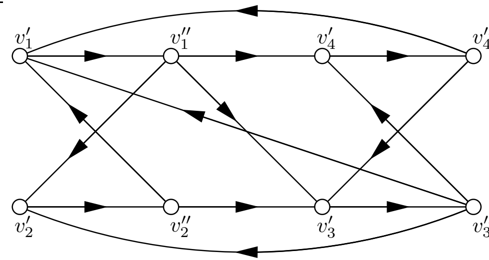
Input: Adjacency matrix D of an undirected graph G of order p and size q with vertex set $V(G)$ and edge set $E(G)$.

Output: Adjacency matrix D' of a directed graph G' of order $2p$ and size $2q + p$ with vertex set $V(G')$ and arc set $E(G')$. All arc weights are set to 1.

```
1:  for i ← 1 to p do
2:    D'(2i − 1, 2i) ← 1
3:  end for
4:  for i ← 1 to p do
5:    for j ← i + 1 to p do
6:      if D(i, j) > 0 then
7:        D'(2i, 2j − 1) ← 1
8:        D'(2j, 2i − 1) ← 1
9:      end if
10:   end for
11: end for
```



(a) The unidirected graph G_{28}.



(b) The digraph G_{28}'.

Figure 4.4: *Graphical representations of the steps in Algorithm 7 to convert the undirected graph G_{28} to the directed graph G_{28}'.*

4.3.4 Constructing internally disjoint paths for a directed graph

Let G be an undirected graph and G' be the directed graph obtained via Algorithm 7 with G as input. Let $P = \{P^{(1)}, P^{(2)}, \ldots, P^{(k)}\}$ be a set of paths between the vertices s and t obtained as output from Algorithm 6. It is required by later algorithms that the edges that make out these paths should be rearranged so as to obtain a set of k internally disjoint paths. This is always possible. A method for obtaining such a set of paths is now described.

Note that the maximum flow allowed through any vertex in the graph G' is limited to 1. This implies that the paths along which the actual flows in the graph take place are internally disjoint. However, the paths returned by Algorithm 6 are not necessarily internally disjoint due to the way in which the paths are constructed.

	v_1	v_2	v_3	v_4
v_1	0	4	5	1
v_2	4	0	3	0
v_3	5	3	0	2
v_4	1	0	2	0

(a) Adjacency matrix for the graph G_{28}.

	v_1'	v_1''	v_2'	v_2''	v_3'	v_3''	v_4'	v_4''
v_1'	0	1	0	0	0	0	0	0
v_1''	0	0	1	0	1	0	1	0
v_2'	0	0	0	1	0	0	0	0
v_2''	1	0	0	0	1	0	0	0
v_3'	0	0	0	0	0	1	0	0
v_3''	1	0	1	0	0	0	1	0
v_4'	0	0	0	0	0	0	0	1
v_4''	1	0	0	0	1	0	0	0

(b) Adjacency matrix for the graph G_{28}'.

Figure 4.5: *The adjacency matrices of the graphs G_{28} and G_{28}'.*

Stellenbosch University http://scholar.sun.ac.za

Every time a valid s-t path has been found, the arcs along the path are reversed due to the path augmentation in steps 12 and 13 of Algorithm 6. Such a reversed arc may now form part of another s-t path. Let a be such an arc, joining the vertices u and v (the direction of the arc becomes unimportant, as every path augmentation reverses its direction). If such an arc is traversed an even number of times, it implies that no net flow occurs along this arc — the flow of half of the paths employing it will cancel the flow of the other half. If such an arc is traversed an odd number of times (say $2i + 1$ times), then it would have been traversed i times in one direction and $i + 1$ times in the other, with the latter case indicating the direction of one unit of flow along the arc a in an optimal solution. The following example illustrates these concepts.

Example 4.5
Consider the undirected graph G_{29} depicted in Figure 4.6(a). Algorithm 7 may now be applied to produce the directed graph G'_{29}, as depicted in Figure 4.6(b). For this example, internally disjoint paths in the graph G'_{29} are sought from the vertex s'' to the vertex t'. Algorithm 6 is executed (starting from line 4, as the graph is already in the desired directed form) and the first path found is $P^{(1)} = s'', v'_1, v''_1, v'_2, v''_2, v'_6, v''_6, t'$. The path is augmented and the updated graph is depicted in Figure 4.6(c). The second path located is $P^{(2)} = s'', v'_4, v''_4, v'_5, v''_5, v'_6, v''_2, v'_2, v''_1, v'_8, v''_8, v'_9, v''_9, t'$ and is augmented (see Figure 4.6(d)). The third and final path to be constructed and augmented by Algorithm 6 is $P^{(3)} = s'', v'_7, v''_7, v'_2, v''_2, v'_3, v''_3, t'$ (see Figure 4.6(e)).

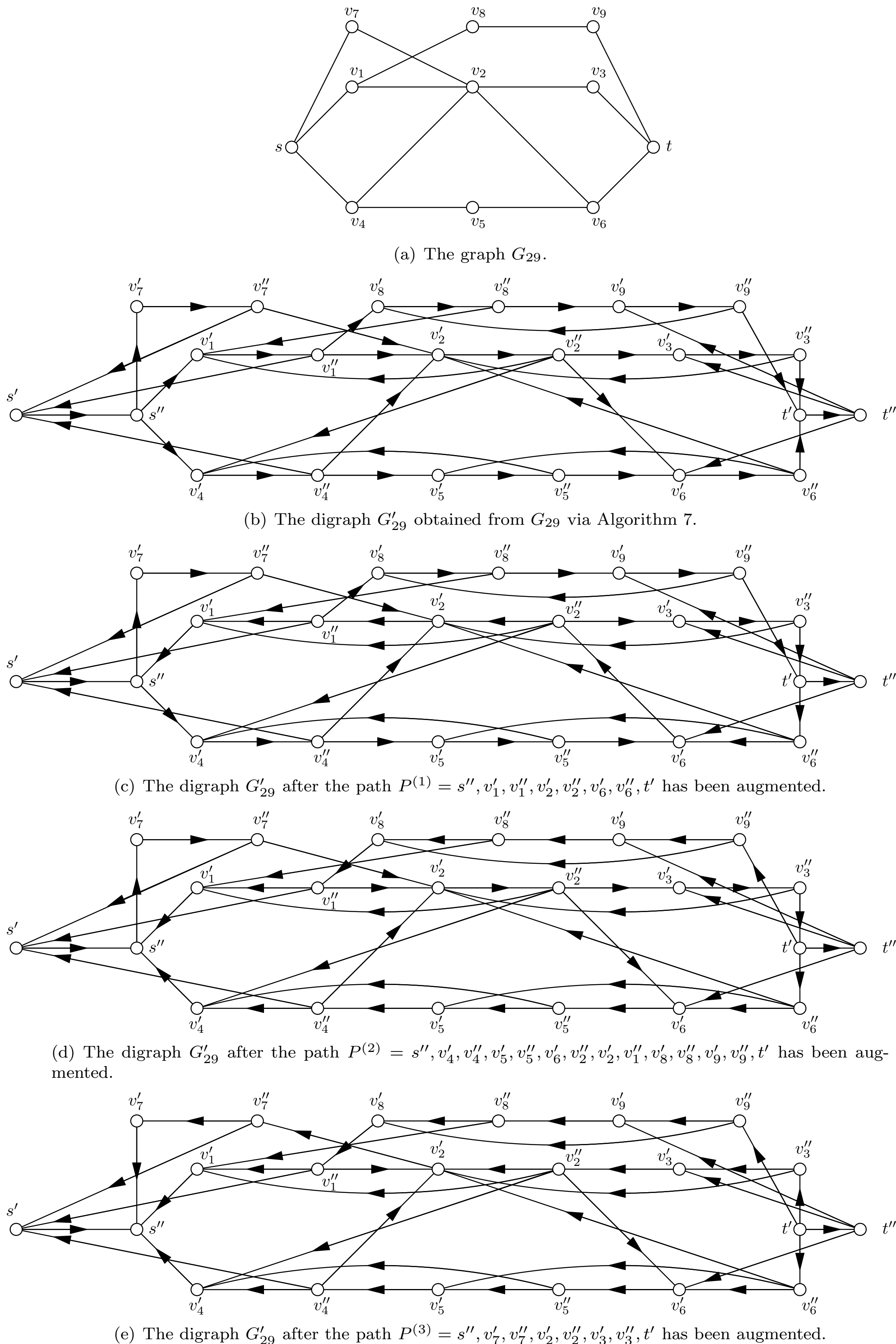
Three arcs were used more than once by the discovered paths — the arc joining the vertices v''_1 and v'_2 and the arc joining the vertices v''_2 and v'_6 were both traversed twice, namely by the paths $P^{(1)}$ and $P^{(2)}$. The net flow over these arcs are zero, as they were traversed an even number of times. The arc joining the vertices v'_2 and v''_2 was traversed once by all three discovered paths. This arc has a flow in the direction from vertex v'_2 to vertex v''_2. ■

An easy method for converting the paths returned by Algorithm 6 to internally disjoint paths is to keep track of how many times each arc has been traversed. An efficient structure for this may be achieved by means of a symmetric traversal matrix $\mathcal{A}$, in which element $(i, j) = (j, i)$ denotes the number of times the arc ij has been traversed in total (modulo 2). The matrix $\mathcal{A}$ thus consists only of ones and zeros. Internally disjoint paths may be obtained from the matrix as follows. A new path is created, starting from the vertex s''. The first non-zero element (say element j) in row s'' is taken as the next vertex on the path (the arc $s''j$ is inserted into the path). Elements $\mathcal{A}(s'', j)$ and $\mathcal{A}(j, s'')$ are removed from the matrix (set to zero) and the path is extended by searching in row j for the first non-zero element. The process of searching for the next vertex that is added to the path is repeated until the vertex t' is reached. If any non-zero elements remain in row s'', then the whole process is repeated to find another path (if k paths were returned by Algorithm 6, then this process will also repeat k times). The reason that this method guarantees a set of k internally disjoint paths is that all arcs causing paths to not be internally disjoint, have been removed from the matrix $\mathcal{A}$. This and the fact that each vertex has a maximum flow of 1 has the consequence that any vertex in the graph G' forms part of exactly one s-t path. Hence all paths created in this fashion are internally disjoint. The pseudocode for obtaining internally disjoint paths from the set of paths returned by Algorithm 6 is presented in Algorithm 8.

The matrix *arcCounter* in Algorithm 8 is the structure used as the symmetric traversal matrix. The nested for loops spanning lines 1 to 5 initializes all elements to zero. The nested for loops spanning lines 6 to 11 counts the number of times each arc is traversed (note that the direction of traversal is not important, hence the symmetry in the traversal matrix). The nested for loops spanning lines 12 to 17 ensure that all elements in *arcCounter* that are even are set to zero and that all elements that are odd are set to one. The for loop spanning lines 19 to 31 is responsible for determining the internally disjoint paths. This is explained with the aid of an example.

Example 4.6 (Continuation of Example 4.5)
Consider again the paths that were returned by Algorithm 6 for the graph G'_{29}. The symmetric traversal matrix *arcCounter* for this graph is depicted in Figure 4.7. All elements of *arcCounter* that are even are now set to zero and all elements that are odd are set to 1 (lines 12 to 17 of Algorithm 8). Thus the elements *arcCounter*(v''_1, v'_2), *arcCounter*(v'_2, v''_1), *arcCounter*(v''_2, v'_6) and *arcCounter*(v'_6, v''_2) are all set to zero, while the elements *arcCounter*(v'_2, v''_2) and *arcCounter*(v''_2, v'_2) are set to 1. Control is now passed

Stellenbosch University http://scholar.sun.ac.za



(a) The graph G_{29}.

(b) The digraph G'_{29} obtained from G_{29} via Algorithm 7.

(c) The digraph G'_{29} after the path $P^{(1)} = s'', v_1', v_1'', v_2', v_2'', v_6', v_6'', t'$ has been augmented.

(d) The digraph G'_{29} after the path $P^{(2)} = s'', v_4', v_4'', v_5', v_5'', v_6', v_2'', v_2', v_1'', v_8', v_8'', v_9', v_9'', t'$ has been augmented.

(e) The digraph G'_{29} after the path $P^{(3)} = s'', v_7', v_7'', v_2', v_2'', v_3', v_3'', t'$ has been augmented.

Figure 4.6: *The graphs G_{29} and G'_{29}. The path augmentation steps of Algorithm 6 are shown in Figures 4.6(c) and (e).*

Stellenbosch University http://scholar.sun.ac.za

Algorithm 8 *Constructing Internally Disjoint Paths in a Directed Graph*

Input: Adjacency matrix D' of a directed graph G' of order p and P, the set of k paths corresponding to the flows obtained by implementing Algorithm 5 with G' as input.
Output: The set P' of k internally disjoint paths for the directed graph G'.

1: **for** $i \leftarrow 1$ to $2p$ **do**
2: **for** $j \leftarrow 1$ to $2p$ **do**
3: $arcCounter(i, j) \leftarrow 0$
4: **end for**
5: **end for**
6: **for** $i \leftarrow 1$ to k **do**
7: **for** $j \leftarrow 1$ to (length of path i in the set P) $- 1$ **do**
8: $arcCounter$(vertex j on path i, vertex $j+1$ on path i) $\leftarrow$ $arcCounter$(vertex j on path i, vertex $j+1$ on path i) $+ 1$
9: $arcCounter$(vertex $j+1$ on path i, vertex j on path i) $\leftarrow$ $arcCounter$(vertex $j+1$ on path i, vertex j on path i) $+ 1$
10: **end for**
11: **end for**
12: **for** $i \leftarrow 1$ to p **do**
13: **for** $j \leftarrow i+1$ to p **do**
14: $arcCounter(i, j) \leftarrow arcCounter(i, j) \pmod 2$
15: $arcCounter(j, i) \leftarrow arcCounter(j, i) \pmod 2$
16: **end for**
17: **end for**
18: $j \leftarrow 1$
19: **for** every index i where $arcCounter(s, i) > 0$ **do**
20: $a \leftarrow s$
21: $b \leftarrow i$
22: **while** $b \neq t$ **do**
23: add arc ab to path $P'^{(j)}$
24: $arcCounter(a, b) \leftarrow 0$
25: $arcCounter(b, a) \leftarrow 0$
26: $a \leftarrow b$
27: $b \leftarrow$ index of first non-zero element in row b
28: **end while**
29: add arc ab to path $P'^{(j)}$ [last vertex in path is added here]
30: $j \leftarrow j + 1$
31: **end for**

to the for loop spanning lines 19 to 31, where the first new path is constructed. As $arcCounter(s'', v'_1)$ equals one, this arc is inserted into the path $P'^{(1)}$ (line 23). The elements $arcCounter(s'', v'_1)$ and $arcCounter(v'_1, s'')$ are both set to zero (lines 24 and 25), after which row v'_1 is investigated. The first non-zero element is vertex v''_1, hence the path $P'^{(1)}$ is extended to include vertex v''_1, after which the arcs $v'_1v''_1$ and $v''_1v'_1$ are removed from the matrix $arcCounter$. Row v''_1 is investigated next. The first non-zero element is vertex v'_8 and the above process is repeated. This process continues until the path $P'^{(1)} = s'', v'_1, v''_1, v'_8, v''_8, v'_9, v''_9, t'$ is constructed. Control is again passed to the start of the for loop spanning lines 19 to 31. In a similar fasion the paths $P'^{(2)} = s'', v'_4, v''_4, v'_5, v''_5, v'_6, v''_6, t'$ and $P^{(3)} = s'', v'_7, v''_7, v'_2, v''_2, v'_3, v''_3, t'$ are constructed. Note that all three paths are internally disjoint. ■

The sets of nested for loops spanning lines 1 to 5 and 12 to 17 in Algorithm 8 both have a worst-case running time of $O\left(p^2\right)$. The length of a path discovered can be at most p vertices. Hence the sets of for loops spanning lines 6 to 11 and 19 to 31 in Algorithm 8 both have a worst-case running time of $O(kp)$, where k is the number of paths discovered. Hence, the worst-case running time of this algorithm is $O\left(\max\left\{p^2, kp\right\}\right)$. The number of paths discovered cannot be greater than $2q + p$, as there are only $2q + p$ arcs in the graph G', and each arc may only be traversed once. Consequently, the worst-case

Stellenbosch University https://scholar.sun.ac.za

	s'	s''	v_1'	v_1''	v_2'	v_2''	v_3'	v_3''	v_4'	v_4''	v_5'	v_5''	v_6'	v_6''	v_7'	v_7''	v_8'	v_8''	v_9'	v_9''	t'	t''
s'	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
s''	0	0	1	0	0	0	0	0	1	0	0	0	0	0	1	0	0	0	0	0	0	0
v_1'	0	1	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
v_1''	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0
v_2'	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0
v_2''	0	0	0	0	1	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
v_3'	0	0	0	0	0	1	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0
v_3''	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0
v_4'	0	1	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0
v_4''	0	0	0	0	0	0	0	0	1	0	1	0	0	0	0	0	0	0	0	0	0	0
v_5'	0	0	0	0	0	0	0	0	0	1	0	1	0	0	0	0	0	0	0	0	0	0
v_5''	0	0	0	0	0	0	0	0	0	0	1	0	1	0	0	0	0	0	0	0	0	0
v_6'	0	0	0	0	0	0	0	0	0	0	0	1	0	1	0	0	0	0	0	0	0	0
v_6''	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	1	0
v_7'	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0
v_7''	0	0	0	0	1	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0
v_8'	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0
v_8''	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	1	0	0	0
v_9'	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	1	0	0
v_9''	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	1	0
t'	0	0	0	0	0	0	0	1	0	0	0	0	0	1	0	0	0	0	0	1	0	0
t''	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Figure 4.7: *The symmetric traversal matrix arcCounter for the graph G'_{29}.*

running time of Algorithm 8 is $O\left(\max\left\{p^2, (2q+p)p\right\}\right) = O\left((2q+p)p\right) = O\left(pq\right)$.

4.3.5 Putting it all together

In this section, the algorithms discussed in §4.3.1 - §4.3.4 are combined to produce a method for finding a maximum set of internally disjoint paths between a source vertex s and a sink or set of sink vertices in an undirected graph G. The steps for obtaining these internally disjoint paths in such a graph G are outlined in Algorithm 9.

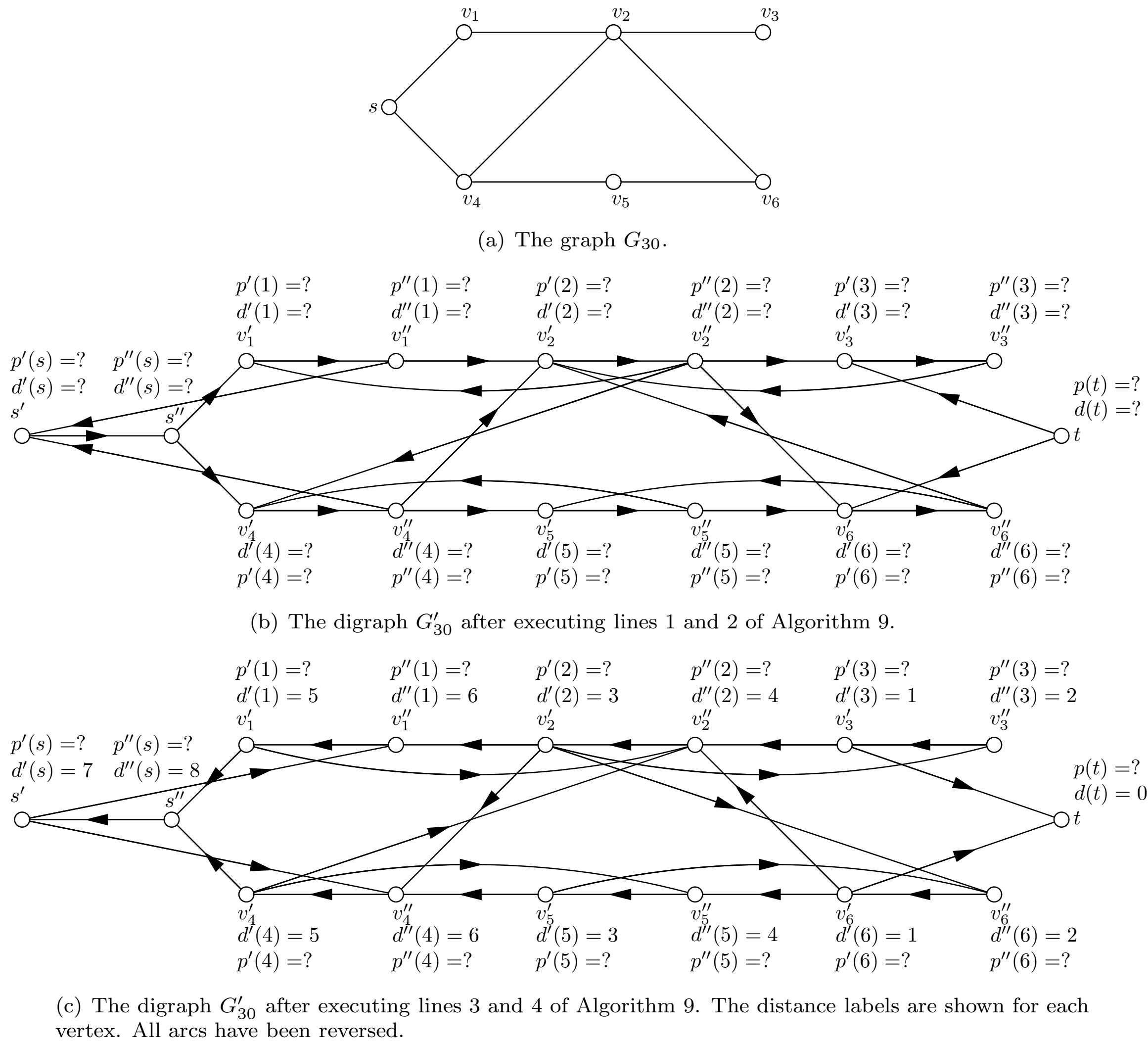
Algorithm 9 *Finding Internally Disjoint Paths in an Undirected Graph*

Input: Adjacency matrix D of an undirected graph G, a source vertex s and a set of sink vertices T.
Output: A maximum set of internally disjoint paths P between vertex s and the set T in the graph G.

1: Convert the graph G to a directed graph G', using Algorithm 7. Add a supersink vertex t to the graph G'. Add arcs to the graph G' from the supersink vertex t to every in-vertex representing a sink vertex in the graph G.
2: Calculate the distance labels for each vertex using Algorithm 5, starting from the supersink vertex t.
3: Reverse all arcs of the graph G'.
4: Append the counter on line 5 of Algorithm 6 such that the while loop will execute as long as $d(s) \leq 2p$, where p is the order of the directed graph G'. Use this version of Algorithm 6, starting from line 4, to calculate the maximum flow from the in-vertex s' in the graph G', representing vertex s of the graph G, to the vertex t.
5: Use Algorithm 8 to convert the set of paths P returned by Algorithm 6 to internally disjoint paths. Call this set of paths P'.
6: Remove the supersink vertex t from the ends of all paths in the set P'.
7: Convert the paths in the set P' to internally disjoint paths for the graph G.

Steps 1 to 5 of Algorithm 9 have already been discussed in the previous sections. The structure of the directed graph obtained using Algorithm 9 differs somewhat from the graph examples presented in the previous sections. This is primarily due to the way in which a supersink is joined to the directed graph.

Stellenbosch University http://scholar.sun.ac.za



(a) The graph G_{30}.

(b) The digraph G'_{30} after executing lines 1 and 2 of Algorithm 9.

(c) The digraph G'_{30} after executing lines 3 and 4 of Algorithm 9. The distance labels are shown for each vertex. All arcs have been reversed.

Figure 4.8: *The graph G_{30} and graphical representation of steps 1 to 2 (Figure 4.8(b)) and 3 to 4 (Figure 4.8(c)) of Algorithm 9.*

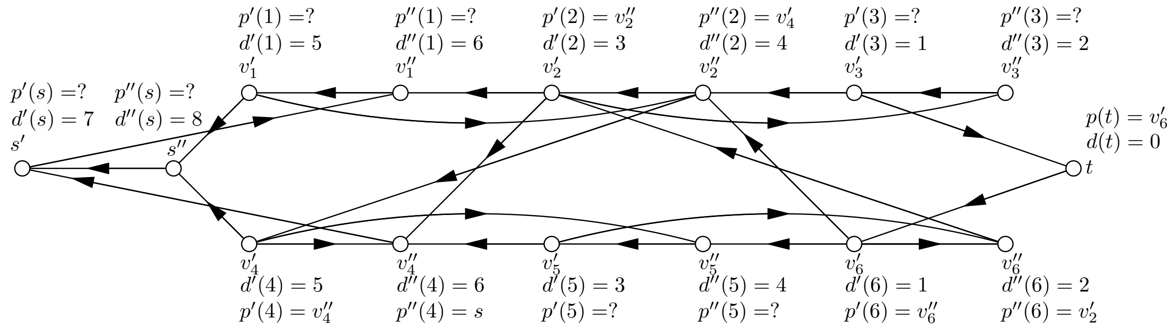
Also note that the enumerating progress of the counter for the while loop on line 5 of Algorithm 6 is changed to extend to $2p$ instead of p, as reflected in step 4 of Algorithm 9. This is required, as the arcs in the graph G' are reversed after the distance labels have been calculated. This may result in some distance labels to reaching the bound p before a complete path has been found, causing Algorithm 6 to terminate prematurely. The problem is alleviated by allowing the counter in the while loop on line 5 of algorithm 6 to continue incrementing its value until a bound of $2p$ is reached. Note that the distance labels are used to test for loops in paths. As the graph G' consists of p vertices, setting this counter to $2p$ ensures that all vertices can be visited along some path before Algorithm 6 will terminate.

A complete worked example, implementing steps 1 to 5 of Algorithm 9 now follows.

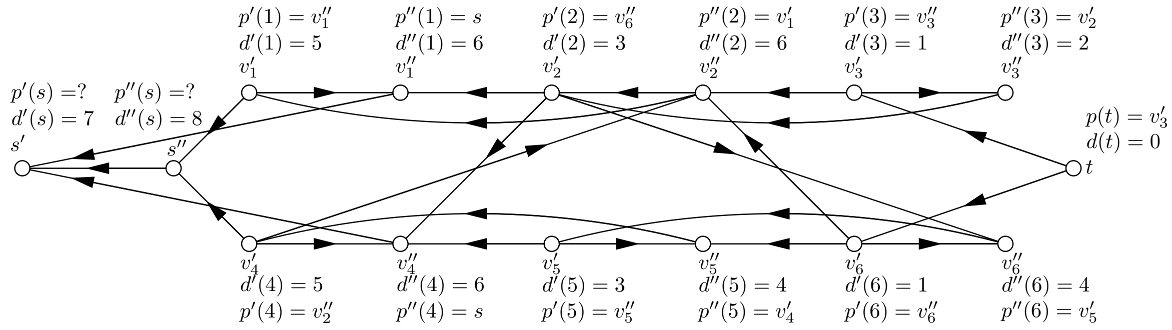
Example 4.7

Consider the undirected graph G_{30} depicted in Figure 4.8(a). Vertex s is the source and vertices v_3 and v_6 are the sinks. Step 1 of Algorithm 9 may be executed to produce the directed graph G'_{30} depicted in Figure 4.8(b). The distance labels are calculated and all arcs are reversed (steps 2 and 3 — see Figure 4.8(c)). Control is now passed to step 4 of Algorithm 9. Let $P^{(1)} = s', v_4'', v_4', v_2'', v_2', v_6'', v_6', t$ be the first path obtained using Algorithm 6 (at this stage, Algorithm 6 has not yet terminated). The updated graph G'_{30} is depicted in Figure 4.9(a). Note that the arcs on the discovered path have been reversed. Algorithm 6 now continues to execute and obtains the path $P^{(2)} = s, v_1'', v_1', v_2'', v_4', v_5'', v_5', v_6'', v_2', v_3'', v_3', t$ and then terminates. The updated graph G'_{30} is depicted in Figure 4.9(b). As the maximum flow has

Stellenbosch University http://scholar.sun.ac.za



(a) The digraph G'_{30} after the path $P^{(1)} = s', v''_4, v'_4, v''_2, v'_2, v''_6, v'_6, t$ has been found and augmented.



(b) The digraph G'_{30} after the path $P^{(2)} = s, v''_1, v'_1, v''_2, v'_4, v''_5, v'_5, v''_6, v'_2, v''_3, v'_3, t$ has been found and augmented.

Figure 4.9: *Graphical representations of the process of obtaining internally disjoint paths from the vertex s to the vertex t in the graph G_{30}.*

been calculated, control is passed to step 5 of Algorithm 9. Note that the arc joining the vertices v'_4 and v''_2 and the arc joining the vertices v'_2 and v''_6 have been traversed by both discovered paths. This can be seen from the symmetric traversal matrix *arcCounter* after the nested for loop spanning lines 6 to 11 in Algorithm 8 completed execution ($arcCounter(v'_4, v''_2) = arcCounter(v'_2, v''_6) = 2$). These arcs are removed from the matrix *arcCounter* after which the two internally disjoint paths $P^{(1)} = s', v''_1, v'_1, v''_2, v'_2, v''_3, v'_3, t$ and $P^{(2)} = s', v''_4, v'_4, v''_5, v'_5, v''_6, v'_6, t$ are calculated in the for loop spanning lines 24 to 36 of Algorithm 8. ■

It remains to show how a set of paths found for the directed graph G' of an undirected graph G are converted to internally disjoint paths in the graph G (step 7 of Algorithm 9). The pseudocode for this operation is presented in Algorithm 10 and illustrated in Example 4.8.

Algorithm 10 *Directed Paths to Undirected Paths*

Input: A set P' of k paths that do not share any arcs in the directed graph G' (of order $2p$ and size $2p + q$) obtained by applying steps 1 to 6 of Algorithm 9 to the graph G (of order p and size q).

Output: A set P of k internally disjoint paths in the undirected graph G.

1: **for** $i \leftarrow 1$ to k **do**
2:   **for** $j \leftarrow 1$ to $2p$ **do**
3:     **if** vertex j on path i in the set P' is an in-vertex **then**
4:       Let u be the vertex in the graph G represented by the jth vertex on path i of the set P'. Also, let v be the vertex in the graph G represented by the $(j + 1)^{\text{th}}$ vertex on path i of the set P'. Add the edge uv to path i of the set P.
5:     **end if**
6:   **end for**
7: **end for**

Stellenbosch University http://scholar.sun.ac.za

Example 4.8 (Continuation of Example 4.7)
Consider again the graph G_{30} in Figure 4.8(a) and its corresponding directed graph G'_{30} in Figure 4.8(c). Two paths that do not share any arcs in G'_{30} were constructed, namely $P'^{(1)} = s', v''_1, v'_1, v''_2, v'_2, v''_3, v'_3, t$ and $P'^{(2)} = s', v''_4, v'_4, v''_5, v'_5, v''_6, v'_6, t$. Control is now passed to step 6 of Algorithm 9 where the vertex t is removed from the paths in the set P'. Hence $P'^{(1)} = s', v''_1, v'_1, v''_2, v'_2, v''_3, v'_3$ and $P'^{(2)} = s', v''_4, v'_4, v''_5, v'_5, v''_6, v'_6$. Step 7 is now executed; hence control is passed to Algorithm 10. For this example the path $P'^{(1)}$ is converted to a path in the graph G_{30}.

In the first iteration of the for loop spanning lines 2 to 6, $j = 1$. Hence vertex j points to vertex s', the first vertex in the path $P'^{(1)}$. As vertex s' is an in-vertex and corresponds to vertex s in the graph G_{30}, and as vertex v''_4 corresponds to vertex v_4 of graph G_{30}, the edge sv_4 is added to the empty path $P^{(1)}$. The for loop spanning lines 2 to 6 now iterates until $j = 3$. The third vertex, vertex v'_4 is an in-vertex, thus the edge v_4v_5 is added to the path $P^{(1)}$, such that $P^{(1)} = s, v_4, v_5$. In a similar fashion, the path is extended to vertex v_6, resulting in the path $P^{(1)} = s, v_4, v_5, v_6$. The for loop spanning lines 2 to 6 now terminates as the end of the path has been reached. In a similar fashion, the path $P^{(2)} = s, v_1, v_2, v_3$ is constructed. ■

Step 1 of Algorithm 9 calls Algorithm 7, which has an $O(p^2)$ worst-case running time. Ford's Algorithm is called in Step 2, which has a worst-case running time of $O(pq)$. As there are $2q + p$ arcs to reverse in step 3, this step has a worst-case running time of $O(2q + p)$. Algorithm 6 is called in step 4 and has a worst-case running time of $O(pq^2)$. The worst-case running time of Algorithm 8, implemented in step 5 of Algorithm 9, has a worst-case running time of $O(pq)$. Finally, the worst-case running time of Algorithm 10 is $O(kp)$ due to the nested for loop spanning lines 1 to 7, where k is the number of paths found in the graph G. As the size of graph G is q, at most q internally disjoint paths can be found. Hence the worst-case running time of Algorithm 10 is $O(pq)$. The worst-case running time for Algorithm 9 is thus dominated by that of Algorithm 6. Consequently, its worst-case running time is $O(pq^2)$.

4.4 Implementation of Whitney's Theorem

One of the most important results in the field of connectivity is due to Whitney (see Theorem 3.7), which states that, for a graph G to be k-connected, there must exist at least k internally disjoint paths between any pair of vertices. The next result, called Algorithm Whitney, is an implementation of this theorem and produces a spanning subgraph G' from a graph G with a connectivity number of k. The minimum connectivity number of the subgraph G' may be varied from zero to k, depending on the number of internally disjoint paths that are retained between each pair of vertices. Note that, from the definition of k-connectivity, this implies that if ℓ internally disjoint paths are present between every pair of vertices in the subgraph G', then the connectivity number of the subgraph is at least ℓ. The pseudocode for this algorithm is presented in Algorithm 11.

Steps 4 to 7 of Algorithm 11 have already been described in previous sections. In step 8, vertices $2j$ and $2j - 1$ are added to each path that was generated in step 7 of Algorithm 11. This is necessary, as the working of Algorithm 6 is such that the last vertex in the path is not included when the paths are constructed, as some input graphs end in a supersink vertex t, that should not form part of the paths being constructed (see, for instance, Example 4.7). For Whitney's Algorithm, no supersink vertex is added to the graph and the last vertex, vertex $2j$, is not included, but is inserted in step 8 of Algorithm 11. The next vertex, vertex $2j - 1$, also has to be inserted in order for Algorithm 10, which is called in step 9 of Algorithm 11, to add vertex j to the paths constructed in the graph G'. Step 10 of Algorithm 11 attempts to optimise the output obtained from this algorithm by selecting the ℓ cheapest paths from the set P' and inserting these paths into the graph G'. This is done in an attempt to minimise the overall cost of the graph H.

Example 4.9
Consider the complete, weighted graph G_{31} depicted in Figure 4.10(a). Suppose that a 3-connected spanning subgraph is desired. Algorithm 11 may be implemented to produce the spanning subgraph G'_{31}, depicted in Figure 4.10(b). The paths inserted into the graph G'_{31} are listed in Table 4.3.

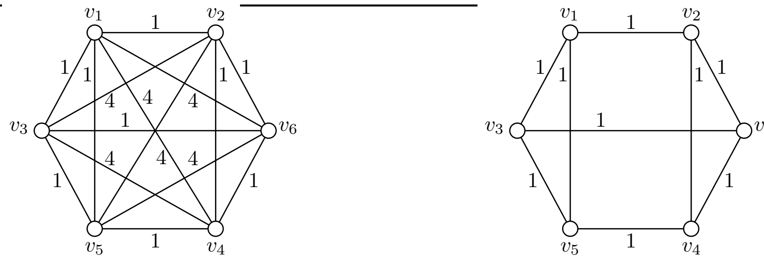
Stellenbosch University http://scholar.sun.ac.za

Algorithm 11 *Whitney's Algorithm*

Input: A graph G of order p, with $\kappa(G) = k$, and ℓ, the desired minimum connectivity number for the spanning subgraph G'.

Output: A spanning subgraph G', for which $\kappa(G') \geq \ell$.

1: Convert the graph G to a directed graph H, using Algorithm 7.
2: **for** $i \leftarrow 1$ to p **do**
3:   **for** $j \leftarrow i + 1$ to p **do**
4:     Calculate the distance labels for each vertex using Ford's Algorithm, starting from the vertex $2j$.
5:     Reverse all arcs of the graph H.
6:     Call Algorithm 6, starting from line 4, to calculate the maximum flow from the in-vertex $2i - 1$ to the out-vertex $2j$ in the graph H.
7:     Call Algorithm 8 to convert the set of paths P returned by Algorithm 6 to internally disjoint paths. Call this set of paths P'.
8:     Add the vertex $2j$, followed by the vertex $2j - 1$, to the end of each path in P'.
9:     Convert the paths in the set P' to internally disjoint paths for the graph G using Algorithm 10.
10:     Calculate the cost of each path in P' by summating over the weights of the edges on each path. Select the ℓ cheapest paths and insert them into the graph G'.
11:   **end for**
12: **end for**



(a) The graph G_{31}, with a total weight of 33. (b) The graph G'_{31}, with a total weight of 9.

Figure 4.10: *Graphical representations of the graphs G_{31} and G'_{31}. The graph G_{31} has a connectivity number of 5 and a weight of 33, whilst the graph G'_{31} has a connectivity number of 3 and a cost of 9.*

	Paths to				
Paths from	v_2	v_3	v_4	v_5	v_6
v_1	v_1, v_2 v_1, v_5, v_2 v_1, v_4, v_3, v_2	v_1, v_2, v_3 v_1, v_4, v_3 v_1, v_5, v_6, v_3	v_1, v_4 v_1, v_2, v_3, v_4 v_1, v_5, v_6, v_4	v_1, v_5 v_1, v_2, v_5 v_1, v_4, v_6, v_5	v_1, v_5, v_6 v_1, v_4, v_6 v_1, v_2, v_3, v_6
v_2	—	v_2, v_3 v_2, v_1, v_4, v_3 v_2, v_5, v_6, v_3	v_2, v_3, v_4 v_2, v_1, v_4 v_2, v_5, v_6, v_4	v_2, v_5 v_2, v_1, v_5 v_2, v_3, v_6, v_5	v_2, v_3, v_6 v_2, v_5, v_6 v_2, v_1, v_4, v_6
v_3	—	—	v_3, v_4 v_3, v_6, v_4 v_3, v_2, v_1, v_4	v_3, v_6, v_5 v_3, v_2, v_5 v_3, v_4, v_1, v_5	v_3, v_6 v_3, v_4, v_6 v_3, v_2, v_5, v_6
v_4	—	—	—	v_4, v_6, v_5 v_4, v_1, v_5 v_4, v_3, v_2, v_5	v_4, v_6 v_4, v_3, v_6 v_4, v_1, v_5, v_6
v_5	—	—	—	—	v_5, v_6 v_5, v_1, v_4, v_6 v_5, v_2, v_3, v_6

Table 4.3: *The list of paths used in the construction of the graph G'_{31} so as to ensure 3-connectivity of G'_{31}.*

Stellenbosch University http://scholar.sun.ac.za

The total weight of the graph G_{31} is 33 and that of G'_{31} is 9. Hence a total weight improvement of 24 was possible. ■

A disadvantage of Algorithm 11 is that it will not always produce a spanning subgraph with fewer edges than the original graph. This occurs if every edge joining a certain pair of vertices (say u and v) is selected as one of the ℓ cheapest u-v paths that are constructed in steps 6 to 10 of Algorithm 11. In this manner, all edges in the original graph are used. For many practical implementations of graphs, an edge joining two vertices indicates the shortest path between these two vertices. The algorithm will not be able to return a subgraph comprising fewer edges for all such graphs.

