Approximation Algorithms for Finding Highly Connected Subgraphs*

Samir Khuller
Dept. of Computer Science
University of Maryland
College Park, MD 20742
samir@cs.umd.edu
(301) 405 6765

*This chapter is dedicated to Prof. Richard Karp whose Turing Award Lecture "Combinatorics, Complexity and Randomness" (*Communications of the ACM*, Feb 1986) inspired this author to start working in the field of algorithms.

Contents

1 Introduction

Let a graph $G = (V, E)$ denote the feasible links of a (proposed) communications network. An edge $e = (a, b)$ denotes the feasibility of adding a link between sites a and b. The weight of this edge, $w(e)$, represents the cost of constructing link e. A minimum spanning tree in G is the lowest weight connected subgraph, i.e., the cheapest network that will allow the sites to communicate. Such a network is highly susceptible to failures, since it cannot even survive a single link or site failure. For more reliable communication, one desires spanning subgraphs of higher connectivity. A network of edge-connectivity λ continues to allow communication between functioning sites even after as many as $\lambda - 1$ links have failed. A graph is said to be λ edge-connected if the deletion of any $(\lambda - 1)$ edges leaves it connected. These definitions extend in a straightforward way to λ vertex-connectivity. The only requirement is that the graph should have at least $\lambda + 1$ vertices, and the deletion of any $(\lambda - 1)$ vertices should leave it connected. Given a graph G with non-negative edge weights, and an integer λ, we consider the problem of finding a minimum-weight λ-connected spanning subgraph. We address the cases of edge connectivity, and vertex connectivity. For most connectivity versions, the associated problems are NP-hard. In this case we would like to obtain sub-optimal solutions in polynomial time. From now on we will refer to sites as vertices, and links as edges.

In this chapter, we address only *uniform* connectivity problems. For results on non-uniform connectivity requirements, see Chapter 4 by Goemans and Williamson. The non-uniform connectivity problems are solved using the "primal-dual" method of linear programming; this usually results in approximation factors that are not as good as the ones obtained here.

Edge connectivity augmentation problems were first studied by Eswaran and Tarjan [9]. They studied the problem of making a given graph 2-connected (both vertex and edge connectivities were considered) and strongly connected with the addition of the least number of edges. They showed that when all potential edges are feasible and have weight 1, the problem can be solved optimally in polynomial time, and when the edges have arbitrary weights the problem is NP-hard. Subsequently, a lot of work was done on the problem of "increasing" the connectivity of a given graph; most of these papers deal with the unweighted case where an edge may be added between *any* pair of vertices. This problem can be solved *optimally* in polynomial time, at least for the edge-connectivity case. We will not survey this body of research in detail here since we are primarily interested in approximation techniques for NP-hard problems. For more information on such problems see recent papers by Frank [10], and Naor, Gusfield and Martel [32]. For the vertex-connectivity case, the problem appears to be significantly harder and no polynomial time algorithm is known for finding the optimal solution. In his doctoral thesis, Hsu [22] gives algorithms for vertex connectivity for small connectivity values. These algorithms are quite complex. It must be pointed out that the problem of constructing a graph with n vertices, and connectivity λ with the least number of edges was first addressed by Harary [19].

The first paper to address the issue of obtaining approximate solutions for the case when edges have weights, is by Frederickson and JáJá [11]. They provide approximation algorithms for the cases of 2-connectivity (edge and vertex) as well as strong connectivity problems. Subsequently, their algorithm was simplified by Khuller and Thurimella[26, 27]. The unweighted case was explored by Khuller and Vishkin [28], and Garg, Santosh and Singla [16]. For any k, fast algorithms for finding sparse certificates were given by Nagamochi and Ibaraki [33] and Cheriyan, Kao and Thurimella [5]. The strong connectivity case is addressed by Khuller, Raghavachari and Young [24, 25]. When parallel edges are allowed, Goemans and Bertsimas provide an approximation algorithm [17].

1.1 Outline of Chapter

The problems we deal with are divided broadly into four categories: edge connectivity, vertex connectivity, strong connectivity and connectivity augmentation. In each case, we study both the weighted and unweighted problems.

In Section 2 we discuss the edge-connectivity results. This section surveys known results for both the weighted case as well as the $\{1/\infty\}$ case (where each edge has weight either 1 or ∞). In other words, the feasibility network is treated as an undirected graph, and each possible link is either feasible or infeasible. In this case we are interested in minimizing the total number of edges in our solution. Section 3 discusses the results on vertex connectivity. In Section 4 we discuss the problem of finding strongly connected spanning subgraphs in directed graphs. In Section 5 we study the problem of increasing the edge-connectivity of a given graph having an arbitrary connectivity, to being λ edge-connected.

2 Edge-Connectivity Problems

We begin this section by describing the algorithm given by Khuller and Vishkin [28] for obtaining an approximation factor of 2 when the edges have weights. In Subsection 2.2 we consider the special case when the weights are either 1 or ∞; for this special case we can achieve approximation ratios less than 2.

2.1 Weighted Edge-Connectivity

Given a graph $G = (V, E)$ with weights on the edges and an integer λ, consider the problem of finding a *minimum* weight spanning subgraph $H = (V, E_H)$ that is λ edge-connected.

An algorithm that achieves an approximation factor of 3 for $\lambda = 2$ follows by the work of Frederickson and JáJá [11]. First find a minimum spanning tree. Now consider the problem of finding the least weight set of edges to add to the tree to obtain a 2 edge-connected subgraph. Not surprisingly, this is NP-hard as well [11]. They give an algorithm with an approximation factor of 2 for the problem of augmenting connectivity, yielding an approximation factor of 3 for the least weight 2 edge-connected spanning subgraph. (In Section 5 we describe a simplification of their algorithm due to Khuller and Thurimella [26].)

We now briefly review the method given by Khuller and Vishkin [28] that yields an approximation algorithm for undirected graphs. Take the undirected graph G, and replace each undirected edge $e = (u, v)$ by two directed edges (u, v) and (v, u) with each edge having weight $w(e)$. Call this graph G^D. Now consider the following problem for directed graphs: given a directed graph G^D with weights on the edges, and a fixed root r, how does one find the *minimum weight* directed subgraph H^D that has λ edge-disjoint paths from a fixed root r to each vertex v? Gabow [13] gives the fastest implementation of a weighted matroid intersection algorithm due to Edmonds [8] to solve this problem optimally in $O(\lambda n(m + n \log n) \log n)$ time. Run Gabow's algorithm on the graph G^D, with an arbitrary vertex r chosen as the root. If at least one of the directed edges (u, v) or (v, u) is picked in H^D, then we add (u, v) to E_H.

Lemma 2.1 *The graph $H = (V, E_H)$ is a λ edge-connected spanning subgraph of G.*

Proof. Suppose (for contradiction) that there is a $\lambda - 1$ edge cut in H. Assume that it separates H into pieces C_1 and C_2. Let r be in C_1, now consider a vertex v in C_2. It is clear that r cannot have λ edge-disjoint directed paths to v. Thus there is no cut set of size $\lambda - 1$. □

Theorem 2.2 *The total weight of E_H is at most twice the weight of the optimal solution.*

Proof. Consider an optimal solution $\mathcal{OPT}(G)$ for the minimum weight λ edge-connected subgraph problem. Consider all the anti-parallel edges corresponding to edges in $\mathcal{OPT}(G)$. We get a directed subgraph in G^D of weight $2w(\mathcal{OPT}(G))$ (where $w(\mathcal{OPT}(G))$ is the total weight of the edges in $\mathcal{OPT}(G)$). From r there are λ edge-disjoint undirected paths to any vertex v; these also yield λ directed paths from r to v that are edge-disjoint. Thus this subgraph has the property of having λ directed edge-disjoint paths from r to any vertex v. The optimum solution found by Gabow's algorithm is only cheaper. $\square$

2.2 Unweighted Edge-Connectivity

Given an undirected graph G with n vertices and m edges, we would like to find a subgraph H that is λ-edge connected and has as few edges as possible. For the general case, Nagamochi and Ibaraki [33] showed how to find a spanning subgraph with at most λn edges (see also Thurimella's doctoral thesis [37]) that has edge-connectivity λ if and only if the original graph G has edge connectivity λ. Since each vertex is required to have degree at least λ, we get $\frac{\lambda n}{2}$ as a lower bound on any λ edge-connected spanning subgraph. Thus this yields an approximation algorithm with a ratio of 2. In this section we describe a simple algorithm given by Khuller and Vishkin [28] that finds a 2 edge-connected spanning subgraph by using Depth First Search. Moreover, it is shown that this algorithm achieves an approximation ratio of 1.5. Combining this the ideas of [33, 34, 37] yields an approximation ratio of $2 - \frac{1}{\lambda}$.

2.2.1 2 Edge-Connectivity

In this section we present a linear time algorithm given by Khuller and Vishkin [28] to obtain a 2 edge-connected spanning subgraph from a given graph G. This algorithm obtains a solution that is at most $\frac{3}{2}$ times the optimal solution.

High-level Description of the Algorithm

We traverse G using depth-first-search (DFS). A DFS rooted tree T is computed; T has at most $n-1$ edges, and all the non-tree edges are *back* edges (i.e., one of the endpoints of the edge is an ancestor of the other in T). All edges of T are picked for E_H. During the depth-first search the algorithm also picks a set of non-tree edges that will increase the edge connectivity by "covering" all the edges in T (since each edge in T is a potential bridge). A back edge may be chosen just before *withdrawing from a vertex for the last time.* Before withdrawing from a vertex v, we check whether the edge $(v, p(v))$, joining v to its parent, is currently a bridge or not. If $(v, p(v))$ is still a bridge, we cover it by adding to E_H a back edge from a descendant of v to low$[v]$, where **low**$[v]$ is the vertex with the smallest dfs-number that can be reached by following zero or more downgoing tree edges from v, and a single back edge.

The Algorithm - a Detailed Description

In this section we give a detailed recursive description of the algorithm. The running time is $O(n+m)$, the algorithm is simple to implement and uses no complicated data structures.

Data Structures:

dfs$[v]$: A serial number given to a vertex the first time it is visited during DFS. For simplicity, we will assume that vertices are numbered by their dfs-number (i.e., $v = \text{dfs}[v]$).

state of a vertex: Each vertex is initially "*unvisited*". After the DFS traversal visits it for the first time, it becomes "*discovered*". When we finally exit from the vertex it becomes "*finished*". (This is to be able to tell when we are looking at back edges from the upper end.)
low$[v]$**:** defined earlier.
low$_H[v]$**:** This is defined to be the smallest numbered vertex that can be reached by following zero or more downgoing tree edges from v, and a *single* back edge that belongs to E_H.
savior$[v]$**:** This is defined to be the descendant end vertex of the back edge that goes to low$[v]$.
Initialization Step: The initial call made is DFS(v, nil) where v is an arbitrary vertex. We assume that G is a 2-edge connected graph (easy to verify this before running the algorithm). Initially, all vertices are "unvisited".
Algorithm *Find 2-EC Spanning Subgraph*
Input: Graph $G = (V, E)$.
Output: A subgraph $H = (V, E_H)$ that is 2-EC.

```
procedure DFS(v, u);      (* u is the parent of v in DFS tree. *)
    mark v discovered;
    low[v] = v;
    low_H[v] = v;
    savior[v] = v;
    for each w ∈ Adj[v] do
        if w is unvisited then begin
            E_H = E_H ⋃ {(v, w)};      (* (v, w) is a tree edge *)
            DFS(w, v);
            low[v] = min(low[v], low[w]); If low[v] changes, set savior[v] = savior[w];
            low_H[v] = min(low_H[v], low_H[w]);
        end
        else if w is discovered then begin
                if w ≠ u then      (* (v, w) is a back edge *)
                    low[v] = min (low[v], w); If low[v] changes, set savior[v] = v;
                (* else (v, w) is already a tree edge *)
            (* else w is finished and is a descendant of v *)
        end
    mark v finished;
    If low_H[v] = v and u ≠ nil then begin
        (* edge (u, v) is threatening to be a bridge *)
        (* add the edge ( savior[v], low[v] ) to cover the bridge *)
        E_H = E_H ⋃{( savior[v], low[v] )};
        low_H[v] = low[v];
    end
end DFS
```

It is quite easy to see that H has edge-connectivity 2, and that the algorithm runs in time $O(n + m)$.

