Graph Sparsification Heuristics

Andrew Smith

Advisor: Dan Spielman

May 5, 2008

Abstract

We consider the problem of graph sparsification. Given a graph $G$, we would like to produce a graph $G'$ with fewer edges, such that certain properties of $G$ are maintained in $G'$. The Benczúr-Karger algorithm produces graph sparsifications that have $O(n \log n/\epsilon^2)$ edges, while preserving the values of all cuts within a factor of $(1 \pm \epsilon)$.

We present two algorithms that improve upon the Benczúr-Karger algorithm for graph sparsification. The first algorithm runs asymptotically faster than the Benczúr-Karger algorithm, and often produces sparsifications with somewhat fewer edges. The second algorithm has the same asymptotic running time as the Benczúr-Karger algorithm, but runs faster in practice. In addition, it produces sparsifications that have even fewer edges than sparsifications produced by first algorithm. Both algorithms have the property that they can be run longer to get fewer edges. The Benczúr-Karger algorithm does not have this property. We test both algorithms on various graphs and analyze the results.

1 Introduction

In [5], we analyzed the Benczúr-Karger algorithm for graph sparsification. Recall that given a graph $G$, the Benczúr-Karger algorithm produces a graph $G'$ of $O(n \log n/\epsilon^2)$ edges, such that all cut values in $G'$ are within a factor of $(1 \pm \epsilon)$ of their original values in $G$. This algorithm runs in time $O(m \log^2 n)$ for unweighted graphs, and in time $O(m \log^3 n)$ for weighted graphs. We found in [5] that for certain graphs, the Benczúr-Karger algorithm actually preserves more than just the values of each cut: it preserves the quantity $x^T L x$ for all $x \in \mathbb{R}^n$ in the graphs we tested.

However, the Benczúr-Karger algorithm has some problems. While it produces a graph with $O(n \log n/\epsilon^2)$ edges, the constant hidden in the big-O notation is often high (around 2-4). This limits the usefulness of the sparsifications Benczúr-Karger outputs. Furthermore, the running time of the algorithm also has a large constant factor. This further limits its usefulness in applications such as speeding up linear system solving.
To overcome these problems, we explored a natural simplification to the Benczúr-Karger algorithm in [5]. This simplified algorithm runs much faster (in time $O(m)$ for unweighted graphs, and time $O(m + n \log n)$ for weighted graphs). However, we determined that this simplification produces a graph with $O(n \log^2 n/\epsilon^2)$ edges for unweighted graphs, and $\max(O(n \log n \log n W/\epsilon^2), |E|)$ edges for weighted graphs. These edge bounds are too large to make the simplification useful.

In this paper, we look at other simplifications to the Benczúr-Karger algorithm. We present two algorithms that improve upon the above problems of the Benczúr-Karger algorithm.

2 The original simplification

In [1], Benczúr and Karger proved the following:

**Theorem 1.** If we sample every edge of a graph $G$ with probability $p_e = \Omega(\ln n \epsilon^{-2})$ and assign each picked edge weight $1/p_e$, the resulting graph would preserve all cut values to within a factor of $(1 \pm \epsilon)$.

In that expression, $k_e$ is the maximum value of $k$ such that a maximal, vertex-induced $k$-connected component of $G$ contains edge $e$. This quantity is called the strong connectivity of edge $e$, and a maximal, vertex-induced $k$-connected component of $G$ is called a $k$-strong component.

This quantity is closely related to the standard edge connectivity of edge $e$, $\lambda_e$. $\lambda_e$ is the value of a minimum cut separating the endpoints of edge $e$. We have that $\lambda_e \geq k_e$, since the existence of a subgraph of $G$ of minimum cut $k$ that includes edge $e$ implies that no cut of $G$ separating $e$'s endpoints can have fewer than $k$ edges.

The following conjecture was mentioned in [1]:

**Conjecture 1.** If we replace $k_e$ with $\lambda_e$ in theorem 1, the resulting graphs still preserve all cut values to within a factor of $(1 \pm \epsilon)$.

Benczúr and Karger could not find any counterexample to this conjecture, and our results from [5] indicate that sampling using $\lambda_e$ in place of $k_e$ approximately preserves $x^T L x \forall x \in \mathbb{R}^n$ for the graphs we tested. This implies that the values of all cuts are preserved (in the graphs we tested), as each cut $C$ corresponds to a characteristic vector $x$ in $\{0, 1\}^n$ such that $x^T L x = val(C)$. We used this conjecture to come up with our original simplification to the Benczúr-Karger algorithm in [5].

Our original simplification of the Benczúr-Karger algorithm used one of its subroutines. This subroutine is an algorithm created by Nagamochi and Ibaraki. The Nagamochi-Ibaraki algorithm calculates a height $h_e$ for each edge $e$. The algorithm partitions $G$ into sets of edges $E_1, E_2, ..., E_k$ of $G$, such that $E_i$ is a spanning forest of $G - \bigcup_{j=1}^{i-1} E_j$. This is equivalent to “peeling off” spanning forests of $G$ without replacement, where $E_i$ is the $i$th spanning forest that was
removed. The height $h_e$ of an edge $e$ is simply the value of $i$ such that $E_i$ contains $e$.

Nagamochi and Ibaraki proved the following theorem.

**Theorem 2.** Let $E_1, E_2, ..., E_k$ be any partition of the edges of $G$, such that $E_i$ is a spanning forest of $G - \cup_{j=1}^{i-1} E_j$. Then $\lambda_e \geq h_e$.

