Biconnectivity Approximations and Graph Carvings *

Samir Khuller † *Uzi Vishkin* ‡

Abstract

A spanning tree in a graph is the smallest connected spanning subgraph. Given a graph, how does one find the smallest (i.e., least number of edges) 2-connected spanning subgraph (connectivity refers to both edge and vertex connectivity, if not specified) ? Unfortunately, the problem is known to be NP-hard.

We consider the problem of finding a better approximation to the smallest 2-connected subgraph, by an efficient algorithm. For 2-edge connectivity our algorithm guarantees a solution that is no more than $\frac{3}{2}$ times the optimal. For 2-vertex connectivity our algorithm guarantees a solution that is no more than $\frac{5}{3}$ times the optimal. The previous best approximation factor is 2 for each of these problems. The new algorithms (and their analyses) depend upon a structure called a *carving* of a graph, which is of independent interest. We show that approximating the optimal solution to within an additive constant is NP-hard as well.

We also consider the case where the graph has edge weights. For this case we show that an approximation factor of 2 is possible in polynomial time for finding a k-edge connected spanning subgraph. This improves an approximation factor of 3 for $k = 2$ due to [FJ81], and extends it for any k (with an increased running time though).

1. Introduction

Let a graph $G = (V, E)$ represent a feasible communications network. An edge (a, b) denotes the feasibility of adding a link from site a to site b. A spanning tree in G is the smallest connected subgraph, i.e., the cheapest network that will allow the sites to communicate. Notice that the 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.

In this paper we consider the problem of finding the smallest 2-connected spanning subgraph (edge or vertex connected). These problems are easily seen to be NP-hard by a reduction from the Hamilton cycle problem (the graph has a Hamilton cycle if and only if it has a 2-connected (edge or vertex) spanning subgraph with n edges). We will study approximation algorithms for this problem.

---

*Partially supported by NSF grants CCR-8906949, and CCR-9111348.

†The work of this author was done while this author was at UMIACS, and also supported by NSF grant CCR-9103135. Current address: Department of Computer Science, University of Maryland, College Park, MD 20742. Email:samir@cs.umd.edu.

‡University of Maryland Institute for Advanced Computer Studies (UMIACS), College Park, MD 20742 and Tel-Aviv University. Email:vishkin@umiacs.umd.edu.

Contributions:
We give linear time algorithms to find a subgraph $H = (V, E_H)$ in $G = (V, E)$, such that H is 2-connected. The number of edges in H is guaranteed to be no more than $c\ OPT$ where OPT is the size of an optimal solution. For the case of 2-edge connectivity we obtain $c = \frac{3}{2}$, and for the case of 2-vertex connectivity we obtain $c = \frac{5}{3}$. From the results presented, a natural question that arises is: what are the limits for the edge and vertex cases ? We also show that unless $P = NP$ there is no polynomial time algorithm that will produce a solution that is guaranteed to be of cost no more than $OPT + C$, where C is some constant.

For the case of weighted graphs we observe that an approximation factor of 2 is possible in $O(nk \log n(m + n \log n))$ time for finding the smallest weight k-edge connected subgraph based on algorithms by [G91a, FT89, Ed79]. This improves upon the approximation factor of 3 due to [FJ81] for $k = 2$.

In designing the approximation algorithms we identified the *carving* and *tree-carving* of a graph as structures that are useful for establishing lower bounds on the optimal solution. These notions might be of independent interest for understanding graph connectivity issues.

Significance of the Approximation Results:
Improving biconnectivity approximation factors can result in significant savings in the design of a physical network since it enables discarding a fraction of the edges that were needed before.

The algorithm has been implemented and tests on random graphs indicates that it achieves approximation factors in the range $1.1 \ldots 1.2$. The approximation factors got better as the graphs got denser; it went down to around 1.02 in the densest graphs that we checked.

Previous Approximation Results:
We note that an approximation factor of 2 is easy to obtain. Do a Depth First Search, and from each vertex (except the root) pick the highest back edge. This gives a 2-connected spanning subgraph with at most $2n - 2$ edges, while n is a trivial lower bound on any optimal solution. Other schemes for obtaining approximation factors of 2 follow as simple consequences of [Wh32, CT91, NI90].

Related Work:
The question of finding minimum cost k-connected spanning subgraphs can be posed in the context of weighted graphs as well. For the case $k = 1$, the problem reduces to that of finding a minimum spanning tree. For $k = 2$, the problem is NP-hard and a scheme that gave an approximation factor of 3 was given by [FJ81] in $O(n^2)$ time. This was made simpler and improved to obtain the same approximation factor in $O(m + n \log n)$ time by [KT93]. (Actually, the problem solved is of increasing the connectivity of an existing network from 1 to 2, but it can be used for an approximation factor of 3 as well.)

There is extensive literature on the k-edge connected spanning subgraph problem [FJ82, GB93, GMS92, MK89, SWK69], for the case where the edge weights satisfy the triangle inequality (and the underlying feasibility graph is a clique),

For the case of edge or vertex connectivity when the underlying feasibility graph is a *clique* (any edge can be added at unit cost), one can solve the problem of the smallest k-connected spanning subgraph optimally [Ha62].

For the case of increasing the edge connectivity of an existing network from any λ to k, when the underlying feasibility graph is a *clique* (any edge can be added at unit cost), the problem can be solved optimally [WN87, NGM90, G91b]. Naor, Gusfield and Martel [NGM90] use a clever extension of the basic DFS approach of [ET76] to generalize the technique to work for any k. (Eswaran and Tarjan solved the case of increasing connectivity from 1 to 2 in their seminal paper, where the problem was first introduced.) For the case of vertex connectivity, for $k = 2, 3$ the best algorithms are due to [ET76, RG77, HR91b] and [HR91a] respectively.

A more general edge connectivity problem was considered by Frank [Fr92] when the feasibility graph is a clique, and shown to be solvable in polynomial time. (Specifically, one is required to find a minimum spanning subgraph where specific connectivity requirements are given for each pair of vertices.) This result has recently been improved by Gabow [G91b].

The problem of finding a *minimal* (not minimum) 2-connected (edge and vertex) spanning subgraph was studied by [KR91] and [HKRT92]. (A graph with property P is *minimal* with respect to property P if it loses property P on deletion of any edge.) The relationship to this paper is that any minimal biconnected graph has at most $2(n-1)$ edges, which gives an approximation factor of 2.

Studying biconnectivity properties of graphs has led to a few fundamental graph algorithmic techniques.
(1) The power of Depth First Search was illustrated through biconnectivity [Ta72].
(2) The Tree Euler Tour technique for a parallel biconnectivity algorithm [TV85].
(3) The design of the Ear Decomposition Search (EDS) algorithm in [MSV86] was originally motivated by extending [Vi85] from a strong orientation algorithm into an alternative biconnectivity algorithm. Its use as a general technique for parallel graph algorithms came at a later stage. The fact that EDS yields an alternative biconnectivity algorithm is noted in [MR86] as well.
(4) Application of Graph Decompositions [GI91, Fr91] to dynamic 2-edge and 3-edge connectivity.