The Approximation Analysis

Our analysis finds a partition of the vertices, called a *tree-carving*, which is used to prove a lower bound on OPT, the number of edges in the optimal solution. The upper bound of $\frac{3}{2}$ on the approximation factor is established using this lower bound. After presenting the concept of a tree-carving, we apply it to the approximation analysis.

Definition 1 *A* **tree-carving** *of a graph is a partition of the vertex set V into subsets $V_1, V_2, \ldots, V_k$ with the following properties. Each subset constitutes a node of a tree Γ. For every vertex $v \in V_j$, all the neighbours of v in G belong either to V_j itself, or to V_i where V_i is adjacent to V_j in the tree Γ. The* **size** *of the tree-carving is k.*

We will refer to the vertices of Γ as *nodes*, and the edges of Γ as *arcs*.

Theorem 2.3 (Tree-Carving Theorem)
If graph $G = (V, E)$ has a tree-carving of size k, then a lower bound on the number of edges of any 2 edge-connected spanning subgraph in G is $2(k-1)$.

It is interesting to note that the same simple proof implies that the smallest λ-connected subgraph of G must have at least $\lambda(k-1)$ edges.
Proof. There are $k-1$ arcs in the tree Γ. Each such arc $e = (V_i, V_j)$ partitions the vertices in G into two sets S_e and $V - S_e$. (Deletion of arc e breaks Γ into two trees Γ_1 and Γ_2, where V_i belongs to Γ_1. S_e is defined to be the union of the sets V_y that belong to Γ_1.) In any 2 edge-connected spanning subgraph we have: (1) at least two edges going from S_e to $V - S_e$, and (2) both these edges must have one endpoint in V_i and another in V_j; from the disjointness of V_i's it follows that for each arc e, there are two distinct edges in the subgraph. Since Γ has $k-1$ arcs, we get a lower bound of $2(k-1)$. $\square$

For an example of a tree-carving see Fig. 1.

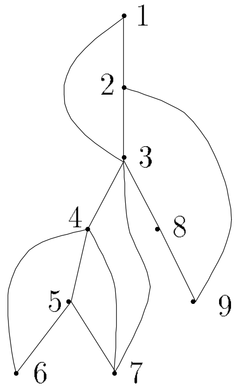
Given T, the DFS spanning tree, we will be interested in the following partition of the vertices of G, called the *DFS-tree partition.* Some recursive calls DFS(v, u) end by adding the back edge (savior$[v]$, low$[v]$) to E_H, and some do not add any edge. For each call DFS(v, u) where a back edge is added to E_H, "remove" the tree edge (u, v) from T; the resulting connected components of T (with some tree edges removed) provides the DFS-partition. Furthermore, T induces a *rooted tree structure* Γ on the sets in the DFS-tree partition. In fact, it is easy to modify the approximation algorithm to find the tree-carving as well; however this is not essential since it is only used for the analysis of the algorithm.

Theorem 2.4 *The DFS-tree partition yields a tree-carving of G.*

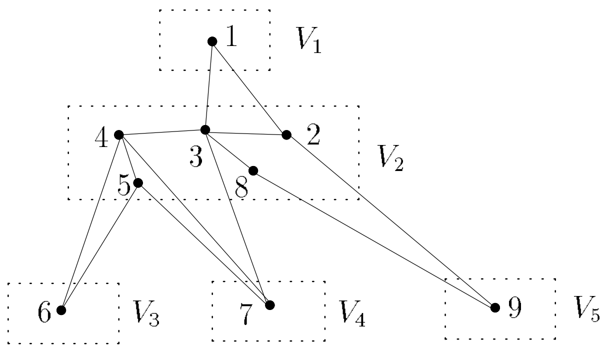
Proof. Let (v_1, v_2) be any non tree edge in G. Suppose that v_1 is in set V_1 of the DFS-tree partition and v_2 is in set V_2. Let us assume that v_1 is an ancestor of v_2. Clearly low$[v_2] \leq v_1$. Thus by the algorithm there can be at most one bridge between them. Hence, either $V_1 = V_2$, or set V_1 is the parent set of set V_2 (in the rooted tree structure Γ). $\square$

Corollary 2.5 *Since the number of arcs in the tree-carving is exactly the same as the number of back edges that are added to E_H we conclude that $OPT \geq 2(k-1)$, where $k-1$ is the number of added back edges.*

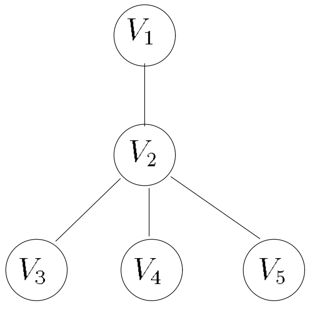
Theorem 2.6 *The algorithm outputs a solution of size no more than $\frac{3}{2}$ OPT.*



(a) G



(b) Tree-Carving of size 5 for G



(c) The tree Γ

Figure 1: Example to show 2 edge-connectivity algorithm and a tree-carving.

Proof. The number of edges added by the algorithm to H is: (i) $(n-1)$, for the tree edges, plus (ii) $k-1$ back edges, where k is also the size of the tree-carving. Hence, the number of edges in E_H is $n-1+k-1$. Let OPT be the number of edges in an optimal solution. A lower bound on OPT is $\max(n, 2(k-1))$, since n is the minimum number of edges in a 2-edge connected graph with n vertices (each vertex should have degree at least 2), and $2(k-1)$ follows from Corollary 2.5. Hence, the ratio of the algorithm's solution to OPT is

$$\leq \frac{n-1+k-1}{\max(n, 2(k-1))}.$$

If $n \geq 2(k-1)$, then clearly the ratio is $< 3/2$. If $n \leq 2(k-1)$, it is again easy to see that the ratio is $< 3/2$. ∎

2.2.2 λ Edge-Connectivity

We now describe a linear time algorithm given by Nagamochi and Ibaraki [33] that finds a λ-connected spanning subgraph of a given graph G that has connectivity at least λ. The algorithm finds a subgraph with at most $\lambda(n-1)$ edges; since every vertex has degree at least λ, we get a lower bound of $\frac{\lambda n}{2}$ for OPT. Hence this is a factor 2 approximation. We then use the previous DFS based algorithm for 2 edge-connectivity to improve this ratio by $\frac{1}{\lambda}$.

The main idea behind their algorithm is to repeatedly find *maximal* spanning forests in the graph, and to delete them. After λ iterations of this method, we obtain λ forests, which form a λ edge-connected spanning subgraph assuming that the input graph was λ edge-connected. More formally, we state the following lemma (also due to [37, 34]).

Lemma 2.7 *For a graph $G = (V, E)$ that has edge connectivity λ, let $F_i = (V, E_i)$ be a maximal spanning forest in $G - E_1 \cup \ldots \cup E_{i-1}$, for $i = 1 \ldots \lambda$; then $G_\lambda = (V, E_1 \cup \ldots \cup E_\lambda)$ has edge connectivity λ.*

Proof. Assume (for contradiction) that G_λ contains a cut C of size $k < \lambda$ whose removal disconnects the graph G_λ into G'_λ and G''_λ. Clearly, at least one forest, say F_j, does not have any edges in C. Since the original graph G was λ edge-connected it must be the case that there is at least one edge in G between the two components G'_λ and G''_λ. Hence in iteration j when we were picking F_j we would pick an edge connecting G'_λ and G''_λ. ∎

It is easy to find the set of forests by repeatedly scanning the graph λ times [37, 34]. The amazing fact about Nagamochi and Ibaraki's algorithm is that they can find all the forests in a *single* scan of the graph. During the search, for each edge e we compute the integer i satisfying $e \in E_i$. In fact, the algorithm assigns each edge to the forest it would have ben assigned if we repeatedly removed spanning forests until the graph was completely exhausted.

For each vertex v, we maintain the rank $r(v)$, and $r(v) = i$ if v has been reached by an edge of the forest F_i.

We now argue that the algorithm in Fig. 2 implements the algorithm that repeatedly finds forests and deletes them. Formally, what is shown is that each $F_i = (V, E_i)$ is a maximal spanning forest in $G - E_1 \cup \ldots \cup E_{i-1}$.

In Fig. 3 we illustrate the execution of the algorithm via a small example.

```
λ Connectivity —
Label all nodes and edges as "unscanned"
r(v) = 0 for all v ∈ V
while there exist "unscanned" nodes do
        Choose an "unscanned" node x with the largest r
        for each "unscanned" edge e = (x, y) do
            if r(x) = r(y) then r(x) = r(x) + 1
            r(y) = r(y) + 1
            E_{r(y)} = E_{r(y)} ∪ e
            Mark e scanned
        Mark x scanned
```

Figure 2: Nagamochi and Ibaraki's Algorithm to find a λ connected graph.

Proposition 2.8 *For a vertex v let $E(v)$ denote the edges incident to v. At the start of each iteration of scanning an unscanned edge*

$$E(v) \cap E_i \neq \emptyset \textit{ for } i = 1, \ldots, r(v)$$

$$E(v) \cap E_i = \emptyset \textit{ for } i = r(v)+1, \ldots, \lambda.$$

Proposition 2.8 immediately implies that each subgraph F_i is acyclic, since we add edge $e = (x, y)$ to E_i only when $r(y)$ first becomes i, so there is no edge in E_i incident on y when e is added.

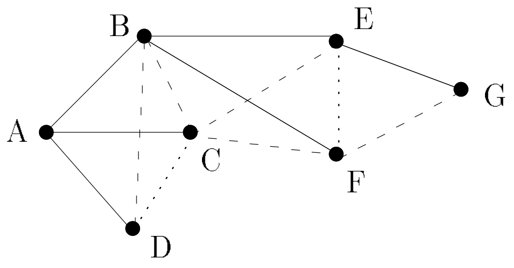
Before we prove that each forest F_i is maximal in $G - E_1 \cup \ldots \cup E_{i-1}$, we give some definitions. If an edge $e = (u, v)$ with $E(u) \cap E_i = \emptyset$ and $E(v) \cap E_i = \emptyset$ is added to E_i then the edge e is called the root edge of E_i. The vertex u is called the root vertex of E_i if it is scanned before v. (The reader should convince themselves that this edge is unique.) The key intuition is that once we create a tree T in E_i, *before* starting a new tree T' in E_i, we will have scanned all the nodes in T. This would guarantee that we do not process an edge between T and T' at some later point of time (and erroneously put that edge in E_{i+1}).

