Graph Connectivity: Relaxations and Algorithms

by

Adrian Roshan Vetta

B.Sc. Economics, London School of Economics 1994
M.Sc. Mathematics, London School of Economics 1995

Submitted to the Department of Mathematics
in partial fulfillment of the requirements for the degree of

Doctor of Philosophy

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

June 2002

Author ....
Department of Mathematics
May 1, 2002

Certified by ....
Santosh Vempala
Assistant Professor
Thesis Supervisor

Accepted by ....
Chair, Applied Mathematics Committee

Accepted by ....
Tomasz S. Mrowka
Chair, Department Committee on Graduate Students

Graph Connectivity: Relaxations and Algorithms

by

Adrian Roshan Vetta

Submitted to the Department of Mathematics
on May 1, 2002, in partial fulfillment of the
requirements for the degree of
Doctor of Philosophy

Abstract

The task of finding minimum cost subgraphs with prescribed connectivity requirements is a fundamental problem in network optimisation. In general, though, this task is NP-hard. As a consequence, there has been much interest in the development of approximation algorithms for such network problems. This thesis focuses predominantly upon the design of approximation algorithms for the two classical problems of this genre: the k-edge connectivity problem and the k-vertex connectivity problem. For the k-vertex connectivity problem in undirected graphs we present two different types of algorithm. The first is a k-phase augmentation algorithm which increases the vertex connectivity of a solution subgraph by one in each phase. The second algorithm is based upon the technique of iteratively rounding a linear program. The iterative rounding method also works for the k-vertex connectivity problem in directed graphs, and for the element connectivity problem. For the special case of bi-connectivity problems in unweighted graphs, we present combinatorial algorithms that give improved approximation guarantees. These algorithms have two broad phases. In the initial phase decomposition methods are applied which allow us to focus upon subgraphs with a simplified structure. In the final phase a feasible solution is obtained by combining techniques from matching theory with local search methods.

Thesis Supervisor: Santosh Vempala
Title: Assistant Professor

Acknowledgments

I am very grateful to my advisor Santosh Vempala. I couldn't have hoped for a better advisor than Santosh; he is also a very nice person. I would also like to thank the other members of my thesis committee, Dimitris Bertsimas and Michel Goemans, for their comments and insights regarding this thesis. I am also grateful to Bruce Shepherd, Madhav Marathe and Joseph Cheriyan for an assortment of reasons.

Thanks go to my old flatmates Pete and Francis for the odd pint, and to Pablo for the footy matches. The visits from home of the following people served as a sanity check: Amber, Amita, Anne, Billy, dad, Dave, Elysia, Justin, Konni, Lambert, Messer, mum, Paul, Ralphie, Rohit, Simmy and Sohit. Last, but not least, thanks to Juli.

Contents

Chapter 1

Introduction

Let $G = (V, E)$ be a graph (undirected or directed), with n vertices and m edges. Associated with each edge e is a cost c_e. Given this framework, the task of finding low cost spanning subgraphs of a prescribed connectivity is a fundamental problem in network optimisation. This thesis focuses predominantly upon the two classical problems of this genre: the *k-edge connectivity problem* (kEC) and the *k-vertex connectivity problem* (kVC). In undirected graphs, the k-edge connectivity problem is to find the minimum cost subgraph in which there are at least k edge disjoint paths between every pair of vertices. In the k-vertex connectivity problem we require at least k vertex disjoint paths between every pair of vertices. For $k = 1$ we have the *minimum spanning tree problem*, which can be solved exactly in polynomial time. In general, however, these problems are hard. For example, even the problems of finding the minimum cardinality 2-edge connected spanning subgraph ($2EC$) or 2-vertex connected spanning subgraph ($2VC$) are NP-hard. This can be seen via a simple reduction from the Hamiltonian cycle problem. Moreover, it is known that both these problem are maxSNP-hard ([9], [3]). In directed graphs, the k-edge connectivity problem is to find the minimum cost subgraph in which, for each pair of vertices u and v, there are at least k edge disjoint directed paths from u to v, and at least k edge disjoint directed paths from v to u. The directed version of the k-vertex connectivity problem is defined analogously.

Given the hardness results, this thesis will concentrate on approximation algorithms for connectivity problems. Hence, we desire subgraphs whose cost is guaranteed to be within a certain factor of that of the optimal solution. An overview of this thesis is as follows. In Chapter 2 we consider connectivity problems in weighted graphs. We begin by presenting a linear programming formulation (with constraints upon the edges induced by two disjoint subsets of vertices) that proves useful in the design and analysis of connectivity algorithms. Since there exist very good approximation algorithms for edge connectivity problems in weighted graphs (see, for example, Jain [25]), our focus will be on vertex connectivity problems. In particular, we will concentrate on the k-vertex connectivity problem. We give two different types of algorithm. The first algorithm, given in Section 2.2, is a k phase augmentation algorithm which increases the vertex connectivity of a solution subgraph by one in each phase. The key structural result, here, is that in phase r, $1 \leq r \leq k$, a suitable r-vertex connected subgraph can be obtained from the union of three easily constructed subgraphs. The performance of the algorithm is analysed via the linear programming relaxation. This algorithm gives an $O(\log k)$ approximation guarantee in undirected graphs containing at least $6k^2$ vertices.

The second algorithm, given in Section 2.3, is based upon the iterative rounding of the setpair linear program (LP). At each step in the algorithm an edge is judiciously selected (by consideration of the weight given to the edge by the LP solution) until a k-vertex connected subgraph is obtained. The existence, at each step, of a high weight edge in the LP solution is sufficient to produce a good approximation guarantee. It is shown that for a large class of functions a basic LP solution is characterised by a "non-overlapping family" of setpairs. Such families, in turn, guarantee the existence of a high weight edge. In particular, for the k-vertex connectivity problem iterative rounding gives an $O(\sqrt{\frac{n}{\varepsilon}})$ approximation guarantee in both undirected and directed graphs, where $k \leq (1-\varepsilon)n$. The previous best approximation guarantee for the undirected k-vertex connectivity problem, due to Kortsarz and Nutov [31], was $O(k)$. For directed graphs the only known non-trivial guarantee was $O(\sqrt{m})$, due to Cheriyan and Vempala [6]. We also show how the iterative rounding method can be applied to give a factor 2 approximation guarantee for the *element connectivity* *problem* (to be defined). Chapter 2 is based upon joint work with Joseph Cheriyan and Santosh Vempala.

In Chapter 3 we consider unweighted graphs, that is, graphs for which $c_e = 1$ for each edge e. Here we present improved approximation guarantees for various bi-connectivity problems. The basic technique is as follows. We apply an assortment of decomposition theorems that allow us to focus upon a class of graphs with a simplified structure. Then, matching theory techniques are used to find subgraphs from which bi-connected subgraphs can be constructed via local improvement methods. These methods are shown to produce factor $\frac{4}{3}$-approximation algorithms for the 2-edge connectivity and 2-vertex connectivity problems in undirected graphs. These results improve upon the previous best guarantees of $\frac{17}{12}$ and $\frac{3}{2}$, due to Cheriyan et al [5] and Garg et al [18], respectively. These methods also produce a factor $\frac{3}{2}$-approximation algorithm for the *strong connectivity problem* (this is the directed 1-edge (vertex) connectivity problem) in directed graphs. This approximation guarantee improves upon the previous best guarantee of 1.617 due to Khuller et al [29]. The results in Chapter 3 regarding undirected graphs are based upon work with Santosh Vempala.

Chapter 2

Vertex Connectivity in Weighted Graphs

In this chapter, we consider connectivity problems in weighted graphs; the results presented here are drawn from the papers [7] and [8]. As stated, our focus will be on approximations algorithms. Good approximation algorithms exist for weighted edge connectivity problems in undirected graphs. In particular, Jain [25] has shown that the technique of iterative rounding may be applied to the cut covering LP formulation, described in Section 2.1, of the generalised Steiner network problem to give a factor 2-approximation algorithm. Consequently, we will focus on vertex connectivity problems in this chapter.

In particular, we present two approximation algorithms for the k-vertex connectivity problem. The first algorithm works by successively augmenting by one the connectivity of an initial subgraph until we obtain a feasible solution. The second algorithm is based upon iterative rounding of the setpair LP relaxation. The augmentation algorithm only works on undirected graphs; the iterative rounding algorithm works for both undirected and directed graphs. We briefly discuss how our results compare with to the previous best approximation guarantee of more than $k/2$ due to Kortsarz and Nutov [31]. The connectivity augmentation algorithm has an approximation guarantee of $O(\log k)$. This algorithm, however, only works on graphs with at least $6k^2$ vertices. Thus the guarantee is a strict improvement on the previous guarantee for $56 \leq k \leq \sqrt{n/6}$, and is comparable (within a constant factor) for $k < 56$. This algorithm also provides, for graphs with at least $6k^2$ vertices, an $O(\log k)$ upper bound on the integrality ratio for the setpair LP relaxation. The iterative rounding algorithm provides an approximation guarantee of $O(\sqrt{n/\epsilon})$, for any $\epsilon > 0$ and $k \leq (1-\epsilon)n$. This is strict improvement for $\Omega(\sqrt{n}) \leq k \leq n - \Omega(1)$; it is comparable for $k = \theta(\sqrt{n})$ and $k \geq n - \theta(1)$.

2.1 Setpair Relaxations

In this section we present two integer program formulations that are useful in network design problems. The first formulation has been widely used in edge connectivity problems. The second formulation generalises the first formulation and can be used for vertex connectivity problems.

2.1.1 The Cut Covering Model

Our first formulation is known as the *cut-covering model*. Here there is a requirement function f that places a constraint upon the edges $\delta(S)$ induced by each cut $(S, V \setminus S)$ in the graph.

$$
\begin{array}{llrcll}
 & \min & \sum_{e \in E} c_e x_e & & & \\
\text{s.t.} & & \sum_{e \in \delta(S)} x_e & \geq & f(S) & \forall\, S \subseteq V \\
 & & x_e & \in & \{0,1\} & \forall\, e \in E
\end{array}
$$

Now consider the following fundamental theorem of Menger.

Theorem 2.1.1. *The maximum number of pairwise edge-disjoint (i,j)-paths equals the minimum number of edges whose deletion separates i and j.* □

By Menger's Theorem, the k-edge connectivity problem may be given using the cut covering formulation. Simply set $f(S) = k$, for all S, $\emptyset \subset S \subset V$. In fact, the *generalised Steiner network problem* may also be formulated via the cut covering model. Here, given two vertices i and j, we require r_{ij} edge disjoint paths between i and j. Setting $f(S) = \max_{i \in S, j \in V \setminus S} r_{ij}$ allows us to use the cut covering model.

2.1.2 The Setpair Model

A more general integer programming formulation, developed by Frank and Jordan [12], is the *setpair model.* A *setpair* $W = (W_t, W_h)$ is an ordered pair of disjoint vertex sets. We say that W_t is the *tail* of W, and W_h is the *head* of W. Let $\mathcal{S}$ denote the set of all setpairs. We abuse our notation slightly and denote by $\delta(W) = \delta(W_t, W_h)$ the set of edges with one end-vertex in W_t and the other in W_h. We set $W' = V \setminus (W_t \cup W_h)$. Then, given a requirement function f that places a constraint upon each edge set induced by a setpair, we obtain the following integer program.

$$
\begin{array}{llrcll}
 & \min & \displaystyle\sum_{e \in E} c_e x_e & & & \\
\text{s.t.} & & \displaystyle\sum_{e \in \delta(W)} x_e & \geq & f(W) & \forall\, W \in \mathcal{S} \\
 & & x_e & \in & \{0,1\} & \forall\, e \in E
\end{array}
$$

Note that a cut $(W_t, V \setminus W_t)$ is induced when the head and tail of W form a complementary pair of vertex sets. Thus, the cut-covering model is just a special case of the setpair model in which requirements are only placed upon setpairs W for which $|W'| = 0$. We remark that Menger's Theorem can be re-formulated in terms of vertex-disjoint paths.

Theorem 2.1.2. *For non-adjacent vertices i and j, the maximum number of pairwise (internally) vertex-disjoint (i,j)-paths equals the minimum number of vertices from $V \setminus \{i,j\}$ whose deletion separates i and j.* $\square$

Then, by Menger's Theorem, the k-vertex connectivity problem may be given using the cut-covering formulation. Simply set $f(W) = \max(k - |W'|, 0)$, for each setpair W.

Given the complexity of our connectivity problems, we do not expect to be able to solve these integer programs exactly in polynomial time. Thus, the focus of our attention will be their linear program relaxations. Observe that there are an exponential number of constraints in these relaxations. However, for many of the specific problems in network design that are captured by these models, the linear programming relaxation can be written as a compact linear program via a "flow formulation". As we shall see, this is true for the k-edge and k-vertex connectivity problems. Moreover, optimal solutions can be computed in strongly polynomial time [38].

2.2 Connectivity Augmentation

In this section, we present an approximation algorithm for the k-vertex connected subgraph problem. The algorithm consists of k phases; during each phase the vertex connectivity of the current solution is augmented by one. The algorithm works under the restriction that $n \geq 6k^2$; the method fails for graphs of smaller order. The approximation guarantee is 6 times the kth harmonic number, $H(k)$, which is $O(\log k)$. This is also an upper bound on the integrality ratio of the setpair LP for the k-vertex connectivity problem. The previous best upper bound on the integrality ratio was $\Theta(k)$. (An $O(\log k)$ approximation guarantee was claimed earlier in [37], but subsequently an error has been found and that claim is withdrawn.) Our algorithm is based on two results:

(i) A polynomial-time algorithm of Frank and Tardos [13] for finding a minimum cost k-outconnected subgraph in a directed graph.

(ii) An upper bound on the order of 3-critical graphs by Mader [32].

The Frank-Tardos algorithm has been applied earlier to the minimum cost k-vertex connected subgraph problem by several authors, beginning with Khuller and Raghavachari [27]; see also [1] and [2]. The scaling trick used in Lemma 2.2.4 has been used earlier by Goemans et al [20], and also by Jain et al [26].

2.2.1 The k-Outconnected Subgraph Problem

Consider again the setpair formulation of the k-vertex connectivity problem. It will be convenient to keep a parameter ℓ, where ℓ is a positive integer, and denote by $P(\ell)$ the LP relaxation of the ℓ-vertex connectivity problem.

$$
\begin{array}{lrcll}
P(\ell): & z(\ell) & = & \min \displaystyle\sum_{e \in E} c_e\, x_e & \\
& \text{s.t.} & & \displaystyle\sum_{e \in \delta(W)} x_e \ \geq\ \ell - |W'| & \forall\, W \in \mathcal{S} \\
& & & 0 \ \leq\ x_e \ \leq\ 1 & \forall\, e \in E
\end{array}
$$

Let $\textsc{opt}(\ell)$ denote the cost of a minimum cost ℓ-vertex connected subgraph. Then $P(\ell)$ gives a lower bound $z(\ell)$ on the cost $\textsc{opt}(\ell)$. In particular, $P(k)$ gives a lower bound $z(k)$ on the cost $\textsc{opt} = \textsc{opt}(k)$ of an optimal solution.

A graph is said to be *k-outconnected* from a *root* vertex r if there exist k openly disjoint paths from r to v, for each vertex v, $v \neq r$. The *minimum cost k-outconnected subgraph problem* is as follows: given an undirected graph $G = (V, E)$ and a root vertex $r \in V$, find a minimum cost subgraph that is k-outconnected from r. This problem is NP-hard for $k \geq 2$. A factor 2 approximation algorithm has, though, been given by Frank and Tardos [13] (see also Khuller and Raghavachari [27]).

Theorem 2.2.1. *There is an algorithm for the minimum cost ℓ-outconnected subgraph problem that produces a solution with cost at most $2z(\ell)$.*

Proof. In the directed version $\widehat{G}$ of G, each edge e of G is replaced by two oppositely oriented arcs, and each of these two arcs has cost c_e. The LP relaxation of the directed minimum cost ℓ-vertex connected subgraph problem on $\widehat{G}$ has optimal value at most $2z(\ell)$; the directed subgraph corresponding to the undirected solution of $P(\ell)$ has double the cost and is feasible for the directed LP.

An optimal solution to this LP is also a feasible solution to the following LP relaxation $\widehat{P}(\ell)$ of the directed minimum cost ℓ-outconnected subgraph problem on $\widehat{G}$ (with any vertex as the root). Denote by $\mathcal{R}$ the set of all setpairs $W = (W_t, W_h)$ such that the root r is in W_t and $W_h \neq \emptyset$; let $\hat{\delta}(W)$ denote the set of arcs (u, v) in $\widehat{G}$ with $u \in W_t,\ v \in W_h$.

$$
\begin{array}{llrcll}
\widehat{P}(\ell): & \min & \displaystyle\sum_{a \in E(\widehat{G})} c_a\, x_a & & & \\
& \text{subject to} & \displaystyle\sum_{a \in \hat{\delta}(W)} x_a & \geq & \ell - |W'| & \forall\, W \in \mathcal{R} \\
& & 0 \ \leq\ x_a & \leq & 1 & \forall\, a \in E(\widehat{G})
\end{array}
$$

Now $\widehat{P}$ has an integer optimal solution (see [11]). The Frank-Tardos algorithm finds a minimum cost subgraph $\widehat{H}$ that is ℓ-outconnected from r, whose cost $c(\widehat{H})$ equals the optimal value of $\widehat{P}$. The undirected version of $\widehat{H}$ is then a subgraph of G that is ℓ-outconnected from r, with cost at most $2z(\ell)$. $\square$

3-Critical Graphs

Let $\kappa(G)$ denote the *vertex connectivity* of G, that is, the minimum number of vertices whose removal results in a disconnected graph. A λ-*separator* of a connected graph is a set of λ vertices whose removal results in a disconnected graph.

A graph $G = (V, E)$ is called *r-critical* if the vertex connectivity decreases by $|S|$ on removing the vertices in any set S of at most r vertices, that is, if

$$\kappa(G \setminus S) = \kappa(G) - |S|, \quad \forall S \subseteq V, |S| \leq r$$

If G is *not* r-critical, then there exists a set S of r vertices such that no $\kappa(G)$-separator contains all the vertices in S. The algorithm which we present will work in polynomial time, if r is a constant, for any graph that is not r-critical. Due to the following result of Mader [32], we will focus our attention on the case $r = 3$.

Theorem 2.2.2. *A 3-critical graph of vertex connectivity k has less than $6k^2$ vertices.*

It is not know if this result is tight. It is known, however, that 3-critical graphs containing $\Omega(k^{\frac{3}{2}})$ vertices do exist [34].

REMARK. Mader's proof [32] is written in German. He discusses this result in two survey papers written in English [33, 34].

2.2.2 Augmentation Algorithm for k-Vertex Connectivity

The Algorithm

The algorithm starts begins with a minimum-cost spanning tree H_1. At iteration i, $1 \leq i \leq k-1$, the algorithm augments H_i to H_{i+1} by adding edges from $E(G) \setminus E(H_i)$ such that the vertex connectivity increases by at least one. In addition, the *augmenting cost*, $c(H_{i+1}) - c(H_i)$, is approximately minimum. A detailed description of an iteration follows. Let $\ell = \kappa(H_i)$. If $\ell = k$, then we stop and output H_i as the desired solution. Now, suppose $\ell < k$. For each edge in H_i, we change the edge cost to zero; the other edges keep the original costs. Hence we use the augmenting cost of the edge. By Mader's theorem, and the fact that n is at least $6k^2$, there exist a set S of at most three vertices such that no ℓ-separator of H_i contains each vertex of S; we call such a set *non-critical*. We find a non-critical set, say r_1, r_2 and r_3, by exhaustively checking for each vertex set S of cardinality three whether $\kappa(H_i \setminus S) > \ell - 3$. For each vertex in our non-critical set, we apply the Frank-Tardos algorithm with root r_j ($j = 1$, 2, or 3) and the modified edge costs, to find a subgraph H_i^j that is $(\ell+1)$-outconnected from r_j. We take H_{i+1} to be the union of (the edge sets of) H_i^1, H_i^2, H_i^3 and H_i.

An Example

To illustrate how the algorithm works, we present a small example. Suppose we wish to find the minimum 2-vertex connected subgraph in the graph shown in Figure 2-1. In the first phase, see Figure 2-2, we begin with $H_0 = \emptyset$. As the vertex connectivity

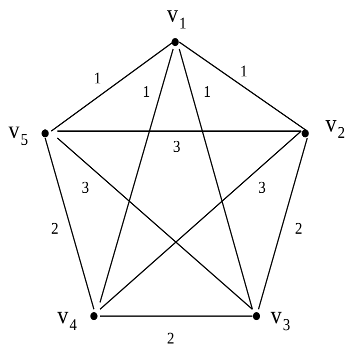

Figure 2-1: An Example

of H_0 is zero, the removal of any vertex does not decrease the connectivity. Thus any vertex induces a non-critical set S. So take v_1 to be our non-critical set. We then find a minimum cost 1-outconnected subgraph H_0^1 (in the directed version of the graph) rooted at v_1 using the augmenting costs. We then set $H_1 = H_0 \cup H_0^1$.

The second phase of the algorithm is shown in Figure 2-3. Since H_1 has vertex connectivity one, the removal of any two vertices reduces the connectivity by at

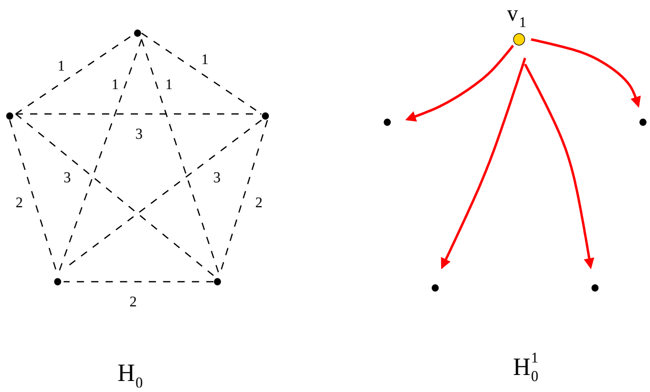

Figure 2-2: Phase I

most one. Thus, any pair of vertices induces a non-critical set S. So take v_1 and v_2 to be our non-critical set. Using the augmenting costs we find minimum cost 2-outconnected subgraphs H_1^1 and H_1^2 rooted at v_1 and v_2, respectively. We then set $H_2 = H_1 \cup H_1^1 \cup H_1^2$. Observe that, as required, H_2 is 2-vertex connected.

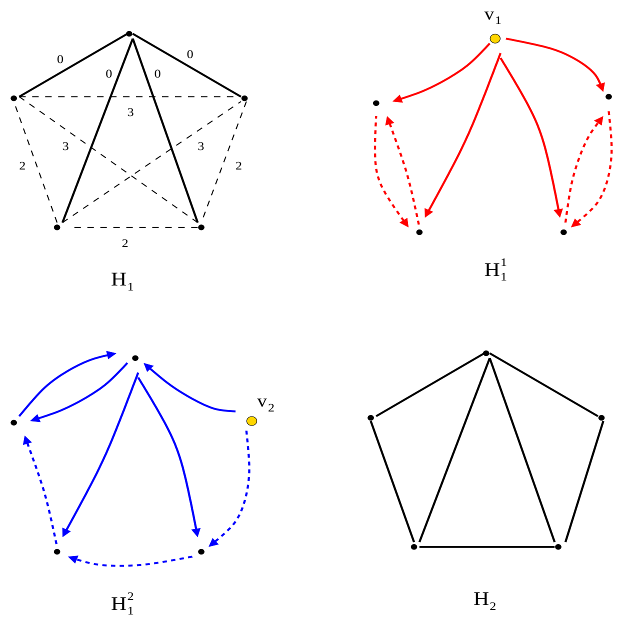

Figure 2-3: Phase II

We also remark that the algorithm works on this example despite the fact that $n < 6k^2$. In general, we will require that $n \geq 6k^2$ in order to guarantee the existance of non-critical sets in each phase of the algorithm.

An Analysis

First we show that the algorithm does indeed work. That is, we show that the vertex connectivity increases by one in each phase of the algorithm.

Lemma 2.2.3. *We have $\kappa(H_{i+1}) \geq \kappa(H_i) + 1$, for each $i \geq 1$.*

Proof. Let $\ell = \kappa(H_i)$. Note that $\ell < k$. Suppose that $\kappa(H_{i+1}) = \ell$. Then H_{i+1} has an ℓ-separator $C \subseteq V$. By Mader's theorem, H_i is not 3-critical since $n \geq 6k^2 > 6\ell^2$. Hence, there exist a set S of at most three vertices in H_i such that for each ℓ-separator of H_i, at least one of vertices of S is absent from the ℓ-separator. The algorithm finds such a non-critical set, say r_1, r_2 and r_3. Assume, without loss of generality, that r_1 is absent from C. The graph H_i^1, which is a subgraph of H_{i+1}, is $(\ell+1)$-outconnected from r_1. Therefore H_{i+1} has $(\ell+1)$ openly disjoint paths between r_1 and any other vertex v. Observe that at least one of these paths survives in $H_{i+1} \setminus C$. Thus $H_{i+1} \setminus C$ is connected, a contradiction. □

We now measure the quality of the solution produced by the algorithm.

Lemma 2.2.4. *For each $i \geq 1$, we have $c(H_{i+1}) - c(H_i) \leq \dfrac{6z(k)}{k-\ell}$, where $\ell = \kappa(H_i)$.*

Proof. Note that $\ell < k$. We will prove that for each of the three supergraphs H_i^j ($j = 1$, 2, or 3) of H_i, the augmenting cost $c(H_i^j) - c(H_i)$ is at most $2z(k)/(k-\ell)$. The lemma then follows immediately. Recall that, during the construction of H_i^j, we use the augmenting edge costs. Let $\boldsymbol{x} : E \to \mathbb{R}_+$ be an optimal solution to the linear program $P(k)$; note that the cost of $\boldsymbol{x}$ is $z(k)$. Let $\boldsymbol{x}' : E \to \mathbb{R}_+$ be given by

$$x'_e = \begin{cases} 1 & \text{if } e \text{ is in } H_i \\ \frac{x_e}{k-\ell} & \text{otherwise} \end{cases}$$

Clearly, $\boldsymbol{x}'$ has modified cost at most $z(k)/(k-\ell)$. We claim that $\boldsymbol{x}'$ is a feasible solution to the linear program $P(\ell+1)$. Then, by Theorem 2.2.1, the Frank-Tardos algorithm finds an $(\ell+1)$-outconnected supergraph of H_i with augmenting cost at most $2z(k)/(k-\ell)$. To see the claim, consider any setpair $W \in \mathcal{S}$ and its constraint in $P(\ell+1)$,

$$\sum_{e\in\delta(W)} x'_e \geq (\ell+1) - |W'|$$

Suppose that H_i has p edges in $\delta(W)$. Then

$$\begin{aligned}\sum_{e\in\delta(W)} x'_e &= \sum_{e\in\delta(W)\cap H_i} x'_e + \sum_{e\in\delta(W)\setminus H_i} x'_e \\ &= p + \frac{1}{k-\ell}\sum_{e\in\delta(W)\setminus H_i} x_e\end{aligned}$$

Thus if $p \geq \ell+1-|W'|$ we are done. Moreover, since H_i is ℓ-vertex connected, we have that $p \geq \ell - |W'|$. Hence, we may assume that $p = \ell - |W'|$. Given that $\boldsymbol{x}$ is an optimal solution to $P(k)$, it follows that $\sum_{e\in\delta(W)} x_e \geq k - |W'|$. So, $\sum_{e\in\delta(W)\setminus H_i} x_e \geq k - |W'| - p = k - \ell$. Thus $\sum_{e\in\delta(W)} x'_e \geq p+1 = (\ell+1) - |W'|$. □

Theorem 2.2.5. *The algorithm produces a k-vertex connected subgraph with cost at most $6\,H(k)\,z(k) \leq 6\,H(k)\,\text{OPT}$.*

Proof. The vertex connectivity of H_i increases by at least one in every iteration, starting from one, so the algorithm terminates with a k-vertex connected subgraph in at most $k-1$ iterations. Note that a minimum spanning tree H_1 in G induces a directed minimum 1-outconnected subgraph in $\widehat{G}$ for any root vertex r. Thus the cost of H_1 is at most $2z(1)$. So the cost of the k-vertex connected subgraph obtained is at most

$$\begin{aligned}c(H_1) + \sum_{i=1}^{k-1}(c(H_{i+1}) - c(H_i)) &\leq \frac{2z(k)}{k} + \sum_{\ell=1}^{k-1}\frac{6z(k)}{k-\ell} \\ &\leq 6\,(1+\frac{1}{2}+\frac{1}{3}+\cdots+\frac{1}{k})\,z(k) \\ &\leq 6\,(1+\frac{1}{2}+\frac{1}{3}+\cdots+\frac{1}{k})\,\text{OPT}\end{aligned}$$

□

Observe that the algorithm does run in polynomial time. Each iteration runs the Frank-Tardos algorithm at most three times, and tests $\kappa(H_i \setminus S)$ for at most n^3 sets S of three vertices. Gabow's algorithm [16] tests the vertex connectivity κ in time $O((n+\kappa^{2.5})\,\kappa n)$, and there is a version of the Frank-Tardos algorithm, due to Gabow [15], that runs in time $O(k^2 m n^2)$.