Improving approximation factors: Considerable attention has been given to improving constant approximation factors. For example, Johnson [Jo82] reports a series of 8 papers that give such improvements for `bin packing`, starting from an approximation factor of 2 down to 1.18333, and recently to $(1+\epsilon)$. For `steiner trees`, a similar series exists [KMB81, TM80, Ze93, BR92].

Outline of Paper:
Section 2 gives basic definitions related to edge and vertex connectivity. Section 3 describes the algorithm for the edge connectivity case (this section is very simple and gives the flavor of the results and analysis for the vertex case, which is more involved). Section 4 describes the algorithm for the vertex connectivity case. Section 5 describes the results for weighted graphs. In Section 6 we show that the problem of finding a constant additive approximation to the optimal solution is NP-hard.

2. Some Definitions

We will be dealing only with connected graphs $G = (V, E)$, with no parallel edges. A graph is said to be *k-vertex (k-edge) connected* if it has at least $(k+1)$ vertices (edges), and the deletion of any $(k-1)$ vertices (edges) leaves the graph connected. A single vertex in a connected graph whose deletion disconnects the graph is called a *cut vertex* (also known as *articulation vertex*). A graph with no cut vertices is called *biconnected.* A *bridge* in a graph is a single edge whose deletion disconnects the graph. A graph with no bridges is called *2-edge connected.* In a rooted tree, the *parent* of a vertex u is denoted by $p(u)$.

3. Edge Connectivity Case

Given a 2-edge connected graph $G(V, E)$, let OPT denote the minimum number of edges in a 2-edge connected spanning subgraph of G. We present an algorithm that finds a subgraph $H = (V, E_H)$ that is 2-edge connected with $\mid E_H \mid$ at most $\frac{3}{2}OPT$.

High-level Description of the Algorithm
We search G using depth-first-search (DFS). A DFS rooted spanning 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 threatens to remain a 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.

3.1. 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);
                    (* 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
```

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

In Fig 1(a), the vertices are shown numbered with the DFS numbering. The back edges are added in the following order: $(6,4),(7,3),(9,2),(3,1)$.

3.2. 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.

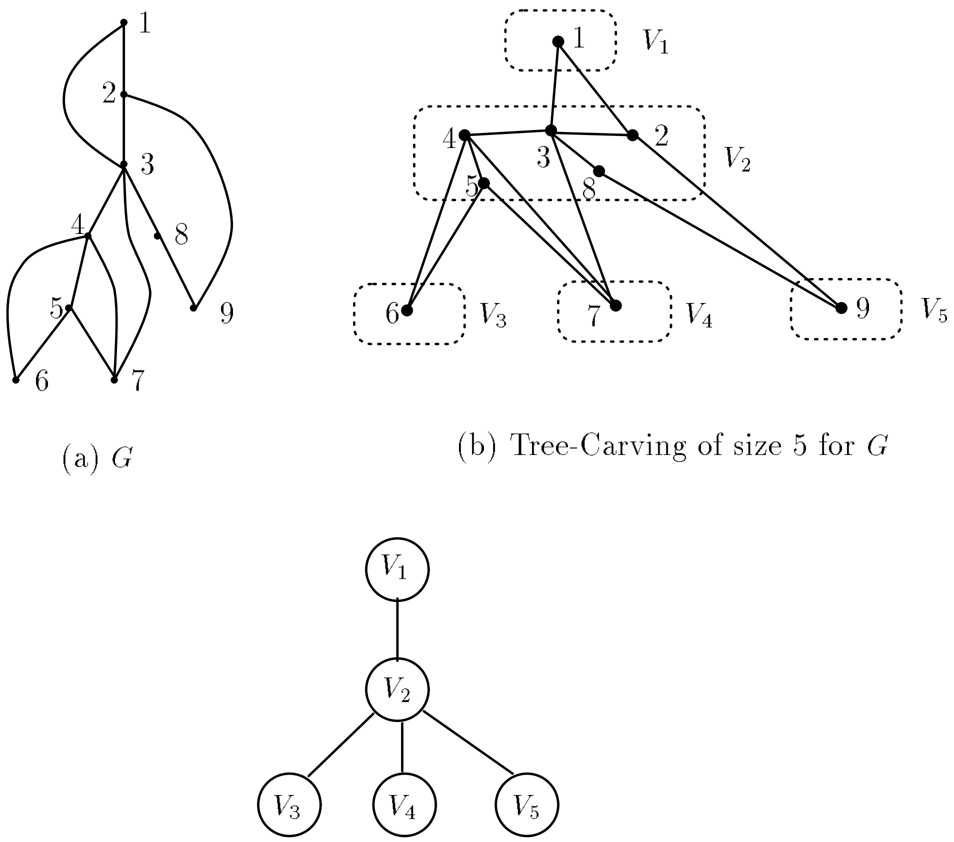
3.2.1. Tree-Carving

Definition 3.1: *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_k where V_k 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*.

An example of a graph G, a tree-carving for it, and its carving tree is shown in Fig. 1.

Theorem 3.2: (Tree-Carving Theorem)
If the graph $G=(V,E)$ has a tree-carving of size k, then a lower bound on the number of edges of any



(a) G

(b) Tree-Carving of size 5 for G

(c) The tree Γ

Figure 1: A graph G, a tree-carving for it, and the carving tree Γ.

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 (for $\lambda > 0$).

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 $2(k-1)$. □

3.2.2. Using Tree-Carvings for the Approximation Analysis

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 Fig 1(a), the vertices are shown numbered with the DFS numbering. The back edges are added in the following order: $(6,4), (7,3), (9,2), (3,1)$. Now consider the tree T, and remove the following tree edges: $(5,6), (5,7), (8,9), (1,2)$. This gives us the DFS-tree partition.

Theorem 3.3: *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$[v_2] \le v_1$. 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). □

Corollary 3.4: *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 $OPT \ge 2(k-1)$, where $k-1$ is the number of added back edges.*