Lemma 2.9 *When we add an edge e to E_i there exists a path $P_{i-1} \subseteq E_{i-1}$ connecting u and v.*

Proof. Suppose there is no path connecting u and v in E_{i-1}. Then there must be two trees T_u and T_v that contain u and v respectively (observe that the label of u and the label of v is $\geq i-1$). Let u_0 and v_0 be the roots of these trees. Let the path from u_0 to u be $P = [u_0, u_1, \ldots, u_k = u]$. W. l. o. g u_0 was scanned before v_0. When u_0 was scanned $r(u_0) = i-2$ and $r(v_0) \leq i-2$. After scanning (u_0, u_1), $r(u_1) = i-1$ and is scanned before v_0. In a similar manner we can argue that all the nodes on P are scanned before v_0 including v (after we scan the node u). This is a contradiction to the assumption that v_0 is the root of T_v. □

Lemma 2.10 *If there is a path $P_j \subseteq E_j$ connecting u and v, then there are paths $P_i \subseteq E_i$ connecting u and v, for all $i < j$.*

Proof. For each edge in P_j, by Lemma 2.9 we know that there is a path in E_{j-1} connecting the endpoints of that edge. Taking the union of all the paths for each edge, gives us a path P_{j-1} from u to v in E_{j-1}. Similarly, we can prove this for $i = j-2, \ldots, 1$ etc. □



— Edges in E_1
- - Edges in E_2
···· Edges in E_3

Table to show r(v) values

	A	B	C	D	E	F	G
Initial	0	0	0	0	0	0	0
Scan A	1	1	1	1	0	0	0
Scan B	1	2	2	2	1	1	0
Scan C	1	2	3	3	2	2	0
Scan D	1	2	3	3	2	2	0
Scan E	1	2	3	3	3	3	1
Scan F	1	2	3	3	3	3	2
Scan G	1	2	3	3	3	3	2

Figure 3: Example to show the running of the Algorithm.

Theorem 2.11 *Each graph* $F_i = (V, E_i)$ *is a maximal spanning forest in* $G - E_1 \cup \ldots \cup E_{i-1}$.

Proof. We argued earlier that F_i is acyclic. If it is not maximal in $G - E_1 \cup \ldots \cup E_{i-1}$ then there is an edge $e \in E_j$ (with $j > i$) such that $(V, E_i \cup e)$ is a forest. By Lemma 2.9 we know that E_i must contain a path P_i from u to v. This would give a contradiction to the fact that $(V, E_i \cup e)$ is a forest. □

We now show how to find a subgraph of edge connectivity λ that is at most $2 - \frac{1}{\lambda}$ times the optimal. Find a 2 edge-connected graph by using the DFS based algorithm described earlier. Let this graph be called H_2. Now add $\lambda - 2$ forests to H_2, by repeatedly removing the edges on each forest (see Fig. 4).

λ Connectivity —
1 Find H_2 a 2-edge connected subgraph using the DFS based algorithm.
2 **for** $i = 3, ..., \lambda$
3 Let T_i be a spanning forest in $G - H_{i-1}$.
4 Let $H_i = H_{i-1} \cup T_i$.

Figure 4: Algorithm to find a λ connected graph.

A simple proof that this yields a λ edge-connected graph can be obtained in a manner similar to the proof of Lemma 2.7. The proof is left as an exercise for the reader.

Now let us bound the total number of edges added by this procedure. The number of edges in H_2 is (i) $n - 1$, for the tree edges plus (ii) $k - 1$ back edges, where k is the size of the tree-carving. In Step 4, we add at most $(\lambda - 2)n$ edges to make the graph λ connected.

An obvious lower bound on the optimal solution is $\frac{\lambda n}{2}$ (by a degree argument); and $\lambda(k - 1)$

using the tree-carving lower bound. Putting this together we get

$$\frac{(n-1)+(k-1)+(\lambda-2)n}{\max\{\lambda(k-1),\frac{\lambda n}{2}\}}$$

Simplifying, we get the upper bound of $(2-\frac{1}{\lambda})$.

Remark: Recently, Khuller and Raghavachari [23] were able to obtain an algorithm with a performance ratio of at most 1.85 for *any* λ. The key idea is to augment the connectivity by two in each stage by adding 2 edge-connected subgraphs. The proof requires a subtle argument, and uses the notion of tree-carvings.

Open Problem: It seems likely that one should be able to obtain algorithms for which the performance ratio improves as λ increases, at least for the unweighted case. However, we have not been able to do this as yet. An increased understanding of higher connectivity seems essential before this can be done.

3 Vertex-Connectivity Problems

3.1 Weighted Vertex-Connectivity

For the general problem no constant factor approximation algorithms are known. The best known algorithm to find a λ-connected subgraph for the weighted case is the algorithm due to Ravi and Williamson [35] that achieves a factor of $2H(\lambda)$, where $H(\lambda) = 1 + \frac{1}{2} \ldots + \frac{1}{\lambda}$. For the case of finding a 2 vertex-connected graph, an approximation algorithm achieving a factor of 3 was given by Frederickson and JáJá, through solving the more general graph augmentation problem. It is possible to obtain an approximation factor of $2+\frac{1}{n}$ by using a technique similar to the one used in Subsection 2.1.

Frank and Tardos [12] extended Edmonds method [8] to show that the following problem can be solved in polynomial time: Given a directed graph G^D with weights on the edges, and a fixed root r. Find the *cheapest* directed subgraph H^D that has λ *internally vertex-disjoint* paths from a fixed root r to each vertex v.

Using this algorithm as a subroutine it is possible to obtain a factor 2 approximation for the weighted case, when $\lambda = 2$.

The idea is as follows: Create a new graph G^D as follows: for each undirected edge $e = (u, v)$ in G create bi-directional edges (u, v) and (v, u) in G^D, each of weight $w(e)$. Let $e' = (x, y)$ be the lowest weight edge in G.

We create a new vertex r as the root and add directed edges (r, x) and (r, y) of weight 0. We now run Frank and Tardos's algorithm to find the minimum weight subgraph H^D with $\lambda = 2$. This will provide two directed vertex-disjoint paths from r to each vertex v. Let E_H be the subset of edges in G such that one of its copies was chosen in H^D. We claim that the graph $H = (V, E_H \cup \{e'\})$ is 2-vertex connected (observe that r is not in H).

Proposition 3.1 *For any vertex v in G, there are paths $P(x, v)$ and $P(y, v)$ in H that are internally vertex disjoint.*

Lemma 3.2 *The graph $H = (V, E_H \cup \{e'\})$ is 2 vertex-connected.*

Proof. Suppose H contains a cut vertex a. Let the deletion of a from $H \cup \{e'\}$ breaks the graph into components $C_1, \ldots, C_k$. Since x and y are adjacent they will be in $a \cup C_i$ (for some i). W. l. o. g assume that x and y belong to $a \cup C_1$. Consider a vertex $v \in C_2$. Clearly, there cannot be two vertex disjoint paths from x and y to v. □

Theorem 3.3 *The total weight of $E_H \cup \{e'\}$ is at most $(2 + \frac{1}{n})$ times the optimal solution.*

Proof. Since every 2 vertex-connected graph contains at least n edges, the minimum weight edge in G is at most $\frac{1}{n}w(\mathcal{OPT}(G))$, where $w(\mathcal{OPT}(G))$ is the weight of a minimum weight 2 vertex-connected spanning subgraph.

Now consider an optimal solution $\mathcal{OPT}(G)$ for the problem. Consider all the anti-parallel edges corresponding to edges in $\mathcal{OPT}(G)$. We get a directed subgraph in G^D of weight $2w(\mathcal{OPT}(G))$. From x and y there are 2 vertex-disjoint paths to any vertex v; these also yield 2 directed paths from r to v that are also internally vertex-disjoint. Thus this subgraph has the property of having 2 directed vertex-disjoint paths from r to any vertex v. The optimum solution found by Frank and Tardos's algorithm can therefore only have lower weight. □

Remark: For the case when the edge weights satisfy triangle inequality, Khuller and Raghavachari [23] present algorithms using similar techniques that achieve a performance ratio of $2 + 2\frac{(\lambda-1)}{n}$.

3.2 Unweighted Vertex-Connectivity

Given an undirected graph G with n vertices and m edges, we would like to find a subgraph H that is λ vertex-connected and has as few edges as possible. In fact, Nagamochi and Ibaraki's algorithm (described earlier) finds a spanning subgraph with at most λn edges that has vertex-connectivity λ if and only if the original graph G has vertex connectivity λ. Since each vertex is required to have degree at least λ, we get that $\frac{\lambda n}{2}$ is a lower bound on any λ edge-connected spanning subgraph. This yields an approximation algorithm with a ratio of 2.

We now describe the algorithm due to Cheriyan and Thurimella [6]. The idea is to "peel" away maximal spanning forests from G, and to repeat this procedure λ times as was done for the edge connectivity case. To obtain a λ vertex-connected subgraph Cheriyan and Thurimella suggest the use of a forest obtained by running Breadth First Search from an arbitrary vertex in each connected component. Taking the union of these forests yields a λ vertex-connected subgraph. (A more efficient parallel implementation using a weaker notion of scan-first search was given by Cheriyan, Kao and Thurimella [5].) The proofs of the fact that this yields a λ vertex-connected subgraph is a little complicated. The reader is referred to the paper [6, 5].

3.2.1 2 Vertex-Connectivity

In this section we describe a simple method given by Khuller and Vishkin [28] that finds a 2 vertex-connected spanning subgraph by using Depth First Search. Combined with the edge discarding technique of Garg, Santosh and Singla [16] one obtains an approximation ratio of 1.5. Garg, Santosh and Singla [16] simplified and improved the algorithm due to Khuller and Vishkin [28] to yield an approximation factor of $\frac{3}{2}$. The algorithm is in two phases. The first phase is similar to the algorithm for the 2 edge-connectivity case described earlier. The second phase achieves two goals: (i) an attempt is made to expunge tree edges and (ii) an attempt is made to modify the choice of back edges so that it will help in expunging tree edges.

High-level Description of the Algorithm

The *first* phase is as follows: In the graph G, do a depth-first-search to compute a DFS spanning tree T. The idea is to now pick a set of back edges that will increase the vertex connectivity of the tree to two by "detouring" around each vertex of the tree T. During the Depth First Search all the tree edges are added to E_H, as well as some subset of back edges. The back edges are chosen when the DFS traversal is visiting a vertex for the last time. When DFS *retreats out of a vertex v for the last time*, we check if the vertex u (parent of v) is potentially a cut vertex or not. If yes, we can cover it by adding to E_H the *highest* going back edge from a descendant of v. (This will at least prevent the separation of v from $p(u)$ under the deletion of u.)

Before discussing the second phase, we define the notion of *carving* of a graph, and point out the key difference between tree-carving and carving.

Definition 2 *A* **carving** *of a graph is a partitioning of the vertex set V into a collection of subsets $V_1, V_2, \ldots, V_k$ with the following properties. Each subset constitutes a node of a rooted tree Γ. Each non-leaf node V_j of Γ has a special grey vertex denoted by $g(V_j)$ that belongs to $p(V_j)$. For every vertex $v \in V_i$, all the neighbours of v that are in ancestor sets of V_i belong to either*

1. *V_i, or*
2. *V_j, where V_j is the parent of V_i in the tree Γ, or*
3. *V_ℓ, where V_ℓ is the grandparent in the tree Γ. In this case however, the neighbour of v can only be $g(p(V_i))$.*

The neighbour of v is required to be either an ancestor of v or a descendant of v.