Lower Bounds on the Integrality Gap

So the integrality gap of the setpair LP for the k-vertex connectivity problem is at most $O(\log k)$. It is easy to see that the integrality gap is at least 2. Take the 1-vertex connectivity problem on a cycle; the optimal LP solution has value $\frac{n}{2}$, whereas the optimal solution contains $n-1$ edges. An important open problem is to determine whether the integrality gap increases with k.

2.3 Iterative Rounding

In this section we show how the iterative rounding method, due to Jain [25], may be applied to the setpair model. As a result, we develop approximation algorithms for network problems with various vertex connectivity requirements. In particular, we give approximation algorithms for the (directed and undirected) k-vertex connectivity problem and the element connectivity problem.

The *iterative rounding* method works as follows. Formulate the problem as an integer program, and then solve the LP relaxation to find a basic (extreme point) optimum solution $\boldsymbol{x}$. Pick an edge e^* of highest weight (i.e., $x_{e^*} \geq x_e$, $\forall e \in E$) and add it to the solution subgraph H. Then update the LP and the integer program (since the variable x_{e^*} is now implicitly fixed at value 1). Specifically, decrease by 1 the right hand side of every constraint in which the variable x_{e^*} occurs, and then remove this variable from the LP. The resulting LP is the same as the original LP for the "reduced" problem where the edge e^* is pre-selected for H. Under appropriate conditions on the requirement function f, the problem turns out to be "self reducible", that is, the essential properties of the original problem are preserved in the reduced problem. Then iteratively solve the reduced problem.

Jain [25] applied this method to the cut covering model, and proved that it achieves an approximation guarantee of 2 provided that the requirement function f is weakly supermodular. (Such requirement functions capture several interesting problems, e.g., the Steiner network problem.) The proof follows from a key property of the LP: every non-zero extreme point has an edge of weight at least $\frac{1}{2}$. This result is based on an extension of a classic result that, under appropriate conditions on the requirement function f, every extreme point of the LP is characterised by a laminar family of "tight sets".

In Section 2.3.1, we introduce a special class of requirement functions f, namely, *skew bisupermodular* functions. These functions are a common generalisation of weakly supermodular (symmetric) functions ([11], [20]) and crossing bisupermodular functions [12]. Skew bisupermodular requirement functions are useful because the extreme points of the LP relaxation of the setpair model are characterised by "non-overlapping families" of setpairs (see Section 2.3.2). Note the correspondence with the cut covering model, where the extreme points of the LP relaxation may be characterised by a laminar family of sets. Based on this, we obtain lower bounds on the maximum weight of an edge for the basic solutions of the LP relaxation. Also, this LP relaxation has the self-reducibility property needed for the iterative rounding method. Hence, these structural results immediately give approximation algorithms.

2.3.1 Setpairs and Skew Bisupermodularity

Crossing Setpairs.

Two setpairs W and Y are said to be *comparable* if

$$\text{(i) } W_t \supseteq Y_t \text{ and } W_h \subseteq Y_h, \quad \text{or} \quad \text{(ii) } W_t \subseteq Y_t \text{ and } W_h \supseteq Y_h.$$

The former case is denoted by $W \preceq Y$ and the latter case by $W \succeq Y$. An example of the former case is given in Figure 2-4. For clarity, the example has W_t and Y_h being disjoint; in general, W_t and Y_h may intersect.

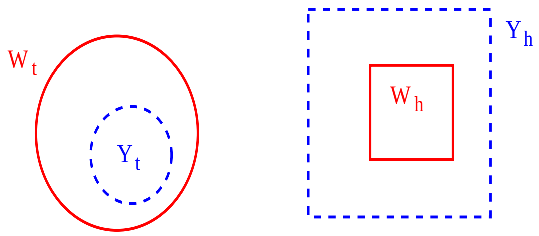

Figure 2-4: Comparable setpairs.

Two setpairs W and Y are said to be *non-crossing* if they are

(i) comparable, or (ii) tail-disjoint, or (iii) head-disjoint.

Otherwise, W and Y are said to *cross*. Note that W and Y cross if and only if their tails intersect, their heads intersect and they are not comparable. An example of crossing setpairs is given in Figure 2-5. In general, W_t and Y_h may intersect and/or W_h and Y_t may intersect. Again, for clarity, this is not the case in the example shown.

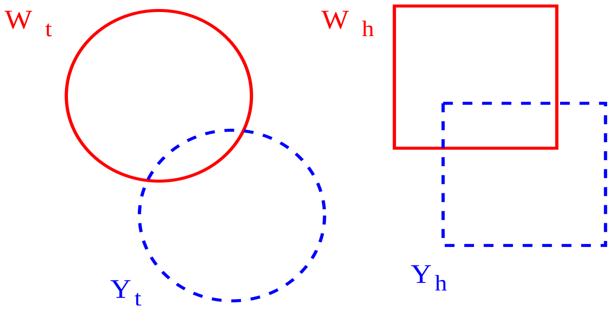

Figure 2-5: Crossing setpairs.

For a setpair $W = (W_t, W_h)$, let $\overline{W} = (W_h, W_t)$ be the *reverse setpair*. Note that $\delta(W) = \delta(\overline{W})$ for an undirected graph. Two setpairs W and Y are *overlapping* if

(i) W and Y cross, or (ii) $\overline{W}$ and Y cross (equivalently, W and $\overline{Y}$ cross).

Otherwise, the setpairs are called *non-overlapping*. A family of setpairs $\mathcal{L} \subseteq \mathcal{S}$ is called non-crossing (non-overlapping) if no two setpairs in $\mathcal{L}$ cross (overlap).

Bisubmodular Functions.

Given two setpairs $W = (W_t, W_h)$ and $Y = (Y_t, Y_h)$, let $W \otimes Y$ denote the setpair $(W_t \cup Y_t,\ W_h \cap Y_h)$, and let $W \oplus Y$ denote the setpair $(W_t \cap Y_t,\ W_h \cup Y_h)$. A real-valued function f on $\mathcal{S}$ is called *bisubmodular* if for any two setpairs W and Y we have

$$f(W) + f(Y) \geq f(W \otimes Y) + f(W \oplus Y)$$

A non-negative, integer-valued function f on $\mathcal{S}$ is called *crossing bisupermodular* if for any two crossing setpairs W and Y with $f(W) > 0$ and $f(Y) > 0$, we have

$$f(W) + f(Y) \leq f(W \otimes Y) + f(W \oplus Y)$$

Let χ_W denote the edge incidence vector of $\delta(W)$. For any two setpairs W and Y, note that if an edge is present in $\delta(W \oplus Y)$ or $\delta(W \otimes Y)$, then it is present in $\delta(W)$ or $\delta(Y)$, and if an edge is present in both $\delta(W \oplus Y)$ and $\delta(W \otimes Y)$, then it is present in both $\delta(W)$ and $\delta(Y)$. Hence, we have $\chi_{W \otimes Y} + \chi_{W \oplus Y} \leq \chi_W + \chi_Y$. Consequently, for any non-negative vector $\boldsymbol{x} : E \to \mathbb{R}_+$ on the edges, the corresponding function on setpairs, $x(\delta(W)) = \sum_{e \in \delta(W)} x_e$, is bisubmodular.

Skew Bisupermodular Functions.

A non-negative, integer-valued function f on $\mathcal{S}$ is called *skew bisupermodular* if for any two overlapping setpairs W and Y with $f(W) > 0$ and $f(Y) > 0$, one of the following holds

$$\begin{aligned} \text{(i)} \quad f(W) + f(Y) &\leq f(W \otimes Y) + f(W \oplus Y) \\ \text{(ii)} \quad f(W) + f(Y) &\leq f(\overline{W} \otimes Y) + f(\overline{W} \oplus Y) \end{aligned}$$

This is motivated by problems on undirected graphs. For these problems, it turns out that the relevant functions f are symmetric, i.e., $f(W) = f(\overline{W})$, $\forall W \in \mathcal{S}$.

Note that every crossing bisupermodular function is skew bisupermodular. Also, every weakly supermodular symmetric function is skew bisupermodular (see the discussion at the start of Section 2.3.4). Thus the skew bisupermodular property is a generalisation of these earlier notions.

2.3.2 Characterising a Basic Solution via a Non-Overlapping Family

The main result of this section, Theorem 2.3.3, characterises a basic solution of the setpair LP via a non-overlapping family of setpairs, assuming that the requirement function f is skew bisupermodular. This result is an extension of classic results to our setting. Our proof is similar to earlier proofs in [25] and [6]; however, when we add a new setpair W to our non-overlapping family, we have to ensure that neither W nor $\overline{W}$ crosses a setpair in the family.

Given a feasible solution $\boldsymbol{x}$ to the LP, a setpair W is called *tight* if $x(\delta(W)) = f(W)$. Now take a skew bisupermodular requirement function f, and let $\boldsymbol{x}$ be a feasible solution such that $x_e > 0$ for all edges $e \in E$. Then we obtain the following lemma.

Lemma 2.3.1. *Let W and Y be overlapping, tight setpairs (with $f(W) > 0, f(Y) > 0$). Then one of the following holds:*
(i) The setpairs $W \otimes Y$ and $W \oplus Y$ are tight, and $\chi_W + \chi_Y = \chi_{W\otimes Y} + \chi_{W\oplus Y}$.
(ii) The setpairs $\overline{W} \otimes Y$ and $\overline{W} \oplus Y$ are tight, and $\chi_W + \chi_Y = \chi_{\overline{W}\otimes Y} + \chi_{\overline{W}\oplus Y}$.

Proof. Assume that $f(W)+f(Y) \leq f(W\otimes Y)+f(W\oplus Y)$. The other case is similar (note that $\overline{W}$ is tight, overlaps Y, and $f(\overline{W}) > 0$). Then

$$\begin{aligned} f(W\otimes Y) + f(W\oplus Y) &\leq x(\delta(W\otimes Y)) + x(\delta(W\oplus Y)) \\ &\leq x(\delta(W)) + x(\delta(Y)) \\ &= f(W) + f(Y) \\ &\leq f(W\otimes Y) + f(W\oplus Y) \end{aligned}$$

Hence, all the inequalities hold as equations, therefore, $W\otimes Y$ and $W\oplus Y$ are both tight. This proves the first part of (i). The second part of (i) follows from the following facts: $\chi_W+\chi_Y \geq \chi_{W\otimes Y}+\chi_{W\oplus Y}$; $x(\delta(W\otimes Y))+x(\delta(W\oplus Y)) = x(\delta(W))+x(\delta(Y))$; $x_e > 0$ for all edges e. □

The next result is in [6] and is used to prove the main result of this section.

Lemma 2.3.2. *Let W, Y and Z be setpairs. If Z crosses $W\otimes Y$ (or $W\oplus Y$) then either Z crosses W or Z crosses Y.* □

Recall we may also assume that $1 > x_e > 0$, otherwise we either fix the edge or remove it and solve the induced problem.

Theorem 2.3.3. *There exists a non-overlapping family $\mathcal{L}$ of tight, linearly independent setpairs that span the support of $\boldsymbol{x}$.*

Proof. We will show that there exists a family $\mathcal{L}$ of tight, non-overlapping setpairs such that:
(a) Every setpair $W \in \mathcal{L}$ has $f(W) \geq 1$.
(b) $|\mathcal{L}| = |E|$.
(c) The vectors χ_W, $W \in \mathcal{L}$, are linearly independent.

(d) $\boldsymbol{x}$ is the unique solution to $\{x(\delta(W)) = f(W), \forall W \in \mathcal{L}\}$.

Since $\boldsymbol{x}$ is a basic solution and none of the constraints $0 \leq x_e \leq 1$ $(e \in E)$ holds with equality, there exist $|E|$ tight setpair constraints $x(\delta(W)) = f(W) > 0$ such that the edge incidence vectors χ_W corresponding to these constraints are linearly independent. Let $\mathcal{L}$ be a maximal, non-overlapping family of tight setpairs and let $span(\mathcal{L})$ denote the vector space spanned by the vectors χ_W, where $W \in \mathcal{L}$. The theorem holds because $span(\mathcal{L})$ equals $span(\mathcal{F})$, where $\mathcal{F}$ is the family of all tight setpairs. We prove this by contradiction. Suppose that there is a tight setpair Y such that $\chi_Y \notin span(\mathcal{L})$. Take such a Y that overlaps the minimum number of setpairs in $\mathcal{L}$ (this is a key step). Choose any setpair $W \in \mathcal{L}$ that overlaps Y; there exists such a W by the maximality of $\mathcal{L}$.

Either part (i) or part (ii) in the statement of Lemma 2.3.1 holds for W and Y. Assume that part (ii) holds. The other case is similar. Thus, the setpairs $\overline{W} \otimes Y$ and $\overline{W} \oplus Y$ are tight. In addition, $\chi_W + \chi_Y = \chi_{\overline{W} \otimes Y} + \chi_{\overline{W} \oplus Y}$. It follows that either $\chi_{\overline{W} \otimes Y} \notin span(\mathcal{L})$ or $\chi_{\overline{W} \oplus Y} \notin span(\mathcal{L})$. Consider the former case. Again, the other case is similar. Let $Q = \overline{W} \otimes Y$ and take any setpair $Z \in \mathcal{L}$. Suppose that Z overlaps Q. Then either Z crosses Q or Z crosses $\overline{Q} = \overline{\overline{W} \otimes Y} = W \oplus \overline{Y}$. Suppose that Z crosses Q. We apply Lemma 2.3.2 to Z and Q, noting that Z does not cross $\overline{W}$, to see that Z crosses Y. Similarly, if Z crosses $\overline{Q}$, then by Lemma 2.3.2, Z crosses $\overline{Y}$. Hence, Z overlaps Y. So every setpair in $\mathcal{L}$ that overlaps Q also overlaps Y.

It follows that Q overlaps fewer setpairs in $\mathcal{L}$ than Y. To see this, note that (1) W overlaps Y, by our choice of W, and (2) W does not overlap Q, because $Q \preceq \overline{W}$ (so W does not cross $\overline{Q}$) and W and Q are head disjoint (so W does not cross Q). Thus we get a contradiction to the choice of Y, since Q is a tight setpair that overlaps fewer setpairs in $\mathcal{L}$ than Y, and $\chi_Q \notin span(\mathcal{L})$. □

Remarks. The results in this section apply to undirected graphs but not to directed graphs, since we use the fact that a setpair W with $f(W) > 0$ is tight iff the reverse setpair $\overline{W}$ has $f(\overline{W}) > 0$ and is tight. In the theorem, if we replace a setpair $W \in \mathcal{L}$ by its reverse $\overline{W}$, then this preserves the properties (a)-(d).

2.3.3 Iterative Rounding and k-Vertex Connectivity

The main result of this section is as follows.

Theorem 2.3.4. *Let $k \leq (1-\epsilon)n$, for some $\epsilon < 1$. Then iterative rounding gives an $O(\sqrt{n/\epsilon})$-approximation algorithm for the k-vertex connectivity problem.* $\square$

Now recall that the k-vertex connectivity problem can be formulated by the setpair model using the following requirement function.

$$f(W) = \begin{cases} \max\{0,\ k - |W'|\} & \text{if } W_t \neq \emptyset \text{ and } W_h \neq \emptyset \\ 0 & \text{otherwise} \end{cases}$$

The k-vertex connectivity requirement function f is crossing bisupermodular [12], hence, f is also skew bisupermodular. This can be seen as follows. For each setpair W, let $p(W)$ denote $|W'|$. For any two setpairs $W = (W_t, W_h)$ and $Y = (Y_t, Y_h)$, we have $p(W) + p(Y) = p(W \otimes Y) + p(W \oplus Y)$, because

$$|(W_t \cup Y_t) \cup (W_h \cap Y_h)| + |(W_t \cap Y_t) \cup (W_h \cup Y_h)| = |W_t| + |W_h| + |Y_t| + |Y_h|.$$

Hence, the function f is crossing bisupermodular (if either $W \otimes Y$ or $W \oplus Y$ has an empty head or tail then note that W and Y do not cross). The results in [12] imply that a basic optimal solution to setpair linear programming relaxation for k-vertex connectivity can be found in polynomial time. Moreover, the requirement function f has the following "self-reducibility" property: for any edge e, the "reduced" requirement function f' is also crossing bisupermodular (and skew bisupermodular), where f' is given by $f'(W) = f(W) - 1$ if the setpair W has $e \in \delta(W)$ and $f'(W) = f(W)$ otherwise (since the zero-one incidence function of e on $\mathcal{S}$ is bisubmodular). Theorem 2.3.4 then follows directly from the next result.

Theorem 2.3.5. *Let $k \leq (1-\epsilon)n$, for some $\epsilon < 1$. Then any nonzero basic solution of the setpair LP for k-vertex connectivity has an edge of weight $\Omega(\sqrt{\frac{\epsilon}{n}})$.*

We give the proof for undirected graphs, using results on skew bisupermodular functions. After that, we sketch the proof for directed graphs via crossing bisupermodular functions. Consider the setpair model LP with a crossing bisupermodular requirement function f. Then we have the following result from [6] (which applies also to directed graphs).

Theorem 2.3.6. *Let $\mathcal{L}$ be a non-crossing family of setpairs characterising a nonzero basic solution $\boldsymbol{x}$. Then there exists an edge e with $x_e \geq \frac{1}{\Omega(\sqrt{|\mathcal{L}|})}$.* $\square$

Focus on a nonzero basic solution $\boldsymbol{x}$. By Theorem 2.3.3, $\boldsymbol{x}$ corresponds to a noncrossing family of setpairs $\mathcal{L}$ such that each setpair has a nonempty head and a nonempty tail. We may assume that each setpair $W \in \mathcal{L}$ has $|W_h| \geq |W_t|$, else we may replace W by $\overline{W}$ and this preserves all the properties of $\mathcal{L}$ in Theorem 2.3.3. Let $\mathcal{P}$ denote the poset (and the Hasse diagram) representing $\mathcal{L}$, and recall that $\mathcal{P}$ has a node W for each setpair W in $\mathcal{L}$ and it has an arc (W, Y), where $Y \in \mathcal{L}$, if $W \preceq Y$ and there is no other $Z \in \mathcal{L}$ such that $W \preceq Z \preceq Y$ (we omit arcs implied by transitivity from $\mathcal{P}$); we call W a *child* of Y, and call Y a *parent* of W. (Note that the terms "node" and "arc" refer to the poset $\mathcal{P}$, and the terms "vertex" and "edge" refer to G.) A *diamond* of a poset is a set of four nodes W, X, Y, Z such that X and Y are incomparable, W is a common descendant of X and Y, and Z is a common ancestor of X and Y. A poset is called *diamond-free* if it contains no diamond. It is easy to show that poset $\mathcal{P}$ representing $\mathcal{L}$ is diamond-free [6].

Lemma 2.3.7. *Suppose that $k \leq (1-\epsilon)n$. Then there are at most $\frac{2}{\epsilon}$ pairwise incomparable setpairs in $\mathcal{L}$ such that all their tails contain a common vertex.*

Proof. Let the setpairs $Y^1, Y^2, \ldots, Y^q \in \mathcal{L}$ be pairwise incomparable, and let v be a vertex that occurs in the tail of each of these setpairs. Since the Y^i are incomparable and their tails intersect, their heads are disjoint. Each head Y_h^i has cardinality at least $\frac{1}{2}(n-k) \geq \frac{1}{2}\epsilon n$; this follows from the properties of $\mathcal{L}$, and the fact that each requirement $f(Y^i)$ is at least one. Hence, there are at most $\frac{2}{\epsilon}$ such setpairs. $\square$

We partition the setpairs in $\mathcal{L}$ into several sets.

(i) W is a type I setpair if it has no parents in $\mathcal{P}$.

(ii) W is a type II setpair if it has at least two parents in $\mathcal{P}$.
(iii) W is a type III setpair if it has exactly one parent Y in $\mathcal{P}$, and $W_t = Y_t$.
(iv) W is a type IV setpair if it has exactly one parent Y in $\mathcal{P}$, and $W_t \neq Y_t$.

Lemma 2.3.8. *The family of setpairs $\mathcal{L}$ contains at most $\frac{2n}{\epsilon}$ setpairs of type IV. In addition, each node (setpair) $W \in \mathcal{P}$ has at most $\frac{2n}{\epsilon}$ children.*

Proof. To each type IV setpair W with parent Y we assign a vertex v in $W_t \setminus Y_t$. We claim that a vertex can be assigned to at most $\frac{2}{\epsilon}$ setpairs of type IV. This will imply the claim. Take a vertex v that is assigned to the type IV setpairs $W^1, W^2, \ldots, W^q$. Let their parents be $Y^1, Y^2, \ldots, Y^q$, respectively. Now $v \in W_t^1 \cap W_t^2 \cap \cdots \cap W_t^q$, but $v \notin Y_t^1 \cup Y_t^2 \cup \cdots \cup Y_t^q$. The setpairs $W^1, W^2, \ldots, W^q$ associated with v must be pairwise incomparable (since v is in the tail of every W^i, but for every proper ancestor of a W^i note that v is not in the tail). Then by Lemma 2.3.7 applied to $W^1, W^2, \ldots, W^q$, there are at most $\frac{2}{\epsilon}$ such setpairs whose tails contain v.

For the second part of the lemma, consider all the children $Y^1, Y^2, \ldots, Y^q$ of a node W. Then $Y^1, Y^2, \ldots, Y^q$ are pairwise incomparable (otherwise, one of the arcs (Y^i, W) in $\mathcal{P}$ is redundant), and moreover, each of the tails Y_t^i, $i = 1, \ldots, q$, contains W_t which is nonempty. Then, applying Lemma 2.3.7 to $Y^1, Y^2, \ldots, Y^q$, we have $q \leq \frac{2}{\epsilon}$. ☐

Proof of Theorem 2.3.5. Let m denote $|\mathcal{L}|$. If $m < 4n/\epsilon$, then by Theorem 2.3.6 there is an edge e with $x_e \geq \sqrt{\frac{\epsilon}{\Theta(n)}}$. Hence, in this case, Theorem 2.3.5 holds. The rest of the proof focuses on the other case, $m \geq 4n/\epsilon$. We show that there exists an edge e with either $x_e \geq \frac{1}{\Theta(1)}$ or $x_e \geq \Omega(\epsilon) \geq \Omega(\sqrt{\frac{\epsilon}{n}})$ (note that $k \leq n-1$, hence $\epsilon \geq \frac{1}{n}$, and so $\epsilon \geq \sqrt{\frac{\epsilon}{n}}$). We have $|\mathcal{L}| = m \geq \frac{4n}{\epsilon}$, hence, by by Lemma 2.3.8, $\mathcal{L}$ has at least $\frac{m}{6}$ setpairs of one of the types I, II, or III.

(Type I) Suppose that there are at least $\frac{m}{6}$ type I setpairs. The type I setpairs are pairwise incomparable. Hence the edge sets $\delta(W)$, where W is a type I setpair, are *quasi-disjoint*, i.e., for each edge (u, v), there are at most two setpairs W, Y (among the type I setpairs) such that $(u, v) \in \delta(W)$ (say $u \in W_t$ and $v \in W_h$) and $(u, v) \in \delta(Y)$

(say $v \in Y_t$ and $u \in Y_h$). Hence, there is a type I setpair $W \in \mathcal{L}$ such that $\delta(W)$ has size at most 12, and so there is an edge $e \in \delta(W)$ with $x_e \geq \frac{1}{12}$.

(Type II) Suppose that there are at least $\frac{m}{6}$ type II setpairs. Let $Y^1, \ldots, Y^q$ be the minimal nodes in $\mathcal{P}$ (those with no children); clearly, these nodes are pairwise incomparable. Each Y^i and its ancestors forms a (directed) tree (since $\mathcal{P}$ is diamond-free). Each node of $\mathcal{P}$ is in at most $\frac{2}{\epsilon}$ of these trees, by Lemma 2.3.7. Focus on one of the trees and note that the number of leaf nodes (maximal nodes of $\mathcal{P}$) of the tree is greater than or equal to the number of type II nodes in the tree. Summing over all the trees, we see that the total number of leaf nodes (maximal nodes of $\mathcal{P}$) is at least $\frac{\epsilon}{2}$ times the number of type II nodes in $\mathcal{P}$, hence, $\mathcal{P}$ has at least $\frac{m\epsilon}{12}$ maximal nodes; clearly, these nodes are pairwise incomparable. Then the edge sets $\delta(W)$, where W is a maximal node of $\mathcal{P}$, are quasi-disjoint, so there is an edge e with $x_e \geq \frac{\epsilon}{24}$.

(Type III) Finally, suppose that there are at least $\frac{m}{6}$ type III setpairs. Consider all of the type III setpairs $W^1, W^2, \ldots W^q$ and their unique parents $Y^1, Y^2, \ldots Y^q$. Observe that $f(W^i)+1 \leq f(Y^i)$, since $|W_t \cup W_h| < |Y_t \cup Y_h|$. Moreover, if W^i and W^j (where $1 \leq i < j \leq q$) have distinct parents Y^i and Y^j, then the edge sets $\delta(Y^i) \setminus \delta(W^i)$ and $\delta(Y^j) \setminus \delta(W^j)$ are quasi-disjoint. To see this, consider an edge $(u, v) \in \delta(Y^i) \setminus \delta(W^i)$ with $u \in Y^i_t = W^i_t$ and $v \in Y^i_h \setminus W^i_h$. Now, every proper ancestor Z of W^i has $v \in Z_h$, so if W^j is an ancestor of W^i, then we cannot have $(u, v) \in \delta(Y^j) \setminus \delta(W^j)$, $u \in Y^j_t = W^j_t$, and $v \in Y^j_h \setminus W^j_h$. Consequently, by Lemma 2.3.8, any edge occurs in at most $\frac{4}{\epsilon}$ of the edge sets $\delta(Y^i) \setminus \delta(W^i)$, and moreover, $x(\delta(Y^i) \setminus \delta(W^i)) \geq 1$, for each $i = 1, \ldots, q$. Hence, there is an edge e with $x_e \geq \frac{\epsilon}{24}$.

This concludes the proof of Theorem 2.3.5 for undirected graphs: any nonzero basic solution of the setpair LP for the k-connectivity problem has an edge of weight $\Omega(\sqrt{\epsilon/n})$.