**Proof.** Proof in [3]. \qed

**Corollary 1.** If conjecture 1 is true, we can sample each edge as above, where $k_e$ is replaced with $h_e$, and still get a graph whose cut values are approximately preserved.

**Proof.** Since $\lambda_e \geq h_e$, we have $\frac{\ln n}{\epsilon^2 h_e} \geq \frac{\ln n}{\epsilon^2 \lambda_e}$. This satisfies conjecture 1. \qed

Corollary 1 motivated the original simplification we analyzed in [5]:

1. Run the Nagamochi-Ibaraki algorithm once, setting $h_e$ to the resulting height of each edge determined by the algorithm.

2. Sample each edge in $G$ with probability $p_e = \Theta\left(\frac{\ln n}{\epsilon^2 h_e}\right)$, giving each edge a weight of $1/p_e$ if included.

The running time of the Nagamochi-Ibaraki Algorithm for unweighted graphs is $O(m)$, which is asymptotically faster than the Benczúr-Karger algorithm (which has running time $O(m \log n)$ if we make a modification to take into account conjecture 1). However, as we mentioned in [5], this simplification produces a sparsified graph with too many edges to be useful in many applications. In an unweighted graph, each spanning forest has at most $(n-1)$ edges, and there are no more than $m$ such spanning forests. We therefore have:

$$\sum_{e \in E} \frac{1}{h_e} \leq \sum_{i=1}^{m} \frac{n-1}{i} = \Theta(n \log n) \quad (1)$$

This gives us the following upper bound on the expected number of edges produced in our simplified algorithm. Let $X_e$ be an indicator variable for edge $e$, with value 1 if edge $e$ is included in our sparsification, and 0 otherwise.

$$E\left[\sum_{e \in E} X_e\right] = \sum_{e \in E} p_e = \sum_{e \in E} \Theta\left(\frac{\ln n}{\epsilon^2 h_e}\right)$$

$$= \Theta\left(\frac{\ln n}{\epsilon^2}\right) \sum_{e \in E} \frac{1}{h_e}$$

$$= O\left(\frac{n \log^2 n}{\epsilon^2}\right)$$

The last step follows from (1).

This edge bound is too high for many applications. For our sparsifications to be useful, we need them to have $O(n \log n/\epsilon^2)$ edges.
3 Adding randomness to the algorithm

3.1 Modification to the Nagamochi-Ibaraki algorithm

The Nagamochi-Ibaraki algorithm for determining heights of each edge is a deterministic algorithm. It produces heights (and therefore sampling probabilities for each edge) that result from “peeling off” the same spanning forests in the same order, each time the algorithm is run on a given graph. However, Nagamochi-Ibaraki proved that $\lambda_e \geq h_e$ regardless of which spanning forests were peeled off to produce the $h_e$ values. Any process that peels off spanning forests without replacement will have $h_e$ values that satisfy $\lambda_e \geq h_e$.

This indicates that we may be able to improve the edge bound of the Nagamochi-Ibaraki algorithm if we introduce randomness to the algorithm. If we select the spanning forests to be peeled off in different ways, the resulting heights for any edge may vary in different trials. Because $\lambda_e \geq h_e$ for any possible $h_e$ value obtained in the process of peeling off spanning forests, we can replace $\lambda_e$ in conjecture 1 with the maximum height for each edge over multiple trials (instead of a single $h_e$ value for each edge determined in one trial). Using larger height values will lower our sampling probabilities for each edge, and thus reduce the expected number of edges in our sparsifications.

For example, given a connected graph, the Nagamochi-Ibaraki algorithm always sets $n - 1$ edges to height 1 (since the first spanning tree that is peeled off has $n - 1$ edges). However, it is likely that many of these edges have $\lambda_e > 1$. In fact, the only way that all $n - 1$ edges of height 1 have $\lambda_e = 1$ is if $G$ itself is a tree.

Similarly, it is likely that $\lambda_e > h_e$ for many edges of any height value. Benczúr and Karger proved in [1] that $\sum_{e \in E} 1/\lambda_e \leq n - 1$. We have shown that $\sum_{e \in E} 1/h_e \leq \Theta(n \log n)$. The latter bound is often tight (and is exactly tight when a connected graph remains after peeling off each spanning tree in all iterations but the last). Thus, when we use $h_e$ values from a single iteration of the Nagamochi-Ibaraki algorithm, $\lambda_e \geq h_e$ is rarely tight for edges of high connectivity. This indicates that there may be room for improvement.

We will begin by attempting to add randomness directly to the Nagamochi-Ibaraki algorithm. Our goal is to select the spanning forest to “peel off” on each iteration as “randomly” as possible.  

Intuitively, the more randomness we add to the algorithm, the more the $h_e$ values will vary in different trials. This leads to higher maximum $h_e$ values (over all trials), and thus fewer edges in our sampled graph.

For reference, we include the Nagamochi-Ibaraki algorithm for unweighted graphs as Algorithm 1. Note that the Nagamochi-Ibaraki algorithm does not determine edge heights by explicitly peeling off spanning trees without replacement, as this would take time $O(n^2)$. Instead, the algorithm implicitly determines all spanning trees at once (where nodes with the same $h_e$ values at

\footnote{One option is to peel off each spanning forest uniformly at random from all possible spanning forests of the remaining subgraph. However, the running time for picking a spanning forest uniformly at random from all possible spanning forests is prohibitive for our purposes.}
the end of the algorithm belong to the same spanning forest). Nagamochi and Ibaraki proved that the set of edges of height \( i \) \((E_i)\) forms a spanning forest of \( G - \bigcup_{j=1}^{i-1} E_j \), which is the property that guarantees \( \lambda_e \geq h_e \) in Theorem 2.

