A Uniform Framework for Approximating Weighted Connectivity Problems *

An Zhu †

Abstract

NP-hard problems are problems that are computationally expensive to solve optimally. In order to get the exact solution, one has to consume a lot of time and resources. So in practice, we consider heuristics that produce solutions very close to the optimal solution while running in manageable (polynomial) time. We measure the heuristics by the quality of the solutions they produce. A heuristic has an approximation factor α if the cost of the solution is guaranteed to be no more than α times the cost of the optimal solution over all instances. Often people design heuristics for individual NP-hard problems.

We introduce a new general algorithmic technique that applies to a family of NP-hard optimization problems [8]. A single algorithmic approach appears to apply successfully to a diverse collection of graph connectivity problems. We demonstrate the power of this method by doing an experimental study of the minimum-weight strongly-connected spanning subgraph problem, as well as the minimum-weight augmentation problem. The key point is that even though we are unable to improve the known worst-case approximation ratios for these problems, using the new method leads to significantly better solutions when we compare it to the previous approximation algorithms for these problems and yields solutions very close to the optimal solutions. We believe that our technique will eventually lead to algorithms that improve the worst-case performance as well.

It is worth pointing out that this family of graph connectivity problems has very rich applications in practice. One direct application of the augmentation problem is VLSI (Very Large Scale Integration) floorplanning, which deals with the problem of giving a circuit represented by a hypergraph, determine the approximate location of each module in a rectangular chip area.

1 Introduction

Let a weighted graph $G = (V, E)$ represent all the feasible links of a potential communications network. A minimum spanning tree in G is the cheapest 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 connectivities. A network of edge-connectivity (vertex-connectivity) k continues to allow communication between functioning sites even after as many as $k-1$ links (sites) have failed. Given an arbitrary graph G with non-negative edge-weights, and an integer k, we consider the basic problem of finding a minimum-weight k-connected spanning subgraph. For references and other work on similar problems see [2, 3, 4].

In digraphs, the corresponding problem is that of finding a minimum weight strongly connected spanning subgraph (SCSS) and its generalizations to higher connectivities. A strongly connected

*A short version of this paper (co-authored with Samir Khuller and Balaji Raghavachari) appeared in 10^{th} Annual SIAM-ACM Symposium on Discrete Algorithms (SODA), Baltimore, Jan. 1999.

†Department of Computer Science, University of Maryland, College Park, MD 20742. Research partly supported by NSF CAREER Award CCR-9501355, and a Senior Summer Scholarship from University of Maryland. E-mail : `anzhu@wam.umd.edu`.

graph is one which has a directed path, between every ordered pair of vertices. Since this problem is NP-hard, we will discuss polynomial time heuristics. One heuristic for this problem works as follows: fix a root r and find a minimum weight in-branching[1] B_I and a minimum weight out-branching B_O [7], and take the union of the two branchings. Frederickson and JáJá [1] used this idea to obtain a 2-approximation for the SCSS problem. Observe that the weight of each branching is at most the weight of the minimum strongly connected subgraph; thus we get an approximation factor of 2.

One can define similar problems in undirected graphs. One natural problem is giving an undirected graph G and a connected subgraph G_0, finding a minimum weight subset of $E - E_0$ whose addition makes G_0 2-edge connected and its generalization to higher connectivities. This augmentation problem has its direct application in VLSI floorplanning, which we discuss briefly in Section 4. One basic heuristic for this problem works as follows: fix a root r, we convert the undirected graph to a directed graph, then find a minimum weight out-branching B_O; and take the edges in $E - E_0$ corresponding to the edges in B_O. This heuristic also has an approximation factor of 2.

Note that the solution to both problems contains a collection of edges, and the two basic heuristics above find one such collection at a single iteration. Our general technique instead selects one edge at a time by some measurement of how useful a particular edge is, and the iterations of this selection process yield a collection of edges which is a better solution at the end. A lot of work has been done on the design of approximation algorithms for undirected graphs [4], but very little is known about directed graphs.

