A Faster Implementation of the Goemans-Williamson Clustering Algorithm

Richard Cole* Ramesh Hariharan† Moshe Lewenstein‡ Ely Porat§

Abstract

We give an implementation of the Goemans-Williamson clustering procedure which is at the core of several approximation algorithms including those for Generalized Steiner Trees, Prize Collecting Travelling Salesman, 2-Edge Connected Subgraph etc. On a graph with n nodes and m edges, our implementation gives $O\left(k(n+m)\log^2 n\right)$ time approximation algorithms for all these problems at the expense of a slight additive degradation of $\frac{1}{n^k}$ in the approximation factor, for any constant k.

1 Introduction

The Goemans-Williamson clustering technique [2] is a technique for obtaining approximation algorithms for a very general class of graph optimization problems, namely that of covering the vertices of a given graph with trees, cycles, or paths satisfying certain constraints. Several fundamental problems fit into this class, e.g., Shortest Path, Minimum Spanning Tree, Minimum Steiner Tree, and generalizations of the Minimum Steiner Tree problems.

The above clustering technique was a generalization of the approximation algorithm of Agrawal, Klein and Ravi [1] for the Generalized Steiner Tree problem. Goemans and Williamson [2] originally used this technique to obtain slightly better than 2 approximation factors for several network design problems, including the Generalized Steiner Tree problem, the Non-Fixed Point-to-Point Connection problem, the Prize Collecting Steiner Tree problem, etc (these problems are defined later in Section 2.).

Subsequently, this technique has been at the core of several other approximation algorithms, e.g., the algorithm of Klein and Ravi [5] for the 2-Edge Connected Subgraph problem, and the algorithm of Williamson, Goemans, Mihail and Vazirani [6] for the Survivable Network Design problem.

Before describing our results, we give a brief overview of the general clustering technique, without reference to any particular problem. The output of the clustering procedure will be a forest, constructed edge by edge over several rounds. In each round, the procedure maintains a partition of the vertex set into subsets. Some of these subsets are designated *active*; which subsets are active depends on the problem at hand. In each round, all the active subsets "grow" at the same rate until two of the subsets (active or inactive) "meet"; the edge on which they meet is added to the forest to be output. Next, the two subsets which met above are joined together into a larger

*Courant Institute, NYU, New York. cole@cs.nyu.edu.

†Indian Institute of Science, Bangalore. ramesh@csa.iisc.ernet.in.

‡Bar-Ilan University, Ramat Gan, and Courant Institute, NYU, New York. moshe@cs.biu.ac.il.

§Bar-Ilan University, Ramat Gan. porately@cs.biu.ac.il.

subset, which is designated as active or inactive, again depending upon the problem at hand. The above procedure continues until no more active subsets are left. Finally, there is a pruning step in which some of the edges determined above are discarded.

Since the above clustering technique is at the core of several approximation algorithms, fast implementations of this technique are of interest. To implement the above procedure, three issues need to be addressed.

1. The first is that of determining which sets are active/inactive in each round.
2. The second is that of determining which edges must be discarded in the final pruning step.
3. The third is that of determining which two subsets to merge in each round.

The first two issues are problem dependent while the third is more general. Though we address the first two issues briefly for specific problems later, the third issue will be the main focus of this paper.

The key problem in determining which two subsets to merge in each round, or equivalently, to decide which edge to choose in each round, is complicated by the fact that there are three categories of edges, i.e., those having no active endpoints, those having one active endpoint, and those having two active endpoints. Note that the category in which an edge is placed keeps changing over the rounds, as subsets combine to form sets which may or may not be active. Therefore, as the rounds progress, it is necessary to keep track of the category of each edge. If done naively, this could be expensive, requiring $\Theta(n \log n)$ time per round to change the categories of the up to $\Theta(n)$ edges incident on the two subsets merged in a round; this will lead to an $O(n^2 \log n)$ time algorithm. Indeed, this was the complexity of the original Goemans-Williamson implementation [2].

The above time complexity was improved by two subsequent implementations. One by Klein [4], who proposed a trade-off solution for changing categories. This algorithm assigns each edge to one of its two endpoints. When two subsets combine to form a new subset S, category changes for edges incident on S and assigned to vertices in S can be performed in one shot; these edges are treated as one block of edges. Category changes for edges incident on S but assigned elsewhere need to be performed explicitly. However, by a proper choice of assignment, the latter time is restricted to examining just $\sqrt{m}$ edges (in the amortized sense), giving a time of $O(n\sqrt{m}\log n)$ on the whole. The other faster implementation was by Gabow, Goemans and Williamson [3] who restrict attention to only the best r edges exiting each subset in each round, thus restricting the time taken per round to $O(n+r\log n)$; however, every n/r rounds, $O(m)$ time is spent on finding the best r edges exiting each subset. Setting $r = \sqrt{\frac{m}{\log n}}$ gives a time bound of $O(n(n+\sqrt{m\log n}))$, which they further improve to $O(n(n+\sqrt{m\log\log n}))$ by a slightly different organization of the priority queues. We mention that Klein's algorithm [4] takes linear space while the Gabow-Goemans-Williamson algorithm [3] takes $O(n^2)$ space.