Theorem 3.5: *The algorithm outputs a solution of size no more than $\frac{3}{2}$ OPT.*

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 $n-1+k-1$. A lower bound on the OPT solution 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 3.4. Hence, the ratio of the algorithm's solution to the OPT solution is

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

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

We have an example to show that the ratio of $\frac{3}{2}$ is asymptotically tight.

3.2.3. Worst Case Example

We provide an example (see Fig. 2) with $n = 16$, where the algorithm outputs E_H with $15 + 8 = 23$ edges. The optimal solution has 16 edges. The figure describes two copies of the graph G; in each copy not all the edges are shown. The left copy shows only the 15 tree edges and the 8 back edges that are added by the algorithm (highest ones). In the right copy, the 16 edges that form a Hamiltonian cycle are shown along with the tree edges.

Clearly we can generalize the example into a graph with n vertices that has $n/2$ leaves attached to $n/2$ vertices in a path (like a "broom"). In this case the ratio will be $\frac{n-1+n/2}{n}$, which converges to $\frac{3}{2}$.

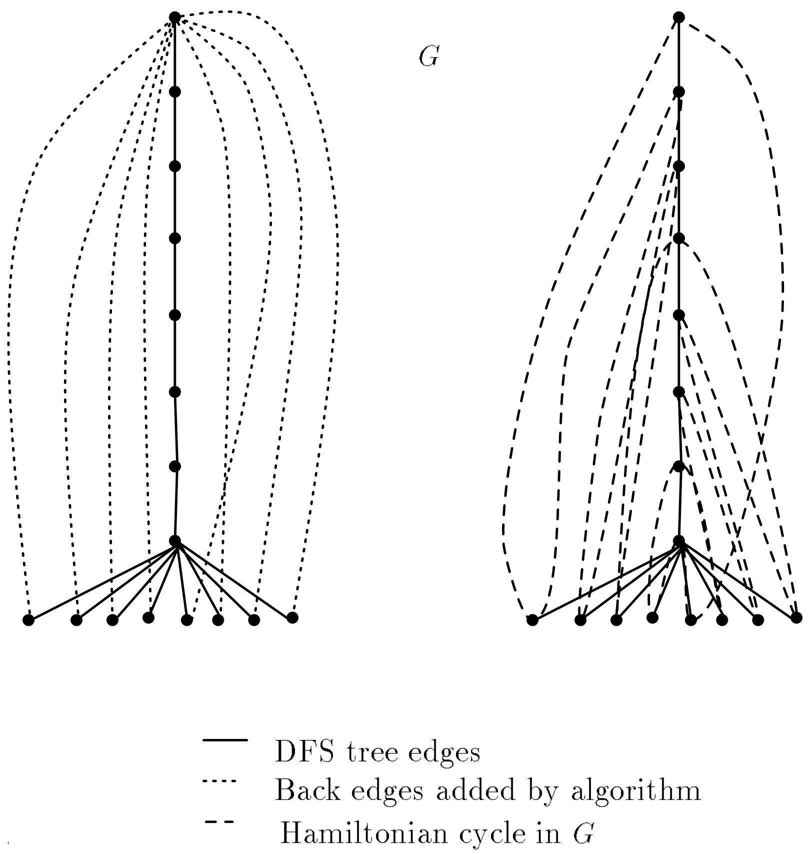


Figure 2: Example to illustrate worst case performance.

4. Vertex Connectivity Case

We now describe the algorithm that finds a 2-vertex connected spanning subgraph $H = (V, E_H)$, of a given 2-vertex connected graph G, with $\mid E_H \mid$ at most $\frac{5}{3}OPT$. To motivate the presentation we start by applying the "greedy" approach of the 2-edge connectivity approximation algorithm of the previous section to the example in Fig. 3. The DFS tree is the straight path and it is easy to see that each back edge must be added to cover a vertex that threatens to be a cut vertex. The graph shown is actually Hamiltonian, and clearly the example can be extended to yield approximation factors that are as close to 2 as we want. For this reason we design the algorithm to identify redundant edges in the DFS tree and *discard* them.

For the analysis of the algorithm, we will define the notion of a *carving* of a graph, which is not as simple as the tree-carving that worked for the edge connectivity case.

High-level Description of the Algorithm
We first provide an overview of the algorithm. 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. Some of the tree edges may be identified as redundant and *discarded* during the DFS. 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.) This is all that is done if v is a leaf in the DFS tree; otherwise, if the back edge emanates from v we discard the tree edge joining v with u (see Fig. 4). This is called the *discarding rule*.

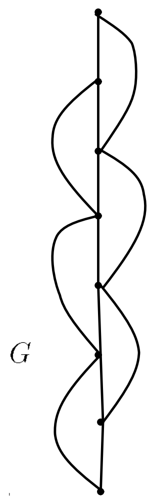


Figure 3: Example to show the necessity of discarding.

4.1. 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. 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 is marked "*discovered*". When we finally exit from the vertex it is marked "*finished*".

low$[v]$**:** 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.

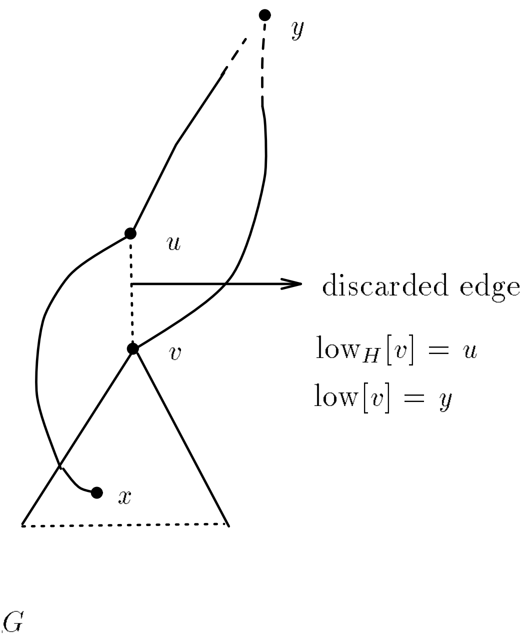


Figure 4: Rule for discarding tree edges.

low$_H[v]$: the vertex with the smallest dfs-number that can be reached by following zero or more down-going tree edges from v, and a *single* back edge that belongs to E_H. Since tree edges are being discarded, this definition may appear to be temporal. Path $Path_D(v)$ in Lemma 4.2 justifies this definition.

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-vertex connected graph (easy to verify this before running the algorithm). Initially all vertices are "unvisited".

Algorithm *Find 2-VC Spanning Subgraph*

Input: Graph $G = (V, E)$.

Output: A subgraph $H = (V, E_H)$ that is 2-VC.