2 General framework

Suppose, given an NP-hard problem P, we could "relax" some of the constraints from problem P, obtaining a set of sub-problems $P_1, P_2, \ldots, P_t$ such that, each P_i can be solved optimally in polynomial time yielding a solution S_i. By taking the union of the solutions to the sub-problems, suppose we obtain a solution S for the original problem P. By proving that $w(S_i) \leq k_i w(OPT)$, where k_i is a constant and OPT is the weight of an optimal solution to problem P, we obtain: $w(S) \leq \sum_{i=1}^{t} w(S_i) \leq w(OPT) \sum_{i=1}^{t} k_i$. This gives a polynomial time approximation algorithm with an approximation factor of $\sum_i k_i$. For strong connectivity, P_1 is a minimum-weight in-branching problem, and P_2 is a minimum-weight out-branching problem. In this case, $k_1 = k_2 = 1$, and the approximation ratio is 2.

Our algorithm can be applied in such a framework as follows. For each edge e, we measure the amount by which it affects $\sum w(S_i)$ if $w(e)$ were made 0. The ratio of the net change in $\sum w(S_i)$ to $w(e)$ is drop(e). An edge has a high drop value if it can be utilized by several (near) optimal solutions to the subproblems. We could then choose an edge with a high drop value and include it in the solution by making its weight 0. This is repeated until there is a zero-weight solution. The algorithm then returns as a solution all the edges that were chosen in one of the steps. Problems that fit in this framework include:

- **k-Edge connectivity:** Given a graph $G = (V, E)$ with a weight function w defined on the edges, the k**ECSS** problem is to find a minimum weight k-edge-connected spanning subgraph.
- **Strong Connectivity:** Given a digraph $G = (V, E)$ with a weight function w defined on the edge, the **SCSS** problem is to find a minimum weight strongly connected spanning subgraph.

[1] In digraphs, an in-branching is a connected subgraph where each vertex has a directed path to the root and the out-degree for those vertices is 1. An out-branching is similar to in-branching with edge direction reversed.

- **k-Augmentation:** Given a graph $G = (V, E)$, a subgraph $H = (V, E_H)$ of G, and a positive integer k, the k**AUG** problem is to find a subset of edges of G with minimum total weight, whose addition to E_H generates a k-edge-connected graph[1, 4]. When the initial subgraph H has no edges then this is the kECSS problem.
- **Directed edge connectivity:** We also consider a natural directed analog of the k-edge-connectivity problem, which generalizes the SCSS problem. The input is a graph $G = (V, E)$ and a positive integer k. The problem is to find a min-weight subgraph which also has k disjoint directed paths between every ordered pair of vertices.
- **Vertex connectivity:** finding a minimum-weight k-connected spanning subgraph.

In this paper, we will discuss Strong Connectivity problem and 2AUG, which is called Augmentation problem below.

3 The Strong Connectivity Problem

The input is a directed graph $G = (V, E)$ with weights on the edges. The goal is to find an SCSS of G of minimum weight.

Let r be an arbitrary root vertex of G. Let B_I denote a minimum weight in-branching in G, rooted at r. Let $w(B_I)$ denote the weight of B_I. Similarly, B_O denotes a minimum weight out-branching rooted at r and $w(B_O)$ denotes its weight. Recall that the union of B_I and B_O will yield a 2-approximation, which is the basic heuristic. Let $s = w(B_I) + w(B_O)$ be the sum of the weights of the branchings. The general technique measures the "importance" of an edge as follows:

Definition: Let $e \in E$ be an edge with $w(e) > 0$. Let B'_I and B'_O be the minimum weight in-branching and out-branchings in G if $w(e)$ is *reduced* to 0. Define **drop(e)** as follows:

$$\text{drop(e)} \; = \frac{(w(B_I) + w(B_O)) - (w(B'_I) + w(B'_O))}{w(e)}$$

In other words, *drop(e)* is defined as the ratio of the *decrease* in the sum of the weights of the two branchings when $w(e)$ is made zero. Intuitively, the higher the value of *drop(e)*, the more it is "shared" by the two branchings. Clearly, the value of *drop(e)* is at most 2 (this is because if the edge e belongs to both B_I and B_O, then reducing its weight to zero, causes a drop in the total weight of these two branchings of $2w(e)$). Also, *drop(e^*)*, the maximum of drop(e) over all edges of non-zero weight, is at least 1. The new heuristic is then the following:

Strong-connectivity(G) —

Pick an arbitrary root vertex r.
while $s = w(B_I) + w(B_O) > 0$ do
  Let e^* be an edge with maximum drop.
  Set $w(e^*) = 0$.
Compute 0-weight in-/out- branchings rooted at r.
return union of edges in the branchings.

Since both branchings are computed with the same root r, the union of the two branchings yields a strongly connected graph. Note that at the very first iteration, the union of B_I and B_O is the solution if we are using the basic heuristic. So our general technique can also be thought of improving the solution produced by the basic heuristic. Section 5 formally proves the new heuristic also has an approximation factor of 2 (i.e., the solution the new heuristic found is no more than twice the cost of the optimal solution).

4 The Augmentation Problem

The input is an undirected graph $G = (V, E)$ and a connected subgraph $G_0 = (V, E_0)$. The goal is to find a subset of $E - E_0$ whose addition to G_0 makes G_0 2-edge connected. 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 $E - E_0$ are the minimum weight edges that connect vertices of different 2-edge connected components of G_0. (Observe that the edges of $E - E_0$ that connect vertices of the same 2-edge connected component are of no use in augmenting G_0. Similarly, among the edges that connect different 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. In case 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 bounded by twice the weight of an optimal augmentation.

Algorithm *Find Edge Aug*
Input: Graph G and a spanning tree G_0.
Output: A set of edges $Aug \subseteq E - E_0$.

(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, 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 Figure 1.)

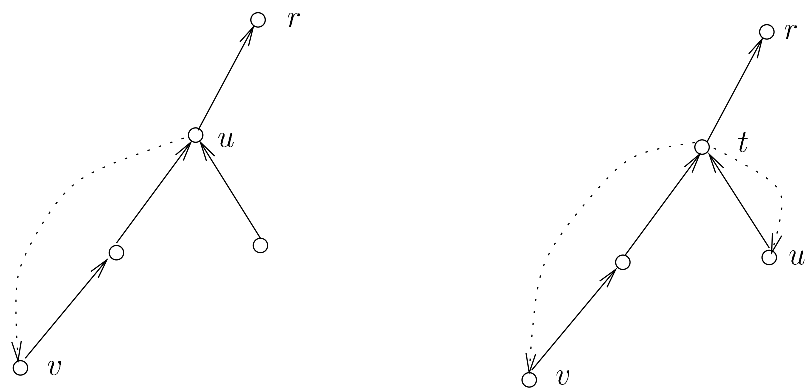
(i) If u is an ancestor of v (the converse is symmetric): then add an edge $u \to v$ in G^D with weight w.

(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 $t \to u$ and $t \to v$ in G^D, each with weight w.

(2) Find a minimum weight out-branching B_O 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). We have the following results [5].

Lemma 4.1 *If G is 2-edge connected, then the directed graph G^D is strongly connected.*



(a) (b)

Figure 1: Construction of G^D.