Our Contribution. We give an extremely simple method based on *Dynamic Edge Splitting* for tackling the third issue above. This method allows an implementation of the Goemans-Williamson clustering procedure in $O((n+m)\log^2 n)$ time and $O(n+m)$ space, provided the first two issues can also be addressed in this time and space bound. This time bound is a substantial improvement on the above mentioned algorithms for graphs which are not too dense. However, our implementation suffers a slight $\frac{1}{n^k}$ additive degradation in the approximation factor, where k can be made as large as required; the running time increases linearly in k.

Since it is indeed the case that Issues 1 and 2 can be addressed in the above mentioned time and space bound for several problems, the above result leads to the following approximation algorithms, all running in $O((n+m)\log^2 n)$ time and linear space.

1. A $2 - \frac{2}{l} + \frac{1}{poly(n)}$ approximation factor for the Generalized Steiner Tree problem, where l is the number of vertices desiring connectivity.

2. A $2 - \frac{2}{l} + \frac{1}{poly(n)}$ approximation factor for the Non-Fixed Point-to-Point Connection problem, where l is the number of sources/destinations.

3. A $3 + \frac{1}{poly(n)}$ approximation factor for the $\{0, 1, 2\}$ Survivable Network Design problem.

4. A $2 + \frac{1}{poly(n)}$ approximation factor for the Prize Collecting Steiner Tree problem.

The above problems are defined in Section 2 and Section 6. We mention that for the last problem above, the best approximation factor of $2 - \frac{1}{n-1}$ due to Goemans and Williamson [2] actually required n iterations of the clustering procedure; this takes $O(n(n + m) \log^2 n)$ time using our implementation of the clustering procedure. However, we show that a single run of the clustering procedure suffices to obtain a $2 + \frac{1}{poly(n)}$ approximation running in $O((n + m) \log^2 n)$ time.

Finally, we mention that Issue 1 is the bottleneck for implementing the algorithm of Williamson, Goemans, Mihail and Vazirani [6] for the Survivable Network Design Problem with higher connectivities in nearly linear time. The current fastest implementation of this algorithm is due to Gabow, Goemans and Williamson [3], which takes $O(c^2 n^2 + cn\sqrt{m \log \log n}))$ time, where c is the maximum connectivity desired. However, as stated in [3], the $\{0, 1, 2\}$ case mentioned above has an important practical application in the design of fibre optic telecommunication networks.

Dynamic Edge Splitting. Recall from above that edges fall into three categories and these categories need to be maintained over the several rounds in the clustering algorithm. Of these three categories, only the two categories corresponding to at least one endpoint active are really significant; the edge chosen in each round will be from one of these two categories. Our edge splitting technique effectively eliminates one of these categories, namely the one with both endpoints active. This happens as follows.

Consider an edge with both endpoints active. The idea is to split this edge at its midpoint and make the new split vertex an inactive singleton subset. Now both the edges resulting from this split have the property that they have at most one endpoint active. These two edges can in turn be split further when the need arises. We will split each edge $O(\log n)$ times, Subsequently, if one of the resulting pieces of this edge has both its endpoints active in some round, then we will treat this edge piece as if only one endpoint is active. We will thus introduce error into the clustering algorithm, possibly compromising the approximation factor. However, the size of this edge piece will be only a $\frac{1}{poly(n)}$ factor of the original edge length and this will enable us to keep the degradation in the approximation factor to $\frac{1}{poly(n)}$.

Thus, we will now have only two categories, namely, one or no endpoints active. This situation is easily handled.

Roadmap. In Section 2, we define some of the network design problems mentioned above and set up some preliminary definitions. In Section 3, we describe the details of the Goemans-Williamson clustering procedure, including its analysis. In Section 4, we describe our new implementation of the Goemans-Williamson clustering procedure as applied to the Generalized Steiner Tree problem. The applications to the Prize Collecting Steiner Tree problem, Non-Fixed Point-to-Point problem, and the $\{0, 1, 2\}$ Survivable Network problem are dealt with in the Appendix, which is organized as follows. Section 5 shows how to apply our algorithm to the Prize Collecting Steiner Tree problem. Section 6 briefly describes how our algorithm applies to the other problems.

2 Preliminaries

Let $G = (V, E)$ be an undirected graph with non-negative edge lengths d_{uv}, and having n vertices and m edges, respectively.

We will describe our algorithm as applied to the Generalized Steiner Tree problem defined below. We will then describe how our algorithm applies to the Prize Collecting Steiner Tree problem, also defined below. The other applications will be defined in Section 6.