We will refer to the vertices of Γ as *nodes*, and the edges of Γ as *arcs*. The root vertex belongs to a special set called the root-set. The key difference between the carving and tree-carving is that in the latter edges are only allowed to go to the parent node. In a carving, edges are allowed to go to a single vertex in the grandparent node as well.

Given T, the DFS spanning tree, we will be interested in the following partition of the vertices of G, called the *DFS-tree partition*. Some recursive calls DFS(v, u) end by adding the back edge (savior$[v]$, low$[v]$) to E_H. For each such call DFS(v, u), "remove" the tree edge (u, v) from T; the resulting connected components of T (with some tree edges removed) provides the DFS-partition. Furthermore, T induces a *rooted tree structure* on the sets in the DFS-tree partition.

The proof of the following theorem is given in [28].

Theorem 3.4 *The DFS-tree partition yields a carving of G.*

In the *second* phase the carving is processed "top-down" (starting with the root set). At each step a modification is made to the choice of the back edge (going upwards) from a carving set. This method is able to delete some of the tree edges as it proceeds (a similar trick was used in [28] but was not powerful enough to give an approximation factor of $\frac{3}{2}$). The deletion of tree edges is justified by the following lemma [16].

Lemma 3.5 *Let G' be a 2 vertex-connected graph; C is a simple cycle in G', and $e = (u, v)$ is a chord in C. Then $G' - \{e\}$ is also 2 vertex-connected.*

The *parent* vertex of a *carving-set* S is the grey vertex of S. Each carving-set (except for the root-set) has a unique parent vertex.

During the "top-down" phase, each time we process a carving-set we either discard a tree edge, or find a new vertex to add to an independent set. Suppose $k - 1$ back edges were added in the

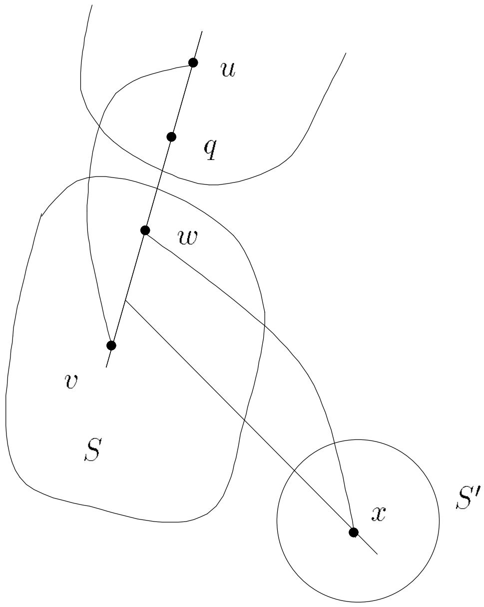


Figure 5: Figure for Case 1.

first phase (k is the number of sets in the carving). In the second phase, for each added back edge, we either discard a tree edge (so the back edge does not cost us anything) or we add a new vertex to the independent set. On termination, if we have p back edges remaining, we are able to find an independent set of size p. Clearly, $2p$ is a lower bound on the optimal solution since an independent set trivially yields a carving of size $p+1$ (by making each vertex of the independent set into a carving set, and all the other vertices into a single carving set).

We shall now jump into the guts of the second phase. When processing a carving-set we decide the back edges out of its child blocks. Consider a set S with the back edge out of S being (v, u) with $v \in S$. Consider the path in T from v to $q = g(S)$, the parent vertex (or grey vertex) of set S. Let w be the first vertex (excluding v) along this path that has no tree edge (other than the edges on the $v - q$ path) incident on it. If there is no such vertex then w is q.

If there is a back edge (x, w) with $x \in S'$, a child of S in Γ. Instead of picking the highest going back edge from S' we pick the back edge (x, w). For all other child blocks we do not modify the choice of back edges. Picking this back edge allows the deletion of the edge connecting w to its child in T. There are two cases:

Case 1 $w = p(v)$: observe that the edge (v, w) is a chord on the cycle $u - v - -x - w - -q - -u$ and can be deleted (see Fig. 5).

Case 2 $w \neq p(v)$: let (r', r) and (r, w) be the last two edges on the path in the DFS tree from v to w. We now have two cases: the first case is when x is a descendant of r'. By our assumption on w, there must be a tree edge incident on vertex r other than the ones going to r' and w. Assume that this tree edge goes to a child S'' of S in Γ. There must be a back edge (x', w') where $x' \in S''$ and w' is on the path from w to q in the DFS tree. The edge (r, w) is a chord on the cycle $r - -x - w - -w' - x' - -r$ and can be deleted (see Fig. 6). The second

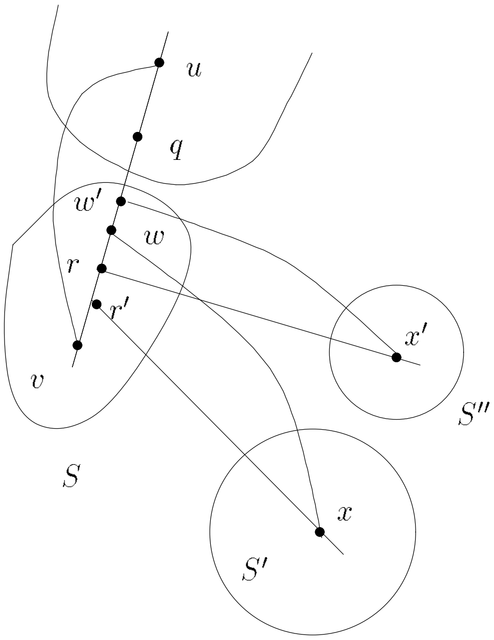


Figure 6: Figure for Case 2.

case is when x is not a descendant of r'. In this case the edge (r, w) is a chord on the cycle $w - -q - -u - v - -r - -x - w$ and can be deleted.

We can use this tree edge to account for the back edge emanating from the set S, and this back edge is thus paid for. We label the set S as "free".

If w has no back edges from any child block, we *mark* w and label the set S as "marked". The root-set has no back edge emanating from it, and is marked "free". In addition each leaf of the DFS tree forms a singleton set, and is marked and the set is labeled "marked". To argue that the set of marked vertices form an independent set, observe that no two marked vertices belong to the same set in the carving. Further, a marked vertex has no edge from any descendant set of the set that contains it. Thus no two marked vertices can have an edge between them.

Upon termination, if we have p "marked" sets, we have $n - 1 + p$ edges in our solution, and the lower bound on the optimal solution is $\max(n, 2p)$. It is easy to see that this is at most $\frac{3}{2}$.

Open Problems: The main open problem is to obtain a constant factor approximation when the edge weights do not satisfy the triangle inequality. Unlike the edge-connectivity case we do not know how to obtain factors *less* than 2, even for the unweighted case.

4 Strong-Connectivity Problems

Most of the network design literature addresses the problem of finding subgraphs having certain connectivities, in undirected graphs only. We now turn our attention to perhaps the simplest corresponding problem in directed graphs. Given a strongly connected directed graph, find a minimum strongly connected spanning subgraph (SCSS). Not surprisingly, this problem is NP-hard by a simple reduction from Hamilton Cycle in directed graphs. This problem was first studied by Frederickson and JáJá [11] for the weighted case, and a algorithm achieving an approximation factor of 2 was obtained. (This is obtained by taking the union of a minimum weight in-branching and a minimum weight out-branching, rooted at an arbitrary vertex.) For the unweighted case, Khuller, Raghavachari and Young [24] obtained an approximation algorithm with a performance ratio of about 1.64, which was improved to 1.61 [25]. The algorithms have a relatively high running time, albeit polynomial. An almost linear time algorithm that achieves a ratio of 1.75 is also described.

The MEG (minimum equivalent graph) problem is the following: "Given a directed graph, find a smallest subset of the edges that maintains all reachability relations between nodes." This problem is NP-hard; in fact, the heart of the MEG problem is the minimum SCSS (strongly connected spanning subgraph) problem — the MEG problem restricted to strongly connected digraphs. The MEG problem reduces in linear time [7] to a single acyclic problem given by the so-called "strong component graph", together with one minimum SCSS problem for each strong component (given by the subgraph induced by that component). Furthermore, the reduction preserves approximation, in the sense that c-approximate solutions to the subproblems yield a c-approximate solution to the original problem. Hence an approximation algorithm for the SCSS problem implies an approximation algorithm for the MEG problem. Moyles and Thompson [31] observe this decomposition and give exponential-time algorithms for the restricted problems. Hsu [21] gives a polynomial-time algorithm for the acyclic MEG problem.

First we describe the basic algorithm that achieves a factor of 1.64 in polynomial time. The algorithm and its analysis are based on the simple idea of contracting long cycles. After that we will describe the nearly linear-time algorithm that achieves a ratio of 1.75. To learn about the improvement to 1.61 the reader is referred to [25].

4.1 Polynomial Time Approximation Algorithms

Given a strongly connected graph, the basic algorithm finds as long a cycle as it can, contracts the cycle, and recurses. The contracted graph remains strongly connected. When the graph finally collapses into a single vertex, the algorithm returns the set of edges contracted during the course of the algorithm as the desired SCSS. The algorithm achieves a performance guarantee of any constant *greater* than $\pi^2/6 \approx 1.645$ in polynomial time.

A natural improvement to the cycle-contraction algorithm is to modify the algorithm to solve the problem optimally once the contracted graph has no cycles longer than a given length c. For instance, for $c = 3$, this modification improves the performance guarantee to $\pi^2/6 - 1/36 \approx 1.617$. We use SCSS_c to denote the minimum SCSS problem restricted to digraphs with no cycle longer than c. The minimum SCSS_2 problem is trivial. The minimum SCSS_3 problem can be solved in polynomial time, as shown by Khuller, Raghavachari and Young [25]. However, further improvement in this direction is limited: we show that the minimum SCSS_5 problem is NP-hard.

Before describing the algorithm we discuss some basic notation used in the rest of the section. To *contract* a pair of vertices u, v of a digraph is to replace u and v (and each occurrence of u or v in any edge) by a single new vertex, and to delete any subsequent self-loops and multi-edges. Each edge in the resulting graph is identified with the corresponding edge in the original graph or, in the case of multi-edges, the single remaining edge is identified with any one of the corresponding edges in the original graph. To contract an edge (u, v) is to contract the pair of vertices u and v. To contract a set S of pairs of vertices in a graph G is to contract the pairs in S in arbitrary order. The contracted graph is denoted by G/S. Contracting an edge is also analogously extended to contracting a set of edges.

Let $\mathcal{OPT}(G)$ be the minimum size of any subset of the edges that strongly connects G. In general, the term "cycle" refers only to simple cycles.

We begin by showing that if a graph has no long cycles, then the size of any SCSS is large.

Lemma 4.1 (Cycle Lemma) *For any directed graph G with n vertices, if a longest cycle of G has length $\mathcal{C}$, then*

$$\mathcal{OPT}(G) \geq \frac{\mathcal{C}}{\mathcal{C}-1}(n-1).$$

Proof. Starting with a minimum-size subset that strongly connects the graph, repeatedly contract cycles in the subset until no cycles are left. Observe that the maximum cycle length does not increase under contractions. Consequently, for each cycle contracted, the ratio of the number of edges contracted to the decrease in the number of vertices is at least $\frac{\mathcal{C}}{\mathcal{C}-1}$. Since the total decrease in the number of vertices is $n-1$, at least $\frac{\mathcal{C}}{\mathcal{C}-1}(n-1)$ edges are contracted. □

Note that the above lemma gives a lower bound which is existentially tight. For all values of $\mathcal{C}$, there exist graphs for which the bound given by the lemma is equal to $\mathcal{OPT}(G)$. Also note that $\mathcal{C}$ has a trivial upper bound of n and, using this, we get a lower bound of n for $\mathcal{OPT}(G)$, which is the known trivial lower bound.