However, Algorithm 11 has a relatively low running time. Step 1 of Algorithm 11 has a worst-case running time of $O\left(p^2\right)$. Ford's Algorithm is called in Step 4, which has a worst-case running time of $O(pq)$. There are $2p+q$ edges to reverse; hence step 5 has a worst-case running time of $O(2q+p)$. Algorithm 6, called in step 6, has a worst-case running time of $O(pq^2)$. The worst-case running time of Algorithm 8, which is called in step 7 of Algorithm 11, has a worst-case running time of $O\left(pq\right)$. The vertices $2j$ and $2j-1$ are added to all paths found in step 8 of Algorithm 11. As the size of the graph G is q, at most q internally disjoint i-j paths can be found; hence the worst-case running time of this step takes $O(q)$ time. Similar to the worst-case running time analysis of Algorithm 9, the worst-case running time of Algorithm 10, called in step 9 of Algorithm 11, is $O\left(pq\right)$. The calculation of the costs for each path on line 10 has a worst-case running time of $O\left(q^2\right)$, as there exists a maximum number of q paths generated between every pair of vertices, each having a maximum length of q. If the well-known Quicksort algorithm is used to sort the paths according to their cost, this step takes a further $O\left(\log\left(q^2\right)\right)$ time; hence the worst-case running time of this step is $O\left(q^2\right)$. The worst-case running time of all of these steps are dominated by that of Algorithm 6. The nested for loop spanning lines 2 to 11 repeats all these steps $O\left(p^2\right)$ times. Consequently, the worst-case running time of Algorithm 11 is $O\left(p^3q^2\right)$.

4.5 Removing the most expensive edge first

In this section a number of brute force algorithms for the construction of connectivity preserving and reducing spanning subgraph of a weighted graph are presented. These algorithms are based on the greedy approach of removing the most expensive edge, testing what the connectivity number of the resulting graph is, and repeating the process, if required. The algorithms are referred to as *Most Expensive Edge First*, or *MEEF* for short.

4.5.1 Construction of a spanning subgraph G' of G, with $\kappa(G') = \kappa(G)$

Let G be a connected, weighted graph with connectivity number k. Algorithm *MEEF: Connectivity Preserving*, as presented in pseudocode as Algorithm 12, repeatedly removes a most expensive edge from G until the removal of the next most expensive edge reduces the connectivity number of the graph G. This requires the recalculation of the connectivity number of the graph G' every time an edge is removed.

The majority of the steps in Algorithm 12 are self explanatory. The while loop spanning lines 7 to 13 removes a most expensive edge $uv \in E(G')$ and adds the cost of uv to the variable *costSaved*. The connectivity number of the graph G' is then recalculated in line 12 by making use of either Algorithm 3 or 4. The while loop terminates once $\kappa(G') < k$. At this point the last edge that was removed, is re-inserted into $E(G')$ and its cost is subtracted from *costSaved* (lines 14 to 15).

Let q and q' be the sizes of the graphs G and G' respectively. The while loop spanning lines 7 to 13 completes $(q-q'+1)$ iterations. To find a most expensive edge $uv \in E(G')$ takes constant time ($O(1)$) if the edges are sorted and a counter is used to point to the next most expensive edge. Sorting may be achieved prior to entering the while loop. If the well-known Quicksort algorithm is used, the sorting operation takes $O(q \log q)$ time, where q is the size of the graph G. The worst-case running time of each iteration of the while loop is thus dominated by the worst-case running time of the algorithm used to compute $\kappa(G')$ on line 12. If Algorithm 3 is used for this purpose, then the worst-case running time of the while loop, and consequently that of Algorithm 12, is $O\left(q-q'\right) \times O\left(p^3q^2\right) = O\left(p^3(q-q')q^2\right) = O\left(p^3q^3\right)$.

Stellenbosch University http://scholar.sun.ac.za

Algorithm 12 MEEF: Connectivity Preserving

Input: A connected graph G with connectivity number k.
Output: A connectivity preserving spanning subgraph G' of the graph G and the cost saved.

1: **if** the graph G is not connected **then**
2: print "Graph is not connected.", stop.
3: **end if**
4: $G' \leftarrow G$
5: $costSaved \leftarrow 0$
6: $tmpConn \leftarrow \kappa(G')$
7: **while** $(tmpConn \geq k)$ **do**
8: find most expensive edge $uv \in E(G')$
9: $Etmp \leftarrow E(G')$
10: $costSaved \leftarrow costSaved + \text{cost}(uv)$
11: $E(G') \leftarrow E(G') \setminus uv$
12: $tmpConn \leftarrow \kappa(G')$
13: **end while**
14: $costSaved \leftarrow costSaved - \text{cost}(uv)$
15: $E(G') \leftarrow Etmp$
16: return G', $costSaved$

The advantage of this algorithm is that it is easy to implement and has a low worst-case running time. There is a disadvantage, however, as is seen in the following example.

Example 4.10
Consider the graph G_{32} with edge-weights as depicted in Figure 4.11. By inspection it follows that $\kappa(G_{32}) = 2$.

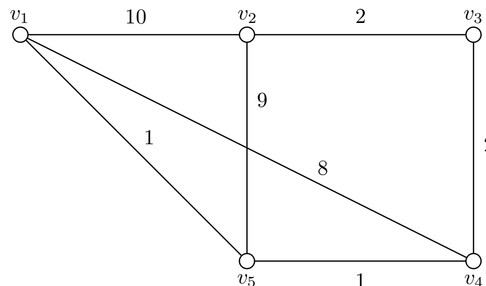


Figure 4.11: *The graph G_{32}.*

If Algorithm 12 is applied to G_{32}, it attempts to remove the edge v_1v_2 first. The connectivity number of the graph still equals two and hence the while loop in the algorithm is repeated. It now attempts to remove the edge v_2v_5. As this produces a spanning subgraph with a connectivity number of the graph to one, the edge is replaced and the algorithm terminates, with a total saved cost of 10. However, if the edges v_1v_4 and v_2v_5 are removed instead, the connectivity number remains two for the new subgraph, with a total saved cost of 17. ■

The previous example illustrates that the way in which expensive edges are scattered throughout the graph has an important effect on the effectiveness of Algorithm 12. More edges will, on average, be removed from graphs where edges with high costs are dispersed more or less evenly throughout the graph. If edges with high costs are not dispersed evenly, fewer edges may be removed on average, resulting in a lower saved cost.

4.5.2 Construction of a spanning subgraph G' of G, with $\kappa(G') < \kappa(G)$

To construct a connectivity reducing spanning subgraph G' with a connectivity number strictly less than that of the original graph G, say with $\kappa(G') = \ell < \kappa(G) = k$, Algorithm 12 may be implemented by

adapting the while loop spanning lines 7 to 13 to terminate only once $\kappa(G') < \ell$. Though this process will work, it makes an unnecessary number of calls to an algorithm for calculating $\kappa(G')$ in each iteration of the while loop. A faster method, based on the same principles as those of Algorithm 12 may be established. However, a number of preliminary results are first required.

Theorem 4.1 *Let G be a graph for which $\kappa(G) = k$. Let $F = \langle E(G) \cup uv \rangle_G$, u, $v \in V(G)$, $uv \notin E(G)$. Then $\kappa(F) \leq k + 1$.*
Proof Let U be a minimum cut-set of the graph G and construct the graph $H = \langle V(G) \setminus U \rangle_G$. It is clear that the graph H is disconnected. Three cases result.

Case 1: Consider the case where $k(H) > 2$, with W_i as the ith component of the graph H. A graphical representation of this case may be found in Figure 4.12(a). As no edge can join more than two components, it follows that $k(\langle E(H) \cup uv \rangle) \geq 2$. Hence $\kappa(F) = k$.

Now consider the case where $k(H) = 2$. A further two cases result.

Case 2: Consider the case where the vertices u and v are contained within the same component (say W_1) of the graph H. A graphical representation of this case may be found in Figure 4.12(b). The inclusion of the edge uv in the graph H does not increase its connectivity number. Hence, $\kappa(G) = \kappa(F) = k$.

Case 3: $|W_1| \geq 1$ and $|W_2| \geq 1$. Let the vertices u and v be contained in the components W_1 and W_2 of the graph H, with $u \in W_1$ and $v \in W_2$. Two subcases result. Firstly, consider the case where U is the only minimum cut-set of the graph G. A graphical representation of this case may be found in Figure 4.12(c). As the edge uv joins the two components of the graph H, it follows that $U \cup v$ is a minimum cut-set of the graph F and hence $\kappa(F) = k + 1$. Secondly, consider the case where the graph G posesses more than one minimum cut-set, say another cut-set U' exists. Let W_1' and W_2' be the two components of the graph $H' = \langle V(G) \setminus U' \rangle_G$. If the vertices u and v are contained within the same component, it follows from *Case 2* that $\kappa(G) = \kappa(F) = k$. Similarly, if either vertex u or v is contained within U', or both are contained within U', it follows that $\kappa(F) = k$, as the two components W_1' and W_2' remain disjoint. This case is represented in Figure 4.12(d). If, however, $u \in W_1'$ and $v \in W_2'$ (without loss of generality) for any minimum cut-set U' for the graph G, it follows that $\kappa(F) = k+1$, as the components W_1' and W_2' are joined by the edge uv. This case is represented in Figure 4.12(e). ■

Note that, if the graph F is taken as the original graph and G as the graph obtained by removing the edge uv, then Theorem 4.1 states that the graph obtained by the removal of an edge from a connected graph will have a connectivity number of at most one fewer than that of the original graph.

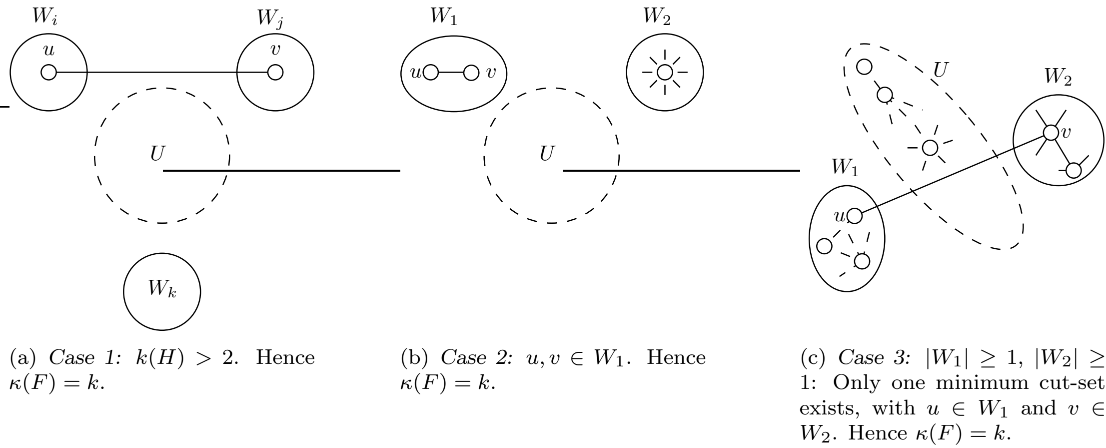
Proposition 4.1 *Let G be a graph for which $\kappa(G) = k$ and let E_X be any subset of edges of $E(G)$. Then $\kappa(G - E_X) \geq \kappa(G) - |E_X|$.*
Proof Consider the graph $H = G - E_X$. It follows from Theorem 4.1 that $\kappa(H \cup \langle E_X \rangle_G) \leq \kappa(H) + |E_X|$. Hence $\kappa(H) \geq \kappa(H \cup \langle E_X \rangle_G) - |E_X| = \kappa(G) - E_X$. Consequently, $\kappa(G - E_X) \geq \kappa(G) - |E_X|$. ■

The following approach may be used to construct a connectivity reducing spanning subgraph G', with $\kappa(G') = \ell$, from a graph G, with $\kappa(G) = k$. It follows from Proposition 4.1 that the removal of any $k - \ell$ edges will result in a graph with a connectivity number of at least ℓ. The actual connectivity number of G' may be computed using either Algorithm 3 or 4. If $\kappa(G') > \ell$ then $\kappa(G') - \ell$ edges may again be removed. This process may be repeated until $\kappa(G') = \ell$. This process is captured in pseudocode form in Algorithm 13.

Let n be the number of iterations of the while loop spanning lines 7 to 14 used to construct a spanning subgraph G', with $\kappa(G') = \ell$, from the graph G, with $\kappa(G) = k$. The for loop spanning lines 8 to 12 has a linear worst-case running time of $O(k - \ell)$ if *tmpConn* $= k$. The recalculation of the connectivity number in line 13 can be achieved in $O\left(p^3 q^2\right)$ time, if Algorithm 3 is implemented. The worst case running time of the while loop is thus dominated by the calculation of $\kappa(G')$. Hence, the running time of Algorithm 13 is $O\left(np^3 q^2\right)$.

Algorithm 13 runs, on average, much faster than an iterative implementation of Algorithm 12, as *tmpConn* $- \ell$ edges are removed at a time before the connectivity number of the graph G' is recalculated, instead of removing just one edge at a time.

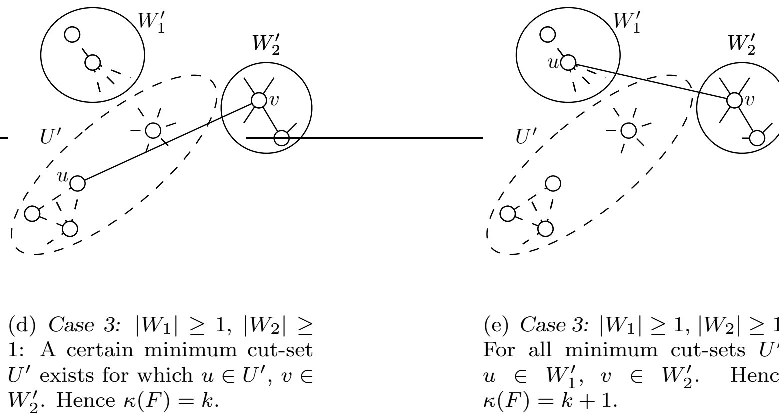
Stellenbosch University http://scholar.sun.ac.za



(a) *Case 1:* $k(H) > 2$. Hence $\kappa(F) = k$.

(b) *Case 2:* $u, v \in W_1$. Hence $\kappa(F) = k$.

(c) *Case 3:* $|W_1| \geq 1$, $|W_2| \geq 1$: Only one minimum cut-set exists, with $u \in W_1$ and $v \in W_2$. Hence $\kappa(F) = k$.



(d) *Case 3:* $|W_1| \geq 1$, $|W_2| \geq 1$: A certain minimum cut-set U' exists for which $u \in U'$, $v \in W_2'$. Hence $\kappa(F) = k$.

(e) *Case 3:* $|W_1| \geq 1$, $|W_2| \geq 1$: For all minimum cut-sets U', $u \in W_1'$, $v \in W_2'$. Hence $\kappa(F) = k + 1$.

Figure 4.12: *Graphical representations of the cases considered in Theorem 4.1.*

Algorithm 13 MEEF: Connectivity Reducing

Input: A connected graph G with connectivity number k. The desired connectivity number $\ell < k$.

Output: A connectivity reducing spanning subgraph G' of the graph G, such that $\kappa(G') = \ell$ and the cost saved.

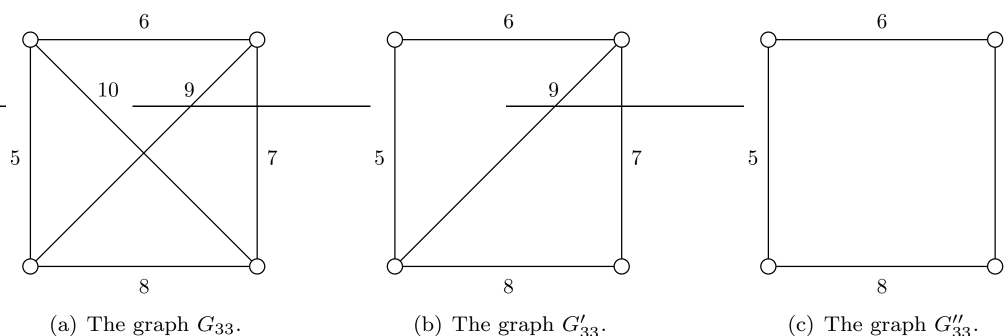
1: **if** the graph G is not connected **then**
2:   print "Graph is not connected.", stop.
3: **end if**
4: $G' \leftarrow G$
5: $costSaved \leftarrow 0$
6: $tmpConn \leftarrow \kappa(G')$
7: **while** $(tmpConn \geq k)$ **do**
8:   **for** $i \leftarrow tmpConn - \ell$ **do**
9:     find most expensive edge $uv \in E(G')$
10:     $costSaved \leftarrow costSaved + \text{cost}(uv)$
11:     $E(G') \leftarrow E(G') \setminus uv$
12:   **end for**
13:   $tmpConn \leftarrow \kappa(G')$
14: **end while**
15: $costSaved \leftarrow costSaved - \text{cost}(uv)$
16: $E(G') \leftarrow Etmp$
17: return G', $costSaved$

Stellenbosch University http://scholar.sun.ac.za

It should also be noted that, as Algorithm 13 also removes most expensive edges first, it has the same drawback as Algorithm 12: It may return a spanning subgraph with a connectivity number considerably higher than a minimum cost spanning subgraph with the same connectivity number (see Example 4.10). However, the cost of the graph G' returned by Algorithm 13 may possibly be reduced further by calling Algorithm 12 with the graph G' obtained from Algorithm 13 as input.

Example 4.11

Consider the graph G_{33} with edge weights as depicted in Figure 4.13(a). To construct a spanning



(a) The graph G_{33}. (b) The graph G'_{33}. (c) The graph G''_{33}.

Figure 4.13: *Graphical representations of the graphs relevant to Example 4.11.*

subgraph G'_{33} with a connectivity number of 2, from the graph G_{33} for which $\kappa(G_{33}) = 3$, Algorithm 13 may be implemented, producing the desired result, as depicted in Figure 4.13(b). The cost saved thus far is 10, but the cost of the graph G'_{33} may be further reduced by implementing Algorithm 12, resulting in the graph G''_{33} as depicted in Figure 4.13(c), with a total saved cost of 19. ■

4.6 Constructing spanning subgraphs by means of $F(x,U)$ fans

In this section, an algorithm, called Algorithm *Fan*, is presented which constructs a connectivity preserving spanning subgraph G' (by means of fans). Note that this algorithm requires that G must contain a cut-set U that is a clique. Fortunately, this is not too much of a disadvantage, as many types of graph constructions exist that contain a complete minimum cut-set. For example, electricity networks of cities are usually constructed in such a way such that the transformers form of a complete minimum cut-set. This ensures that, should one or more of the transformers fail, electricity can still be provided to the city. As another example, servers in a computer network may also form a complete minimum cut-set, where backup servers can take over control of the network, should the main servers fail.

In this section, the working of the algorithm is discussed, followed by proofs of the theoretical principles that it is based upon. Thereafter, an algorithm is described for constructing a connectivity reducing spanning subgraph from the graph G' in near-linear time.

4.6.1 Working of Algorithm *Fan*

Let G be a graph with connectivity number k and suppose it contains a minimum cut-set that is a clique. Call this cut-set U. Algorithm *Fan* then generates a connectivity preserving spanning subgraph $G' \subseteq G$ as follows. For every vertex $x \in V(G \setminus U)$, an $F(x,U)$ fan is constructed in the graph G', by making use of the edge set $E(G)$ to construct the fans. This may be achieved in $O\left(pq^2\right)$ worst-case time for each fan being constructed if Algorithm 9 is used for this purpose, with the vertex x acting as the source and the cut-set as T, the set of sinks. The algorithm also constructs $F(u,U)$ fans for every vertex $u \in U$, but these paths are not inserted into the graph G'. All paths constructed (by means of both the $F(x,U)$ and $F(u,U)$ fans) are stored in a path list *Paths*, which may be used later to construct a spanning subgraph with lower connectivity. The final step of the algorithm is to insert all edges between the vertices of the

Stellenbosch University http://scholar.sun.ac.za

cut-set into the graph G'. The graph G' may be formulated mathematically as

$$G' = \langle U \rangle_G \bigcup \left(\bigcup_{x \in V(G) \setminus U} F(x, U) \right) \subseteq G.$$

The pseudocode for Algorithm *Fan* is presented in Algorithm 14.

Algorithm 14 *Fan: Connectivity Preserving*

Input: A connected graph G with connectivity number k

Output: A connectivity preserving spanning subgraph G' of the graph G.

1: Find a minimum cut-set U of G that forms a clique. If no such cut-set can be found, then **stop**.

2: For every vertex $x \in V(G \setminus U)$, construct $F(x, U)$ fans by implementing Algorithm 9, with vertex x as the source s and U as T, the set of sinks. Insert the constructed paths into the graph G'. Store all paths constructed in this fashion in the path list *Paths*.

3: For every vertex $u \in U$, construct $F(u, U)$ fans by implementing Algorithm 9, with vertex u as the source s and U as T, the set of sinks. Store all paths constructed in this fashion in the path list *Paths*.

4: Insert all edges between the vertices of the cut-set into the graph G'.

A minimum cut-set for the graph G may be found using an adapted version of Algorithm 4 (step 1 of Algorithm 14). This adapted version should search through all minimum cut-sets (a total of $\binom{p}{k}$ possible cut-sets exist) for the graph G in an attempt to locate a set that is also a clique. The algorithm is not shown, as it is very similar to Algorithm 4. It was shown in section §4.2, however, that this algorithm has the same worst-case running time of $O\left(p^{k+3}\right)$, as that of Algorithm 4. As fans are constructed (by implementing Algorithm 9) from every vertex in $V(G')$ to the cut-set U, steps 2 and 3 of Algorithm 14 have a combined worst-case running time of $O\left(p^2q^2\right)$. As the cut-set is a clique, step 4 has a worst-case running time of $O(\binom{k}{2}) = O\left(k^2\right)$. Consequently, the worst-case running time of Algorithm 14 is $O\left(\max\left\{p^{k+3}, p^2q^2\right\}\right)$. The theory on which Algorithm 14 is based is now developed.

Theorem 4.2 *Suppose a graph G is a k-connected graph and that U is any set of k vertices of G. If $|V(G) \setminus U| = k - \ell$, for some $\ell \in \{1, \ldots, k-1\}$, then $\langle U \rangle_G$ is ℓ-connected.*

Proof Suppose $\langle U \rangle_G$ is not ℓ-connected, such that the removal of $\ell - 1$ vertices from $\langle U \rangle_G$ disconnects it. If a certain $\ell - 1$ vertices that disconnects $\langle U \rangle_G$ are removed, together with the vertices in $V(G) \setminus U$, then there exists a cut-set of G of cardinality $(\ell - 1) + (k - \ell) = (k - 1)$, which contradicts the fact that the smallest cut-set of G has cardinality k. ■

Theorem 4.3 *Suppose G is a graph with connectivity number k and that U is a minimum cut set of G such that $\langle U \rangle_G$ is a complete graph. Let $F(x, U)$ denote an $x - U$ fan for some $x \in V(G) \setminus U$ and let*

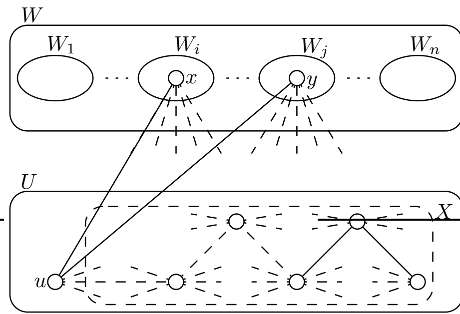
$$G' = \langle U \rangle_G \bigcup \left(\bigcup_{x \in V(G) \setminus U} F(x, U) \right) \subseteq G. \tag{4.1}$$

Then $\kappa(G') = k$.

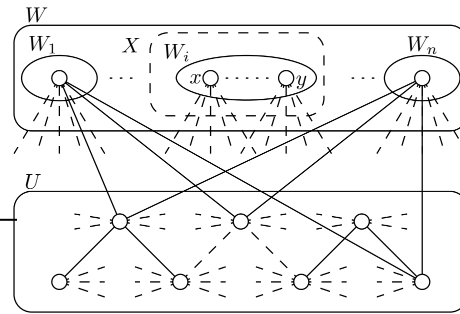
Proof Let $W = V(G) \setminus U$ and let X be any set of $k - 1$ vertices of G' and let $G'^* = \langle V(G') \setminus X \rangle'_G$. It is shown that G'^* is connected, by considering three cases. These cases and further subcases that result are presented graphically in Figure 4.14. The various cases are represented graphically in a tree format in Figure 4.15.

Case 1: $X \subset U$. A graphical representation of this case may be found in Figure 4.14(a). Let u be the vertex in $U \setminus X$. Consider any two components W_i and W_j of $\langle W \rangle_G$ and let $x \in W_i$, $y \in W_j$. Because u is an end-vertex of both fans $F(x, U)$ and $F(y, U)$, the components W_i and W_j are connected in G'^*. Hence G'^* is connected.

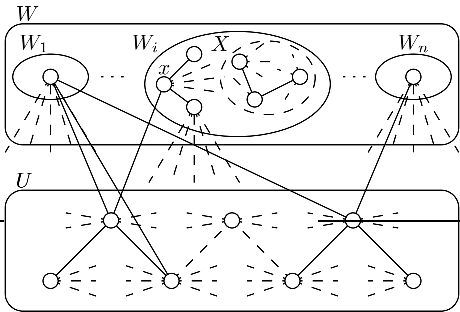
Stellenbosch University http://scholar.sun.ac.za



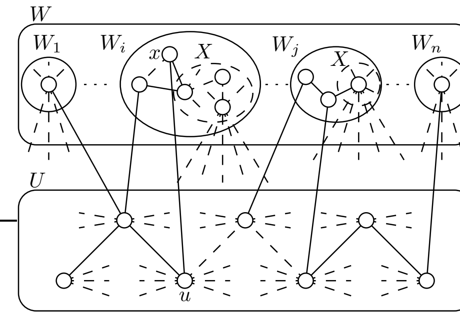
(a) *Case 1:* $X \subset U$.



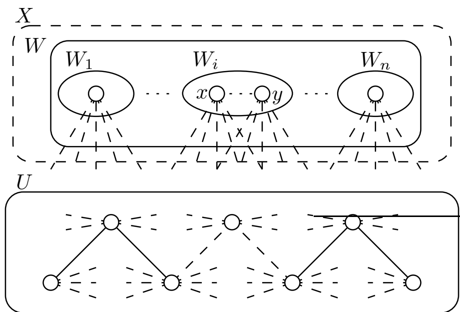
(b) *Case 2.1 (a):* $X \subseteq W$. Vertices are removed from a single component of W_i of W. The whole component is removed.



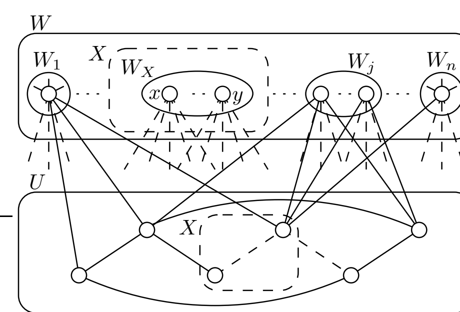
(c) *Case 2.1 (b):* $X \subseteq W$. Vertices are removed from a single component of W_i of W. Part of the component is removed.



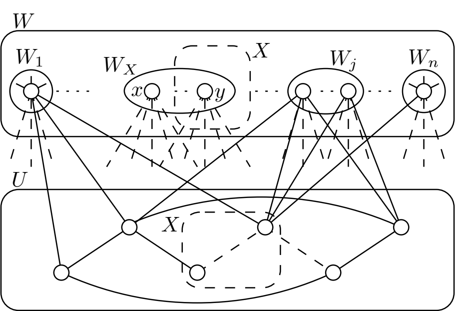
(d) *Case 2.2 (a):* $X \subseteq W$. Vertices are removed from more than one component of W, $|W| > |X|$ and $\langle U \rangle'_G$ is connected.



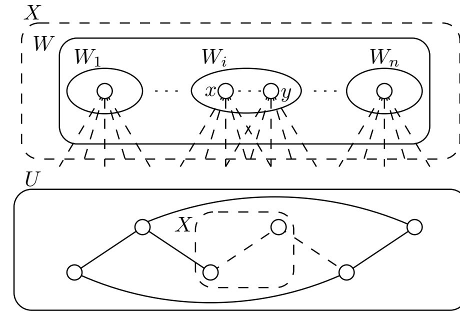
(e) *Case 2.2 (b):* $X = W$. All vertices in W are removed.



(f) *Case 3.1 (a):* $X \cap U \neq \emptyset$ and $X \cap W \neq \emptyset$. The whole of W_X and some vertices from U are removed.



(g) *Case 3.1 (b):* $X \cap U \neq \emptyset$ and $X \cap W \neq \emptyset$. Vertices are removed from U and only one component W_X of W. Not all vertices in W_X are removed.



(h) *Case 3.2 (b):* $X \cap U \neq \emptyset$ and $X \cap W \neq \emptyset$. The whole of W and some vertices from U are removed.

Figure 4.14: *Graphical representations of the cases considered in Theorem 4.3.*

Stellenbosch University https://scholar.sun.ac.za

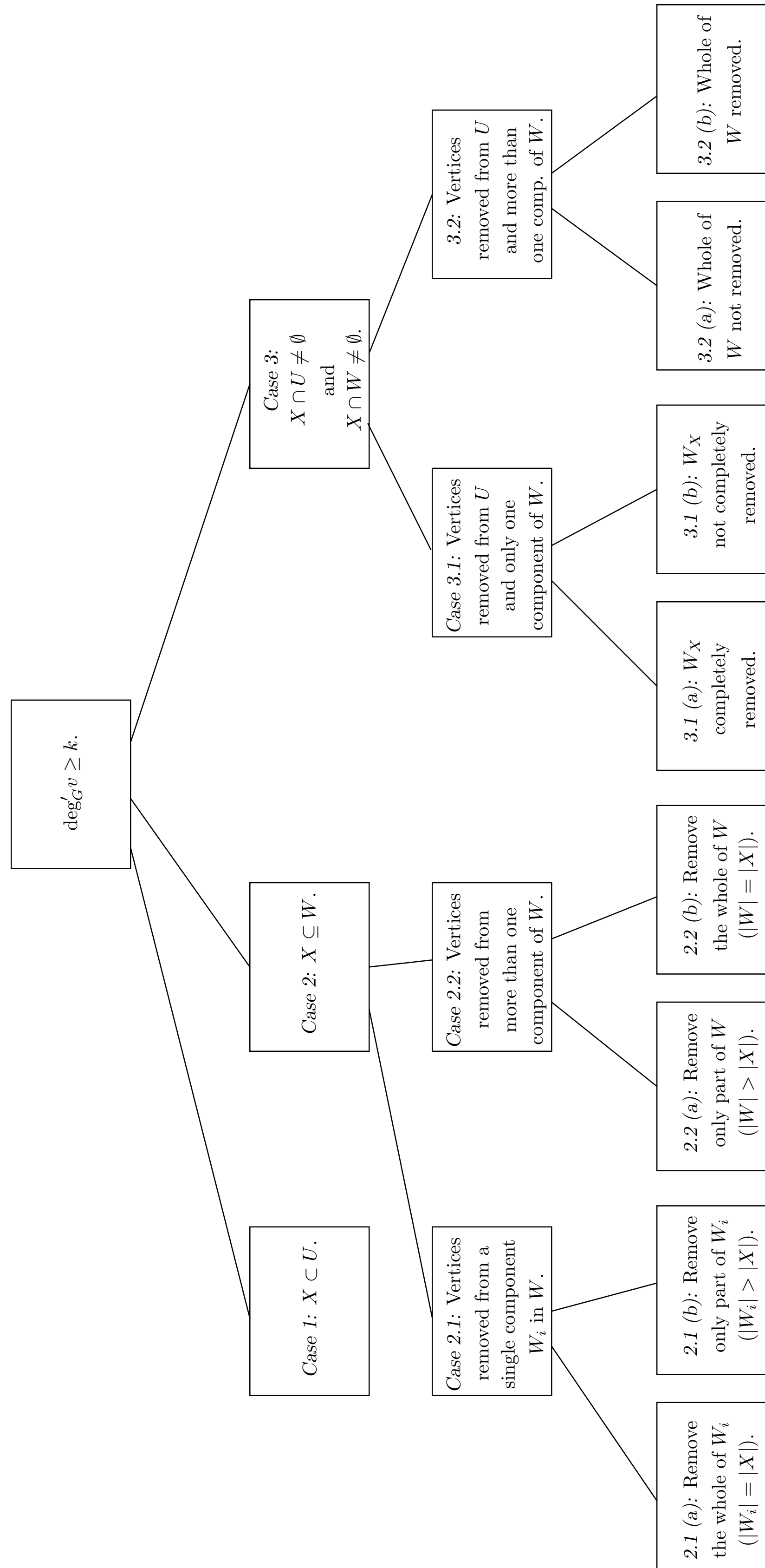


Figure 4.15: *Tree representation of the cases considered in Theorem 4.3.*

Stellenbosch University http://scholar.sun.ac.za

Case 2: $X \subseteq W$. Two cases result.
Case 2.1: If $|X|$ vertices are removed from a single component W_i in W, two sub-cases result.

(a) Firstly, the whole of W_i may be removed (thus $W_i = X$). A graphical representaion of this sub-case may be found in Figure 4.14(b). The remaining components (recall that W must consist of at least two components for G' to be k-connected) of W are still connected to U, as the different components of W have disjoint sets of paths to the vertices in U. Thus, G'^* is connected.

(b) Secondly, the cardinality of W_i may be greater than $k-1$. A graphical representation of this sub-case may be found in Figure 4.14(c). For this case, let x be one of the vertices in $W_i \setminus X$. Due to the construction of the fan $F(x,U)$, the removal of $|X| = k-1$ vertices from W_i cannot disconnect x from U (recall that the $x-U$ fan consists of k internally disjoint paths to the k vertices in U, hence, the removal of $k-1$ vertices can eliminate at most $k-1$ of these paths). The vertex x thus remains connected to at least one vertex in U. Furthermore, every other component of W (there must exist at least one other component in W other than W_i) is still connected to every vertex in U by means of fans. Therefore there remains at least one path from x to the other components of W. Hence, G'^* is connected.

Case 2.2: If $|X|$ vertices are removed from more than one component in W, a further two sub-cases result.

(a) First consider the case where $|W| > |X|$. A graphical representation of this sub-case may be found in Figure 4.14(d). Let $x \in W_i \setminus X$. As the removal of $|X|$ vertices can eliminate at most $k-1$ of the paths of any fan $F(x,U)$, x remains connected to at least one vertex in U. Consequently, as $\langle U \rangle$ is connected, G'^* remains connected.

(b) Secondly, the whole of W may be removed ($W = X$). A graphical representation of this case may be found in Figure 4.14(e). For this case $|W| \leq k-1$. Thus, G'^* will be connected if, after the removal of all the vertices in W, U remains connected. But this is guaranteed by Theorem 4.2, as $|W| = k-1$.

Case 3: $X \cap U \neq \emptyset$ *and* $X \cap W \neq \emptyset$. Two cases result.
Case 3.1: Firstly consider the case where $|X|$ vertices are removed from U and only one component of W. Let W_X be the component of W from which vertices are removed and let W_j be any other component of W. Two sub-cases result.

(a) Firstly, component W_X may be removed completely by the removal of $|X|$ vertices. A graphical representation of this subcase may be found in Figure 4.14(f). Because W_j is connected to every remaining vertex in U, G'^* is connected.

(b) For the second case, W_X may not be completely removed by the removal of $|X|$ vertices. A graphical representation of this subcase may be found in Figure 4.14(g). The number of vertices removed from W_X is strictly less than $k-1$. Let X_{W_X} and X_U denote the number of vertices removed from W_X and U respectively. For this case the number of vertices removed may be written as follows:

$$|X| = \underbrace{(k-i)}_{X_{W_X}} + \underbrace{(i-1)}_{X_U} = k-1, \qquad i = 2, \ldots, k-1.$$

Due to the construction of the fans from the vertices in W to U, the remaining vertices in W_X are connected by at least i remaining paths (the removal of $k-i$ vertices from W_X can eliminate at most $k-i$ paths, hence, i internally disjoint paths must remain from the remaining vertices in W_X to the vertices in U) to the vertices in U. Furthermore, only $i-1$ vertices are removed from U. Consequently, there remains at least $i-(i-1) = 1$ vertex in U to which the remaining vertices in W_X are connected. Furthermore, because W_j is connected to every vertex in U, G'^* is connected.

Case 3.2: Secondly, consider the case where $|X|$ vertices are removed from U and more than one component of W. Again, two subcases result.

(a) Firstly, consider the subcase where vertices remain in W after the removal of $|X|$ vertices. Let $X = A \cup B$, $A \subset U$, $B \subset W$. Since $\langle U \rangle'_G$ remains connected after the removal of $|A|$ vertices from U, G'^* remains connected. This follows as the removal of $|B| \in \{1, \ldots, k-1-|A|\}$ vertices from W can disconnect at most $|B|$ of the k paths of any fan $F(x,U)$. Hence x remains connected to the vertices in U.

(b) Now, consider the subcase where the whole of W and some vertices of U are removed by the removal of $|X|$ vertices. A graphical representation of this case may be found in Figure 4.14(h). For this case to occur, the cardinality of W must be such that $|W| = k-\ell$, $\ell \in \{2, \ldots, k-1\}$ so that $\ell - 1$ vertices can be removed from U, with $|U| = k > \ell$. Again, let X_U denote the number of vertices removed

Stellenbosch University http://scholar.sun.ac.za

from U. Hence, the expression

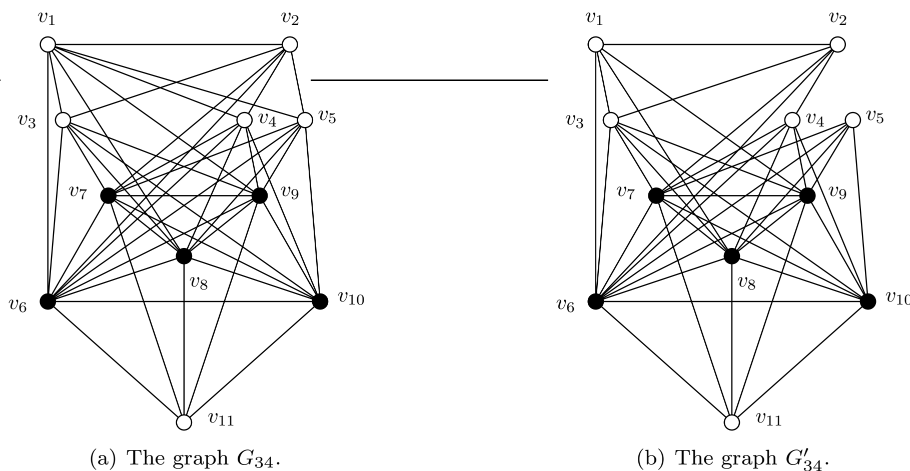
$$|X| = \underbrace{(k-\ell)}_{|W|} + \underbrace{(\ell-1)}_{X_U} = k-1, \qquad \ell = 2, \ldots, k-1$$

must hold. Theorem 4.2 guarantees that $\langle U \rangle'_G$ is ℓ-connected and hence remains connected after the removal of $\ell - 1$ vertices. ■

A clear disadvantage of Algorithm 14 is that it does not provide a means by which the paths that are entered into the spanning subgraph can be chosen. The paths that are selected by Algorithm 9 are constructed from flows obtained from Algorithm 6. These flows take place along certain edges irrespective of the edge weights. However, flows take place along the shortest paths. The paths obtained to construct a spanning subgraph may be such that the solution obtained from the algorithm is far from a minimum weighted spanning subgraph.

The following example illustrates the working of Algorithm 14. All edge weights are equal in the example. Hence, comparison between the original graph and the spanning subgraph is performed in terms of the number of edges each graph contains.

Example 4.12
Consider the graph G_{34} depicted in Figure 4.16(a), with $\kappa(G_{34}) = 5$. Note that the graph posesses a cut-set (the 5 vertices coloured black) that forms a clique. Algorithm 14 may be implemented to produce the spanning subgraph G'_{34}, also with a connectivity number of 5. The path list *Paths* constructed in steps 2 and 3 of Algorithm 14 are presented in Table 4.4.