**Algorithm 1** Nagamochi-Ibaraki algorithm for unweighted graphs

**Require:** \( G = (V, E) \)

**Ensure:** \( h_e > 0 \quad \forall e \in E \)

1: Label all nodes \( v \in V \) and all edges \( e \in E \) as unscanned
2: \( r_v \leftarrow 0 \quad \forall v \in V, \quad h_e \leftarrow 0 \quad \forall e \in E \)
3: while there exist unscanned vertices do
4: Pick unscanned node \( v \in V \) with largest \( r_v \)
5: for all unscanned edges \( e = (v, u) \) do
6: \( h_e \leftarrow r_u + 1 \)
7: \( r_u \leftarrow r_u + 1 \)
8: Mark edge \( e \) as scanned
9: end for
10: Mark vertex \( v \) as scanned
11: end while

On line 4, the algorithm picks the unscanned node with largest \( r \)-value. However, it is often the case that multiple nodes have the same \( r \)-value. For example, in the first iteration, all nodes have \( r_v = 0 \). In this case, the way the algorithm breaks the tie will greatly affect what happens during future iterations (because the node that is picked first determines where in the graph the algorithm starts growing the spanning forests). The current algorithm breaks the tie arbitrarily (but deterministically).

We will modify the algorithm to break ties randomly. If we assign an ordering to the vertices, we can have \( r_v \)-value ties broken by picking which vertex is lowest in the ordering. To implement this, we will first pick a random permutation \( \sigma \). Instead of initially setting \( r_v = 0 \), we will set \( r_v = \delta \sigma_v \), where \( \delta \) can be any constant strictly less than \( 1/n \).

This ensures that initially, all \( r_v \) values are distinct. The first vertex picked will be the one that came first in the permutation \( \sigma \). The condition that \( \delta < 1/n \) ensures that the perturbations in the initial \( r \)-values won’t affect the order in which vertices are picked when there isn’t a tie. To ensure that the random perturbations in the initial \( r \)-values won’t affect the output values, we will modify line 6 to use the floor of \( r_u \). Algorithm 2 is our modified algorithm.

Figure 1 contains an example run of one trial of Algorithm 2 on the complete graph of 5 vertices.

Note that the correctness of Algorithm 2 follows directly from the correctness of the Nagamochi-Ibaraki algorithm, and the running time is the same for our purposes. ²

²The implementation that Nagamochi-Ibaraki gives does not allow fractional \( r_v \) values for their algorithm on unweighted graphs. We will solve this problem by using the implementation of their algorithm on weighted graphs, which uses a priority queue to pick the vertex \( v \) with
Algorithm 2 Randomized Nagamochi-Ibaraki algorithm for unweighted graphs

Require: $G = (V, E)$
Ensure: $h_e > 0 \ \forall e \in E$

1: Label all nodes $v \in V$ and all edges $e \in E$ as unscanned
2: $\sigma \leftarrow$ Random permutation of integers 1 to $n = |V|$ for $i = 1$ to $n$
3: $r_i \leftarrow \sigma_i/2n$ for $i = 1$ to $n$
4: $h_e \leftarrow 0 \ \forall e \in E$
5: while there exist unscanned vertices do
6: Pick unscanned node $v \in V$ with largest $r_v$
7: for all unscanned edges $e = (v, u)$ do
8: $h_e \leftarrow \lfloor r_u \rfloor + 1$
9: $r_u \leftarrow r_u + 1$
10: Mark edge $e$ as scanned
11: end for
12: Mark vertex $v$ as scanned
13: end while

Since $\lambda_e \geq h_e$ in Algorithm 2, we can run this algorithm several times, and let $h^*_e$ be the maximum $h_e$ value over all trials (for each edge). We can then sample to get our sparsification, using the $h^*_e$ values in Corollary 1.

3.2 Experimental results

First, we will look at how many edges there are in sparsifications that Algorithm 2 produces. We will run Algorithm 2 for various numbers of trials. For each test of $i$ trials, we will calculate the quantity

$$C = \frac{1}{n} \sum_{e \in E} \frac{1}{h^*_e}$$

where $h^*_e$ is the maximum value of $h_e$ over all $i$ trials. Note that $C$ is the constant multiple of $\ln n$ in the expected number of edges our algorithm produces (i.e. our algorithm produces sparsifications with an expected number of edges of $C n \ln n$). We would like this quantity to be as low as possible. We expect that $C = \ln n$ for one trial (as we derived in section 2), but much lower for multiple trials. Figure 2 contains the results for running Algorithm 2 for up to 10 trials.

We see that running Algorithm 2 for one trial yields approximately $C = \ln n$, as expected. Once the algorithm is run for 2 or more trials, the number of edges produced drops significantly. $C$ approaches 1 as the number of trials increases.

However, $C$ does not always decrease as the number of trials increases. The results vary widely, though running multiple trials clearly results in fewer edges compared to one trial.

This increases the running time from $O(m)$ to $O(m + n \log n)$. However, this increase is irrelevant for our purposes. We are only looking at graphs with $\Omega(n \log n)$ edges. A graph with fewer edges would not need sparsification to $O(n \log n)$ edges in the first place.
Figure 1: *Example run of Algorithm 2 on the complete graph with 5 vertices*

We also examine the accuracy of sparsifications resulting from the heights determined in Algorithm 2. As in [5], we would like to determine how much the quantity $x^T L x$ varies between sparsifications and the original graph. In particular, we would like to determine the constants $a$ and $b$ that satisfy

$$ax^T L' x \leq x^T L x \leq \frac{1}{b} x^T L' x \quad \forall x \in \mathbb{R}^n$$

for each sparsification. $a$ and $b$ are therefore the following two quantities:

$$a = \max_{x \in \mathbb{R}^n} \frac{x^T L x}{x^T L' x} \quad \text{and} \quad b = \max_{x \in \mathbb{R}^n} \frac{x^T L' x}{x^T L x}$$

The quantities $a$ and $b$ are often referred to as the *supports* of a sparsification.