Lemma 4.2 *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.*

Lemma 4.3 *The weight of Aug is less than twice the optimal augmentation. (Weight$(Aug) \leq 2C^*$ where C^* is the optimal augmentation weight.)*

Applying our general technique, we can similarly define the "importance" of a certain edge e and design a new heuristic as follows:

Definition: Let $e \in E$ be an edge with $w(e) > 0$. Let B'_O be the minimum weight out-branching in G^D if $w(e)$ is *reduced* to 0. Define **drop(e)** as follows:

$$\text{drop(e)} \ = \frac{w(B_O) - w(B'_O)}{w(e)}$$

AUGMENTATION(G, G_0) —

Pick an arbitrary root vertex r. Compute G^D from G.
while $s = w(B_O) > 0$ do
  Let e^* be an edge with maximum drop. Set $w(e^*) = 0$.
Compute 0-weight B_O rooted at r in G^D.
return the edges of $E - E_0$ corresponding to the edges of B_O

The Analysis section will prove the new heuristic has an approximation factor of 2 also.

4.1 Application of Augmentation problem in VLSI Design

The Augmentation problem stated above has its application in the VLSI design [6], which we describe the basic ideas below.

Consider the following floorplanning system: In phase one, we represent a given circuit by a planar graph $G = (V \cup \{X\}, E)$ with weights on all vertices of V and all edges. Vertex X represents the set of I/O pads, and each vertex of V and its weight correspond to a functional block and the area required to realize the block. Each edge and its weight represent interconnection requirements and the number of wires between two functional blocks corresponding to the end vertices of the edge. A planar embedding D of a dual graph of G such that each face and the total graph enclosure are rectangles and the exterior face of D corresponds to a specific vertex X is called a **rectangular dual**. If G has a rectangular dual, then G can be drawn in as small area as possible. In phase two, among all the possible rectangular duals D of G, one then has to find a most area-efficient drawing of D such that the above area and connection constrains are met.

In [6], Tsukiyama, Kioke and Shirakawa considered the problems arising in phase two, where the goal is to transform G into a graph that has rectangular dual. It is known that the sufficient condition for G to have rectangular dual is that G is maximal and G has no complex triangles [6]. A maximal planar graph is one in which every face is bounded by a triangle, since no edge can be added without violating planarity. A triangle of a maximal planar graph which is not a boundary of a face is called a complex triangle. So in order to get a rectangular dual from a general graph G, we first make G planar, then maximal, then eliminate all the complex triangles. The 3^{rd} step is of most interest in this problem.

We eliminate a complex triangle as follows: split the edge e on the complex triangle into two edges by introducing a new vertex L and two new edges incident to L to keep the graph maximal. This process does not introduce any new complex triangles, hence can be applied repeatedly. Now in the new graph G', we can find a rectangular dual, with additional faces generated by vertices like L. The minimization of such additional area contributes to the reduction of the whole chip area. Let T be the set of complex triangles of G, and let $G(T) = (V(T), E(T))$ be a subgraph of G consisting of vertices and edges contained in complex triangles of T. Thus the problem of eliminating all complex triangles in a maximal planar graph $G = (V \cup \{X\}, E)$ can be formulated into a problem of finding a minimum weight set E^* of edges in $G(T)$ such that E^* contains at least one edge from every complex triangle, which is called the COVER problem.

The COVER problem is NP-hard. We solve the COVER problem by transforming COVER to the Augmentation problem (G^*, G_0). We introduce a graph $G^* = (r \cup V^*, E^*)$, where r is the root of G_0 and each vertex in V^* corresponds to a complex triangle in T. G_0 is roughly a tree that represents the parent-child relationship between the complex triangles in T. Triangles in T with no parent will have r as the parent. And the edge set $E^* - E_0$ is made up of edges in $E(T)$. $\forall e \in E(T)$, if e is incident to the two smallest complex triangles u and v that can possibly contain e[2], then we simply add an edge of weight $w(e)$ between the two corresponding vertices. Otherwise we add an edge of weight $w(e)$ between the smallest triangle and the parent of the largest triangle containing e. Thus one can prove that solving the augmentation problem on instance (G^*, G_0) yield a solution to the COVER problem with approximation factor of 2 [6].

5 Analysis:

Since our general technique uses the notion drop(e), we named the new heuristic the **Drop Algorithm**. The Drop Algorithm has an approximation factor of 2 for both problems. The following proof works for both the Strong Connectivity and the Augmentation problem.