```
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);
                (* else (v, w) is already a tree edge *)
            (* else w is finished and is a descendant of v *)
        end
    mark v finished;
    If u ≠ nil then begin
        If low_H[v] = v then begin (* if v is a leaf then add highest back edge *)
            E_H = E_H ⋃{( savior[v], low[v] )};
            low_H[v] = low[v];
        end
        If low_H[v] = u then begin (* v is a non-leaf in the DFS tree *)
            E_H = E_H ⋃{( savior[v], low[v] )}; (* add highest back edge *)
            low_H[v] = low[v];
            If savior[v] = v (* the discarding rule *) then
                E_H = E_H − {(u, v)};       (* delete the tree edge *)
        end
    end
end DFS
```

Complexity:
It is quite easy to see that the algorithm runs in time $O(n+m)$.

4.2. Correctness of the Algorithm

The correctness of the algorithm is established via the following theorem.

Theorem 4.1: *The subgraph $H = (V, E_H)$ obtained by the algorithm is 2-vertex connected.*

Proof: For the proof, it will be helpful to think of the algorithm as working in two phases.

Phase 1: Traverse the graph using DFS, add all the tree edges to E_H.

Phase 2: Traverse the graph using DFS, add the required back edges to E_H, and discard the tree edges by applying the discarding rule.

We first prove two lemmas that are used in the proof of the theorem.

Lemma 4.2: *For each non-root vertex v, the following paths exist in H, from the end of Phase 1 until the end of the algorithm.*

1. *$Path_U(v)$: from v to its parent $p(v)$, using vertices outside the DFS subtree rooted at v (except for v itself).*
2. *$Path_D(v,x)$: from v to each x that is a descendant of v, using only vertices in the subtree rooted at v. By $Path_D(v)$, we will refer to the set of the paths $Path_D(v,x)$, for all descendants x.*

Proof: The Lemma clearly holds before the starting of Phase 2. $Path_U(v)$ is the single tree edge joining v to $p(v)$. $Path_D(v,x)$ is the unique tree path in T from v to its descendant x.

The proof is by induction on the order in which the tree edges are being discarded.

We will show that as tree edge $(w, p(w))$ is discarded, we can alter the paths $Path_U(v)$ and $Path_D(v, x)$, for every v and every x in the subtree of v, so that they do not use the deleted edge. The inductive hypothesis is that prior to discarding the tree edge $(w, p(w))$, paths $Path_U(v)$ and $Path_D(v, x)$ exist for every v, and every x in the subtree of v. There are three cases to consider:

1. $v = w$. $Path_D(v)$ is unaffected. $Path_U(v)$ takes the back edge $(v, \mathrm{low}[v])$, and the path from low$[v]$ to $p(v)$ in T (that still exists since we have not yet discarded edges that high in the tree).
2. w is a proper ancestor of v. $Path_D(v)$ is unaffected. Assume that $Path_U(v)$ uses $(w, p(w))$. The deleted edge can be replaced by the path $Path_U(w)$ that was obtained in the previous case.
3. v is a proper ancestor of w. $Path_U(v)$ is not affected. Assume that some $Path_D(v, x)$ uses $(w, p(w))$. There must be a back edge from a vertex y to $p(w)$ since the edge $(w, p(w))$ was discarded, where y is a descendant of w. We can replace the edge $(w, p(w))$ by the path $Path_D(w, y)$, which exists by the inductive hypothesis, and the back edge $(y, p(w))$.

This completes the proof of the lemma. □

Lemma 4.3: *Let v be a vertex, and suppose that neither v nor $p(v)$ are the root. When the algorithm terminates, there is a path from v to its grandparent w (i.e., $w = p(p(v))$) that does not use $p(v)$.*

Proof: Observe that low$_H[v] < p(v)$ when the algorithm terminates. Let the back edge added by the algorithm to low$_H[v]$ be $(u, \mathrm{low}_H[v])$ ($u = \mathrm{savior}_H[v]$). We define savior$_H[v]$ to be the descendant end vertex of the back edge that goes to low$_H[v]$. We are ready to specify a path from v to w that does not use $p(v)$. Using $Path_D(v, u)$ and $(u, \mathrm{low}_H[v])$ we can advance from v to low$_H[v]$. We can get from w to low$_H[v]$ as follows: concatenate $Path_U(w)$, with $Path_U(p(w))$, and so on until we reach low$_H[v]$. Clearly $p(v)$ is not used on this path. □

We complete the proof of the theorem by showing that no single vertex can disconnect H. Observe that the root of the DFS tree cannot be a cut vertex. (Since G is biconnected, the root has only one child v and using $Path_D(v)$, v can reach all the vertices, without using the root.) Let v be a non-root vertex. The following will prove that H remains connected on deletion of v, by showing that every remaining vertex has a path to $p(v)$. Consider deleting v from T. We obtain a connected component corresponding to each child of v, and one corresponding to the parent (that contains the root). (1) Let u be a child of v. Using $Path_D(u)$, clearly u is connected (in H) to all vertices in its subtree in T. Using Lemma 4.3, we can connect each such u with $p(v)$. (2) We now consider the component of T containing $p(v)$. Let the path from $p(v)$ to the root in T be $Q = [v_1 = p(v), v_2, \ldots v_k = root]$. Clearly $p(v)$ is connected (in H) to all of its ancestors on path Q by using $Path_U(v_1)$ to v_2, and $Path_U(v_2)$ to v_3 etc. Consider a vertex x (other than v), that is a child of some v_i. Using $Path_D(x)$ paths, x is connected (in H) to all the vertices in its subtree in T, including savior$_H[x]$. Using the back edge (savior$_H[x]$, low$_H[x]$), x can connect to the path Q, and thereby to $p(v)$, as well. □

4.3. The Approximation Analysis

We would first like to motivate the need for a slightly different structure than the one in the previous section, by showing the short-comings of the tree-carving in handling the vertex connectivity case. Consider the graph shown in Fig. 5. It consists of ℓ units of 4 vertices each. The "root" of each unit is connected to v in the DFS tree, and v is connected to r. Clearly the number of back edges added by the algorithm equals 3ℓ. It should be clear that we cannot find a tree-carving of size greater than $2\ell + 2$. The 2ℓ leaves form singleton sets in the carving, and since they all have edges to v, the set containing v in the tree-carving contains all the other vertices (except for r). Since $n = 4\ell + 2$, and the number of added edges is 3ℓ, we get a ratio of $\frac{7\ell+1}{4\ell+2}$, and this is not as good as we would like to claim.