The proof extends to directed graphs via Theorem 2.3.6. We partition the non-crossing family of setpairs $\mathcal{L}$ into two subfamilies $\mathcal{L}_1$ and $\mathcal{L}_2$, depending on whether or not a setpair $W \in \mathcal{L}$ has $|W_h| \geq |W_t|$. Suppose that $|\mathcal{L}_1| \geq |\mathcal{L}_2|$; otherwise, we use a symmetric argument. Then we apply the arguments for undirected graphs to $\mathcal{L}_1$ (partitioning it into types I, II, III, IV, etc.). We obtain similar lower bounds on

$\max\{x_e\}$, the main difference being that we lose a factor of two in the lower bound (since we have $|\mathcal{L}_1| \geq \frac{m}{2}$ rather than $|\mathcal{L}| = m$). Hence, Theorem 2.3.5 holds for directed graphs. $\square$

Theorem 2.3.4 follows from Theorem 2.3.5: for $k \leq (1-\epsilon)n$, iterative rounding gives an $O(\sqrt{n/\epsilon})$ approximation algorithm for the (directed or undirected) k-vertex connectivity problem.

REMARK. There is an example of the undirected problem that has an extreme point x for which each edge e has $x_e \leq \frac{1}{\Omega(\sqrt{k})}$. The example is obtained from the example in Figure 1 of the Erratum of [37] with appropriate modifications. Hence, in Theorem 2.3.5, the lower bound on the maximum weight of an edge cannot be improved beyond $\frac{1}{\Omega(\sqrt{k})}$. Note, however, this does not imply a similar lower bound on the integrality gap of the linear program. As we have seen, the intergrality gap is at most $O(\log k)$.

2.3.4 Iterative Rounding and Element Connectivity

In the *element connectivity problem* [26], there is a set of *terminal* vertices $T \subseteq V$ and between each pair of distinct terminals i and j there is a connectivity requirement r_{ij} (each r_{ij} is a non-negative integer). Terminals are reliable and do not fail. Edges and non-terminal vertices are unreliable and are subject to failure. The edges and non-terminals are called *elements*. The problem is to find a minimum-cost subgraph H that contains r_{ij} element disjoint paths between each pair of terminals i and j. Alternatively, for every pair of terminals i and j, H should contain a path between i and j despite the failure of up to $r_{ij} - 1$ elements. The problem is NP-hard. Recently, Fleischer, Jain, and Williamson [14] gave a 2-approximation algorithm for this problem, improving on the previous best approximation guarantee of $2\log(\max\{r_{ij}\})$ due to Jain et al [26]. The work in this section is independent of [14] and gives a different proof of the same factor 2 approximation guarantee. The element connectivity problem is a generalisation of the Steiner network problem, see Goemans et al [20] and Jain [25]. Even for rather special cases of this problem, the best approximation guarantee known is 2, and moreover, the LP relaxation has an integrality ratio of at least $2 - \frac{2}{n}$.

Now, for any $W \in \mathcal{S}$, let $g(W) = \max_{i \in W_t, j \in W_h} r_{ij}$. Define a requirement function f on the setpairs $W \in \mathcal{S}$ as follows:

$$f(W) = \begin{cases} \max\{0, g(W) - |W'|\} & \text{if } T \subseteq W_t \cup W_h \\ 0 & \text{otherwise} \end{cases}$$

Note that if a setpair W has a positive requirement, then all the terminals are in $W_t \cup W_h$. It is easily seen that the setpair model with the requirement function f models the element connectivity problem (for more details see [26]). For the element connectivity problem, we obtain the following lemma concerning the setpair requirement function.

Lemma 2.3.9. *The element connectivity requirement function f is skew bisupermodular.*

Proof. We need to show that the function $f = g - p$ is skew bisupermodular. We have already seen that $p(W) + p(Y) = p(W \otimes Y) + p(W \oplus Y)$. Thus, it suffices to show that g is skew bisupermodular. Since $f(W)$ and $f(Y)$ are positive it follows that $T \subseteq (W_1 \cap Y_1) \cup (W_1 \cap Y_2) \cup (W_2 \cap Y_1) \cup (W_2 \cap Y_2)$. Let β_1 be the value of the maximum requirement between a node in $(W_1 \cap Y_1)$ and a node in $(W_2 \cap Y_2)$. Similarly let α_1 be the value of the maximum requirement between a node in $(W_1 \cap Y_1)$ and a node in $(W_2 \cap Y_1)$. We define $\beta_2, \alpha_2, \alpha_3$ and α_4 in a similar fashion, as shown in Figure 2-6a). Here the edges are also labelled by their values. It follows that

$$\begin{aligned} g(W) + g(Y) &= \max(\beta_1, \beta_2, \alpha_1, \alpha_3) + \max(\beta_1, \beta_2, \alpha_2, \alpha_4) \\ g(W \otimes Y) + g(W \oplus Y) &= \max(\beta_1, \alpha_3, \alpha_4) + \max(\beta_1, \alpha_1, \alpha_2) \\ g(\bar{W} \otimes Y) + g(\bar{W} \oplus Y) &= \max(\beta_2, \alpha_2, \alpha_3) + \max(\beta_2, \alpha_1, \alpha_4) \end{aligned}$$

Observe that each of the terms $g(W)$, $g(Y)$, $g(W \otimes Y)$ etc. may thus be represented by a single edge. Note that the edge representing $g(W \otimes Y)$ is one of the three edges incident to $(W_2 \cap Y_2)$, whilst the edge representing $g(W \otimes Y)$ is one of the three edges incident to $(W_1 \cap Y_1)$. These subsets are shown as triangles in Figure 2-6b). Similarly

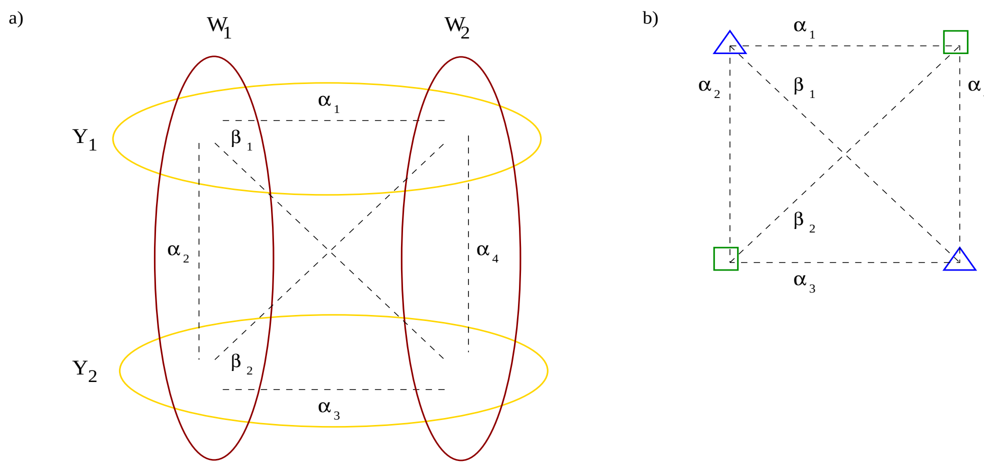

Figure 2-6: An illustration for skew bisupermodularity.

the edge representing $g(\bar{W} \otimes Y)$ is one of the three edges incident to $(W_1 \cap Y_2)$, whilst the edge representing $g(\bar{W} \oplus Y)$ is one of the three edges incident to $(W_2 \cap Y_1)$. These subsets are shown as squares in Figure 2-6b).

Thus $g(W) + g(Y)$ can be represented by a pair of (possibly identical) edges. We need to show that this pairing is also a feasible pairing for either $g(W \otimes Y) + g(W \oplus Y)$ or $g(\bar{W} \otimes Y) + g(\bar{W} \oplus Y)$. It suffices to show that either the two triangles are incident to one edge each in the optimal edge pairing for $g(W) + g(Y)$ or the two squares are. In the former case $g(W) + g(Y) \leq g(W \otimes Y) + g(W \oplus Y)$, whilst in the latter case $g(W) + g(Y) \leq g(\bar{W} \otimes Y) + g(\bar{W} \oplus Y)$. Note that each edge $\alpha_i, 1 \leq i \leq 4$, is incident to both a square and a triangle. Thus we are clearly done if the optimal pairing is (α_i, α_j), where $i \neq j$. The pairing (α_i, α_i) is not a feasible pairing for $g(W) + g(Y)$. So suppose the optimal pairing contains one of the β_i. By symmetry we may assume it contains β_1. Thus the optimal pairing is either (β_1, β_1) or $(\beta_1, \alpha_i), 1 \leq i \leq 4$. In these cases both triangles are incident to different edges in the pairing and thus $g(W) + g(Y) \leq g(W \otimes Y) + g(W \oplus Y)$. Hence g, and thus f, is skew bisupermodular. □

A basic optimal solution to LP relaxation for the element connectivity problem can be found in polynomial time. This follows because a strong separation oracle may be obtained using standard network flow techniques (for more details, see [26]).

To prove the key result for the 2-approximation guarantee for element connectivity, namely, there exists an edge of value at least $\frac{1}{2}$ in any non-zero basic solution of the setpair LP relaxation. we need to develop some preliminaries and a key lemma. Based on this, the rest of the proof is an extension of the counting arguments used in [25]. Let $\boldsymbol{x}$ be a non-zero basic solution to the LP. We may assume that each edge $e \in E$ has $x_e > 0$, since any other edges may be discarded. In addition, we may assume that each edge $e \in E$ has $x_e < 1$, otherwise we are done. Let $\mathcal{L}$ be the corresponding non-overlapping family of setpairs of Theorem 2.3.3. Since E is the support of a basic solution, $|E| = |\mathcal{L}|$. Let m denote $|E|$. Let τ be an arbitrarily chosen terminal vertex in T. For each setpair in $\mathcal{L}$, we assume that the head contains τ. Otherwise, if τ is in the tail of $W \in \mathcal{L}$, then we replace W by $\overline{W}$ and this preserves all the properties of $\mathcal{L}$ in Theorem 2.3.3. Then the tails of the setpairs in $\mathcal{L}$ form a laminar family. This holds because the setpairs in $\mathcal{L}$ are pairwise non-overlapping, and the heads of any two setpairs in $\mathcal{L}$ intersect (since τ is in every head). We may also view $\mathcal{L}$ as a forest of rooted trees. Call $W \in \mathcal{L}$ the *parent* of $Y \in \mathcal{L}$ (and Y a *child* of W) if $W_t \supseteq Y_t$ and any other setpair $Z \in \mathcal{L}$ with $Z_t \supseteq Y_t$ either has $Z_t \supset W_t$ or has $Z_t = W_t$ and $Z_h \subset W_h$. We may refer to the setpairs in $\mathcal{L}$ as *nodes* of the forest. We say that a setpair $Y \in \mathcal{L}$ is *smaller* than a setpair $W \in \mathcal{L}$ if $Y \succeq W$, i.e., $Y_t \subseteq W_t$ and Y is a descendant of W in the forest.

In the rest of this section, we take the edges in E to be bidirected. That is, we replace every undirected edge (p,q) in E by a pair of directed arcs (p,q) and (q,p). For a setpair $W = (W_t, W_h)$, let $\delta(W)$ denote the set of arcs (p,q) with $p \in W_t$ and $q \in W_h$. (Though we use the same notation $\delta()$ for undirected edges and directed arcs, the context will resolve any ambiguity.) The other notation, $\boldsymbol{x}$, χ_W, etc. remains the same. An arc (p,q) is called *good* if there is a setpair $W \in \mathcal{L}$ such that (p,q) is in $\delta(W)$. Note that, in the undirected setting, every edge e is in $\delta(W)$ for some $W \in \mathcal{L}$, but in the directed setting, there may exist arcs that are not good.

The proof of the main result hinges on assigning *token arcs* to the setpairs in $\mathcal{L}$. Here is a brief sketch; the details are given in the proof of Theorem 2.3.12. We have $|\mathcal{L}| = |E| = m$ and thus a total of $2m$ arcs. If a basic solution $\boldsymbol{x}$ of has $x_e < \frac{1}{2}$ for each edge e, then we will be able to distribute the token arcs among the setpairs in $\mathcal{L}$ such that each gets at least two token arcs and some setpair gets at least three token arcs. Thus, we end up with at least $2|\mathcal{L}|+1 = 2m+1$ arcs. This contradiction shows that our assumption on $\max_e \{x_e\}$ is false. We use the following two rules to distribute the token arcs such that no arc is assigned to two different setpairs.

RULE 1. If (p,q) is a good arc, then we assign it to the smallest setpair $W \in \mathcal{L}$ such that $(p,q) \in \delta(W)$.

RULE 2. If (p,q) is not a good arc, then we assign it to the smallest setpair $W \in \mathcal{L}$ such that $q \in W_t$ and $p \notin W_h$ (note that neither (p,q) nor (q,p) is in $\delta(W)$).

We note that this second rule is essential in the sense that the analysis fails if we insist on assigning arcs (p,q) only to those setpairs W such that $(p,q) \in \delta(W)$.

Given two setpairs W and Y, let $\delta(W) \triangledown \delta(Y)$ denote the symmetric difference of $\delta(W)$ and $\delta(Y)$. Given $S^1, S^2, \ldots, S^\ell$, a collection of mutually disjoint vertex sets, we denote by $\gamma(S^1, S^2, \ldots, S^\ell)$ the set of arcs (p,q) such that p and q are in different sets S^i and S^j, where $1 \le i,\, j \le \ell$.

Lemma 2.3.10. *Let W be a setpair in the forest $\mathcal{L}$, and let its children be the setpairs $Y^1, Y^2, \ldots, Y^\ell$. Then for every arc (p,q) in*

$$(\delta(W) \triangledown (\delta(Y^1) \cup \cdots \cup \delta(Y^\ell))) \ \setminus \ \gamma(Y_t^1, \ldots, Y_t^\ell)$$

either (p,q) or (q,p) is assigned to W by Rule 1 or Rule 2.

Proof. Note that the lemma applies to all arcs in the symmetric difference of $\delta(W)$ and $\bigcup_{i=1}^{\ell} \delta(Y^i)$ except for arcs whose two end-vertices are in the tails of different children. Figure 2-7 illustrates the arguments that follow, for the particular case of $\ell = 2$. [Note, for example, that the arc a_4 is in $\gamma(Y_t^1, Y_t^2)$.]

We have two cases to deal with.

(a) $(p,q) \in \delta(W) \setminus \bigcup_{i=1}^{\ell} \delta(Y^i)$.

Now $q \in W_h \subseteq Y_h^i$, for $i = 1, \ldots, \ell$. However, by assumption, $p \in W_t \setminus \cup_{i=1}^{\ell} Y_t^i$ and so W is the smallest setpair in $\mathcal{L}$ such that p is in the tail. Thus, W is the smallest setpair in $\mathcal{L}$ with $(p,q) \in \delta(W)$, and hence (p,q) is assigned to W by Rule 1. [The

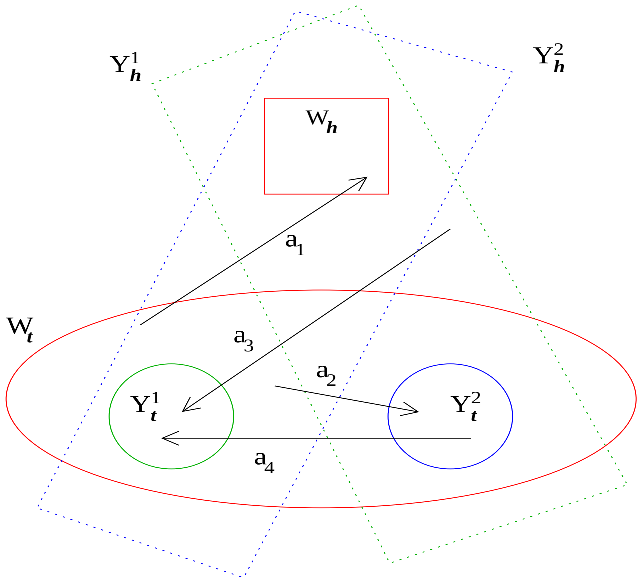

Figure 2-7: Arc assignment rules.

arc a_1 in Figure 2-7 is an example.]

(b) $(p, q) \in \delta(Y^i) \setminus \delta(W)$, for some $i \in \{1, \ldots, \ell\}$, and $(p, q) \notin \gamma(Y_t^1, \ldots, Y_t^\ell)$.

Note that $p \in Y_t^i \subseteq W_t$ and $q \notin W_h$. We have two subcases.

(i) $q \in W_t$.

First, we show that (q, p) is not a good arc. Clearly, $q \notin Y_t^i$ as $(p, q) \in \delta(Y^i)$. For any other child Y^j, observe that $q \notin Y_t^j$ as $(p, q) \notin \gamma(Y_t^1, \ldots, Y_t^\ell)$. Hence, $q \in W_t \setminus \cup_{j=1}^{\ell} Y_t^j$. Then W is the smallest setpair in $\mathcal{L}$ whose tail contains q, and there is no setpair $Z \in \mathcal{L}$ such that $(q, p) \in \delta(Z)$ (otherwise, $p \in Z_h \cap W_t$ and $q \in Z_t \subseteq W_t$, which is impossible). Thus (q, p) is not a good arc. Moreover, $p \notin W_h$ (since $p \in W_t$), and so Rule 2 assigns (q, p) to W. [The arc a_2 in Figure 2-7 is an example.]

(ii) $q \notin W_t$.

We first show that (q, p) is not a good arc. Suppose $Z \in \mathcal{L}$ is a setpair such that $(q, p) \in \delta(Z)$. We derive a contradiction by showing that W and Z overlap; in particular, W and $\overline{Z}$ cross. Note that W_t and Z_t are disjoint, because $p \in W_t \setminus Z_t$ and $q \in Z_t \setminus W_t$. Now Z_t and W_h have a terminal in common, because Z_t has at least one terminal and $W_t \cup W_h$ contains all the terminals. Then Z_t and W_h intersect properly, because

(1) There is a terminal in $Z_t \cap W_h$

(2) The terminal τ is in $W_h \setminus Z_t$

(3) We have $q \in Z_t \setminus W_h$

Finally, note that Z_h and W_t intersect since both contain the vertex p. Hence, $\overline{Z}$ and W cross. Therefore, (q,p) is not a good arc and Rule 2 applies to (q,p). Rule 2 assigns (q,p) to W since $p \in W_t$, $q \notin W_h$, and for any smaller setpair $X \in \mathcal{L}$ with $p \in X_t$ we have $q \in X_h$ (since $p \in Y_t^i$ and $q \in Y_h^i$). [The arc a_3 in Figure 2-7 is an example.] □

We need the following lemma in order to apply the parity arguments required in the proof of Theorem 2.3.12.

Lemma 2.3.11. *If Y and Z are tail disjoint setpairs in $\mathcal{L}$ then $Y_t \subseteq Z_h$ and $Z_t \subseteq Y_h$.*

Proof. Let $Y = (Y_t, Y_h)$ and $Z = (Z_t, Z_h)$ be setpairs in $\mathcal{L}$. To see the lemma, note that Y_t and Z_h have a terminal in common, since Y_t has at least one terminal, and $Z_t \cup Z_h$ contains all the terminals. Similarly, Z_t and Y_h have a terminal in common. Suppose that Y_t is not a subset of Z_h. Then it can be seen that Y_t and Z_h intersect properly. Hence, $\overline{Z}$ crosses Y, and so Y and Z overlap. This contradiction shows that $Y_t \subseteq Z_h$. Similarly, we have $Z_t \subseteq Y_h$. □

Theorem 2.3.12. *Any non-zero basic solution of the setpair LP for element connectivity has an edge of weight at least $\frac{1}{2}$.*

Proof. Let $\boldsymbol{x} \neq \boldsymbol{0}$ be a basic solution such that $x_e < \frac{1}{2}$ for each edge e. Then the following claim implies that we end up with at least $2|\mathcal{L}|+1$ arcs. This is a contradiction. Hence, $\max_e \{x_e\} \geq \frac{1}{2}$. Define the *corequirement* of an edge e, denoted $\alpha(e)$, to be $\frac{1}{2} - x_e > 0$, and the *corequirement* of a setpair W, denoted $\alpha(W)$, to be $\sum_{e \in \delta(W)} \alpha(e)$. Note that a tight setpair W has $\alpha(W) = \frac{1}{2}|\delta(W)| - f(W) > 0$. We will use the fact that if a setpair W has $\alpha(W) = \frac{1}{2}$, then $|\delta(W)|$ is odd.

Claim. *Suppose that $x_e < \frac{1}{2}$, $\forall e \in E$. Then the arcs may be redistributed to the nodes in the forest $\mathcal{L}$ such that for each rooted subtree of $\mathcal{L}$, each node gets at least two token arcs and the root gets at least three token arcs. Moreover, a root with corequirement $\neq \frac{1}{2}$ gets at least four token arcs.*

Again, we prove the claim by induction. For the base case take a leaf node $W \in \mathcal{L}$. Then $\delta(W)$ has at least 3 arcs. By Rule 1, all these arcs are assigned to W. If W gets exactly 3 token arcs, then note that $f(W) = 1$ and $|\delta(W)| = 3$, so we have $\alpha(W) = \frac{1}{2}$. Thus W satisfies the induction hypothesis.

For the induction step, consider a subtree rooted at a node $W \in \mathcal{L}$. We have four cases.

(a) W has at least four children.

By the induction hypothesis, each child has at least 3 token arcs. We reassign one token arc from each child to W. Hence, W gets at least 4 token arcs.

(b) W has three children.

Call the children X, Y and Z. If one of the children has at least 4 token arcs, then we can reassign the token arcs from X, Y and Z to W so that W has at least 4 token arcs and each of X, Y and Z is left with at least 2 token arcs.

Otherwise, by the induction hypothesis, each of X, Y and Z has 3 token arcs, so each has corequirement $\frac{1}{2}$ and each of $|\delta(X)|$, $|\delta(Y)|$ and $|\delta(Z)|$ is odd. We reassign one token arc from each of X, Y and Z to W. Now focus on the symmetric difference of $\delta(W)$ and $\delta(X) \cup \delta(Y) \cup \delta(Z)$. By the linear independence of χ_W, χ_X, χ_Y and χ_Z, there is at least one arc (p, q) in the symmetric difference. If possible, choose $(p, q) \notin \gamma(X_t, Y_t, Z_t)$. Then by Lemma 2.3.10 either (p, q) or (q, p) is assigned to W as a token arc. Hence, W ends up with at least 4 token arcs. Otherwise, every arc in the symmetric difference is in $\gamma(X_t, Y_t, Z_t)$. Therefore $\delta(W) \subset \delta(X) \cup \delta(Y) \cup \delta(Z)$, and for each arc (p, q) in $(\delta(X) \cup \delta(Y) \cup \delta(Z)) \setminus \delta(W)$ the arc (q, p) is also in the same set, by Lemma 2.3.11. Hence, $|\delta(W)|$ is odd, and so the corequirement $\alpha(W)$ is a *semi integer* (i.e., an odd integer multiple of $\frac{1}{2}$). Moreover, $\alpha(W) < \alpha(X) + \alpha(Y) + \alpha(Z) = \frac{3}{2}$. Hence, $\alpha(W) = \frac{1}{2}$, and so W satisfies the induction hypothesis.

(c) W has two children.

Call the children Y and Z. If we can assign 4 token arcs to W by reassigning token arcs from Y and Z, and by applying Lemma 2.3.10 to $(\delta(W) \triangledown (\delta(Y) \cup \delta(Z))) \setminus \gamma(Y_t, Z_t)$, then we are done. Otherwise, we are left with two subcases.

(i) Each of Y and Z has 3 token arcs and a corequirement of $\frac{1}{2}$, and there is one arc

(p, q) in $(\delta(W) \triangledown (\delta(Y) \cup \delta(Z))) \setminus \gamma(Y_t, Z_t)$. Then W gets 3 token arcs, one each by reassigning arcs from Y and Z and one by Lemma 2.3.10. Both $|\delta(Y)|$ and $|\delta(Z)|$ are odd (since $\alpha(Y) = \alpha(Z) = \frac{1}{2}$) and the arcs (if any) in $\gamma(Y_t, Z_t)$ occur in pairs (u, v) and (v, u), by Lemma 2.3.11. Therefore, $|\delta(W)|$ is odd and $\alpha(W)$ is a semi integer. Moreover, $0 < \alpha(W) \leq \alpha(Y) + \alpha(Z) + \alpha(\{p, q\}) < \frac{3}{2}$. Hence, $\alpha(W) = \frac{1}{2}$, and so W satisfies the induction hypothesis.

(ii) At least one of the children, say Z, has 3 token arcs and a corequirement of $\frac{1}{2}$, and there are no arcs in $(\delta(W) \triangledown (\delta(Y) \cup \delta(Z))) \setminus \gamma(Y_t, Z_t)$. We will prove that this case cannot occur. First, note that $|\delta(W)|$ and $|\delta(Y)|$ have opposite parity (one is odd and the other is even), because $|\delta(Z)|$ is odd and the arcs in the symmetric difference of $\delta(W)$ and $\delta(Y) \cup \delta(Z)$ occur in pairs (u, v) and (v, u). Furthermore, note that χ_W, χ_Y and χ_Z are linearly independent. It follows that there is an arc in $\gamma(Y_t, Z_t)$ and there is an arc in $\delta(Z) \cap \delta(W)$. Hence, $\alpha(W) < \alpha(Y) + \alpha(Z)$ (due to the arc in $\gamma(Y_t, Z_t)$) and $\alpha(Y) < \alpha(W) + \alpha(Z)$ (due to the arc in $\delta(Z) \cap \delta(W)$). Therefore, $\alpha(Y) - \frac{1}{2} < \alpha(W) < \alpha(Y) + \frac{1}{2}$, implying that $\alpha(Y) = \alpha(W)$ (since $\alpha(W)$ is an integer multiple of $\frac{1}{2}$). Then $|\delta(W)|$ and $|\delta(Y)|$ have the same parity. This contradiction shows that this case cannot occur.

(d) W has one child.

Call the child Y. If the symmetric difference of $\delta(W)$ and $\delta(Y)$ has exactly 2 arcs and $\alpha(Y) = \frac{1}{2}$, then W gets 3 token arcs (two by Lemma 2.3.10 and one reassigned from Y). In this case, note that $f(W) = f(Y)$ and $|\delta(W)| = |\delta(Y)|$. As a result, $\alpha(W) = \alpha(Y) = \frac{1}{2}$. Otherwise, W gets at least 4 token arcs. In either case, W satisfies the induction hypothesis. This completes the proof of the claim.

The theorem follows. $\square$

Thus, we get a factor 2 approximation for the element connectivity matching the result of Fleischer et al [14].

Theorem 2.3.13. *There is a polynomial-time algorithm for the element connectivity problem that finds a solution of cost at most twice the optimal cost.* $\square$

Chapter 3

Bi-Connectivity in Unweighted Graphs

In this chapter, we consider bi-connectivity problems in unweighted graphs. For undirected graphs we tackle the minimum 2-edge connected spanning subgraph problem ($2EC$) and the minimum 2-vertex connected spanning subgraph problem ($2VC$). The results given in this chapter are from the papers [39] and [40]. Since these bi-connectivity problems are maxSNP-hard ([9], [3]) our focus will remain approximation algorithms. Khuller and Vishkin [30] gave a $\frac{3}{2}$-approximation algorithm for $2EC$. Cheriyan et al [5] have recently improved upon this, with a $\frac{17}{12}$-approximation algorithm. In [30], Khuller and Vishkin also gave a $\frac{5}{3}$-approximation algorithm for $2VC$. This was improved to $\frac{3}{2}$ by Garg et al [18]. This chapter presents $\frac{4}{3}$-approximation algorithms for both $2EC$ and $2VC$. The ratio $\frac{4}{3}$ has a special significance; a celebrated conjecture in combinatorial optimisation says that the travelling salesman problem on metrics is approximable to within $\frac{4}{3}$ via a linear programming relaxation called the *subtour relaxation*. A (previously unverified) implication of this conjecture is that $2EC$ is also approximable to within $\frac{4}{3}$.