(a) The graph G_{34}.

(b) The graph G'_{34}.

Figure 4.16: *The graph G_{34} and spanning subgraph G'_{34} (constructed from G_{34} using Algorithm 14), both having a connectivity number of 5. The three edges v_1v_4, v_1v_5 and v_2v_5 are not included in the graph G'_{34}. The black vertices in each graph denote a minimum cut-set for each graph.*

Note that the graph G'_{34} contains 3 edges fewer than the graph G_{34}, namely the three edges v_1v_4, v_1v_5 and v_2v_5. ■

4.6.2 Construction of a spanning subgraph G' of G, with $\kappa(G') \leq \kappa(G)$

Let H be a spanning subgraph generated from a graph G for which $\kappa(G) = k$, as described in Theorem 4.2. In this section a method is presented whereby a spanning subgraph H' of H may be isolated such that $\kappa(H') \geq \ell$, with $\ell < k$. Two lemmas that form the basis of this method are first established.

Lemma 4.1 *Let G be a graph for which $\kappa(G) = k$ and let $x \in V(G)$. Then $\kappa(G - x) \geq \kappa(G) - 1$.*
Proof By contradiction. Suppose $\kappa(G - x) \leq \kappa(G) - 2$. Then there exists a subset $V^* \subseteq V(G)$ of cardinality at most $\kappa(G) - 2$ such that $\langle V(G) - V^* \rangle_{G-x} = \langle V(G) - (\{x\} \cup V^*) \rangle_G$ is disconnected. This is a contradiction, because $V^* \cup \{x\}$ has cardinality at most $\kappa(G) - 1$, but G is disconnected by the removal of this set. ■

Stellenbosch University http://scholar.sun.ac.za

	Paths to				
Paths from	v_6	v_7	v_8	v_9	v_{10}
v_1	v_1, v_6	v_1, v_2, v_7	v_1, v_8	v_1, v_9	v_1, v_3, v_{10}
v_2	v_2, v_6	v_2, v_7	v_2, v_1, v_8	v_2, v_3, v_9	v_2, v_4, v_{10}
v_3	v_3, v_6	v_3, v_7	v_3, v_8	v_3, v_9	v_3, v_{10}
v_4	v_4, v_6	v_4, v_7	v_4, v_8	v_4, v_9	v_4, v_{10}
v_5	v_5, v_6	v_5, v_7	v_5, v_8	v_5, v_9	v_5, v_{10}
v_6	—	v_6, v_7	v_6, v_8	v_6, v_9	v_6, v_{10}
v_7	v_7, v_6	—	v_7, v_8	v_7, v_9	v_7, v_{10}
v_8	v_8, v_6	v_8, v_7	—	v_8, v_9	v_8, v_{10}
v_9	v_9, v_6	v_9, v_7	v_9, v_8	—	v_9, v_{10}
v_{10}	v_{10}, v_6	v_{10}, v_7	v_{10}, v_8	v_{10}, v_9	—
v_{11}	v_{11}, v_6	v_{11}, v_7	v_{11}, v_8	v_{11}, v_9	v_{11}, v_{10}

Table 4.4: *The list of paths constructed in steps 2 and 3 of Algorithm 14 that are inserted into the graph G'_{34}.*

Lemma 4.2 *Let G be a graph for which $\kappa(G) = k$, let U be a minimum cut-set U of G, and let $u \in U$. Then $\kappa(G - u) = \kappa(G) - 1$ and $U - u$ is a minimum cut-set of $G - u$.*
Proof It is clear that $\kappa(G - u) \leq \kappa(G) - 1$, because removing a vertex from a cut-set of the graph G necessarily reduces its connectivity number by at least one. But it also follows by Lemma 4.1 that $\kappa(G - u) \geq \kappa(G) - 1$. Consequently, $\kappa(G - u) = \kappa(G) - 1$. Suppose there exists a cut-set of cardinality at most $|\langle U - u\rangle_G| - 1 = |U| - 2$ for the graph $G - u$. This is a contradiction, as it follows from the result that $\kappa(G - u) = \kappa(G) - 1$, that $\kappa(G - u) = \kappa(G) - |u| = |U| - 1$. Hence $U - u$ is a minimum cut-set of $G - u$. ■

The algorithm for isolating the subgraph H' of H, as described above, hinges on the following result.

Proposition 4.2 *Let G be a graph for which $\kappa(G) = k$ and let U_X be any set of $|X|$ vertices of a minimum cut-set U of G. Then $\kappa(G - U_X) = \kappa(G) - |U_X|$.*
Proof The result follows by applying Lemma 4.2 $|U_X|$ times to the graph G. ■

Consider a spanning subgraph H of a graph G for which $\kappa(H) = k$ that was obtained by means of Algorithm 14. An algorithm that may be used to construct a connectivity reducing spanning subgraph from H by the removal of certain edges is discussed next. The pseudocode for the algorithm is presented in Algorithm 15. Graphical representations accompanying each step of the algorithm are given in Figure 4.17.

Algorithm 15 *Fan: Connectivity Reducing*

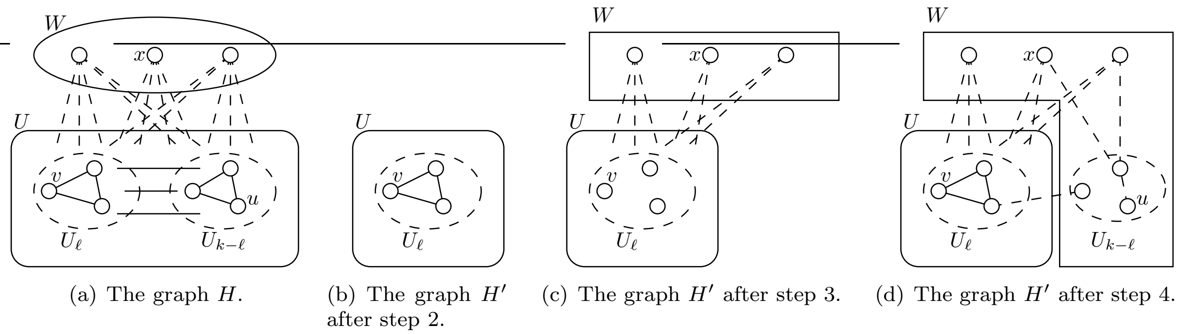
Input: A spanning subgraph H for which $\kappa(H) = k$, generated from a graph G for which $\kappa(G) = k$, by means of Algorithm 14. The parameter $\ell < k$, the desired minimum connectivity number of the spanning subgraph H. A minimum cut-set U for the graph H, with the vertices of U forming a clique. The path list *Paths* that was generated by Algorithm 14.
Output: A connectivity reducing spanning subgraph $H' \subseteq H$ for which $\kappa(H') \geq \ell$.

1: Let $W = V(H) \setminus U$. Furthermore, let $U_{k-\ell}$ be any set of $k - \ell$ vertices in U and let $U_\ell = \langle U \setminus U_{k-\ell}\rangle_H$.
2: Insert all edges between the vertices of the set U_ℓ into the graph H'.
3: Let $v \in U_\ell$. Then, for every $x \in V(H \setminus W)$, insert all x-v paths in the list *Paths* into the graph H'.
4: Let $v \in U_\ell$. Then, for every $u \in U_{k-\ell}$, insert all u-v paths in the list *Paths* into the graph H'.

The worst-case running time of Algorithm 15 is now established. Line 1 of Algorithm 15 has a worst-case running time of $O(k)$, as each vertex in the set U is moved to either the set U_ℓ or $U_{k-\ell}$. As $p - \ell$ paths are entered into the graph H', line 2 has a worst-case running time of $O(p - \ell)$. All $\binom{\ell}{2}$ edges between vertices in U_ℓ are inserted in the graph H' in line 3; hence it has a worst-case running time of $O(\ell^2)$. Consequently, the worst-case running time of algorithm 15 is $O\left(\min\left\{p - \ell, \ell^2\right\}\right)$.

The correct working of Algorithm 15 is established in the following theorem.

Stellenbosch University http://scholar.sun.ac.za



(a) The graph H. (b) The graph H' after step 2. (c) The graph H' after step 3. (d) The graph H' after step 4.

Figure 4.17: *Graphical representation of the different steps of Algorithm 15. The dashed lines from the vertices in W represent possible edges inserted by $F(x,U)$ fans, $x \in W$. Not all vertices or edges are necessarily shown.*

Theorem 4.4 *For the spanning subgraph H' formed by the procedure described in Algorithm 15 it holds that $\kappa(H') \geq \ell$.*

Proof Suppose a spanning subgraph H satisfying $\kappa(H) = k$ is deduced from a graph G with $\kappa(G) = k$, as described in Theorem 4.3. Furthermore, let $U_{k-\ell}$ be any set of $k-\ell$ vertices of the minimum cut-set U of H. By Proposition 4.2 it follows that $\kappa(H \setminus U_{k-\ell}) = k - (k-\ell) = \ell$, as $\kappa(H) = k$. Construct the graph H' as described in steps 2 and 3 of Algorithm 14. As H' is a spanning subgraph of $H \setminus U_{k-\ell}$, it follows from Theorem 4.3 that $\kappa(H') = \kappa(H \setminus U_{k-\ell}) = \ell$. Now complete the construction of the spanning subgraph H' as described in step 4 of Algorithm 14. As each vertex $u \in U_{k-\ell}$ is connected to every vertex in U_ℓ by means of $F(u, U_\ell)$ fans, it follows from Theorem 3.14 that $\kappa(H') \geq \ell$. ■

The following example illustrates the working of Algorithm 15.

Example 4.13 (Continuation of Example 4.12)

Consider again the graph G_{34} depicted in Figure 4.16(a). Algorithm 15 may be applied to produce a subgraph G''_{34} with connectivity number ranging from zero to 5. For this example, when Algorithm 15 is used to construct a spanning subgraph with a smaller connectivity number of ℓ, then the first $k-\ell$ vertices are used to represent the set $U_{k-\ell}$. All six resulting graphs are depicted in Figure 4.18. The number of edges present in each graph are also given.

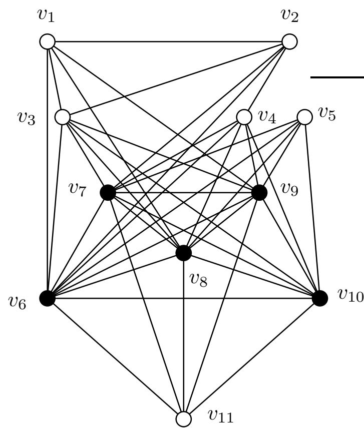
The paths that are inserted for each version of the graph G''_{34} may be obtained from Table 4.4. For instance, Figure 4.18(d) represents the graph G''_{34} with a connectivity number of 3. Table 4.5 lists the paths that are inserted into the graph G''_{34} such that its connectivity number is 3. The vertices v_9 and

	Paths to		
Paths from	v_6	v_7	v_8
v_1	v_1, v_6	v_1, v_2, v_7	v_1, v_8
v_2	v_2, v_6	v_2, v_7	v_2, v_1, v_8
v_3	v_3, v_6	v_3, v_7	v_3, v_8
v_4	v_4, v_6	v_4, v_7	v_4, v_8
v_5	v_5, v_6	v_5, v_7	v_5, v_8
v_9	v_9, v_6	v_9, v_7	v_9, v_8
v_{10}	v_{10}, v_6	v_{10}, v_7	v_{10}, v_8
v_{11}	v_{11}, v_6	v_{11}, v_7	v_{11}, v_8

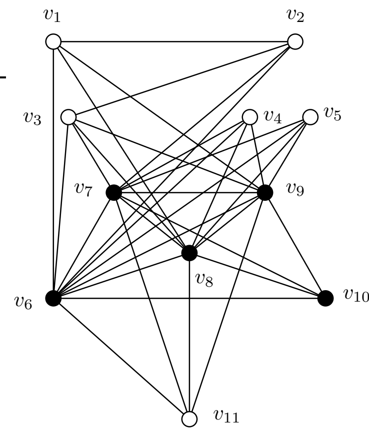
Table 4.5: *The list of paths used to construct the graph G''_{34} with a connectivity number of 3 (see Figure 4.18(d) for a graphical representation of this graph).*

v_{10} comprise the set $U_{k-\ell}$ and are removed from the cut-set of the graph G. This implies that, for every path starting in a vertex $x \notin U$, only the paths listed in Table 4.4 that do not end in vertices in the set $U_{k-\ell}$ are included in the graph G''_{34}. Furthermore, all paths starting from vertices in the set $U_{k-\ell}$ and ending in vertices in the remaining cut-set U_k, are included in the graph G''_{34}. ■

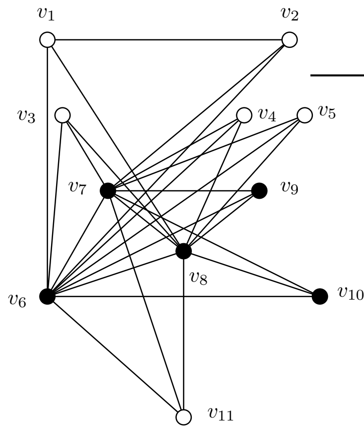
Stellenbosch University http://scholar.sun.ac.za



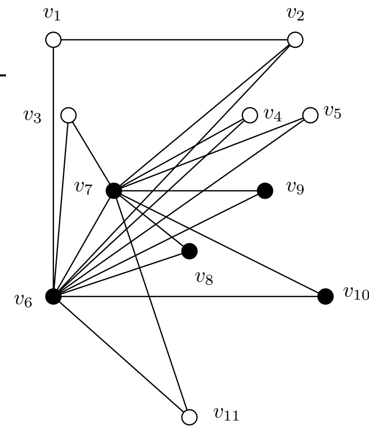
(a) The graph G''_{34} consisting of 39 edges, with $\kappa(G''_{34}) = 5$.



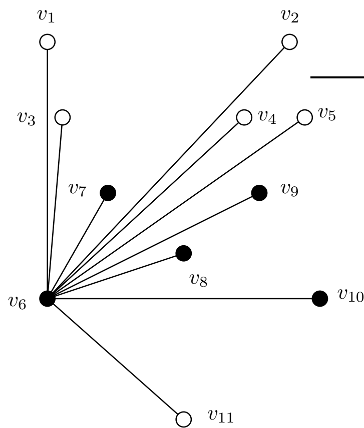
(b) The graph G''_{34} consisting of 33 edges, with $\kappa(G''_{34}) = 4$.



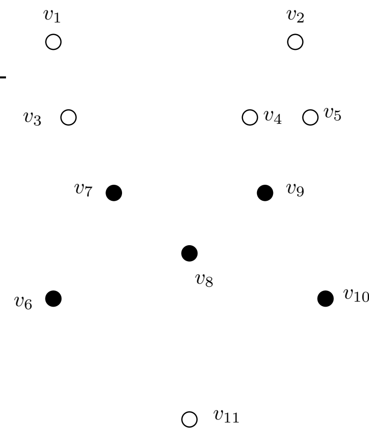
(c) The graph G''_{34} consisting of 26 edges, with $\kappa(G''_{34}) = 3$.



(d) The graph G''_{34} consisting of 19 edges, with $\kappa(G''_{34}) = 2$.



(e) The graph G''_{34} consisting of 10 edges, with $\kappa(G''_{34}) = 1$.



(f) The graph G''_{34} consisting of 0 edges, with $\kappa(G''_{34}) = 0$.

Figure 4.18: *Different versions of the graph G''_{34}, obtained as output from Algorithm 15, with connectivities ranging from 5 to zero in Figures 4.18(a)–(f) respectively. The vertices comprising the selected cut-set are coloured black.*

Stellenbosch University http://scholar.sun.ac.za

4.7 Comparison of Algorithms

All of the algorithms discussed in this section require knowledge of a minimum cut-set of a given graph. Algorithm *Whitney* only requires the connectivity number of a graph in order to determine the maximum number of paths that should be constructed between every pair of vertices. For instance, if a graph G has a connectivity number of k, then there exists at least one pair of vertices in the graph for which there exist exactly k internally disjoint paths connecting them. No set of $k+1$ paths can be constructed between such a pair of vertices.

Algorithm *MEEF* also only requires the connectivity number of the input graph. This is recalculated every time an edge or set of edges are removed from the original graph. This process is terminated once the graph obtained in this fashion has the desired connectivity number (see Algorithms 12 and 13). The calculation of the connectivity number may be achieved by means of the efficient Algorithm 3.

To be able to implement Algorithm *Fan*, the input graph must contain a minimum cut-set that is also a clique. To find such a set of vertices, Algorithm 4 is required, which has a relatively high running time compared to Algorithm 3.

After the connectivity number of the graph (for Algorithm *Whitney*) or a minimum complete cut-set (for Algorithm *Fan*) has been calculated, both Algorithms *Whitney* and *Fan* have very fast running times. The calculation of the connectivity number or a minimum complete cut-set is performed only once for these algorithms. Algorithm *MEEF*, on the other hand, recalculates a minimum cut-set every time an edge or set of edges is removed (see Algorithms 12 and 13). The running times of the three algorithms are provided in Table 4.6 with and without (where possible) the calculation of information of a minimum cut-set. Two columns are shown. The first contains running times for the case where the algorithms are used to construct a spanning subgraph G' with the same connectivity k as the original graph G. The second column contains running times for the case where the algorithms are used to construct a spanning subgraph which is at least ℓ-connected.

	Worst-case running time			
	$\kappa(G') = \kappa(G)$		$\kappa(G') \geq \ell$	
	With cut-set	Without cut-set	With cut-set	Without cut-set
Whitney	$O(p^3q^2)$	$O(p^3q^2)$	$O(p^3q^2)$	$O(p^3q^2)$
MEEF	$O(p^3q^3)$	N/A	$O(np^3q^2)$	N/A
Fan	$O(\min\{p^{(k+3)}, p^2q^2\})$	$O(p^2q^2)$	$O(\min\{p^{(k+3)}, p-l, l^2\})$	$O(\min\{p-l, l^2\})$

Table 4.6: *Summary of the worst-case running times of Algorithms* Whitney, MEEF *and* Fan.

From the results in Table 4.6 it may be seen that Algorithms *Whitney* and *Fan* have the lowest worst-case running times, with Algorithm *Fan* being an order of magnitude faster for constructing a spanning subgraph G' for which $\kappa(G') = \kappa(G)$. The reason for the near linear worst-case running time of Algorithm *Fan* for constructing a spanning subgraph that is at least ℓ-connected, is that it simply reads the paths to insert into the spanning subgraph from a list that was constructed when a subgraph with the same connectivity as the original graph was constructed.

A disadvantage of Algorithm *Whitney* mentioned earlier is that the spanning subgraph it generates may contain all or almost all of the edges of the original graph, due to the sets of internally disjoint paths between every pair of vertices that are inserted into the spanning subgraph. Algorithm *Fan* usually outperforms Algorithm *Whitney* in this regard, as only $(p-k)$ sets of internally disjoint paths are inserted into the spanning subgraph (there are $(p-k)$ vertices in the graph that do not form part of a specific cut-set, each having a set of k internally disjoint paths to the vertices in that cut-set), compared to the $\binom{p}{2}$ sets inserted by Algorithm *Whitney*.

It should be noted that the three algorithms in this chapter only attempt to provide a method whereby relatively low cost spanning subgraphs can be constructed. The results obtained may be far from optimal. The problem of constructing a minimum cost spanning subgraph that is k-connected is *NP*-complete (see Kortsarz & Nutov [34]). It is clear from Table 4.6, however, that Algorithm *Fan* has the best worst-case running time of all three algorithms.

 Stellenbosch University http://scholar.sun.ac.za

Only a rough comparison can be made between the running times of the approximation algorithm that was developed by Kortsarz & Nutov [34], referred to as the *K-N* algorithm, and Algorithm *Fan*. The *K-N* algorithm has a worst-case running time of $O(k^2pq^2)$. The connectivity number k may be as large as $p-1$; hence $k = O(p)$. The worst-case running time for the *K-N* algorithm then becomes $O(p^3q^2)$, which is an order of magnitude larger than that of the worst-case running time for Algorithm *Fan* for constructing a connectivity preserving spanning subgraph (assuming a complete minimum cut-set has already been calculated). The *K-N* algorithm suffers the drawback that it cannot construct connectivity reducing spanning subgraphs. Here, the near-linear worst-case running time of Algorithm *Fan* for constructing connectivity reducing spanning subgraphs does show its worth. Algorithm *Fan* has a worst-case running time of $O(\min\{p-l, l^2\})$.

In their article, Kortsarz & Nutov [34] proved that the spanning subgraph obtained from their algorithm always returns a spanning subgraph of weight at most a fixed factor of $O(\ln k \cdot \min\{\sqrt{k}, p/(p-k)\ln k\})$ more than a minimum weighted spanning subgraph with a connectivity number of k. No such approximation is known for any of the algorithms discussed in this chapter. Even though these algorithms do not produce minimum cost spanning subgraphs, their results may hopefully aid in the process of finding and constructing a spanning subgraph with a near-minimum weighting.

4.8 Chapter Summary

In this chapter, three algorithms, namely Algorithms *Whitney*, *MEEF* and *Fan*, were introduced. The theoretical background for each algorithm was also provided. Each of these algorithms have advantages and disadvantages associated with them, which may be weighed up against each other in order to select an Algorithm that is suitable to implement in a given scenario. The chapter closed with a comparison of the algorithms amongst each other and also with the *K-N* algorithm recently developed by Kortsarz & Nutov [34]. Algorithm *Fan* proves to be a more efficient algorithm in terms of speed, but it has the drawback that it is not known how suboptimal the solution obtained is.

Stellenbosch University http://scholar.sun.ac.za

Stellenbosch University http://scholar.sun.ac.za

Chapter 5

Decision Support System

In this chapter an overview of a decision support system (DSS) is given which implements the connectivity algorithms *Whitney*, *MEEF* and *Fan* — these algorithms were discussed in §4. The DSS is called *Connectivity Algorithms* and allows the user to construct spanning subgraphs with various connectivity numbers that may then be compared to one another. There are two reasons why the development of this DSS was deemed necessary. Firstly, most input graphs consist of a large number of vertices and edges, making it practically impossible to implement these algorithms by hand, as many calculations are done (a large number of calculations are done even for small graphs consisting of just a few vertices, when implementing Algorithms *Whitney* and *Fan* to find internally disjoint paths). These calculations require little computer time, and, given that the algorithms are coded correctly, the user has the assurance that these calculations are performed correctly. Secondly, the output resulting from the three algorithms mentioned for constructing a spanning subgraph with a given connectivity number may differ quite drastically. This allows the user to choose between different constructions. The user may, for instance, decide to choose a constructed spanning subgraph with a larger total weight than another constructed spanning subgraph, because the layout of that graph may better match the setup required by the user.

The first section of this chapter provides the reader with technical information on the DSS. This is followed by an explanation of the different components of the system. A complete worked example is discussed next, in which all three algorithms are used to generate output for the same graph. This is done so as to give the reader a better feel for how these algorithms operate. The chapter is concluded with a case study on the connectivity of a spider's web, where the running times and solution qualities of the three algorithms are compared. The chapter is concluded with a chapter summary.

5.1 Technical aspects & limitations of *Connectivity Algorithms*

Connectivity Algorithms was developed using Microsoft Visual Basic 6.0 [61]. It is a 32-bit programming language, implying that the largest integer that it can store is a 31-bit value (the last bit is used to indicate the sign of the number stored). This imposes a physical limitation in the sense that only graphs comprising 31 or fewer vertices can be specified as input. The reason for this limitation is that each vertex contained in a specific minimum cut-set of the input graph is represented by a 1 in a 31-bit variable. As any of the 31 vertices may form part of a possible minimum cut-set, 31 bits are required (see §4.2 for a detailed explanation on how the vertices comprising a minimum cut-set are stored using Algorithm *Cut-Vertex Set*). It is possible to circumvent this limitation by creating a bit-vector comprising an array of 32-bit bit-vectors. This was deemed unnecessary, as the running time of Algorithms *Fan* implemented in the DSS takes a long time to execute, due to the slower running time of Algorithm 4 that is used to find a complete minimum cut-set in a graph. A faster algorithm for calculating such a minimum cut-set would first need to be developed before the DSS would be able to construct spanning subgraphs in reasonable time for graphs comprising more than 31 vertices.

The CD accompanying this thesis contains the DSS executable — the reader is referred to Appendix

Stellenbosch University http://scholar.sun.ac.za

A for instructions on how to utilise the CD. Adjacency matrices of the graphs in the example and case study presented in this chapter are also included on the CD.

5.2 Introduction to the system components

In this section the various components of *Connectivity Algorithms* are discussed. Once the program is started, the *Main window* of the program is displayed (see Figure 5.1).

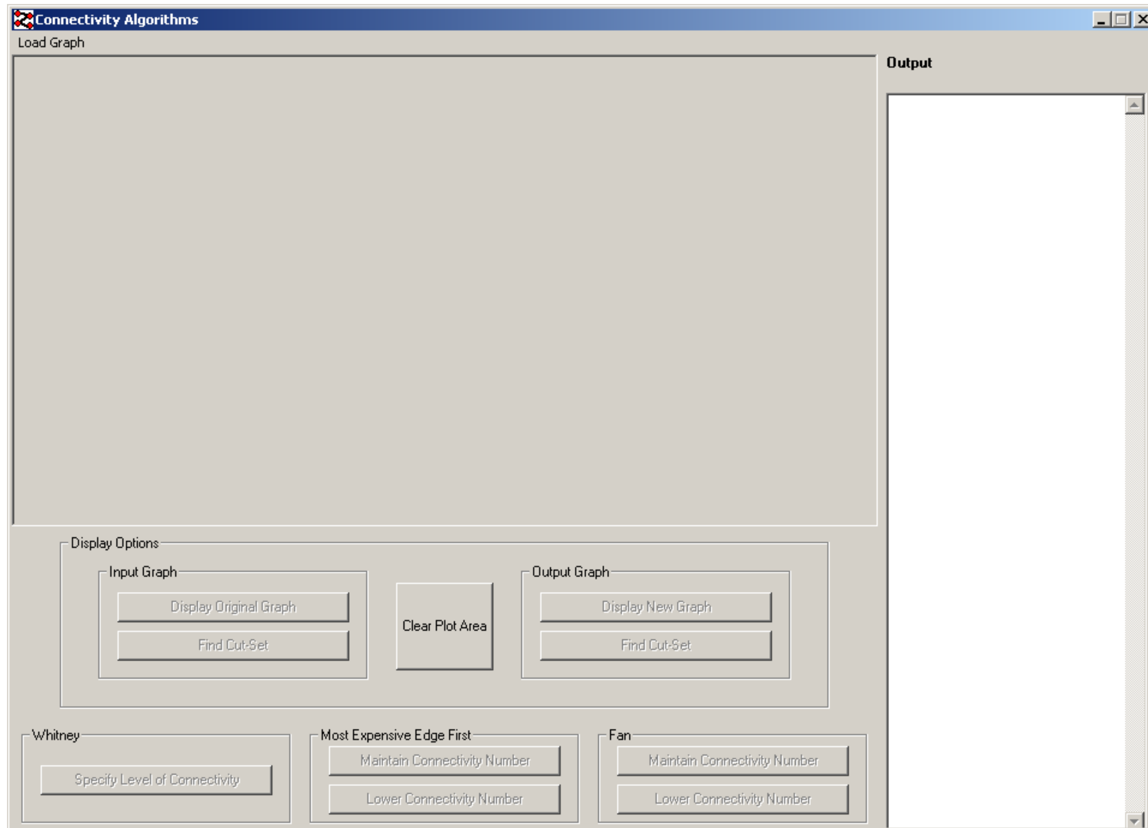


Figure 5.1: *Main window of the program* Connectivity Algorithms.

The main window consists of a number of components. The *Plot Area* is the rectangular space located to the left of the *Output* window and above the *Display Options* frame box. The *Plot Area* is used to represent an input graph graphically. When the DSS is started, no graph is loaded and the *Plot Area* is initialised as a clear screen. The *Output* window provides output information, depending on what algorithm is implemented. Information, such as the name of the algorithm implemented and the weight improvement in the subgraph generated, are displayed here. A number of buttons are located below the *Plot Area*. The first set of buttons is located in the *Display Options* frame box and is used to display either the original graph or spanning subgraph generated by some algorithm. A minimum cut-set for both graphs may also be displayed by clicking on the *Find Cut-Set* button in either the *Input Graph* or *Output Graph* frame boxes. If no graph is loaded, the only button that is enabled is the *Clear Plot Area* button, which, as its name suggests, clears the *Plot Area*. The second set of buttons is used to execute one of the three algorithms, *Whitney*, *MEEF* or *Fan*, on the graph loaded.

The *Load Graph* button in the menu bar (top left of Figure 5.1) is used to load a new graph. This option opens the *Load New Graph* window, depicted in Figure 5.2.

The first three windows in the *Load New Graph* window are used to search for an Excel [48] file that contains the adjacency matrix for a graph that is used to construct the input graph required by the DSS. The first of these three windows specifies the drive on which the file is located. The window below this

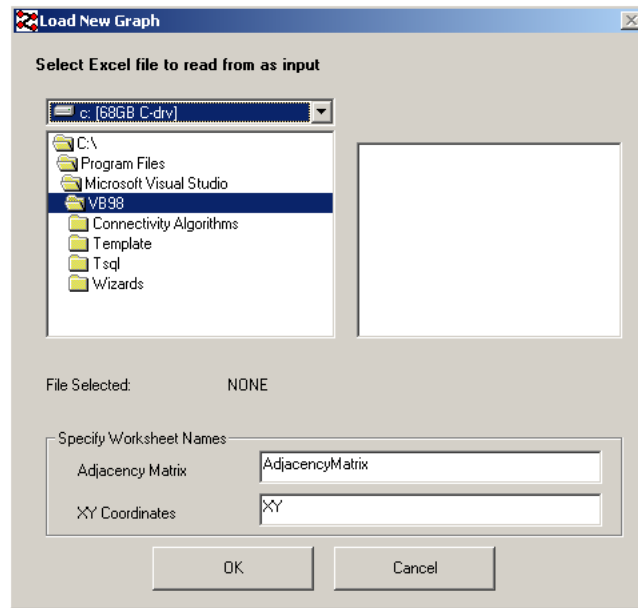


Figure 5.2: *The window,* Load New Graph*, is used to select an input graph for the DSS.*

specifies the directory and the window to its right is used to select a specific Excel [48] file as input. Once a file is selected, its name is displayed next to the *File Selected* label just below these three windows. Two worksheet names are required. The first worksheet specifies the x- and y-coordinates of all vertices in the graph. This information is stored as follows in the Excel [48] spreadsheet. Column i contains the coordinates of vertex i, with the x-coordinate stored in row 1 and the y-coordinate stored in row 2. No other information should be stored on this worksheet. These coordinates are used and then scaled to fit the resulting graph in the *Plot Area* window on the main form. The second worksheet specifies the adjacency matrix for the input graph. Element $(i, j) = (j, i)$ of the adjacency matrix is stored in the ith row and j^{th} column of the worksheet specified, and indicates the weight of the edge $v_iv_j = v_jv_i$. The DSS automatically calculates the number of vertices by calculating the number of non-empty rows on this worksheet; hence no information other than the adjacency matrix should be included on this worksheet.

5.3 A worked example

In this section the working of the DSS is discussed with the aid of an example. All three algorithms are implemented, so as to give the user a feel for the different algorithms.

Consider again the graph G_{34} depicted in Figure 4.16 (see Example 4.12). The graph is reproduced in Figure 5.3, for referencing purposes.

For the purpose of this example, the edges are weighted. All weights are reflected in the adjacency matrix for the graph that is stored in the Excel [48] file *testgraph.xls* on the worksheet *AdjacencyMatrix* (see Figure 5.4). The file *testgraph.xls* is located on the CD accompanying this thesis. The edge weights are not shown in Figure 5.3 so as to render the graph less cluttered.

Note that the adjacency matrix is symmetric, as the graph is undirected (the algorithms discussed in this thesis are designed specifically for undirected graphs). The coordinates for each vertex is stored in the worksheet *XY* of the file *testgraph.xls* (see Figure 5.5).

The DSS may be loaded by executing the file *connalgs.exe* from the CD accompanying this thesis. Once the program is launched, the file *testgraph.xls* may be loaded by clicking on *Load Graph* and browsing to the location where the file is stored (see Figure 5.6).

After the file has been selected, the filename *testgraph.xls* appears next to the *File Selected* label. The default names for the worksheets containing the adjacency matrix and coordinates of the vertices are

Stellenbosch University http://scholar.sun.ac.za

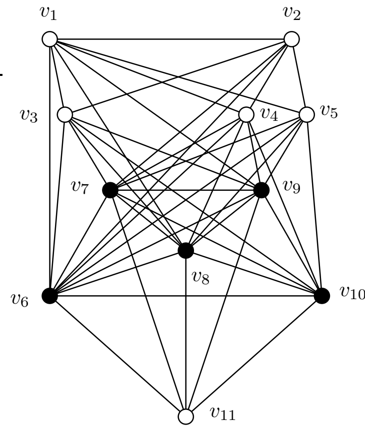


Figure 5.3: *Graphical representation of the graph* G_{34}.

Microsoft Excel - testgraph.xls

	A	B	C	D	E	F	G	H	I	J	K
1	0	2	4	7	1	8	0	2	2	0	0
2	2	0	1	1	5	4	2	0	0	0	0
3	4	1	0	0	0	7	6	5	1	2	0
4	7	1	0	0	0	3	2	1	2	3	0
5	1	5	0	0	0	1	1	2	2	7	0
6	8	4	7	3	1	0	7	3	9	3	1
7	0	2	6	2	1	7	0	1	2	1	3
8	2	0	5	1	2	3	1	0	3	2	1
9	2	0	1	2	2	9	2	3	0	4	1
10	0	0	2	3	7	3	1	2	4	0	5
11	0	0	0	0	0	1	3	1	1	5	0

XY | AdjacencyMatrix

Figure 5.4: *The constructed adjacency matrix for the graph* G_{34} *is stored in the worksheet AdjacencyMatrix of the file testgraph.xls.*

Microsoft Excel - testgraph.xls

	A	B	C	D	E	F	G	H	I	J
1	2.5	4.5	3	4	5	1	2.1	3	3.9	5
2	5	5	4	4	4	2	3.6	2.5	3.6	2

XY | AdjacencyMatrix

Figure 5.5: *The coordinates for the vertices of the graph* G_{34} *are stored in the worksheet XY of the file testgraph.xls.*

AdjacencyMatrix and *XY* respectively; hence the loading operation is completed by simply clicking on the *OK* button, after which the main window of the DSS reappears. The graph may now be displayed in the *Plot Area* window by clicking on the *Display Original Graph* button in the *Input Graph* frame box (see Figure 5.7).

A minimum cut-set may be found and displayed in the *Plot Area* window by clicking on the *Find Cut-Set* button in the *Input Graph* frame box. Vertices are coloured light green by default. After clicking on the *Find Cut-Set* button in the *Input Graph* frame box, vertices comprising a minimum cut-set that is found by the DSS for the original graph are coloured red. By clicking on the *Find Cut-Set* button in the

 Stellenbosch University http://scholar.sun.ac.za

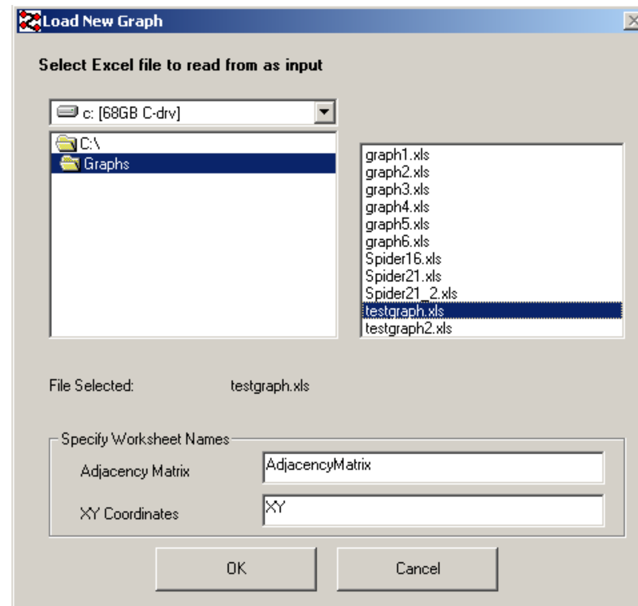


Figure 5.6: *Visual representation of the loading process of the file testgraph.xls.*

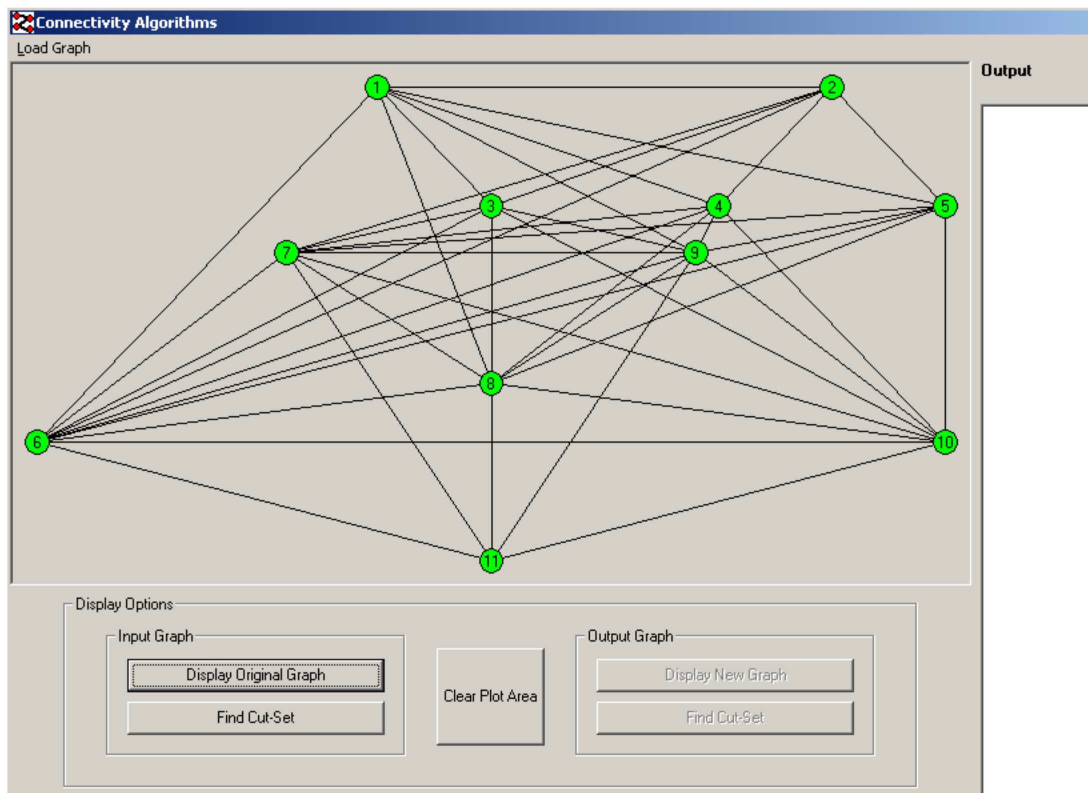


Figure 5.7: *The graph G_{34}, represented by the information in the file testgraph.xls, is displayed after clicking on the Display Original Graph button in the Input Graph frame box.*

Output Graph frame box, vertices comprising a minimum cut-set for the relevant spanning subgraph are coloured yellow. For this example, five vertices are coloured red; hence the connectivity number for the original graph is 5 (see Figure 5.8).

The three algorithms *Whitney*, *MEEF* and *Fan* may now be implemented on this graph. The connectivity number of the output spanning subgraph is varied using each of these algorithms, generating spanning subgraphs with connectivity numbers ranging from 5 down to 0.