Let $h^*_e$ be the heights generated by 10 trials of Algorithm 2. We will test sparsifications that use sampling probabilities $\frac{d \ln n}{\epsilon h^*_e}$ for various values of the sampling constant $d$. Our results appear in Table 1. Table 1 also includes results for sparsifications of the same graphs using the Benczúr-Karger algorithm (with a slight modification to take into account Conjecture 1). $m'$, $a$, and $b$ are the results using the heights from Algorithm 2, while $m'_{bk}$, $a_{bk}$, and $b_{bk}$ are the results using the heights from the Benczúr-Karger algorithm.
For $d = 1$, the resulting values of $a$ and $b$ from Algorithm 2 are about $2 - 3.5$ times higher than the values generated by the Benczúr-Karger algorithm. However, the Benczúr-Karger algorithm produces sparsifications with about $1.7 - 4$ times as many edges as the number of edges that Algorithm 2 produces.

For larger values of $d$, the edge-bound improvement of Algorithm 2 over Benczúr-Karger is lower (and for $d = 4$, Benczúr-Karger actually produced a lower number of edges than Algorithm 2). However, the accuracy improvement of Benczúr-Karger over Algorithm 2 is also lower for larger values of $d$. The reason the edge bound improvement of Algorithm 2 over Benczúr-Karger decreases as our sampling constant $d$ increases is most likely due to rounding. This is because we always round $\frac{d \ln n}{cH^2}$ down to 1 if it is above 1 (since it is a probability). As the sampling constant rises, more and more of the sampling probabilities rise above 1 (and have to be rounded down). This behavior is not present in the random graph where $p = 1/2$, since the graph has high connectivity (resulting in larger heights, which make the sampling probabilities well below 1).

Even though the resulting accuracy of sparsifications produced by Algorithm 2 is somewhat worse than the accuracy of sparsifications produced by the Benczúr-Karger algorithm, the support values $a$ and $b$ of Algorithm 2-sparsifications are still low enough to make the sparsifications useful for solving linear systems quickly.

Figure 2: Plot of $c$-value vs. number of trials for Algorithm 2
### Table 1: Accuracy of Algorithm 2

<table>
<thead>
<tr>
<th>Graph</th>
<th>d</th>
<th>$m'$</th>
<th>a</th>
<th>b</th>
<th>$a_{bk}$</th>
<th>$b_{bk}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Airplane P4</td>
<td>1</td>
<td>13121</td>
<td>3.0812</td>
<td>2.0442</td>
<td>22320</td>
<td>2.2255</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>25787</td>
<td>1.8846</td>
<td>1.5921</td>
<td>32441</td>
<td>1.3399</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>50421</td>
<td>1.4396</td>
<td>1.3327</td>
<td>42066</td>
<td>1.204</td>
</tr>
<tr>
<td>n = 1376</td>
<td>2</td>
<td>22320</td>
<td>2.2255</td>
<td>1.7686</td>
<td></td>
<td></td>
</tr>
<tr>
<td>m = 84192</td>
<td>4</td>
<td>50421</td>
<td>1.4396</td>
<td>1.3327</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Airplane P5</td>
<td>1</td>
<td>13237</td>
<td>3.2181</td>
<td>2.1004</td>
<td>30484</td>
<td>1.5955</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>26557</td>
<td>2.0034</td>
<td>1.6436</td>
<td>42803</td>
<td>1.3105</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>50421</td>
<td>1.4606</td>
<td>1.3866</td>
<td>54140</td>
<td>1.2066</td>
</tr>
<tr>
<td>n = 1376</td>
<td>2</td>
<td>26557</td>
<td>2.0034</td>
<td>1.6436</td>
<td></td>
<td></td>
</tr>
<tr>
<td>m = 123475</td>
<td>4</td>
<td>50421</td>
<td>1.4606</td>
<td>1.3866</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Grid P5</td>
<td>1</td>
<td>8662</td>
<td>4.0364</td>
<td>2.1180</td>
<td>17943</td>
<td>1.2579</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>17586</td>
<td>2.0378</td>
<td>1.5246</td>
<td>23490</td>
<td>1.1406</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>32768</td>
<td>1.4172</td>
<td>1.2484</td>
<td>30947</td>
<td>1.0897</td>
</tr>
<tr>
<td>n = 1000</td>
<td>2</td>
<td>17586</td>
<td>2.0378</td>
<td>1.5246</td>
<td></td>
<td></td>
</tr>
<tr>
<td>m = 44772</td>
<td>4</td>
<td>32768</td>
<td>1.4172</td>
<td>1.2484</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Grid P6</td>
<td>1</td>
<td>8561</td>
<td>3.4462</td>
<td>2.0507</td>
<td>15424</td>
<td>1.5689</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>17022</td>
<td>2.2484</td>
<td>1.6902</td>
<td>23356</td>
<td>1.2496</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>33808</td>
<td>1.4999</td>
<td>1.3077</td>
<td>31654</td>
<td>1.0884</td>
</tr>
<tr>
<td>n = 1000</td>
<td>2</td>
<td>17022</td>
<td>2.2484</td>
<td>1.6902</td>
<td></td>
<td></td>
</tr>
<tr>
<td>m = 63512</td>
<td>4</td>
<td>33808</td>
<td>1.4999</td>
<td>1.3077</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Random $p = 1/\sqrt{n}$</td>
<td>1</td>
<td>6988</td>
<td>4.6963</td>
<td>1.9378</td>
<td>13769</td>
<td>1.5069</td>
</tr>
<tr>
<td>n = 1000</td>
<td>2</td>
<td>13230</td>
<td>1.6418</td>
<td>1.3176</td>
<td>16009</td>
<td>1.0</td>
</tr>
<tr>
<td>m = 15867</td>
<td>4</td>
<td>16009</td>
<td>1.0</td>
<td>1.0</td>
<td>16009</td>
<td>1.0</td>
</tr>
<tr>
<td>Random $p = 1/2$</td>
<td>1</td>
<td>6815</td>
<td>5.2498</td>
<td>2.2190</td>
<td>27181</td>
<td>1.6937</td>
</tr>
<tr>
<td>n = 1000</td>
<td>2</td>
<td>13708</td>
<td>3.4965</td>
<td>1.7874</td>
<td>53782</td>
<td>1.6015</td>
</tr>
<tr>
<td>m = 250188</td>
<td>4</td>
<td>27351</td>
<td>2.1773</td>
<td>1.5921</td>
<td>108119</td>
<td>1.2111</td>
</tr>
</tbody>
</table>