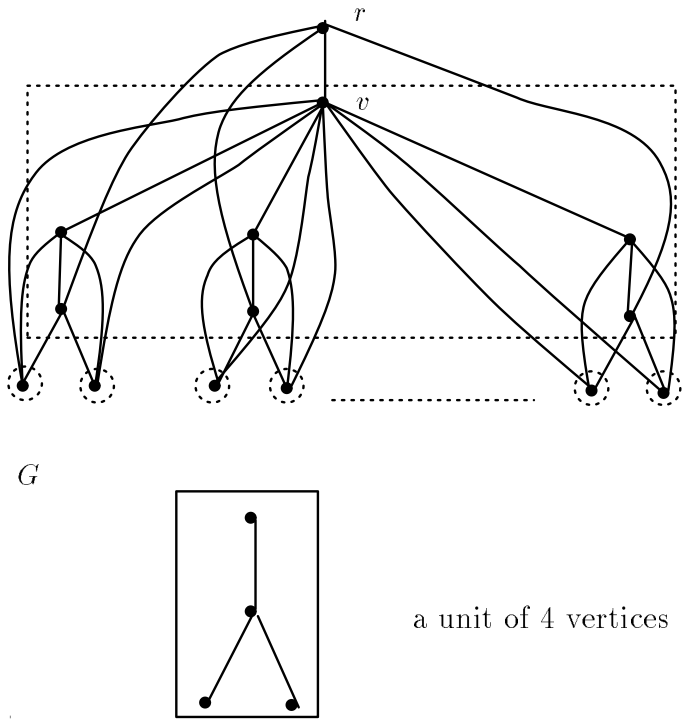


Figure 5: Example to show short-comings of tree-carving.

Our analysis finds a partition of the vertices, called a *carving*, which is used to prove a lower bound on OPT, the number of edges in the optimal solution. The upper bound of $\frac{5}{3}$ on the approximation factor is established using this lower bound. After presenting the concept of a carving, we apply it to the approximation analysis.

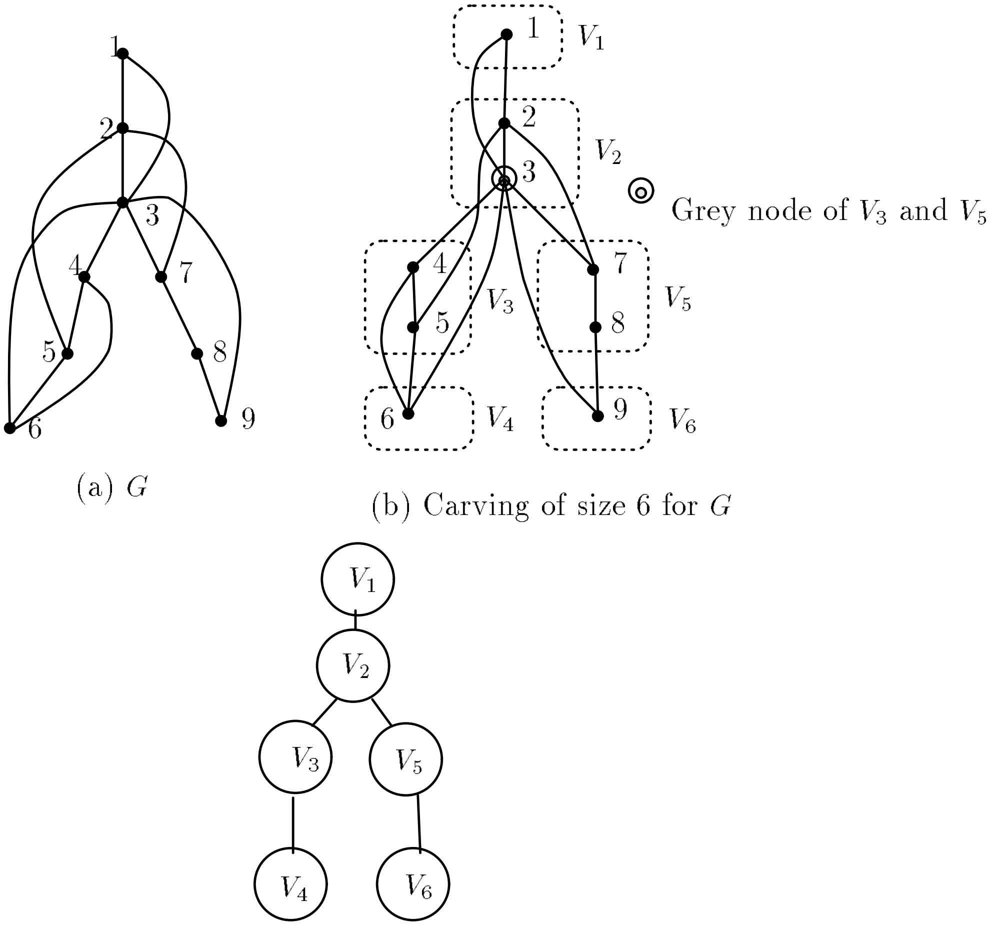
4.3.1. Carving

Definition 4.4: *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.*
2. *V_j, where V_j is the parent of V_i in the tree Γ.*
3. *V_ℓ, where V_ℓ is the grandparent in the tree Γ. In this case however, the neighbour of v can only be $g(V_j)$.*

The neighbour of v is required to be either an ancestor of v or a descendant of v. The **size** *of the carving is k.*

We will refer to the vertices of Γ as *nodes*, and the edges of Γ as *arcs*. An example of a graph G together with a carving for it is shown in Fig. 6.



(a) G

(b) Carving of size 6 for G

(c) The tree Γ

Figure 6: A graph G, a carving for it, and the carving tree Γ.

Theorem 4.5: (Carving Theorem)
If the graph $G = (V, E)$ has a carving of size k with ℓ leaves in Γ, then a lower bound on the number of edges of any 2-vertex connected spanning subgraph in G is $(k + \ell - 1)$.

Proof: Consider the rooted tree Γ. Each node of the tree other than the root has a unique parent node. Consider any leaf node X; *Claim* (1): in any 2-vertex connected spanning subgraph there must be at least two edges with exactly one endpoint in X. This implies that there are at least 2ℓ edges in the OPT solution. Now consider any non-leaf node X that is not the root. *Claim* (2): in any 2-vertex connected spanning subgraph there must be at least one edge that has one endpoint in X and the other in an ancestor set of X. (Some vertices that belong to the children sets of X may have edges going to $g(X)$, but since $g(X)$ is not a cut vertex, Claim (2) follows.) There is no overlap in the counting scheme for different sets. There are $k-\ell-1$ nodes satisfying Claim (2) in the tree Γ. This shows that $2\ell+(k-\ell-1)$ is a lower bound on the size of *any* 2-vertex connected spanning subgraph. □

4.3.2. Using Carvings for a Lower Bound on OPT

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 *without discarding* any tree edge. (These are the recursive calls that cause a net increase in the number of back edges.) 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.

Let ℓ denote the number of back edges emanating from the leaves, and x denote the net increase due to the other back edges. The net increase in the number of edges is $\ell + x$.

Theorem 4.6: *The DFS-tree partition yields a carving of G.*

Before describing the proof, we give some definitions that make the proof clearer. The algorithm adds an edge to E_H just before it leaves v for the last time and discovers that $\text{low}_H[v] = v$ (which occurs if and only if v is a leaf), or discovers that $\text{low}_H[v] = u$ (which implies that u is threatening to be a cut vertex). In the first case we add to E_H the highest back edge from the leaf v, and create a singleton leaf set in the DFS-tree partition. In the second case (only when no tree edge is discarded), we pick the highest back edge from a descendant of v, and define a new set for the DFS-tree partition. The *grey* vertex of this set is defined to be u. (Recall that the grey vertex is a vertex in the parent set.)

Definition 4.7: *Given a set of vertices V_i, its root is the vertex with the smallest dfs number in the set. It is denoted as root(V_i).*

Proposition 4.8: *The root of a set in the carving is the vertex v such that the tree edge $(v, p(v))$ was removed from the DFS tree.*

Proof (Of Theorem 4.6): Color red all the vertices that are roots of a set in the carving. Each vertex joins the set corresponding to its closest red ancestor (in the DFS tree). It is clear that this gives us a "tree" structure on the sets. Consider a non-tree edge $e = (v, w)$. Since there are no cross edges assume that w is an ancestor of v. Assume that $v \in V_i$. We need to prove that the endpoint w either (1) belongs to the same set V_i, (2) to the parent set V_j, (3) or w is the grey vertex of V_j, in the parent set of V_j.

Suppose that w is in set V_h such that V_h is neither V_i, nor V_j. So V_h is a proper ancestor of V_j in Γ. The algorithm picked the highest going back edge when marking root(V_i) finished. This must go to a vertex x with $x \leq w$. Hence when marking $v' = \text{root}(V_j)$ finished, $\text{low}_H[v'] \leq x$. The vertex v' can be colored red only if $\text{low}_H[v'] = p(v') = x = w$, in which case w is the grey vertex of V_j.

□

Corollary 4.9: (Lower Bound)
The number of arcs in the carving is exactly the same as the net number of back edges that are added to E_H ($k-1 = \ell + x$), and the number of leaves in the carving tree is the same as the number of back

edges added from leaf vertices. Thus we conclude that $OPT \geq 2\ell + x$, where x is the net increase due to non-leaf back edges and ℓ is the number of leaves in the carving tree Γ.

Proof: Corresponding to each back edge we have one set in the carving and thus $k - 1 = \ell + x$. Each leaf is put in a singleton leaf set in the tree Γ, and since a back edge is added from each leaf the claim follows. Substituting for k in Theorem 4.5 yields $OPT \geq 2\ell + x$. □

A lower bound for the OPT solution is $\max(n, 2\ell + x)$. (The bound of n edges follows from a degree argument since each vertex should have degree at least 2. The other bound follows from Theorem 4.5 and Corollary 4.9.)

4.3.3. An Upper Bound on $| E_H |$

The number of edges added by the algorithm is as follows: first $(n - 1)$ edges (for the tree edges), then an extra $(\ell + x)$ edges (this denotes the net increase).

Theorem 4.10: *The number of edges added by the algorithm (net increase) can be upper bounded by $\frac{1}{2}(n + \ell)$ where n is the number of vertices and ℓ is the number of leaves in the DFS tree T.*