Lemma 4.2 (Contraction Lemma) *For any directed graph G and set of edges S,*

$$\mathcal{OPT}(G) \geq \mathcal{OPT}(G/S).$$

Proof. Any SCSS of G, contracted around S (treating the edges of S as pairs), is an SCSS of G/S. □

The algorithm is the following. Fix k to be any positive integer.

```
CONTRACT-CYCLES_k(G) —
for i = k, k − 1, k − 2, ..., 2
    while the graph contains a cycle with at least i edges
        Contract the edges on such a cycle.
return the contracted edges
```

We will show that the algorithm runs in polynomial time for any fixed value of k. It is clear that the edge set returned by the algorithm strongly connects the graph. The following theorem establishes an upper bound on the number of edges returned by the algorithm.

Theorem 4.3 Contract-Cycles$_k(G)$ *returns at most $c_k \cdot \mathcal{OPT}(G)$ edges, where*

$$\frac{\pi^2}{6} \leq c_k \leq \frac{\pi^2}{6} + \frac{1}{(k-1)k}.$$

Proof. Initially, let the graph have n vertices. Let n_i vertices remain in the contracted graph after contracting cycles with i or more edges ($i = k, k-1, ..., 2$).

How many edges are returned? In contracting cycles with at least k edges, at most $\frac{k}{k-1}(n - n_k)$ edges are contributed to the solution. For $i < k$, in contracting cycles with i edges, $\frac{i}{i-1}(n_{i+1} - n_i)$ edges are contributed. The number of edges returned is thus at most

$$\frac{k}{k-1}(n-n_k) + \sum_{i=2}^{k-1} \frac{i}{i-1}(n_{i+1} - n_i) \leq \left(1 + \frac{1}{k-1}\right) n + \sum_{i=3}^{k} \frac{n_i - 1}{(i-1)(i-2)}.$$

Clearly $\mathcal{OPT}(G) \geq n$. For $2 \leq i \leq k$, when n_i vertices remain, no cycle has more than $i-1$ edges. By Lemmas 4.1 and 4.2, $\mathcal{OPT}(G) \geq \frac{i-1}{i-2}(n_i - 1)$. Thus the number of edges returned, divided by $\mathcal{OPT}(G)$, is at most

$$\frac{\left(1+\frac{1}{k-1}\right)n}{\mathcal{OPT}(G)} + \sum_{i=3}^{k} \frac{\frac{n_i-1}{(i-1)(i-2)}}{\mathcal{OPT}(G)} \leq \frac{(1+\frac{1}{k-1})n}{n} + \sum_{i=3}^{k} \frac{\frac{n_i-1}{(i-1)(i-2)}}{\frac{i-1}{i-2}(n_i-1)} = \frac{1}{k-1} + \sum_{i=1}^{k-1} \frac{1}{i^2} = c_k.$$

Using the identity (from [30, p.75]) $\sum_{i=1}^{\infty} \frac{1}{i^2} = \frac{\pi^2}{6}$, we get

$$\begin{aligned} \frac{\pi^2}{6} \quad \leq \quad c_k &= \frac{\pi^2}{6} + \frac{1}{k-1} - \sum_{i=k}^{\infty} \frac{1}{i^2} \\ &\leq \frac{\pi^2}{6} + \frac{1}{k-1} - \sum_{i=k}^{\infty} \frac{1}{i\,(i+1)} \\ &= \frac{\pi^2}{6} + \frac{1}{k-1} - \frac{1}{k} \\ &= \frac{\pi^2}{6} + \frac{1}{(k-1)k}. \end{aligned}$$

□

If desired, standard techniques can yield more accurate estimates of c_k, e.g., $c_k = \frac{\pi^2}{6} + \frac{1}{2k^2} + O\left(\frac{1}{k^3}\right)$. If the graph initially has no cycle longer than ℓ ($\ell \geq k$), then the analysis can be generalized to show a performance guarantee of $\frac{k^{-1}-\ell^{-1}}{1-k^{-1}} + \sum_{i=1}^{k-1} 1/i^2$. For instance, in a graph with no cycle longer than 5, the analysis bounds the performance guarantee (when $k=5$) by 1.424.

Table 1 gives lower and upper bounds on the performance guarantee of the algorithm for small values of k and in the limit as $k \to \infty$. The lower bounds are shown in [24].

k	Upper Bound	Lower Bound
3	1.750	1.750
4	1.694	1.666
5	1.674	1.625
∞	1.645	1.500

Table 1: Bounds on the performance guarantee.

For any fixed k, Contract-Cycles$_k$ can be implemented in polynomial time using exhaustive search to find long cycles. For instance, if a cycle of size at least k exists, one can be found in polynomial time as follows. For each simple path P of $k-1$ edges, check whether a path from the head of P to the tail exists after P's internal vertices are removed from the graph. If k is even, there are at most $m^{k/2}$ such paths; if k is odd, the number is at most $n\,m^{(k-1)/2}$. It takes $O(m)$ time to decide if there is a path from the head of P to the tail of P. For the first iteration of the for loop, we may have $O(n)$ iterations of the while loop. Since the first iteration is the most time consuming, the algorithm can be implemented in $O(n\,m^{1+k/2})$ time for even k and $O(n^2\ m^{(k+1)/2})$ time for odd k.

4.2 Nearly Linear-Time Implementation

We now describe a practical, near linear-time implementation of CONTRACT-CYCLES$_3$. The performance guarantee achieved is $c_3 = 1.75$. CONTRACT-CYCLES$_3$ consists of two phases: (1) repeatedly finding and contracting cycles of three or more edges (called *long* cycles), until no such cycles exist, and then (2) contracting the remaining 2-cycles.

High-level description of the algorithm

To perform Phase (1), the algorithm does a depth-first search (DFS) of the graph from an arbitrary root. During the search, the algorithm identifies edges for contraction by adding them to a set S. At any point in the search, G' denotes the subgraph of edges and vertices traversed so far. The rule for adding edges to S is as follows: when a new edge is traversed, if the new edge creates a long cycle in G'/S, the algorithm adds the edges of the cycle to S. The algorithm thus maintains that G'/S has no long cycles. When the DFS finishes, G'/S has only 2-cycles. The edges on these 2-cycles, together with S, are the desired SCSS.

Because G'/S has no long cycles and the fact that the original graph is strongly connected, G'/S maintains a simple structure:

Lemma 4.4 *After the addition of any edge to G' and the possible contraction of a cycle by adding it to S: (i) The graph G'/S consists of an outward branching and some of its reverse edges. (ii) The only reverse edges that might not be present are those on the "active" path: from the super-vertex containing the root to the super-vertex in G'/S containing the current vertex of the DFS.*

Proof. Clearly the invariant is initially true. We show that each given step of the algorithm maintains the invariant. In each case, if u and w denote vertices in the graph, then let U and W denote the vertices in G'/S containing u and w, respectively.

When the DFS traverses an edge (u, w) to visit a new vertex w:

Vertex w and edge (u, w) are added to G'. Vertex w becomes the current vertex. In G'/S, the outward branching is extended to the new vertex W by the addition of edge (U, W). No other edge is added, and no cycle is created. Thus, part (i) of the invariant is maintained. The super-vertex containing the current vertex is now W, and the new "active path" contains the old "active path". Thus, part (ii) of the invariant is also maintained.

When the DFS traverses an edge (u, w) and w is already visited:

If $U = W$ or the edge (U, W) already exists in G'/S, then no cycle is created, G'/S is unchanged, and the invariant is clearly maintained. Otherwise, the edge (u, w) is added to G' and a cycle with the simple structure illustrated in Fig. 7 is created in G'/S. The cycle consists of the edge (U, W), followed by the (possibly empty) path of reverse edges from W to the lowest-common-ancestor (lca) of U and W, followed by the (possibly empty) path of branching edges from lca(U, W) to U. Addition of (U, W) to G'/S and contraction of this cycle (in case it is a long cycle) maintains part (i) of the invariant. If the "active path" is changed, it is only because part of it is contracted, so part (ii) of the invariant is maintained.

When the DFS finishes visiting a vertex w:

No edge is added and no cycle is contracted, so part (i) is clearly maintained. Let u be the new current vertex, i.e., w's parent in the DFS tree. If $U = W$, then part (ii) is clearly maintained. Otherwise, consider the set D of descendants of w in the DFS tree. Since the original graph is strongly connected, some edge (x, y) in the original graph goes from the set D to its complement $V - D$. All vertices in D have been visited, so (x, y) is in G'. By part (i) of the invariant, the vertex

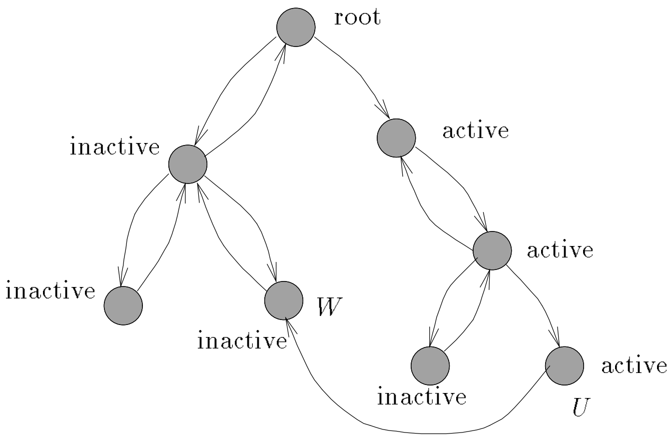


Figure 7: Contracted graph G'/S.

in G'/S containing x must be W, while the vertex in G'/S containing y must be U. Otherwise the edge corresponding to (x, y) in G'/S would create a long cycle. □

The algorithm maintains the contracted graph G'/S using a union-find data structure [36] to represent the vertices in the standard way and using three data structures to maintain the branching, the reverse edges discovered so far, and the "active path". When a cycle arises in G'/S, it must be of the form described in the proof of Lemma 4.4 and illustrated in Fig. 7. Using these data structures, the algorithm discovers it and, if it is long, contracts it in a number of union-find operations proportional to the length of the cycle. This yields an $O(m\alpha(m,n))$-time algorithm.

The vertices of G'/S are represented in union-find sets as follows:

MAKE-SET(v): Adds the set $\{v\}$ corresponding to the new vertex of G'/S.

FIND(v): Returns the set in G'/S that contains vertex v.

UNION(u, v): Joins into a single set the two sets corresponding to the vertices in G'/S containing G''s vertices u and v.

The data structures representing the branching, reverse edges, and the active paths, respectively are:

from-root$[W]$: For each branching edge (U, W) in G'/S, from-root$[W] = (u, w)$ for some $(u, w) \in (U \times W) \cap E$.

to-root$[U]$: For each reverse edge (U, W) in G'/S, to-root$[U] = (u, w)$ for some $(u, w) \in (U \times W) \cap E$.

to-active$[U]$: For each vertex U on the "active path" in G'/S, to-active$[U] = (u, w)$ where $(u, w) \in (U \times W) \cap E$ and W is the child of U for which the recursive DFS call is currently executing, unless no recursive DFS is executing, in which case to-active$[U] =$ **current**.