Lemma 5.1 *At all iterations,* drop(e*)*, the maximum of drop(e) over all edges of non-zero weight, is at least 1.*

[2]i.e. there is no other complex triangles within u or v that contains edge e

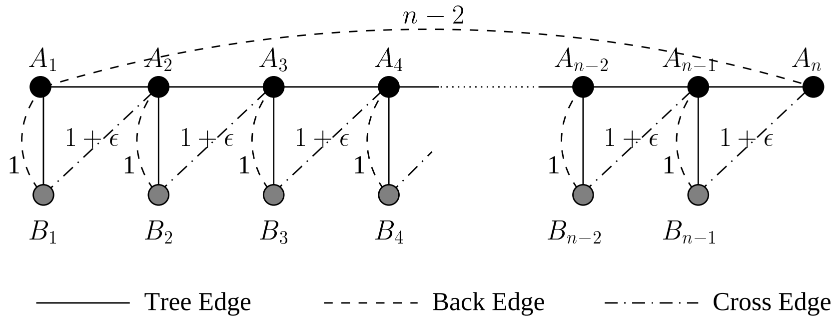


Figure 2: Tight Example for Weighted Augmentation Problem.

Proof. At all iterations, the weight of the branching is positive. This implies that the branching includes at least one edge e^0 of positive weight, then $drop(e^*) \geq drop(e^0) \geq 1$ ∎

Lemma 5.2 *At the very first iteration, s is no more than twice the cost of optimum solution.*

Proof. For the Strong Connectivity problem, $s = w(B_I) + w(B_O) \leq w(OPT) + w(OPT) = 2 \times w(OPT)$. For the Augmentation problem $s = w(B_O) \leq 2 \times w(OPT)$ [5]. ∎

Theorem 5.3 *The drop algorithm has an approximation factor of 2 for both Strong Connectivity and Augmentation problem.*

Proof. Now consider either of the two problems. The solution of the drop algorithm is no more than the $\sum_{e \in D} w(e)$ where D is the set of edges whose weights are dropped to 0 during the procedure. And we know $s = \sum_{e \in D} drop(e) \times w(e)$. Since e at the time being dropped must assume the maximum drop value, which means $drop(e) \geq 1 \; \forall e \in D$. So the solution Drop algorithm gives is no more than s. By previous Lemma the Drop algorithm has an approximation factor of 2. ∎

Theorem 5.4 *The drop algorithm has a tight approximation factor of 2 for the Augmentation problem, even in the unweighted case.*

Proof. See Figure 2 for a tight example for weighted graphs. This example shows that the Drop algorithm will produce a solution with cost arbitrarily close to $2 \times w(OPT)$. There are n nodes of type A $(A_1, \ldots, A_n)$ and $n-1$ nodes of type B $(B_1, \ldots, B_{n-1})$ in the graph. The tree edges consists of edges going from A_i to A_{i+1} and B_i, for $i = 1, \ldots, n-1$. The set of possible augmenting edges are made up of back edges and cross edges, with respect to the spanning tree rooted at A_1. There is one heavy back edge going from A_n to A_1 with weight $n-2$, and $n-1$ unit-weight back edges going from B_i to A_i for $i = 1, \ldots, n-1$. There are $n-1$ cross edges going from B_i to A_{i+1} with weight $1+\epsilon$ each. The OPT in this case consists of all $n-1$ cross edges, which altogether has a cost of $(n-1)(1+e)$. Originally, the minimum weight out-branching B_O consists of all back edges with cost $2n-3$. The Drop algorithm will stop at the very first iteration since all cross edges have drop value < 1 (see next section). So the cost to our solution is $2n-3$. The ratio is then $\frac{2n-3}{(n-1)\times(1+\epsilon)} \to 2$ as $n \to \infty$, $\epsilon \to 0$.