Bi-connectivity in directed graphs corresponds to the property of *strong connectivity*. A directed graph is strongly connected if, for every pair of vertices u and v, there is a directed path from u to v and vice versa. Thus, for directed graphs we tackle the minimum strongly connected spanning subgraph problem (SCS). A slightly more general problem is the minimum equivalent digraph problem, denoted MED. Here we are given a directed graph G and we wish to find the smallest subgraph that maintains the reachability relations between all pairs of vertices in G. This is a long studied problem, with Moyles and Thompson [36] showing that MED decomposes into two parts; a solution to the strongly connected subgraph problem on each strongly connected component of G and a solution of MED on an induced acyclic graph. The latter problem can be solved exactly in polynomial time [24]. It follows that an α-approximation algorithm for the strongly connected subgraph problem can easily be converted into an α-approximation algorithm for the minimum equivalent digraph problem. The current best approximation algorithm for SCS was given by Khuller, Raghavachari and Young ([28],[29]). Their algorithm has an approximation factor of just over 1.617. In this chapter, we improve this factor to $\frac{3}{2}$.

The basic algorithmic approach we take is the same for all three problems, that is $2EC$, $2VC$ and SCS. An overview of our approach is as follows. In Section 3.2 we present the basic lower bound against which the performance of our algorithm will be measured. To obtain this bound we find the minimum sized spanning subgraph in which every vertex has degree at least two (or in-degree and out-degree at least one in directed graphs). This can be done in polynomial-time via a reduction to the maximum matching problem. Section 3.3 contains decomposition theorems which are the basis of our algorithm. These decompositions allow us to focus on graphs which do not contain certain problematic structures. In such graphs the lower bound solution can be modified to obtain a solutions to $2EC$, $2VC$ and SCS without increasing their size significantly. The bulk of the analysis lies in showing that such a low-cost modification is possible. This involves detailed arguments for several tricky configurations; these are given in Section 3.6, Section 3.7 and Section 3.8 for the edge connectivity, vertex connectivity, and strong connectivity cases, respectively.

3.1 Two Simple Approximation Algorithms

In this section we give two simple approximation algorithms for the $2EC$ problem. Both algorithms give factor 2 guarantees for the problem. We present these algorithms as the basic ideas they rely on will be incorporated into our algorithms for $2EC$, $2VC$ and SCS.

3.1.1 The DFS Tree Method

The first algorithm is a "depth first search tree based" algorithm. First we grow a depth first search (DFS) spanning tree T rooted at any vertex r in G. A property of DFS trees is that any non-tree edge is incident to two vertices that lie on a path from the root to a leaf in the tree. We say that an edge (u, w) is a back edge with respect to v if w, v, and u lie (in that order) on the same path from the root to a leaf node l. Observe that for each vertex v there is at least one associated back edge. If not then the removal of the tree edge above v in T disconnects the subtree rooted at v from the rest of the graph; thus G is not 2-edge (vertex) connected. Our solution is the edges of T along with a back edge with respect to each vertex. It is easy to see that such a solution is 2-edge connected. In an n vertex graph this produces a solution with at most $2n - 2$ edges. Since the optimum has at least n edges, this is a 2-approximation.

3.1.2 The Cycle Contraction Method

The second algorithm is a "cycle-contraction" algorithm. Find a cycle C in G that contains a specified vertex r, called the *root vertex*. We will take C to form part of our final solution. Next consider $G' = G/C$, the graph formed by the contraction of the cycle C. Let the super-vertex formed by the contraction of C be our root vertex r' in G'. Recurse until the graph contains a single vertex. At this point it is clear that the cycles found in each iteration together form a strongly connected subgraph in G. We now show that this cycle contraction algorithm has a performance guarantee of 2. Given a contracted cycle C, charge the edges in the cycle to the vertices of the cycle as follows. The root vertex r is not charged. One of the vertices neighbouring r in C, say v_2, is charged the two edges adjacent to it in the cycle. The other vertices in the cycle (if C contains more than two vertices) are charged for one adjacent edge each. Notice that a vertex receives a charge only when it is contracted into the root vertex. Since every edge in our solution is charged to some vertex and every vertex is charged at most 2 edges, we see that our solution contains at most $2n-2$ edges. Hence this approach does indeed give a 2-approximation algorithm.

Observe that the cycle-contraction algorithm does not necessarily give a 2-vertex connected subgraph; this, though, will not be of concern to us. Note, however, that directed versions of both these algorithms work for the SCS problem.

3.2 The Lower Bound

3.2.1 The Lower Bound in Undirected Graphs

The idea behind the algorithms is to take a minimum sized spanning subgraph with minimum degree 2 and, at a small cost, make it 2-edge connected. We will denote by $U2$ the problem of finding, in an undirected graph, a minimum sized spanning subgraph in which each vertex has degree at least two. Notice that the solution to $U2$ (which may not be connected) provides a lower bound for both $2EC$ and $2VC$.

Lemma 3.2.1. *The size of the optimal solution to $U2$ is a lower bound on the size of the optimal solutions to $2EC$ and $2VC$.*

Proof. Any 2-connected subgraph must have minimum degree at least two. □

This gives a possible method for approximating $2EC$ and $2VC$. Find an optimal solution to $U2$. Alter the solution in some way to get a 2-connected graph. The rest of this chapter is devoted to specifying the "some way" and proving that the method gives a solution whose size is at most $\frac{4}{3}$ the size of the optimum solution for $U2$. Hence it is at most $\frac{4}{3}$ the size of the optimum solution for $2EC$ or $2VC$.

The conversion of a solution of $U2$ to a solution of $2EC$ or $2VC$ will only take $O(m)$ time and, thus, the overall running time is dominated by the time to solve $U2$.

The latter problem can be solved exactly in polynomial time. One fast way to do this is to find a maximum sized subgraph in which each vertex has degree at most two (this is the *maximum cycle-path cover* problem which asks for the maximum sized subgraph consisting of disjoint cycles and paths) and add an arc to the end vertices of each path. This will give an optimal solution to $U2$. This follows from the fact that if the minimum $U2$ solution contains $n+p$ edges then there are $2p$ edges whose removal induces a collection of p disjoint paths plus some disjoint cycles. The maximum cycle-path cover problem has a simple reduction to maximum matching so the running time is $O(m\sqrt{n})$.

3.2.2 The Lower Bound in Directed Graphs

For the SCS problem we use a directed version of the lower bound for the $2EC$ and $2VC$ problems. Consider the problem of finding, in a directed graph, the minimum sized subgraph in which every node has both in-degree and out-degree at least one. Call this problem $D2$. Again, an optimal solution to $D2$ can be computed in polynomial time via a reduction to the undirected maximum matching problem.

Lemma 3.2.2. *$D2$ gives a lower bound on the size of the optimal solutions to SCS.*

Proof. Every node must be incident to both an in-arc and an out-arc in any strongly connected subgraph. $\square$

We will take an optimal solution to $D2$ and alter it in order to obtain a strongly connected subgraph. We will show that this transformation process gives a strongly connected subgraph that is at most $\frac{3}{2}$ times the size of the optimum solution for SCS.

3.3 The Decomposition Theorems

How good are our lower bounds though? Take the lower bound $U2$ for our undirected problems. The following examples show that, on its own, it is not sufficient for our purposes. Consider finding a minimum 2-edge connected subgraph in the graph shown in Figure 3-1i). The optimal solution is $\frac{3}{2}$ times larger than the optimal solution to

$U2$. To see this, assume that the number of vertices is odd. The minimum 2-edge connected subgraph is the whole graph and, thus, contains $\frac{3}{2}(n-1)$ edges. However, the graph contains a Hamiltonian path. Therefore, the lower bound contains $(n-1)+2$ edges (we need to add an extra edge at each end-vertex of the path to give a feasible $U2$ solution). Similarly if we wish to find a minimum 2-vertex connected subgraph in the graph in Figure 3-1ii), it is easy to see that this is also $\frac{3}{2}$ times larger than the optimal solution to $U2$.

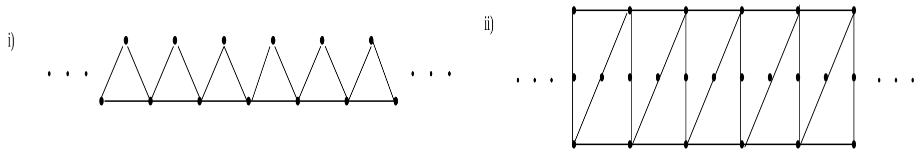

Figure 3-1: Troublesome examples.

In this section we will show how to strengthen our lower bounds via the use of decompositions.

3.3.1 2-Edge Connectivity

Cut Vertices

Dealing with the difficulty in the example given in Figure 3-1i) is not difficult. We assume that for the problem $2EC$ our graph contains no cut vertices. This condition entails no loss of generality. An α-approximation algorithm for 2-vertex connected graphs can be used to give α-approximate solutions for general graphs. Simply apply the algorithm separately to each maximal 2-vertex connected block, then combine the solutions to each block.

We may also assume that G has no adjacent degree 2 vertices. This also entails no loss of generality. Observe that both edges incident to a degree two vertex must be contained in any 2-edge connected spanning subgraph. Hence, for a path whose internal vertices are all degree 2, every edge in the path must be chosen. This allows us to transform an instance of the general case into an instance with non-adjacent degree 2 vertices. Given G, contract, to just two edges, any such path. Call the resulting graph G'. Given the trivial bijection between solutions to G and solutions to G', it follows that any α-approximate solution to G' induces an α-approximate solution to G.

Beta Structures

Beta Structures The example of Figure 3-1ii) is more troublesome. To counter it we need to eliminate a certain graph structure, which we will call a *beta-structure*. A vertex u is a *beta-vertex* if the removal of two vertices v_1 and v_2 induces at least three components, one of which consists of just u. This situation is illustrated in Figure 3-2, with the other two components labelled C_1 and C_2. Without loss of generality, we may assume that the component C_1 is smaller (i.e. contains fewer vertices) than C_2.

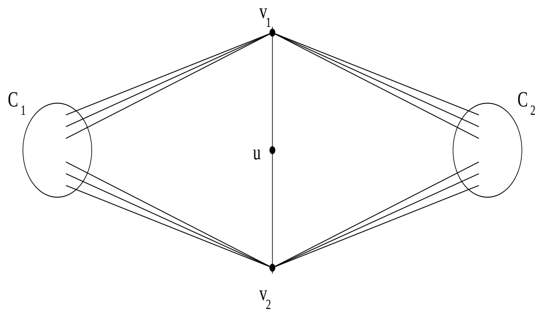

Figure 3-2: A beta-vertex.

We say that two adjacent vertices u_1 and u_2 are a *beta-pair* if there exist two nodes, v_1 and v_2, whose removal leaves at least 3 components, one of which contains just u_1 and u_2.

This situation, with the other two components labelled C_1 and C_2, is shown in Figure 3-3. Note, the figure shows all four edges $(u_1, v_1), (u_1, v_2), (u_2, v_1)$ and (u_2, v_2) present. It may be the case that, for a beta-pair, only three of these edges are present. This will not affect our argument in any way. In addition, for both beta-vertices and beta-vertices we may assume that the edge (v_1, v_2) is not present in the graph G. To

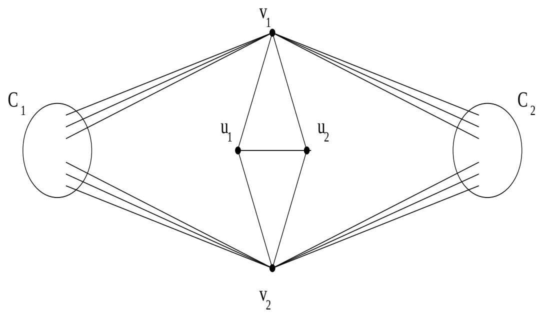

Figure 3-3: A beta-pair.

see this note that there is some optimal subgraph in G that does not contain the edge (v_1, v_2); thus it suffices to find an α-approximate solution in $G - (v_1, v_2)$. We say that a graph without a beta-vertex or beta-pair is *beta-free*.

Here we will show how to recursively obtain, in polynomial time, a lower bound $l(G)$ on the optimal solution. In the following sections we will show how to augment the solution induced by the lower bound to give a feasible 2-edge connected subgraph whose size is at most $\frac{4}{3}$ that of the lower bound. The lower bound is obtain by decomposing the graph around beta-structures.

First we show that the case of beta-pairs can be reduced to that of beta-vertices. Observe that both edges incident to a beta-vertex u must be in any solution. These two edges induce a path of length two from v_1 to v_2. Similarly given a beta-pair u_1 and u_2 it is easy to show that there is some optimal solution which contains a path of length three from v_1 to v_2 using the edge (u_1, u_2). It follows that if we find a beta-pair u_1 and u_2 in G we may contract the edge (u_1, u_2) and solve our problem on a smaller graph G' that contains a beta-vertex. Note, also, that this edge contraction reduces the size of the optimal solution in the graph by exactly one; the lower bound that we present will also be reduced by exactly one. Any approximation guarantee we may obtain, via the lower bound, for the contracted graph G' will then apply to the original graph G. Thus, it suffices to assume that all our beta-structures are beta-vertices.

Our basic technique to deal with a beta-vertex is as follows. We will decompose the graph into two pieces around the beta-vertex and iterate the procedure. Eventually we will have a decomposition into beta-free graphs. We will then work on these pieces and show that the sub-solutions obtained can be combined to give a good solution to the whole graph. Given a beta-vertex, u, with components C_1 and C_2, let G_1 be the subgraph formed by removing C_2 and its incident edges. Let G_1' be the graph formed by contracting $V - C_1$. These situations are shown in Figure 3-4i) and ii), where the super-node formed by contracting $V - C_1$ is labelled v'. The graphs G_2 and G_2' are defined analogously. Let k_1, k_1', k_2 and k_2' be the sizes of the optimal 2-edge connected

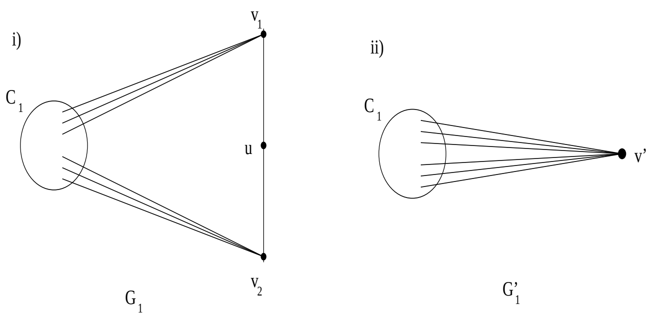

Figure 3-4: A split.

spanning subgraphs of G_1, G_1', G_2 and G_2', respectively. Let $\text{OPT}(G)$ be the size of the optimal solution to the whole graph.

Lemma 3.3.1. *Given a 2-edge connected graph,* $\text{OPT}(G) = \min(k_1 + k_2', k_1' + k_2)$.

Proof. It is clear that the combination of a solution to G_1 and G_2', or vice versa, gives a 2-edge connected subgraph for G. Conversely any solution to G can be decomposed to give a solution to G_1 and G_2', or to give a solution to G_1' and G_2. □

In addition, there is the following relationship between k_i and k_i'.

Lemma 3.3.2. *For* $i = 1, 2$, *we have* $k_i' + 2 \le k_i \le k_i' + 3$.

Proof. First note that to connect up u we must use both its incident edges. It follows that $k_i' + 2 \le k_i \le k_i' + 3$, $i = 1, 2$. This is because, given a 2-edge connected subgraph for G_i', we are forced to use 2 extra edges in connecting up u and we may need to use an extra edge from C_i in certain instances in which the solution to G_i' had all its edges leaving C_i going to the same vertex (either v_1 or v_2). $\square$

From Lemma 3.3.1 and Lemma 3.3.2, we obtain the following simple corollary.

Corollary 3.3.3. *If G has a beta-vertex inducing the components C_1 and C_2 then*

$$\begin{array}{llcl} (a) & k_1' + k_2' + 2 & \leq & \text{OPT}(G) \leq k_1' + k_2' + 3 \\ (b) & k_1 + k_2 - 3 & \leq & \text{OPT}(G) \leq k_1 + k_2 - 2 \end{array} \qquad \square$$

(Note that if the components C_1 and C_2 were induced by a beta-pair not a beta-vertex then we would obtain, for example, the inequalities $k_i' + 3 \leq k_i \leq k_i' + 4$, $i = 1, 2$. The lower bound defined below may then be constructed with regards to both beta-pairs and beta-vertices. However, it is cleaner to use the fore-mentioned reduction of beta-pairs to beta-vertices.)

To compute optimally the values k_1, k_1', k_2 and k_2' is as hard as the original problem. So instead we will need to find approximate solutions. First, though, we present a lower bound on $\text{OPT}(G)$ that is defined recursively. If G is beta-free then let $l(G)$ be given by the lower bound provided by $D2$. If G contains a beta-vertex then we evaluate $l(G_1)$ and $l(G_1')$ and, based upon their relative magnitudes, we either set $l(G) = l(G_1) + l(G_2')$ or set $l(G) = l(G_1') + l(G_2)$. The exact procedure is:

`LowerBound(G)`

{

`If` G `has no beta-vertex`

  `Then let` $l(G) = |D2(G)|$

`Else if` G `has a beta-vertex`

  `Then if` $l(G_1) \leq l(G_1') + 2$ `let` $l(G) = l(G_1) + l(G_2')$

  `Else if` $l(G_1) \geq l(G_1') + 3$ `let` $l(G) = l(G_1') + l(G_2)$

}

This recursive lower bound procedure induces the following decomposition into beta-free subgraphs.

```
Decompose(G)
{
If G is beta-free
   Then output G
Else if G has a beta-vertex
   If l(G_1) ≤ l(G'_1) + 2 then output Decompose(G_1) and Decompose(G'_2)
   If l(G_1) ≥ l(G'_1) + 3 then output Decompose(G'_1) and Decompose(G_2)
}
```

Since we have that $|C_1| \leq |C_2|$, this recursive procedure can be applied in polynomial time.

Theorem 3.3.4. *The decomposition procedure can be implemented in polynomial time.*

Proof. Let $f(n)$ be the running time for our algorithm on a graph of n vertices. The time required to search for possible decompositions is $O(mn)$ in G. In addition, if G is beta-free then we require $O(m\sqrt{n})$ to find a solution to $D2$. For graphs with beta-structures, the running time is then at most $f(n) = f(n-a) + f(a+1) + f(a+3) + O(mn)$, where $1 \leq a \leq \frac{1}{2}n$. Hence, $f(n) \leq \max(f(n-a) + 2f(a+3) + O(mn), O(m\sqrt{n}))$. Solving the recurrence we get that $f(n) = O(mn^2)$. □

We now prove that $l(G)$ is indeed a lower bound on the size of the optimal 2-edge connected subgraph.

Theorem 3.3.5. *Given a 2-edge connected graph, $l(G) \leq \text{OPT}(G)$.*

Proof. We proceed by induction. If G is beta-free then $l(G)$ is given by the solution to $D2$, a lower bound on the optimal solution. So assume that G contains a beta-vertex. By construction of $l(G)$, we have two possibilities:

(i) $l(G_1) \leq l(G'_1) + 2$: thus we have

$$\begin{aligned} l(G) &= l(G_1) + l(G'_2) && \text{by LowerBound}(G) \\ &\leq l(G'_1) + 2 + l(G'_2) \\ &\leq k'_1 + 2 + k'_2 && \text{by induction} \\ &\leq \text{OPT}(G) && \text{by Corollary 3.3.3a)} \end{aligned}$$

(ii) $l(G_1) \geq l(G'_1) + 3$: thus we have

$$\begin{aligned} l(G) &= l(G'_1) + l(G_2) && \text{by LowerBound}(G) \\ &\leq l(G_1) - 3 + l(G_2) \\ &\leq k_1 - 3 + k_2 && \text{by induction} \\ &\leq \text{OPT}(G) && \text{by Corollary 3.3.3b)} \end{aligned}$$

□

Corollary 3.3.6. *Given an algorithm that finds a solution with at most $\alpha|D2(G)|$ edges for any beta-free graph, we get an α-approximation algorithm for any graph.*

Proof. Consider the decomposition tree induced by the decomposition procedure. Let $H_1, H_2, \ldots H_p$ be its leaves. Then $\text{OPT}(G) \geq l(G) = |D2(H_1)| + |D2(H_2)| + \cdots + |D2(H_p)|$, whilst the number of edges used by the algorithm is at most $\alpha\left(|D2(H_1)| + |D2(H_2)| + \cdots + |D2(H_p)|\right)$. □

Thus, it suffices to consider only beta-free graphs. It is in these graphs that we will find $\frac{4}{3}$-approximate solution based upon the subgraph induced by the $D2$ solution.

3.3.2 2-Vertex Connectivity

Beta Structures

Consider the case of a beta-vertex u for the 2-vertex connectivity problem. We deal with this situation as follows. Given G, let $G' = G \setminus \{u\}$. See Figure 3-5i) and ii). Given that the two edges incident to u are forced in any solution to G, we have the following simple lemma.

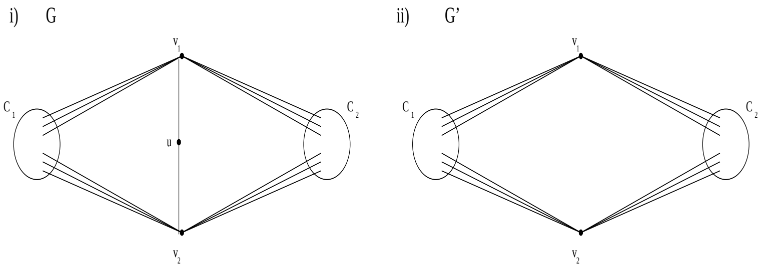

Figure 3-5: Beta-decompositions in vertex connectivity.

Lemma 3.3.7. *Consider the problem $2VC$ on a graph G that contains a beta-vertex u. An α-approximation for G can be obtained from an α-approximation for G' by adding the arcs incident to u.* □

So for $2VC$ our decomposition is straightforward. Remove the beta-vertex and consider the induced subproblem.

3.3.3 Strong Connectivity

Cut Vertices

In directed graphs, a vertex v is a *cut vertex* if there are two components C_1 and C_2 such that for every pair of vertices, $u_1 \in C_1$ and $u_2 \in C_2$, each directed path from u_1 to u_2, and vice versa, passes through v.

Lemma 3.3.8. *It is sufficient to consider only graphs that contain no cut vertices.*

Proof. Suppose G contains a cut vertex v. Then every strongly connected subgraph of G is the arc-disjoint union of a strongly connected subgraph on $C_1 \cup \{v\}$ and a strongly connected subgraph on $C_2 \cup \{v\}$. Hence we can decompose the problem into two and work on the pieces. Combining α-approximations for the pieces gives an α-approximation for G. □

Recall that we will present a factor $\frac{3}{2}$-approximation algorithm for SCS. The graph in Figure 3-6i) shows that this approximation guarantee is the best we will be able to obtain using the $D2$ lower bound in the absence of cut vertices. The solution to $D2$, shown in Figure 3-6ii), is a perfect cycle cover and hence contains n arcs. There is a $\frac{3}{2}$ gap, though, between this solution and the optimal solution to SCS, which is illustrated in Figure 3-6iii).

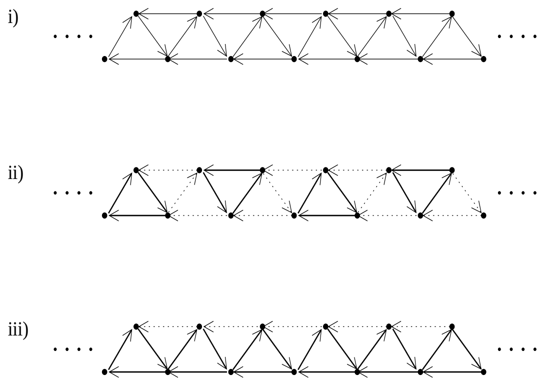

Figure 3-6: A tight example.

3.4 The Component Graph

3.4.1 2-Edge Connectivity

Our optimal solution to $U2$ can be divided into connected components of minimum degree two. Our first step is to partition these connected components into sub-components. These sub-components will be either 2-edge connected subgraphs or paths. This we can achieve in linear time in the following manner. Take a component of the solution to $U2$. Find and contract a cycle in the component. Recurse on the resultant contracted graph. This process terminates with a tree, T, consisting of vertices from the original graph and mega-vertices (representing 2-edge connected components). A diagram will help clarify this procedure, see Figure 3-7a). The 2-edge connected components (circled in the diagram) constitute nodes in T.

Given T, we find our sub-components as follows. All the mega-vertices will be chosen as sub-components. In addition, we partition the other vertices of the tree into path sub-components. To do this, we partition the tree into paths via, say, depth first

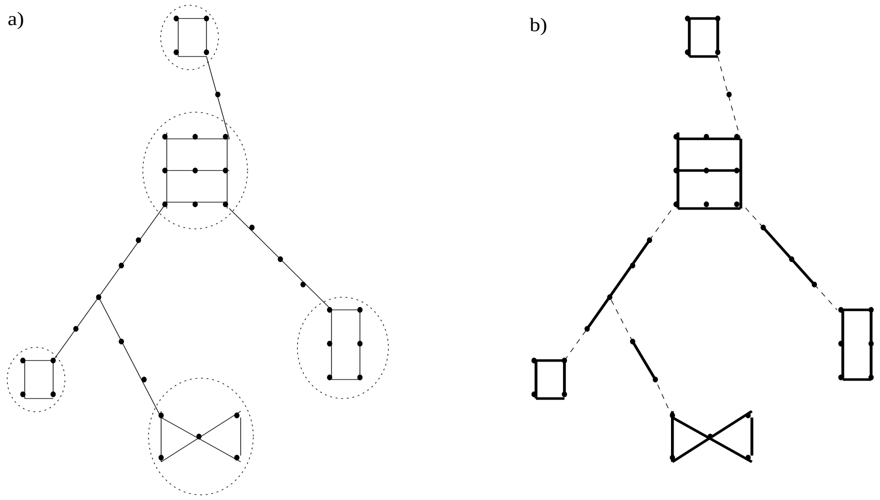

Figure 3-7: Mega-vertices.

search. We add the condition that any path, derived in this manner, which contains an internal mega-vertex be split into two disjoint paths. The key observation here is that associated with each path sub-component are two "free" edges; these are edges that are part of the lower bound but are not being used by any of the sub-components, This implies that we will be able to add extra edges later to make up for these free edges. Again a diagram illuminates this process, see Figure 3-7b). Here the sub-components are shown in bold, whilst the "free edges" are dashed. We denote by $\mathcal{G}$ the graph obtained by contracting these components so that they form nodes. This induced graph is called the *component graph*. Note that, for clarity, we refer to nodes in $\mathcal{G}$ and vertices in G.

For motivation, assume for a moment that each of the components is 2-edge connected and contains at least six edges. Let there be p such components and hence p nodes in the component graph $\mathcal{G}$. Performing the depth first search tree based algorithm gives a 2-edge connected subgraph on $\mathcal{G}$ containing at most $2p-2$ edges. Hence, at most two edges are associated with each node or component. These edges together with the edges in the original 2-edge connected components form a 2-edge connected subgraph on G. Since two additional edges are "charged" to each component and, in turn, each component is of size at least 6 this represents a $\frac{4}{3}$-optimal solution. This idea forms the basis of our algorithm: apply the depth first search tree based algorithm to the component graph $\mathcal{G}$. We have seen that this approach is appropriate when the components are 2-edge connected and contain at least 6 arcs. The components, though, may also represent small cycles or paths. In such circumstances we need to be more careful. For example, charging two arcs to a 3-cycle results in a local blow-up factor of $\frac{5}{3}$, which is unacceptable. We also need to ensure that we induce a 2-edge connected subgraph when path components are considered. These issues will be dealt with in subsequent sections.