The Generalized Steiner Tree Problem. In this problem, some of the vertices of G have colours associated with them. The aim is to choose a least cost subset of the edges so that the subgraph induced by these chosen edges has all vertices of any particular colour in a single connected component (a connected component could have vertices of different colours, though). Clearly this induced subgraph must be a forest.

The Prize Collecting Steiner Tree Problem. In this problem, each vertex of G has an associated penalty. The aim is to find a tree on a subset of the vertices, such that the cost of this tree plus the penalties of the vertices not in the tree is minimized.

3 The Goemans-Williamson Clustering Technique

We give an overview of the GW-clustering technique as applied to the Generalized Steiner Tree problem to obtain a forest whose cost is within a factor of 2 of the optimum forest.

The algorithm proceeds in two steps, Step A and Step B. The first step has several rounds, each of which identifies one new edge. The edges identified by the various round constitute a forest. The second step, called the pruning step deletes some of the above edges. The remaining edges constitute the final forest to be output.

Step A. In each round, the algorithm maintains a partition of V into disjoint subsets; some of these subsets are *active* and the rest are *inactive*. We denote the set containing vertex u by S_u. Note that S_u varies as the rounds progress. For each vertex u, the algorithm also maintains a length d_u, with the following intuitive significance: each edge $e = (u, v)$ such that S_u, S_v are distinct has a portion d_u of its length d_{uv} "inside" S_u and a portion d_v "inside" S_v. This leaves a portion of length $d_{uv} - d_u - d_v$; we call this the *exposed* portion. It is always the case that if S_u, S_v are distinct then $d_u + d_v \leq d_{uv}$; we call this condition the *edge inequality.*

At the beginning of the first round, the subsets and weights are as follows. Each coloured vertex is a singleton active subset and each uncoloured vertex is an singleton inactive subset. d_u is initially 0, for all u.

A general round proceeds as follows. All active subsets "grow" in a round, meaning that the values d_u for all u's contained in active sets increase, all at the same rate. This happens until the edge inequality becomes an equality for some edge $e = (u, v)$ with S_u and S_v being distinct (we say that edge e goes *tight*). This edge is now added to the forest output by Step A. Further S_u, S_v are now replaced by $S_u \cup S_v$, which is designated as active for the next round if and only if there exist two vertices of the same colour, one inside $S_u \cup S_v$ and another outside $S_u \cup S_v$.

The rounds continue until there are no more active subsets remaining. Note that in each round, the edge which goes tight is the edge with the least *addition time*, i.e., the least value of $\frac{d_{uv}-d_u-d_v}{f_u+f_v}$, where $f_u = 1$ if and only if S_u is active in this round.

Definition. Given a forest F and a partition of the vertices V into subsets, the forest induced in F by this partition is defined as the forest obtained from F by shrinking each subset into a single vertex and removing self-loops.

Step B. Recall that Step 1 returns a forest F; some of the edges in this forest will be discarded in this step. Consider an edge e in F and consider the subsets into which V is partitioned just before e was added. Consider the forest induced in F by this partition. Edge e is removed if and only if one of its endpoints in this forest is an inactive leaf (inactivity is with respect to the round which added e). The reason for this criterion will emerge in the analysis (see Claim 2 below).

3.1 Analysis

First, we show correctness. That the forest F returned by Step A above has all vertices of a particular colour in one tree of the forest is easy to see. This follows from the fact that any subset which contains some but not all vertices of the above colour continues to be active in Step A. Step B never disconnects vertices of the same colour. Therefore, the forest output by Step B has all vertices of one colour in one tree, as required.

Next, we describe how the approximation factor is bounded. This is based on two crucial claims, which require the following definition.

Definition. Recall Step A above and consider any subset S that is active in some round. This active subset would have come into existence in some round and would have grown in each subsequent round, until it combined with another set to give a new set. The total growth of S, i.e., the increase in d_u for each $u \in S$, over all these rounds is denoted by $Y(S)$.

Claim 1. The sum L of the $Y(S)$ values over all subsets S which were ever active in Step A, is a lower bound on the total cost of optimum forest.

The proof of this claim hinges on the following properties. Consider any edge e and all the sets S which were active in any round and in which $e = (u, v)$ has exactly one endpoint. These sets form a *laminar family*, i.e, any two such sets are either disjoint or one is contained inside the other. In addition, the sum of the $Y(S)$ values for these sets is at most the length d_{uv} of the edge e. In fact, these $Y(S)$ values can be viewed as "consuming" disjoint portions of the length d_{uv}, as illustrated in Fig.1. In this figure, the set S_4 only charges the portion of the edge e which lies in the annulus between S_4 and S_3, and likewise for the other sets.

To prove the claim, just charge the value of each $Y(S)$ to a portion of some edge in the optimum forest with exactly one endpoint in S (such an edge must always exist as S is an active set). By the previous paragraph, this charging can be done in such a way that distinct sets S charge distinct portions of any edge. The claim follows.