For all other vertices, to-active$[U] =$ **nil**.

```
CONTRACT-CYCLES_3(G = (V, E))   —   Pseudo-code.
S ← {}
Choose r ∈ V.
DFS(r)
Add 2-cycles remaining in G'/S to S.
return S

DFS(u)   —
to-active[FIND(u)] ← current
for each vertex w adjacent to u     — traverse edge (u, w) —
    if (w is not yet visited)     — new vertex —
        MAKE-SET(w)
        to-active[FIND(u)] ← from-root[FIND(w)] ← (u, w)
        DFS(w)
        to-active[FIND(u)] ← current
    else     — edge creates cycle in G'/S —
        if (FIND(u) ≠ FIND(w))     — cycle length at least 2 —
            (x, y) ← from-root[FIND(u)]
            if (FIND(x) = FIND(w))     — length two cycle through parent, U – W – U —
                to-root[FIND(u)] ← (u, w)     — record edge to parent —
            else
                (x, y) ← from-root[FIND(w)]
                if (FIND(x) ≠ FIND(u))     — not a forward edge to child; length of cycle ≥ 3 —
                    CONTRACT-CYCLE(w)
                    S ← S ∪ {(u, w)}
to-active[FIND(u)] ← nil
```

Figure 8: Practical implementation of CONTRACT-CYCLES$_3$.

Pseudo-code for the algorithm is given in Figures 8 and 9.

By the preceding discussion, the algorithm implements CONTRACT-CYCLES$_3$. It is straightforward to show that it runs in $O(m\alpha(m,n))$ time. Hence, we have the following theorem.

Theorem 4.5 *There is an $O(m\alpha(m,n))$-time approximation algorithm for the minimum SCSS problem achieving a performance guarantee of* 1.75 *on an m-edge, n-vertex graph.*

Here $\alpha(m,n)$ is the inverse-Ackermann function associated with the union-find data structure [36].
Open Problems: The main open problem is to obtain a performance ratio better than 2 for the weighted strong connectivity problem.

5 Connectivity Augmentation

Let $G = (V, E)$ be a graph with a non-negative weight function w on the edges. Let $G_0 = (V, E_0)$ be a subgraph of G. The goal is to add a minimum weight set of edges Aug, to G_0, such that the resulting graph is λ-connected for a given λ. We are permitted to only add edges from the graph G. For $\lambda > 1$, the problem is NP-hard. For $\lambda = 2$, an approximation algorithm that achieved a

```
CONTRACT-CYCLE(w)  —
while (to-active[FIND(w)] ≠ current) do
        if (to-active[FIND(w)] = nil) then     — Go up towards l. c. a. along reverse edges. —
            (c, p) ← to-root[FIND(w)]
            a ← to-active[FIND(p)]
        else     — Go down from l. c. a. along active path. —
                (p, c) ← to-active[FIND(w)]
                a ← to-active[FIND(c)]
          — Contract parent p and child c. —
        f ← from-root[FIND(p)]
        t ← to-root[FIND(p)]
        UNION(p, c)
        to-active[FIND(w)] ← a
        from-root[FIND(w)] ← f
        to-root[FIND(w)] ← t
```

Figure 9: Subroutine CONTRACT-CYCLE.

factor of 2 was given by Frederickson and JàJà [11] when G_0 is a connected graph. (If G_0 is not connected initially, we may add a minimum spanning tree to connect its connected components.) Here we present a simplification of the algorithm developed by Khuller and Thurimella [27]. We describe algorithms for both the edge and vertex connectivity problems. We also show that an approximation factor of 2 can be achieved in polynomial time for any λ. This is done by an extension of the algorithm described in Subsection 2.1.

We first describe some notation used in this section. The 2 vertex-connected components of a graph are also referred to as *blocks*. For a vertex v in a rooted tree Γ, let the components formed by the deletion of v be called $C_1(v), C_2(v), \ldots, C_{d(v)}(v)$, where $d(v)$ is the degree of v in Γ. If v is not the root, we will assume that $C_1(v)$ is the component that contains the root, and the other components correspond to subtrees rooted at the children of vertex v. In a rooted tree, for a vertex u we denote its parent by $p(u)$.

Notation: we refer to an undirected edge between two vertices x and y as (x, y). On the other hand, a directed edge from x to y is denoted by $x \rightarrow y$.

5.1 Increasing Edge Connectivity from 1 to 2

Notice that we only need to show how to increase the edge connectivity of a tree due to the following observation. If we are given G_0 with nontrivial 2 edge-connected components, then we can shrink the vertex sets of these components into single vertices, resulting in a tree whose edges are the bridges of G_0. The edges to be retained from *Feasible* are the minimum weight edges that connect vertices in different 2 edge-connected components of G_0. (Observe that the edges of *Feasible* that connect vertices of the same 2 edge-connected component are of no use in augmenting G_0. Similarly, among the edges that connect two distinct 2 edge-connected components only the minimum weight edge is of interest.)

From G_0, we will construct a directed graph G^D and find a minimum weight branching from a vertex r. (If there is no branching that spans all the vertices, we can show that there is no way to increase the connectivity of the network.) Using a minimum weight branching of G^D, we can find a set of edges of $G - G_0$ whose addition will increase the connectivity of G_0. We can also show that the total weight of the edges added by this technique is no more than twice the weight of an optimal augmentation.

The algorithm is as follows:

(1) (*Construct* $G^D = (V, E_D)$)

(a) Pick an arbitrary leaf r and root the tree G_0 at r by directing all the edges towards the root. Denote the resulting tree by Γ.

(b) Add to E_D the directed tree edges of Γ and set their weight to zero.

(c) Consider the edges that belong to $G = (V, E)$ but do not belong to G_0 (edges in $E - E_0$). For each such edge (u, v), if (u, v) is a back edge (i.e., it connects a vertex to one of its ancestors), we add one directed edge to E^D (shown below); otherwise, we add two directed edges to E^D. (We will refer to these directed edges as *images* of (u, v), and we say these directed edges are *generated* by (u, v).)

Suppose that the edge e with weight $w(e)$, joins vertices u and v belonging to the tree Γ. There are two cases depending on the relative locations of u and v in the tree Γ (see Fig 10).

(i) If u is an ancestor of v (the converse is symmetric): then add an edge $u \rightarrow v$ in G^D with weight $w(e)$.

(ii) If neither u nor v is an ancestor of the other: let $t = l.c.a(u, v)$ (least common ancestor in the rooted tree Γ). Add edges $t \rightarrow u$ and $t \rightarrow v$ in G^D, each with weight $w(e)$.

(2) Find a minimum weight branching in G^D rooted at r. For each directed edge e that is picked as part of the branching, and that does not belong to the directed tree Γ, add the corresponding edge in $E - E_0$ that generated e. The set of edges added is Aug.

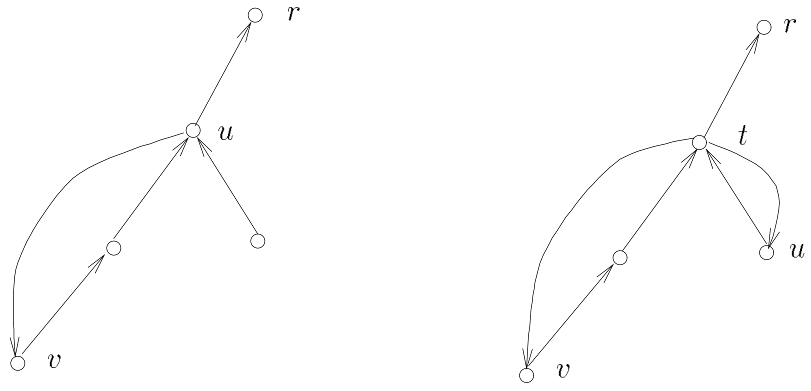
Observe that all edges of $G^D - \Gamma$ are such that they connect a vertex to one of its descendants in Γ.

Lemma 5.1 *If G is 2 edge-connected, then the directed graph G^D is strongly connected.*

Proof. Clearly, all the vertices of G^D can reach the root r using edges from the tree Γ. Further, let us assume that G^D is not strongly connected. Of all the vertices that cannot be reached from the root, let u be the vertex that is closest to the root in Γ. Clearly, the entire subtree rooted at u must consist of unreachable vertices. Since the image of the edge $(u, p(u))$ in G is not a bridge in G, there must be another edge (v, s) in G going from a vertex v that is in the subtree rooted at u, to vertex s that is not in this subtree.

Such an edge would have generated a directed edge from a vertex w to v in G^D where w is an ancestor of v (specifically the least-common-ancestor of v and s). Since w is a proper ancestor of u, it is reachable from r in G^D. Therefore v is reachable from r, and hence u as well. Thus we obtain a contradiction. $\square$

Lemma 5.2 *If G is 2 edge-connected, then the edge connectivity of the graph G_0 together with the edges in Aug is at least 2.*



(a) (b)

Figure 10: Construction of G^D.

Proof. Assume G is 2 edge-connected. Then by the previous lemma, we can find a minimum weight branching in G^D. Next, assume that despite the addition of the edges in Aug to G_0, the resulting graph has bridges. All such bridges are the tree edges in Γ. Let $(u, p(u))$ be one such edge of Γ that is closest to the root (it does not have to be unique). Since vertices in the subtree rooted at u, are reached from r in the branching it must be the case that there is a directed edge $w \rightarrow v$, from a vertex w (ancestor of u) to v in the minimum weight branching. Such an edge would have been generated by an edge connecting v to a vertex not in the subtree rooted at u. This edge would belong to Aug and hence the edge $(u, p(u))$ is not a bridge. □

Lemma 5.3 *The weight of Aug is less than twice the optimal augmentation.*

Proof. We prove the lemma by exhibiting a branching whose weight is at most twice the weight of the optimal augmentation. Consider the minimum weight set of edges Aug^* that would increase the connectivity from 1 to 2. Consider all the directed edges that are "generated" by edges that belong to Aug^*. These directed edges together with the tree edges yield a strongly connected graph with total weight on the edges at most $2C^*$ (each edge of weight w generated at most two directed edges, each of weight w). Hence the branching that we find has total weight at most $2C^*$. □

Theorem 5.4 *There is an approximation algorithm to find an augmentation to increase the edge connectivity of a connected graph to 2 with weight less than twice the optimal augmentation that runs in $O(m + n \log n)$ time.*

Proof. The correctness of the algorithm follows from Lemma 2 and Lemma 3. Since the bridge-connected components can be found in $O(m+n)$ time [3] and a minimum weight branching can be found in $O(m + n \log n)$ time [14]. Since the least common ancestors for the m pairs can be found in $O(m + n)$ time by using the algorithm of Harel and Tarjan [20], the theorem follows. □