### 4 Changing the way we pick spanning forests

While the experimental results of Algorithm 2 are encouraging, the number of edges produced by the algorithm does not always seem to be a decreasing function of the number of trials (even in the few graphs we tested). We will now explore a different algorithm for peeling off spanning forests that often produces even fewer edges. We will use the same idea of peeling off spanning trees to assign $h_e$ values for each trial, and then we will take the maximum $h_e$ value over all trials to use when we sample each edge.

In algorithm 2, randomness is used only when breaking ties. However, the likelihood of a tie in $r$-values tends to decrease as the algorithm progresses. This limits the variation in forests between different trials. This is partly due to the behavior of the algorithm. The algorithm “grows” spanning forests out of a starting node, and the edges on each height remain connected throughout the process. The method of constructing spanning forests for each height is somewhat similar to Prim’s algorithm.

Instead of “growing” our spanning forests out of a starting location in our graph, we will use an algorithm similar to Kruskal’s minimum spanning tree algorithm to generate each random spanning forest.
4.1 Generating a single spanning forest

To generate a single random spanning forest, we can first obtain a random permutation \( \sigma \) of the edges. We can then add each edge \( \sigma_i \) of the permutation (in order) to our spanning forest, such that the addition of \( \sigma_i \) doesn’t create a cycle in our forest so far. To determine whether or not adding a particular edge will create a cycle, we can use a particular implementation of the disjoint-set data structure used in Kruskal’s algorithm. This data structure keeps track of disjoint sets of objects. It allows two operations: \texttt{union} (which merges two sets of objects into one larger set), and \texttt{find} (which returns which set an object belongs to).

Our objects will be vertices. Each vertex will start out as its own set. Prior to adding any edge to the spanning forest, we will call \texttt{find} on its endpoints. If both endpoints belong to the same set in the disjoint-set data structure, we will ignore the edge. If both endpoints belong to different sets, we will add the edge to the spanning forest, and then call \texttt{union} on the two sets containing the edges’ endpoints. The disjoint-set data structure thus keeps track of the different connected components of our spanning forest during each iteration (each set in the data structure corresponds to a connected component of vertices in our spanning forest). Because both endpoints of an edge belong to the same component iff adding that edge would create a cycle, an edge is added to our spanning forest iff adding that edge does not create a cycle. Thus, the resulting set of edges added is a spanning forest of the original graph.

In our implementation of the disjoint-set data structure, each set will be represented internally as a tree. The root of the tree serves as a representation of the set that contains it. Each vertex will start out containing a pointer to itself (each vertex starts out in its own set). This means that each vertex is the root of its own set. When \texttt{union} is called on two sets, the pointer on the root of one set will be modified to point to the root of the other set (merging both sets under one root). When \texttt{find} is called on a vertex, the root of the set is determined by following the pointers from the vertex to the root of its set. (Note that when \texttt{union} is called on two vertices, \texttt{union} must call \texttt{find} on both vertices to determine the roots of both sets.) The depth of a set is the maximum distance from an element in a set to the root of that set.

Two optimizations are used to make the \texttt{union} and \texttt{find} operations efficient. The first optimization ensures that the depth of any one tree doesn’t get too large after \texttt{union} operations (as this would make \texttt{find} calls inefficient). When \texttt{union} merges two sets with different depths, the root of the set with lower depth will be made to point to the root of the set with higher depth. This is implemented by storing a \texttt{rank} for each root (initially 0). When two sets are merged, the root of the set with lower rank is joined to the root of the set with higher rank. If the ranks of both roots are equal, the rank of one of the two roots (arbitrarily picked) is incremented, and the other root is set to point to this root.

The rank optimization guarantees that all trees are approximately balanced. We can show by induction that a tree with rank \( r \) has at least \( 2^r \) elements. A
tree with rank 0 has at least 0 elements, and a tree with rank \( r \) was created by merging two trees of rank \( r - 1 \) (each with at least \( 2^{r-1} \) elements by induction). Thus, the rank of a tree of \( k \) elements is \( O(\log k) \leq O(\log n) \) ([2]).