Claim 2. After Step B, the cost of the forest F obtained by the algorithm is within a factor of $2 - 2/l$ of the above lower bound, where l is the total number of coloured vertices. This claim can be shown as follows.

Consider any edge $e = (u, v)$ in F. Consider all the subsets S which were active in any round and in which e has exactly one endpoint. The sum of the $Y(S)$ values for these sets is equal to the length d_{uv} of the edge e. We charge each such subset S an amount $Y(S)$ for the edge e. Thus, over all edges in F, each subset S which is active in any round in Step A is charged an amount equal to $Y(S)$ multiplied by the *degree* of S, i.e., the number of edges in F which have exactly one endpoint in S. It now suffices to bound the sum U of $Y(S) * degree(S)$, over all subsets S which were active in any round in Step A.

To bound U, consider a particular round in Step A. Consider the forest F' induced in F by the partition of the vertices into subsets in this round. Note that the degree of the vertex corresponding to a particular subset in this forest is exactly the degree of the subset (as defined above). Each active subset S grows by an amount, say x, in this round, contributing amount $x * degree(S)$ to U

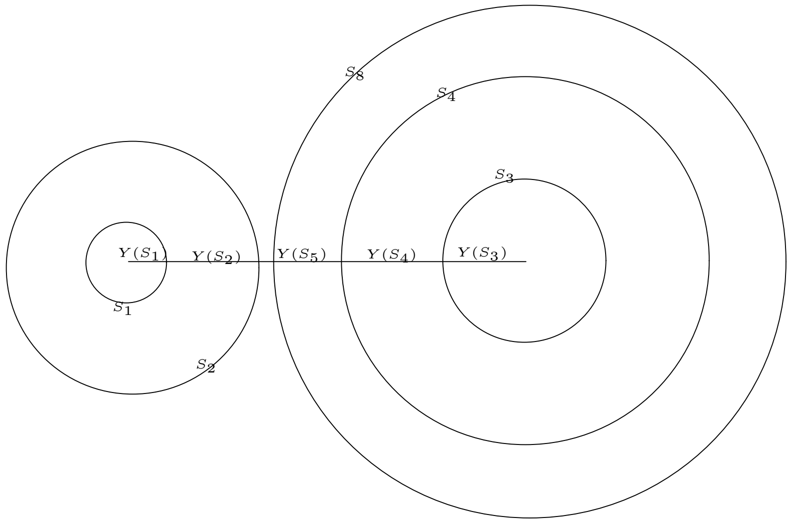

Figure 1: $Y()$ values on an edge.

and an amount x to L, the lower bound from Claim 1. By virtue of the pruning in Step B, it can also be seen that the leaves of F' always correspond to subsets active in the current round; these subsets have degree 1.

With all these properties, we are left with the following question: given any subset of the nodes in the forest F' necessarily including all the leaves of F', what is the ratio of the sum of the degrees of these nodes to the total number of these nodes? This ratio is clearly at most $2-2/|F'| \leq 2-2/l$, as required.

4 Our Algorithm

Issues 1 and 2 from the introduction (namely determining whether a newly formed subset is active or not, and implementing Step B), are easily addressed in $O(n \log^2 n)$ time; we skip the details here. Hereafter, we concentrate on Step A. First, we describe the key ideas in our algorithm. This is followed by a description of the data structures required, the precise algorithm description, and finally the analysis.

4.1 The Main Ideas

We avoid the problem of maintaining edges in two categories and modifying categories in each round by splitting an edge dynamically into $O(k \log n)$ successively halving pieces in such a way that the following property holds: each such piece (with one exception to be described later) is in category 1 or 0, namely, the categories with a denominator of 1 or 0 in the addition time, in all the rounds (see Section 3 for the definition of addition time). In fact, we will ensure that the following stronger property holds.

Definition. The new vertices we introduce when splitting edges will be called *s-vertices*.

Property 1. Consider any edge e. But for one of the edge pieces of e, all other pieces $e' = (u'v')$ of e satisfy the following property: if $S_{u'} \neq S_{v'}$ in any round then one of $S_{u'}, S_{v'}$ is a singleton inactive subset (corresponding to an s-vertex) in that round.

We call a piece of $e = (u, v)$ whose length is at most $\frac{d_{uv}}{n^k}$ a *terminal* piece. The above exception piece will be a terminal piece, created by the last split on e. Such a piece could be in category 2 (i.e., the category with denominator 2 in the addition time); however, we will treat it as if it is in category 1, thus dispensing with category 2 altogether. This leads to a new criterion for edge tightness in each round.

New Edge Tightness Criterion. In a round, all the $d()$ values for vertices in active subsets increase at the same rate until the following event happens: for some edge $e = (u, v)$ with $S_u \neq S_v$, either $d_u = d_{uv}$ or $d_v = d_{uv}$. This edge e is said to be tight and is added to the forest to be output. Contrast this to the earlier description (see Section 3) where the condition for an edge $e = (u, v)$ going tight was bidirectional, i.e., $d_u + d_v = d_{uv}$.