Note that this example stands even for an interesting extension of the Drop algorithm: Instead of considering one edge per iteration, we consider a group of k edges, and drop the whole group (with maximum drop value) of k edges' weight to zero per iteration. If we change the weight of the long back edge from $n-2$ to $n-k-1$, all groups of k edges will also have drop value < 1. So this extension has a tight approximation factor of 2 also.

For unweighted graphs, Figure 3 shows an example of the similar merit. We now have an additional $n-1$ nodes of type C ($C_1, \ldots, C_{n-1}$) in the graph. The tree edges are of the following: A_i has an edge to B_i and C_i, and C_i has an edge to A_{i+1}, for $i = 1, \ldots, n-1$. The back edges are of the following: B_i has an edge to A_i as before ($i = 1, \ldots, n-1$), A_n has an edge to C_{n-2}, C_i has an edge to C_{i-1} ($i = n-2, \ldots, 3$), and C_2 has an edge to A_1. Cross edges remain the same. Now all possible augmenting edges are of unit weight. Here the reader can verify that the drop value of each cross edge, as well as back edge, is 1. In the worst case, the algorithm will end up selecting all the back edges, with a total cost of $2n-3$, while the $n-1$ drop edges are the optimal set of augmenting edges. This example, however, does not generalize to the extension of considering a group of k edges simultaneously. Also notice that the algorithm could be clever and only drop cross edges to obtain better results (see next section).

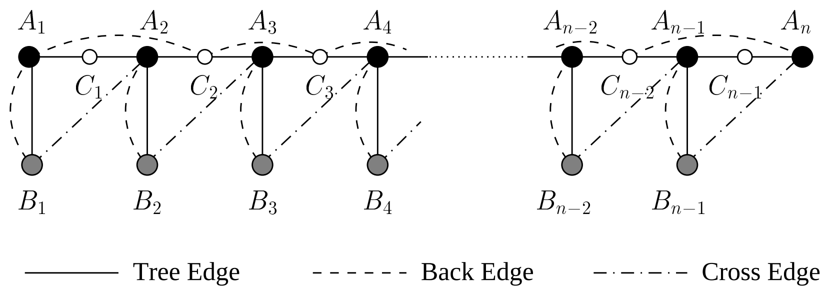


Figure 3: Tight Example for Unweighted Graphs

□

6 Implementation Issues:

The iterations of the drop algorithm stop when the total weight of branching reaches 0. This could be costly since the number of iterations is bounded by the number of edges in the solution. We can speed up the process by defining a new test condition. One natural condition is to stop as soon as the maximum drop value reaches exactly 1, because including an edge with drop value 1 does not improve our solution. However, it is worth pointing out that the maximum drop values aren't monotonously decreasing, i.e., it could happen that at iteration n the maximum drop value is 1, but at iteration $n+1$, the maximum drop value is as large as 2. Thus stopping early might prevent us from improving the solution further more. The experimental result shows stopping early can reduce the running time by 30 % to 40 %, while obtaining the same solution compared to running the program all the way, with very rare exceptions of small deviation.

For the augmentation problem, one can further reduce the running time by only considering cross edges (i.e. the two end vertices don't have ancestor-descendant relationship). The following lemmas prove the correctness of such consideration.

Lemma 6.1 *Back edges have maximum drop value of 1.*

Proof. Since when converting G to G^D, back edge e only contribute one directed edge, thus the changes in branching weight can be no more than $w(e)$. □

Lemma 6.2 *If a back edge e is dropped at iteration n, the drop values $drop_{n+1}(e')$ for all other edges e' at iteration $n+1$ will not increase compared to $drop_n(e')$, the drop value of e' at iteration n.*