A second optimization is used on \texttt{find} calls. Whenever \texttt{find} is called on a node, all intermediate nodes in the path from that node to the root of its set are pointed directly to the root of that set. This improves the running time of later \texttt{find} calls. This optimization is called path compression.

When both of these optimizations are implemented, we get the following result.

**Theorem 3.** The running time of \( k \) \texttt{union/find} operations is \( O(ka(k)) \).

\[ \alpha(k) = \alpha'(k, k) \], where \( \alpha'(n, n) \) is the inverse Ackermann function. This gives us an amortized running time of \( \alpha(k) \) per \texttt{union/find} operation. \( \alpha(n) \) is often considered a constant, as \( \alpha(n) \leq 4 \) for any practical value of \( n \). Theorem 3 is proven in [6].

Constructing a single random spanning forest involves \( 2m \) \texttt{find} calls and at most \( m \) \texttt{union} calls. Therefore, the total running time for generating one random spanning forest is \( O(\text{ma}(m)) \), which is essentially linear in \( m \).

### 4.2 Generating all spanning forests simultaneously

In our application, we need to peel off spanning forests without replacement. This requires that on the \( i \)th level, we peel off a spanning forest of \( G - \cup_{j=1}^{i-1} E_j \). We could apply the above algorithm directly for each level, but the running time would then be \( O(nma(m)) \), which is prohibitive for our purposes.

The first modification we can make to this is to use a single permutation for all levels. Whenever an edge is reached in the permutation whose endpoints are not in the same component on the current level \( (i) \), its endpoints are then unified on level \( i \) and \( h_e \) is set to \( i \). That edge is then removed from the permutation for further consideration on future levels. Whenever an edge is reached in the permutation whose endpoints are in the same component on the current level \( (i) \), that edge is ignored for the rest of the iteration.

This modification is a slight improvement upon our first attempt, as we no longer have to generate multiple permutations. However, the running time of this algorithm is still \( O(nma(m)) \).

We can now modify the algorithm to generate all spanning trees in one pass. In the \( i \)th iteration, whenever we reach an edge \( e = (u, v) \) whose endpoints are in the same component in level \( i \), we currently ignore the edge. We will eventually unify the edges’ endpoints in a later iteration (at a higher level). However, processing the rest of the edges on level \( i \) does not change the level that \( e \) is eventually unified on (as unifications on level \( i \) do not affect the disjoint-set data structure for levels \( i + 1 \) or higher). There is therefore no reason why we need to wait for a later iteration on a later level to process the edge. We can instead determine the first level such that \( u \) and \( v \) do not belong to the same component (call it level \( k \)), and then set \( h_e \) to \( k \) (and unify edge \( e \) on level \( k \)). This allows us to generate all of the height values in one pass. For each edge \( e = (u, v) \) (in
order of the random permutation), the algorithm finds the highest level \( i \) such that \( u \) and \( v \) are not unified. It then unifies \( u \) and \( v \) on level \( i \), and sets \( h_e = i \).

We therefore have three fundamental operations in our multi-level disjoint-set data structure that we use in our algorithm: \( \text{union}(u, v, \text{level}) \), \( \text{find}(u, \text{level}) \), and \( \text{first}(u, v) \). \( \text{First}(u,v) \) finds the first level such that \( u \) and \( v \) are not unified. We first present algorithms for the \( \text{union} \) and \( \text{find} \) operations as Algorithm 3 and Algorithm 4. All pointers for the disjoint-set data structure are stored in a table \( \text{parent}[\text{vertex}][\text{level}] \), indexed by vertex and level. The ranks for the vertices are stored in a table \( \text{rank}[\text{vertex}][\text{level}] \), again indexed by vertex and level.

**Algorithm 3** Find

**Require:** Vertex \( u \), level \( i \)

**Ensure:** The root of the set containing vertex \( u \) is returned

**Ensure:** All intermediate nodes on the path from \( u \) to the root point directly to the root (path compression)

1: if \( \text{parent}[u][i] = u \) then
2: \hspace{1em} return \( u \)
3: else
4: \hspace{1em} \text{parent}[u][i] \leftarrow \text{Find}(\text{parent}[u][i], i)
5: \hspace{1em} return \text{parent}[u][i]
6: end if

**Algorithm 4** Union

**Require:** Vertices \( u, v \), level \( i \)

**Ensure:** \( u \) and \( v \) are unified, with the root of the lower-rank set pointing to the root of the higher-rank set (union-by-rank)

**Ensure:** If the ranks of the roots of both sets are equal, one is picked arbitrarily to be the new root of the larger set, and the rank of this root is incremented

1: \( \text{root}_u \leftarrow \text{Find}(u, i) \)
2: \( \text{root}_v \leftarrow \text{Find}(v, i) \)
3: if \( \text{rank}[\text{root}_u][i] < \text{rank}[\text{root}_v][i] \) then
4: \hspace{1em} \text{parent}[\text{root}_u][i] \leftarrow \text{root}_v
5: else
6: \hspace{1em} \text{parent}[\text{root}_v][i] \leftarrow \text{root}_u
7: \hspace{1em} if \( \text{rank}[\text{root}_u][i] = \text{rank}[\text{root}_v][i] \) then
8: \hspace{2em} \text{rank}[\text{root}_u][i] \leftarrow \text{rank}[\text{root}_u][i] + 1
9: \hspace{end if}
10: \hspace{end if}

We now describe how to implement \( \text{first}(u, v) \). To determine the first level such that \( u \) and \( v \) are not unified, we can compare the roots of the set containing \( u \) and \( v \) for each level, starting at 1. However, the running time for this operation would then be \( O(n) \), again resulting in an \( O(n \alpha(m)) \) algorithm. To speed up this operation, we will use the following lemma:
Lemma 1. Let $i$ be the first level in our algorithm such that the endpoints of $e$ do not belong to the same component. Then the endpoints of $e$ are not unified on any level $l > i$.