Stellenbosch University http://scholar.sun.ac.za

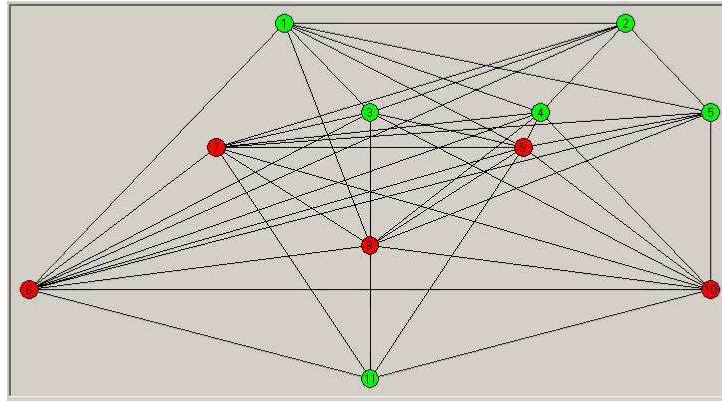


Figure 5.8: *The graph G_{34} represented by the information in the file testgraph.xls after clicking on the Find Cut-Set button in the Input Graph frame box. The 5 vertices comprising minimum cut set obtained by the DSS are coloured red.*

5.3.1 Implementation of Algorithm *Whitney*

Algorithm *Whitney* may be implemented by clicking on the *Specify Level of Connectivity* button in the *Whitney* frame box. A window called *Enter the desired connectivity number* is displayed, prompting the user to enter the desired level of connectivity (see Figure 5.9).

Figure 5.9: *The window* Enter the desired connectivity number *appears after clicking on Specify Level of Connectivity in the Whitney frame box. The desired connectivity number for a spanning subgraph, which is bounded by the connectivity number of the original graph, may now be entered.*

Note that the connectivity number for the spanning subgraph is bounded from above by the connectivity number of the original graph. The number entered must be between zero and this value inclusive (in this case the connectivity number of the original graph is 5). Suppose a value of 5 is entered, after which Algorithm *Whitney* is implemented. For this case, no edges could be removed and the user is notified that no cheaper spanning subgraph with connectivity number 5 could be constructed (see Figure 5.10).

Figure 5.10: *A notification window appears informing the user that a cheaper spanning subgraph could not be constructed.*

The process is repeated, this time specifying a desired connectivity number of 4 for the spanning subgraph. Whitney's Algorithm is now able to produce a spanning subgraph with a reduced weight. The adjacency matrix for the constructed spanning subgraph is stored in an Excel [48] file with the same name as the input file, followed by the letters *Output*, *i.e.* the output file is called *testgraphOutput.xls*. The output file is depicted in Figure 5.11.

The weight improvement for the subgraph generated is reflected in the *Output* window. The spanning subgraph generated may be displayed by clicking on the *Display New Graph* button in the *Output Graph* frame box. A minimum cut-set for this graph may also be displayed as yellow vertices by clicking on the *Find Cut-Set* button in the *Output Graph* frame box (see Figure 5.12). Note that the connectivity number of the subgraph that was generated remained 5 — the desired connectivity level of 4 was thus achieved. By iteratively clicking on the *Display New Graph* and *Display Original Graph* buttons, it may be seen that the two edges v_5v_{10} and v_6v_9 are not included in the spanning subgraph generated. The edge weights for these two edges may be read from the adjacency matrix of the original graph, shown

Microsoft Excel - testgraphOutput.xls

	A	B	C	D	E	F	G	H	I	J	K
1	0	2	4	7	1	8	0	2	2	0	0
2	2	0	1	1	5	4	2	0	0	0	0
3	4	1	0	0	0	7	6	5	1	2	0
4	7	1	0	0	0	3	2	1	2	3	0
5	1	5	0	0	0	1	1	2	2	0	0
6	8	4	7	3	1	0	7	3	0	3	1
7	0	2	6	2	1	7	0	1	2	1	3
8	2	0	5	1	2	3	1	0	3	2	1
9	2	0	1	2	2	0	2	3	0	4	1
10	0	0	2	3	0	3	1	2	4	0	5
11	0	0	0	0	0	1	3	1	1	5	0

Whitney(Span Subgr)

Figure 5.11: *The file testgraphOutput.xls, displaying the adjacency matrix that was constructed using Algorithm Whitney. The connectivity number of the spanning subgraph is at least 4.*

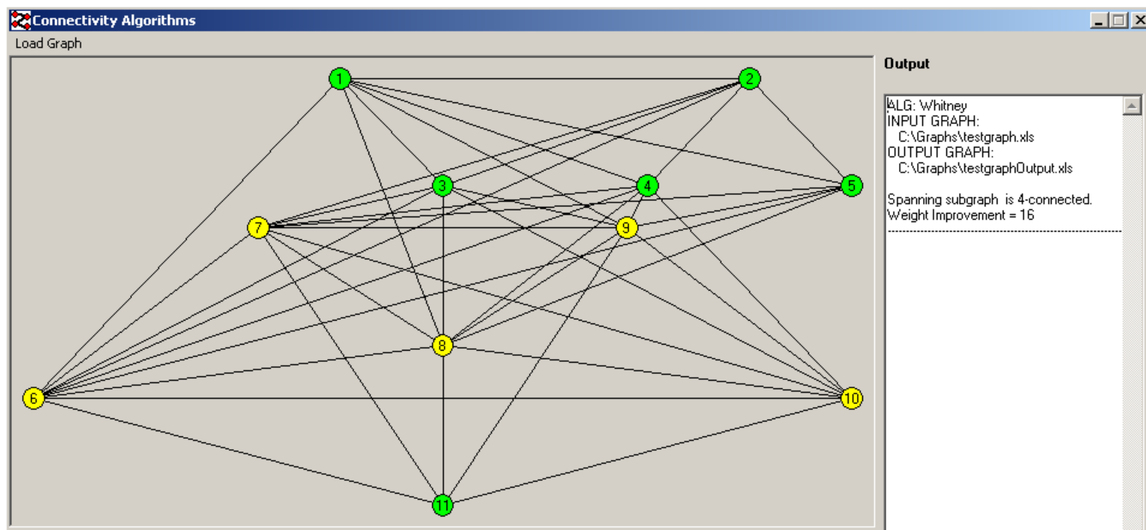


Figure 5.12: *The 4-connected spanning subgraph of G_{34} that was constructed using Algorithm Whitney. A minimum cut-set found by the algorithm is depicted as yellow vertices. A weight improvement of 16 was achieved and is reflected in the Output window.*

in Figure 5.4; edge v_5v_{10} having an edge weight of 7 and edge v_6v_9 having an edge weight of 9, with a combined weight of 16.

Similar spanning subgraphs may be constructed in this fashion using Whitney's Algorithm, with connectivities ranging from 0 to 3. The spanning subgraphs obtained in this fashion, are depicted in the following four figures (Figures 5.13 to 5.16), with connectivities ranging from 3 down to 0 (in descending order). A minimum cut-set for each spanning subgraph is displayed as yellow vertices on each graph. Note that once an output file has been created, the user is prompted by Excel [48] to overwrite the output file when an algorithm is run on the same input graph more than once.

The steps taken to produce spanning subgraphs as described above may be repeated using the *MEEF* and *Fan* algorithms. Note that the *Most Expensive Edge First* and *Fan* frame boxes both contain two buttons (see Figure 5.1). The first button, *Maintain Connectivity Number*, (in both frame boxes) is used to construct a spanning subgraph with the same connectivity number as that of the original graph. The second button, *Lower Connectivity Number*, is used to construct an ℓ-connected spanning subgraph, with ℓ bounded from above by the connectivity number of the original graph. When a user clicks on this button, the window depicted in Figure 5.9 is displayed, prompting the user for the desired level of connectivity.

Stellenbosch University http://scholar.sun.ac.za

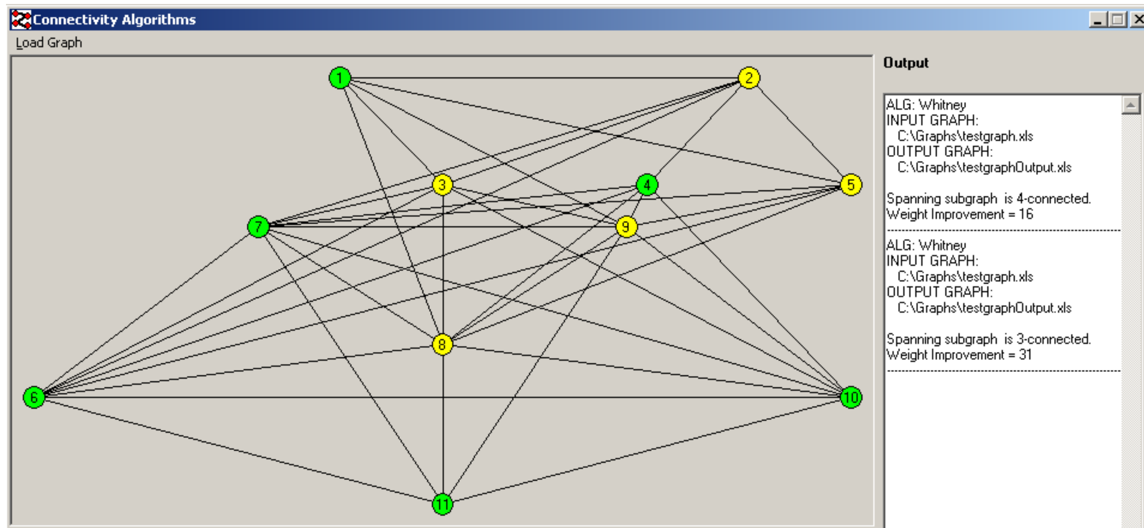


Figure 5.13: *The 3-connected spanning subgraph of G_{34} constructed using Algorithm Whitney. A minimum cut-set found by the algorithm is depicted as yellow vertices. Note that this graph has a connectivity number of 5. A weight improvement of 31 was achieved and is reflected in the Output window.*

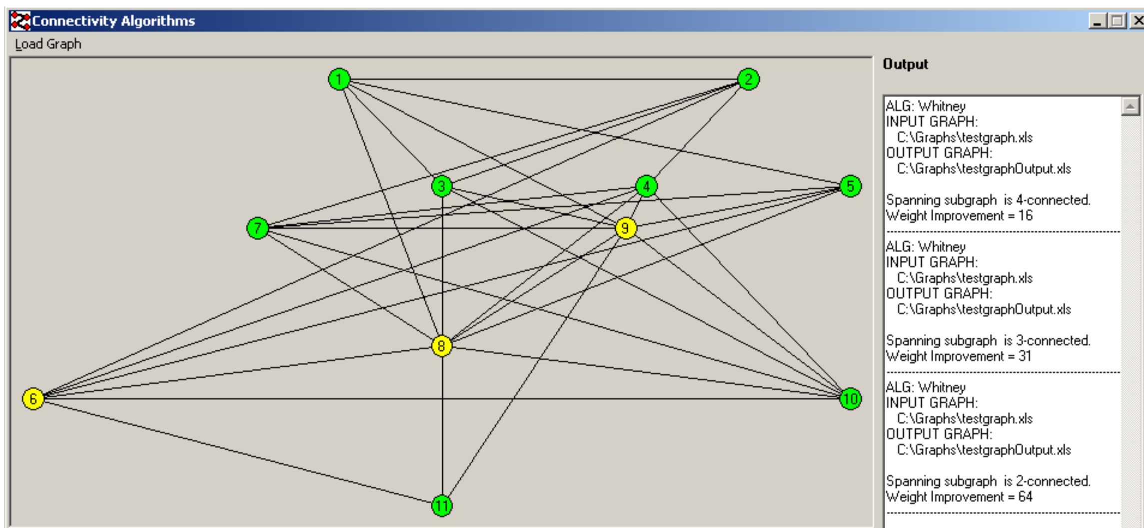


Figure 5.14: *The 2-connected spanning subgraph of G_{34} constructed using Algorithm Whitney. A minimum cut-set found by the algorithm is depicted as yellow vertices. Note that the graph has a connectivity number of 3. A weight improvement of 64 was achieved and is reflected in the Output window.*

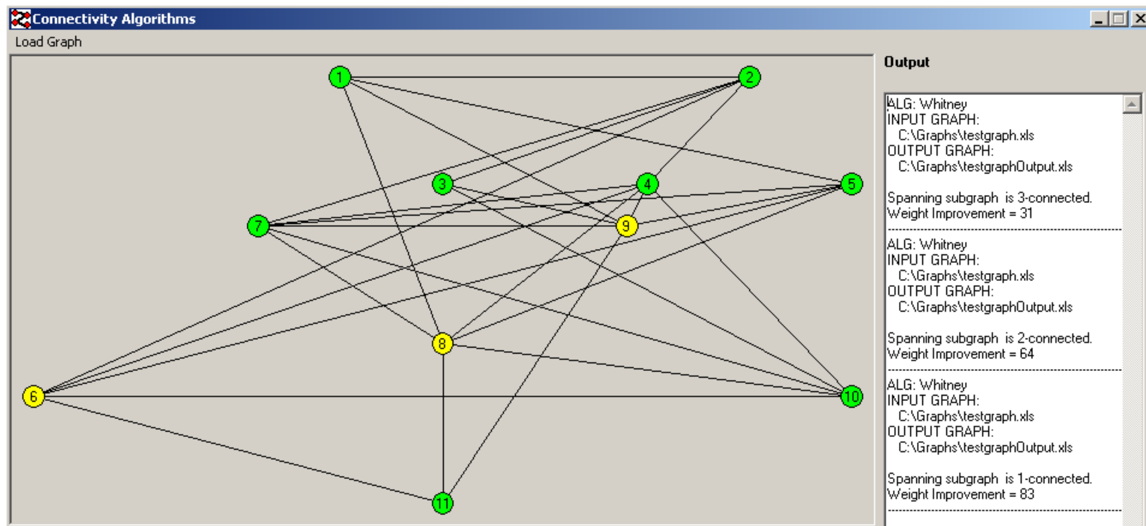


Figure 5.15: *The 1-connected spanning subgraph of G_{34} constructed using Algorithm Whitney. A minimum cut-set found by the algorithm is depicted as yellow vertices. Note that the graph has a connectivity number of 3. A weight improvement of 83 was achieved and is reflected in the Output window.*

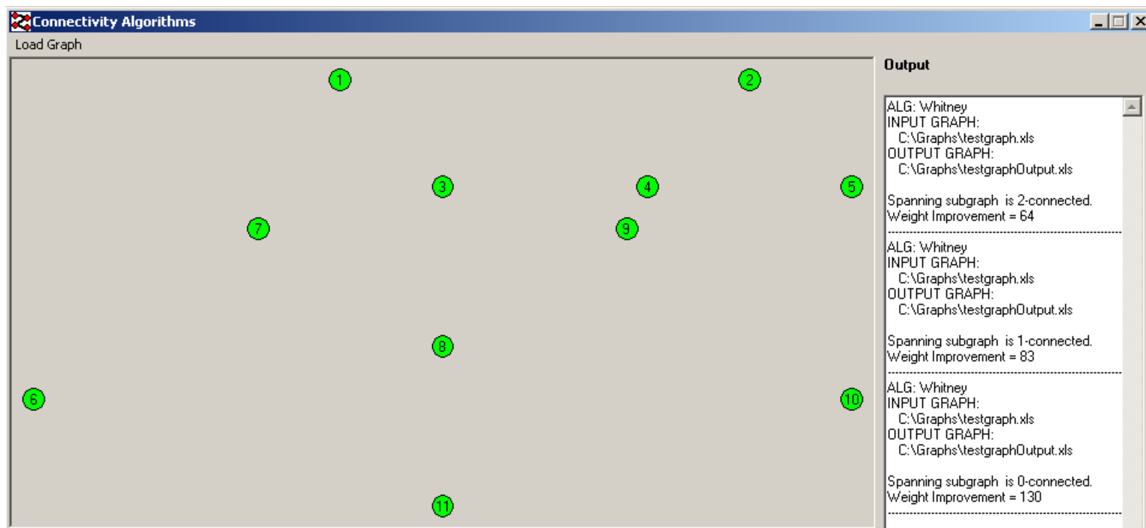


Figure 5.16: *The 0-connected spanning subgraph of G_{34} constructed using Algorithm Whitney. No edges are included in the graph. A weight improvement of 130 was achieved and is reflected in the Output window.*

Stellenbosch University http://scholar.sun.ac.za

5.3.2 Implementation of Algorithm *MEEF*

Figures 5.17 to 5.22 depict the spanning subgraphs of G_{34} generated by the *MEEF* algorithm, with connectivities ranging from 5 down to 0 respectively.

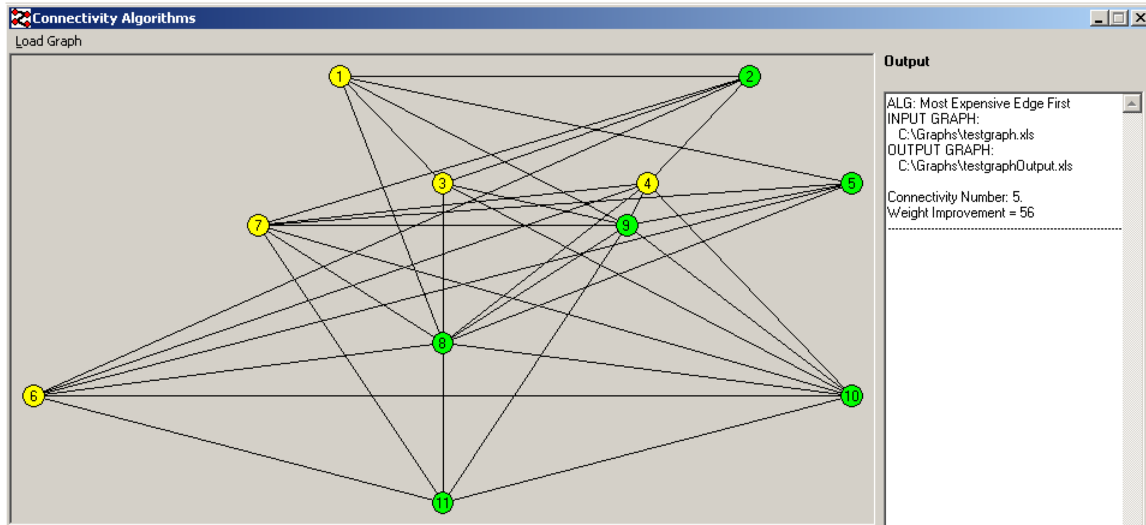


Figure 5.17: *The 5-connected spanning subgraph of G_{34} constructed using Algorithm MEEF. A minimum cut-set found by the algorithm is depicted as yellow vertices. A weight improvement of 56 was achieved and is reflected in the Output window.*

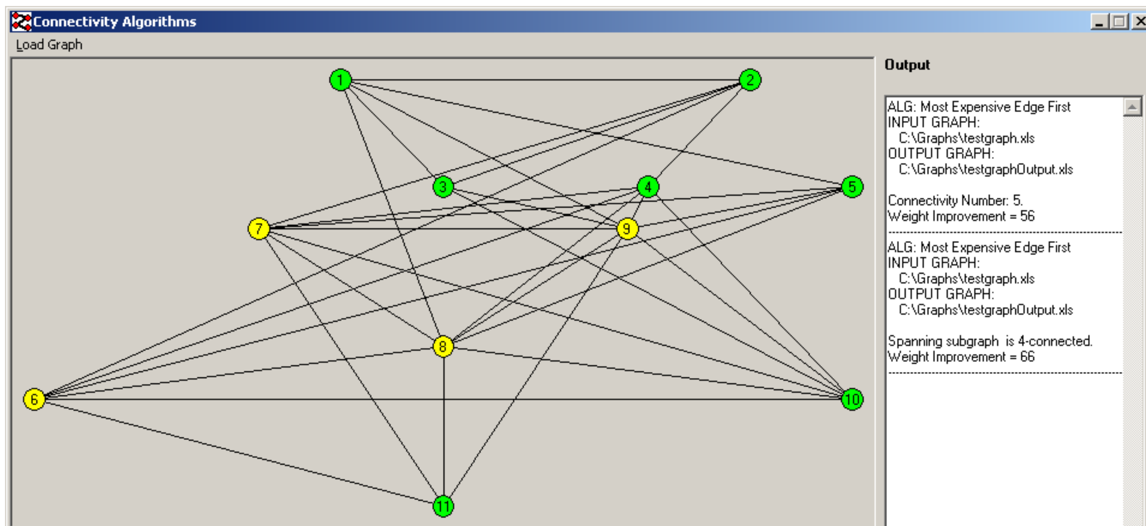


Figure 5.18: *The 4-connected spanning subgraph of G_{34} constructed using Algorithm MEEF. A minimum cut-set found by the algorithm is depicted as yellow vertices. A weight improvement of 61 was achieved and is reflected in the Output window.*

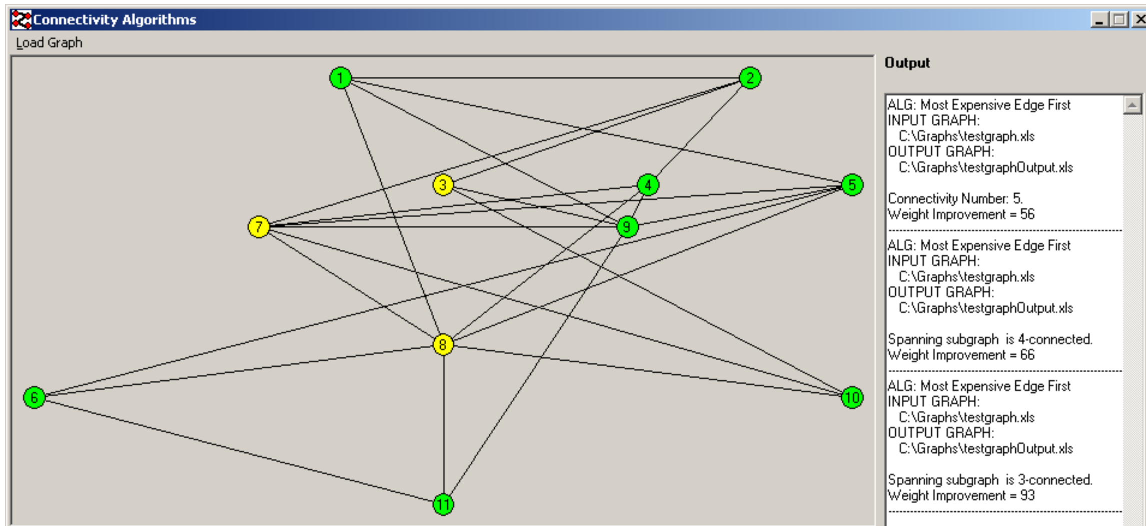


Figure 5.19: *The 3-connected spanning subgraph of G_{34} constructed using Algorithm MEEF. A minimum cut-set found by the algorithm is depicted as yellow vertices. A weight improvement of 70 was achieved and is reflected in the Output window.*

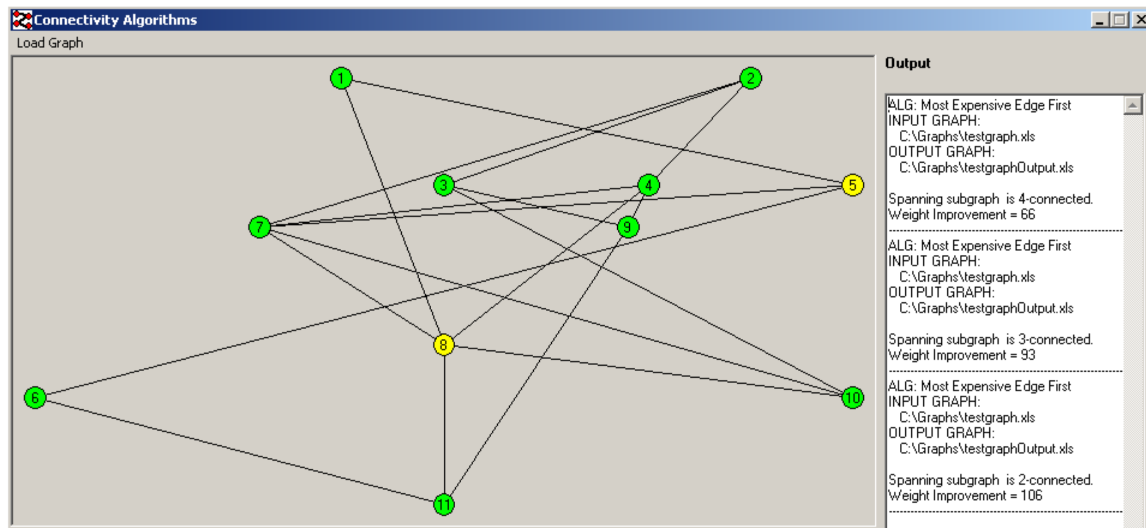


Figure 5.20: *The 2-connected spanning subgraph of G_{34} constructed using Algorithm MEEF. A minimum cut-set found by the algorithm is depicted as yellow vertices. A weight improvement of 96 was achieved and is reflected in the Output window.*

Stellenbosch University http://scholar.sun.ac.za

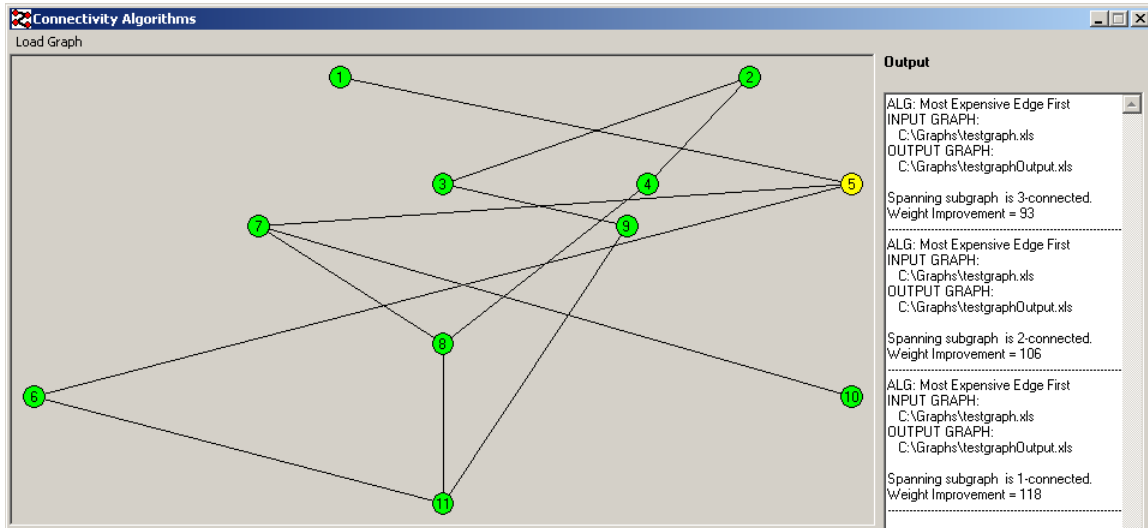


Figure 5.21: *The 1-connected spanning subgraph of G_{34} constructed using Algorithm MEEF. A minimum cut-set found by the algorithm is depicted as yellow vertices. A weight improvement of 108 was achieved and is reflected in the Output window.*

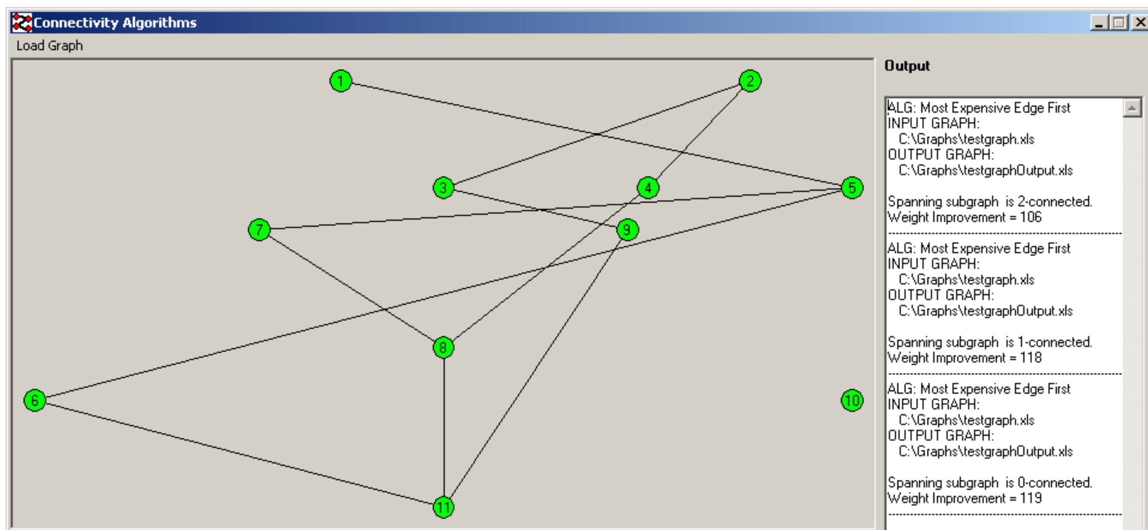


Figure 5.22: *The 0-connected spanning subgraph of G_{34} constructed using Algorithm MEEF. A weight improvement of 119 was achieved and is reflected in the Output window.*

5.3.3 Implementation of Algorithm *Fan*

Figures 5.23 to 5.28 depict the spanning subgraphs of G_{34} generated by the *Fan* algorithm, with connectivities ranging from 5 down to 0 respectively.

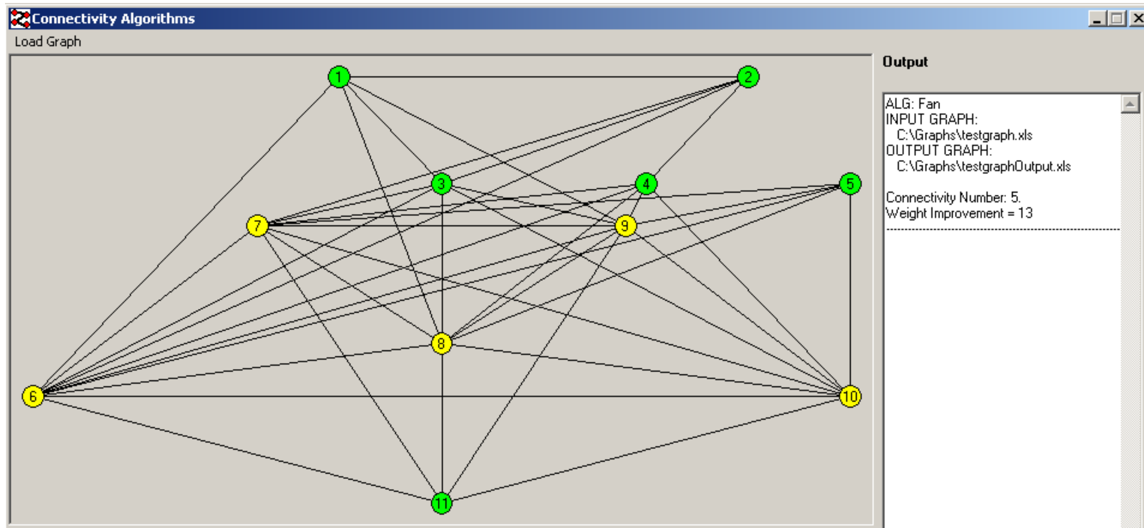


Figure 5.23: *The 5-connected spanning subgraph of G_{34} constructed using Algorithm Fan. A minimum cut-set found by the algorithm is depicted as yellow vertices. A weight improvement of 13 was achieved and is reflected in the Output window.*

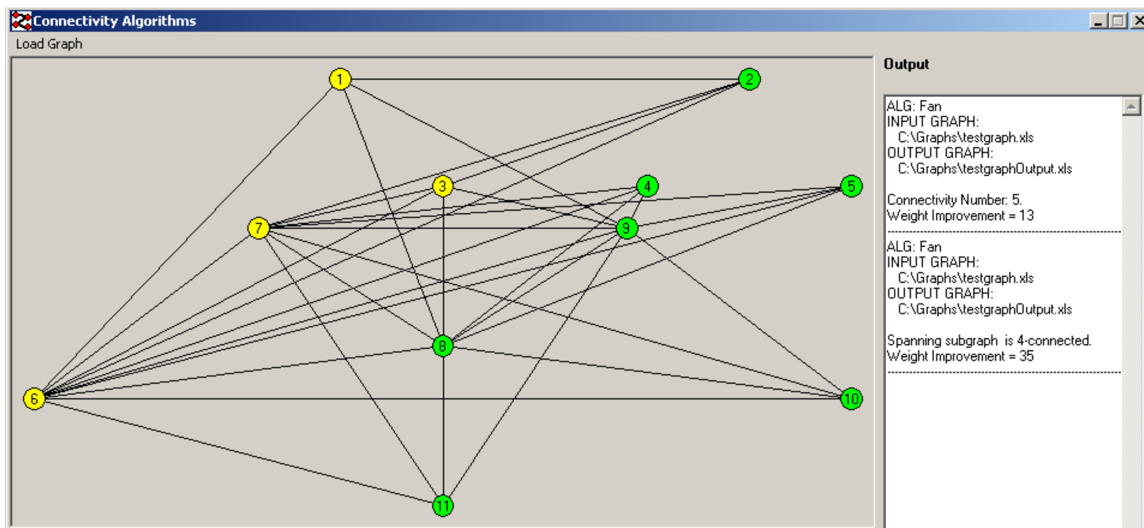


Figure 5.24: *The 4-connected spanning subgraph of G_{34} constructed using Algorithm Fan. A minimum cut-set found by the algorithm is depicted as yellow vertices. A weight improvement of 35 was achieved and is reflected in the Output window.*

Stellenbosch University http://scholar.sun.ac.za

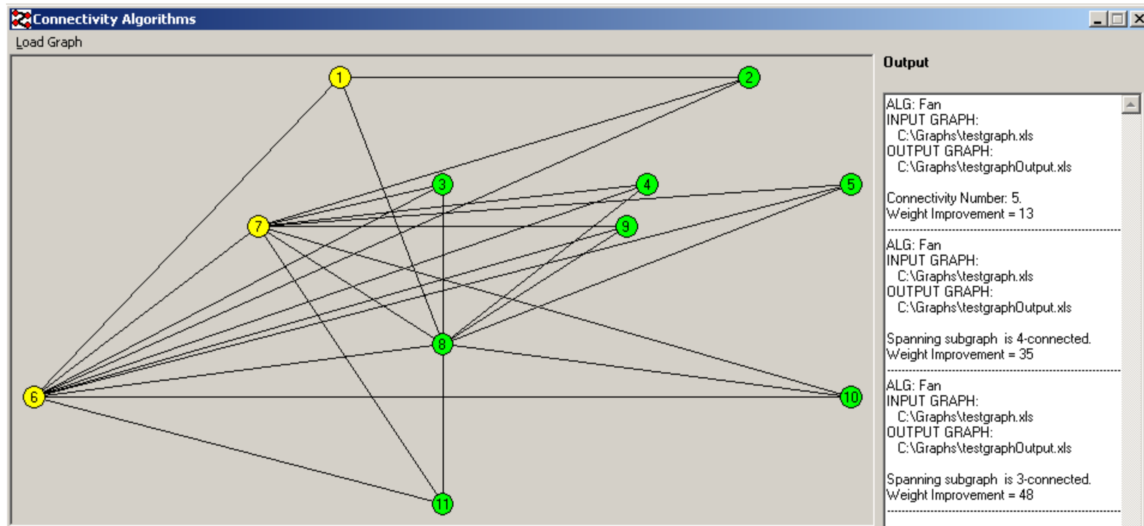


Figure 5.25: *The 3-connected spanning subgraph of G_{34} constructed using Algorithm Fan. A minimum cut-set found by the algorithm is depicted as yellow vertices. A weight improvement of 48 was achieved and is reflected in the Output window.*

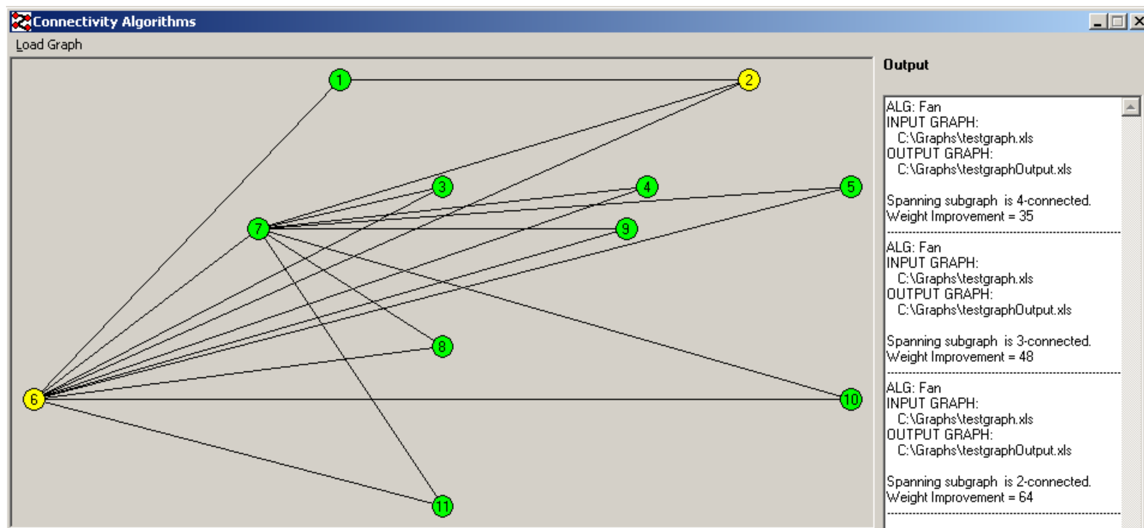


Figure 5.26: *The 2-connected spanning subgraph of G_{34} constructed using Algorithm Fan. A minimum cut-set found by the algorithm is depicted as yellow vertices. A weight improvement of 64 was achieved and is reflected in the Output window.*

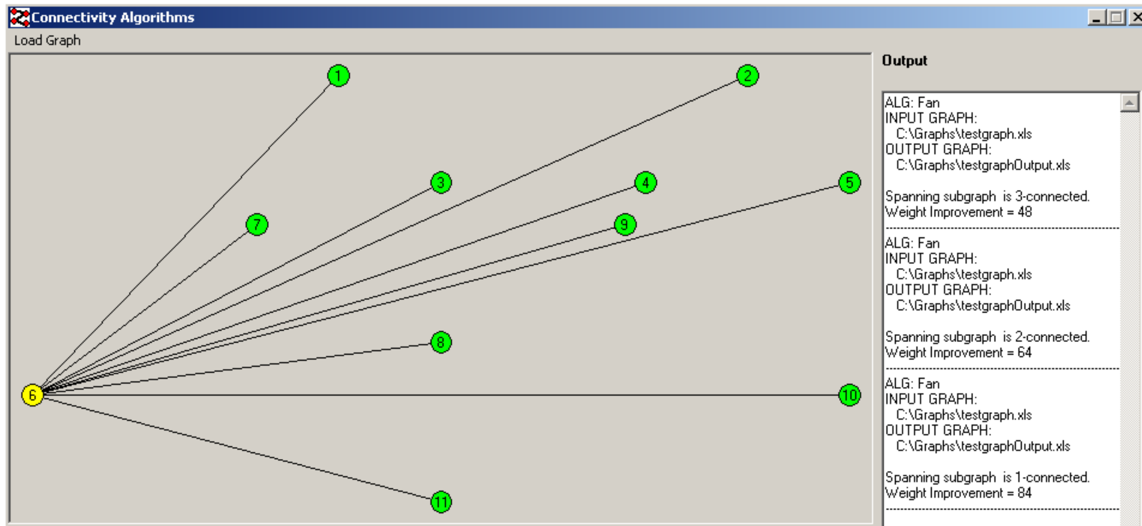


Figure 5.27: *The 1-connected spanning subgraph of G_{34} constructed using Algorithm Fan. A minimum cut-set found by the algorithm is depicted as yellow vertices. A weight improvement of 84 was achieved and is reflected in the Output window.*

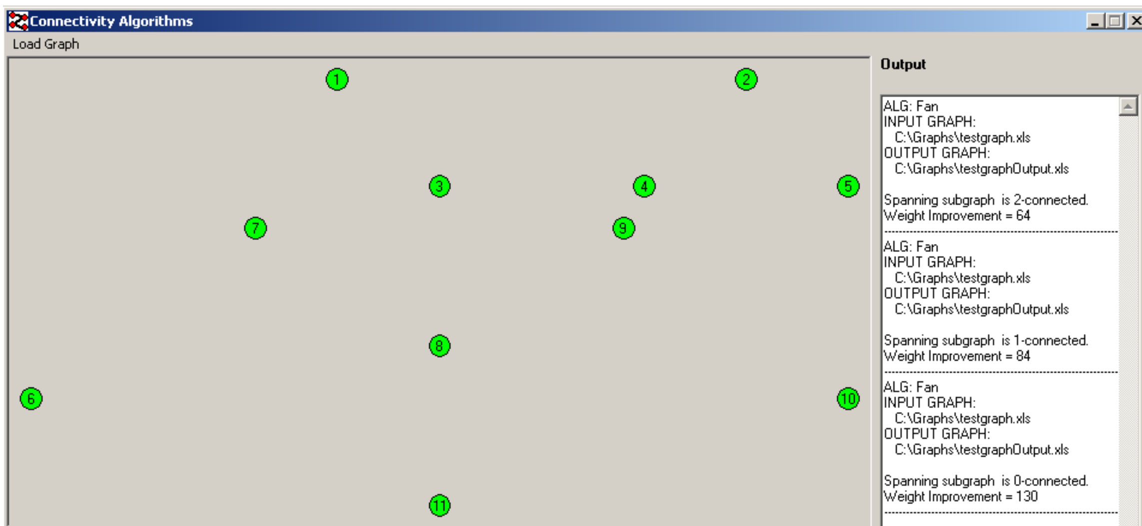


Figure 5.28: *The 0-connected spanning subgraph of G_{34} constructed using Algorithm Fan. A weight improvement of 130 was achieved and is reflected in the Output window.*

Stellenbosch University http://scholar.sun.ac.za

It should be noted that Algorithm *Fan* can only be implemented on a graph that contains a minimum cut-set which also forms a clique. If the input graph does not contain such a minimum cut-set, the user is notified of this fact. All minimum cut-sets are displayed in the *Output* window, in an attempt to aid the user in deciding what edges may be inserted into the graph in order to produce a cut-set that is also a clique. As an example, consider the graph depicted in Figure 5.29.