Edge Inequality Violation. A consequence of the new edge tightness criterion is that that the edge inequality could now be violated in some round, i.e., there could be edge pieces or edges $e = (u, v)$ for which $S_u \neq S_v$ but $d_u + d_v > d_{uv}$. Then the lower bound L in Claim 1 will no longer hold. We fix this by showing that the edge inequality holds approximately, as stated in Property 2 below. The first part of this property will follow easily from Property 1 and the edge tightness criterion. The second part will be shown in Lemma 1.

Property 2. Each edge $e = (u, v)$ satisfies the following:

1. All edge pieces $e' = (u', v')$ of e, except possibly one of the two terminal pieces, satisfy the following properties: as long as $S_{u'} \neq S_{v'}$, either $d_{u'}$ or $d_{v'}$ is 0, and $d_{u'} + d_{v'} \leq d_{u'v'}$.

2. As long as $S_u \neq S_v$, $d_u + d_v \leq (1 + \frac{1}{n^k})d_{uv}$.

Dynamic Edge Splitting. We now describe how edge splitting is performed.

Prior to the first round, we split each edge into two half-edge pieces e_1, e_2. This ensures that at the beginning of the first round, all edges pieces are in category 1 or 0 (because the new vertex splitting an edge into two halves is not a coloured vertex and therefore the singleton subset containing this vertex is not active). Property 1 thus holds at the beginning. To maintain this property in subsequent rounds, we do the following.

Consider a particular round and consider an edge piece $e' = (u', v')$ of edge $e = (u, v)$ which is not a terminal piece. At the beginning of this round, it will be the case that at least one of u', v' (say, v') will be an s-vertex and that this vertex will be in a singleton inactive component. To maintain this property at the end of the round as well, we will do the following.

Suppose v' joins some other active set to form a new non-singleton set $S_{v'}$ for the next round. Consider the values of $d_{u'}, d_{v'}$ at the end of the above round. $d_{v'}$ is necessarily 0 because v' has just joined $S_{v'}$ and was inactive in all prior rounds. The edge piece e' will now be repeatedly split (see Fig.2) by adding a sequence of vertices on e' at distances $\lceil\frac{d_{u'v'}}{2}\rceil, \lceil\frac{d_{u'v'}}{4}\rceil, \lceil\frac{d_{u'v'}}{8}\rceil, \cdots, \lceil\frac{d_{u'v'}}{2^i}\rceil$ from v', where i is the least number such that one of the following two properties holds:

1. $d_{u'v'} - \lceil\frac{d_{u'v'}}{2^i}\rceil > d_{u'}$.

2. $\lceil\frac{d_{u'v'}}{2^i}\rceil \leq \frac{d_{uv}}{n^k}$. Note here that the right-hand-side has d_{uv} and not $d_{u'v'}$; the aim is to ensure that the total number of splits on an original edge e is $O(k \log n)$.

In fact, it suffices to add the last two vertices above (i.e., u'', u''' in Fig.2).

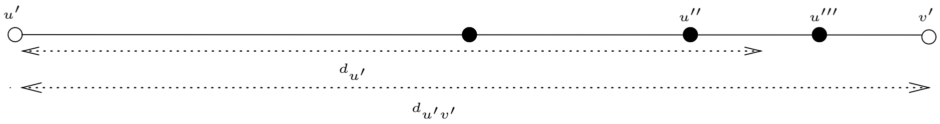

Figure 2: Splitting e' with 3 s-vertices.

Of the above new vertices added, those within distance $d_{u'}$ from u' will be made part of $S_{u'}$ (all resulting edge pieces between u' and u'' in Fig.2 will be added to the forest to be output). The only other vertex added (u''' in Fig.2) will be made a singleton inactive subset, thus ensuring that Property 1 continues to hold. The edge piece e' will now be discarded.

4.2 Data Structures Required

We describe the required data structures before giving the detailed algorithm.

(a) A Heap Structure H: We need a structure which maintains, for each subset S encountered in Step 1, a heap *heap*(S) of the edge pieces having at least one point in this subset and supporting the operations listed below. Edge pieces which are contained completely in this subset could be present in the heap but they they will be discarded when they are considered for addition to the forest, and are therefore irrelevant. The key of an edge piece $e = (u, v)$ in this heap will be $d_{uv} - d_u$ (assuming $u \in S, v \notin S$). An important point to note is that each edge piece $e = (u, v)$ is present in both *heap*(S_u) and in *heap*(S_v), possibly with different keys in each heap. The heaps for the subsets S support the following operations.

1. Insert, Delete, Meld: Insert a new edge piece with a specified key into $Heap(S)$, delete an edge piece from *heap*(S), meld two specified heaps *heap*(S'), *heap*(S'').