Proof. Proof by induction on $n$, the number of edges examined prior to edge $e$. When $n = 0$, no edge has yet been looked at yet, so the endpoints of edge $e$ will not have yet been unified on any level.

We now assume that if the endpoints of the first $k - 1$ edges examined are not unified on some level, then the endpoints will remain in separate components in all subsequent levels. Consider edge $e = (u, v)$, the $k$th edge being examined. Let $i$ be the first level such that $u$ and $v$ are not unified on level $i$. Assume by contradiction that there exists a level $j > i$ such that $u$ and $v$ are not unified on level $i$, but are unified on level $j$. Since $u$ and $v$ belong to the same component on level $j$, there exists a path of edges from $u$ to $v$ that have been unified on level $j$. Because all edges on this path were examined prior to the $k$th edge, we can apply the induction hypothesis and declare that the edges on this path have been unified on all levels $l < j$ as well (since the induction hypothesis says there are no gaps between levels in which any two vertices are unified). But this implies that $u$ and $v$ belong to the same component on level $i < j$, contradicting the fact that $i$ is the first level on which $u$ and $v$ are not unified. Therefore, there is no level $j > i$ such that $u$ and $v$ are unified on level $j$. ☐

We can now present as Algorithm 5 an efficient binary-search algorithm to determine the first level such that $u$ and $v$ are not unified. Note that in this algorithm, we assume all vertices are unified on level 0 (as a definition).

### Algorithm 5

**First level such that two vertices are not unified**

**Require:** Vertices $u, v$

**Ensure:** The level returned is the first level on which vertices $u$ and $v$ are not unified.

1. $a \leftarrow 1$, $b \leftarrow m$, $mid$
2. while $a < b$ do
3. $mid \leftarrow \lfloor (a + b)/2 \rfloor$
4. $root^u_{mid-1} \leftarrow \text{Find}(u, mid - 1)$
5. $root^v_{mid-1} \leftarrow \text{Find}(v, mid - 1)$
6. $root^u_{mid} \leftarrow \text{Find}(u, mid)$
7. $root^v_{mid} \leftarrow \text{Find}(v, mid)$
8. if $root^u_{mid-1} = root^v_{mid-1}$ and $root^u_{mid} \neq root^v_{mid}$ then
9. return $mid$
10. else if $root^u_{mid-1} \neq root^v_{mid-1}$ and $root^u_{mid} \neq root^v_{mid}$ then
11. $b \leftarrow mid$
12. else if $root^u_{mid-1} = root^v_{mid-1}$ and $root^u_{mid} = root^v_{mid}$ then
13. $a \leftarrow mid + 1$
14. end if
15. end while
16. return $mid$
Theorem 4. Algorithm 5 produces the level $i$ such that $u$ and $v$ are unified on levels $l < i$ but not on level $i$.

Proof. Proof by induction on the quantity $b - a$. If $b - a = 0$, then the first level on which $u$ and $v$ are not unified in the range $[a, b]$ must be $a = b$.

Now, assume that for $b - a < k$, $\text{first}(u, v)$ correctly returns the first level such that $u$ and $v$ are not unified (within the range $[a, b]$). Consider the case where $b - a = k$. Let $i$ be the return value of the algorithm. In any iteration, if $u$ and $v$ are unified on level $\text{mid} - 1$ but not on level $\text{mid}$, then $i = \text{mid}$ by Lemma 1. If $u$ and $v$ are not unified on levels $\text{mid} - 1$ or $\text{mid}$, then $i$ must fall in the range $[a, \text{mid}]$ by Lemma 1. If $u$ and $v$ are unified on levels $\text{mid} - 1$ and $\text{mid}$, then $i$ must fall in the range $[\text{mid} + 1, b]$ by Lemma 1. In both of the latter two cases, the range $b - a$ decreases, so we can apply the induction hypothesis and declare that $\text{first}(u, v)$ returns the first level such that $u$ and $v$ are not unified.

Theorem 5. Algorithm 5 runs in the time it takes for $O(\log n)$ find operations.

Proof. $b - a$ is initially $m - 1$, and $b - a$ decreases by half after each iteration. During each iteration, the running time is dominated by the time it takes for the four find operations. Thus, the total running time is the time it takes for $O(\log_2 m) = O(\log n)$ find operations.

We can now present as Algorithm 6, a simple formulation of our main algorithm that determines the height values for each edge.

**Algorithm 6** Randomized algorithm for determining $h_e$ values for each edge

Require: $G = (V, E)$

Ensure: $h_e > 0 \ \forall e \in E$

1: Initialize tables $\text{parent}[\text{vertex}][\text{level}]$ and $\text{rank}[\text{vertex}][\text{level}]$
2: $\sigma \leftarrow$ Random permutation of edges 1 to $m$
3: for $i = 1$ to $m$ do
4: \hspace{1em} $(e = (u, v)) \leftarrow \sigma_i$
5: \hspace{1em} $h_e \leftarrow \text{First}(u, v)$
6: \hspace{1em} $\text{Union}(u, v, h_e)$
7: end for

Figure 2 contains an example run of one trial of Algorithm 6 on the complete graph of 5 vertices.

Theorem 6. All edges with height value $i$ ($E_i$) form a spanning forest on $G - \cup_{j=1}^{i-1} E_j$.