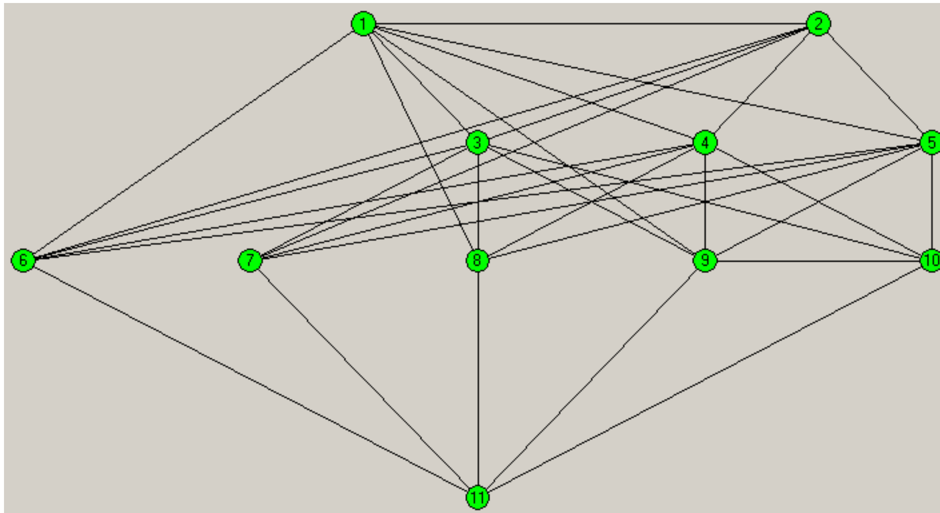


Figure 5.29: *Graphical representation of a graph that does not have a complete cut-set.*

If Algorithm *Fan* is implemented, the message shown in Figure 5.30 is displayed, notifying the user that no complete cut-set exists in the graph.

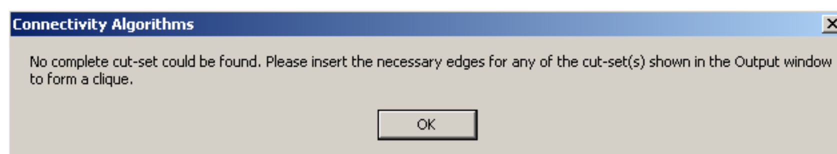


Figure 5.30: *A message is displayed, notifying the user that no complete cut-set exists in the input graph.*

Apart from this, a list of all minimum cut-sets for the graph are displayed in the *Output* window (see Figure 5.31). The user may use this information to insert the necessary edges into the input graph in order to obtain a minimum cut-set that is also a clique.

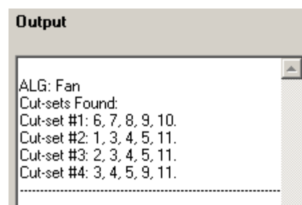


Figure 5.31: *The Output window displays a list of all minimum cut-sets in the input graph. The user may adapt the adjacency matrix of the input graph, using this information to change one of these cut-sets into a complete cut-set.*

Once this is done, the input graph may be loaded again, after which Algorithm *Fan* may be implemented.

5.3.4 Summary of results obtained

A summary of the weight improvements obtained for each k-connected graph, $k = 0, \ldots, 5$, generated by the different algorithms is presented in Table 5.1.

	Algorithm		
Connectivity	*Whitney*	*MEEF*	*Fan*
5	0	56	13
4	16	66	35
3	31	93	48
2	64	106	64
1	83	118	84
0	130	119	130

Table 5.1: *The weight improvement obtained for each k-connected graph, $k = 0, \ldots, 5$, by implementing the algorithms Whitney, MEEF and Fan on the graph G_{34} stored in the file testGraph.xls.*

The weight improvement results of each algorithm may be compared graphically, as shown in Figure 5.32, to determine which algorithm is the most efficient for generating a certain spanning subgraph for the graph at hand.

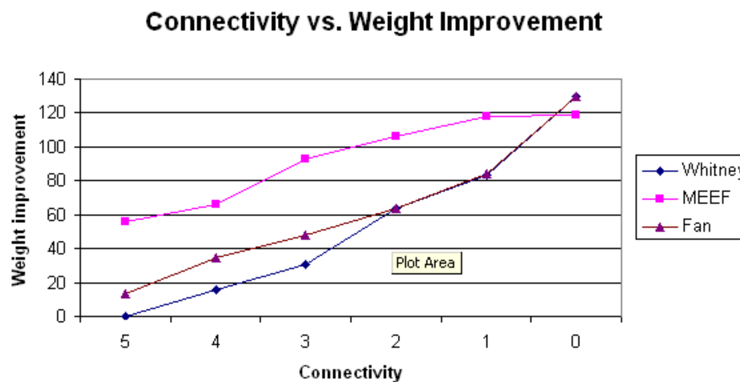


Figure 5.32: *Graphical representation of the k-connectivity level vs. the weight improvement for each spanning subgraph calculated by the various algorithms.*

From Figure 5.32 it may be seen that, for this example, Algorithm *MEEF* produces the best results for 1- to 5-connected spanning subgraphs. Algorithms *Whitney* and *Fan* both produce a maximum weight reduction of 130 for a 0-connected graph. To see which algorithm produces the best results regarding spanning subgraphs with a specific connectivity number, a more careful look should be taken at the graphs that were generated by the DSS. For instance, a 3-connected spanning subgraph that was constructed using Algorithm *Whitney* is depicted in Figure 5.13. A minimum cut-set for this graph consists of 5 vertices, indicating that the connectivity number for this graph is 5. Algorithm *MEEF* always constructs a spanning subgraph for which the desired connectivity level equals the connectivity number of the subgraph. However, Algorithms *Whitney* and *Fan* may construct spanning subgraphs for which the connectivity number is bounded from below by the specified connectivity level. For this example, the connectivity number of some of the spanning subgraphs that were constructed using Algorithm *Whitney* have a higher connectivity number than the desired connectivity level. The weight improvement obtained by each algorithm for a constructed spanning subgraph with a given connectivity number are depicted in Table 5.2, followed by a graphical representation in the form of a bar chart (see Figure 5.33).

From this graph it may be seen that Algorithm *MEEF* still produces the lowest weighted spanning subgraphs for all connectivity numbers, although Algorithm *Whitney* produces only a slightly heavier weighted spanning subgraph with a connectivity number of 3.

Stellenbosch University http://scholar.sun.ac.za

	Algorithm		
$\kappa(G)$	*Whitney*	*MEEF*	*Fan*
5	31	56	13
4		66	35
3	83	93	48
2		106	64
1		118	84
0	130	119	130

Table 5.2: *The weight improvement obtained for each subgraph G constructed, such that $\kappa(G) = k$, $k = 0, \ldots, 5$, by implementing the algorithms Whitney, MEEF and Fan on the graph G_{34} stored in the file testGraph.xls.*

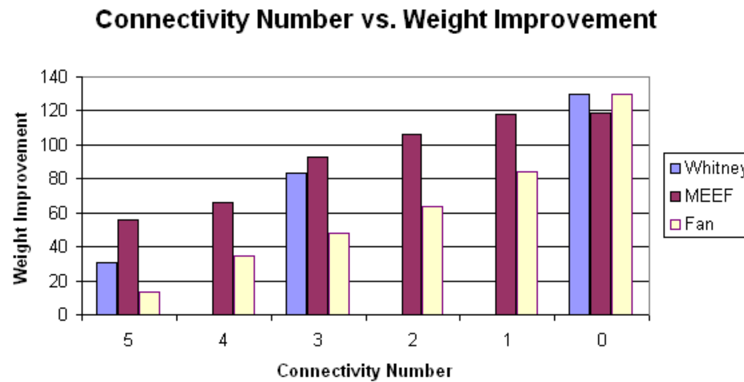


Figure 5.33: *Graphical representation of the connectivity number vs. the weight improvement for each spanning subgraph calculated by the various algorithms.*

5.4 Case study: The connectivity of a Spider's Web

As a small case study, the DSS is used to determine which parts of a spider's web may be removed without reducing its connectivity number too much. A graphical representation of the web used for this case study is presented in Figure 5.34 (file *Spider21.xls* on the CD). Note that the three vertices v_3, v_5 and v_6 is a minimum cut-set that is also a clique.

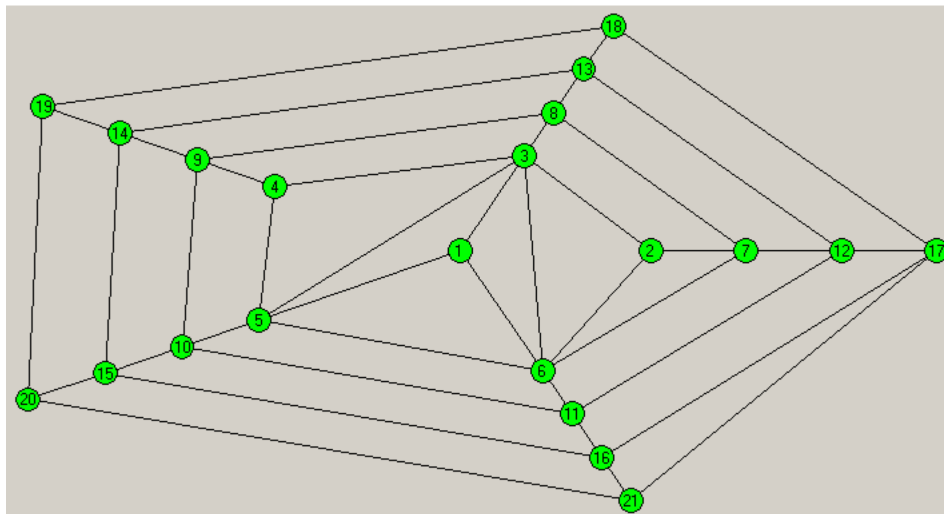


Figure 5.34: *Graphical representation of the file Spider21.xls depicting a spider's web.*

The adjacency matrix for this graph is too large to present graphically, but may be found on the accompanying CD (see the worksheet *AdjacencyMatrix* in the file *Spider21.xls*). The weights of the edges are taken as the Euclidean distance between every pair of adjacent vertices.

Stellenbosch University http://scholar.sun.ac.za

The aim of this case study is to show how the size and weight of a graph can greatly influence the running time of some of the algorithms and their solution qualities. The running time of Algorithm *Fan* is influenced the most by the size of an input graph, due to the high worst-case running time of $O\left(p^{k+3}\right)$ of Algorithm 4, that is used to search for a complete minimum cut-set of the input graph. This causes the first implementation of Algorithm *Fan* to be slow, but subsequent calculations are typically very quick, as a mininimum cut-set only needs to be calculated once for every input graph. For this example, the calculation of all minimum cut-sets took just over 3 minutes on a 3GHz Intel Pentium, a computation time that contrasts starkly with the mere seconds it took to calculate a minimum cut-set for the graph stored in the file *testgraph.xls*, showing the steep rise in running time.

The running time of Algorithms *Whitney* and *MEEF* are less affected by the size of the input graph, as these algorithms only require the connectivity number of the input graph, which may be calculated using the more efficient Algorithm 3. The working of Algorithm *Whitney* itself does not require any knowledge of the cut-set *per se*. However, the implemented version in the DSS does require the connectivity number of the input graph — when the user enters the desired connectivity level for a spanning subgraph (see Figure 5.9), this number must be bounded from above by the connectivity number of the input graph. Algorithm *MEEF*, on the other hand, recalculates the connectivity number of the spanning subgraph for every iteration in the algorithm where an edge or set of edges are removed.

An implementation of Algorithm *Whitney* was unable to produce a spanning subgraph with fewer edges than the input graph, for connectivity numbers ranging from 1 to 3. For a connectivity number of zero, Algorithm *Whitney* removed all of the edges in the input graph. Each attempt to construct a spanning subgraph with connectivity number ranging from 1 to 3 took 5 seconds to complete. A spanning subgraph with a connectivity number of 0 can be constructed almost immediately, as no paths are required to be calculated and hence, all edges are simply omitted.

An implementation of Algorithm *MEEF* was also unable to produce a 3-connected spanning subgraph with fewer edges than that of the input graph. However, a 2-connected spanning subgraph could be constructed in less than one second. The subgraph corresponds to a weight improvement of 3.351, the weight of the removed edge $v_{20}v_{21}$. A graphical representation of the spanning subgraph is depicted in Figure 5.35.

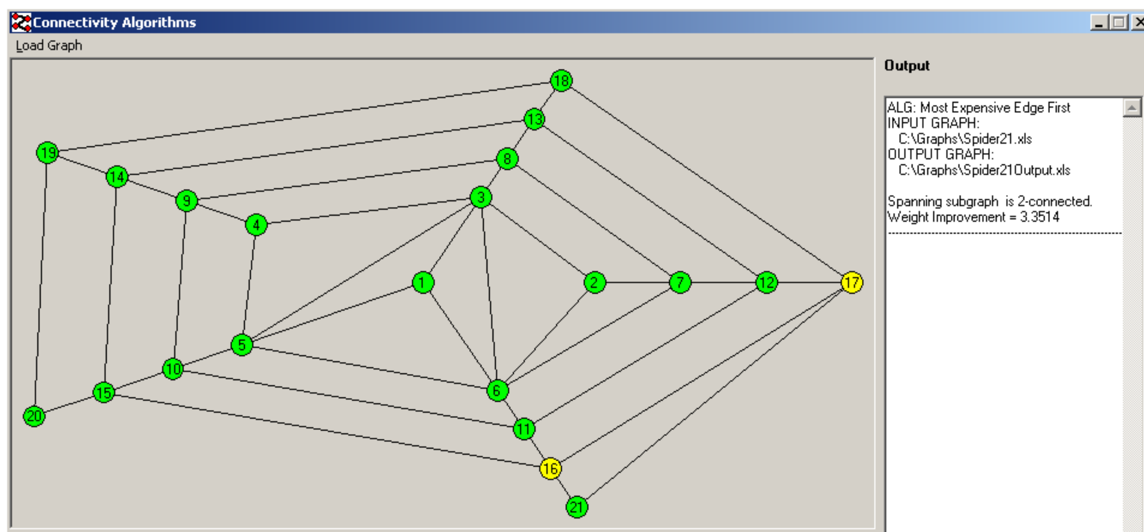


Figure 5.35: *The 2-connected spanning subgraph constructed using Algorithm MEEF. A minimum cut-set found by the algorithm is depicted as yellow vertices. A weight improvement of 3.351 (the weight of the edge $v_{20}v_{21}$) was achieved and is reflected in the Output window; hence only one edge was removed. Note that the graph has a connectivity number of 2.*

Algorithm *MEEF* could also be used to construct a 1-connected spanning subgraph. This operation took 3 seconds to complete. The subgraph corresponds to a weight improvement of 50.294, the combined weight of the 21 removed edges v_2v_6, v_3v_6, v_3v_5, v_4v_5, v_5v_6, v_6v_7, v_7v_8, v_8v_9, v_9v_{10}, $v_{10}v_{11}$, $v_{11}v_{12}$, $v_{12}v_{13}$, $v_{13}v_{14}$, $v_{14}v_{15}$, $v_{15}v_{16}$, $v_{16}v_{17}$, $v_{17}v_{18}$, $v_{18}v_{19}$, $v_{19}v_{20}$, $v_{20}v_{21}$ and $v_{21}v_{17}$. A graphical representation of the spanning subgraph is depicted in Figure 5.36.

Stellenbosch University http://scholar.sun.ac.za

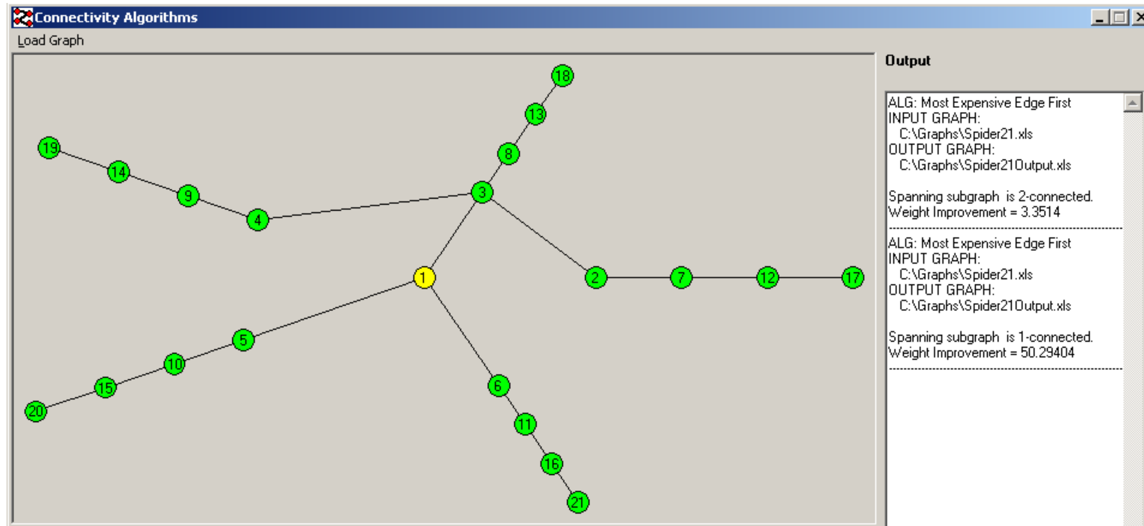


Figure 5.36: *The 1-connected spanning subgraph constructed using Algorithm MEEF. A minimum cut-set found by the algorithm is depicted as yellow vertices. A weight improvement of 50.294 was achieved and is reflected in the Output window. Note that the graph has a connectivity number of 1.*

The construction of a 0-connected spanning subgraph using Algorithm *MEEF* took 3 seconds. The subgraph corresponds to a weight improvement of 51.648. The same 21 edges that were removed when constructing a 1-connected spanning subgraph are removed, as well as the additional edge v_3v_4. A graphical representation of the spanning subgraph is depicted in Figure 5.37.

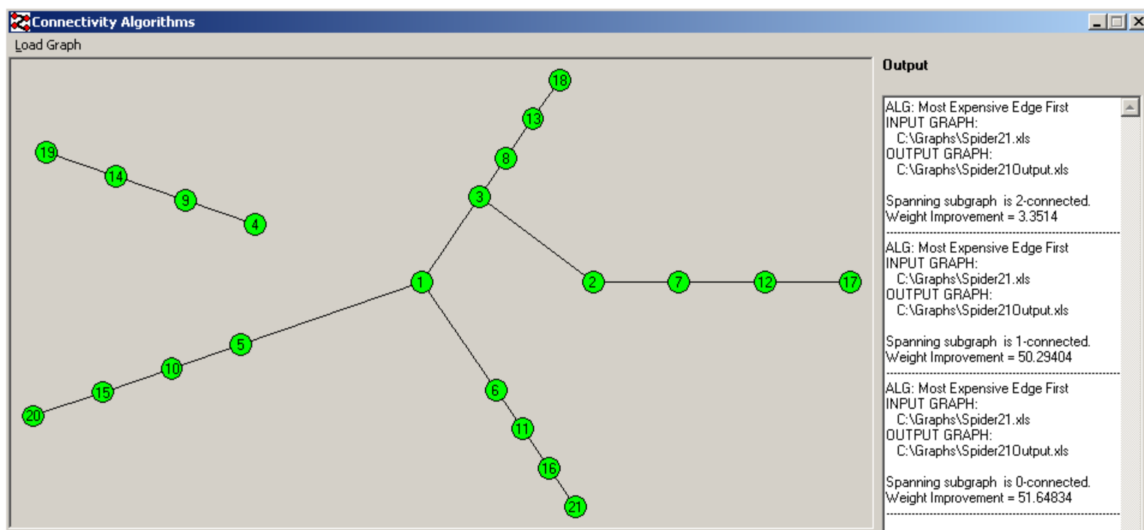


Figure 5.37: *The 0-connected spanning subgraph constructed using Algorithm MEEF. A weight improvement of 51.648 was achieved and is reflected in the Output window.*

To implement Algorithm *Fan*, a complete minimum cut-set must first be calculated. This operation took just over 3 minutes. The implementation of Algorithm *Fan* was unable to produce a 3-connected spanning subgraph with fewer edges than that of the input graph. However, a 2-connected spanning subgraph could be constructed in less than a second. The subgraph has a weight improvement of 5.154, which is the combined weight of the removed edges v_1v_6, v_2v_6, v_6v_7, v_6v_{11} and $v_{16}v_{21}$. A graphical representation of the spanning subgraph is depicted in Figure 5.38.

Algorithm *Fan* produced relatively good results when it is applied to obtain a 1-connected graph (the actual operation again took less than a second) — a weight improvement of 28.917 was achieved. A graphical representation of the spanning subgraph is depicted in Figure 5.39.

Similar to Algorithm *Whitney*, Algorithm *Fan* could construct a 0-connected graph by removing all of the edges in the original graph (this graph is not shown).

Stellenbosch University http://scholar.sun.ac.za

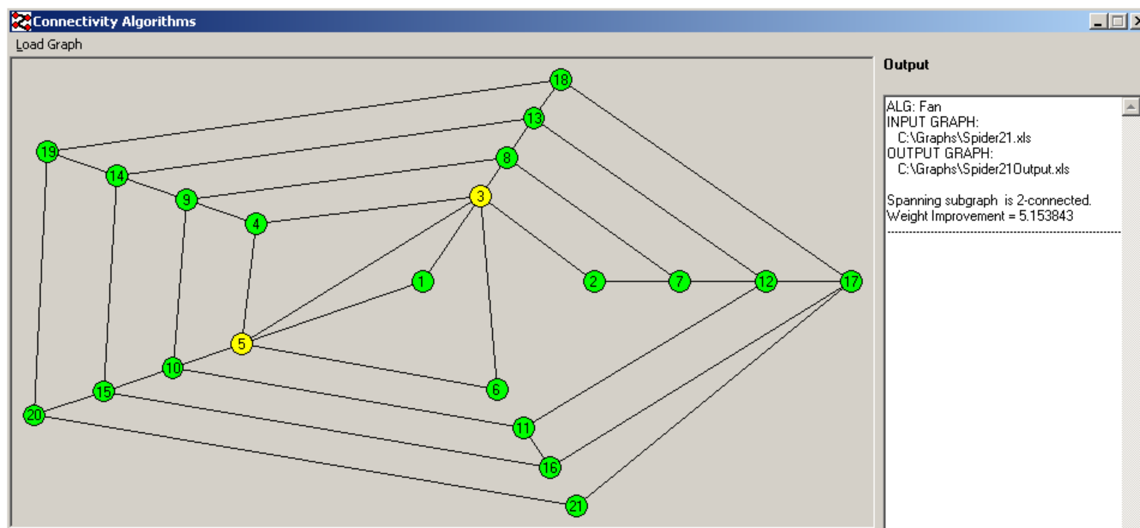


Figure 5.38: *The 2-connected spanning subgraph constructed using Algorithm Fan. A minimum cut-set found by the algorithm is depicted as yellow vertices. A weight improvement of 5.154 was achieved and is reflected in the Output window. Note that the graph has a connectivity number of 2.*

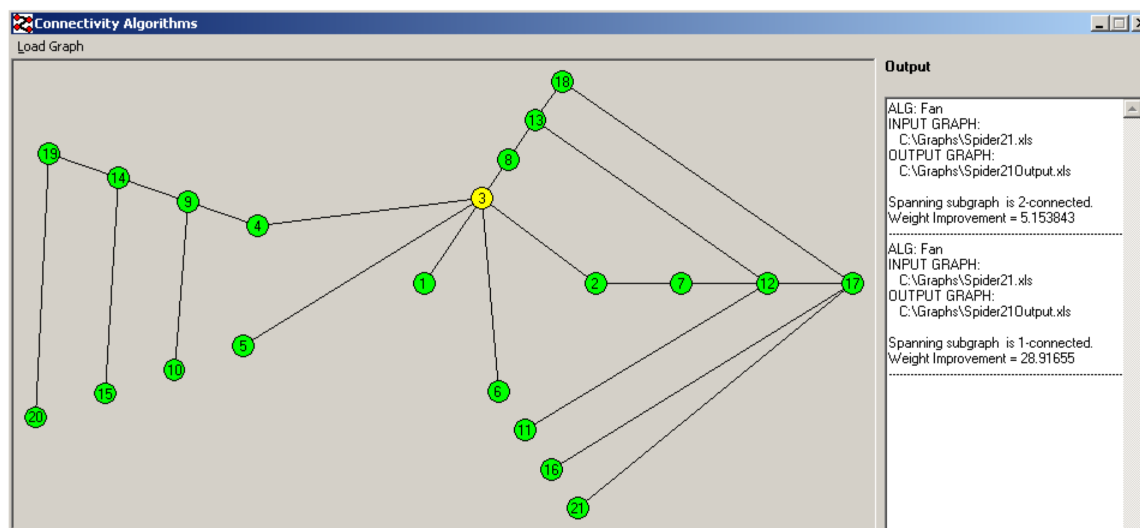


Figure 5.39: *The 1-connected spanning subgraph constructed using Algorithm Fan. A minimum cut-set found by the algorithm is depicted as yellow vertices. A weight improvement of 28.917 was achieved and is reflected in the Output window. Note that the graph has a connectivity number of 1.*

Thus far, the solution qualities produced by Algorithm *MEEF* were the best, when compared to the other two algorithms. For this example, the running time of Algorithm *MEEF* is barely slower than that of Algorithm *Fan* (after a complete minimum cut-set has been calculated), making it a good candidate for implementation on similar types of graphs.

However, there are cases where Algorithm *Fan* outperforms Algorithm *MEEF* in terms of both running time and solution qualities. In particular, graphs where the edge weights are either all the same, or graphs where edges that exist near each other in a graph have larger weights compared to the rest of the graph, usually produce very inferior solution qualities when using Algorithm *MEEF*. To illustrate this, consider again the spider's web as illustrated above, this time where all edges have a weighting of one. The adjacency matrix for this graph is stored in the file *spider21_2.xls*.

An implementation of Algorithm *MEEF* yields results for 0- to 3-connected spanning subgraphs. A 3-connected spanning subgraph could be constructed in 2 seconds. The subgraph corresponds to a weight improvement of 2, which is the combined weight of the removed edges $v_{11}v_{16}$ and $v_{12}v_{13}$. A graphical representation of the spanning subgraph is depicted in Figure 5.40.

A 2-connected spanning subgraph could be constructed in 2 seconds. The subgraph corresponds to a

Stellenbosch University http://scholar.sun.ac.za

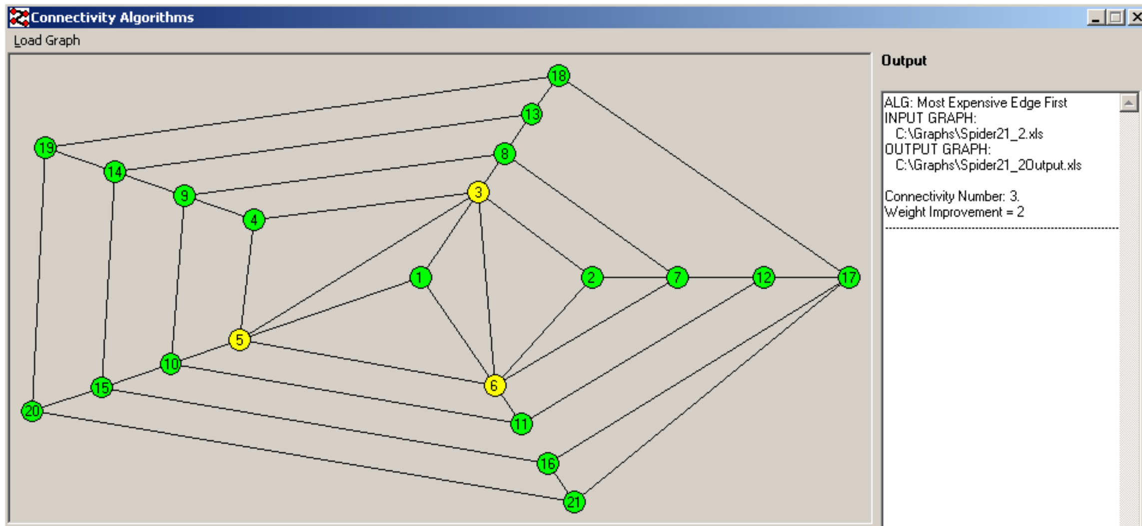


Figure 5.40: *The 3-connected spanning subgraph constructed using Algorithm MEEF. A minimum cut-set found by the algorithm is depicted as yellow vertices. A weight improvement of 2 (the weight of the edge $v_{20}v_{21}$) was achieved and is reflected in the Output window; hence only one edge was removed. Note that the graph has a connectivity number of 2.*

weight improvement of 4, the combined weight of the removed edges $v_{11}v_{16}$, $v_{12}v_{13}$, $v_{12}v_{17}$ and $v_{13}v_{14}$. A graphical representation of the spanning subgraph is depicted in Figure 5.41.

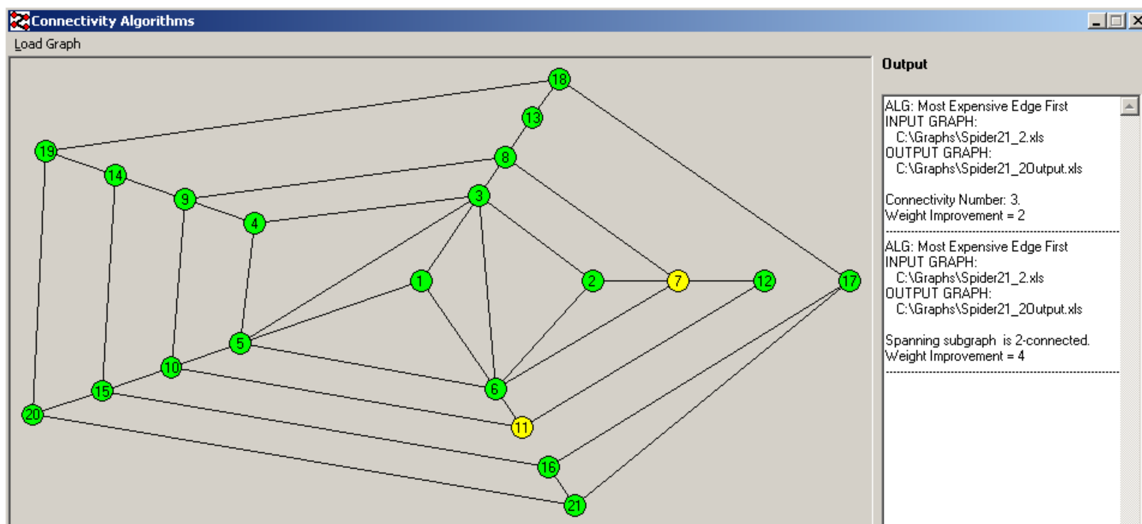


Figure 5.41: *The 2-connected spanning subgraph constructed using Algorithm MEEF. A minimum cut-set found by the algorithm is depicted as yellow vertices. A weight improvement 4 was achieved and is reflected in the Output window; hence only one edge was removed. Note that the graph has a connectivity number of 2.*

Algorithm *MEEF* could also be used to construct a 1-connected spanning subgraph. This operation took 3 seconds to complete. The subgraph corresponds to a weight improvement of 8, the combined weight of the removed edges v_4v_9, $v_{10}v_{11}$, $v_{11}v_{16}$, $v_{11}v_{12}$, $v_{12}v_{13}$, $v_{12}v_{17}$, $v_{13}v_{14}$ and $v_{13}v_{18}$. A graphical representation of the spanning subgraph is depicted in Figure 5.42.

The construction of a 0-connected spanning subgraph using Algorithm *MEEF* took 3 seconds. The subgraph corresponds to a weight improvement of 9. The same 8 edges that were removed when constructing a 1-connected spanning subgraph is removed, as well as the additional edge $v_{10}v_{15}$. A graphical representation of the spanning subgraph is depicted in Figure 5.43.

An implementation of Algorithm *Fan* was also used to construct 0- to 3-connected spanning subgraphs. The calculation of a minimum cut-set again took approximately 3.5 minutes. The implementation of Algorithm *Fan* was unable to produce a 3-connected spanning subgraph with fewer edges than that of

Stellenbosch University http://scholar.sun.ac.za

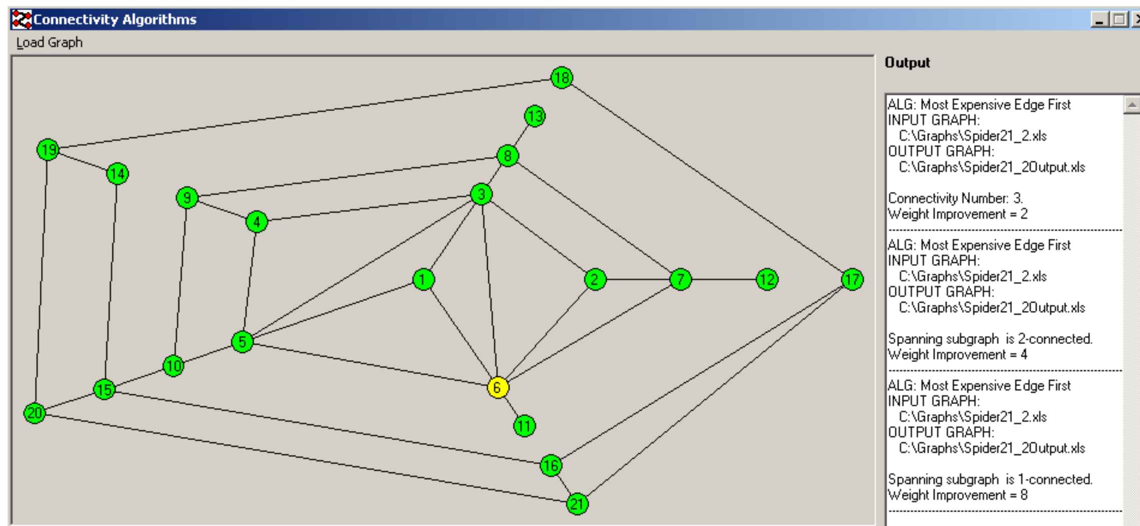


Figure 5.42: *The 1-connected spanning subgraph constructed using Algorithm MEEF. A minimum cut-set found by the algorithm is depicted as yellow vertices. A weight improvement of 50.294 was achieved and is reflected in the Output window. Note that the graph has a connectivity number of 1.*

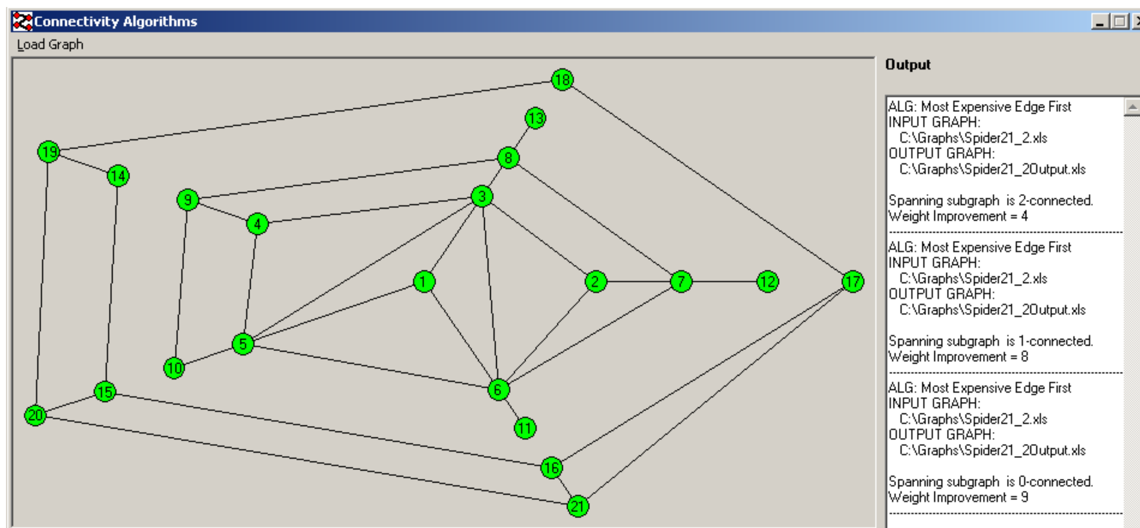


Figure 5.43: *The 0-connected spanning subgraph constructed using Algorithm MEEF. A weight improvement of 51.648 was achieved and is reflected in the Output window.*

the input graph. However, a 2-connected spanning subgraph could be constructed in less than a second. The subgraph represents a weight improvement of 5, which is the combined weight of the removed edges v_1v_6, v_2v_6, v_6v_7, v_6v_{11} and $v_{16}v_{21}$. A graphical representation of the spanning subgraph is depicted in Figure 5.44.

Algorithm *Fan* obtained a 1-connected graph (the actual operation again took less than a second) with a weight improvement of 19, which is the combined weight of the removed edges v_1v_5, v_1v_6, v_2v_6, v_4v_5, v_5v_6, v_5v_{10}, v_6v_7, v_6v_{11}, v_7v_8, v_8v_9, $v_{10}v_{11}$, $v_{10}v_{15}$, $v_{11}v_{16}$, $v_{13}v_{14}$, $v_{15}v_{16}$, $v_{15}v_{20}$, $v_{16}v_{21}$, $v_{18}v_{19}$ and $v_{20}v_{21}$. A graphical representation of the spanning subgraph is depicted in Figure 5.45.

Algorithm *Fan* could also, as mentioned before, construct a 0-connected graph by removing all of the edges in the original graph (this graph is not shown).