Proof. Since at each iteration, the maximum drop value is at least 1, and by the lemma above, $drop_n(e) = 1$. Let S_n denote the weight of the branching at iteration n, S'_n denote the weight of the branching after $w(e)$ is set to zero, which will also be S_{n+1}, and $S_{n+1} + w(e) = S_n$. At iteration $n+1$, for any other edge e', consider $B_{e'}$ the branching after $w(e')$ is dropped to zero. If $B_{e'}$ does not contain e, then

$$drop_{n+1}(e') = \frac{S_{n+1} - w(B_{e'})}{w(e')} \leq \frac{S_n - w(B_{e'})}{w(e')} = drop_n(e')$$

If $B_{e'}$ contains e, then

$$drop_{n+1}(e') = \frac{S_{n+1} - w(B_{e'})}{w(e')} \leq \frac{(S_{n+1} + w(e)) - (w(B_{e'}) + w(e))}{w(e')} \leq drop_n(e')$$

That's because increase the weight of e to its original weight in $B_{e'}$ yields a valid branching at iteration n. □

This shows that we can compute drop values on just cross edges. When the maximum drop value of cross edges reaches below 1, then we can stop the iterations. At that stage we take the minimum weight out-branching B_O, including all the 0-weight edges in B_O and any back edges selected by B_O. So the Drop algorithm can be improved as follows:

AUGMENTATION(G, G_0) —

Pick an arbitrary root vertex r. Compute G^D from G.
do
    Compute drop values on only cross edges.
    Let e^* be an cross edge with maximum drop value.
    Set $w(e^*) = 0$ if $drop(e^*) \geq 1$ (or > 1).
while $drop(e^*) \geq 1$ (or > 1)
Compute minimum-weight B_O rooted at r in G^D.
return the edges of $E - E_0$ corresponding to the 0-weight edges and back edges of B_O.

Also note that the Drop algorithm described here pick an arbitrary vertex as the root. In practice one can often get a better solution by trying all vertices as the root and select the best one.

7 Experimental Results:

We first show some results for the strong connectivity problem. The first class of graphs is with weights chosen randomly. The second class is unweighted Hamiltonian graphs. The size of the graph corresponds to the number of vertices in the graph. For each graph size, we generated 7 random graphs each for 3 different types of density (which corresponds to the number of edges in a given size graph): Sparse, Median and Dense. The **Average** of the total 21 graphs is also listed. For example, 6.7% means that the method gave a solution that was more than the optimal solution by a factor of 1.067 (on average). Drop Alg I tries a single vertex as a root, and Drop Alg II tries all possible vertices as the root. For the Simple Algorithm, we tried all possible roots and took the best solution. So the correct comparison is between **bold** columns 1 and 3.

Results for Strong Connectivity in Weighted Graphs:

Size	Simple Alg (%)				Drop Alg I (%)				Drop Alg II (%)			
	A	S	M	D	**A**	S	M	D	**A**	S	M	D
10	**6.7**	3.3	7.7	9.0	**2.8**	1.8	4.2	1.7	**0.4**	0	0.7	0.4
15	**7.2**	3.7	9.2	8.6	**2.4**	3.3	2.4	1.3	**0.7**	0.4	0.4	1.2
20	**10.8**	6.4	12.9	12.9	**3.5**	2.6	2.6	5.4	**0.9**	0.4	1.1	1.1
25	**12.3**	7.9	14.5	14.4	**3.2**	2.6	4.3	2.6	**0.8**	0.7	1.7	0.2

Results for Strong Connectivity in Unweighted Graphs:

Size	Simple Alg (%)				Drop Alg I (%)				Drop Alg II (%)			
	A	S	M	D	**A**	S	M	D	**A**	S	M	D
10	**42.9**	32.9	42.9	52.9	**8.1**	12.9	7.1	4.3	**1.0**	0	2.9	0
15	**47.9**	41.9	43.8	58.1	**10.2**	13.3	9.5	7.6	**3.5**	4.8	3.8	2.0
20	**53.5**	42.1	49.3	69.3	**11.2**	16.4	12.9	4.3	**5.9**	9.3	7.1	1.4
25	**55.8**	44.6	52.6	70.3	**9.9**	14.3	11.4	4.0	**5.3**	8.6	7.4	0

For Augmentation problem, we have similar experimental setups. However in this problem, there is no need for Hamiltonian graphs.