Notice that this gives an upper bound of $\frac{1}{2}(n - \ell)$ for x (the net increase $= \ell + x$).

Proof: We prove this theorem by a simple charging scheme. The back edges that are added can be partitioned naturally into three categories.

1. (Type A) Back edges that emanate from a leaf v in T.
2. (Type B) Back edges that emanate from a non-leaf v in T, and delete the tree edge $(v, p(v))$ when they are added.
3. (Type C) Back edges that emanate from a non-leaf v in T, and do not delete any tree edges when they are added.

It is clear that the net increase in the number of edges is due only to edges of types A and C. We give a simple proof to upper bound the number of added edges. For each edge of type A emanating from a leaf v, we put a charge of 1 to v. For each edge of type C emanating from a vertex v, we put a charge of $\frac{1}{2}$ to v and a charge of $\frac{1}{2}$ to $p(v)$. The following lemma shows that no non-leaf vertex can get charged more than once.

Lemma 4.11: *By this process each non-leaf vertex gets a charge of at most $\frac{1}{2}$.*

Proof: Suppose that there is a vertex v that gets charged more than once due to edges of type C being added. There are two cases; either it could get charged due to back edges emanating from children u_1 and u_2, or it could get charged due to a back edge emanating from v and a back edge emanating from a child u_1 (all of type C).

First notice that if any u_i is a leaf vertex, then it would not charge v. Hence we can assume that the u_i's are non-leaf vertices and hence $\text{low}_H[u_i] < u_i$ when we mark u_i finished. In the first case, if either $\text{low}_H[u_1]$ or $\text{low}_H[u_2]$ is equal to v, before the back edges emanating from u_1 and u_2 are added, then that back edge is of type B. Now suppose that both $\mathrm{low}_H[u_1]$ and $\mathrm{low}_H[u_2]$ are less than v. In this case only one of the back edges emanating from u_1 or u_2 can be chosen (the one that goes higher). This is because when we mark u_1 and u_2 finished we do not add any back edges. The only back edges that are added will be from an ancestor of v, in which case we will pick only the highest going back edge out of either u_1 or u_2.

In the second case, $\mathrm{low}_H[u_1] \leq v$ when we mark u_1 finished. If it is equal to v, then the edge emanating from u_1 is of type B. If it is $< v$ then again only one of the two back edges emanating from v and u_1 can be chosen (the one that goes higher). □

Hence after this process, each leaf vertex has a charge of 1, and each non-leaf vertex has a charge of at most $\frac{1}{2}$. Thus the total number of added edges (net increase) is no more than $\ell + \frac{1}{2}(n-\ell) = \frac{1}{2}(n+\ell)$. Thus Theorem 4.10 follows. □

4.3.4. Wrapping up the Approximation Analysis

Theorem 4.12: *The algorithm outputs a solution of size no more than $\frac{5}{3}$ OPT.*

Proof: The ratio of the algorithm's solution to the OPT solution is upper bounded by $\frac{n-1+\ell+x}{\max(n,2\ell+x)}$.

By Theorem 4.10 we know that $x \leq \frac{1}{2}(n-\ell)$. The approximation ratio of the algorithm is upper bounded by the maximum possible value of the following function:

$$\frac{n+(\ell+x)}{\max(n, 2\ell+x)}$$

Case 1: $n \geq 2\ell + x$
We wish to compute the maximum of $1 + \frac{\ell+x}{n}$ subject to the constraints on ℓ, x. The constraints are $n - 2\ell \geq x$ and $\frac{1}{2}(n-\ell) \geq x$. Under these constraints we have to maximize $\frac{(\ell+x)}{n}$. We thus obtain $x + 2\ell \leq n$ and $2x + \ell \leq n$, hence $(\ell + x) \leq \frac{2}{3}n$. This yields $\frac{5}{3}$ as an upper bound.

Case 2: $n \leq 2\ell + x$
We wish to compute the maximum of $\frac{n+\ell+x}{2\ell+x}$. Replacing n by $2\ell + x$, (since its an upper bound) we get,

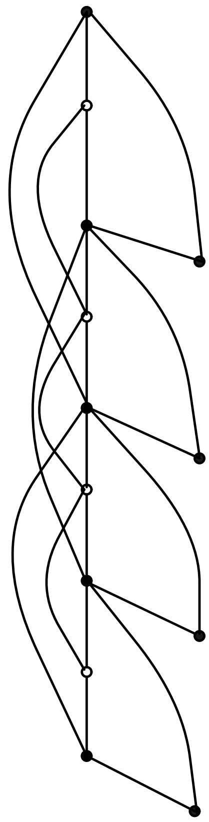
$$\frac{3\ell+2x}{2\ell+x} = 2 - \frac{1}{u}$$

where $u = 2 + \frac{x}{\ell}$. To compute the maximum value we wish to maximize u. The constraints are $n - 2\ell \leq x$ and $\frac{1}{2}(n-\ell) \geq x$. Under these constraints we have to maximize $\frac{x}{\ell}$. This time we get $(x = \ell)$, hence we get $u = 3$ and hence the maximum value is $\frac{5}{3}$.

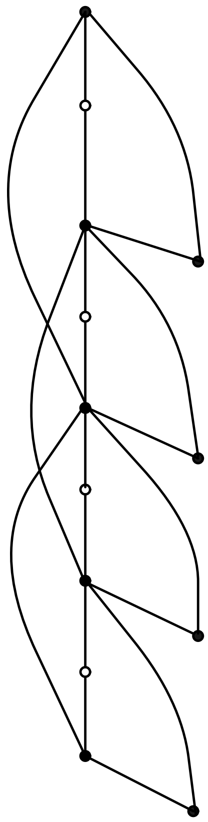
□

4.3.5. Worst Case Example

Fig. 7 describes an instance of an example where the algorithm achieves an approximation factor that is as bad as $\frac{5}{3}$ in the limit. (The example is due to Huzur Saran, Vijay Vazirani and Neal Young.) There is a path of alternating "black" and "white" vertices of length $2m+1$, with m white vertices, with $n = 3m+1$. (In this instance $m = 4$.) There are m leaves that are connected to the black vertices (one to each, except at the root). This describes the DFS tree T completely. There are back edges that connect each leaf to the closest black ancestor that is not the parent. There are also back edges that connect adjacent white vertices on the path, and back edges connect alternate black vertices on the path. The number of back edges added from the leaves is m, and the number of back edges added from non-leaf vertices is $m-1$. The total number of edges in the subgraph obtained by the algorithm is $5m-1$, but the graph is easily seen to be Hamiltonian. (Take all the leaves and their adjacent edges, and add to it a path connecting successive white vertices together with the extreme edges of the alternating path.) Thus the ratio is $\frac{5m-1}{3m+1}$, and approaches $\frac{5}{3}$ asymptotically.



(a) The graph G



(b) The edges picked by Algorithm 2-VC

Figure 7: Example to illustrate worst case performance.

5. Weighted Graphs

Consider the following problem: Given a graph $G=(V,E)$ with weights on the edges, find the *smallest* weight spanning subgraph $H=(V,E_H)$ that is k edge connected (for any k).

The problem is known to be NP-hard [GJ78]. An algorithm that achieves an approximation factor of 3 for $k=2$ is implied by [FJ81] as follows. Find the minimum spanning tree. Consider the problem of adding 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 [GJ78]. They gave an algorithm with an approximation factor of 2 for the problem of augmenting connectivity, yielding an approximation factor of 3 for the least weighted 2-edge connected subgraph. (The same factor for 2-vertex connectivity is obtained as well.)