It is clear from the results above that, for this example, Algorithm *Fan* outperformed Algorithm *MEEF* in terms of both the running time and the weight reductions obtained for the various cases. Note that Algorithm *MEEF* produces even worse solution qualities if the input graph consists of a number of edges with considerably higher weighting than edges in the rest of the graph, that are located close to each other or even possibly share a vertex. It is clear that these edges will be removed first, hence

Stellenbosch University http://scholar.sun.ac.za

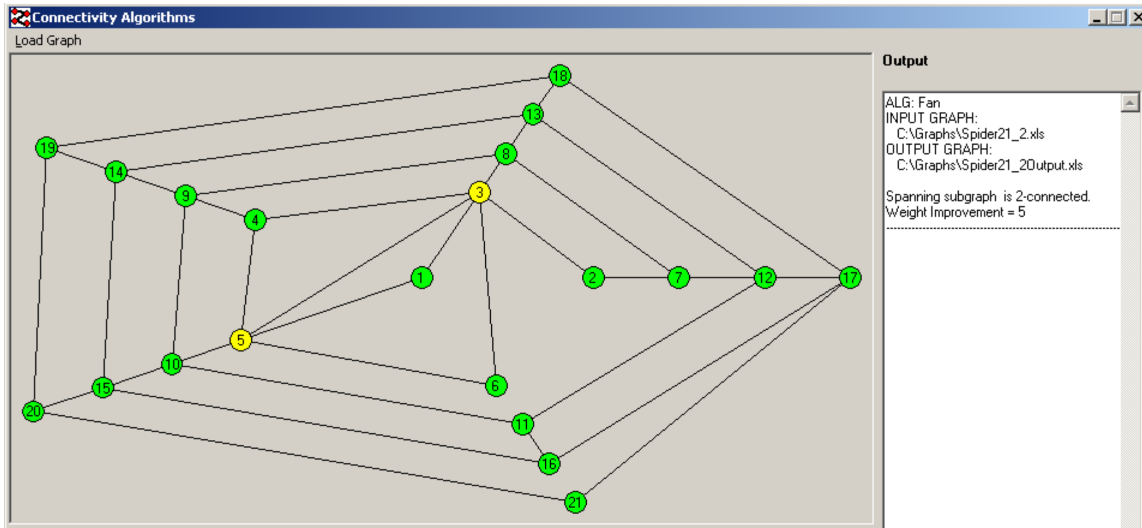


Figure 5.44: *The 2-connected spanning subgraph constructed using Algorithm Fan. A minimum cut-set found by the algorithm is depicted as yellow vertices. A weight improvement of 5.154 was achieved and is reflected in the Output window. Note that the graph has a connectivity number of 2.*

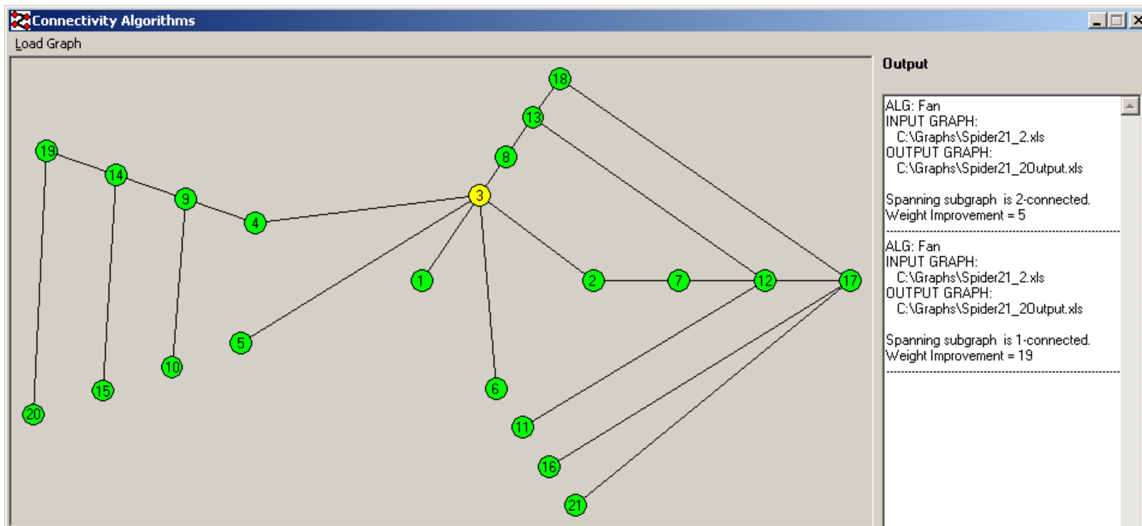


Figure 5.45: *The 1-connected spanning subgraph constructed using Algorithm Fan. A minimum cut-set found by the algorithm is depicted as yellow vertices. A weight improvement of 28.917 was achieved and is reflected in the Output window. Note that the graph has a connectivity number of 1.*

quickly reducing the connectivity number of the graph, but succeeding in removing only a few edges. For example, consider again a spider's web (as above) with equal edge weights of one for each edge, except, set the weights of the edges $v_{18}v_{13}$, $v_{18}v_{17}$ and $v_{18}v_{19}$ to 2. If Algorithm *MEEF* is implemented on this graph, say for instance, to obtain a spanning subgraph with a connectivity number of 1, then only two of these three edges are removed. An implementation of Algorithm *Fan*, on the other hand, removes many more edges, resulting in a lower weighted spanning subgraph.

It should be noted that other methods exist for constructing minimum-cost spanning subgraphs with a connectivity number of one or zero. For a connectivity number of zero, all edges may simply be removed. To construct a minimum-cost spanning subgraph with a connectivity number of one, any minimum-cost spanning tree algorithm may be implemented (see for instance Kruskal's algorithm [12, pp.567–570] or Prim's algorithm [12, pp.570–573]). However, these calculations are done with Algorithms *Whitney*, *MEEF* and *Fan*, to further illustrate the working of the algorithms. These results may then be compared to the (known) optimal solutions obtained by removing all edges (for a connectivity number of zero), or implementing the minimum-cost spanning tree algorithm (for a connectivity number of 1).

To conclude, a few remarks about the design of the spider's web are in order. It is clear that the original spider's web, depicted in Figure 5.34, has a connectivity number of 3. As the outer vertices all (except

vertex v_{17}) have a degree of 3, the removal of the three edges joined to any specific outer vertex (except vertex v_{17}) produces a spanning subgraph with a connectivity number of 0. In this sense, the spider's web does not seem to be very strong, but all other inner vertices, with the exception of the vertices v_1, v_2 and v_4, have a degree of 4 or more, making it slightly harder to break the inner part of the spider's web (*i.e.* to separate a vertex or set of vertices in the inner part of the graph from the rest of the graph). No vertex has a very high degree, which indicates an efficient design in terms of the amount of web used (or the total weight of the graph); hence the web is 3-connected, but far from being 4-connected. This implies that, a large amount of additional silk would be required to increase the web's connectivity number to 4. At the other end of the scale, comparing Figures 5.39 and 5.34, it may be seen how much additional silk was required just to increase the connectivity number of the web from 1 (see Figure 5.39) to 3 (see Figure 5.34).

5.5 Chapter Summary

In this chapter, the DSS *Connectivity Algorithms*, was discussed in some detail. A technical description of the DSS was given, followed by a description of its various components. A complete worked example was given to further introduce the user to the system. This was followed by a short analysis of the output obtained by the three algorithms that are implemented in the system. A case study on the connectivity of a spider's web was also given. The main aim of this case study was to compare the running times of the different algorithms. As expected, Algorithm *Fan* produced the fastest running times, once a complete minimum cut-set was found. For the spider's web with varying edge weights (see the adjacency matrix stored in the file *spider21.xls*), Algorithm *MEEF* produced the lowest weighted spanning subgraphs for constructed graphs with connectivity numbers of 1 and 2. Algorithms *Whitney* and *Fan* produced minimum weighted 0-connected graphs by removing all edges of the input graph. A spider's web with equal edge weights was also analysed (see the adjacency matrix stored in the file *spider21_2.xls*) which illustrated conditions under which Algorithm *Fan* will outperform Algorithm *MEEF* in terms of producing lower weighted spanning subgraphs.

Chapter 6

Conclusion

This final chapter comprises two sections. In §6.1 the work contained in this thesis is summarised, and in §6.2 various avenues of further research are highlighted.

6.1 Thesis Summary

In Chapter 2 an overview of basic prerequisites from graph and complexity theory was given, thereby achieving thesis Objective I, as listed in §1.3. A concise survey of literature relevant to graph connectivity was presented in Chapter 3, in fulfilment of Objective II.

Chapter 4 opened with a discussion on two algorithms that may be used for calculating the connectivity number of a graph (see §4.1 and 4.2), in partial fulfilment of Objective III (Algorithms 3 and 4 are used by Algorithms *Whitney*, *MEEF* and *Fan* for calculating the connectivity number of a minimum cut-set of a graph). Algorithm 3 has a much better worst-case running time than Algorithm 4, but the latter algorithm has the advantage of also finding a minimum cut-set for the graph. These algorithms are used by the three algorithms *Whitney*, *MEEF* and *Fan*, which are capable of constructing lower weighted connectivity preserving or reducing spanning subgraphs of a given graph (see §4.4, 4.5 and 4.6). Objectives IV and V have thus been achieved in these sections. Theoretical results supporting these algorithms were developed and illustrated by means of examples, rounding off the requirement stipulated by Objective III. The chapter closes with a comparison of the various algorithms developed in this thesis (see §4.7), in partial fulfilment of Objective VI. A comparison was also made between the algorithms in this thesis and the very recent *K-N* algorithm by Kortsarz and Nutov [34]. Algorithm *Fan* proved to be very competetive in terms of its worst-case running time for constructing connectivity preserving spanning subgraphs. However, it should be noted that the worst-case running time of the *K-N* algorithm had to be generalised in order to compare it to that of Algorithm *Fan*. With this generalisation, Algorithm *Fan* is an order of magnitude faster than the *K-N* algorithm. Algorithm *Fan* may also be used for constructing connectivity reducing spanning subgraphs in near-linear worst-case running time. A drawback of the *K-N* algorithm is that it cannot construct such spanning subgraphs.

A decision support system (DSS), called *Connectivity Algorithms*, was developed to implement the algorithms of this thesis using Microsoft Visual Basic 6. This DSS is described in detail in Chapter 5, achieving Objective VII. A technical description of the DSS is given in §5.1. The DSS suffers the drawback that the input graph may not consist of more than 31 vertices. This is due to a physical limitation of the data types used in Visual Basic 6, but this limitation can be eliminated, if required. The working of the DSS is explained in §5.2. The DSS allows the user to enter any graph by means of an adjacency matrix stored in an Excel file, after which any of the three algorithms, *Whitney*, *MEEF* or *Fan* may be implemented to construct spanning subgraphs satisfying certain connectivity requirements. The connectivity number selected by the user may range from zero to $\kappa(G)$, the connectivity number of the original graph G. The results of the various algorithms were compared numerically, rounding off the requirements stipulated by Objective VI. In §5.3, a complete worked example was given in order to familiarise the

reader with the different components of the DSS. The chapter closed with a short case study on the strength of a spider's web (see §5.4). Algorithms *MEEF* and *Fan* produced the best results in terms of computation time and the total weight of the spanning subgraph produced.

6.2 Future Work

In Chapters 4 and 5 it became clear that the worst-case running time of Algorithm 4 constitutes a bottleneck in the worst-case running time of Algorithm *Fan*. No other algorithm capable of finding a minimum cut-set in a graph is known to the author. More time-efficient methods for the computation of a minimum cut-set in a graph should be investigated.

The total weight improvement of the spanning subgraph produced by using Algorithm *Fan* may be increased if a method can be developed by choosing the ℓ internally disjoint paths with the lowest weight to insert from every vertex not in the selected minimum cut-set, to that cut-set. The current method for finding these internally disjoint paths (discussed in §4.3) chooses the ℓ paths based on flows returned by Algorithm 6. This method is (as is) not able to differentiate between differently weighted paths.

There is currently no way of knowing how sub-optimal the results obtained by Algorithms *Whitney*, *MEEF* and *Fan* are with respect to the weight of a minimum-cost spanning subgraph solution for any given graph. In their article Kortsarz and Nutov [34] show that the result obtained by their *K-N* algorithm differs by at most a fixed factor from a minimum-cost solution. This factor is known as an *approximation ratio*. Finding such an approximation ratio for the algorithms developed in this thesis should contribute to the field of connectivity and future research in this area is strongly suggested.

The methods developed in this thesis for the construction of a spanning subgraph with a certain connectivity number are very diverse. It should be interesting to see whether further methods can be developed using other branches of mathematics.

Finally, another avenue of further work would be to investigate whether characterisations can be developed for some graphs that do not contain a complete minimum cut-set, so that Algorithm *Fan* may be implemented on these graphs. For example, an algorithm similar in some aspects to Algorithm *Fan*, but differing in the way in which a spanning subgraph is constructed, may be required. Extensive research was done by the author, but no such algorithm could be developed, nor could any graph characterisations be found so that such Algorithm *Fan* could be implemented on a graph that does not contain a complete minimum cut-set.

References

[1] Balena F, 1999, *Programming Microsoft Visual Basic 6.0*, Microsoft Press, Redmond, ISBN: 0-735-60558-0.

[2] Beineke LW, Oellermann OR & Pippert RE, 2002, *The average connectivity of a graph*, Discrete Mathematics, **252**, pp. 31–45.

[3] Boesch FT & Chen S, 1978, *A generalization of line connectivity and optimally invulnerable graphs*, SIAM Journal on Applied Mathematics, **34**, pp. 657–665.

[4] Bollobás B, 1978, *Extremal graph theory*, Academic Press, London, ISBN: 0-121-11750-2.

[5] Bollobás B, 1979, *Graph theory—An introductory course*, Springer-Verlag, New York, ISBN: 0-387-90399-2.

[6] Bollobás B, Goldsmith DL & Woodall DR, 1981, *Indestructive deletions of edges from graphs*, Journal of Combinatorial Theory, Series B, **30**, 263–275.

[7] Bondy JA, 1969, *Properties of graphs with constraints on degrees*, Studia Scientiarum Mathematicarum Hungarica, **4**, pp. 473–475.

[8] Chartrand G & Harary F, 1968, *Graphs with prescribed connectivities*, in Theory of graphs: Proceedings of the Colloquium held at Tihany, Hungary, September 1966, Academic Press, New York, ISBN: B-000-0CO76-J.

[9] Chartrand G, Kaugars A & Lick DR, 1972, *Critically n-connected graphs*, Proceedings of the American Mathematical Society, **32**, pp. 63–68.

[10] Chartrand G, Kapoor SF, Lesniak L & Lick DR, 1984, *Generalized connectivity in graphs*, Bulletin of the Bombay Mathematical Colloquium, **2**, pp. 1–6.

[11] Chartrand G & Oellerman OR, 1993, *Applied and algorithmic graph theory*, McGraw-Hill, New York, ISBN: 0-071-12575-2.

[12] Cormen T, Leiserson C, Rivest R, Stein C, 2001, *Introduction to Algorithms (Second Edition)*, The MIT Press, Cambridge, ISBN: 0-262-03293-7.

[13] Day DP, Oellermann OR & Swart HC, 1994, *On the ℓ-connectivity of a digraph*, Discrete Mathematics, **127**, pp. 95–104.

[14] Day DP, Oellermann OR & Swart HC, 1999, *Bounds on the size of graphs of given order and ℓ-connectivity*, Discrete Mathematics, **197/198**, pp. 217–223.

[15] Diestel R, 2000, *Graph theory (Second Edition)*, Springer, New York, ISBN: 0-387-98976-5.

[16] Elias P, Feinstein A & Shannon CE, 1956, *Note on maximum flow through a network*, IEEE Transactions on Information Theory, **2(4)**, pp. 117–119.

[17] Esfahanian AH & Hakimi SL, 1984, *On computing the connectivities of graphs and digraphs*, Networks, **14(2)**, pp. 355–366.

[18] Eswaran KP & Tarjan RE, 1976, *Augmentation problems*, SIAM Journal on Computing, **5(4)**, pp. 653–665.

[19] Ford LR, 1956, *Network flow theory*, (Unpublished) Technical Report P–923, The RAND Corporation, Santa Monica.

[20] Ford LR & Fulkerson DR, 1956, *Maximal flow through a network*, Canadian Journal of Mathematics, **8**, pp. 399–404.

[21] Frank A, 1976, *Combinatorial algorithms, algorithmic proofs*, *PhD dissertation, Eötvös University*, Budapest.

[22] Gabow HN, 1993, *A representation for crossing set families with application to submodular flow problems*, Proceedings of the 4th Annual ACM–SIAM Symposium on Discrete Algorithms (SODA), SIAM, Philadelphia, pp. 202–211.

[23] Goldberg AV & Tarjan RE, 1988, *A new approach to the maximum flow problem*, Journal of the Association for Computing Machinery, **35**, pp. 921–940.

[24] Goldsmith DL, 1980, *On the second-order edge-connectivity of a graph*, Congressus Numerantium, **29**, pp. 479–484.

[25] Goldsmith DL, Manvel B & Faber V, 1980, *Separation of graphs into three components by the removal of edges*, Journal of Graph Theory, **4**, pp. 213–218.

[26] Goldsmith DL, 1981, *On the nth-order edge-connectivity of a graph*, Congressus Numerantium, **32**, pp. 375–382.

[27] Györi E, 1978, *On division of graphs to connected subgraphs*, Colloquia Mathematica Societatis János Bolyai, **18**, pp. 485–494.

[28] Hajós G, 1934, *Zum mengerschen Graphensatz*, Acta Litterarum ac Scientiarum Regiae Universitatis Hungaricae Francisco-Josephinae, Sectio Scientiarum Mathematicarum [Szeged], **7**, pp. 44–47.

[29] Harary F, 1962, *The maximum connectivity of a graph*, Proceedings of the National Academy of Sciences of the United States of America, **48**, pp. 1142–1146.

[30] Hsu T & Ramachandran V, 1991, *A linear time algorithm for triconnectivity augmentation*, Proceedings of the 32nd Annual IEEE Symposium on Foundations of Computer Science, pp. 548–559.

[31] Hsu T, 2000, *On four-connecting a triconnected graph*, Journal of Algorithms, **35(2)**, pp. 202–234.

[32] Jordán T, 1995, *On the optimal vertex-connectivity augmentation*, Journal of Combinatorial Theory, Series B, **63(1)**, pp. 8–20.

[33] König D, 1933, *Über trennende Knotenpunkte in Graphen*, Acta Litterarum ac Scientiarum Regiae Universitatis Hungaricae Francisco-Josephinae, Sectio Scientiarum Mathematicarum [Szeged], **6**, pp. 155–179.

[34] Kortsarz G & Nutov Z, 2005, *Approximating k-node connected subgraphs via critical graphs*, SIAM Journal on Computing, **35(1)**, pp. 247–257.

[35] Lick DR, 1972, *Minimally n-line connected graphs*, Journal für die Reine und Angewandte Mathematik, **252**, pp. 178–182.

[36] Lovász L, 1977, *A homology theory for spanning trees of a graph*, Acta Mathematica Academiae Scientiarum Hungaricae, **30**, pp. 241–251.

[37] Lovász L, Saks M & Schrijver A, 1989, *Orthogonal representations and connectivity of graphs*, Linear Algebra and its Applications, **114/115**, pp. 439–454.

[38] Lovász L, 1993, *Combinatorial problems and exercises (Second Edition)*, North-Holland Publishing Company, Amsterdam, ISBN: 0-444-81504-X.

[39] Mader W, 1971, *Eine eigenschaft der Atome endlicher Graphen*, Archiv der Mathematik, **22**, pp. 333–336.

[40] Mader W, 1971, *Minimale n-fach kantenzusammenhängende Graphen*, Mathematische Annalen, **191**, pp. 21–28.

[41] Mader W, 1972, *Ecken vom grad n in minimalen n-fach zusammenhängenden Graphen*, Archiv der Mathematik, **23**, pp. 219–224.

[42] Mader W, 1974, *Kantendisjunkte wege in Graphen*, Monatshefte für Mathematik, **78**, pp. 395–404.

[43] Mader W, 1978, *A reduction method for edge-connectivity in graphs*, Annals of Discrete Mathematics, **3**, pp. 145–164.

[44] Mader W, 1979, *Connectivity and edge-connectivity in finite graphs*, Proceedings of the Seventh British Combinatorial Conference, Cambridge, Surveys in Combinatorics, London Mathematical Society Lecture Notes Series, **38**, Cambridge University Press, Cambridge, ISBN: 0-521-22846-8.

[45] Mader W, 1979, *Zur struktur minimal n-fach zusammenhängender Graphen*, Abhandlungen aus dem Mathematischen Seminar der Universität Hamburg, **49**, pp. 49–69.

[46] Menger K, 1927, *Zur allgemeinen Kurventheorie*, Fundamenta Mathematicae, **10**, pp. 96–115.

[47] Menger K, 1932, *Kurventheorie*, American Mathematical Society, Leipzig, ISBN: 0-828-40172-1.

[48] Microsoft Office Excel, 2002, *Microsoft Office Excel*, [Online], [Cited October 30th, 2006], Available from `http://mvp.support.microsoft.com/communities/mvp.aspx?product=1&competency=Microsoft+Office+Excel`.

[49] Mitzenmacher M & Upfal E, 2005, *Probability and computing—Randomized algorithms and probabilistic analysis*, Cambridge University Press, Cambridge, ISBN: 0-521-83540-2.

[50] Nagamochi H & Ibaraki T, 1992, *Computing edge-connectivity in multigraphs and capacitated graphs*, SIAM Journal on Discrete Mathematics, **5**, pp. 54–66.

[51] Nagamochi H & Ibaraki T, 2000, *A fast algorithm for computing minimum 3-way and 4-way cuts*, Mathematical Programming, **88(3)**, pp. 507–520.

[52] Nagamochi H, Katayama S & Ibaraki T, 2000, *A faster algorithm for computing minimum 5-way and 6-way cuts*, Journal of Combinatorial Optimization, **4(2)**, pp. 151–169.

[53] Naor D, Gusfield D & Martel C, 1997, *A fast algorithm for optimally increasing the edge connectivity*, SIAM Journal on Computing, **26(4)**, pp. 698–707.

[54] Picard J & Queyranne M, 1980, *On the structure of all minimum cuts in a network and applications*, Mathematical Programming Study, **13**, pp. 8–16.

[55] Plesník J, 1976, *Minimum block containing a given graph*, Archiv der Mathematik, **27(6)**, pp. 668–672.

[56] Rosenthal A & Goldner A, 1977, *Smallest augmentations to biconnect a graph*, SIAM Journal on Computing, **6(1)**, pp. 55–66.

[57] Sipser M, 1997, *Introduction to the theory of computation*, PWS Publishing Company, Boston, ISBN: 0-534-94728-X.

[58] Slater PJ, 1974, *A classification of 4-connected graphs*, Journal of Combinatorial Theory. Series B, **17**, pp. 281–298.

[59] Slater PJ, 1975, *Leaves of trees*, Proceedings of the Sixth Southeastern Conference on Combinatorics, Graph Theory, and Computing, Congressus Numerantium, **14**, pp. 549–559.

[60] Tutte WT, 1961, *A theory of 3-connected graphs*, Indagationes mathematicae / Koninklijke Nederlandse Akademie van Wetenschappen, **23**, pp. 441–455.

[61] Visual Basic Developer Center, 1998, *Visual Basic Home*, [Online], [Cited October 30th, 2006], Available from `http://msdn2.microsoft.com/en-us/vbasic/default.aspx`.

[62] Watanebe T & Nakamura A, 1988, *3-Connectivity augmentation problems*, Proceedings of the IEEE International Symposium on Circuits and Systems, **2**, IEEE Conference Proceeding, pp. 1847–1853.

[63] West DB, 1996, *Introduction to graph theory*, Prentice Hall, London, ISBN: 0-132-27828-6.

[64] Whitney H, 1932, *Congruent graphs and the connectivity of graphs*, American Journal of Mathematics, **54**, pp. 150–168.

Appendix A

How to use the CD

The contents of the CD attached to this thesis are described in this appendix. The CD contains a compiled version of the DSS *Connectivity Algorithms*, the source code of the DSS as well as graphs that may be used for testing the DSS. All graphs that were mentioned in Chapter 5 are also included.

The contents of the CD is stored in three folders, namely *DSS*, *Code* and *Graphs*. The folder *DSS* contains the program *Connectivity Algorithms*. To install the program, the reader should copy the two folders, *DSS* and *Graphs*, to desired locations on his/her hard drive. The DSS may then be run by browsing to the folder named *DSS* on the hard drive and then simply executing the file *connalgs.exe*. This program will run on any Windows-based machine. The folder *Code* contains the source code of the DSS *Connectivity Algorithms*. The folder *Graphs* contains adjacency matrices of graphs that may be used to test the system. These adjacency matrices are stored as Excel files with the sheet names already set to the default names that are used by the DSS; hence the coordinates of the vertices of a graph are stored in the sheet labelled *XY* and the adjacency matrix is stored in the sheet labelled *AdjacencyMatrix*. A list of the graphs that may be found in the folder *Graphs* on the CD is presented in Table A.1.

File	Description	$\kappa(G)$
graph1.xls	Graph on 4 vertices.	1
graph2.xls	Graph on 4 vertices (no complete minimum cut-set exists).	2
graph3.xls	Graph on 4 vertices (contains a complete minimum cut-set).	2
graph4.xls	Graph on 8 vertices (no complete minimum cut-set exists).	3
graph5.xls	Graph on 11 vertices (no complete minimum cut-set exists).	5
graph6.xls	Graph on 11 vertices (contains a complete minimum cut-set).	5
spider16.xls	Representation of a spider's web on 16 vertices. Edge weights are taken as the euclidean distance between adjacent vertices (no complete minimum cut-set exists).	3
spider21.xls	Representation of a spider's web on 21 vertices. Edge weights are taken as the euclidean distance between adjacent vertices (contains a complete minimum cut-set).	3
spider21_2.xls	Representation of a spider's web on 21 vertices. All edges have an equal weighting of 1 (contains a complete minimum cut-set).	3

Table A.1: *Description of a list of graphs included in the folder* Graphs *that may be used to test the DSS.*