3.4.2 2-Vertex Connectivity

We find a slightly different set of components for the vertex connectivity case. Recall that we observed that we could find an optimal solution to $U2$ by adding arcs to the end vertices of each path in an optimal solution to the maximum cycle-path cover problem. As such, for the vertex connectivity case, our components will be the paths and cycles given by the solution to the maximum cycle-path cover problem. Note that we again have two "free" edges associated with each path.

3.4.3 Strong Connectivity

We use a similar method for the strong connectivity problem. Our lower bound is a minimum sized subgraph in which every vertex is the tail of some arc and the head of another. This subgraph consists of disjoint components that are either strongly connected or they consist of strongly connected sub-components connected together by directed paths. An example of this latter type is shown in Figure 3-8i).

We break apart these components as before. We separate the strongly connected sub-components. We also remove the two end arcs of each path. These arcs, which we call the "free arcs" associated with a path, will not be forgotten, though, as they are included in our lower bound. This method is illustrated in Figure 3-8ii). After this process is complete we have a collection of disjoint sub-components. For clarity we will subsequently refer to these sub-components as components. These components are either paths or are strongly connected. Note that the paths may consist of just one vertex.

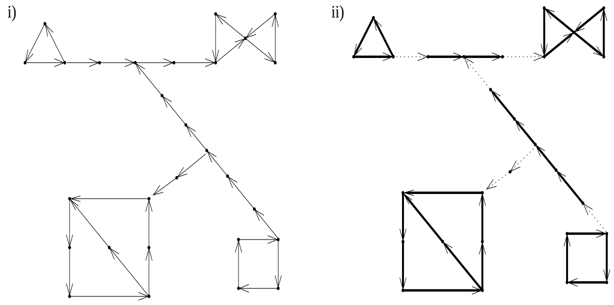

Figure 3-8: Finding components.

For motivation, assume for a moment that each of the components is strongly connected and contains at least four arcs. As with our undirected problems, the graph, $\mathcal{G}$, induced by contracting these components is called the *component graph*. Let there be p such components and hence p nodes in our induced graph. Performing the cycle contraction algorithm gives a strongly connected subgraph on $\mathcal{G}$ containing at most $2p-2$ arcs. Hence, at most two arcs are charged to each node or component. These arcs together with the arcs in the original strongly connected components form a strongly connected subgraph on G. Since two additional arcs are associated with each component and, in turn, each component is of size at least 4 this represents a $\frac{3}{2}$-optimal solution.

This idea forms the basis of our algorithm: apply the cycle contraction algorithm to the component graph $\mathcal{G}$. We have seen that this approach is appropriate when the components are strongly connected and contain at least 4 arcs. The components, though, may also represent 2-cycles (digons), 3-cycles or paths. In such circumstances we need to be more careful. For example, charging two arcs to a digon results in a local blow-up factor of 2, which is unacceptable. We also need to ensure that we induce a strongly connected subgraph when path components are contracted via the cycle-contraction algorithm. These issues will be dealt with in subsequent sections.

3.5 Incorporating a Depth First Search Tree

3.5.1 2-Edge Connectivity

We now have a component graph $\mathcal{G}$. Our next step is to add a collection of edges so that these components become connected. In particular, we will grow a depth first search spanning tree, $\mathcal{T}$, on $\mathcal{G}$. Note that the tree $\mathcal{T}$ is not the same as the tree T, previously described. For example, the tree formed may use other edges in the graph, not just those that were in our $U2$ solution.

Since the nodes in $\mathcal{G}$ may contain several vertices we need to clarify exactly how to build a DFS tree on the component graph. As we grow the tree we order the vertices within each component. We then examine the vertices for new tree edges according to that ordering. We use the notation $x \succ y$ when a vertex x is given priority over a vertex y. We have three cases.

1. The node represents a cycle. First search for edges incident to vertices adjacent to v in the cycle, where v is the vertex incident to the tree edge. Next consider the vertices at distance 2 from v along the cycle, etc. Finally consider v itself. (We add one extra technical condition. Note there are two vertices on different sides of the cycle that are the same distance from v. Should we encounter a vertex u at distance i that produces no children in the tree, then of the two vertices at distance $i + 1$ we give higher priority to the vertex on the same side as u.)

2. The node represents a path $v_0, v_1, \ldots, v_k$. Use the ordering $v_0 \succ v_1 \succ \cdots \succ v_k$.

3. The node is 2-edge connected: in this case, where the node is not a cycle, search the vertices in any order.

Building the tree in a depth first search manner gives the nice property that every edge in the graph is between a descendant node and one of its ancestor nodes (except for edges between vertices in the same node). We will call such edges "back edges" signifying that they go back towards the root.

3.5.2 2-Vertex Connectivity

For the vertex connectivity problem we apply exactly the same method as for the edge connectivity problem.

3.5.3 Strong Connectivity

We wish to apply the cycle contraction algorithm to the general case where the components may be digons, 3-cycles or paths. In order to apply this method efficiently we first grow a depth first search tree $\mathcal{T}$, as with the undirected graph problems, on the component graph $\mathcal{G}$. The tree will help us discover appropriate cycles in $\mathcal{G}$. In addition, the special structure provided by depth first search trees will help simplify the proofs concerning the performance of the algorithm.

We incorporate the tree as follows. We grow our tree from a node (component) r, henceforth termed the *root node* or *root component*. The tree, $\mathcal{T}$, will be an out-tree, a spanning tree with the property that there is a directed path from r to every other node. The tree is built via a DFS approach with root node r. Recall the important property of DFS trees that for any non-tree arcs (u, v), either v was added to the DFS tree before u or u lies on the directed path from r to v in the tree. Again, as the nodes in $\mathcal{G}$ may contain many vertices, we need to give a priority scheme for growing the tree.

1. Suppose that in growing the DFS tree we reach a component that represents a directed path. Let the path consist of the vertices $v_k, v_{k-1}, \ldots v_0$ with arcs (v_{i+1}, v_i). Use the ordering $v_0 \succ v_1 \succ \cdots \succ v_k$.

2. If the component represents a digon on the vertices v_0 and v_1, where v_0 is the vertex incident to the tree arc, use the ordering $v_1 \succ v_0$.

3. If the component represents a 3-cycle, $\{(v_0, v_1), (v_1, v_2), (v_2, v_0)\}$, where v_0 is the vertex incident to the tree arc, use the ordering $v_0 \succ v_2 \succ v_1$.

4. For a component representing a strongly connected component of size at least 4, we may order the vertices in any manner.

Given $\mathcal{T}$ and a component $C \in \mathcal{G}$, let T_C (or just T if there is no confusion) be the subtree of $\mathcal{T}$ rooted at C. If C contains the vertices $v_1, v_2, \ldots, v_k$ then $T_i \subseteq T_C$ is the subtree with root vertex $v_i \in C$. Denote by C_i the child components of v_i in T_i. There may be several such C_i for each v_i. In such circumstances, the context will make it apparent to which C_i we are referring.

We will use the following labels. We call an arc a *down arc* with respect to a component C, if it is directed from $G \setminus T_C$ to T_C. We call an arc a *back arc* with respect to a component C, if it is directed in the opposite direction.

We call the first non-root component C, discovered in growing $\mathcal{T}$, the *target component*. Our aim will be to apply a version of the cycle contraction algorithm in which we find and contract a cycle through the root and the target component, and then recurse. This method will be described in detail in the Section 3.8.2.

3.6 Edge Connectivity

3.6.1 The Local Improvement Method

Given our current subgraph, these components within a tree structure, how do we create a good 2-edge connected subgraph? We attempt to 2-connect up the tree, working up from the leaves, adding extra back edges as we go. Each edge that we add will be *charged* to a component. Our proof will follow from the fact that no component is charged more than one third its own weight.

First some terminology. Each component has one tree edge going up towards the root node. This edge we will call an *upper tree edge* w.r.t. the component. The component may have several tree edges going down towards the leaf nodes. These will be called *lower tree edges* w.r.t. the component. Initially each non-root component is charged one. This charge is for its incident upper tree edge.

Lemma 3.6.1. *For nodes representing* 2*-edge connected components the charge is at most* 2*.*

Proof. If we examine such a component then we may add any back edge from the node, in addition to the upper tree edge. This gives a charge of two to the component. □

Recall that if the component contains at least 6 edges then the associated charge is at most one third the components weight. It remains to deal with 2-edge connected components of at most 5 edges, and with paths.

Lemma 3.6.2. *For path nodes the charge is at most* $\lceil \frac{k}{3} \rceil + 2$*, where* k *is the number of edges in the path.*

Proof. Consider a path, P, of k edges. Let the node representing P be called the path node, N_0. Let the upper tree edge, e_0, charged to the path node be incident to the vertex v_0. Let the subtree rooted at the path node be called T_0. Let the children of the path node, $N_1, \ldots, N_r$, have associated upper tree edges $e_1, \ldots, e_r$. We may assume that these edges are incident to the vertices $v_1, \ldots, v_r$ on the path P. These

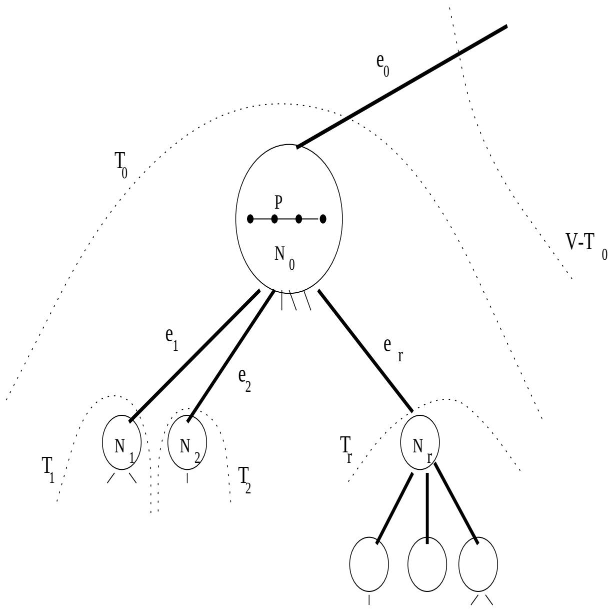

Figure 3-9: Subtrees.

child nodes are the root nodes of r induced subtrees, $T_1, \ldots, T_r$. This situation is shown in Figure 3-9.

Recall we work our way up from the leaf nodes. Hence, we may assume that all of the subtrees, $T_1, \ldots, T_r$, are already 2-connected up when we consider N_0. This allows us to carry out some preprocessing. First contract $V \setminus T_0$ into v_0. Consider the subtree T_1. There are at least two selected edges going beyond its root node N_1. One of these edges is e_1, let the other be e_1'. Let e_1' be incident to the path P at vertex v_1' (note that if e_1' goes beyond the path, P, then, since we contracted $V \setminus T_0$ into v_0, it is now incident to $v_1' = v_0$). Then the section of the path between v_1 and v_1' is already 2-connected. Hence we may contract into one super vertex that section of the path and T_1. We repeat this contraction process, in turn, for each subtree $T_2, \ldots, T_r$.

After this contraction process we are left with a single path, P' of at most k vertices. Two observations, concerning edges between vertices in P', follow from the DFS priority scheme used to choose tree edges. First, there are no edges that correspond to edges incident to a vertex in T_i and another in T_j, $i \neq j$. Such edges cannot exist as they would have been chosen instead of the tree edge e_j, where we assume that e_i was chosen before e_j. Second, consider an edge in P' corresponding to an edge between a vertex in T_i and a vertex in P. This edge is between v_i and some vertex v_j to the right of v_i in the path P', (possibly $v_i = v_j$), otherwise it would have been selected before e_i.

Suppose we 2-connect P'. We know the tree edge e_0 goes up to $V \setminus T_0$. If another such edge is added in 2-connecting up P', or if such an edge was already chosen for one of the subtrees $T_1, \ldots, T_r$, then we have successfully 2-connected up the subtree T_0. Otherwise, in addition to the edges chosen for P' we need one extra edge (e_0 has been chosen and we need one more). So the cost of connecting up T_0 is, at most, the cost of 2-connecting P' plus two edges.

So what is the cost of 2-connecting P'? In the worst case P' also contains k edges, and it turns out that the cost of 2-connecting such a path is at most $\lceil \frac{k}{3} \rceil$. To show this, we break up the path into blocks of at least 3 vertices such that each block is charged only one extra edge in the connection process.

Start with the leftmost vertex in the path. We add rightward edges wherever a path edge threatens to be a bridge. On adding an edge it may be possible to remove some current edge. If not, then all of the vertices to the left of the current bridge, not contained in any other block, form a new block. The blocks formed have the property that, except for the blocks containing the path endpoints, they contain at least three vertices. To see this, let (u_0, u_1) be the current bridge, with the last block closed at u_0. Let u_2 and u_3 be the two vertices to the right of u_1. Choose the edge, e, from the last block reaching furthest to the right. Now e is not incident to u_1, otherwise u_1 is a cut vertex. If e is incident to u_2 then there is a rightward edge e' out of u_1, otherwise u_2 is a cut vertex. We can choose e' for free by dropping the edge (u_1, u_2), see Figure 3-10. If e goes to u_3 or beyond then we are already done. Note that in our pre-processing we may have merged subtrees from below, i.e. some contracted vertices might consist of two or more vertices. In this case the arguments with regard to cut vertices may not apply; however, since these contracted vertices contain free edges, the proof is easier.

Since the block containing the right endpoint does not contribute an extra edge,

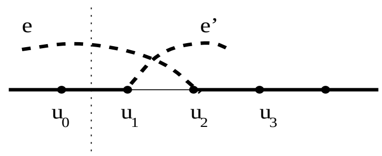

Figure 3-10: Paths.

we have added an extra $\lceil \frac{k}{3} \rceil$ edges. So to deal with T_0 we add, at most, an extra $\lceil \frac{k}{3} \rceil + 2$ edges. Recall, that associated with P we have two free edges, (i.e. $k+2$ edges in total), so these extra two edges are accounted for. Note that $\lceil \frac{k}{3} \rceil \leq \frac{1}{3}(k+2)$ so the associated blow-up factor for each path is $\frac{4}{3}$. $\square$

Now consider the case of small 2-connected components. First note that the only 2-edge connected components of at most 5 edges are cycles of at most 5 edges (for example 4-cycles with chords could not have been chosen in the solution to $U2$). Our aim is to show that in picking an extra back edge to 2-connect up such cycles we can remove some edge. In effect, we get the back edge for free and the charge to the node is just one.

Lemma 3.6.3. *For cycle nodes of size* 3*, the charge is one.*

Proof. We will label a back edge from the node going up the DFS tree by b. A back edge from below that is incident to the node will be labelled f. The deepest back edge from the descendants of the node will be labelled d. In all of our diagrams we use the following scheme. Solid edges were already selected at the point of examination of the component; dashed edges were not. Bold edges are selected by the algorithm after the examination; plain edges are not.
(I) Lower tree edges are all adjacent at the same vertex as the upper tree edge. Assume this vertex is v_1. Note that neither v_2 nor v_3 have incident edges going down the tree, otherwise the DFS algorithm would have chosen them as tree edges. Now v_1 is not a cut vertex, so there must be a back edge b from v_2 or v_3. Suppose it is incident to v_3; the other case is similar. We may add b and remove the edge (v_1, v_3) to 2-connect upwards the node. This is shown in Figure 3-11.I).

(II) The upper tree edge is incident to a vertex which has an adjacent vertex that is incident to a lower tree edge.

1. There is a back edge b from v_2 or v_3. We may assume it is from v_3. The other case is similar. Add b and remove the edge (v_1, v_3). This is shown in Figure 3-11.II)1).

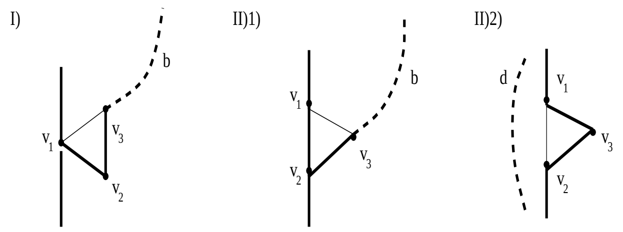

Figure 3-11: 3 cycles.

2. Otherwise, since v_1 is not a cut vertex, there is an edge d going beyond v_1. Add d and remove (v_1, v_2). See Figure 3-11.II)2). ☐

Lemma 3.6.4. *For cycle nodes of size* 4*, the charge is one.*

Proof.

(I) Lower tree edges are all adjacent at the same vertex as the upper tree edge.

1. There is a back edge b from v_2 or v_4. We may assume it is from v_4; the other case is similar. Add b and remove the edge (v_1, v_4). This is shown in Figure 3-12.I)1).

2. Otherwise, since v_1 is not a cut vertex, there is a back edge b incident to v_3. In addition, v_4 is not a beta-vertex. Therefore, as there is no edge from v_4 going down the tree (otherwise it would be a tree edge) there must be an edge between v_2 and v_4. Add b and (v_2, v_4). Remove (v_1, v_4) and (v_2, v_3). See Figure 3-12.I)2).

(II) The upper tree edge is incident to a vertex which has an adjacent vertex that is incident to a lower tree edge.

1. There is an edge d going beyond v_1. Add d and remove (v_1, v_2). See Figure 3-12.II)1).

2. There is a back edge b from v_2 or v_4. We may assume it is from v_4; the other case is similar. Add b and remove the edge (v_1, v_4). This is shown in Figure 3-12.II)2)).

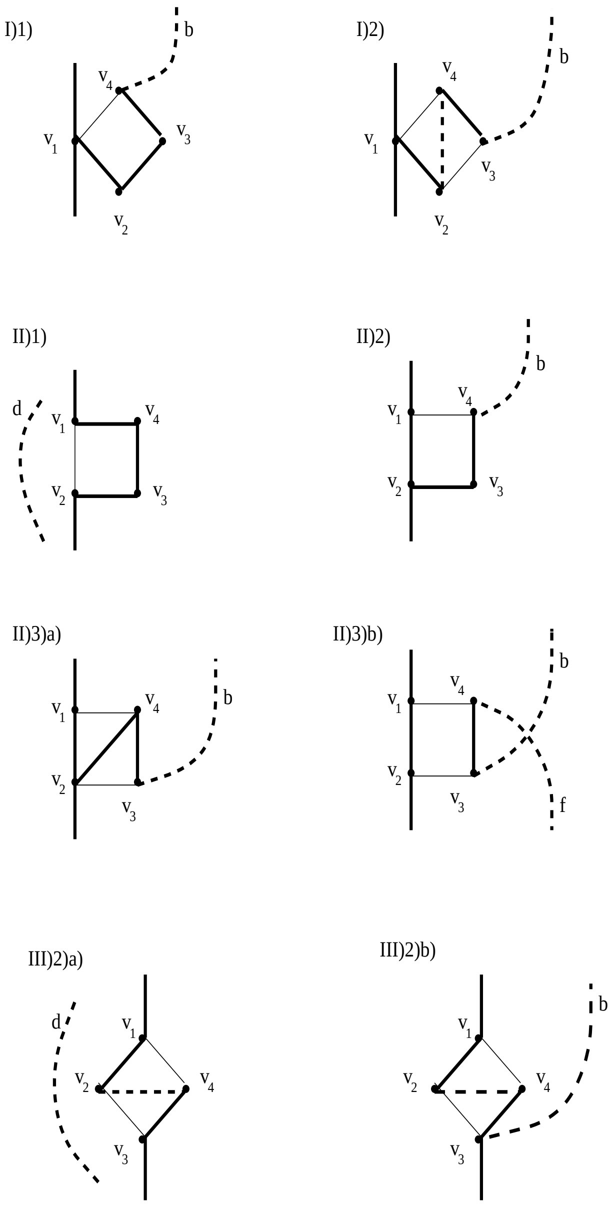

Figure 3-12: 4 cycles.

3. Otherwise, since v_1 is not a cut vertex, there is a back edge b incident to v_3. In addition, v_4 is not a beta-vertex. Therefore, either there is an edge between v_2 and v_4 or there must be an edge f from v_4 going down the tree. In the former case, Figure 3-12.II)3)a), add b and (v_2, v_4). Remove (v_1, v_4) and (v_2, v_3). In the latter case, figure 3-12b)iii)B), we have two possibilities. First suppose that the lower tree edge and f are incident to the same subtree below the 4-cycle. Then add b and f, remove

also (v_1, v_4) and (v_2, v_3). Otherwise the lower tree edge and f originate in separate subtrees with root nodes below the 4-cycle. It this case observe that the edge f is an upper tree edge for a lower component. As such we receive the edge for "free" and connecting up the 4-cycle is then easy. Similar situations to this second case, where there is such a type f edge, may arise again. In these situations we can always show that we obtain such a "free" edge. Thus, in the cases that follow, we will assume that all such type f edges belong to the same subtree as the lower tree edge.

(III) The upper tree edge is incident to a vertex which has an opposite vertex that is incident to a lower tree edge, but adjacent vertices are not incident to a lower tree edge.

1. There is a back edge b from v_2 or v_4. We may assume it is from v_4. Add b and remove the edge (v_1, v_4).

2. Otherwise, note that v_4 is not a beta-vertex. Therefore, as there is no edge from v_4 going down the tree (otherwise it would be a tree edge) there must be an edge between v_2 and v_4. In addition, since v_1 is not a cut vertex, there is either an edge d going beyond v_1 or there is a back edge b incident to v_3. We deal with both cases in a similar fashion, see Figures 3-12.III)2)a) and 3-12.III)2)b). Add (v_2, v_4) and either d or b. Remove (v_1, v_4) and (v_2, v_3). □

So by dealing with each node individually we were able to obtain a $\frac{7}{5}$ approximation factor. We will begin by continuing this localised approach for the situation in which nodes represent 5-cycles. It will turn out that such an approach is not quite sufficient in all instances. We will deal with these instances by pairing nodes and charging them together.

The following observations will also be useful. Suppose there is a cycle in G consisting of five edges, $(v_0, v_1), (v_1, v_2), (v_2, v_3), (v_3, v_4)$ and (v_4, v_0). Note that the vertices $v_0, \ldots, v_4$ may induce up to ten edges in G. Suppose that every optimal minimum 2-connected subgraph contains at least 4 of these ten possible arcs. Then we may choose the edges in the 5-cycle to be part of our solution, contract the cycle, and find an α-optimal subgraph on the resultant graph. The following lemma shows that this approach will be sufficient to produce an α-approximation factor, provided that $\alpha \geq \frac{5}{4}$. Recall, we use the notation G/C to denote the graph resulting from contracting a set of vertices $C \subseteq V$.

Lemma 3.6.5. *If* 5 *vertices induce* 4 *edges in every optimal solution then we may choose any* 5*-cycle through the vertices and consider the resultant sub-problem.*

Proof. Let the optimal solution have OPT edges, 4 of which are induced by $v_0, \ldots, v_4$. So $k = \text{OPT} - 4$ arcs are used on the edge set induced by $G' = G/\{v_0, \ldots, v_4\}$. Let OPT' be the size of the optimal solution to D'. Clearly $\text{OPT}' \leq k$. So if we can find an α-optimal solution to D' and then add the 5 arcs in the cycle we will have a solution to G. This solution will use at most $\alpha\,\text{OPT}' + 5 \leq \alpha k + 5 \leq \alpha(k+4) = \alpha\,\text{OPT}$ edges, provided that $\alpha \geq \frac{5}{4}$. Hence we will have a α-optimal solution to G. $\square$

This lemma will be useful in bounding the charge on a 5-cycle. As mentioned earlier, we cannot bound the charge on each 5-cycle, but we can bound the amortized charge.

Lemma 3.6.6. *The amortized charge on a 5-cycle is at most $\frac{5}{3}$.*

Proof. First, let's begin by attempting to deal with 5-cycles using our localised approach.

(I) Lower tree edges are all adjacent at the same vertex as the upper tree edge.

1. There is a back edge b from v_2 or v_5. We may assume it is from v_5. Add b and remove the edge (v_1, v_5).

2. Otherwise, since v_1 is not a cut vertex there is a back edge b from either v_3 or v_4. These cases are symmetric so assume it is from v_4. In addition, for the same reason, there must be an edge d. Also, v_5 is not a beta-vertex so one of the following situations must arise.

   (a) There is an edge (v_2, v_5). Add b and (v_2, v_5). Remove (v_1, v_2) and (v_4, v_5). See Figure 3-13.2)a).

   (b) Otherwise every optimal solution induces four edges within the 5-cycle, since two must be incident to v_2 and two incident to v_5. These four edges are clearly distinct so we may apply Lemma 3.6.5 and solve on the smaller subproblem.

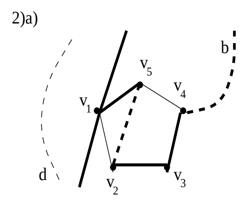

Figure 3-13: A 5 cycle

(II) The upper tree edge is incident to a vertex which has an adjacent vertex that is incident to a lower tree edge.

1. There is an edge d going beyond v_1. Add d and remove (v_1, v_2). See Figure 3-14.1).

2. There is a back edge b from v_2 or v_5. We may assume it is from v_5; the other case is similar. Add b and remove the edge (v_1, v_5).

3. There is a back edge b incident to v_4.

(a) There is an edge (v_2, v_5). Add b and (v_2, v_5). Remove (v_1, v_2) and (v_4, v_5). See Figure 3-14.3)a).

(b) There is an edge (v_3, v_5). Add b and (v_3, v_5). Remove (v_1, v_5) and (v_3, v_4). See Figure 3-14.3)b).

(c) There is a down edge f from v_5. Add b and f, remove (v_1, v_2) and (v_4, v_5). See Figure 3-14.3)c).

(d) Otherwise, since v_5 is not a beta-vertex, there must be a back edge b' from v_3. Add b and b', remove (v_3, v_4) and the upper tree edge incident to v_1. See Figure 3-14.3)d). Note that we have removed a tree edge. We will return to this case subsequently.

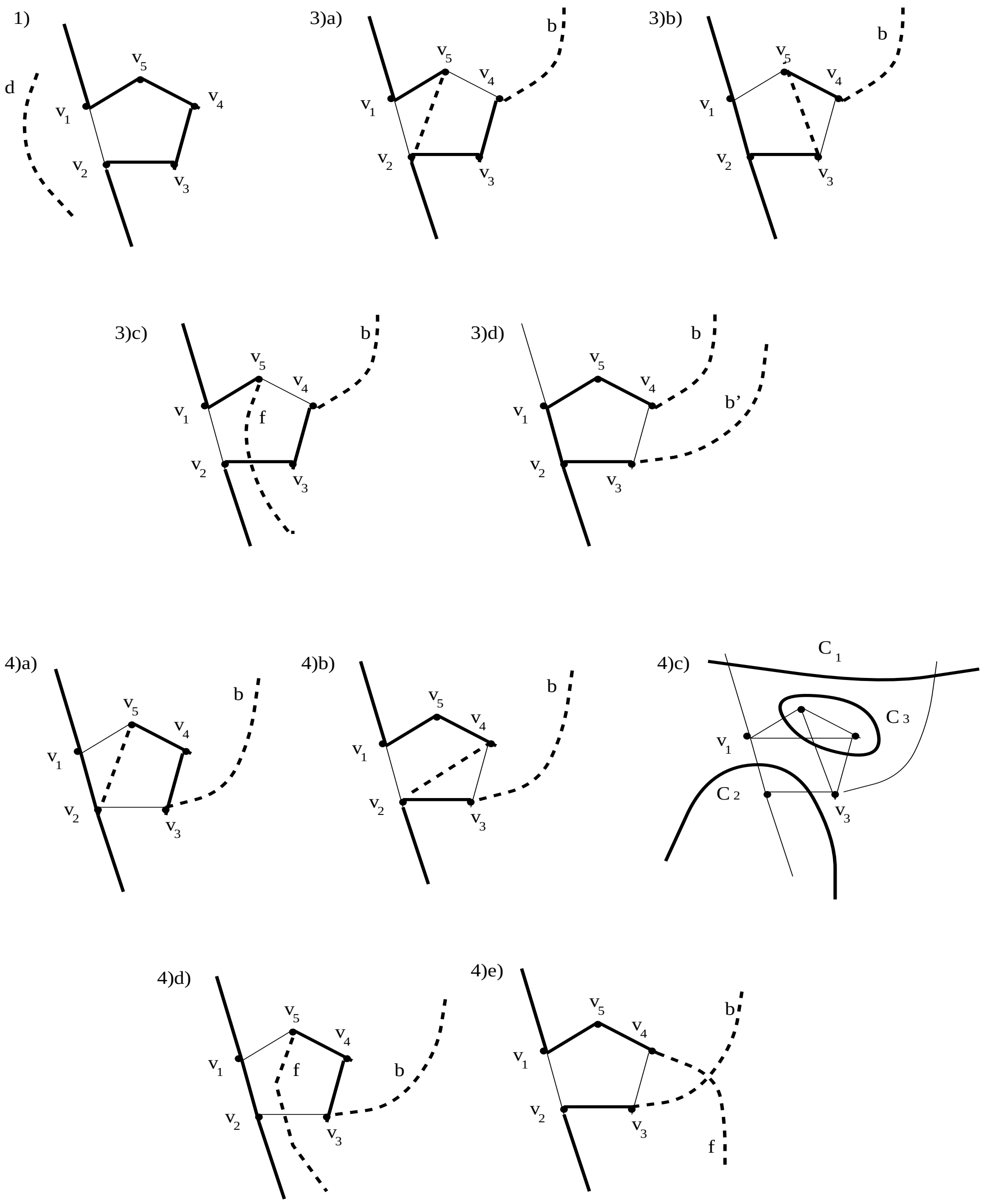

Figure 3-14: More 5 cycles.