Consider a directed graph G with weights on the edges, and a fixed root r. How does one find the *cheapest* directed subgraph H^D that has k edge-disjoint paths from a fixed root r to each vertex v ? Gabow [G91a] gives the fastest implementation of a weighted matroid intersection algorithm to solve this problem in $O(kn(m + n \log n) \log n)$ time. (See also [Ed79, FT89].)

To solve our problem (approximation algorithm) take the undirected graph G, and replace each undirected edge (u, v) by two directed edges (u, v) and (v, u) with each edge having 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. 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 5.1: *The graph E_H is a k edge connected spanning subgraph of G.*

Proof: Suppose in contradiction that there is a $k-1$ 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 k edge disjoint directed paths to v. Thus, there cannot be a cut set of size $k-1$. □

Theorem 5.2: *The total weight of E_H is at most twice the weight of the OPT solution.*

Proof: Consider the OPT solution for the problem. Consider all the antiparallel edges corresponding to edges in OPT. We get a directed subgraph in G^D of cost $2c(OPT)$ (where $c(OPT)$ is the total weight of the edges in OPT). From r there are k edge-disjoint undirected paths to any vertex v; they also yield k directed paths from r to v that are edge disjoint. Thus, this subgraph has the property of having k directed edge disjoint paths from r to any vertex v. The optimum solution found by Gabow's algorithm can therefore only be cheaper. □

6. What can we hope for ?

In this paper we showed that we can get multiplicative approximation factors of $\frac{3}{2}$ and $\frac{5}{3}$ for the 2-connected (edge and vertex respectively) spanning subgraph problem. In this section we ask: Is a multiplicative constant the best that we could hope for ? Can one hope to get an additive constant ? We answer this question negatively by proving that no additive constant is possible.

Theorem 6.1: *If $P \neq NP$, then for any constant C there is no polynomial time approximation algorithm that can obtain a solution to 2-edge connected spanning subgraph that is $\leq OPT + C$.*

Proof: Let us assume that there exists a polynomial time algorithm A that achieves an additive approximation of $+C$. Consider a graph G for which we wish to solve the 2-edge connected spanning subgraph problem. Make $C+1$ identical copies of the graph G, say $G_1, \ldots, G_{C+1}$. Add 2 new vertices v_1, v_2. Pick a vertex u_i from each G_i and add edges from it to each v_j ($j = 1, 2$). This gives us a 2-edge connected graph G' with $(C+1)n+2$ vertices (assuming G was 2-edge connected and has n vertices). We now input the graph G' to algorithm A. Clearly, the optimal solution for G' has exactly $(C+1)OPT(G) + 2(C+1)$ edges. (The best strategy is to pick the optimal solutions from each G_i and add in the edges of the "gadget" that connects all these pieces.) The algorithm A is guaranteed to produce a solution with at most $(C+1)OPT(G) + 2(C+1) + C$ edges (within $OPT(G') + C$). The edges introduced by the gadget are all essential edges (since each of them is part of a 2 edge cut). Thus all the edges incident to the vertices v_j are picked. Thus the algorithm picks $(C+1)OPT(G)+C$ edges from the $(C+1)$ copies of G. Hence the average number of edges picked from each copy is

$$\frac{(C+1)OPT(G)+C}{C+1}.$$

Thus there must be at least one copy from which we pick exactly $OPT(G)$ edges (since we cannot pick any fewer edges). This gives us the optimal solution for G, solving an NP-hard problem. $\square$

Notice that a similar proof will not work for the vertex case even if we "attach" the copies of the graph together by picking two vertices u_i and w_i from G_i simply because we may "disturb" the costs of the optimal solutions in each copy.

Theorem 6.2: *If $P \neq NP$, then for any constant C there is no polynomial time approximation algorithm that can obtain a solution to 2-vertex connected spanning subgraph that is $\leq OPT + C$.*

Proof: Let us assume that there exists a polynomial time algorithm A that achieves an additive approximation of $+C$. Consider a graph G for which we wish to solve the Hamilton Path problem (Is there a Hamiltonian Path from x to y ?). Delete the edge from x to y if such an edge exists (since it will not be used in a Hamiltonian path if G has at least 3 vertices). Add a new vertex z to G with edges (z,x) and (z,y) to obtain graph G'. Clearly G' has a Hamiltonian cycle passing through x, z, y in sequence if and only if G has a Hamiltonian path from x to y. In fact all Hamiltonian cycles in G' must pass through x, z, y in sequence.

Make $C+1$ identical copies of the graph G', say $G'_1, \ldots, G'_{C+1}$. Let vertices x_i and y_i be the copies of vertices x, y in G'_i. Add 2 new vertices v_1, v_2. Add edges (v_1, x_i) and (v_2, y_i) for $i = 1, \ldots, C+1$. This gives us a 2-vertex connected graph G'' with $(C+1)(n+1)+2$ vertices (assuming G' was 2-vertex connected and has $n+1$ vertices). We now input the graph G'' to algorithm A. Clearly if G has a Hamiltonian path from x to y, the optimal solution for G' has exactly $n+1$ edges (recall that a new vertex z was added to G). Hence if G has a Hamiltonian path then the smallest biconnected spanning subgraph of G'' has $(C+1)(n+1)+2(C+1)$ edges. (The best strategy is to pick the optimal solutions from each G'_i and add in the edges of the "gadget" that connects all these pieces.) The algorithm A guarantees to produce a solution with at most $(C+1)(n+1)+2(C+1)+C$ edges (within $OPT(G'')+C$). The edges introduced by the gadget are all essential edges (since each is part of a 2 cut). Thus all the edges incident to the vertices v_i are picked. Thus the algorithm picks $(C+1)(n+1)+C$ edges from the $(C+1)$ copies of G'. Hence the average number of edges picked from each copy is

$$\frac{(C+1)(n+1)+C}{C+1}.$$

Thus there must be at least one copy from which we pick exactly $(n+1)$ edges (since we cannot pick any fewer edges). This gives us a Hamiltonian cycle in G' and thus a Hamiltonian path in G from x to y (if one exists), solving an NP-hard problem. $\square$