Appendix B

Source Code for the Program *Connectivity Algorithms*

In this appendix, the source code for the algorithms that were implemented in the DSS *Connectivity Algorithms* (discussed in §5) is given. The algorithms upon which the DSS is based are discussed in §4. All source code was written using Microsoft Visual Basic 6.0. More information on how to program in Microsoft Visual Basic 6.0 may be found in Balena [1]. Comments in the code are italicized and keywords appear in bold.

```
Option Explicit
Public p As Integer
Public maxInt As Long
Dim D() Array containing Distance matrix.
Private XY() Array containing XY coordinates of vertices.
Private xMin, xMax, yMin, yMax As Double
Public k As Integer Actual connectivity number of the graph.
Public ell As Integer New connectivity number of the graph.
Private edgeList() As Variant
Private cutSetIG(), cutSetOG() As Variant
Private D2() As Variant
Private edgeListSorted As Boolean
Private DHasValue, cutSetsInputGraphHasValue, cutSetsOutputGraphHasValue As Boolean
Public filePath, distanceSheet, XYSheet As String
Private graphPaths() As Variant
Private graphPathsIndex As Integer
Private totalCostSaved As Single
Private oldCutVertex() As Boolean Used to distinguish between paths in graphPaths -
paths originating from vertices that belonged to the original cut-set should not be
included in certain for-loops.  Mark these old cut-vertices with flags.
Const PlotAreaWidth = 10815 Width of PlotArea - PlotAreaWidth measured in twips.
Const PlotAreaHeight = 5895 Height of PlotArea - PlotAreaHeight measured in twips.
Const extraSpace = 567 / 2 Number of Twips to add to sides of graph.  Scaling will then take place.

Dim prev(), dist()

Private Declare Sub CopyMemory Lib ‘‘kernel32" Alias ‘‘RtlMoveMemory" (dest As _
  Any, Source As Any, ByVal bytes As Long)

Function xl_Col(ByRef Col_No) As String
Returns Excel column name from numeric position (e.g.:  col_no 27 returns ‘‘AA").
example usage:
sColName = xl_Col(7)

Only allow valid columns.
  If Col_No < 1 Or Col_No > 256 Then Exit Function

  If Col_No < 27 Then Single letter.
    xl_Col = Chr(Col_No + 64)
  Else Two letters.
    xl_Col = Chr(Int((Col_No - 1) / 26) + 64) & _
        Chr(((Col_No - 1) Mod 26) + 1 + 64)
  End If
End Function
```

```
Function xl_ColNo(Col_Name) As Integer
Returns an Excel column number from its name (e.g.:  col_name ''AA" returns 27).
example usage:
iColNo = xl_ColName(''Z")

   Col_Name = UCase(Trim(Col_Name))
   Select Case Len(Col_Name)
      Case 1:  xl_ColNo = Asc(Col_Name) - 64
      Case 2:  xl_ColNo = ((Asc(Left(Col_Name, 1)) - 64) * 26) _
            + (Asc(Right(Col_Name, 1)) - 64)
   End Select
End Function

Function ArrayDims(arr As Variant) As Integer
Returns the number of dimensions of an array.
Dim ptr As Long
Dim VType As Integer
Const VT_BYREF = &H4000&

   Get the real VarType of the argument.
   CopyMemory VType, arr, 2

   Exit if not an array.
   If (VType And vbArray) = 0 Then Exit Function

   Get the address of the SAFEARRAY descriptor.  This is stored in the second half of the Variant parameter that has
   received the array.
   CopyMemory ptr, ByVal VarPtr(arr) + 8, 4

   See whether the routine was passed a Variant that contains an array, rather than directly an array in the former case
   ptr already points to the SA structure.

   If (VType And VT_BYREF) Then
      ptr is a pointer to a pointer.
      CopyMemory ptr, ByVal ptr, 4
   End If

   Get the address of the SAFEARRAY structure.  This is stored in the descriptor.  Get the first word of the SAFEARRAY
   structure which holds the number of dimensions but first check that saAddr is non-zero, otherwise this routine bombs
   when the array is uninitialized.
   If ptr Then
      CopyMemory ArrayDims, ByVal ptr, 2
   End If
End Function

Public Function CountElements(ByVal SimpleArray As Variant) As Long
Returns the number of elements in an array.
   Ignore error if array not dimensioned.
   On Error Resume Next
   If Not IsArray(SimpleArray) Then Exit Function
   CountElements = Abs((LBound(SimpleArray) - UBound(SimpleArray))) + 1
End Function

Public Function DecToBin(DeciValue As Long) As String
Converts a decimal value to a binary string.  Dim i As Integer
Dim NoOfBits As Integer
NoOfBits = p
Ensures that there are enough bits to contain the number.
Do While DeciValue > (2^NoOfBits − 1
   NoOfBits = NoOfBits + 8
Loop
DecToBin = vbNullString
Build the string.
For i = 0 To (NoOfBits - 1)
   DecToBin = CStr((DeciValue And 2^i) / 2^i) & DecToBin
Next i
End Function

Public Function MaxArr1D(ByVal D As Variant) As Variant
Returns a maximum value of an array.  Dim i, j As Integer
Dim X As Variant
   X = D(1)
   For i = 2 To UBound(D)
      If X < D(i) Then
         X = D(i)
      End If
   Next
   MaxArr1D = X
End Function

Public Function MaxArr2D(ByVal D As Variant) As Variant()
Returns a maximum element of a one-dimensional array or, for a two-dimensional array, an array with element i a maximum
element of column i.
```

```
Dim i, j As Integer
Dim X() As Variant
  ReDim X(1 To UBound(D, 2))
  For j = 1 To UBound(D, 2)
    X(j) = D(1, j)
    For i = 2 To UBound(D, 1)
      If X(j) < D(i, j) Then
        X(j) = D(i, j)
      End If
    Next
  Next
  MaxArr2D = X
End Function

Sub QuickSort(arr As Variant, Optional numEls As Variant, Optional descending As Boolean)
Implementation of the well-known Quick-Sort algorithm.
Dim value As Variant, temp(1 To 3) As Variant
Dim sp As Integer
Dim leftStk(32) As Long, rightStk(32) As Long
Dim leftNdx As Long, rightNdx As Long
Dim i As Long, j As Long

  Account for Optional arguments.
  If IsMissing(numEls) Then numEls = UBound(arr, 1)
  Init pointers.
  leftNdx = LBound(arr, 1)
  rightNdx = numEls
  Init stack.
  sp = 1
  leftStk(sp) = leftNdx
  rightStk(sp) = rightNdx

  Do
    If rightNdx > leftNdx Then
      value = arr(rightNdx, 1)
      i = leftNdx - 1
      j = rightNdx
      Find the pivot item.
      If descending Then
        Do
          Do:  i = i + 1:  Loop Until arr(i, 1) <= value
          Do:  j = j - 1:  Loop Until j = leftNdx Or arr(j, 1) >= value
          temp(1) = arr(i, 1)
          temp(2) = arr(i, 2)
          temp(3) = arr(i, 3)

          arr(i, 1) = arr(j, 1)
          arr(i, 2) = arr(j, 2)
          arr(i, 3) = arr(j, 3)

          arr(j, 1) = temp(1)
          arr(j, 2) = temp(2)
          arr(j, 3) = temp(3)
        Loop Until j <= i
      Else
        Do
          Do:  i = i + 1:  Loop Until arr(i, 1) >= value
          Do:  j = j - 1:  Loop Until j = leftNdx Or arr(j, 1) <= value
          temp(1) = arr(i, 1)
          temp(2) = arr(i, 2)
          temp(3) = arr(i, 3)

          arr(i, 1) = arr(j, 1)
          arr(i, 2) = arr(j, 2)
          arr(i, 3) = arr(j, 3)

          arr(j, 1) = temp(1)
          arr(j, 2) = temp(2)
          arr(j, 3) = temp(3)
        Loop Until j <= i
      End If
      Swap found items.
      temp(1) = arr(j, 1)
      temp(2) = arr(j, 2)
      temp(3) = arr(j, 3)

      arr(j, 1) = arr(i, 1)
      arr(j, 2) = arr(i, 2)
      arr(j, 3) = arr(i, 3)

      arr(i, 1) = arr(rightNdx, 1)
      arr(i, 2) = arr(rightNdx, 2)
      arr(i, 3) = arr(rightNdx, 3)
```

```
        arr(rightNdx, 1) = temp(1)
        arr(rightNdx, 2) = temp(2)
        arr(rightNdx, 3) = temp(3)
        Push on the stack the pair of pointers that differ most.
        sp = sp + 1
        If (i - leftNdx) > (rightNdx - i) Then
          leftStk(sp) = leftNdx
          rightStk(sp) = i - 1
          leftNdx = i + 1
        Else
          leftStk(sp) = i + 1
          rightStk(sp) = rightNdx
          rightNdx = i - 1
        End If
      Else
        Pop a new pair of pointers off the stacks.
        leftNdx = leftStk(sp)
        rightNdx = rightStk(sp)
        sp = sp - 1
        If sp = 0 Then Exit Do
      End If
  Loop
End Sub

Public Function Floyd(ByVal D As Variant, ByVal p As Integer) As Variant()
Floyd's Algorithm determines the shortest distance between any two vertices.
Dim i, j, k As Integer
Dim a() As Variant
ReDim a(1 To p, 1 To p) As Variant
  For i = 1 To p
    For j = 1 To p
      If (D(i, j) = 0) And (i <> j) Then
        D(i, j) = maxInt
      End If
    Next
  Next

  For k = 1 To p
    For i = 1 To p
      For j = 1 To p
        If D(i, j) <= (D(i, k) + D(k, j)) Then
          a(i, j) = D(i, j)
        Else
          a(i, j) = D(i, k) + D(k, j)
        End If
      Next
    Next
    D = a
  Next
  Floyd = a
End Function

Public Function Ford(ByVal DFlow, ByVal s As Integer, ByVal nrVertices As Integer) As Variant()
Determines the distance labels for a graph from the vertex s.
Dim setA(), setB()
ReDim setA(1 To nrVertices)
ReDim setB(1 To nrVertices)
Dim ACount, BCount As Integer
Dim i, j, minElement, minDist As Integer
Dim u, v As Integer Two vertices in the graph.
Dim labelCount() As Integer
ReDim labelCount(1 To nrVertices) As Integer
  ReDim prev(1 To nrVertices)
  ReDim dist(1 To nrVertices)
  For i = 1 To nrVertices
    setA(i) = False
    setB(i) = False
    dist(i) = maxInt
    labelCount(i) = 0
  Next
  dist(s) = 0
  prev(s) = 0
  ACount = 0
  setB(s) = True
  BCount = 1
  While ACount <= nrVertices
    Find vertex in setB with minimum distance.
    minElement = -1 Indicates that minimum element has not yet been set.
    minDist = 0
    For i = 1 To nrVertices
      If (setB(i) = True) Then
        If minElement = -1 Then
          minElement = i
          minDist = dist(i)
```

```
        End If
        If (dist(i) < minDist) Then
          minElement = i
          minDist = dist(i)
        End If
      End If
    Next
    u = minElement
    setA(u) = True
    ACount = ACount + 1
    setB(u) = False
    BCount = BCount - 1
    labelCount(u) = labelCount(u) + 1
    If labelCount(u) = nrVertices Then
      MsgBox (``Negative cycle found in graph.")
      Exit Function
    End If
    Find all neighbours v of u.
    For v = 1 To nrVertices
      If DFlow(u, v) = 1 Then
        If dist(v) > dist(u) + 1 Then
          dist(v) = dist(u) + 1
          prev(v) = u
          If setA(v) = True Then
            setA(v) = False
            ACount = ACount - 1
            setB(v) = True
            BCount = BCount + 1
          ElseIf (setB(v) = False) And (setA(v) = False) Then
            setB(v) = True
            BCount = BCount + 1
          End If
        End If
      End If
    Next
    If BCount = 0 Then
      No temporary nodes to set permanent.
      Ford = dist
      Exit Function
    End If
  Wend
End Function

Public Function ShortestAugPath(ByVal DFlow, ByVal s As Integer, ByVal t As Integer, ByVal pathList,_
ByVal nrPaths, ByRef pathCount, ByVal nrVertices, ByVal addT As Boolean) As Variant() pathCount counts the
actual number of paths found (pathCount <= nrPaths).
Finds Paths in a graph by which the maximum flow can take place from vertex s to vertex t.
Dim i, j As Integer
Dim u, v As Integer
Dim minFound As Boolean
Dim minDist, minElement As Integer
Dim pathListIndex As Integer
  ReDim pred(1 To nrVertices)
  pathListIndex = 1
  For i = 1 To nrVertices
    pred(i) = 0
  Next
  u = s
  While dist(s) <= 2 * nrVertices
    minFound = False
    For v = 1 To nrVertices
      If (DFlow(u, v) = 1) Then
        If minFound = False Then
          minFound = True
          minElement = v
          minDist = dist(v)
        End If
        If dist(v) < minDist Then
          minElement = v
          minDist = dist(v)
        End If
      End If
    Next
    v = minElement
    If (DFlow(u, v) > 0) And (dist(u) = dist(v) + 1) Then
      prev(v) = u
      u = v
      If v = t Then
        j = 1
        If addT Then Add the vertex t to the path.
          pathList(pathListIndex, j) = v Note that t forms part of the path.
          v = prev(v)
          j = j + 1
          While v <> s
```

```
                Augment path if vertex v is not vertex t.
                DFlow(prev(v), v) = 0
                DFlow(v, prev(v)) = 1
                Find path.
                pathList(pathListIndex, j) = v Add vertices between s and t to the path.
                v = prev(v)
                j = j + 1
            Wend
            pathList(pathListIndex, j) = v Add vertex s to the path.
        Else
            While v <> s
                Augment path.
                DFlow(prev(v), v) = 0
                DFlow(v, prev(v)) = 1
                Find path.
                v = prev(v)
                pathList(pathListIndex, j) = v Note that t does not form part of the path.
                j = j + 1
            Wend
        End If
        pathListIndex = pathListIndex + 1
        Reset u.
        u = s
    End If
  Else
    dist(u) = dist(u) + 1
    If u <> s Then
      u = prev(u)
    End If
  End If
 Wend
 pathCount = pathListIndex - 1
 ShortestAugPath = pathList
End Function

Public Function Zeros(ByVal row, ByVal col As Integer) As Variant()
Creates a matrix of zeros with specified dimensions.
Dim i, j As Integer
Dim a() As Variant
    ReDim a(1 To row, 1 To col) As Variant
    For i = 1 To row
      For j = 1 To col
        a(i, j) = 0
      Next
    Next
  Zeros = a
End Function

Public Function GetDFlow(ByVal D) As Variant()
Constructs a directed graph from a graph with adjacency matrix D. The returned graph has twice as much vertices as the
graph D. GetDFlow is constructed for use in the Shortest Augmenting Path calculations.  Dim DFlow() As Variant
Dim i, j As Integer
  ReDim DFlow(1 To 2 * p, 1 To 2 * p)
  For i = 1 To 2 * p index up to 2p as sink has not been added yet.
    For j = 1 To 2 * p
      DFlow(i, j) = 0
    Next
  Next
  For i = 1 To p
    DFlow(2 * i - 1, 2 * i) = 1
  Next
  For i = 1 To p
    For j = i + 1 To p
      If D(i, j) > 0 Then
        DFlow(2 * i, 2 * j - 1) = 1
        DFlow(2 * j, 2 * i - 1) = 1
      End If
    Next
  Next
  GetDFlow = DFlow
End Function

Public Function ConnNumber(ByVal D As Variant) As Integer
Implementation of Algorithm 3.
  Dim i, j, NMin, u, v As Integer Special variables as defined in Algorithm 3.
Dim a, b, m, n, degCurr, degPrev, index As Integer
Dim neighbours() As Integer
Dim done As Boolean
Dim DFlow(), DFlowOld() As Variant
Dim pathList As Variant
Dim nrPaths, pathCount As Integer
Dim cutSetEdgeCount As Integer
Dim reRun As Boolean
Dim paths() As Variant
```

```
Dim distOld() As Variant
Dim cutSetIGList() As Variant
Dim completeCutSetFound As Boolean
Dim cutSetRowCount As Integer
Dim oldCutVertexIndex As Integer
  done = False
  i = 1
  NMin = p - 1

  Find vertex u in D with minimum degree.
  For a = 1 To p
    degCurr = 0
    For b = 1 To p
      If D(a, b) > 0 Then
        degCurr = degCurr + 1
      End If
    Next
    If a = 1 Then
      degPrev = degCurr
      index = a
    Else
      If degCurr < degPrev Then
        degPrev = degCurr The minimum degree.
        index = a Points to a vertex with minimum degree.
      End If
    End If
  Next
  u = index Vertex u has a minimum degree of degPrev.

  If degPrev = 0 Then
    ConnNumber = degPrev
    Exit Function
  End If
  Store neighbours of u in variable neighbours.
  ReDim neighbours(1 To degPrev) As Integer
  b = 1
  For a = 1 To p
    If D(u, a) > 0 Then
      neighbours(b) = a
      b = b + 1
    End If
  Next

  For v = 1 To p
    If (v <> u) And (D(v, u) = 0) Then
    (For all vertices in V(G) \ (u and the neighbours of u) do...)
      Compute  N(u, v) and set  NMin = min{NMin, N(u, v)}.

      Construct digraph DFlow.
      DFlow = GetDFlow(D)
      DFlowOld = DFlow

      Reset graphPathsIndex.
      Get distance weighting for the digraph DFlow.  Distance labels need to be recalculated, as Shortest Augmenting
      Paths algorithm changes these labels.
      dist = Ford(DFlow, 2 * (u) - 1, 2 * p)

      DFlow = DFlowOld
      Swop all arcs around.
      DFlow = SwopArcs(DFlow, 2 * p)

      nrPaths = 0
      Find nr of paths for pathList matrix (min degree of vertex s).
      For m = 1 To p
        If D(u, m) > 0 Then
          nrPaths = nrPaths + 1
        End If
      Next
      ReDim pathList(1 To nrPaths, 1 To 2 * p + 1) As Variant
      pathList now empty and ready to populate  (2p + 1) columns to keep index of for loops when testing for valid edges
      in bounds.  When a path is shorter than 2p vertices, the other open places in pathList are filled with zeros.
      Vertex t is not included in the paths).
      Clear pathList.
      For m = 1 To nrPaths
        For n = 1 To 2 * p + 1
          pathList(m, n) = 0
        Next
      Next
      pathList = ShortestAugPath(DFlow, 2 * (u) - 1, 2 * (v) - 1, pathList, _
      nrPaths, pathCount, 2 * p, True) NOTE: Path starts from in-vertex, not out-vertex (Arcs are reversed, hence, the
      in-vertex acts as the out-vertex).

      If pathCount < NMin Then
        NMin = pathCount
```

```
      End If
    End If
  Next
  While done = False
    If (i + 1) > (degPrev - 1) Then
    Special condition to terminate while loop if for loop cannot execute.
      done = True
    Else
      For j = i + 1 To degPrev - 1
        If (i >= degPrev - 2) Or (i >= NMin) Then
          done = True
          Exit For
        End If
        If D(neighbours(i), neighbours(j)) = 0 Then
          Compute N(neighbours(i), neighbours(j)) and set  NMin = min{NMin, N(neighbours(i), neighbours(j))}.

          Construct digraph DFlow.
          DFlow = GetDFlow(D)
          DFlowOld = DFlow

          Reset graphPathsIndex.
          Get distance weighting for the digraph DFlow.  Distance labels need to be recalculated, as Shortest Augmenting
          Paths algorithm changes these labels.
          dist = Ford(DFlow, 2 * (neighbours(i)) - 1, 2 * p)

          DFlow = DFlowOld
          Reverse all arcs.
          DFlow = SwopArcs(DFlow, 2 * p)

          nrPaths = 0
          Find number of paths for pathList matrix (min degree of vertex s).
          For m = 1 To p
            If D(neighbours(i), m) > 0 Then
              nrPaths = nrPaths + 1
            End If
          Next
          ReDim pathList(1 To nrPaths, 1 To 2 * p + 1) As Variant
          pathList now empty and ready to populate  (2p + 1) columns to keep index of for loops when testing for
          valid edges in bounds.  When a path is shorter than 2p vertices, the other open places in pathList are filled
          with zeros.  Vertex t is not included in the paths).
          Clear pathList.
          For m = 1 To nrPaths
            For n = 1 To 2 * p + 1
              pathList(m, n) = 0
            Next
          Next
          pathList = ShortestAugPath(DFlow, 2 * (neighbours(i)) - 1, _
          2 * (neighbours(j)) - 1, pathList, nrPaths, pathCount, 2 * p, _
          True) NOTE: Path starts from in-vertex, not out-vertex (Arcs are reversed, hence, the in-vertex is actually
          now the out-vertex).

          If pathCount < NMin Then
            NMin = pathCount
          End If
        End If
      Next
    End If
    If done = False Then
      i = i + 1
    End If
  Wend
  ConnNumber = NMin
End Function

Public Function CVSList(ByVal D As Variant, ByRef nrRows As Integer) As Variant()
Implementation of Algorithm 4.  Finds all minimum cut-sets in a graph represented by the adjacency matrix D.
Dim i, j, k, count As Integer
Dim zeroMatrix() As Variant
Dim DShortestPaths() As Variant
Dim val() As Variant
Dim maxVal As Variant
Dim errFound As Boolean
Dim index, m, seqStart, seqStop, pos As Integer
Dim CVSListStarted, VStarted As Boolean
Dim Dtmp(), nodes, v() As Variant
Dim List(), ListOld() As Variant
Dim minCutSize As Integer
  errFound = False
  count = 0
  zeroMatrix = Zeros(p, p)

  For i = 1 To p
    For j = 1 To p
      If D(i, j) = zeroMatrix(i, j) Then
```

```
        count = count + 1
      End If
    Next
  Next

  DShortestPaths = Floyd(D, p)
  maxVal = MaxArr1D(MaxArr2D(DShortestPaths)) If there remains any inf value in the
  matrix, then the graph is disconnected.

  It is a necessary condition for CutVertexSet that the graph must be connected before the algoritm can start.
  If (count = p^2) Or (maxVal = maxInt) Then count = p^2 => D is a zero matrix.
    errFound = True
  End If
  If errFound = True Then
    ReDim List(1 To 1, 1 To 2)
    List(1, 1) = 0
    List(1, 2) = 0
    CVSList = List
    Exit Function
  Else
    index = 1
    CVSListStarted = False Binary flag to indicate when the first CVS has been found.
    For m = 1 To p - 1
      seqStart = 0
      seqStop = 0
      For i = 1 To m
        seqStart = seqStart + 2^(i-1)
        seqStop = seqStop + 2^(p-i)
      Next
      For i = seqStart To seqStop
        nodes = DecToBin(CLng(i))
        count = 0
        For j = 1 To p
          If Mid(nodes, j, 1) = ``1" Then
            count = count + 1
          End If
        Next
        If count = m Then
        Ensures that only pCm combinations are checked.  Eg.  if p=3 and m=1, then the for loop will now only test 001,
        010, 100, excluding 011.
          VStarted = False
          pos = 1
          For j = 1 To p
            If Mid(nodes, p + 1 - j, 1) = ``0" Then Search from right to left.
            Remember that element 1 is not in the first position, but in the last position.
              If VStarted = False Then
                ReDim v(1 To 1) As Variant
                VStarted = True
              Else
                ReDim Preserve v(1 To UBound(v) + 1) As Variant
              End If
              v(pos) = j Stores positions of the bit-vector nodes that contains zeros.  These positions will indicate
              the columns to keep to check if the remaining vertices are still connected.
              pos = pos + 1
            End If
          Next
          Only the vertices that were not removed from the cut-set should now remain in D. Get the valid edges from N by
          indexing with V.
          Dtmp = Zeros(UBound(v), UBound(v))
          For j = 1 To UBound(v)
            For k = 1 To UBound(v)
              Dtmp(j, k) = D(v(j), v(k))
            Next
          Next
          DShortestPaths = Floyd(Dtmp, UBound(v))
          maxVal = MaxArr1D(MaxArr2D(DShortestPaths)) If there remains any inf value in the matrix, then the graph is
          disconnected.
          If maxVal = maxInt Then
            Minimum cut-set found.
            If CVSListStarted = False Then
              CVSListStarted = True
              ReDim List(1 To 1, 1 To 2)
              minCutSize = m
            End If
            If minCutSize = m Then
              ReDim ListOld(1 To UBound(List, 1), 1 To 2)

              ListOld = List Makes backup of CVSList.
              If Not IsEmpty(ListOld(1, 1)) Then
                ReDim List(1 To UBound(ListOld, 1) + 1, 1 To 2)
              End If
              Now copy ListOld over to List.
              For j = 1 To UBound(ListOld, 1)
```

```
                List(j, 1) = ListOld(j, 1)
                List(j, 2) = ListOld(j, 2)
              Next

              List(index, 1) = m Ordinal value of CVS.
              List(index, 2) = nodes
              index = index + 1
              nrRows = index - 1
            Else
              nrRows = index - 1
              CVSList = List
              Exit Function
            End If
          Else
            If (i = seqStop) And (m = p - 1) And (CVSListStarted = 0) Then
              Special case where no cut-vertex set could be
              found.  The graph thus has to be a complete graph.
              ReDim List(1 To 1, 1 To 2) As Variant
              List(1, 1) = maxInt
              List(1, 2) = maxInt
              Exit Function
            End If
          End If
        End If
      Next
    Next
  End If
  CVSList = List
End Function

Public Function CVS(ByVal D As Variant) As Variant()
Implementation of Algorithm 4.  Finds a minimum cut-sets in a graph represented by the adjacency matrix D.
  Dim i, j, k, count As Integer
  Dim zeroMatrix() As Variant
  Dim DShortestPaths() As Variant
  Dim val() As Variant
  Dim maxVal As Variant
  Dim errFound As Boolean

  Dim index, m, seqStart, seqStop, pos As Integer
  Dim CVSFound, VStarted As Boolean
  Dim Dtmp(), nodes, v() As Variant
  Dim cutSet(), ListOld() As Variant

  errFound = False
  count = 0
  zeroMatrix = Zeros(p, p)

  For i = 1 To p
    For j = 1 To p
      If D(i, j) = zeroMatrix(i, j) Then
        count = count + 1
      End If
    Next
  Next

  DShortestPaths = Floyd(D, p)
  maxVal = MaxArr1D(MaxArr2D(DShortestPaths)) If there remains any inf value in the matrix, then the graph is disconnected.

  It is a necessary condition for CutVertexSet that the graph must be connected before the algorithm can start.
  If (count = p^2) Or (maxVal = maxInt) Then count = p^2, hence D is a zero matrix.
    errFound = True
  End If
  If errFound = True Then
    ReDim cutSet(1 To 2)
    cutSet(1) = 0
    cutSet(2) = 0
    CVS = cutSet
    Exit Function
  Else
    CVSFound = False Binary flag to indicate when the first CVS has been found.
    For m = 1 To p - 1
      seqStart = 0
      seqStop = 0
      For i = 1 To m
        seqStart = seqStart + 2^(i-1)
        seqStop = seqStop + 2^(p-i)
      Next
      For i = seqStart To seqStop
        nodes = DecToBin(CLng(i))
        count = 0
        For j = 1 To p
          If Mid(nodes, j, 1) = "1" Then
```

```
            count = count + 1
          End If
        Next
        If count = m Then
        Ensures that only (p m) combinations are checked. Eg. if p=3 and m=1, then the for loop will now only check 001,
        010, 100, excluding 011.
          VStarted = False
          pos = 1
          For j = 1 To p
            If Mid(nodes, p + 1 - j, 1) = ‘‘0" Then Search from right to left (element 1 is not in the first
            position, but in the last position).
              If VStarted = False Then
                ReDim v(1 To 1) As Variant
                VStarted = True
              Else
                ReDim Preserve v(1 To UBound(v) + 1) As Variant
              End If
              v(pos) = j Stores positions of the bit-vector nodes that contains zeros. These will indicate the columns
              to keep to check if the remaining vertices are still connected.
              pos = pos + 1
            End If
          Next
          Only the vertices that were not removed from the cut-set should now remain in D. Get the valid edges from N by
          indexing with V.
          Dtmp = Zeros(UBound(v), UBound(v))
          For j = 1 To UBound(v)
            For k = 1 To UBound(v)
              Dtmp(j, k) = D(v(j), v(k))
            Next
          Next
          DShortestPaths = Floyd(Dtmp, UBound(v))
          maxVal = MaxArr1D(MaxArr2D(DShortestPaths)) If there remains any inf value in
          the matrix, then the graph is disconnected.
          If maxVal = maxInt Then
            CVS found
            ReDim cutSet(1 To 2)
            cutSet(1) = m Ordinal value of CVS.
            cutSet(2) = nodes
            CVS = cutSet
            Exit Function
          Else
            If (i = seqStop) And (m = p - 1) And (CVSFound = 0) Then
              Special case where no cut-vertex set could be found. The graph thus has to be a complete graph.
              ReDim cutSet(1 To 1, 1 To 2) As Variant
              cutSet(1) = maxInt
              cutSet(2) = maxInt
              Exit Function
            End If
          End If
        End If
      Next
    Next
  End If
  CVS = cutSet
End Function

Private Sub DisplayGraph(ByVal A As Variant, ByVal clearGraph As Boolean)
Draws the graph presented by the adjacency matrix A in PlotArea.
Dim i, j As Integer
  If clearGraph Then
    plotArea.Cls
  End If

  plotArea.Scale (0 - extraSpace, 0 - extraSpace)-(PlotAreaWidth + extraSpace, PlotAreaHeight _
  + extraSpace)

  For i = 1 To p
    For j = i + 1 To p
      If A(i, j) > 0 Then
        plotArea.Line (XY(1, i), PlotAreaHeight - XY(2, i))-(XY(1, j), _
        PlotAreaHeight - XY(2, j)), QBColor(0)
      End If
    Next
  Next

  Plot the vertices.
  plotArea.FillStyle = 0
  plotArea.FillColor = QBColor(10) RGB(70, 255, 0)
  For i = 1 To p
    plotArea.Circle (XY(1, i), PlotAreaHeight - XY(2, i)), extraSpace / 2, QBColor(0),,,1
    If i < 10 Then
      plotArea.CurrentX = XY(1, i) - 100
    Else
      plotArea.CurrentX = XY(1, i) - 140 Shifts number a bit more to the left if larger than 9.
```

```
    End If
    plotArea.CurrentY = PlotAreaHeight - XY(2, i) - 100
    plotArea.Print i
  Next

End Sub

Private Sub DisplayCutSet(ByVal cutSet As Variant, ByVal color As Integer)
Displays the cut-set represented by the vertices stored in cutSet in PlotArea.
Dim i, j As Integer
  For i = 1 To p
    If Mid(cutSet(2), p + 1 - i, 1) = ‘‘1" Then
      Plot the vertices.
      plotArea.FillStyle = 0
      plotArea.FillColor = QBColor(color) RGB(70, 255, 0).

      plotArea.Circle (XY(1, i), PlotAreaHeight - XY(2, i)), extraSpace / 2, QBColor(0),,,1
      If i < 10 Then
        plotArea.CurrentX = XY(1, i) - 100
      Else
        plotArea.CurrentX = XY(1, i) - 140 Shifts number a bit more to the left if larger than 9.
      End If
      plotArea.CurrentY = PlotAreaHeight - XY(2, i) - 100
      plotArea.Print i
    End If
  Next
End Sub

Private Function AddSink(ByVal DFlow, ByVal cutSetIG As Variant) As Variant()
Adds a sink vertex to the graph represented by the adjacency matrix DFlow.
Dim i, j As Integer
Dim NewDFlow() As Variant
  ReDim NewDFlow(1 To 2 * p + 1, 1 To 2 * p + 1) As Variant
  For i = 1 To 2 * p
    For j = 1 To 2 * p
      NewDFlow(i, j) = DFlow(i, j)
    Next
  Next
  Insert zeros for every element in row and column of sink vertex.
  For i = 1 To 2 * p + 1
    NewDFlow(i, 2 * p + 1) = 0
    NewDFlow(2 * p + 1, i) = 0
  Next

  For i = 1 To p
    If Mid(cutSetIG(2), i, 1) = ‘‘1" Then
      NewDFlow(2 * p + 1, 2 * (p - i + 1) - 1) = 1
    End If
  Next
  AddSink = NewDFlow
End Function

Private Function disjPathsWhitney(ByVal pathList As Variant, ByVal pathCount As Integer, ByVal s As Integer, _
ByVal t As Integer) As Variant()
This function constructs internally disjoint paths from the flow between any vertex s and t for Algorithm Whitney.
Dim i, j, m As Integer
Dim a, b, col As Integer
Dim edgeCounter() As Integer

  Construct matrix edgeCounter.  The value in position (i, j) indicates the number of times that edge has been traversed.
  ReDim edgeCounter(1 To 2 * p, 1 To 2 * p) As Integer Sink vertex not included.
  For i = 1 To pathCount
    For j = 1 To 2 * p
      If (j = 1) And (pathList(i, j + 1) = 0) Then If vertex is a cut-vertex and no path exists (Special case)
        Exit For
      End If
      If pathList(i, j + 1) <> 0 Then
        If pathList(i, j) = t Then Make edgeCounter(pathList(i, j), pathList(i, j+1) = 1.
          edgeCounter(pathList(i, j), pathList(i, j + 1)) = 1
          edgeCounter(pathList(i, j + 1), pathList(i, j)) = 1
        Else
          edgeCounter(pathList(i, j), pathList(i, j + 1)) = edgeCounter(pathList(i, j), _
          pathList(i, j + 1)) + 1
          edgeCounter(pathList(i, j + 1), pathList(i, j)) = edgeCounter(pathList(i, j + 1), _
          pathList(i, j)) + 1
        End If
      End If
    Next
  Next

  Remove all arcs that were traversed an even number of times.
  For i = 1 To 2 * p
    For j = 1 To 2 * p
      If edgeCounter(i, j) > 0 Then
```

```
        edgeCounter(i, j) = edgeCounter(i, j) Mod 2
      End If
    Next
  Next

  Clear pathList.
  For i = 1 To pathCount
    For j = 1 To 2 * p + 1
      pathList(i, j) = 0
    Next
  Next

  Repopulate pathList with internally disjoint paths.
  j = 1
  col = 1
  For i = 1 To 2 * p
    If edgeCounter(s, i) > 0 Then
      a = s
      b = i
      While (b <> t + 1) While b is not the out-vertex of t (remember that the out-vertex is actually the in-vertex due
      to edges that were reversed.
        pathList(j, col) = a
        pathList(j, col + 1) = b
        col = col + 1
        Take the edges out of edgeCounter.
        edgeCounter(a, b) = 0
        edgeCounter(b, a) = 0
        a = b
        find next value for b.
        For b = 1 To 2 * p
          If edgeCounter(a, b) > 0 Then
            Exit For
          End If
        Next
      Wend
      Add cut-vertex (b) (in- and out-vertex) and vertex preceding it to path.
      pathList(j, col) = a
      pathList(j, col + 1) = b (out-cut-vertex)
      pathList(j, col + 2) = b - 1 (in-cut-vertex)
      col = 1
      j = j + 1
    End If
  Next

  disjPathsWhitney = pathList
End Function

Private Function disjPaths(ByVal pathList As Variant, ByVal pathCount As Integer, ByVal s As Integer) As Variant()
This function constructs internally disjoint paths from the flow between a vertex t and the cut-set cutsetIG.
Dim i, j, m As Integer
Dim a, b, col As Integer
Dim edgeCounter() As Integer

  Construct matrix edgeCounter.  The value in position (i, j) indicates the number of times that edge has been traversed.
  ReDim edgeCounter(1 To 2 * p, 1 To 2 * p) As Integer Sink vertex not included.
  For i = 1 To pathCount
    For j = 1 To 2 * p
      If (j = 1) And (pathList(i, j + 1) = 0) Then If vertex is a cut-vertex and no path exists (Special case).
        Exit For
      End If
      If pathList(i, j + 1) <> 0 Then
        edgeCounter(pathList(i, j), pathList(i, j + 1)) = edgeCounter(pathList(i, j), pathList(i, j + 1)) + 1
        edgeCounter(pathList(i, j + 1), pathList(i, j)) = edgeCounter(pathList(i, j + 1), pathList(i, j)) + 1
      End If
    Next
  Next

  Remove all arcs that were traversed an even number of times.
  For i = 1 To 2 * p
    For j = 1 To 2 * p
      If edgeCounter(i, j) > 0 Then
        edgeCounter(i, j) = edgeCounter(i, j) Mod 2
      End If
    Next
  Next

  Clear pathList.
  For i = 1 To pathCount
    For j = 1 To 2 * p + 1
      pathList(i, j) = 0
    Next
  Next

  Repopulate pathList with internally disjoint paths.
```

```
j = 1
col = 1
For i = 1 To 2 * p
  If edgeCounter(s, i) > 0 Then
    a = s
    b = i
    If b Mod 2 = 1 Then
    b is uneven.
      m = (b + 1) / 2
    Else
    b is even.
      m = b / 2
    End If
    While (Mid(cutSetIG(2), p - m + 1, 1) <> ''1") While m is not a cut-vertex, do...
      pathList(j, col) = a
      pathList(j, col + 1) = b
      col = col + 1
      Take the edges out of edgeCounter.
      edgeCounter(a, b) = 0
      edgeCounter(b, a) = 0
      a = b
      find next value for b.
      For b = 1 To 2 * p
        If edgeCounter(a, b) > 0 Then
          Exit For
        End If
      Next
      If b Mod 2 = 1 Then
        m = (b + 1) / 2
      Else
        m = b / 2
      End If
    Wend
    Add cut-vertex (b) (in- and out-vertex) and vertex preceding it to path.
    pathList(j, col) = a
    pathList(j, col + 1) = b (out-cut-vertex)
    pathList(j, col + 2) = b - 1 (in-cut-vertex)
    col = 1
    j = j + 1
    Take the edges out of edgeCounter.
    edgeCounter(a, b) = 0
    edgeCounter(b, a) = 0
  End If
Next


Remove all edges traversed more than once from DFlow and return to the calling procedure.
disjPaths = pathList
End Function

Private Function SortPaths(ByVal pathList As Variant, ByVal pathCount As Integer) As Variant()
Sorts the paths in pathList according to the weight of each path in ascending order.
Dim i, j As Integer
Dim paths() As Variant
  ReDim paths(1 To pathCount, 1 To 3) As Variant
  Set cost of each path to zero.
  For i = 1 To pathCount
    paths(i, 1) = 0
  Next
  Find the cost of each path.
  For i = 1 To pathCount
    For j = 1 To 2 * p Sink vertex not included.
      If pathList(i, j + 1) <> 0 Then
        Test if vertex j is an out vertex.
        If (pathList(i, j) Mod 2) = 1 Then
          paths(i, 1) = paths(i, 1) + D((pathList(i, j) + 1) / 2, pathList(i, j + 1) / 2)
        End If
      End If
    Next
    paths(i, 2) = i Index of path in pathList.  This is necessary as paths will now be sorted from the cheapest to the
    most expensive path.
    paths(i, 3) = 0 Dummy value of zero allocated to the variable.
  Next

  Sort paths.
  Call QuickSort(paths, pathCount, False)
  SortPaths = paths
End Function

Private Sub InsertPaths(ByVal paths As Variant, ByVal pathList As Variant, ByVal nrPaths As Integer)
Inserts paths into the matrix D2.  Also builds a list, graphPaths, containing all paths inserted into the graph.
Dim i, j As Integer
Dim col As Integer
  For i = 1 To nrPaths Only insert the cheapest nrPaths paths.
```

```
    col = 1
    For j = 1 To 2 * p
      If pathList(paths(i, 2), j + 1) <> 0 Then
        Test if vertex j is an in vertex.
        If (pathList(paths(i, 2), j) Mod 2) = 1 Then
          D2((pathList(paths(i, 2), j) + 1) / 2, pathList(paths(i, 2), j + 1) / 2)
          = D((pathList(paths(i, 2), j) + 1) / 2, pathList(paths(i, 2), j + 1) / 2)
          D2(pathList(paths(i, 2), j + 1) / 2, (pathList(paths(i, 2), j) + 1) / 2)
          = D(pathList(paths(i, 2), j + 1) / 2, (pathList(paths(i, 2), j) + 1) / 2)
          graphPaths(graphPathsIndex, col) = (pathList(paths(i, 2), j) + 1) / 2
          col = col + 1
        End If
      End If
    Next
    Insert last element in path into graphPaths.
    graphPaths(graphPathsIndex, col) = (pathList(paths(i, 2), 2 * col - 1) + 1) / 2
    graphPathsIndex = graphPathsIndex + 1
  Next
End Sub

Private Sub InsertGraphPaths(ByVal paths As Variant, ByVal pathList As Variant)
Inserts paths into the matrix D2.  Also builds a list, graphPaths, containing all paths inserted into the graph.
Dim i, j As Integer
Dim col As Integer
  For i = 1 To k Only insert the cheapest k paths.
    col = 1
    For j = 1 To 2 * p
      If pathList(paths(i, 2), j + 1) <> 0 Then
        Test if vertex j is an out vertex.
        If (pathList(paths(i, 2), j) Mod 2) = 1 Then
          graphPaths(graphPathsIndex, col) = (pathList(paths(i, 2), j) + 1) / 2
          col = col + 1
        End If
      Else
        Exit For
      End If
    Next
    If (j = 1) And (pathList(paths(i, 2), j) = 0) Then
      graphPaths(graphPathsIndex, col) = 0
      graphPathsIndex = graphPathsIndex + 1
    Else
      Insert last element in path into graphPaths.
      graphPaths(graphPathsIndex, col) = (pathList(paths(i, 2), 2 * col - 1) + 1) / 2
      graphPathsIndex = graphPathsIndex + 1
    End If
  Next
End Sub

Private Sub InsertCutSet(ByVal cutSet As Variant)
Inserts all edges between vertices of the cut-set used into the adjacency matrix D2.
Dim i, j As Integer
  For i = 1 To p
    If Mid(cutSet(2), i, 1) = ‘‘1" Then
      For j = i + 1 To p
        If Mid(cutSet(2), j, 1) = ‘‘1" Then
          If D(p - i + 1, p - j + 1) > 0 Then
            D2(p - i + 1, p - j + 1) = D(p - i + 1, p - j + 1)
            D2(p - j + 1, p - i + 1) = D(p - j + 1, p - i + 1)
          End If
        End If
      Next
    End If
  Next
End Sub

Private Function MatrixCost(ByVal matrix, ByVal cost As Single) As Single
Calculates the sum of the weight of all edges stored in matrix.
Dim i, j As Integer
  cost = 0
  For i = 1 To p
    For j = i + 1 To p
      If matrix(i, j) > 0 Then
        cost = cost + matrix(i, j)
      End If
    Next
  Next
  MatrixCost = cost
End Function

Private Function SwopArcs(ByVal DFlow, ByVal nrVertices As Integer) As Variant()
Reverses all arcs in the directed graph represented by the adjacency matrix DFlow.
Dim i, j As Integer
Dim NewDFlow() As Variant
  ReDim NewDFlow(1 To nrVertices, 1 To nrVertices) As Variant
```

```
  For i = 1 To nrVertices
    For j = 1 To nrVertices
      If DFlow(i, j) = 1 Then
        NewDFlow(j, i) = 1
      Else
        NewDFlow(j, i) = 0
      End If
    Next
  Next
  SwopArcs = NewDFlow
End Function

Private Sub cmdFanReduceConn_Click()
  If cutSetIG(1) = 0 Then
    Graph not connected.
    MsgBox (``The graph is not connected.  Please insert more edges into the graph.")
    Exit Sub
  End If
  frmMain.Enabled = False
  frmFanEnterConnNumber.Show
End Sub

Public Sub FanReduceConn()
Constructs a spanning subgraph with a specified connectivity number using Algorithm 15.
Dim i, j, m, n As Integer
Dim ellCutSet As String
Dim totalCost As Single
Dim DCost As Single
  Let U be the cut-set.  Let w be a vertex in V(G)  U. Remove all paths between (k-ell) vertices in U and any w.
  Select ell of the cut-vertices.
  j = 0
  For i = 1 To p
    If (Mid(cutSetIG(2), i, 1) = ``1") And (j < k - ell) Then
      ellCutSet = ellCutSet & ``1"
      j = j + 1
    Else
      ellCutSet = ellCutSet & ``0"
    End If
  Next

  Reconstruct D2.
  For i = 1 To p
    For j = 1 To p
      D2(i, j) = 0
    Next
  Next

  totalCost = 0
  For i = 1 To k * p
    If oldCutVertex(i) = False Then
      m = 1
      While (graphPaths(i, m) <> 0)
        m = m + 1
      Wend
      m = m - 1
      If Mid(ellCutSet, p - graphPaths(i, m) + 1, 1) = ``0" Then If this path does not end in a vertex in ellCutSet.
        For j = 1 To p
          If (j = 1) And (graphPaths(i, j) = 0) Then
            Exit For Special case where vertex in graphpaths(i, j) is actually a cut-vertex and no edges exist.
          End If
          If graphPaths(i, j + 1) <> 0 Then
            If D2(graphPaths(i, j), graphPaths(i, j + 1)) = 0 Then Tests whether edge has already been inserted.
              totalCost = totalCost + D(graphPaths(i, j), graphPaths(i, j + 1))
              D2(graphPaths(i, j), graphPaths(i, j + 1)) = D(graphPaths(i, j), graphPaths(i, j + 1))
              D2(graphPaths(i, j + 1), graphPaths(i, j)) = D(graphPaths(i, j + 1), graphPaths(i, j))
            End If
          Else
            Exit For
          End If
        Next
      End If
    End If
  Next

  Insert paths from vertices in ellCutSet to vertics in reduced cut-set.
  For i = 1 To p
    If Mid(ellCutSet, i, 1) = ``1" Then
      For j = 1 To p
        If (Mid(ellCutSet, j, 1) = ``0") And (Mid(cutSetIG(2), j, 1) = ``1") Then
        If both vertices are not in ellCutSet.
          Insert path from vertex (p − i + 1) to vertex (p − j + 1) .
          m = k * (i - 1) + 1 list graphPaths is sorted from vertex p to vertex 1.  m now points to the first path from
          vertex (p − i + 1) to vertex (p − j + 1).
          Search for correct path to add.  Cut vertex at end must equal (p − j + 1).
```

```
          n = 1
          While graphPaths(m, n) <> p - j + 1
            If graphPaths(m, n) = 0 Then
              m = m + 1
              n = 1
            Else
              n = n + 1
            End If
          Wend
          Path should have been found now.  Insert path.
          For n = 1 To p
            If graphPaths(m, n + 1) <> 0 Then
              If D2(graphPaths(m, n), graphPaths(m, n + 1)) = 0 Then Tests whether edge has already been inserted.
                totalCost = totalCost + D(graphPaths(m, n), graphPaths(m, n + 1))
                D2(graphPaths(m, n), graphPaths(m, n + 1)) = D(graphPaths(m, n), graphPaths(m, n + 1))
                D2(graphPaths(m, n + 1), graphPaths(m, n)) = D(graphPaths(m, n + 1), graphPaths(m, n))
              End If
            Else
              Exit For
            End If
          Next
        End If
      Next
    End If
  Next
  Insert all edges between vertices in the remaining CutSet.
  For i = 1 To p
    If (Mid(cutSetIG(2), i, 1) = ``1") And (Mid(ellCutSet, i, 1) = ``0") Then
      For j = i + 1 To p
        If (Mid(cutSetIG(2), j, 1) = ``1") And (Mid(ellCutSet, j, 1) = ``0") Then
          totalCost = totalCost + D(p - i + 1, p - j + 1)
          D2(p - i + 1, p - j + 1) = D(p - i + 1, p - j + 1)
          D2(p - j + 1, p - i + 1) = D(p - j + 1, p - i + 1)
        End If
      Next
    End If
  Next

  DCost = MatrixCost(D, DCost)

  If DCost > totalCost Then
    Dim xlApp As Excel.Application
    Dim wb As Workbook
    Dim ws As Worksheet

    Set xlApp = New Excel.Application

    Dim nrSheets As Integer
    nrSheets = xlApp.SheetsInNewWorkbook
    xlApp.SheetsInNewWorkbook = 1

    Set wb = xlApp.Workbooks.Add
    xlApp.SheetsInNewWorkbook = nrSheets

    Set ws = wb.Worksheets(1) Specify the worksheet name.
    ws.Name = ``Fan(Lower Conn)"


    ws.Range(``A1:" & xl_Col(p) & p).value = D2

    On Error GoTo errHandler
      wb.SaveAs (Mid(filePath, 1, Len(filePath) - 4) & ``Output.xls")
    On Error GoTo 0
    wb.Close
    xlApp.Quit
    Set ws = Nothing
    Set wb = Nothing
    Set xlApp = Nothing

    txtOutput.Text = txtOutput.Text & ``ALG: Fan" & Chr(13) & Chr(10)
    txtOutput.Text = txtOutput.Text & ``INPUT GRAPH:" & Chr(13) & Chr(10) & `` " & filePath & Chr(13) & Chr(10)
    txtOutput.Text = txtOutput.Text & ``OUTPUT GRAPH:" & Chr(13) & Chr(10) & `` " & Mid(filePath, 1, _
    Len(filePath) - 4) & ``Output.xls" & Chr(13) & Chr(10) & Chr(13) & Chr(10)
    txtOutput.Text = txtOutput.Text & ``Spanning subgraph is " & ell & ``-connected." & Chr(13) & Chr(10)
    txtOutput.Text = txtOutput.Text & ``Weight Improvement = " & DCost - totalCost & Chr(13) & Chr(10) _
    & ``-----------------------------------------------------------------" & Chr(13) & Chr(10)
    cmdNewGraph.Enabled = True

    Enable Algorithm Fan-Reduce connectivity button.
    cmdFanReduceConn.Enabled = True
    cutSetsOutputGraphHasValue = False Used to search for new cut-set.
    cmdFindCutsetOutputGraph.Enabled = False
    Exit Sub errHandler:
    Select Case Err.Number
```

```
        Case 1004
          MsgBox (``The output could not be saved, as the file " & Mid(filePath, 1, Len(filePath) - 4) & ``Output.xls" & ``,
          is currently open.  Please close the file and try again.")
          wb.Close
          xlApp.Quit
          Set ws = Nothing
          Set wb = Nothing
          Set xlApp = Nothing
          Exit Sub
      End Select
   Else
      MsgBox (``A cheaper subgraph could not be found.")
   End If
End Sub

Private Sub cmdFanSpanSubgraph_Click()
Constructs a spanning subgraph with the same connectivity number as the original graph using Algorithm 14.
Dim i, j, m As Integer
Dim DFlow() As Variant
Dim pathList As Variant
Dim nrPaths, pathCount As Integer
Dim cutSetEdgeCount As Integer
Dim reRun As Boolean
Dim paths() As Variant
Dim DCost, D2Cost As Single
Dim distOld() As Variant
Dim cutSetIGList() As Variant
Dim completeCutSetFound As Boolean
Dim cutSetRowCount As Integer
Dim oldCutVertexIndex As Integer
   Initialise button for Algorithm Fan - reduce connectivity to be disabled.
   frmMain.cmdFanReduceConn.Enabled = False
   cutSetsOutputGraphHasValue = False
   Initialise totalCostSaved to 0.
   totalCostSaved = 0

   completeCutSetFound = False
   cutSetsInputGraphHasValue = True
   cutSetIGList = CVSList(D, cutSetRowCount)
   k = cutSetIGList(1, 1)
   If cutSetIGList(1, 1) = 0 Then
      Graph not connected.
      MsgBox (``The graph is not connected.  Please insert more edges into the graph.")
      Exit Sub
   End If
   If cutSetIGList(1, 1) = maxInt Then
      MsgBox (``The graph is a complete graph.  The removal of any edge will reduce the connectivity number of the
      graph.")
      Exit Sub
   End If

   Test if there exists a cut-set that is a complete graph.
   For m = 1 To cutSetRowCount
      cutSetEdgeCount = 0
      For i = 1 To p
         If Mid(cutSetIGList(m, 2), i, 1) = ``1" Then
            For j = i + 1 To p
               If Mid(cutSetIGList(m, 2), j, 1) = ``1" Then
                  If D(p - i + 1, p - j + 1) > 0 Then
                     cutSetEdgeCount = cutSetEdgeCount + 1
                  End If
               End If
            Next
         End If
      Next
      If cutSetEdgeCount = (k * (k - 1)) / 2 Then
         completeCutSetFound = True
         ReDim cutSetIG(1 To 2) As Variant
         cutSetIG(1) = cutSetIGList(m, 1)
         cutSetIG(2) = cutSetIGList(m, 2)
         cutSetsInputGraphHasValue = True
         cutSetOG = cutSetIG
         cutSetsOutputGraphHasValue = True
         Exit For
      End If
   Next
   If completeCutSetFound = False Then
      txtOutput.Text = txtOutput.Text & ``ALG: Fan" & Chr(13) & Chr(10)
      txtOutput.Text = txtOutput.Text & ``Cut-sets Found:" & Chr(13) & Chr(10)
      For m = 1 To cutSetRowCount
         j = 0 Counts the number of cut-vertices found.  Used to deterimine if end of cut-set is reached (hence, a
         full-stop and not a comma should be displayed in the Output window).
         txtOutput.Text = txtOutput.Text & ``Cut-set #" & m & ``:  "
         For i = 1 To p
```

```
        If Mid(cutSetIGList(m, 2), p - i + 1, 1) = ‘‘1" Then
          j = j + 1
          If j < k Then
            txtOutput.Text = txtOutput.Text & i & ‘‘, "
          Else
            txtOutput.Text = txtOutput.Text & i & ‘‘."
          End If
        End If
      Next
      txtOutput.Text = txtOutput.Text & Chr(13) & Chr(10)
    Next
    txtOutput.Text = txtOutput.Text & ‘‘----------------------------------------------------------------" & Chr(13)
    & Chr(10)
    MsgBox (‘‘No complete cut-set could be found.  Please insert the necessary edges for any of the cut-set(s) shown in
    the Output window to form a clique.")
    Exit Sub
  End If
  Enable button cmdFanReduceConn.
  cmdFanReduceConn.Enabled = True
  cmdFindCutsetOutputGraph.Enabled = False

  Clear the matrix D2.
  ReDim D2(1 To p, 1 To p) As Variant
  For i = 1 To p
    For j = 1 To p
      D2(i, j) = 0
    Next
  Next

  ReDim oldCutVertex(1 To k * p) As Boolean
  For i = 1 To p * k
    Initialise oldCutVertex.
    oldCutVertex(i) = False
  Next

  Construct digraph DFlow.
  DFlow = GetDFlow(D)
  Add Sink that is connected to all vertices in the cut-set.
  DFlow = AddSink(DFlow, cutSetIG)
  Get distance weighting for the digraph DFlow.
  dist = Ford(DFlow, 2 * p + 1, 2 * p + 1)
  distOld = dist

  Reverse all arcs.
  DFlow = SwopArcs(DFlow, 2 * p + 1)
  Reset graphPathsIndex.
  graphPathsIndex = 1
  Resize graphPaths list and initialise.  must contain k rows for every vertex not in the cut-set.
  ReDim graphPaths(1 To k * p, 1 To p + 1) As Variant
  For i = 1 To k * p
    For j = 1 To p + 1
      graphPaths(i, j) = 0
    Next
  Next
  For each vertex u not in the cut-set, calculate the set of paths to the cut-set.
  Sort each set and choose k cheapest paths to insert into the graph.
  For i = 1 To p
    oldCutVertexIndex = graphPathsIndex
    dist = distOld
    nrPaths = 0
    Find nr of paths for pathList matrix (min degree of vertex s).
    For j = 1 To p
      If D(p - i + 1, j) > 0 Then Remember that the bit-vector cutSetIG starts with vertex p and ends with vertex 1.
        nrPaths = nrPaths + 1
      End If
    Next
    ReDim pathList(1 To nrPaths, 1 To 2 * p + 1) As Variant
    pathList now empty and ready to populate (2p+1 columns to keep index of for loops when testing for valid edges in
    bounds.  When a path is shorter than 2p vertices, the other open places in pathList are filled with zeros.  Vertex t
    is not included in the paths).
    Clear pathList.
    For m = 1 To nrPaths
      For j = 1 To 2 * p + 1
        pathList(m, j) = 0
      Next
    Next
    pathList = ShortestAugPath(DFlow, 2 * (p - i + 1) - 1, 2 * p + 1, pathList, nrPaths, pathCount, 2 * p + 1, False)
    NOTE: Path starts from in-vertex, not out-vertex (Arcs are reversed, hence, the in-vertex is actually now the
    out-vertex.  addT tests if vertex t should be included on the paths.
    Test if some edges are shared.  If true, remove these edges and call ShortestAugPath again.
    pathList = disjPaths(pathList, pathCount, 2 * (p - i + 1) - 1)
    Sort paths.
    ReDim paths(1 To pathCount, 1 To 3) As Variant
    paths = SortPaths(pathList, pathCount)
```

```
    Insert the cheapest k paths into matrix D2.
    If Mid(cutSetIG(2), i, 1) = ''0" Then
      Call InsertPaths(paths, pathList, k)
    Else
      For j = 1 To k
        oldCutVertex(oldCutVertexIndex) = True
        oldCutVertexIndex = oldCutVertexIndex + 1
      Next
      Call InsertGraphPaths(paths, pathList) This Sub does everything as InsertPaths, exept, it does not insert paths
      into D2.  The paths inserted are used later in Alg.  Fan - reduce connectivity.
    End If
  Next
  Insert cutset into D2.
  Call InsertCutSet(cutSetIG)
  Calculate cost saved.
  DCost = MatrixCost(D, DCost)
  D2Cost = MatrixCost(D2, D2Cost)
  totalCostSaved = DCost - D2Cost
  D2Backup = D2.

  If DCost - D2Cost > 0 Then
    Dim xlApp As Excel.Application
    Dim wb As Workbook
    Dim ws As Worksheet

    Set xlApp = New Excel.Application

    Dim nrSheets As Integer
    nrSheets = xlApp.SheetsInNewWorkbook
    xlApp.SheetsInNewWorkbook = 1

    Set wb = xlApp.Workbooks.Add
    xlApp.SheetsInNewWorkbook = nrSheets

    Set ws = wb.Worksheets(1) Specify the worksheet name.
    ws.Name = ''Fan(Span Subgr)"


    ws.Range(''A1:" & xl_Col(p) & p).value = D2

    On Error GoTo errHandler
      wb.SaveAs (Mid(filePath, 1, Len(filePath) - 4) & ''Output.xls")
    On Error GoTo 0
    wb.Close
    xlApp.Quit
    Set ws = Nothing
    Set wb = Nothing
    Set xlApp = Nothing

    txtOutput.Text = txtOutput.Text & ''ALG: Fan" & Chr(13) & Chr(10)
    txtOutput.Text = txtOutput.Text & ''INPUT GRAPH:" & Chr(13) & Chr(10) & '' " & filePath & Chr(13) & Chr(10)
    txtOutput.Text = txtOutput.Text & ''OUTPUT GRAPH:" & Chr(13) & Chr(10) & '' " & Mid(filePath, 1, _
    Len(filePath) - 4) & ''Output.xls" & Chr(13) & Chr(10) & Chr(13) & Chr(10)
    txtOutput.Text = txtOutput.Text & ''Connectivity Number:  " & k & ''." & Chr(13) & Chr(10)
    txtOutput.Text = txtOutput.Text & ''Weight Improvement = " & DCost - D2Cost & Chr(13) & Chr(10) _
    & ''-----------------------------------------------------------------" & Chr(13) & Chr(10)
    cmdNewGraph.Enabled = True

    Enable Algorithm Fan-Reduce connectivity button.
    cmdFanReduceConn.Enabled = True
    Exit Sub errHandler:
    Select Case Err.Number
      Case 1004
        MsgBox (''The output could not be saved, as the file " & Mid(filePath, 1, Len(filePath) - 4) & ''Output.xls" & '',
        is currently open.  Please close the file and try again.")
        wb.Close
        xlApp.Quit
        Set ws = Nothing
        Set wb = Nothing
        Set xlApp = Nothing
        Exit Sub
    End Select
  Else
    MsgBox (''A cheaper subgraph could not be found.")
  End If

End Sub

Private Sub cmdFindCutsetOriginalGraph_Click()
  If cutSetsInputGraphHasValue = False Then
    cutSetsInputGraphHasValue = True
    cutSetIG = CVS(D)
  End If
  k = cutSetIG(1)
```

```
  If cutSetIG(1) = maxInt Then
    Graph is a complete graph.
    k = p - 1
    MsgBox (``The graph is a complete graph.  Any " + p - 1 + `` vertices may be chosen as a cutset")
    Exit Sub
  End If
  Call DisplayCutSet(cutSetIG, 12)
End Sub

Private Sub cmdFindCutsetOutputGraph_Click()
  If cutSetsOutputGraphHasValue = False Then
    cutSetsOutputGraphHasValue = True
    cutSetOG = CVS(D2)
  End If
  If cutSetOG(1) = maxInt Then
    Graph is a complete graph.
    k = p - 1
    MsgBox (``The graph is a complete graph.  Any " + p - 1 + `` vertices may be chosen as a cutset")
    Exit Sub
  End If
  Call DisplayCutSet(cutSetOG, 14)
End Sub

Private Sub cmdMEEFReduceConn_Click()
  If k = 0 Then
    Graph not connected.
    MsgBox (``The graph is not connected.  Please insert more edges into the graph.")
    Exit Sub
  End If
  frmMain.Enabled = False
  frmMEEFEnterConnNumber.Show
End Sub

Public Sub MEEFReduceConn()
Constructs a spanning subgraph with the same connectivity number as the original graph using Algorithm 13.
Dim i, j, index As Integer
Dim q As Integer
Dim tmpConn As Integer
Dim Dtmp() As Variant
Dim CostSaved, CostSavedTmp As Single
Dim kTmp As Integer
  cmdFindCutsetOutputGraph.Enabled = False
  cutSetsOutputGraphHasValue = False
  Sort edgeList if necessary.
  Count the number of edges.
  q = 0
  For i = 1 To p
    For j = i + 1 To p
      If D(i, j) > 0 Then
        q = q + 1
      End If
    Next
  Next
  If q = 0 Then
  No edges to remove.
    MsgBox (``The graph is an empty graph.  Please add some edges to the input graph first.")
    Exit Sub
  End If
  Now do the sorting.
  index = 1
  If edgeListSorted = False Then
    Build edgeList.
    edgeListSorted = True
    ReDim edgeList(1 To q, 1 To 3) As Variant
    For i = 1 To p
      For j = i + 1 To p
        If D(i, j) > 0 Then
          edgeList(index, 1) = D(i, j)
          edgeList(index, 2) = i
          edgeList(index, 3) = j
          index = index + 1
        End If
      Next
    Next
    Sort edgeList.
    Call QuickSort(edgeList, q, True)
  End If

  Algorithm here.
  index = 0
  CostSaved = 0
  D2 = D
  tmpConn = ConnNumber(D)
  If tmpConn <> 0 Then 'first condition tests whether graph is disconnected
```

```
    While (tmpConn >= ell)
      Dtmp = D2
      CostSavedTmp = CostSaved
      If tmpConn = 0 Then
        Condition to terminate the while loop.
        tmpConn = ell - 1
      Else
        If tmpConn = ell Then
          index = index + 1
          Dtmp = D2
          CostSaved = CostSaved + D2(edgeList(index, 2), edgeList(index, 3))
          D2(edgeList(index, 2), edgeList(index, 3)) = 0
          D2(edgeList(index, 3), edgeList(index, 2)) = 0 Clear both upper and lower triangle of symmetric matrix.
          tmpConn = ConnNumber(D2)
        Else
          For i = 1 To tmpConn - ell
            index = index + 1
            CostSaved = CostSaved + D2(edgeList(index, 2), edgeList(index, 3))
            D2(edgeList(index, 2), edgeList(index, 3)) = 0
            D2(edgeList(index, 3), edgeList(index, 2)) = 0 Clear both upper and lower triangle of symmetric matrix.
          Next
          tmpConn = ConnNumber(D2)
        End If
      End If
    Wend
  End If
  CostSaved = CostSavedTmp
  D2 = Dtmp

  If CostSaved > 0 Then
    Dim xlApp As Excel.Application
    Dim wb As Workbook
    Dim ws As Worksheet

    Set xlApp = New Excel.Application

    Dim nrSheets As Integer
    nrSheets = xlApp.SheetsInNewWorkbook
    xlApp.SheetsInNewWorkbook = 1

    Set wb = xlApp.Workbooks.Add
    xlApp.SheetsInNewWorkbook = nrSheets

    Set ws = wb.Worksheets(1) Specify the worksheet name.
    ws.Name = ``MEEF(Lower Conn)"


    ws.Range(``A1:" & xl_Col(p) & p).value = D2

    On Error GoTo errHandler
      wb.SaveAs (Mid(filePath, 1, Len(filePath) - 4) & ``Output.xls")
    On Error GoTo 0
    wb.Close
    xlApp.Quit
    Set ws = Nothing
    Set wb = Nothing
    Set xlApp = Nothing

    txtOutput.Text = txtOutput.Text & ``ALG: Most Expensive Edge First" & Chr(13) & Chr(10)
    txtOutput.Text = txtOutput.Text & ``INPUT GRAPH:" & Chr(13) & Chr(10) & `` " & filePath & Chr(13) & Chr(10)
    txtOutput.Text = txtOutput.Text & ``OUTPUT GRAPH:" & Chr(13) & Chr(10) & `` " & Mid(filePath, 1, _
    Len(filePath) - 4) & ``Output.xls" & Chr(13) & Chr(10) & Chr(13) & Chr(10)
    txtOutput.Text = txtOutput.Text & ``Spanning subgraph is " & ell & ``-connected." & Chr(13) & Chr(10)
    txtOutput.Text = txtOutput.Text & ``Weight Improvement = " & CostSaved & Chr(13) & Chr(10) _
    & ``-----------------------------------------------------------------" & Chr(13) & Chr(10)

    cmdNewGraph.Enabled = True
    Exit Sub errHandler:
    Select Case Err.Number
      Case 1004
        MsgBox (``The output could not be saved, as the file " & Mid(filePath, 1, Len(filePath) - 4) & ``Output.xls" & ``,
        is currently open.  Please close the file and try again.")
        wb.Close
        xlApp.Quit
        Set ws = Nothing
        Set wb = Nothing
        Set xlApp = Nothing
        Exit Sub
    End Select
  Else
    MsgBox (``A cheaper subgraph could not be found.")
  End If
End Sub
```

```
Private Sub cmdMEEFSpanSubgraph_Click()
Constructs a spanning subgraph with a specified connectivity number using Algorithm 12. Dim i, j, index As Integer
Dim q As Integer
Dim tmpConn As Integer
Dim Dtmp() As Variant
Dim CostSaved As Single
  cmdFindCutsetOutputGraph.Enabled = False
  Sort edgeList if necessary.
  Count the number of edges.
  cutSetsOutputGraphHasValue = False
  q = 0
  For i = 1 To p
    For j = i + 1 To p
      If D(i, j) > 0 Then
        q = q + 1
      End If
    Next
  Next
  If q = 0 Then
  No edges to remove.
    MsgBox (``The graph is an empty graph.  Please add some edges to the input graph first.")
    Exit Sub
  End If
  Now do the sorting.
  index = 1
  If edgeListSorted = False Then
    Build edgeList.
    edgeListSorted = True
    ReDim edgeList(1 To q, 1 To 3) As Variant
    For i = 1 To p
      For j = i + 1 To p
        If D(i, j) > 0 Then
          edgeList(index, 1) = D(i, j)
          edgeList(index, 2) = i
          edgeList(index, 3) = j
          index = index + 1
        End If
      Next
    Next
    Sort edgeList.
    Call QuickSort(edgeList, q, True)
  End If

  index = 0
  CostSaved = 0
  If Not cutSetsInputGraphHasValue Then
    k = ConnNumber(D)
  End If
  If k = 0 Then
    Graph not connected.
    MsgBox (``The graph is not connected.  Please insert more edges into the graph.")
    Exit Sub
  End If
  D2 = D
  tmpConn = k
  While (tmpConn <> 0) And (tmpConn >= k) First condition tests whether graph is disconnected.
    index = index + 1
    Dtmp = D2
    CostSaved = CostSaved + D2(edgeList(index, 2), edgeList(index, 3))
    D2(edgeList(index, 2), edgeList(index, 3)) = 0
    D2(edgeList(index, 3), edgeList(index, 2)) = 0 Clear both upper and lower triangle of symmetric matrix.
    tmpConn = ConnNumber(D2)
  Wend
  CostSaved = CostSaved - D(edgeList(index, 2), edgeList(index, 3)) Use D, as it still contains all the edge costs.
  D2 = Dtmp

  If CostSaved > 0 Then
    Dim xlApp As Excel.Application
    Dim wb As Workbook
    Dim ws As Worksheet

    Set xlApp = New Excel.Application

    Dim nrSheets As Integer
    nrSheets = xlApp.SheetsInNewWorkbook
    xlApp.SheetsInNewWorkbook = 1

    Set wb = xlApp.Workbooks.Add
    xlApp.SheetsInNewWorkbook = nrSheets

    Set ws = wb.Worksheets(1) Specify the worksheet name.
    ws.Name = ``MEEF(Span Subgr)"
```

```
    ws.Range(‘‘A1:" & xl_Col(p) & p).value = D2

    On Error GoTo errHandler
      wb.SaveAs (Mid(filePath, 1, Len(filePath) - 4) & ‘‘Output.xls")
    On Error GoTo 0
    wb.Close
    xlApp.Quit
    Set ws = Nothing
    Set wb = Nothing
    Set xlApp = Nothing

    txtOutput.Text = txtOutput.Text & ‘‘ALG: Most Expensive Edge First" & Chr(13) & Chr(10)
    txtOutput.Text = txtOutput.Text & ‘‘INPUT GRAPH:" & Chr(13) & Chr(10) & ‘‘ " & filePath & Chr(13) & Chr(10)
    txtOutput.Text = txtOutput.Text & ‘‘OUTPUT GRAPH:" & Chr(13) & Chr(10) & ‘‘ " & Mid(filePath, 1, _
    Len(filePath) - 4) & ‘‘Output.xls" & Chr(13) & Chr(10) & Chr(13) & Chr(10)
    txtOutput.Text = txtOutput.Text & ‘‘Connectivity Number:  " & k & ‘‘." & Chr(13) & Chr(10)
    txtOutput.Text = txtOutput.Text & ‘‘Weight Improvement = " & CostSaved & Chr(13) & Chr(10) _
    & ‘‘-------------------------------------------------------------" & Chr(13) & Chr(10)
    cmdNewGraph.Enabled = True
    Exit Sub errHandler:
    Select Case Err.Number
      Case 1004
        MsgBox (‘‘The output could not be saved, as the file " & Mid(filePath, 1, Len(filePath) - 4) & ‘‘Output.xls" & ‘‘,
        is currently open.  Please close the file and try again.")
        wb.Close
        xlApp.Quit
        Set ws = Nothing
        Set wb = Nothing
        Set xlApp = Nothing
        Exit Sub
    End Select
  Else
    MsgBox (‘‘A cheaper subgraph could not be found.")
  End If
End Sub

Private Sub cmdDisplayOriginalGraph_Click()
Displays the original graph in PlotArea.
  If Not DHasValue Then
    DHasValue = True
    Dim xlApp As Excel.Application
    Dim wb As Workbook
    Dim wsDistance, wsXY As Worksheet
    Dim i, j, xTmp, yTmp As Integer
    Dim zeroMatrix() As Integer
    Dim arrDims As Integer

    Set xlApp = New Excel.Application

    Set wb = xlApp.Workbooks.Open(filePath)

    On Error GoTo errHandler
      Set wsDistance = wb.Worksheets(distanceSheet) Specify the worksheet name.
    On Error GoTo 0
    p = wsDistance.Range(‘‘A1").End(xlDown).row
    D = wsDistance.Range(‘‘A1:" & xl_Col(p) & p).value
    On Error GoTo errHandler
      Set wsXY = wb.Worksheets(XYSheet) Specify the worksheet name.
    On Error GoTo 0
    Set wsXY = wb.Worksheets(XYSheet) Specify the worksheet name.
    XY = wsXY.Range(‘‘A1:" & xl_Col(p) & ‘‘2").value
    ReDim zeroMatrix(1 To p, 1 To p) As Integer
    For i = 1 To p
      For j = 1 To p
        zeroMatrix(i, j) = 0
      Next
    Next

    Find the minimum and maximum x and y coordinates.
    xMin = XY(1, 1)
    xMax = XY(1, 1)
    yMin = XY(2, 1)
    yMax = XY(2, 1)
    For i = 1 To p
      If XY(1, i) < xMin Then
        xMin = XY(1, i)
      End If
      If XY(1, i) > xMax Then
        xMax = XY(1, i)
      End If
      If XY(2, i) < yMin Then
        yMin = XY(2, i)
      End If
      If XY(2, i) > yMax Then
```

```
        yMax = XY(2, i)
      End If
    Next

    Scale XY coordinates.
    For i = 1 To p
      XY(1, i) = ((XY(1, i) - xMin) / (xMax - xMin)) * PlotAreaWidth
      XY(2, i) = ((XY(2, i) - yMin) / (yMax - yMin)) * PlotAreaHeight
    Next

    wb.Close
    xlApp.Quit
    Set wsDistance = Nothing
    Set wsXY = Nothing
    Set wb = Nothing
    Set xlApp = Nothing
  End If

  Call DisplayGraph(D, True)
  cmdFindCutsetOriginalGraph.Enabled = True
  cmdMEEFSpanSubgraph.Enabled = True
  cmdMEEFReduceConn.Enabled = True
  cmdFanSpanSubgraph.Enabled = True
  cmdWhitneySpanSubgraph.Enabled = True
  Exit Sub
errHandler:
  Select Case Err.Number
    Case 9
      MsgBox (``The worksheets " & distanceSheet & `` and/or " & XYSheet & `` does not exist in the input file " &
      filePath & `` as specified.  Please ensure that the worksheets are labelled correctly.")
      wb.Close
      xlApp.Quit
      Set wsDistance = Nothing
      Set wsXY = Nothing
      Set wb = Nothing
      Set xlApp = Nothing
      DHasValue = False
      Exit Sub
  End Select
End Sub

Private Sub cmdNewGraph_Click()
Loads a new graph.
  Call DisplayGraph(D2, True)
  cmdFindCutsetOutputGraph.Enabled = True
End Sub

Private Sub cmdClearGraph_Click()
Clears PlotArea.
  plotArea.Cls
End Sub

Private Sub cmdWhitneySpanSubgraph_Click()
Dim i, j As Integer
Dim q As Integer
  q = 0
  For i = 1 To p
    For j = i + 1 To p
      If D(i, j) > 0 Then
        q = q + 1
      End If
    Next
  Next
  If q = 0 Then
  No edges to remove.
    MsgBox (``The graph is an empty graph.  Please add some edges to the input graph first.")
    Exit Sub
  End If

  If Not cutSetsInputGraphHasValue Then
    k = ConnNumber(D)
  End If
  If k = 0 Then
    Graph not connected.
    MsgBox (``The graph is not connected.  Please insert more edges into the graph.")
    Exit Sub
  End If

  frmMain.Enabled = False
  frmWhitneyEnterConnNumber.Show
End Sub

Public Sub WhitneyConnectivity()
Constructs a spanning subgraph with a specified connectivity number using Algorithm 11.
```

```
Dim i, j, m, n As Integer
Dim DFlow(), DFlowOld() As Variant
Dim pathList As Variant
Dim nrPaths, pathCount As Integer
Dim cutSetEdgeCount As Integer
Dim reRun As Boolean
Dim paths() As Variant
Dim DCost, D2Cost As Single
Dim distOld() As Variant
Dim cutSetIGList() As Variant
Dim completeCutSetFound As Boolean
Dim cutSetRowCount As Integer
Dim oldCutVertexIndex As Integer
  Set cutSetsOutputGraphHasValue to False.
  cutSetsOutputGraphHasValue = False
  Clear the matrix D2.
  ReDim D2(1 To p, 1 To p) As Variant
  For i = 1 To p
    For j = 1 To p
      D2(i, j) = 0
    Next
  Next

  Construct digraph DFlow.
  DFlow = GetDFlow(D)
  DFlowOld = DFlow


  Reset graphPathsIndex.
  graphPathsIndex = 1

  If ell > 0 Then
    ReDim graphPaths(1 To ell * (p * (p - 1)) / 2, 1 To p + 1) As Variant p Combination 2 entries exist, each
    consisting of ell paths, hence the first index.
    For i = 1 To ell * (p * (p - 1)) / 2
      For j = 1 To p + 1
        graphPaths(i, j) = 0
      Next
    Next

    For i = 1 To p
      For j = i + 1 To p
        Get distance weighting for the digraph DFlow.  Distance labels need to be recalculated, as Shortest Augmenting
        Paths algorithm changes these labels.
        dist = Ford(DFlow, 2 * (p - i + 1) - 1, 2 * p)

        DFlow = DFlowOld
        Reverse all arcs.
        DFlow = SwopArcs(DFlow, 2 * p)

        nrPaths = 0
        Find nr of paths for pathList matrix (min degree of vertex s).
        For m = 1 To p
          If D(p - i + 1, m) > 0 Then Remember that the bit-vector cutSetIG starts with vertex p and ends with vertex 1.
            nrPaths = nrPaths + 1
          End If
        Next
        ReDim pathList(1 To nrPaths, 1 To 2 * p + 1) As Variant pathList now empty and ready to populate (2p + 1)
        columns to keep index of for loops when testing for valid edges in bounds.  When a path is shorter than 2p
        vertices, the other open places in pathList are filled with zeros.  Vertex t is not included in the paths).
        Clear pathList.
        For m = 1 To nrPaths
          For n = 1 To 2 * p + 1
            pathList(m, n) = 0
          Next
        Next
        pathList = ShortestAugPath(DFlow, 2 * (p - i + 1) - 1, 2 * (p - j + 1) - 1, pathList, nrPaths, pathCount, _
        2 * p, True) NOTE: Path starts from in-vertex, not out-vertex (Arcs are reversed, hence, the in-vertex is
        actually now the out-vertex).
        Test if some edges are shared.  If true, remove these edges and call ShortestAugPath again.
        pathList = disjPathsWhitney(pathList, pathCount, 2 * (p - i + 1) - 1, 2 * (p - j + 1) - 1)
        Sort paths.
        ReDim paths(1 To pathCount, 1 To 3) As Variant
        paths = SortPaths(pathList, pathCount)
        Insert the cheapest ell paths into matrix D2.
        Call InsertPaths(paths, pathList, ell)
      Next
    Next
  End If

  DCost = MatrixCost(D, DCost)
  D2Cost = MatrixCost(D2, D2Cost)
  totalCostSaved = DCost - D2Cost
```

```
If DCost - D2Cost > 0 Then
    Dim xlApp As Excel.Application
    Dim wb As Workbook
    Dim ws As Worksheet

    Set xlApp = New Excel.Application

    Dim nrSheets As Integer
    nrSheets = xlApp.SheetsInNewWorkbook

    xlApp.SheetsInNewWorkbook = 1

    Set wb = xlApp.Workbooks.Add
    xlApp.SheetsInNewWorkbook = nrSheets

    Set ws = wb.Worksheets(1) Specify the worksheet name.
    ws.Name = ``Whitney(Span Subgr)"


    Set ws = wb.Worksheets.Add(``MostExpEdgeFirst").
    ws.Range(``A1:" & xl_Col(p) & p).value = D2

    On Error GoTo errHandler
        wb.SaveAs (Mid(filePath, 1, Len(filePath) - 4) & ``Output.xls")
    On Error GoTo 0
    wb.Close
    xlApp.Quit
    Set ws = Nothing
    Set wb = Nothing
    Set xlApp = Nothing

    txtOutput.Text = txtOutput.Text & ``ALG: Whitney" & Chr(13) & Chr(10)
    txtOutput.Text = txtOutput.Text & ``INPUT GRAPH:" & Chr(13) & Chr(10) & `` " & filePath & Chr(13) & Chr(10)
    txtOutput.Text = txtOutput.Text & ``OUTPUT GRAPH:" & Chr(13) & Chr(10) & `` " & Mid(filePath, 1, _
    Len(filePath) - 4) & ``Output.xls" & Chr(13) & Chr(10) & Chr(13) & Chr(10)
    txtOutput.Text = txtOutput.Text & ``Spanning subgraph is " & ell & ``-connected." & Chr(13) & Chr(10)
    txtOutput.Text = txtOutput.Text & ``Weight Improvement = " & DCost - D2Cost & Chr(13) & Chr(10) & _
    ``-----------------------------------------------------------------" & Chr(13) & Chr(10)
    cmdNewGraph.Enabled = True

    Exit Sub errHandler:
    Select Case Err.Number
        Case 1004
            MsgBox ("The output could not be saved, as the file " & Mid(filePath, 1, Len(filePath) - 4) & "Output.xls" & ",
            is currently open.  Please close the file and try again.")
            wb.Close
            xlApp.Quit
            Set ws = Nothing
            Set wb = Nothing
            Set xlApp = Nothing
            Exit Sub
    End Select
Else
    MsgBox (``A cheaper subgraph could not be found.")
End If
End Sub

Private Sub Form_Load()
Dim tmp As String
Dim tmp2 As Variant
Dim tmp3() As Variant
    maxInt = 2^31 - 1
    edgeListSorted = False

    DHasValue = False
    cutSetsInputGraphHasValue = False
    cutSetsOutputGraphHasValue = False

    filePath = ""

End Sub

Private Sub mnuInputFile_Click()
    frmMain.cmdDisplayOriginalGraph.Enabled = False
    frmMain.cmdFindCutsetOriginalGraph.Enabled = False
    frmMain.cmdFindCutsetOutputGraph.Enabled = False
    frmMain.cmdMEEFSpanSubgraph.Enabled = False
    frmMain.cmdMEEFReduceConn.Enabled = False
    frmMain.cmdFanSpanSubgraph.Enabled = False
    frmMain.cmdFanReduceConn.Enabled = False
    frmMain.cmdWhitneySpanSubgraph.Enabled = False
    frmMain.cmdNewGraph.Enabled = False
    frmMain.plotArea.Cls
```

```
  DHasValue = False
  edgeListSorted = False
  cutSetsInputGraphHasValue = False
  cutSetsOutputGraphHasValue = False

  frmMain.Enabled = False
  frmInput.Show
End Sub

Option Explicit

Private Declare Function GetSystemMenu Lib "user32"(ByVal hWnd As Long, _
  ByVal bRevert As Long) As Long
Private Declare Function RemoveMenu Lib "user32" (ByVal hMenu As Long, _
  ByVal nPosition As Long, ByVal wFlags As Long) As Long

Remove the Close menu item and disable the Close button from a window.
Public Sub RemoveCloseMenuItem(ByVal hWnd As Long)
  Const SC_CLOSE = &HF060
  Const MF_BYCOMMAND = 0

  Dim hMenu As Long
  Get the system menu's handle.
  hMenu = GetSystemMenu(hWnd, 0)
  Remove the Close item.
  RemoveMenu hMenu, SC_CLOSE, MF_BYCOMMAND
End Sub

Private Sub cmdCancel_Click()
  frmMain.Enabled = True
  Unload Me
End Sub

Private Sub cmdOK_Click()
  If frmMain.filePath = "" Then
    MsgBox ("No input file Selected.  Please Select an input file")
  Else
    frmMain.distanceSheet = txtDistance.Text
    frmMain.XYSheet = txtXY.Text

    frmMain.Enabled = True
    frmMain.cmdDisplayOriginalGraph.Enabled = True
    Unload Me
  End If
End Sub

Private Sub Dir1_Change()
  File1.Path = Dir1.Path
End Sub

Private Sub Drive1_Change()
  Dir1.Path = Drive1.Drive
End Sub

Private Sub File1_Click()
  frmMain.filePath = Dir1.Path
  If Right$(Dir1.Path, 1) <> "\" Then frmMain.filePath = frmMain.filePath & "\"
  frmMain.filePath = frmMain.filePath & File1.FileName
  lblFilePath.Caption = File1.FileName
End Sub

Private Sub Form_Load()
  Call RemoveCloseMenuItem(frmInput.hWnd)
End Sub

Option Explicit

Private Declare Function GetSystemMenu Lib "user32" (ByVal hWnd As Long, _
  ByVal bRevert As Long) As Long

Private Declare Function RemoveMenu Lib "user32" (ByVal hMenu As Long, _
  ByVal nPosition As Long, ByVal wFlags As Long) As Long

Remove the Close menu item and disable the Close button from a window.

Public Sub RemoveCloseMenuItem(ByVal hWnd As Long)
  Const SC_CLOSE = &HF060
  Const MF_BYCOMMAND = 0

  Dim hMenu As Long
  Get the system menu's handle.
  hMenu = GetSystemMenu(hWnd, 0)
  Remove the Close item.
  RemoveMenu hMenu, SC_CLOSE, MF_BYCOMMAND
```

```
End Sub

Private Sub cmdCancel_Click()
  frmMain.Enabled = True
  Unload Me
End Sub

Private Sub cmdOK_Click()
  If txtEll = "" Then
    MsgBox ("Please enter a value for the desired connectivity number.")
  Else
    On Error GoTo errHandler
    If (txtEll.Text >= "0") And (txtEll.Text <= frmMain.k) Then
      frmMain.ell = txtEll.Text
      frmMain.Enabled = True
      Call frmMain.WhitneyConnectivity
      Unload Me
    Else
      MsgBox ("The value entered must be between 0 and " & frmMain.k & ".")
    End If
    On Error GoTo 0
  End If
  Exit Sub
errHandler:
    Select Case Err.Number
      Case 13
        MsgBox ("Please enter a value in the specIfied range.")
        Exit Sub
    End Select
End Sub

Private Sub Form_Load()
  Call RemoveCloseMenuItem(frmWhitneyEnterConnNumber.hWnd)
  lblRange.Caption = "(Min:  0, Max:  " & frmMain.k & ")"
End Sub

Option Explicit
Private Declare Function GetSystemMenu Lib "user32" (ByVal hWnd As Long, _
  ByVal bRevert As Long) As Long
Private Declare Function RemoveMenu Lib "user32" (ByVal hMenu As Long, _
  ByVal nPosition As Long, ByVal wFlags As Long) As Long

Remove the Close menu item and disable the Close button from a window.

Public Sub RemoveCloseMenuItem(ByVal hWnd As Long)
  Const SC_CLOSE = &HF060
  Const MF_BYCOMMAND = 0

  Dim hMenu As Long
  Get the system menu's handle.
  hMenu = GetSystemMenu(hWnd, 0)
  Remove the Close item.
  RemoveMenu hMenu, SC_CLOSE, MF_BYCOMMAND
End Sub

Private Sub cmdCancel_Click()
  frmMain.Enabled = True
  Unload Me
End Sub

Private Sub cmdOK_Click()
  If txtEll = "" Then
    MsgBox ("Please enter a value for the desired connectivity number.")
  Else
    On Error GoTo errHandler
    If (txtEll.Text >= "0") And (txtEll.Text <= frmMain.k) Then
      frmMain.ell = txtEll.Text
      frmMain.Enabled = True
      Call frmMain.MEEFReduceConn
      Unload Me
    Else
      MsgBox ("The value entered must be between 0 and " & frmMain.k & ".")
    End If
    On Error GoTo 0
  End If
  Exit Sub
errHandler:
    Select Case Err.Number
      Case 13
        MsgBox ("Please enter a value in the specIfied range.")
        Exit Sub
    End Select
End Sub
```

```
Private Sub Form_Load()
  Call RemoveCloseMenuItem(frmMEEFEnterConnNumber.hWnd)
  lblRange.Caption = "(Min:  0, Max:  " & frmMain.k & ")"
End Sub

Option Explicit

Private Declare Function GetSystemMenu Lib "user32" (ByVal hWnd As Long, _
  ByVal bRevert As Long) As Long
Private Declare Function RemoveMenu Lib "user32" (ByVal hMenu As Long, _
  ByVal nPosition As Long, ByVal wFlags As Long) As Long

Remove the Close menu item and disable the Close button from a window.

Public Sub RemoveCloseMenuItem(ByVal hWnd As Long)
  Const SC_CLOSE = &HF060
  Const MF_BYCOMMAND = 0

  Dim hMenu As Long
  Get the system menu's handle.
  hMenu = GetSystemMenu(hWnd, 0)
  Remove the Close item.
  RemoveMenu hMenu, SC_CLOSE, MF_BYCOMMAND
End Sub

Private Sub cmdCancel_Click()
  frmMain.Enabled = True
  Unload Me
End Sub

Private Sub cmdOK_Click()
  If txtEll = "" Then
    MsgBox ("Please enter a value for the desired connectivity number.")
  Else
    On Error GoTo errHandler
    If (txtEll.Text >= "0") And (txtEll.Text <= frmMain.k) Then
      frmMain.ell = txtEll.Text
      frmMain.Enabled = True
      Call frmMain.FanReduceConn
      Unload Me
    Else
      MsgBox ("The value entered must be between 0 and " & frmMain.k & ".")
    End If
    On Error GoTo 0
  End If
  Exit Sub
errHandler:
    Select Case Err.Number
      Case 13
        MsgBox ("Please enter a value in the specIfied range.")
        Exit Sub
    End Select
End Sub

Private Sub Form_Load()
  Call RemoveCloseMenuItem(frmFanEnterConnNumber.hWnd)
  lblRange.Caption = "(Min:  0, Max:  " & frmMain.k & ")"
End Sub
```