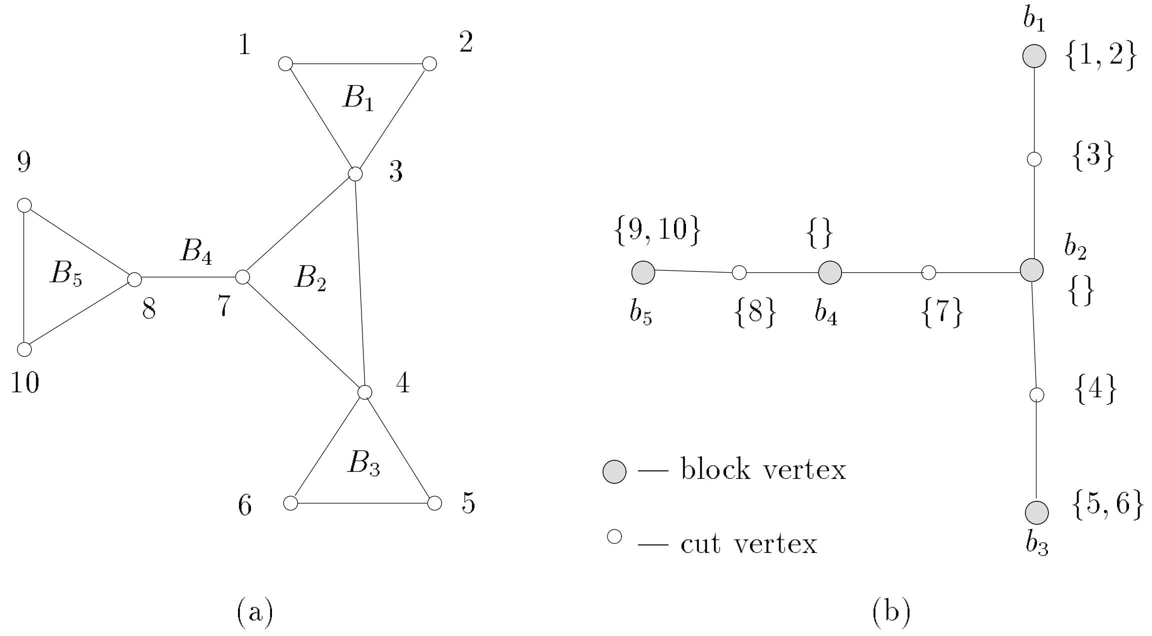


Figure 11: Construction of block cut vertex tree Γ.

5.2 Increasing Vertex Connectivity from 1 to 2

We can assume w.l.o.g. that G_0 is a connected graph just as in the case of edge connectivity. Our overall strategy is similar to the one used in the previous section. That is, we first obtain a tree structure Γ of the blocks of G_0, construct a weighted, directed graph G^D using Γ and G. Then find a minimum weight branching in G^D which will indicate the edges of $E - E_0$ that are to be added to increase the connectivity of G_0. We remark that Γ, in the case of vertex connectivity, is quite different from that of the previous section.

We first describe an algorithm to construct the block cut tree.

(1) Let $a_1, a_2, \ldots$ and $B_1, B_2, \ldots$ be the articulation points and blocks of $G_0 = (V, E_0)$, respectively. The vertex set $V(\Gamma)$ is a union of V_a and V_b where $V_a = \{a_1, a_2, \ldots\}$ and $V_b = \{b_i \mid B_i$ is a block of $G_0\}$. Associated with each vertex in $V(\Gamma)$, is a set. For $a_i \in V_a$, $X_i = \{a_i\}$. For $b_i \in V_b$, $Y_i = \{v_j \mid v_j \in V$ and v_j is not a cut vertex in $G_0\}$.

(2) The edge set $E(\Gamma)$ consists of edges (a_i, b_j) where a_i is an articulation point that belongs to block B_j.

Fig. 11 illustrates the above construction via an example.

Observation: In the block cut tree Γ, each edge is between a vertex in V_a and a vertex in V_b.

Observation: Consider the sets associated with the vertices of Γ. Each vertex of G_0 belongs to exactly one such set.

In the rest of the section, for a vertex u of V, the vertex of Γ that *corresponds* to u is u if u is an articulation point, and b_i otherwise where B_i is the unique block containing u. In the following, by *superimposing* an edge $(x, y) \in G$ on Γ, we mean adding an edge between $a, b \in V(\Gamma)$ where the associated sets X and Y contain x and y respectively.

The algorithm is as follows:

(1) Superimpose all the edges of $E - E_0$ on Γ. Discard all the self-loops. Among the multiple edges retain the cheapest edge, discarding the rest.

(2) (*Construct* $G^D = (V, E_D)$)
(a) Pick an arbitrary leaf of Γ to be the root r, and direct all the edges of Γ towards r. Continue to denote the resulting tree by Γ.

(b) Add to E_D the directed tree edges of Γ and set their weight to zero.

(c) Consider the superimposed edges of $E - E_0$ on Γ. Let (u, v) be one such superimposed edge. If (u, v) is a back edge (i.e. it connects a vertex to one of its ancestors), we add one directed edge to E^D (shown below); otherwise, we add four directed edges to E^D. (We will refer to these directed edges as *images* of (u, v), and we say these directed edges are *generated* by (u, v).)

Suppose that the edge e with weight $w(e)$, joins vertices u and v belonging to the tree Γ. There are two cases depending on the relative locations of u and v in the tree Γ (see Fig. 12).

(i) If u is an ancestor of v (the other case is symmetric): then add an edge $u \rightarrow v$ in G^D with weight $w(e)$.

(ii) If neither u nor v is an ancestor of the other: let $t = l.c.a(u, v)$ (least common ancestor in the rooted tree Γ). Add edges $t \rightarrow u$ and $t \rightarrow v$ in G^D, each with weight $w(e)$. Also add edges $u \rightarrow v$ and $v \rightarrow u$, each with weight $w(e)$.

(d) Modify E_D as follows. For every $u \in V_a$, if there is an outgoing edge from u to a descendant v, then replace that edge with $u_v \rightarrow v$ where where u_v is the child of u on the tree path from u to v.

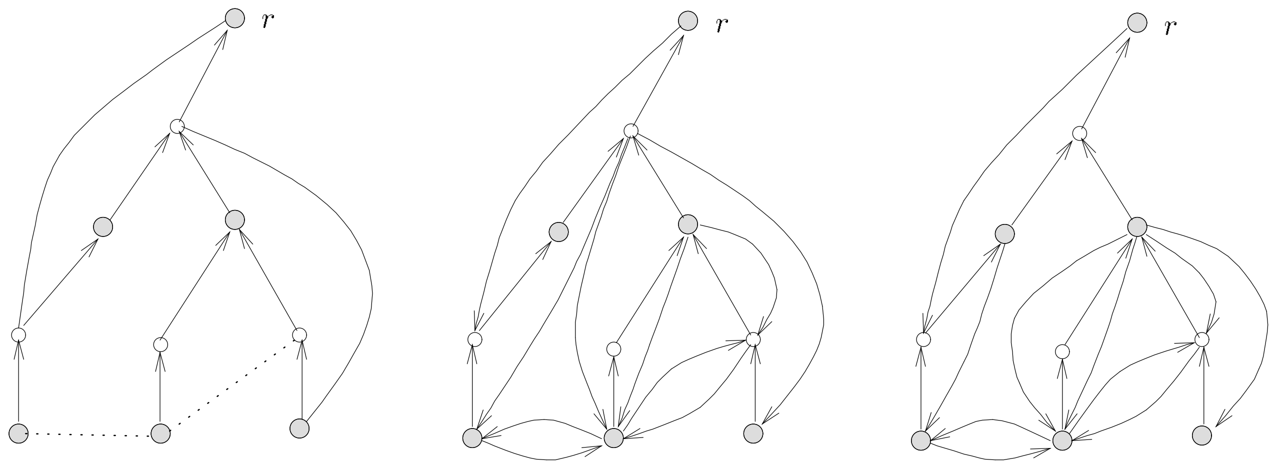
(3) Find a minimum weight branching in G^D rooted at r. For each directed edge e that is picked as part of the branching, and does not belong to the directed tree Γ, add the corresponding edge in $E - E_0$ that generated e. The set of edges added is Aug.

In the directed graph G^D there are no outgoing edges from a cut vertex to any of its descendants in Γ.

Observation: Consider the components formed on the deletion of a vertex $u \in V_a$ from Γ. The edges of G when superimposed on $\Gamma - u$ connect all these components.

Lemma 5.5 *If G is 2 vertex-connected, then the directed graph G^D is strongly connected.*

Proof. Clearly all the vertices of G^D can reach the root r using edges of the tree Γ. Let us assume that G^D is not strongly connected. Of all the vertices that cannot be reached from the root, let u be a vertex that is closest to the root in Γ. Clearly, the entire subtree rooted at u must consist of unreachable vertices and every proper ancestor of u is reachable from r. The proof is a little involved and we break it into cases.



(a) Rooted tree Γ and edges of $E - E_0$

(b) G^D after step 2(c)

(c) G^D after step 2(d)

Figure 12: Construction of G^D in the case of vertex connectivity.

Case 1: $u \in V_a$.

Since u is not a cut vertex in G, there must be at least one edge connecting a vertex in $C_1(u)$ to some vertex in $C_i(u)$ by the Observation. Let this edge be (v, s) where $s \in C_1(u)$ and $v \in C_i(u)$. Now there are two subcases to consider:

(a) s is an ancestor of v.

(i) $s \in V_a$.
Corresponding to edge (s, v) we added an edge in G^D, from the child s_v of s (on the path from s to v) to v. Since s_v is a block vertex, it is distinct from u. Clearly s_v is an ancestor of u, and hence reachable from r. Thus v is reachable from r and so is u, yielding a contradiction.

(ii) $s \in V_b$.
We add an edge in G^D from s to v. Since s is reachable from r (because it is an ancestor of u), so is v and hence u, yielding a contradiction.

(b) s is not an ancestor of v. Let $t = l.c.a(s, v)$.

(i) $t \in V_a$.
Corresponding to edge (s, v) we added an edge in G^D, from the child t_v of t (on the path from t to v) to v. Since $t_v \in V_b$, it is distinct from u. It is reachable from r and hence v is reachable from
r, and so is u, yielding a contradiction.

(ii) $t \in V_b$.
Clearly t is an ancestor of u, hence reachable from r. We added an edge in G^D from

t to v, hence v is reachable and u as well, yielding a contradiction.

Case 2: $u \in V_b$.

Consider the cut vertex $p(u)$. Notice that $p(u) \neq r$ since $r \in V_b$. Let the roots of the subtrees $C_1(p(u)), C_2(p(u)), \ldots C_k(p(u))$ be $r_1(= r), r_2, \ldots r_k$, where k is the degree of $p(u)$. Assume that $C_2(p(u))$ refers to the component containing u (hence $r_2 = u$). Partition the components into two groups as follows. The first group contains all the components whose roots are reachable from r, and the second group contains the rest. (Notice that both the groups are non-empty.) Since G is biconnected there must exist an edge (s, v) where s belongs to a vertex in $C_i(p(u))$ and v belongs to a vertex in $C_j(p(u))$, such that $C_i(p(u))$ and $C_j(p(u))$ belong to the first and second groups respectively.

(a) s is an ancestor of v.

(i) $s \in V_a$.
We added an edge from the child s_v of s to v in G^D. Since s_v is a block vertex, it is distinct from $p(u)$ and reachable from r. Hence v is reachable from r, and so is r_j giving a contradiction.

(ii) $s \in V_b$.
We added an edge in G^D from s to v. Since s is an ancestor of $p(u)$ it is reachable from r. Hence v is reachable from r, and so is r_j, giving a contradiction.

(b) s is not an ancestor of v. Let $t = l.c.a(s, v)$.

(i) $t \neq p(u)$.
There is an edge in G^D from either t or t_v, to v. Since both t and t_v are reachable from r, so is v and hence r_j, giving a contradiction.