Acknowledgments: We thank Yossi Matias and Neal Young for help in the generation of random graphs. We thank Huzur Saran, Vijay Vazirani and Neal Young for providing the example in Subsection 4.3.5 on the occasion of the first authors wedding. We thank Ramki Thurimella for useful discussions. We also thank the referees for their comments.

References

[BR92] P. Berman and V. Ramaiyer, "An approximation algorithm for the steiner tree problem," 3^{rd} *Annual Symposium on Discrete Algorithms*, pp. 325–334, (1992).

[CT91] J. Cheriyan and R. Thurimella, "Algorithms for parallel k-vertex connectivity and sparse certificates," 23^{rd} *Annual Symposium on Theory of Computing*, pp. 391–401, (1991).

[Ed79] J. Edmonds, "Matroid intersection," *Annals of Discrete Mathematics*, No. 4, pp. 185–204, (1979).

[ET76] K. P. Eswaran and R. E. Tarjan, "Augmentation problems," *SIAM Journal on Computing*, Vol. 5, No. 4, pp. 653–665, (1976).

[FJ81] G. N. Frederickson and J. JáJá, "Approximation algorithms for several graph augmentation problems," *SIAM Journal on Computing*, Vol. 10, No. 2, pp. 270–283, (1981).

[FJ82] G. N. Frederickson and J. JáJá, "On the relationship between the biconnectivity augmentation and traveling salesman problems," *Theoretical Computer Science*, Vol. 19, No. 2, pp. 189–201, (1982).

[Fr92] A. Frank, "Augmenting graphs to meet edge-connectivity requirements," *Siam Journal on Discrete Mathematics*, Vol. 5, No. 1, pp. 25–53, (1992).

[Fr91] G. N. Frederickson, "Ambivalent data structures for dynamic 2-edge connectivity and k smallest spanning trees," 32^{nd} *Annual Symposium on Foundations of Computer Science*, pp. 632–641, (1991).

[FT89] A. Frank and E. Tardos, "An application of submodular flows," *Linear Algebra and its Applications*, 114/115, pp. 320–348, (1989).

[G91a] H. N. Gabow, "A matroid approach to finding edge connectivity and packing arborescences," 23^{rd} *Annual Symposium on Theory of Computing*, pp. 112–122, (1991).

[G91b] H. N. Gabow, "Applications of a poset representation to edge connectivity and graph rigidity," 32^{nd} *Annual Symposium on Foundations of Computer Science*, pp. 812–822, (1991).

[GB93] M. X. Goemans and D. J. Bertsimas, "Survivable networks, linear programming relaxations and the parsimonious property," *Mathematical Programming*, Vol. 60, No. 2, pp. 145–166, (1993).

[GI91] Z. Galil and G. Italiano, "Fully dynamic algorithms for edge connectivity problems," 23^{rd} *Annual Symposium on Theory of Computing*, pp. 317–327, (1991).

[GJ78] M. R. Garey and D. S. Johnson, "Computers and Intractability: A guide to the theory of NP-completeness", *Freeman, San Francisco* (1978).

[GMS92] M. Groetschel, C. L. Monma and M. Stoer, "Computational results with a cutting plane algorithm for designing communication networks with low-connectivity constraints," *Operations Research*, Vol. 40, No. 2, pp. 309–330, (1992).

[Ha62] F. Harary, "The maximum connectivity of a graph," *Proc. Nat. Acad. Sci.*, 48, pp. 1142–1146, (1962).

[HKRT92] X. Han, P. Kelsen, V. Ramachandran and R. E. Tarjan, "Computing minimal spanning subgraphs in linear time," 3^{rd} *Annual Symposium on Discrete Algorithms*, pp. 146–156, (1992).

[HR91a] T. S. Hsu and V. Ramachandran, "A linear time algorithm for triconnectivity augmentation," 32^{nd} *Annual Symposium on Foundations of Computer Science*, pp. 548–559, (1991).

[HR91b] T. S. Hsu and V. Ramachandran, "On finding a smallest augmentation to biconnect a graph," 2^{nd} *Annual International Symposium on Algorithms*, Springer Verlag LNCS 557, pp. 326–335, (1991).

[Jo82] D. S. Johnson, "The NP-completeness column: An ongoing guide," *Journal of Algorithms*, Vol. 3, pp. 288–300, (1982).

[KMB81] L. Kou, G. Markowsky and L. Berman, "A fast algorithm for steiner trees," *Acta Informatica*, 15, pp. 141–145, (1981).

[KR91] P. Kelsen and V. Ramachandran, "On Finding Minimal Two-Connected Subgraphs," 2^{nd} *Annual Symposium on Discrete Algorithms*, pp. 178–187, (1991).

[KT93] S. Khuller and R. Thurimella, "Approximation Algorithms for Graph Augmentation," *Journal of Algorithms*, Vol. 14, No. 2, pp. 214–225, (1993).

[MK89] C. L. Monma and C. W. Ko, "Methods for designing survivable communication networks," *NATO Advanced Research Workshop*, Denmark, (1989).

[MR86] G. Miller and V. Ramachandran, "Efficient parallel ear decomposition with applications," *manuscript*, (1986).

[MSV86] Y. Maon, B. Schieber, and U. Vishkin. "Parallel Ear Decomposition Search (EDS) and *st*-numbering in graphs," *Theoretical Computer Science*, Vol. 47, pp. 277–298, (1986).

[NGM90] D. Naor, D. Gusfield and C. Martel, "A fast algorithm for optimally increasing the edge-connectivity," 31^{st} *Annual Symposium on Foundations of Computer Science*, pp. 698–707, (1990).

[NI90] H. Nagamochi and T. Ibaraki, "Linear time algorithms for finding a sparse k-connected spanning subgraph of a k-connected graph," *Algorithmica*, Vol. 7, No. 5/6, pp. 583–596, (1992).

[RG77] A. Rosenthal and A. Goldner, "Smallest augmentations to biconnect a graph," *SIAM Journal on Computing*, Vol. 6, No. 1, pp. 55–66, (1977).

[SWK69] K. Steiglitz, P. Weiner and D. J. Kleitman, "The design of minimum-cost survivable networks," *IEEE Transactions on Circuit Theory*, CT-16, 4, pp. 455–460, (1969).

[Ta72] R. E. Tarjan, "Depth-first search and linear graph algorithms," *SIAM Journal on Computing*, Vol. 1, No. 4, pp. 146–159, (1972).

[TM80] H. Takahashi and A. Matsuyama, "An approximate solution for the steiner tree problem in graphs," *Math. Japonica*, 24, pp. 573–577, (1980).

[TV85] R. E. Tarjan and U. Vishkin, "An efficient parallel biconnectivity algorithm," *SIAM Journal on Computing*, 14, pp. 862–874, (1985).

[Vi85] U. Vishkin, "On efficient parallel strong orientation," *Information Processing Letters*, Vol. 20, pp. 235–240, (1985).

[Wh32] H. Whitney, "Non-separable and planar graphs," *Trans. Amer. Math Soc.*, 34, pp. 339–362, (1932).

[WN87] T. Watanabe and A. Nakamura, "Edge-connectivity augmentation problems," *J. of Comp. and Sys. Sciences*, 35 (1), pp. 96–144, (1987).

[Ze93] A. Zelikovsky, "An 11/6-approximation algorithm for the network steiner problem," *Algorithmica*, Vol. 9, No. 5, pp. 463–470, (1993).