2. Delete-Min: Find the edge piece with the minimum key value over heaps for active subsets.

2. Findkey: Given an edge e and an endpoint u of e, find the key value of e in the heap *heap*(S_u).

3. Offset: Decrease all keys in the heaps for active subsets by a given number.

4. Change-Activity: Change the activity status of a specified heap *heap*(S) from active to inactive or vice versa.

5. Add-Subset: Add *heap*(S) for a new subset S of specified activity.

Standard heaps (e.g., Binomial Heaps or Fibonacci heaps) can be easily augmented with Offset, Findkey, and Change-Activity operations. With this, all the above operations can be supported in $O(\log n)$ worst case time. We skip the details here.

(b). A Union-Find data structure to identify which edge pieces are completely inside a subset and which have only one endpoint inside a subset.

4.3 The Algorithm Details

The algorithm begins by splitting each edge into two pieces, making each vertex a singleton subset, each coloured vertex active, and initializing the heap structure H with each edge piece having key value equal to its length.

In each round the algorithm performs the following tasks.

Step 1: Choosing the Next Edge. First, it performs Delete-Mins from H repeatedly until an edge piece connecting distinct subsets is found. This edge piece $e' = (v', w')$ is now added to the forest to be output. Let $key(e')$ be the key of e' in either $heap(S_{v'})$ or $heap(S_{w'})$, whichever is active (if both are active, take the smaller of the two keys). $key(e')$ can be found using the Findkey operation. All active subsets have "grown" by $key(e')$ in this round, so H is Offset by $key(e')$. Then the sets $S'_{v'}, S'_{w'}$ are unioned.

Step 2: Splitting Edges. Note that at least one of $S_{v'}, S_{w'}$ must have been active at the beginning of the current round for the two subsets to combine in this round. It follows that at most one of v', w' can have the property that it is an s-vertex and is contained in an inactive singleton set at the beginning of the current round. If neither v', w' has this property then we skip this step. Otherwise, without loss of generality, let v' have this property. Being an s-vertex, v' has exactly one edge piece other than e' incident on it. We will split this edge piece $f = (u', v')$ as follows.

To split f, $d_{u'}$ is needed (recall how splits are performed from above). This is not directly available, but can be computed using the Findkey operation on f, $S_{u'}$ and then using the relation that the key for f in $heap(S_{u'})$ is $d_{u'v'} - d_{u'}$. Once $d_{u'}$ is determined, f is split as described earlier, adding one new singleton inactive subset (corresponding to u''' in Fig.2) to the Union-Find Structure and the heap structure, and adding the two new edges pieces incident on u''' to the heap structure; all other new s-vertices created are made part of $S_{u'}$ and all other new edge pieces created are added to the forest to be output. The edge piece f itself is removed from all the structures.

Step 3: Melding Heaps. If neither v' nor w' have the above property required for edge splitting (this will happen only if e' is a terminal edge piece, by Property 1), then $heap(S_{v'}), heap(S_{w'})$, are melded, the active status of $S_{v'} \cup S_{w'}$ is determined, and a Change-Activity operation on H is performed, as necessary.

Total Number of Heap Operations. The above procedure has $O(n)$ rounds and splits each edge $O(k \log n)$ times. The total number of heap operations performed is $O(k(m+n)\log n)$. The total time taken is thus $O(k(m+n)\log^2 n)$.

4.4 Analysis

We prove the critical lemma showing Property 2 first. The first part of this property follows directly from the new edge tightness criterion and from Property 1. The second part of this property is shown below.

Lemma 1 *For any edge $e = (u, v)$, as long as $S_u \neq S_v$, $d_u + d_v \leq (1 + \frac{1}{n^k}) d_{uv}$, where k is a constant fixed above at the beginning of Section 4.1.*

Proof. For a contradiction, suppose the lemma is false. Consider the first instant at which $d_u + d_v \geq d_{uv}$ and $S_u \neq S_v$. Consider the situation just before the above instant. At this point, $d_u + d_v < d_{uv}$. By virtue of Property 1 and by the way edges are split into pieces, there must exist a terminal piece $e' = (u', v')$ such that $u' \in S_u$ and $v' \in S_v$. The length of a terminal piece on e is at most d_{uv}/n^k. Therefore, the key for e' must be at most d_{uv}/n^k in the bottom level queues for both S_u, S_v. It follows that $d_u + d_v$ can never exceed $d_{uv}(1 + \frac{1}{n^k})$, as long as $S_u \neq S_v$. $\square$

Lemma 2 *The above algorithm gives an approximation factor of* $(2 - \frac{2}{l})(1 + \frac{1}{n^k}) \leq 2$, *where* l *is the number of coloured vertices.*

Proof. Recall Claims 1 and 2 from Section 3.1.