4. There is a back edge b incident to v_3. Now, at least one of v_4 and v_5 has degree greater than two, otherwise we would have contracted the edge (v_4, v_5).

(a) There is an edge (v_2, v_5). Add b and (v_2, v_5). Remove (v_1, v_5) and (v_2, v_3). See Figure 3-14.4)a).

(b) There is an edge (v_2, v_4). Add b and (v_2, v_4). Remove (v_1, v_2) and (v_3, v_4). See Figure 3-14.4)b).

(c) There are edges (v_1, v_4) and/or (v_3, v_5). This situation is prohibited as we have a beta-pair. The removal of vertices v_1 and v_3 leaves three components, one of which C_3 is of size two. The three components are shown in Figure 3-14.4)c).

(d) There is a down edge f from v_5. Add b and f, remove (v_1, v_5) and (v_2, v_3). See Figure 3-14.4)d).

(e) There is a down edge f from v_4. Add b and f, remove (v_1, v_2) and (v_3, v_4). See Figure 3-14.4)e).

(III) The upper tree edge is incident to a vertex which has an opposite vertex that is incident to a lower tree edge, but adjacent vertices are not incident to a lower tree edge.

1. There is a back edge b from v_2 or v_5. We may assume it is from v_5; the other case is similar. Add b and remove the edge (v_1, v_5).

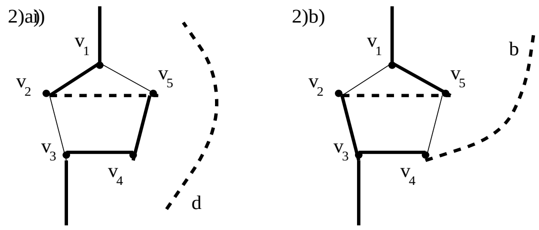

Figure 3-15: Even more 5 cycles.

2. There is an edge (v_2, v_5). Note that v_1 is not a cut vertex.

(a) There is an edge d from below the node that goes beyond v_1. Add d and Remove (v_1, v_5) and (v_2, v_3). See Figure 3-15.2)a).

(b) There is a back edge b from v_4 (the case concerning v_3 is similar). Add b and (v_2, v_5), remove (v_1, v_2) and (v_4, v_5). See Figure 3-15.2)b).

3. Otherwise the 5-cycle induces four arcs in any optimal solution and we may contract the cycle.

So we're not done yet. We have one troubling situation: when an upper tree edge was removed. See Figure 3-14.3)d). This tree edge is a lower tree edge for the component above the 5-cycle. Since our arguments for this component relied upon the positioning of its lower tree edges, our previous arguments may become invalid for that component. We will thus attempt to pair our troublesome component 5-cycle with the above component. We will then add edges to connect up both components at the same time. The components will be charged the extra edges as a pair.

So we need to reconsider the case of components that lie above these types of 5-cycle. We attempt to pair off these 5-cycles with the component above them. If the node above is a 2-connected component of size 4 or more then we are done as we have two spare edges to play with, since the 5-cycle is charged only one edge. So suppose the node represents a 3-cycle. Let e be the tree edge removed, and let b and b' be the two back edges chosen instead.

(I) The 3-cycle had lower tree edges all adjacent at the same vertex, y_1, as the upper tree edge. If either b and b' is incident to y_1 then our precious arguments for three cycles apply, since swapping e for b, say, does not change the structure of the lower tree edges for the 3-cycle. Neither b nor b' are incident to y_2 or y_3 as otherwise they would have been chosen as lower tree edges for the 3-cycle, not edge e. So both b and b' go beyond the 3-cycle. In this case, see Figure 3-16.I), since y_1 is not a cut vertex there is a back edge from either y_2 or y_3. Adding this edge allows us to remove one from the 3-cycle, and hence the net charge to the 3-cycle and the 5-cycle is just two edges.

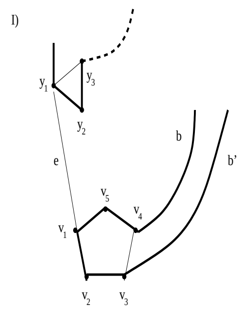

Figure 3-16: A pair of cycles.

(II) The 3-cycle had a lower tree edge and the upper tree edge incident at adjacent vertices.

1. Some edge from the 5-cycle goes beyond the 3-cycle. The case in which it is edge b is shown in Figure 3-17.II)1). Add back in edge e and (v_4, v_5), remove edges b' and (y_1, y_2). The net charge to the two components is then just two edges.

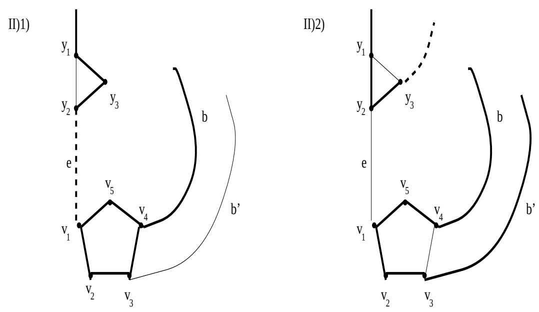

Figure 3-17: More pairs of cycles.

2. Both b and b' are incident to y_1. If either is incident to y_2 or y_3 then it has the same priority as e; hence the loss of e for b would have no effect on our previous proof method. Since y_1 is not a cut vertex either y_2 or y_3 has a back edge. If there is a back edge then we can drop an edge from the 3-cycle, Figure 3-17.II)2).

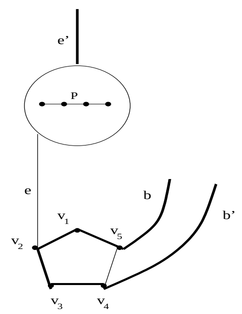

Figure 3-18: A cycle and path pair.

Finally suppose the node above the 5-cycle represents a path, P. Let e be the original tree edge charged to the 5-cycle. Suppose that in 2-connecting up the path another edge (in addition to e) from the 5-cycle or below is used. Then this edge also two connects up the 5-cycle which gets the edge for "free", giving a net charge of one to the 5-cycle. So no such edges are used. In which case adding b and b', whilst removing (v_4, v_5) has no effect on the argument concerning the path. Hence they may be considered separately. See Figure 3-18. □

Our result follows.

Theorem 3.6.7. *The algorithm is a $\frac{4}{3}$-approximation algorithm for $2EC$.* □

3.6.2 Comparison with the Subtour Relaxation

Recall the cut-covering LP relaxation for $2EC$.

$$\begin{array}{rcll} \min & \sum_{e \in E} x_e & & \\ \sum_{e \in \delta(S)} x_e & \geq & 2 & \forall S \subset V,\ S \neq \emptyset \\ x_e & \geq & 0 & \forall e \in E \end{array}$$

It may be noted that this relaxation is similar to the familiar *subtour relaxation* for the Travelling Salesman Problem (TSP). The subtour relaxation imposes the additional constraints that $\sum_{e \in \delta(\{v\})} x_e = 2,\ \forall v \in V$. For metric costs, though, there is an optimal solution to the LP relaxation which is also optimal for the subtour relaxation. A famous conjecture is that, for a graph with metric costs, $\text{OPT}(TSP) \leq \frac{4}{3}\,\text{OPT}(\text{LP})$. Whilst this conjecture remains unresolved, some progress has been made. For example, Goemans [19] considered the effects of adding classes of valid TSP inequalities to the subtour constraints. He showed that, for a range of classes, the optimal solution produced was always within a factor $\frac{4}{3}$ of the solution produced by just the subtour constraints. In addition, Monma et al [35] have shown a weaker result, that, given metric costs, $\text{OPT}(TSP) \leq \frac{4}{3}\,\text{OPT}(2EC)$. One implication of the conjecture is that for a 2-edge connected graph, $\text{OPT}(2EC) \leq \frac{4}{3}\,\text{OPT}(\text{LP})$. Carr and Ravi [4] provide a proof of the implication for the special case of half-integral solutions of the relaxation. Unfortunately, the implication (in its entirety) does not follow simply from our result. This is due to the fact that our lower bound, based upon integral degree 2 subgraphs, may be stronger in some situations than the bound offered by the linear program. The graph in Figure 3-19a) is an example. Figure 3-19b) shows the half integral optimal solution given by the LP. Here, edges given weight 1 are solid whilst edges of weight $\frac{1}{2}$ are dashed. This has total value 9, whereas the minimum integral degree 2 subgraph, Figure 3-19c), has value 10. A weaker implication of the TSP conjecture is that $2EC$ is approximable to within $\frac{4}{3}$, and our result does indeed verify this implication.

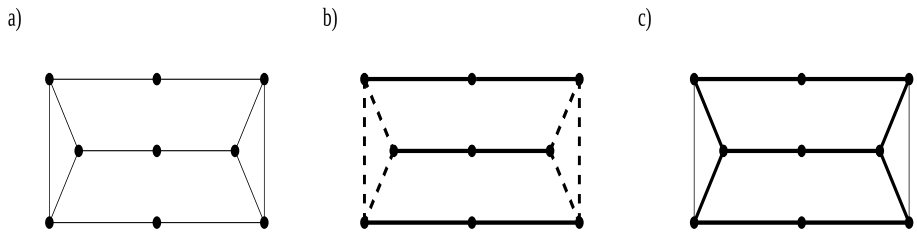

Figure 3-19: Subtour vs integral degree 2 subgraph.

3.7 Vertex Connectivity

Our approach for the problem $2VC$ is similar to that of $2EC$. Recall that our lower bound is a solution to the maximum cycle-path cover problem rather than a solution to $U2$. This ensures our initial components are either paths or 2-vertex connected components. Then, given our tree of components we attempt to 2-vertex connect up the graph using a limited number of extra edges. In doing so, though, we maintain the following property. Again, given the tree of components $\mathcal{T}$, denote by T_C the subtree in $\mathcal{T}$ rooted at a component C

PROPERTY 1. After dealing with component C we require that at least two edges cross from T_C to $G \setminus T_C$. In addition these two edges must be vertex-disjoint.

As before we charge additional edges to the components.

Lemma 3.7.1. *For path nodes the charge is at most $\lceil \frac{k}{3} \rceil + 2$, where k is the number of edges in the path.*

Proof. We take a similar approach to the method used for edge connectivity. Starting at the leftmost vertex in the path we add rightward edges (these edges may originate from components below the path as before). We attempt to split the path into blocks of at least three vertices with the property that we add only one edge per block.

Suppose the last edge added, a, is incident to u_0. We close the previous block at the vertex in the path before u_0. We show that the next block consists of at least three vertices. Take the edge, e, from the previous block that goes furthest rightwards. If e reaches u_3 then we are done. Suppose e reaches u_2. If there is an e' from u_1 going beyond u_2 then we get this edge for free by removing (u_1, u_2). See Figure 3-20a). Otherwise there is an e' originating at u_0 going beyond u_2. If there is an edge e'' from the previous block entering u_1 then we get e' for free. This is shown in Figure 3-20a). Suppose there is no such e''. Since u_1 is not a Beta-vertex we then have the possibilities shown in Figure 3-20c). In cases i) and ii) by consideration of the subtrees rooted at u_1 and u_0, respectively, we can remove a path edge as shown. We avoid case iii), by forcing an edge going down from u_1 to be a tree edge. $\square$

Lemma 3.7.2. *For cycle nodes of size 3, the charge is one.*

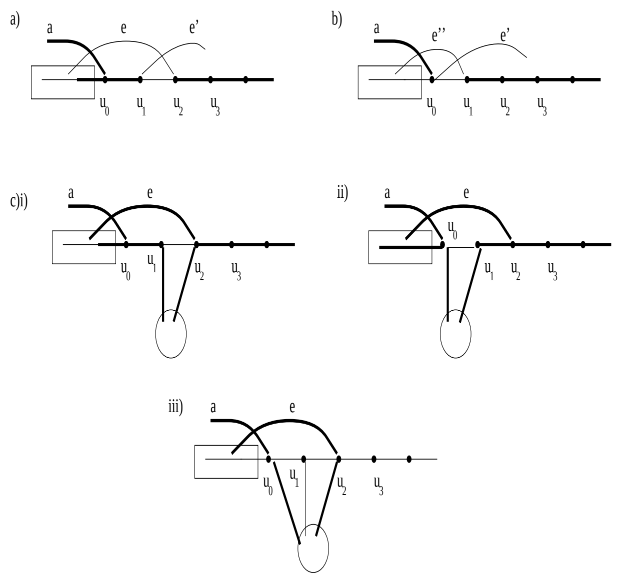

Figure 3-20: Paths and vertex connectivity.

Proof. We will use the following notation. Call the two back edges from the subtree below the 3-cycle a and a'. Let them originate at the vertices z_1 and z_2. We will presume that a is the lower tree edge for the 3-cycle. In contrast to the $2EC$ case our arguments for $2VC$ are based upon both of the edges from below, not just one of them. Let the upper tree edge from the 3-cycle be $e = (v_1, y_1)$. In the course of these proofs we will take y_2 to be a generic vertex in $T \setminus (T_C \cup \{y_1\})$. We will assume that any back edge leaving T_C, that is not incident to y_1, is incident to the same vertex y_2. This is the worst case scenario. The proofs are simpler if this does not happen to be the case. See Figure 3-21.

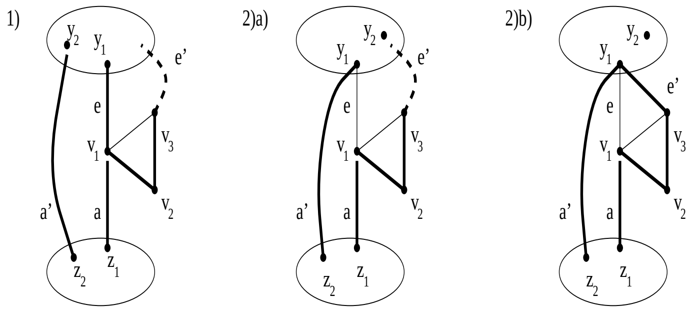

Figure 3-21: 3 cycles.

(I) Lower tree edges are all adjacent at the same vertex, v_1, as the upper tree edge. Notice that a' is not incident to the 3-cycle. It can not be incident to v_1 as a is. It is not incident to v_2 or v_3 as otherwise it would have been chosen as a lower tree edge not a.

1. Edge a' is incident to a vertex $y_2 \neq y_1$. Since v_1 is not a cut vertex there is a back edge e' from either v_2 or v_3. These cases are symmetric, the latter is shown in Figure 3-21.1). We may add e' and remove (v_1, v_3) for a net charge of one to the cycle.

2. Edge a' is incident to y_1. Again there must be a back edge e' from v_2 or v_3. If e' is incident to y_2, add e' and remove e and (v_1, v_3) for zero net charge to the cycle. See Figure 3-21.2)a). If e' is incident to y_1, add e' and remove e and (v_1, v_3). We still have one spare edge available. Choose any back edge that is incident to some $y_2 \neq y_1$. See Figure 3-21.2)b).

(II) The upper tree edge e is incident to v_1 and the lower tree edge a is incident to an adjacent vertex v_2.

1. If a' is incident to y_2 then we are already done. In fact, we may also remove the edge (v_1, v_2). See Figure 3-22.1)a). If a' is incident to y_1 then remove (v_1, v_2). We then have one extra edge to play with. This we add to connect the 3-cycle upwards to some vertex y_2. This is shown in Figure 3-22.1)b).

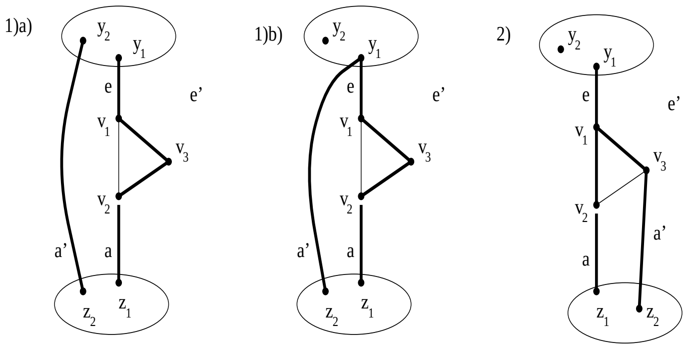

Figure 3-22: More 3 cycles.

2. Otherwise a' is incident to v_1 or v_3. The cases are similar and we deal with the latter. We may remove (v_2, v_3) and so have one extra edge available. See Figure 3-22.2). □

Lemma 3.7.3. *For cycle nodes of size* 4 *the charge is one.*

Proof.

(I) All lower tree edges are incident to v_1.

1. The back arc a' goes beyond the 4-cycle to $y_2 \neq y_1$. Note that v_1 is not a cut vertex so there is a back arc e' from one of v_2, v_3 or v_4.

(a) The back edge e' is incident to y_1. Add e' and remove e. The case where e' is incident to v_3 is shown in Figure 3-23.1)a). The charge to the 4-cycle is just one.

(b) The back edge e' goes to y_2 from either v_2 or v_4. Suppose it is from v_4. Add e' and remove (v_1, v_4). See Figure 3-23.1)b). The other case is similar.

(c) The back edge e' goes from v_3 to y_2. Since v_2 and v_4 are not Beta-vertices there must be an edge (v_2, v_4). Add e' and (v_2, v_4). Remove (v_1, v_4) and (v_2, v_3). See Figure 3-23.1)c).

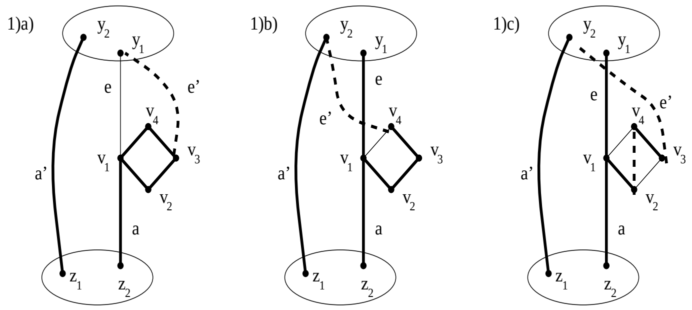

Figure 3-23: 4 cycles.

2. The back arc a' goes beyond the 4-cycle to y_1. This case is dealt with in a similar fashion to case 1.

(II) There is a lower tree edge a incident to v_3, but none incident to v_2 or v_4.

1. The back arc a' goes beyond the 4-cycle to $y_2 \neq y_1$. Then we are already 2-vertex connected beyond the 4-cycle. See Figure 3-24.1).

2. The back arc a' goes beyond the 4-cycle to y_1.

(a) There is a back edge e' from v_2 or v_4 to y_2. We show the latter case in Figure 3-24.2)a). Add e' and remove e and (v_3, v_4). This leads to zero net charge to the 4-cycle.

(b) There is a back edge e' from v_2 or v_4 to y_1. We show the latter case in Figure 3-24.2)b). Add e' and remove e and (v_3, v_4). We still have one edge to play with. We use this edge to connect us up to y_2.

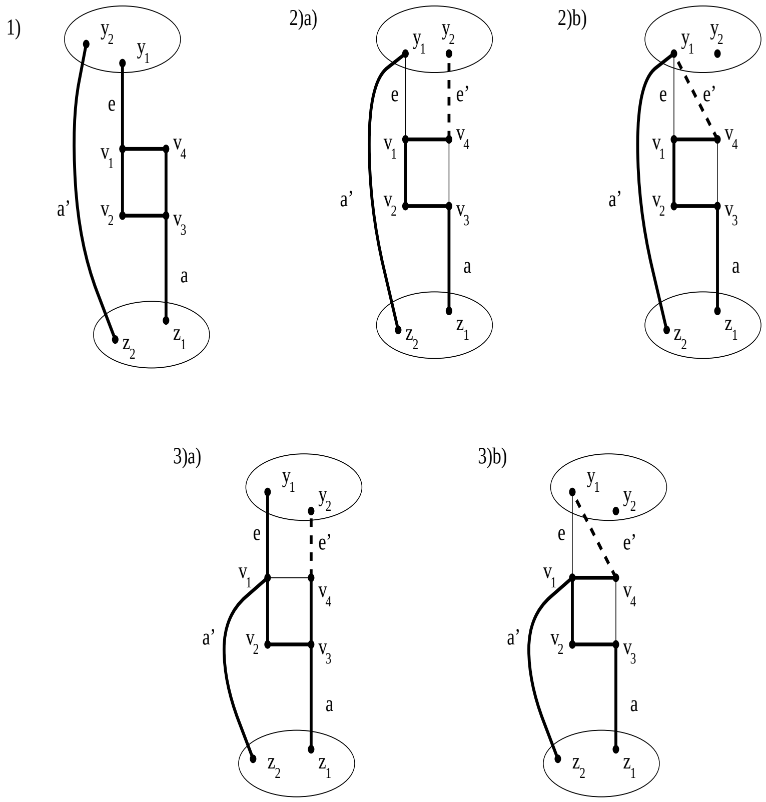

Figure 3-24: More 4 cycles.

3. The back edge a' is incident to v_1.

(a) There is a back edge e' from v_2 or v_4 to y_2. Both cases are similar. For the latter case add e' and remove (v_1, v_4). See Figure 3-24.3)a).

(b) There is a back edge e' from v_2 or v_4 to y_1. For the latter case add e', remove e and (v_1, v_4). We then still have one edge to play with. We pick any suitable back edge that connects us up to y_2. See Figure 3-24.3)b).

(c) Otherwise since v_2 and v_4 are not Beta-vertices there must be an edge (v_2, v_4). This we deal with as before.

(III) There is a lower tree edge a incident to v_2 or v_4. These cases are symmetric so we will deal with the former case.

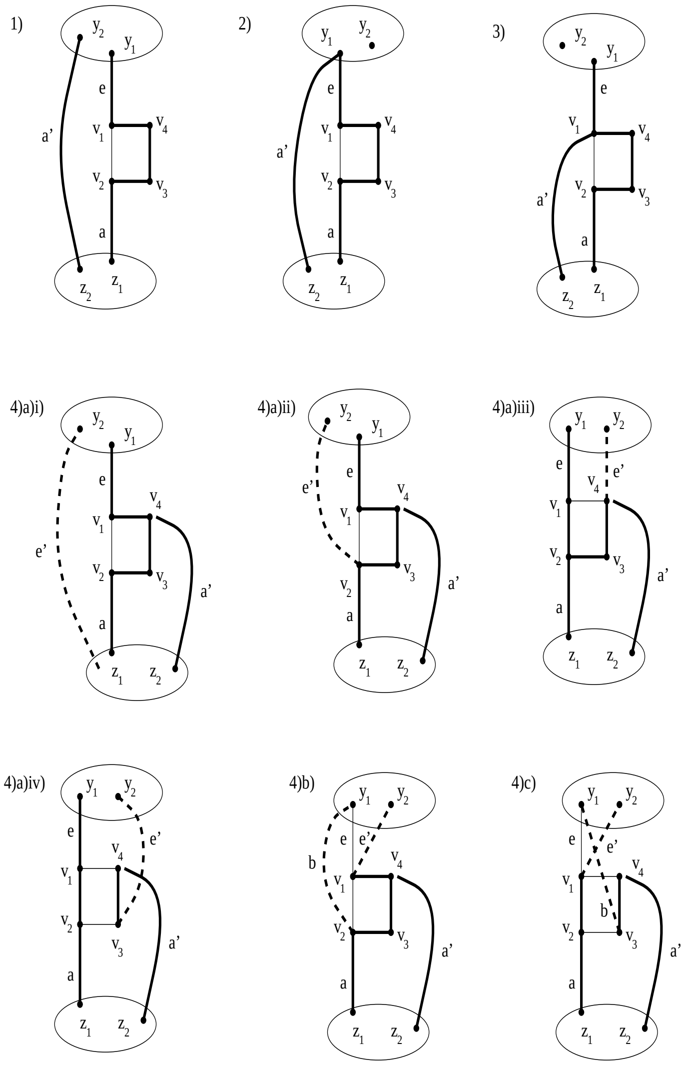

Figure 3-25: Even more 4 cycles.

1. The back arc a' goes beyond the 4-cycle to $y_2 \neq y_1$. Then we are already 2-vertex connected beyond the 4-cycle. See Figure 3-25.1). In fact we can also remove the edge (v_1, v_2).
2. The back arc a' goes beyond the 4-cycle to y_1. Remove (v_1, v_2). See Figure 3-25.2).

We may then add any appropriate back edge to connect us up to y_2.

3. The back arc a' is incident to v_1 or v_3. Here we may remove one edge from the 4-cycle and then choose any appropriate back edge to y_2. The case where a' is incident to v_1 is shown in Figure 3-25.3).

4. The back arc a' is incident to v_4. Consider how we may reach y_2.

(a) Let e' be a back edge into y_2. The cases where e' is from below the 4-cycle or where it originates from v_2, v_3 or v_4 are easily dealt with. See Figures 3-25.4)a)i)-iv).

(b) Then let e' be incident to v_1. Suppose there is a back edge b from v_2 to y_1. Then add b and e', remove e and (v_1, v_2). See Figure 3-25.4)b). The situation in which there is a back edge from v_4 to y_1 is similar.

(c) If there is a back edge b from v_3 to y_1 then add b and e'. Remove e, (v_1, v_4) and (v_2, v_3). See Figure 3-25.4)c).

(d) Otherwise there must be a back edge b into y_1 originating from below the 4-cycle. Here we add b and e' and remove e and (v_1, v_2).

$\square$

To deal with the case of 5-cycles we need the following lemma.

Lemma 3.7.4. *Suppose a beta-free graph G contains two vertices, v_2 and v_5, of degree 2 in a 5-cycle $C = \{v_1, \ldots, v_5\}$. Then G contains an optimal 2-vertex connected subgraph that does not contain the edge (v_3, v_4).*

Proof. Let S be a minimum 2-vertex connected subgraph. Suppose that S contains (v_3, v_4). Now S must contain two disjoint edges, e_1 and e_2, from C to $G \setminus C$. If e_1 and e_2 are incident to v_3 and v_4 then $S \setminus (v_3, v_4)$ is also 2-vertex connected, a contradiction. So, since v_2 and v_5 are of degree 2, without loss of generality e_1 is incident to v_1 and e_2 is incident to v_3. But G contains no Beta vertices and so there must be an edge e from v_4 to $G \setminus C$. Now $S \setminus (v_3, v_4) \cup e$ is also 2-vertex connected and the result follows. This is shown in Figure 3-26. $\square$

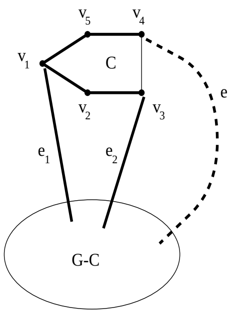

Figure 3-26: Easy to deal with 5 cycles.