(ii) $t = p(u)$.
Note that r_i is reachable from r. Because of edge (s, v) we generate the following edges in G^D: $r_i \to s, r_j \to v, s \to v, v \to s$. Hence v is reachable from r, and so is r_j, yielding a contradiction.

□

Lemma 5.6 *If G is 2 vertex-connected, then the vertex connectivity of the graph G_0 together with the edges in Aug is at least 2.*

Proof. Assume that despite the addition of the edges in Aug to G_0, the resulting graph has a cut vertex u. We will now show that u is destroyed as a cut vertex in the tree Γ, and hence in G_0. Consider the components $C_1(u), \ldots, C_{d(u)}(u)$ in Γ. Partition the components into two groups as follows. The first group contains all the components that get connected to $C_1(u)$ (by an edge or a path) when the edges of Aug are superimposed on Γ. The second group contains the rest. Notice that both the groups are non-empty. Since G^D is strongly connected all the vertices are reachable from the root in the minimum weight branching. Since there are no outgoing edges from u to its descendants by the previous observation, there must be an edge $s \to v$ in the branching that satisfies the following. This edge has the property that $s \in C_i(u)$ and $v \in C_j(u)$, where $C_i(u)$ and $C_j(u)$ belong to the first and second groups respectively. The edge that generated $s \to v$ in Aug would connect $C_i(u)$ to $C_j(u)$ in $G_0 + Aug$, yielding a contradiction. □

Lemma 5.7 *The weight of Aug is less than twice the optimal augmentation.*

Proof. We prove the lemma by exhibiting a branching whose weight is at most twice the weight of the optimal augmentation. Consider the minimum weight set of edges Aug^* that would increase the connectivity from 1 to 2. Consider all the directed edges that are "generated" by edges that belong to Aug^*.

These directed edges together with the tree edges yield a strongly connected graph with total weight on the edges at most $4C^*$ (each edge of weight w_i generated at most four directed edges, each of weight w_i). Now pick a minimum weight branching in this graph. Notice that for each cross edge (u, v) (when neither u nor v is an ancestor of the other) even though we generate four directed edges in G^D, no minimum weight branching will use more than two of these four edges. (Otherwise, it will not be a valid branching.) Hence the branching that we find has total weight at most $2C^*$. □

Theorem 5.8 *There is an approximation algorithm to find an augmentation to increase the vertex connectivity of a connected graph to 2 with weight less than twice the optimal augmentation that runs in $O(m + n \log n)$ time.*

Proof. The correctness of the algorithm follows from Lemma 5 and Lemma 6. Since the biconnected components can be found in $O(m + n)$ time [3] and a minimum weight branching can be found in $O(m + n \log n)$ time [14]. Since the least common ancestors for the m pairs can be found in $O(m + n)$ time by using the algorithm of Harel and Tarjan [20], the theorem follows. □

5.3 Increasing Connectivity to λ

We argue that it is possible to obtain an approximation factor of 2 for increasing the edge connectivity of a graph to any λ. The algorithm takes as input an undirected graph $G_0(V, E_0)$ on n vertices and a set *Feasible* of m weighted edges on V, and finds a subset *Aug* of edges which when added to G_0 make it λ-edge connected. The weight of *Aug*, is no more than twice the weight of the least weight subset of edges of *Feasible* that increases the connectivity. We also observe that the problem is NP-hard (for any λ) by extending the proof that was given by [11] for incrementing 1-connected graphs to 2-connected optimally.

Consider a directed graph G with weights on the edges, and a fixed root r. How does one find the *minimum weight* directed subgraph H^D that has λ-edge disjoint paths from a fixed root r to each vertex v ? Gabow [13] gives the fastest implementation of a weighted matroid intersection algorithm to solve this problem in $O(\lambda n(m + n \log n) \log n)$ time.

To solve our problem (approximation algorithm), in the undirected graph G_0 replace each undirected edge (u, v) by two directed edges $u \to v$ and $v \to u$ with each edge having weight 0. For each edge in the set *Feasible* (u, v) we replace it by two directed edges $u \to v$ and $v \to u$ with weight $w(u, v)$ (the weight of the undirected edge). Call this graph G^D. Now run Gabow's algorithm on the graph G^D, asking for λ-edge disjoint paths from each vertex to the root. If the directed edge $u \to v$ is picked in H^D and $w(u, v) > 0$ (we can assume all edges of set *Feasible* have weight > 0 else we can always include it in *Aug*) we add (u, v) to *Feasible*. (This is a generalization of the scheme for the case when E_0 is empty.)

Open Problems: The main open problem is to obtain factors better than 2 for the unweighted augmentation problem. Even simple greedy algorithms appear to have a performance ratio of 1.5.

Acknowledgements: I am grateful to Dorit Hochbaum and Balaji Raghavachari for useful comments. Support by NSF grant CCR-9307462 is gratefully acknowledged.

References

[1] A. Agrawal, P. Klein and R. Ravi, *When trees collide: An approximation algorithm for the generalized Steiner problem on networks*, Proc. 23rd ACM Symposium on Theory of Computing, pp. 134–144, (1991).

[2] A. V. Aho, M. R. Garey and J. D. Ullman, *The transitive reduction of a directed graph*, SIAM Journal on Computing, 1 (2), pp. 131–137, (1972).

[3] A. V. Aho, J. E. Hopcroft and J. D. Ullman, *The design and analysis of computer algorithms*, Addison-Wesley, (1974).

[4] S. Arnborg, J. Lagergren and D. Seese, *Easy problems for tree-decomposable graphs*, Journal of Algorithms, 12 (2), pp. 308–340, (1991).

[5] J. Cheriyan, M. Y. Kao and R. Thurimella, *Algorithms for parallel k-vertex connectivity and sparse certificates*, SIAM Journal on Computing, 22 (1), pp. 157–174, (1993).

[6] J. Cheriyan and R. Thurimella, *Algorithms for parallel k-vertex connectivity and sparse certificates*, Proc. 23rd Annual Symposium on Theory of Computing, pp. 391–401, (1991).

[7] T. H. Cormen, C. E. Leiserson, and R. L. Rivest, *Introduction to algorithms*, The MIT Press, (1989).

[8] J. Edmonds, *Matroid intersection*, Annals of Discrete Mathematics, 4, pp. 185–204, (1979).

[9] K. P. Eswaran and R. E. Tarjan, *Augmentation problems*, SIAM Journal on Computing, 5 (4), pp. 653–665, (1976).

[10] A. Frank, *Augmenting graphs to meet edge-connectivity requirements*, SIAM Journal on Discrete Mathematics, 5(1), pp. 25–53, (1992).

[11] G. N. Frederickson and J. JáJá, *Approximation algorithms for several graph augmentation problems*, SIAM Journal on Computing, 10 (2), pp. 270–283, (1981).

[12] A. Frank and E. Tardos, *An application of submodular flows*, Linear Algebra and its Applications, 114/115, pp. 320–348, (1989).

[13] H. N. Gabow, *A matroid approach to finding edge connectivity and packing arborescences*, Proc. 23rd Annual Symposium on Theory of Computing, pp. 112–122, (1991).

[14] H. N. Gabow, Z. Galil, T. Spencer and R. E. Tarjan, *Efficient algorithms for finding minimum spanning trees in undirected and directed graphs*, Combinatorica, 6 (2), pp. 109–122, (1986).

[15] M. R. Garey and D. S. Johnson, *Computers and intractability: A guide to the theory of NP-completeness*, Freeman, San Francisco, (1979).

[16] N. Garg, V. Santosh and A. Singla, *Improved approximation algorithms for biconnected subgraphs via better lower bounding techniques*, Proc. 4th Annual ACM-SIAM Symposium on Discrete Algorithms, pp. 103–111, (1993).

[17] M. Goemans and D. Bertsimas, *Survivable Networks, Linear Programming Relaxations and the Parsimonious Property*, Mathematical Programming, 60, pp. 145–166, (1993).

[18] M. Goemans and D. Williamson, *A general approximation technique for constrained forest problems*, Proc. 3rd Annual ACM-SIAM Symp. on Discrete Algorithms, pp. 307–316, (1992).

[19] F. Harary, *The maximum connectivity of a graph*, Proc. Nat. Acad. Sci., 48, pp. 1142–1146, (1962).

[20] D. Harel and R. E. Tarjan, *Fast algorithms for finding nearest common ancestors*, SIAM Journal on Computing, 13(2), pp. 338–355, (1984).

[21] H. T. Hsu, *An algorithm for finding a minimal equivalent graph of a digraph*, Journal of the ACM, 22 (1), pp. 11–16, (1975).

[22] T. S. Hsu, *Graph Augmentation and Related Problems: Theory and Practice*, Ph. D thesis, Dept. of Computer Science, University of Texas, Austin, TX (1993).

[23] S. Khuller and B. Raghavachari, *Improved Approximation Algorithms for Uniform Connectivity Problems*, manuscript (1994).

[24] S. Khuller, B. Raghavachari and N. Young, *Approximating the minimum equivalent digraph*, Proc. 5th Annual ACM-SIAM Symposium on Discrete Algorithms, pp. 177–186, (1994). To appear in *SIAM Journal on Computing*.

[25] S. Khuller, B. Raghavachari and N. Young, *On strongly connected digraphs with bounded cycle length*, UMIACS-TR-94-10/CS-TR-3212, (1994).

[26] S. Khuller and R. Thurimella. *Approximation algorithms for graph augmentation*, Proc. 19th International Colloquium on Automata, Languages and Programming Conference, pp. 330–341, (1992).

[27] S. Khuller and R. Thurimella. *Approximation algorithms for graph augmentation*, Journal of Algorithms, 14 (2), pp. 214–225, (1993).

[28] S. Khuller and U. Vishkin, *Biconnectivity approximations and graph carvings*, Journal of the ACM, 41 (2) pp. 214–235, (1994).

[29] P. N. Klein and R. Ravi, *When cycles collapse: A general approximation technique for constrained two-connectivity problems*, Proc. 3rd Integer Programming and Combinatorial Optimization Conference, pp. 39–56, (1993).

[30] D. E. Knuth, *Fundamental Algorithms*, Addison-Wesley, Menlo Park, CA, (1973).

[31] D. M. Moyles and G. L. Thompson, *An algorithm for finding the minimum equivalent graph of a digraph*, Journal of the ACM, 16 (3), pp. 455–460, (1969).

[32] D. Naor, D. Gusfield and C. Martel, *A fast algorithm for optimally increasing the edge-connectivity*, Proc. 31st IEEE Symposium on Foundations of Computer Science, pp. 698–707, (1990).

[33] H. Nagamochi and T. Ibaraki, *Linear time algorithms for finding sparse k-connected spanning subgraph of a k-connected graph*, Algorithmica, 7 (5/6), pp. 583–596, (1992).

[34] T. Nishizeki and S. Poljak, *Highly connected factors with a small number of edges*, to appear in *Discrete Applied Mathematics.*

[35] R. Ravi and D. Williamson, *An approximation algorithm for minimum-cost vertex-connectivity problems*, to appear in Proc. 6th Annual ACM-SIAM Symposium on Discrete Algorithms, (1995).

[36] R. E. Tarjan, *Data structures and network algorithms*, Society for Industrial and Applied Mathematics, (1983).

[37] R. Thurimella, *Techniques for the design of parallel graph algorithms*, Ph. D thesis, Dept. of Computer Science, University of Texas, Austin, TX (1989).