Claim 1 stated that the sum L of the $Y(S)$ values over all sets S which were ever active in Step 1 is a lower bound on the total cost of optimum forest. This claim was a consequence of the following property: consider any edge e and all the sets S which were active in any round and in which $e = (u, v)$ has exactly one endpoint; the sum of the $Y(S)$ values for these sets is at most the length d_{uv} of the edge e.

For our algorithm, the last statement above needs to be modified to: the sum of the $Y(S)$ values for these sets is at most $(1 + \frac{1}{n^k})d_{uv}$ by Lemma 1. With this modification, the proof of Claim 1 shows that $\frac{L}{1+1/n^k}$ is a lower bound.

Claim 2 which shows the upper bound needs the following modification. Consider any edge $e = (u, v)$ in F. Consider all the subsets S which were active in any round and in which e has exactly one endpoint. The sum of the $Y(S)$ values for these sets is at least (and not necessarily equal to) the length d_{uv} of the edge e. In spite of this modification, the proof of Claim 2 holds unchanged.

The lemma follows directly from the modified Claim 1 and Claim 2 together. $\square$

Theorem 3 *There exists an* $O(k(n + m) \log^2 n)$ *time, linear space algorithm to find a Generalized Steiner Tree having cost within factor* $(2 - \frac{2}{l})(1 + \frac{1}{n^k}) \leq 2$ *of the optimal.*

Proof. It suffices to show the time and space bounds. Step B and the determination of whether or not the new subset created by merging two sets in each round is active can be done in $O(n \log^2 n)$ time; we skip the details here. Consider our algorithm for Step A next. As stated above in Section 4.2, the time taken for heap operations is $O(k(n + m) \log^2 n)$; this dominates the total time.

A naive implementation will take $O(km \log n)$ space, as each edge could be broken into $O(k \log n)$ pieces. However, only $O(1)$ of these pieces need to be maintained explicitly at any point. The space can easily be brought down to $O(n + m)$ using this observation. $\square$.

References

[1] A. Agrawal, P. Klein, R. Ravi. *When trees collide: An approximation algorithm for the generalized steiner problem on networks*, SIAM Journal on Computing, 24(3), pp. 440–456, 1995.

[2] M. Goemans, D. Williamson. *A general approximation technique for constrained forest problems*, SIAM Journal on Computing, 24(2), pp. 296–317, 1995.

[3] H. Gabow, M. Goemans, D. Williamson. *An efficient approximation algorithm for the survivable network design problem*, Proceedings of IPCO, pp. 57–74, 1993. To appear in *Math. Programming.*

[4] P. Klein. *A data structure for bicategories with applications to speeding up an approximation algorithm*, Information Processing Letters, 52, pp. 303–307, 1994.

[5] R. Ravi and P. Klein. *When cycles collapse: A general approximation technique for constrained two connectivity problems*, Proceedings of IPCO, pp. 39–55, 1993.

[6] D. Williamson, M. Goemans, M. Mihail, V. Vazirani. *A primal-dual approximation algorithm for generalized steiner network problems*, Combinatorica, 15, pp. 435–454, 1995.

Appendix

5 The Prize Collecting Steiner Tree Problem

First, we sketch the Goemans-Williamson approximation algorithm [2] for this problem. Each vertex of the graph is considered in turn to be a root node of the desired tree. This chosen root is given a *potential* of 0 and all other vertices are given a potential equal to their respective penalties. Then the clustering algorithm above (i.e., Step A) is run with the following definition of active subsets: initially, all vertices with non-zero potentials are active; subsequently, a subset S is active if and only if the sum of the $Y(S')$ values for subsets S' of S is less than the sum of the potentials of the vertices in S, and S does not contain the root. Next, Step B is performed as before. Finally, all components of the resulting forest, except that containing the root, are discarded. The root component is the desired tree. The best such tree (the one which minimizes sum of tree-edge lengths plus sum of non-tree vertex potentials) over all possible roots is the final answer. The time taken by the Goemans-Williamson algorithm was $O(n^3 \log^2 n)$, $O(n^2 \log n)$ per root. The approximation factor guaranteed was $2 - \frac{1}{(n-1)}$.

5.1 Our Algorithm

We show how to get a $2 + \frac{1}{n^k}$ approximation factor in $O(k(n+m)\log^2 n)$ time. To accomplish this, we cannot afford to run the clustering algorithm once for each root. Instead, we run the clustering algorithm (i.e., Step A) just once, with potentials equal to the penalties and with no vertex specially designated as root. Then we perform Step B. Next, we perform the following Step C.

Step C. We consider each vertex v in turn, and identify a tree containing v. The best tree over all v, i.e., the one which minimizes sum of tree-edge lengths plus sum of non-tree vertex potentials, is the tree that is output.

The tree for v is determined as follows. Consider the tree containing v before Step B. Imagine that Step B is now performed on this tree but without ever disconnecting any inactive component containing v (i.e., if the removal of an edge disconnects v from this tree then this removal is not performed). The resulting tree is the tree associated with vertex v. We claim that the total time taken to compute the cost of such trees over all vertices v is linear.