Lemma 3.7.4 implies that if we find a pentagon that has two such degree two vertices then we may recurse on the graph $G \setminus (v_3, v_4)$. In fact we can generalise this result.

Lemma 3.7.5. *Let G be a beta-free graph that contains a 5-cycle $C = \{v_1, \ldots, v_5\}$. Suppose v_2 has degree 2 and that v_5 has no neighbours outside C. Then G contains an optimal 2-vertex connected subgraph that does not contain the edge (v_3, v_4).*

Proof. Similar to the proof of Lemma 3.7.4. □

Lemma 3.7.6. *For cycle nodes of size 5 the charge is one.*

Proof.

(I) All lower tree edges are incident to v_1. Note that v_1 is not a cut vertex.

1. The back edge a' is incident to y_2.

(a) There is a back edge e' into y_1 from v_2, v_3, v_4 or v_5. We add e' and drop e. An example is shown in Figure 3-27.1)a).

(b) There is a back edge e' into y_2 from v_2 or v_5. We illustrate the latter case in Figure 3-27.1)b). Add e' and remove (v_1, v_5).

(c) There is a back edge e' into y_2 from v_3. The case from v_4 is symmetric. The situation in which there is an edge (v_2, v_5) is shown in Figure 3-27.1)c). The situation in which there is an edge (v_2, v_4) is similar. Otherwise v_2 is of degree

2 and v_5 has only neighbours within the 5-cycle. Hence we may apply Lemma 3.7.5 and consider the graph $G \setminus (v_3, v_4)$.

2. The back edge a' is incident to y_1. This case is dealt with in a similar manner to the previous case.

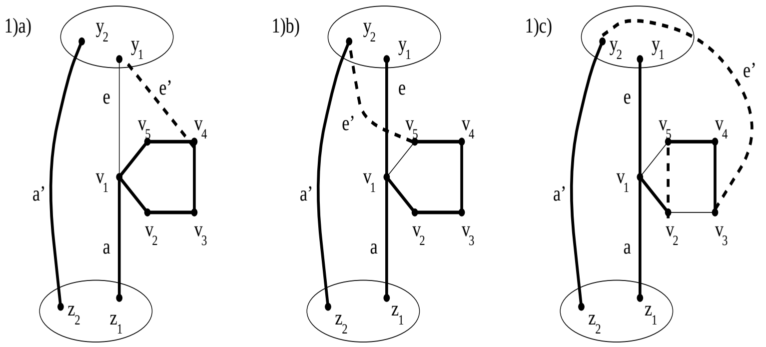

Figure 3-27: 5 cycles.

(II) There is a lower tree edge a incident to v_3 or v_4, but none incident to v_2 or v_4. These cases are symmetric so we will deal with the former case.

1. The back edge a' is incident to y_2. Then we are already done. See Figure 3-28a).

2. The back edge a' is incident to y_1.

(a) There is a back edge e' into y_2 from any of the vertices in the 5-cycle except v_3. Add e' and remove e. An example is given in Figure 3-28.2)a).

(b) There is a back edge into y_1 from either v_2 or v_4. Both cases are similar. The latter is shown in Figure 3-28.2)b). Add e' and remove e and (v_3, v_4). We then add an appropriate edge incident to y_2.

(c) The situations in which there are edges (v_2, v_5) or (v_2, v_4) are easy to deal with.

(d) So v_2 has degree 2. It follows that there must be an edge $e' = (v_5, y_1)$, otherwise we apply Lemma 3.7.5 and consider $G \setminus (v_3, v_4)$. In addition there must be a back edge b into v_4, otherwise we may apply Lemma 3.7.5 and consider $G \setminus (v_1, v_5)$. If b is incident to z_1 then we may add b and e' and remove a and (v_4, v_5) as shown in Figure 3-28.2)d). Hence we have one free edge left in which to connect us up to y_2.

(e) Finally consider the situation in which b is not incident to z_1. The back edge incident to y_2 must come from v_3 or below the 5-cycle. Both cases are similar. The former case is shown in Figure 3-28.2)e). Add b and b'. Remove e and (v_3, v_4).

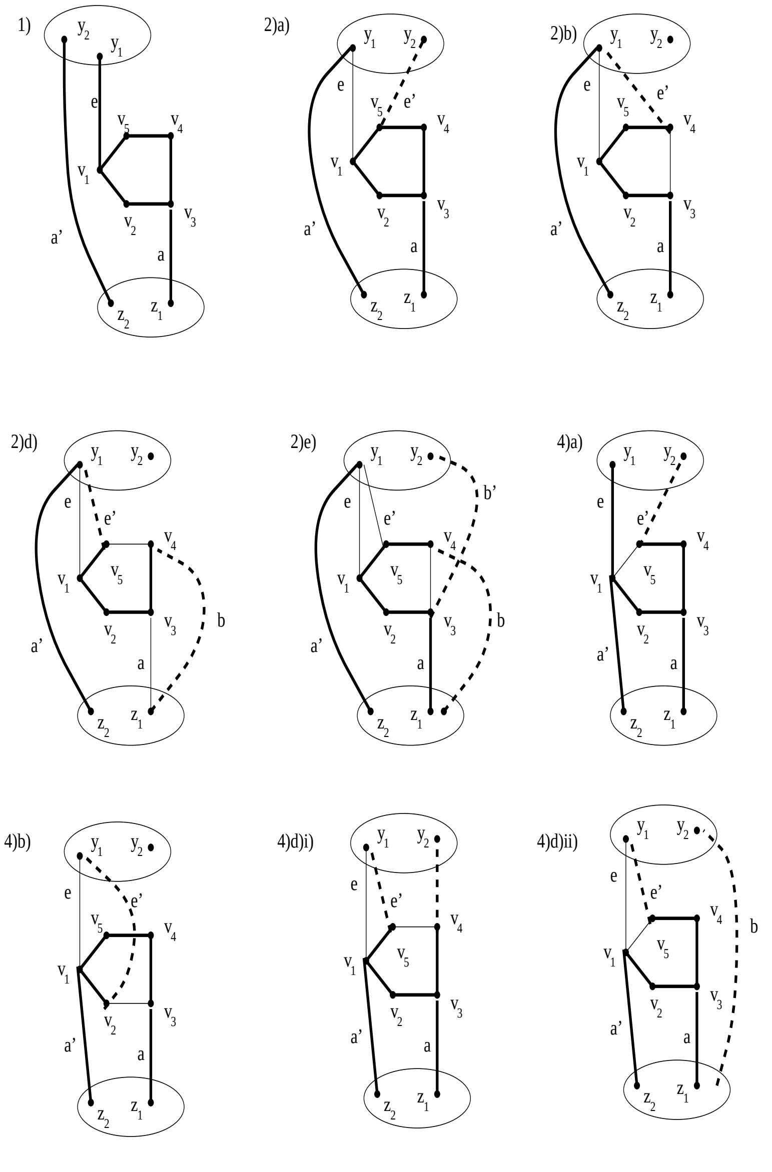

Figure 3-28: More 5 cycles.

3. The back edge a' is incident to v_4. This case follows from the observation that we

may remove the edge (v_3, v_4).

4. The back edge a' is incident to v_1.

   (a) There is a back edge e' to y_2 from either v_2 or v_5. We show the later case in Figure 3-28.4)a).

   (b) There is a back edge e' to y_1 from v_2. Add e', remove e and (v_2, v_3). Then add an appropriate back edge to y_2. See Figure 3-28.4)b).

   (c) The situations in which there are edges (v_2, v_5) or (v_2, v_4) are easy to deal with.

   (d) There is a back edge e' to y_1 from v_5. If there is also a back edge b from v_4 to y_2 then we proceed as in Figure 3-28.4)d)i)). Suppose then that there is a back edge b to y_2 from v_1, v_2 or from below the 5-cycle. These cases are similar. We illustrate the last of these cases in Figure 3-28.4)d)ii).

   (e) Suppose then the only back edge b into y_2 comes from v_3. Here the situation is more complex. Notice first that removing v_3 and y_1 disconnects the graph into two components C_1 and C_2. This is shown in Figure 3-29.4)e). Notice that this implies that in any optimal solution there is a path from y_1 to v_3 whose internal vertices are all in C_1. In addition these internal vertices can only be adjacent to other vertices on the path. Finally, note that, since v_2 is of degree 2 any solution uses the edges (v_1, v_2) and (v_2, v_3). This forces some path of the type shown in

   Figure 3-29 to be in any optimal solution. Observe that this implies, by a similar argument to that given in Lemma 3.7.4, that there is some optimal solution which does not contain the edge (v_1, y_1). Hence we may consider the graph $G \setminus (v_1, y_1)$.

   (f) Otherwise we apply Lemma 3.7.5 and consider $G \setminus (v_3, v_4)$.

(III) There is a lower tree edge a incident to v_2 or v_5. These cases are symmetric so we will deal with the former case.

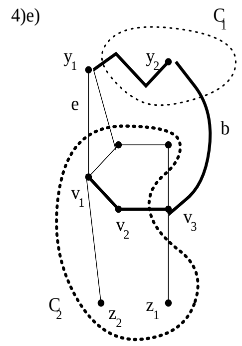

Figure 3-29: A tricky 5 cycle.

1. The back edge a' is incident to y_2. Then we are already done. See Figure 3-30.1). We may also remove the edge (v_1, v_2).

2. The back edge a' is incident to y_1. Remove the edge (v_1, v_2). We have one spare edge which we use to connect up to y_2. See Figure 3-30.2).

3. The back edge a' is incident to v_1 or v_3. In either case we may remove one edge from the cycle which gives us a spare edge to play with.

4. The back edge a' is incident to v_5. By consideration of the back edges incident to y_2 we get to remove either (v_1, v_2) or (v_1, v_5).

5. The back edge a' is incident to v_4.

   (a) The cases in which there is a back edge e' to y_2 from v_2, v_3 or v_5 are all easy to deal with.

   (b) If there is a back edge e' from below the 5-cycle to y_2 we then may add e' and remove (v_1, v_2).

   (c) Suppose there is an edge $e' = (v_1, y_2)$ along with a back edge into y_1 from any of v_2, v_3 or v_5. These are dealt with in a similar manner to those in case 5a). Similarly if there is a back edge into y_1 from below the 5-cycle.

   (d) Hence the only vertex incident to y_2 is v_4. Again the situations where there is a back edge into y_1 from any of v_2, v_3 or v_5 are easy to deal with.

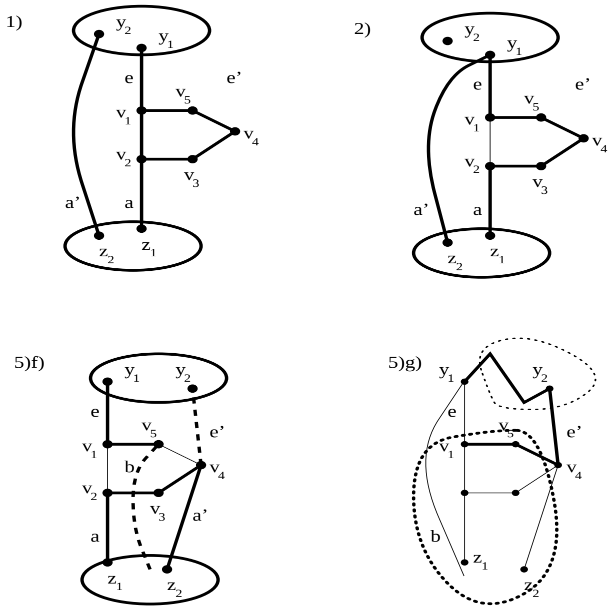

Figure 3-30: Even more 5 cycles.

(e) We use the usual methods to deal with the edges $(v_1, v_3), (v_3, v_5)$ or (v_2, v_5), if they are present.

(f) Suppose there is a back edge b into v_5. Then add b and e', remove (v_1, v_2) and (v_4, v_5). This is shown in Figure 3-30.5)f)).

(g) There must then be a back edge b into y_1 since v_5 is not a Beta-vertex. In addition, note that v_5 is of degree 2. Observe also that the removal of y_1 and v_4 disconnects the graph. It follows, by a similar argument to the one given in case II)4)e), that we may consider the graph $G \setminus (v_1, y_1)$. See 3-30.5)g)).

□

Theorem 3.7.7. *The algorithm is a $\frac{4}{3}$-approximation algorithm for 2VC.* □

3.8 Strong Connectivity

3.8.1 Tools and Operations

Let a *digon* be a directed cycle consisting of just two arcs, $a = (u_1, u_2)$ and its *complement* $\bar{a} = (u_2, u_1)$. We have the following lemma.

Lemma 3.8.1. *Let G be a strongly connected graph containing at least three vertices. Then, if G contains no cut vertices, there exists a minimum strongly connected subgraph which does not contain any digons.*

Proof. Take a minimum strongly connected subgraph, S, containing the fewest possible number of digons. Suppose it contains a digon $a = (u_1, u_2)$ and $\bar{a} = (u_2, u_1)$. Notice there is no directed path from u_1 to u_2 in $S \setminus \{a\}$. Otherwise a is superfluous and S is not minimal. Similarly there is no directed path from u_2 to u_1 in $S \setminus \{\bar{a}\}$. So, consider an out-tree, T_1, rooted at u_1 in $S \setminus \{a\}$. Clearly, $u_2 \notin T_1$. Similarly, $u_1 \notin T_2$, where T_2 is an out-tree rooted at u_2 in $S \setminus \{\bar{a}\}$.

Now $A = T_1 \cap T_2$ is empty. Otherwise A induces a cut $\delta(A)$ in which the only arcs in $S \cap \delta(A)$ are all directed into A. Hence S is not a strongly connected arc set. Similarly $V = T_1 \cup T_2$. Suppose $B = V \setminus (T_1 \cup T_2)$ is non-empty. Then $S \cap \delta(B)$ consists only of arcs directed out of B. Again, this contradicts the fact that S is strongly connected. So there are no arcs between $T_1 \setminus \{v_1\}$ and $T_2 \setminus \{v_2\}$. It follows that both $G \setminus T_1$ and $G \setminus T_2$ are strongly connected when we consider S restricted to $G \setminus T_1$ and S restricted to $G \setminus T_2$, respectively. Now a and $\bar{a}$ are not the only arcs in D connecting one vertex in T_1 and another in T_2. Otherwise u_1 and/or u_2 would be cut vertices. So let a' be an arc crossing from T_1 to T_2. Without loss of generality we may assume it is directed out of T_1 and into T_2. In addition, it is incident to exactly one of u_1 or u_2. Then $S \setminus \{a\} \cup \{a'\}$ is a minimum strongly connected subgraph with one less digon than S, a contradiction. $\square$

Corollary 3.8.2. *If an arc a is in an optimal solution to G then we may consider the graph $G \setminus \{\bar{a}\}$ instead.*

Proof. If a is in an optimal solution then by Lemma 3.8.1 its complement $\bar{a}$ (if it is present in G) is not in some optimal solution. Hence we may solve the problem on the smaller graph $G \setminus \{\bar{a}\}$. □

We say that an arc a that is not present in some optimal solution is *redundant*. If ever we find a redundant arc then we may instead consider the graph $G \setminus \{a\}$. For example, if an arc is in a digon and we can show that its complement is forced to be in every optimal solution then by Corollary 3.8.2 the arc is redundant. More generally, redundant arcs may also occur in the presence of forced paths. We say that directed path, P, consisting of the vertices $v_0, v_1, \ldots v_r$, is *forced* if it is in some optimal solution. In addition, suppose that we have a vertex u and arcs (u, v_0) and (u, v_r). Choosing the arc (u, v_0) will always be preferable to (u, v_r). This follows as, given that P is present, (u, v_0) also creates a path from u to v_k as well as from u to v_0. Hence (u, v_r) is redundant and we may consider the smaller graph $G \setminus (u, v_r)$. Similarly, given such a forced path P and arcs (v_0, u) and (v_r, u), we see that (v_0, u) is redundant.

Now consider a cycle in G consisting of three arcs, $(v_0, v_1), (v_1, v_2)$ and (v_2, v_0). First, note that the nodes v_0, v_1 and v_2 may induce up to six arcs in G, the three cycle arcs and their complements. Suppose there is an optimal minimum strongly connected subgraph containing at least 2 of these six possible arcs. Then we may choose the three cycle arcs, contract the cycle, and find a $\frac{3}{2}$-optimal subgraph on the resultant graph. The following lemma shows that this approach will be sufficient to produce a $\frac{3}{2}$-approximation factor.

Lemma 3.8.3. *If three nodes induce two arcs is some optimal solution then we may choose any* 3*-cycle through the nodes and consider the resultant sub-problem.*

Proof. Let the optimal solution have OPT arcs, two of which are induced by v_0, v_1 and v_2. So $k = \text{OPT} - 2$ arcs are used on the arc set induced by $G' = G/\{v_0, v_1, v_2\}$, the graph formed by contracting the 3-cycle. Let OPT' be the size of the optimal solution to G'. Clearly $\text{OPT}' \leq k$. So if we can find $\frac{3}{2}$-optimal solution to G' and then add the three arcs in the cycle we will have a solution to G. This solution will use at most

$\frac{3}{2}\textsc{opt}' + 3 \leq \frac{3}{2}k + 3 = \frac{3}{2}(k+2) = \frac{3}{2}\textsc{opt}$ arcs. Hence we have a $\frac{3}{2}$-optimal solution to G. □

We will apply Lemma 3.8.3 in the following manner. If we ever find a 3-cycle that induces two arcs in some optimal solution then we will consider the graph resultant from contracting the cycle.

There is one other operation that we will use in our algorithm, that of *component merging*. Suppose we find a cycle $\mathcal{C}'$ in $\mathcal{G}$ that strongly connects together several components (which do not include the root component). In such circumstances we may treat the group formed by such additions as one mega-component. We may define a new component graph $\mathcal{G}'$ on the same original components, with the exception that we will use the mega-component rather than its many base-components. This process of refining our component graph will be termed component merging. We will then grow a new tree $\mathcal{T}'$ on $\mathcal{G}'$ and begin again. Here we must bear in mind that we must also charge the arc set $\mathcal{C}'$ to the the mega-component, on top of any other arcs that may subsequently be charged to it.

Note that contractions, deletions and component mergings can only be carried out a polynomial number of times. As such, repeatedly carrying out these operations (and solving on the resultant graphs) will not affect the polynomial nature of our algorithm.

3.8.2 The Local Improvement Method

We are now ready to describe how to implement our version of the cycle contraction algorithm. Recall that we wish to efficiently contract a cycle (or circuit), in the component graph, through the root and the target component, and repeat. Given such a cycle, $\mathcal{C}$ in $\mathcal{G}$, suppose that the contraction of $\mathcal{C}$ results in the contraction of a set of vertices $V' \subseteq V$. In order to show our algorithm has the desired performance, it is sufficient to prove the following theorem.

Theorem 3.8.4. *Suppose that the algorithm calls for the contraction of a cycle $\mathcal{C}$. Let this induce the contraction of the vertex set V'. If V' induces r arcs in our final* *solution then V' induces at least $\frac{2}{3}r$ arcs in the lower bound.*

Proof. Our proof of Theorem 3.8.4 will be split into several parts. We consider first the case in where the cycle, $\mathcal{C}$ in $\mathcal{G}$, that we contract passes only through strongly connected components. We then partition our proof depending upon whether the the target component contains at least 4 arcs or is a digon or 3-cycle. Finally we examine the situation in which the root component, target component or other component contained in $\mathcal{C}$ is a path.

So we wish to contract a cycle through the root and the target component. In addition, we desire the property that the vertices contracted in this process induce enough arcs in the lower bound to cover, with our $\frac{3}{2}$ performance guarantee, the number of arcs they induce in the final solution. We have seen that this is easy when the target component is strongly connected and contains at least four arcs, since this gives us two extra arcs to play with. We now show that Theorem 3.8.4 holds for the other types of target component.

DIGONS. Suppose the target component is a digon D. We have seen that it is too costly to charge 2 extra arcs to this component. Recall that we begin with the assumption that all of the components which we encounter in this section are strongly connected. The situation in which some of the components may be paths will be dealt with subsequently. We begin with some easy situations.

a) There is a back arc b emanating from the subtree T_1. Suppose b is incident to v_1, see Figure 3-31a)i). In this case we may add the arc b and remove the arc (v_1, v_0). Notice that the removal of the arc (v_1, v_0) implies that the net charge to the digon is just one arc, giving our desired $\frac{3}{2}$ factor. Suppose, instead, that b emanates from a component C in $T_1 - \{v_1\}$, where T_1 is the subtree in $\mathcal{T}$ rooted at v_1. See Figure 3-31a)ii). In this case we may add the arc b and remove the arc (v_1, v_0). Along with the appropriate tree arcs this creates a cycle $\mathcal{C}$ to contract. This cycle also connects into the root the all components between D and C in the subtree T_1. Note each such component receives a net charge of one arc. Since these components contain at least two arcs, we have at worst a $\frac{3}{2}$ blow-up factor.

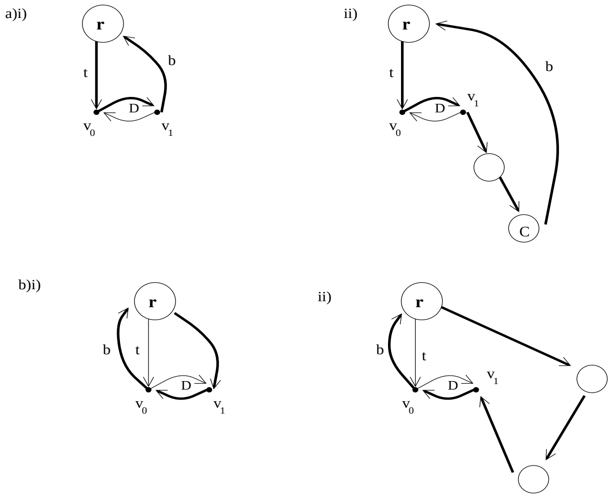

Figure 3-31: A target digon.

b) There is a back arc b emanating from v_0. Now since v_0 is not a cut vertex there must be a path from r to v_1 that does not pass through v_0. This path may consist of a single arc from r to v_1, see Figure 3-31a)i). The general case is shown in Figure 3-31a)ii). Combining this path with the arc b allows us to drop the arc (v_0, v_1) and still create a cycle $\mathcal{C}$. Again the net charge to the digon (and any other component in the cycle) is just one.

Observe that Figures 3-31a)ii) and b)ii) generalise Figures 3-31a)ii) and b)ii). In particular, Figure 3-31a)ii) contains a path from v_1 to the root passing through various components of $\mathcal{G}$. In Figure 3-31a)i) this path is a trivial path consisting of a single arc from v_1 to the root. The situation with regards to Figures 3-31b)i) and ii) is similar, with the path, this time, being directed from the root to v_1. Note that (since we are assuming that each node in $\mathcal{G}$ is strongly connected and contains at least two arcs) if non-trivial paths are found then the charge associated with each additional component is just one arc, giving a blow-up factor of at most $\frac{3}{2}$. Observe that the general case of non-trivial paths is just a combination of the trivial path case and a path through these additional components. Hence, it will be the case that the trivial path case provides a worst case analysis. If we can show that the blow-up associated with the trivial path case is at most $\frac{3}{2}$, then it will also follow for the general case.

As a result, unless otherwise stated, we will make this worst case assumption that connections between specific vertices will be via a single arc rather than via a path in the component graph.

c) The only possible situation remaining is the case of a back arc b going from $T_0 \setminus \{v_0\}$ to the root component. Again, for a worse case scenario we may assume it emanates from C_0 a child component of v_0 in T_0. In such circumstances we will be forced to contract into the root several components together. We will find a circuit, rather than a cycle, to contract. In particular we will contract two cycles (one of which does not contain the root component) and argue that the process can be done efficiently.

Now suppose there is a back arc path from v_1 to v_0, or vice versa, via C_1. In the worst case this path consists of just two arcs. See Figure 3-33a). So by adding both these path arcs whilst removing (v_0, v_1) we connect up, into the root component, the digon and C_0 and C_1. Thus if C_0 and C_1 both contain at least three arcs then we use 12 arcs in total for the three components. Since these three components contain 8 arcs this gives a $\frac{3}{2}$ blow-up. Notice that we do indeed contract a circuit which consists of two cycles that intersect at D.

What, though, if C_0 or C_1 are themselves digons? We will eliminate digons via a recursive process. In order to show how this will be done we first introduce some more notation. Given a digon D_j, label the vertices of the digon as follows. Let the $v_{ja} \in D_j$ be incident to arcs that currently connect T_{D_j} to $G \setminus T_{D_j}$. Let the v_{jb} be the other vertex in D_j. We say that a child component C of D_j is a *rung* if there is a path in the subtree T_{D_j} from v_{ja} to v_{jb}, or vice versa, that passes through C. In the worst case this path will contain just the single component C. In this case we will label the path arcs p_1 and p_2, see Figure 3-32a) and b). Note that the use of such paths allows the removal of either the arc (v_{ja}, v_{jb}) or the arc (v_{jb}, v_{ja}). Hence the path forms a 3-cycle through the digon D_j, and its three arcs will be charged to the digon. This observation will be used in a form of recursion that enables us to "ignore" digons.

In addition, we say that a rung C is a *base rung* if either C contains at least 3 arcs or if C is a digon D_{j+1} and the path arcs are incident to different vertices in D_{j+1}. Notice that in the latter case there is a 4-cycle through the vertices of D_j and D_{j+1}.

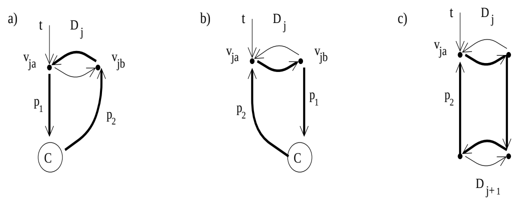

Figure 3-32: Rungs.

One such possibility is shown in Figure 3-32c). Should we find a digon D_{j+1} that is a base rung and a child of another digon D_j we will then perform our component merging operation. We replace these two digon components in the component graph $\mathcal{G}$ by the 4-cycle that passes through them.

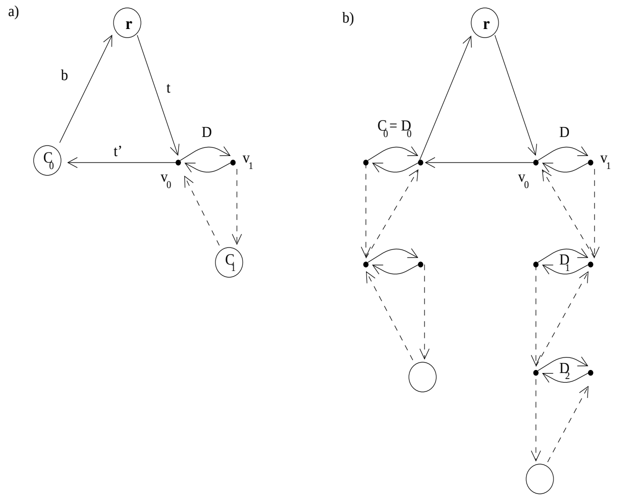

Figure 3-33: Growing a Pair of Digon Ladders

Thus in the previous case, where we assumed that C_0 and C_1 both contain at least three arcs, the component C_1 is a base rung child of the digon D (and C_0, whilst not technically a base rung, plays a similar role).