Index

$F(x,U)$ fan, 4, 55
ℓ-connectivity number, 25
ℓ-connectivity sequence, 26
ℓ-edge-connectivity number, 25
ℓ-way cut, 25
    minimum, 25
k-connected, 20
    critically, 24
    minimally, 24
k-edge-connected, 20
    critically, 24
    minimally, 24
n-partite, 9
    complete, 9
    complete, balanced, 9
r-regular, 7

adjacency, 3
adjacency matrix, 7, 42
algorithm, 12
algorithmic complexity, 12
    polynomial time reducible, 14
    space complexity, 12
    the class *co-NP*, 13
    the class *NP*, 13
    the class *NP*-complete, 14, 64
    the class *NP*-hard, 14
    the class *P*, 13
    time complexity, 12
approximation algorithm, 29
approximation ratio, 29
automorphism, 5
average connectivity number, 26
    maximum, 28

basic operations, 12
bipartite, 9
bit-vector, 32
breadth-first search, 36
bridge, 10, 17

centre, 9
certificate, 13
circuit, 4
clique, 12, 55
clique number, 12
closed neighbourhood, 4
closed walk, 4
complete n-partite, 9
complete graph, 8
complexity, 12
component, 10, 53, 56
computation problem, 15
connected, 10
connected graph, 10, 32
connectivity number, 17, 31
connectivity preserving, 29, 31
connectivity reducing, 29, 31
cut-set
    edge cut-set, 17
    minimum, 32
    vertex cut-set, 17
cut-vertex, 10, 17
cycle, 4, 7
    even, 7
    odd, 7

decision problem, 13
decision theory, 13
degree, 4
    maximum, 5
    minimum, 5
degree sequence, 5
deletion, 5
digraph, 3
    strongly (n,ℓ)-connected, 26
directed graph, 3, 38
disconnected graph, 10

edge
    join, 3
    weight, 6
edge contraction, 19
edge cut-set, 17
edge disjoint, 18
edge weight, 6
edge-connectivity number, 17
edge-separator, 18
edges, 3
end-vertex, 5
equality of graphs, 5

flow units, 38
forest, 9
Fundamental Theorem of Graph Theory, 5

graph, 3
  n-partite, 9
  bipartite, 9
  complete graph, 8
  digraph, 3
  directed graph, 3
  edges, 3
  join, 7
  multipartite, 9
  null graph, 8
  pseudograph, 10
  simple graph, 3
  size, 3
  star, 9
  tree, 9
  trivial graph, 3
  vertices, 3

hub, 10

in-neighbour, 4
in-vertex, 41
incident, 3
independence, 11
  maximal, 11
  number, 11
induced subdigraph, 5
induced subgraph, 5
internal vertex, 4
internally disjoint, 4, 18
interval halving scheme, 15
intractable problem, 12
isolated vertex, 5
isomorphic, 5
isomorphism, 5

leaf, 9
length, 4, 45
loop, 10

matching, 7
  maximum, 7
  number, 7
  perfect, 7
maximal independence, 11
maximum degree, 5
maximum flow, 38
minimum degree, 5
multipartite, 9

neighbourhood, 4
  closed, 4
  open, 4
null graph, 8

open neighbourhood, 4
open walk, 4
order, 3, 4
order of magnitude, 12
out-neighbour, 4, 36
out-vertex, 41

path, 4, 7, 45
  even, 7
  odd, 7
polynomial time, 12
polynomial time reducible, 14
pseudograph, 10

regular, 7
residual capacity, 39

sequence of strong connectivity numbers, 26
simple graph, 3
sink, 36
size, 3
source, 36
spanning subdigraph, 5
spanning subgraph, 5, 6, 49, 51, 55
spokes, 10
star graph, 9
strong ℓ-arc-connectivity number, 26
strong ℓ-connectivity number, 26
strong component, 26
strong independce number, 26
strongly connected, 26
subdigraph, 5
  induced, 5
  spanning, 5
subgraph, 5
  induced, 5
  spanning, 5, 49, 51, 55
supersink, 39

tractable problem, 13
trail, 4
tree, 9
  forest, 9
  leaf, 9
  trivial tree, 9
trivial graph, 3
trivial tree, 9

uniformly k-connected, 28
union, 7
units, 38

vertex
  adjacent vertices, 3, 41
  degree, 4
vertex connectivity number, 17
vertex cut-set, 17
vertex splitting, 11
vertex-separator, 18

vertices, 3

walk, 4
wheel, 10
    hub, 10
    spokes, 10