We now have the following lemma.

Lemma 4 *The above algorithm for the Prize Collecting Steiner Tree problem returns a solution within a factor $2 + \frac{1}{n^k}$ of the optimal.*

Proof. Suppose the optimum tree contains vertex v. Consider the forest returned by our algorithm and take the tree T containing v. We will show the above approximation factor for this tree; the best tree will be no worse.

Lower Bound. Define $Y'(S) = \frac{Y(S)}{1+\frac{1}{n^k}}$. We claim that $L' = \sum_{S|v \notin S} Y'(S)$ is a lower bound. This is seen as follows.

First, we distribute the total sum L' over the vertices in such a way that no vertex gets a charge more than its potential; vertex v gets no charge and vertices outside T get charges equal to their potentials. This is done as follows.

Each subset S, $v \notin S$, distributes its $Y'(S)$ to the two subsets S', S'' whose merger formed S. If S' (S'', respectively) is inactive then $Y'(S)$ is moved to S'' (S', respectively). If neither is inactive, then $Y'(S)$ is distributed over S', S'' in the ratio of their residual potentials (for a subset A, residual potential is defined as the sum of the potentials of vertices inside A minus the sum of the $Y'()$ values for subsets B of A). The subsets S', S'' will distribute their own $Y'()$ value along with the value inherited from S further down to their included subsets, until all the charges lie on individual vertices. It is easily seen that no vertex is charged more than its potential, that v is left uncharged, and that vertices not in T get charges equal to their potentials (Step C is necessary for this last property to hold).

Next, we claim that if the optimum chose some collection C of the vertices along with v in its tree then the sum of the charges on the vertices in C is a lower bound on the sum of the edge lengths of this tree. To see this, note the following fact: if Step A is run on all vertices with penalties strictly smaller than the charges assigned then vertex v stays in a singleton subset through all the rounds (this is because subsets containing v have not been considered in the above charging process). It follows that the sum of the charges on the vertices, which in turn equals L', is a lower bound.

Upper Bound. Finally, as in Claim 2 and in Lemma 2, the upper bound is $\sum_S Y(S) * degree(S)$, where $degree(S)$ is the number of edges in the forest output by Step B having exactly one endpoint in S. Note that active subsets containing v contribute to this upper bound, but not to the lower bound L'; all other active subsets contribute to both the bounds. Then, as in Claim 2, we are left with the following question: given any collection of m nodes in the forest F' including all but possibly one of the leaves of F', what is the ratio of the sum of the degrees of these nodes to $m-1$ (we have $m-1$ and not m here because the subset containing v in the current round does not contribute to the lower bound; this subset may or may not contribute to the upper bound)? This ratio is clearly at most 2.

The lemma follows. $\square$

Theorem 5 *There exists an $O(k(n+m)\log^2 n)$ time algorithm to find a Prize Collecting Steiner Tree having cost within factor $2 + \frac{1}{n^k}$ of the optimal.*

Proof. It suffices to show the time bound. Steps B, C can be implemented in linear time. Step A takes $O(k(n+m)\log^2 n)$ time as before. $\square$

6 Other Applications

We define two other problems for which our technique speeds up known approximation algorithms.

Non-Fixed Point-to-Point Connection Problem. In this problem, a set of source vertices S and a set of destination vertices T is given. The aim is to find a forest in which each component has the same number of S and T vertices.

The Goemans-Williamson [2] technique applies to this problem as well and gives an approximation factor slightly less than 2. For this problem, a set is designated active if the number of S and T vertices in it is not identical. With this definition, determining active/inactive sets in each round in Step A and performing Step B in $O(n)$ time is easy. Thus our implementation of Step A gives an $O(k(n+m)\log^2 n)$ time approximation algorithm for this problem as well at the expense of a $(1+\frac{1}{n^k})$ factor in the approximation ratio.

The $\{0,1,2\}$ Survivable Network Problem. In this problem, each vertex v is given a label $r_v = 0, 1, 2$. The aim is to choose a collection of edges of minimum cost so that each pair of vertices v, w has $\min\{r_v, r_w\}$ edge disjoint paths between them.

The 3-approximation algorithm in [6] for this problem uses two runs of the Goemans-Williamson clustering procedure. The first one is routine and runs in the above claimed time bounds using our implementation. The second one is more involved in terms of the definition of active sets and in the implementation of a counterpart of Step B. However, Gabow, Goemans and Williamson [3] have shown that Step B can be implemented in $O(n)$ time (even when higher connectivity is desired). We observe that determining active sets in each round in Step A can also be performed efficiently, so that the total time taken using our implementation is still $O(k(n+m)\log^2 n)$ and the approximation factor is $3(1+\frac{1}{n^k})$. We skip the details here.