We now illustrate the basic recursive argument that will be applied on a specific example.. Let C_1 be a digon D_1. Hence C_1 is a rung. Suppose also that it is not a base rung. Suppose there are arcs $p_1 = (v_1, v_{1a})$ and $p_2 = (v_{1a}, v_0)$. The reader may observe that were we to contract the 3-cycle formed by p_1, p_2 and (v_0, v_1) then we would essentially be in the same situation as in Figure 3-33a) but with the digon D

replaced by the digon D_1. Suppose then that D_1 has a child component C_2 that is a base rung. Now if C_2 contains at least three arcs then the blow-up factor of merging the circuit illustrated is again at most $\frac{3}{2}$. If $C_2 = D_2$ is a digon that is a base rung then there is a 4-cycle through D_1 and D_2 and we consider a new component graph $\mathcal{G}'$ with the 4-cycle as a node. There is one final case to consider. If the digon D_2 has a back arc that goes beyond D_1 and into D then it is easy to verify that such a circumstance can only be beneficial.

We now return to the general case and formalise our arguments. We will successively eliminate digons, until we reach a base rung. Assume that C_0 is a digon and consider the picture shown in Figure 3-33b).

We take a child component of D that is a rung (if there are many such child components we may choose the one with the highest priority in $\mathcal{T}$). If this component is a base rung then we stop. Otherwise, if this rung is a digon D_1 that is not a base rung, we recurse on the children of D_1. This process forms a "ladder of digons", $D, D_1, D_2, \ldots D_p$. Similarly we grow another ladder of digons, by the same method, beneath D_0. Such ladders are shown in Figure 3-33b). If both of these processes terminate successfully then we have a guarantee of the desired blow-up factor.

So suppose one of the ladders does not terminate with a base rung. Consider the last digon D_p in the ladder. Then either D_p has no children or it has no child components that are rungs. Consider the former case. Now a back arc b must emanate from v_{pb} otherwise the arc (v_{pb}, v_{pa}) is forced and we may consider the graph $G \backslash (v_{pa}, v_{pb})$. In addition, the head of b must also be incident to the tail of p_1 otherwise D_p is a base rung. Again its is only beneficial if b goes beyond D_{p-1}. Now there must be a down arc d entering into v_{pb}, otherwise the arc (v_{pa}, v_{pb}) is forced and we may consider the graph $G \setminus (v_{pb}, v_{pa})$. In the worst case d emanates from the digon D_{p-1}. Here we may construct a 4 cycle through the vertices of D_{p-1} and D_p. This is shown in Figure 3-34a) and b). We then apply the merging components method, using the 4-cycle instead of the two digons.

Similar arguments apply in the latter case where D_p has child components but none are rungs. Hence the ladder growing process terminates and we are done.

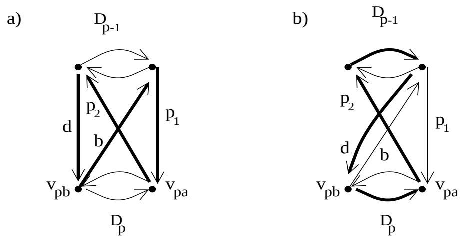

Figure 3-34: Merging digons.

Finally, note that any component that is incident to an arc a that emanates from the root component may be chosen as a target node. This follows as we may force a to be chosen as the first arc in $\mathcal{T}$. In fact, any component that is adjacent to the root (via an in-arc or an out-arc) may be chosen as the target node as we may as well have chosen $\mathcal{T}$ to be an in-tree rooted at r. By symmetry, the arguments that follow with respect to an out-tree $\mathcal{T}$ would also apply to an in-tree. This observation allows us to now assume that no components adjacent to the root component are digons (or, in fact, strongly connected with at least 4 arcs).

3-Cycles. Let our target component be a 3-cycle. Again, we will assume that none of the components which we encounter in this section are paths. In addition, should we encounter a component that represents a digon then the "ladder of digons" argument may be applied in the same manner as before. This observation allows us to assume that all of the components which we encounter are strongly connected and contain at least three arcs.

We may assume that all cycles in component graph $\mathcal{G}$ that pass through the root component and the target component are digons. Suppose otherwise and such a cycle passes through three or more components. The two components adjacent to the root component contain at least three arcs each. The blow-up factor is then at most $\frac{3}{2}$. This factor occurs for the case in which there are three components (i.e. three arcs are added) and the non-root components are both 3-cycles. This is illustrated in Figure 3-35i).

Let the 3-cycle consist of the nodes v_0, v_1 and v_2 with the tree arc, from the root

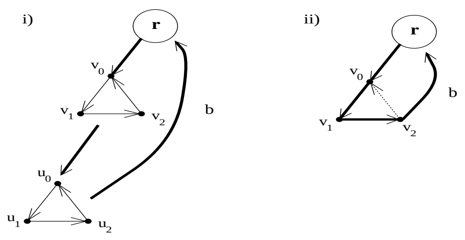

Figure 3-35: Some 3-cycles.

to the cycle, being incident to v_0. By the arguments given above, there must be a back arc b emanating from the 3-cycle. If b is incident to v_2 then we can choose b and remove the arc (v_2, v_0). This situation is shown in Figure 3-35ii). Thus, we need only 4 arcs in total and they can all be accounted for by the 3 arcs in the cycle. Hence, since v_0 is not a cut vertex there must be a back arc b emanating from v_1. This implies that there is no arc from the root component to v_2, otherwise we may omit the arc (v_1, v_2) in a similar manner to that of Figure 3-35ii).

There is another relatively straightforward situation. Suppose a tree arc t emanates from the node v_2 and that it leads to a child component C_2. If for such a subtree, T_2, there emanates a back arc b' into v_0 then we can connect up the 3-cycle and component C_2 for a cost of 3 arcs. This follows as in choosing d and b' we may remove arc (v_2, v_0). Since C_2 contains at least 3 arcs this gives the desired blow-up factor. Note that if b' emanates from a component lower than C_2 in T_2 then this only helps us, as we connect up any additional component for a charge of just one arc. This situation is shown in Figure 3-36i). The corresponding cases for subtrees T_1 and T_0, with child components C_1 and C_0, are shown in Figures 3-36ii) and iii).

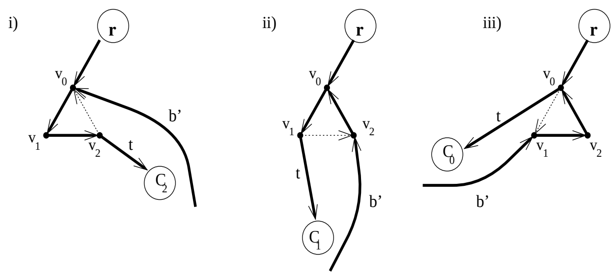

Figure 3-36: More 3-cycles.

Now consider the remaining possibilities. Recall there is a back arc b out of v_1. By the ordering given by DFS method, there are no arcs from vertices in T_0 to vertices in either T_1 or T_2. There are also no arcs from vertices in T_2 to vertices in T_1.

a) There is a back arc b' coming into v_2 from T_0. Choosing b' allows us to drop the arc (v_1, v_2). The net charge is $4 - 1 = 3$ arcs which we charge to the cycle and C_0. See Figure 3-37a). Since we can deal with the case of back arcs entering into v_1, or going beyond the 3-cycle, we may now assume that for all the C_0, every back arc goes into v_0.

b) There is a back arc b' coming into v_1 from T_2. This situation is dealt with analogously to the previous part, and is shown in Figure 3-37b).

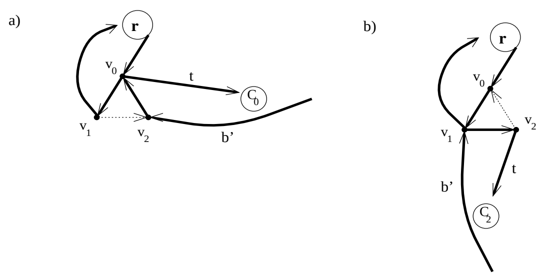

Figure 3-37: Yet more 3-cycles.

c) There is an arc a leaving T_2 and entering T_0. Since the only back arcs leaving T_0 enter into v_0 itself, a creates a path between T_2 and v_0. Thus a acts like an arc entering v_0 and can be dealt with in the same way (Figure 3-36i)), allowing the removal of (v_2, v_0). Hence we may assume that for all the C_2, all the back arcs go into v_2.

d) There is an arc a leaving T_1 and entering T_2. Now a acts like an arc entering v_2 and can be dealt with in the same way (Figure 3-36ii)), thus allowing the removal of (v_1, v_2).

e) In what follows we are interested in the possible routes from v_1 to v_2. Suppose there is an arc a from v_2 to a vertex $u \in T_0 \setminus \{v_0\}$. Now in any optimal solution there must be a directed path P from u to v_0 consisting only of vertices in that subtree. As such, there must be an optimal solution that does not contain the arc (v_2, v_0). This follows, from a forced path argument, as the arc a will also connect v_2 to v_0 via P.

Using a then is at least as good as using (v_2, v_0) since it also provides a path from v_2 to the vertices of P. Hence we may remove the arc (v_2, v_0) and resolve the problem on $G \setminus (v_2, v_0)$. By a similar argument there is no arc a from v_1 to a vertex in T_2.

f) Otherwise the only way to reach v_2 from v_1 is via the arc (v_1, v_2), or if it is present (v_0, v_2). In addition, the only way up the tree from v_2 is via the arc (v_2, v_0) or, if it is present, (v_2, v_1). Hence, at least two arcs induced by the vertices v_0, v_1 and v_2 are forced to be in an optimal solution. So by Lemma 3.8.3 we may contract the 3-cycle and recurse on the smaller graph.

Paths. We now consider the situation in which a component may represent a path P. There are three cases to analyse. The cases where the path is the root component, the target component or a component in a contracted cycle $\mathcal{C}$ associated with a different target component. Let P consist of the vertices $v_k, v_{k-1}, \ldots v_0$ with k arcs of the form (v_{i+1}, v_i). We will draw the path so that the arcs are directed from right to left. We will call an arc (v_i, v_j), $i \leq r < j$, a *forward* arc wrt v_r. We call an arc (v_j, v_i), $i \leq r < j$, a *back* arc wrt v_r. We aim to connect up the path with at most a factor $\frac{3}{2}$ blow-up. To do this we will start at the leftmost vertex of the path and add arcs to strongly connect it up to its immediate neighbours. We continue this process rightwards until all the vertices on the path are strongly connected. An example will be of use here, see Figure 3-38a). Let v_0 be the leftmost vertex. Suppose there is a forward arc d going to v_2. Then adding d strongly connects v_0, v_1 and v_2. Now these three vertices induce two path arcs, (v_2, v_1) and (v_1, v_0). We may charge d to these two arcs to give a blow-up factor $\frac{3}{2}$. We may then think of v_0, v_1 and v_2 as being contracted together to form a super-node on the path. Thus our general picture remains the same as that of Figure 3-38a) after each step.

We will show that the charge associated, in strongly connecting up the path, is at most half of the length of the path. Consider the first case, where the component representing the path is the root component. Hence we only need to ensure that the vertices on the path become strongly connected, and not worry about connecting the path upwards into the root component. We will also assume that for each T_i the

C_i will represent strongly connected components. Again, the case where the C_i are paths will be dealt with later.

During the following connection process the vertex v_0 may represent multiple nodes but $v_1, v_2 \dots v_k$ can not. From now on, let d be the forward arc reaching deepest along the path from v_0. Since v_0 may represent multiple nodes, there may be several arcs reaching the same node. Whether the choice between the arcs is important will be made apparent. Note also that, by the DFS ordering, there are no arcs from a vertex in T_i to a vertex in T_j, where $i < j$.

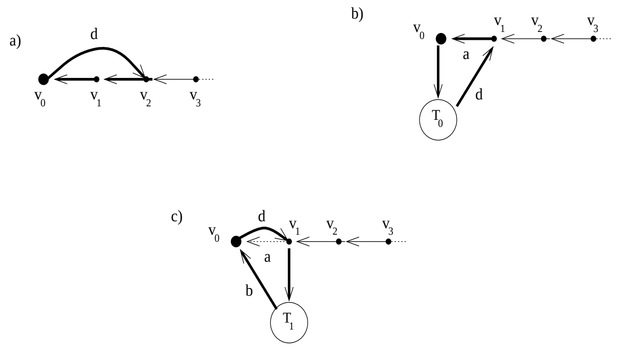

Figure 3-38: Some paths.

a) Take v_0 and consider d. If d goes to v_2 or beyond then we add d and charge it to the (at least) two path arcs. See Figure 3-38a).

b) So suppose d only reaches v_1. Now if d comes from T_0 then we consider the component C_0. Suppose that C_0 contains at least three arcs. Now adding d strongly connects together v_0, v_1 and T_0. See Figure 3-38b). We need to charge the tree arc a_0 and d. We charge them to C_0 *and* (v_1, v_0). Hence we charge these two arcs to at least four arcs contained in the lower bound, a $\frac{3}{2}$ factor. Suppose instead that C_0 is a digon. Here we may apply the "ladder of digons" argument to obtain the same factor.

c) There is a back arc b from T_1 into T_0. See Figure 3-38c). We may add d, b and the tree arc a_1 whilst removing (v_1, v_0). Similar arguments to those of part b) now apply.

d) Otherwise unravel our process one step. For clarity we use negative indices to label the vertices of the path after unravelling the process. If the previous deepest arc d'

covered at least 3 path arcs then we can charge d and d' to the (at least) four path arcs.

e) If the previous deepest arc d' covered just one path arc then d is a forced arc as it is the only way in which to construct a directed path from v_0 to v_1. Hence we may consider the graph $G \setminus (v_1, v_0)$.

f) So assume that d' covered 2 path arcs. In addition, there is an arc a from v_{-1} to v_1. Otherwise d is forced and we again consider $G \setminus (v_1, v_0)$. See Figure 3-39f). Note also that since v_1 is not a cut vertex, there must be a directed path P', that does not contain v_1, from a vertex $v_r, r \geq 2$ to a vertex $v_s, s \leq 0$. First assume that P' consists of a single arc b between vertices in P.

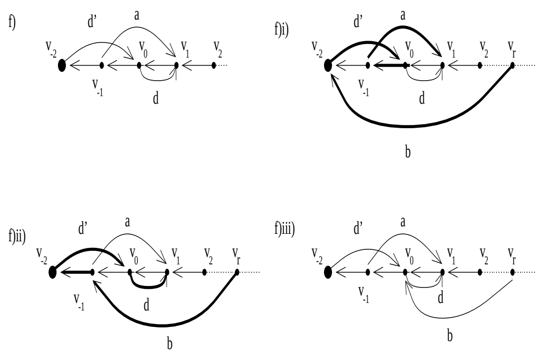

Figure 3-39: Unravelling a path.

i) If there is an arc b into v_{-2} then use the arcs $b, d', (v_0, v_{-1})$ and a. See Figure 3-39f)i). These four arcs are covered by the lower bound of the three path arcs from v_{-2} to v_1. We don't yet have a path from v_1 to v_0 but later in connecting up the rest of the path we must find a directed path from v_1 to v_r. This will complete a cycle through v_1 and v_0. In fact, the next stage in the process will then be similar to our basic case except that, in effect, we get the edge b for free (since it has already been charged to the arcs on the path between v_{-2} and v_1). This, of course, can only help us and our iterative process continues as before.

ii) If there is an arc b into v_{-1} then we proceed as in Figure 3-39f)ii) and again use 4 arcs associated with the three path arcs.

iii) Finally suppose that all the type b arcs are incident to v_0. See Figure 3-39f)iii). Consider one such arc (v_r, v_0). Now the only way to reach v_r from a vertex $v_s, s \leq 0$ in the path P is via the vertex v_1. Hence in any optimal solution there is a directed path from v_1 to v_r that does not use any of the vertices $v_s, s \leq 0$. Hence, by a forced path argument, using the arc (v_r, v_0) will always be preferable to using the arc (v_1, v_0). Thus (v_1, v_0) is a redundant arc and we may consider the graph $G \setminus (v_1, v_0)$.

Now consider the case where P' does not consist of a single arc but goes through other components in the component graph $\mathcal{G}$. The arguments above still apply with the other components incident to P' being charged just one arc.

It follows that in the specific case where the path is the root component we use at most an additional $\lfloor \frac{k}{2} \rfloor$ arcs.

Now consider the general case where P is a non-root component i.e. it is the target component or is a component in a cycle $\mathcal{C}$ contracted for a different target component. Let P be the root of a subtree T. For a worst case analysis, we may assume that there are no back arcs from a component in $T \setminus P$ to $G \setminus T$. First remove the DFS tree arc connecting $G \setminus T$ and P. Add an arc a from P to $G \setminus T$ with the property that a originates as far to the left on P as possible, say from v_j. Figure 3-40i) shows the resulting situation. Here the other possible arcs connecting P to $G \setminus T$ are labelled b, b' and d. This notation is used as the b arcs may be used as back arcs and the d arcs may be used as forward arcs as before. The b' arcs are slightly different. Adding such an arc does not create a "back path" via $G \setminus T$ unless an arc from $G \setminus T$ to P (i.e. a type b or type d arc) has already been chosen.

This allows us to proceed as before adding forward and back arcs to strongly connect up the path. We make one proviso. Should a b' arc from P to $G \setminus T$ be required as a back arc, and no arc from $G \setminus T$ to P has currently been selected, then we add an appropriate arc and remove a. An example where the addition of b' leads to the removal of a and addition of b is shown in Figure 3-40ii). Note that such a method still leads to a net charge of one when b' is added.

The previous arguments, used for the specific case of the path being the root

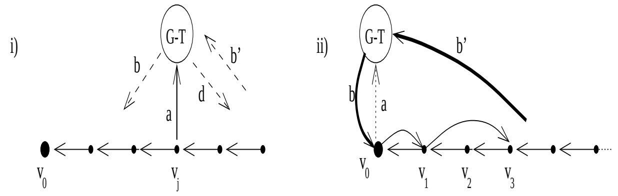

Figure 3-40: A non-root path.

component, may now be applied to this general case. Hence $\lfloor \frac{k}{2} \rfloor$ arcs are required to internally strongly connect the path. Now we have two possible situations after we have internally connected up the path.

Type I) No arcs between P and $G \setminus T$ were chosen in the process (except for our original choice of a). In this case we need to add an extra arc a' to connect the path upwards. So our total cost is at most $\lfloor \frac{k}{2} \rfloor + 2$ (including one each for a and a'). Notice that a and a' play no part in internally connecting up the path. This observation will be of use later.

Type II) Some arcs, apart from a, between P and $G \setminus T$ were chosen in the process. These arcs are included in the $\lfloor \frac{k}{2} \rfloor$ charge to the path but they also succeed in strongly connecting the path upwards. Hence the charge is $\lfloor \frac{k}{2} \rfloor + 1$ (including one for a).

Recall that associated with each path were two "free" arcs and so we actually have $k + 2$ arcs to charge these additional arcs to. It follows that if the path is a target component then the total number of arcs used is at most $k + \lfloor \frac{k}{2} \rfloor + 2 < \frac{3}{2}(k + 2)$, as required.

So consider the case in which the path is a component in a cycle $\mathcal{C}$ for which the target component was a digon or 3-cycle. Again associated with the target path are two free arcs. Thus, we have at least three arcs (due to the $\frac{3}{2}$ performance guarantee one extra arc is also allowed with the two free arcs) which we can use in strongly connecting up a path component. It now follows easily that it would only have been beneficial to encounter any paths in any of the circuits we found in the previous sections, where the target components were digons and 3-cycles.

Finally, consider the case in which the path is a component in a circuit for which the target component is another path. Recall that this occurs when we reallocate path arcs to a child component C_i below. We postponed the discussion of the situation in which these child components were themselves paths. We deal with this topic here. Call the upper path P' and the lower path P. In this situation we allocate one arc from P' to the P and force two other arcs to be chosen that strongly connect upwards P. A total of three extra arcs. Now, if the lower path, P, is of type I) the arcs a and a' are superfluous so the charge to P is $\lfloor \frac{k}{2} \rfloor + 3$. Whereas, if P is of type II) then the arc a was needed in connecting up the path so with these extra three arcs we have a charge of $\lfloor \frac{k}{2} \rfloor + 4$. Given that we also reassigned one arc from P' to P we now have $k + 3$ arcs associated with P. Now since P has a single parent such extra arcs can be reallocated from P' to P at most once. This completes the proof that the blow-up factor for paths is at most $\frac{3}{2}$ since $k + \lfloor \frac{k}{2} \rfloor + 4 < \frac{3}{2}(k + 3)$. The theorem follows. $\square$

Corollary 3.8.5. *The algorithm is a $\frac{3}{2}$-approximation algorithm for the minimum strongly connected subgraph problem.* $\square$

Bibliography

[1] V. Auletta, Y. Dinitz, Z. Nutov and D. Parente, "A 2-approximation algorithm for finding an optimum 3-vertex-connected spanning subgraph", *Journal of Algorithms*, **32**, pp21-30, 1999.

[2] J. Cheriyan, T. Jordan and Z. Nutov, "On rooted node-connectivity problems", *Algorithmica*, **30**, pp353-375, 2001.

[3] A. Czumaj and A. Lingas, "On approximability of the minimum-cost k-connected spanning subgraph problem", *Proc. 10th Annual SODA*, pp281-90, 1999.

[4] R. Carr and R. Ravi, "A new bound for the 2-edge connected subgraph problem", *Integer programming and combinatorial optimization*, Springer-Verlag, Lecture Notes in Computer Science, **1412**, pp112-25, 1998.

[5] J. Cheriyan, A. Sebö and Z. Sigeti, "Improving on the 1.5-approximation of a smallest 2-edge connected spanning subgraph", *SIAM J. Discrete Math*, **14**, pp170-180, 2001.

[6] J. Cheriyan and S. Vempala, "Edge coverings of setpairs and the iterative rounding method", *Proc. 9th Annual International IPCO*, pp30-44, 2001.

[7] J. Cheriyan, S. Vempala and A. Vetta, "Approximation algorithms for minimum-cost k-vertex connected subgraphs", to appear *Proc. 34th Annual STOC*, 2002.

[8] J. Cheriyan, S. Vempala and A. Vetta, "Network design via iterative rounding of setpair relaxations", submitted for publication, 2001.

[9] C. Fernandes, "A better approximation ratio for the minimum k-edge-connected spanning subgraph problem", *Proc. 8th Annual SODA*, pp629-638, 1997.

[10] A. Frank, "Augmenting graphs to meet edge-connectivity requirements", *SIAM J. Discrete Math.*, **5**, pp25-53, 1992.

[11] A. Frank, "Connectivity augmentation problems in network design", in J. Birge and K. Murty (eds.), *Mathematical Programming: State of the Art 1994*, University of Michigan, Ann Arbor, pp34-63, 1994.

[12] A. Frank and T. Jordan, "Minimal edge-coverings of pairs of sets", *Journal of Combinatorial Theory, Series B*, **65**, pp73-110, 1995.

[13] A. Frank and E. Tardos, "An application of submodular flows", *Linear Algebra and its Applications*, **114/115**, pp329-348, 1989.

[14] L. Fleischer, K. Jain, and D. Williamson, "An iterative rounding 2-approximation algorithm for the element connectivity problem", *Proc. 42nd FOCS*, pp339-47, 2001.

[15] H. Gabow, "A representation for crossing set families with applications to submodular flow problems", *Proc. 4th Annual SODA*, pp202-211, 1993.

[16] H. Gabow, "Using expander graphs to find vertex connectivity", *Proc. 41st FOCS*, pp410-20, 2000.

[17] M. Garey and D. Johnson, *Computers and Intractability : A Guide to the Theory of NP-completeness*, Freeman, San Francisco, 1979.

[18] N. Garg, A. Singla and S. Vempala, "Improved approximations for biconnected subgraphs via better lower bounding techniques", *Proc. 6th Annual SODA*, pp103-11, 1993.

[19] M. Goemans, "Worst-case Comparison of Valid Inequalities for the TSP", *Mathematical Programming*, **69**, pp335-49, 1995.

[20] M. Goemans, A. Goldberg, S. Plotkin, D. Shmoys, E. Tardos and D. Williamson, "Improved approximation algorithms for network design problems", *Proc. 5th Annual SODA*, pp223-232, 1994.

[21] M. Goemans and D. Williamson, "A general approximation technique for constrained forest problems", *SIAM Journal on Computing*, **24**, pp296-317, 1995.

[22] M. Grötschel, L. Lovász and A. Schrijver, *Geometric algorithms and combinatorial optimization*, Springer-Verlag, New York, 1988.

[23] D. Hartvigsen, "Extensions of Matching Theory", Ph.D. thesis, GSIA, Carnegie Mellon University, 1984.

[24] H. Hsu, "An Algorithm for Finding a Minimal Equivalent Graph of a Digraph", *Journal of the Association for Computing Machinery*, **22(1)**, pp11-16, 1975.

[25] K. Jain, "A factor 2 approximation algorithm for the generalized Steiner network problem", *Combinatorica*, **21**(1), pp39-60, 2001.

[26] K. Jain, I. Mandoiu, V. Vazirani and D. Williamson, "A primal-dual schema based approximation algorithm for the element connectivity problem", *Proc. 10th Annual SODA*, pp484-489, 1999.

[27] S. Khuller and B. Raghavachari, "Improved approximation algorithms for uniform connectivity problems", *Journal of Algorithms*, **21**, pp434-450, 1996.

[28] S. Khuller, B. Raghavachari and N. Young, "Approximating the Minimum Equivalent Digraph", *Proc. 5th Annual SODA*, **41(4)**, pp859-872, 1995.

[29] S. Khuller, B. Raghavachari and N. Young, "On Strongly Connected Digraphs with Bounded Cycle Length", *Discrete Applied Mathematics*, **69(3)**, pp281-289, 1996.

[30] S. Khuller and U. Vishkin, "Biconnectivity approximations and graph carvings", *Journal of the ACM*, **41**, pp214-35, 1994.

[31] G. Kortsarz and Z. Nutov, "Approximating node connectivity problems via set covers", in K. Jansen and S. Khuller (eds.), *Approximation Algorithms for Combinatorial Optimization*, Springer-Verlag, Lecture Notes in Computer Science, **1913**, pp194-205, 2000.

[32] W. Mader, "Endlichkeitsätze für k-kritische Graphen", *Math. Ann.*, **229**, pp143-153, 1977.

[33] W. Mader, "Connectivity and edge-connectivity in finite graphs", *Surveys in Combinatorics*, London Mathematical Society Lecture Note Series, **38**, Cambridge University Press, pp66-95, 1979.

[34] W. Mader, "On k-critically n-connected graphs", in J. Bondy and K. Murty (eds.), *Progress in Graph Theory*, Academic Press, Toronto, pp389-398, 1984.

[35] C. Monma, B. Munsun and W. Pulleyblank, "Minimum-weight two-connected spanning networks", *Math. Programming*, **46**, No.2 (Series A), pp153-71, 1990.

[36] D. Moyles and G. Thompson, "An Algorithm for Finding a Minimum Equivalent Graph of a Digraph", *Journal of the Association for Computing Machinery*, **16(3)**, pp455-460, 1969.

[37] R. Ravi and D. Williamson, "An approximation algorithm for minimum-cost vertex-connectivity problems", *Algorithmica*, **18**, pp21-43, 1997.
Erratum, submitted to *Algorithmica*, 2001.

[38] E. Tardos, "A strongly polynomial algorithm to solve combinatorial linear programs", *Operations Research*, **34**, pp250-256, 1986.

[39] S. Vempala and A. Vetta, "Factor $\frac{4}{3}$ Approximations for Minimum 2-Connected Subgraphs", in K. Jansen and S. Khuller (eds.), *Approximation Algorithms for Combinatorial Optimization*, Springer-Verlag, Lecture Notes in Computer Science, **1913**, pp262-73, 2000.

[40] A. Vetta, "Approximating the minimum strongly connected subgraph via a matching lower bound", *Proc. 12th Annual SODA*, pp417-26, 2001.