Results for Augmentation in Weighted Graphs:

Size	Simple Alg (%)				Drop Alg I (%)				Drop Alg II (%)
	A	S	M	D	**A**	S	M	D	
10	**1.6**	4.5	0.1	0	**0**	0	0	0	**0**
15	**3.9**	4.9	2.9	3.7	**0**	0	0	0	**0**
20	**2.7**	0.4	3.1	4.6	**0.2**	0	0.2	0.5	**0**
25	**5.9**	2.2	8.0	5.6	**0.5**	0	1.5	0	**0**
50	**6.9**	6.1	7.1	7.5	**0.7**	0.1	1.8	0.3	**0**

Results for Augmentation in Unweighted Graphs:

Size	Simple Alg (%)				Drop Alg I (%)				Drop Alg II (%)			
	A	S	M	D	**A**	S	M	D	**A**	S	M	D
10	**9.5**	4.8	4.8	19.0	**1.6**	0	0	4.8	**0**	0	0	0
15	**22.7**	8.8	2.9	56.4	**1.9**	0	2.9	2.9	**0**	0	0	0
20	**28.1**	18.4	8.2	58.6	**2.9**	2.9	0	6.0	**0**	0	0	0
25	**31.9**	10.6	15.6	69.4	**3.7**	4.2	2.4	4.4	**0.8**	0	0	2.4
50	**42.9**	20.6	23.9	83.4	**4.3**	5.2	5.6	2.1	**0.7**	1.1	1.0	0

It is very clear that the Drop algorithm in practice produce results much better than the two basic heuristics. And it is interesting to note that especially for the Augmentation problem, Drop algorithm can find optimal solutions most of the time. In any case, Drop algorithm produces results very close to optimal solution.

8 Acknowledgments

First and last, I would like to thank Dr. Samir Khuller, who really inspired and helped me to work on this project and many other things. This paper was awarded the J.R. Dorfman prize for undergraduate research. Really this award goes to Dr. Khuller. Finally my great condolence goes to the 3 victims and their families in NATO's bombing of Chinese embassy on May 8th, 1999.

Theorem 5.4 was revised on Dec. 24, 2001 to include the tight example for unweighted graphs. Though this example had been worked out by the author in the summer after her graduation from Maryland, it is now finally in the paper for completeness. The author would like to dedicate this first ever full research paper of hers to her grandmother, who passed away on Dec. 17, 2001.

References

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

[2] M. X. Goemans, A. V. Goldberg, S. Plotkin, D. Shmoys, E. Tardos and D. P. Williamson, Improved approximation algorithms for network design problems, *Proc. 5th Annual ACM-SIAM Symp. on Discrete Algorithms*, pp. 223–232, (1994).

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

[4] S. Khuller, Approximation algorithms for finding highly connected subgraphs, In *Approximation Algorithms for NP-hard problems*, Ed. D. Hochbaum. PWS Publishing Company (1996).

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

[6] S. Tsukiyama, K. Kioke and I. Shirakawa, An Algorithm to Eliminate All Complex Triangles in a Maximal Planar Graph for Use in VLSI FLoorplan In *Algorithmic Aspects of VLSI Layout*, Ed. M. Sarrafzadeh and D. T. Lee. World Scientific Publishing Company (1993).

[7] H. N. Gabow, Z. Galil, T. Spencer and R. E. Tarjan. Effecient Algorithms for Finding Minimum Spanning Tree in Undirected and Directed Graphs, *Combinatorica*, pp. 109–122, (1986).

[8] S. Khuller, B. Raghavachari and A. Zhu. A Uniform Framework for Approximating Weighted Connectivity Problems, *Proc.* 10^{th} *Annual ACM-SIAM Symp. on Discrete Algorithms*, pp. S937–S938, (1999).