Proof. First, note that $E_i$ is a subset of $G - \cup_{j=1}^{i-1} E_j$ (since any edge $e$ with height $h_e < i$ belongs to $\cup_{j=1}^{i-1} E_j$ by definition of $E_j$). Furthermore, every edge with height $j > i$ was added when its endpoints were unified on level $i$. Therefore, $E_j$ includes all edges of $G - \cup_{j=1}^{i-1} E_j$ that would not create a cycle if added to $E_i$. $E_i$ is therefore a spanning forest on $G - \cup_{j=1}^{i-1} E_j$. 

14
Theorem 7. The total running time of the algorithm is $O(m \log n \alpha(m \log n))$.

Proof. The initialization of the tables $parent$ and $rank$ takes constant time. We can achieve this by not setting any values in advance. If parent[vertex][level] is not set, then parent[vertex][level] = vertex implicitly. Likewise, if rank[vertex][level] is not set, then rank[vertex][level] = 0 implicitly.

Calculating the random permutation takes time $O(m)$.

The rest of the algorithm consists of $m$ first operations and $m$ union operations. The $m$ first operations consist of $O(\log n)$ find operations. Therefore, the total number of union/find operations is $O(m + m \log n) = O(m \log n)$. Let $k$ be the total number of union/find operations, with $k_1$ of them occurring on level 1, $k_2$ of them on level 2, and in general, $k_i$ of them on level $i$.

According to theorem 3, the cost of $k'$ union/find operations on one particular level takes time $O(k'\alpha(k'))$. Therefore, the running time for all union/find operations on all levels is:

$$
\sum_{i=1}^{m} O(k_i \alpha(k_i)) \leq \sum_{i=1}^{m} O(k_i) = \alpha(k) \sum_{i=1}^{m} O(k_i) = \alpha(k) O(k) = O(m \log n \alpha(m \log n))
$$

Note that this running time is $O(m \log n)$ in practice. □
We now have an algorithm that produces height values that can be used to generate sparsifications. We can now run several trials of Algorithm 6, and let $h^*_e$ be the maximum $h_e$ value over all trials for each edge. We can then sample with the $h^*_e$ values according to corollary 1.

4.3 Experimental results

We will first look at the expected number of edges that are produced by an Algorithm 6-sparsification. As for Algorithm 2, we will look at the quantity

$$C = \frac{1}{n} \sum_{e \in E} \frac{1}{h^*_e}$$

This quantity is the constant multiple of $n \ln n$ of edges produced in Algorithm 6-sparsifications. Figure 4 contains results for runs of up to 10 trials of Algorithm 6.

The results indicate a huge drop in the number of edges produced once the number of trials is 2 or larger. We get a better edge-bound than Algorithm 2 for any run of Algorithm 6 with more than 1 trial. For the graphs we tested, $C$ seems to be a decreasing function of the number of trials, which was not true for Algorithm 2. Furthermore, $C$ approaches 1 much more rapidly as the number of trials increases than the $C$ values produced by Algorithm 2. In fact, the resulting
values are almost identical for any given number of trials, regardless of the input graph. These tests we ran do not necessarily indicate a general property of the algorithm on all graphs. However, if similar results hold for all graphs, we can conjecture that for 2 or more trials, \( C \) is a constant (or an extremely-slow-growing function) with respect to the number of vertices of the graph. This is in contrast to our original simplification in section 2 (where \( C = \ln n \)), or in our results for Algorithm 2 (where the results were less conclusive, even for the graphs we tested).

Table 2 contains results for the accuracy of 10-trial Algorithm 6-sparsifications.

<table>
<thead>
<tr>
<th>Graph</th>
<th>d</th>
<th>( m' )</th>
<th>a</th>
<th>b</th>
<th>( m'_{bk} )</th>
<th>a_{bk}</th>
<th>b_{bk}</th>
</tr>
</thead>
<tbody>
<tr>
<td>Airplane P4</td>
<td>1</td>
<td>11079</td>
<td>4.1266</td>
<td>2.1454</td>
<td>22320</td>
<td>2.2255</td>
<td>1.7686</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>21904</td>
<td>2.0692</td>
<td>1.6048</td>
<td>32441</td>
<td>1.3399</td>
<td>1.3725</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>43750</td>
<td>1.6350</td>
<td>1.3356</td>
<td>42066</td>
<td>1.204</td>
<td>1.1402</td>
</tr>
<tr>
<td>Airplane P5</td>
<td>1</td>
<td>21043</td>
<td>3.0643</td>
<td>2.0266</td>
<td>11484</td>
<td>1.5655</td>
<td>1.6536</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>21787</td>
<td>2.4550</td>
<td>1.7369</td>
<td>22803</td>
<td>1.3105</td>
<td>1.2719</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>43736</td>
<td>1.5294</td>
<td>1.4300</td>
<td>54140</td>
<td>1.2066</td>
<td>1.1667</td>
</tr>
<tr>
<td>Grid P5</td>
<td>1</td>
<td>7596</td>
<td>3.0809</td>
<td>2.0376</td>
<td>17943</td>
<td>1.2579</td>
<td>1.2783</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>15374</td>
<td>1.8394</td>
<td>1.6380</td>
<td>23490</td>
<td>1.1406</td>
<td>1.1314</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>30116</td>
<td>1.3682</td>
<td>1.2615</td>
<td>30947</td>
<td>1.0897</td>
<td>1.0635</td>
</tr>
<tr>
<td>Grid P6</td>
<td>1</td>
<td>7596</td>
<td>3.0809</td>
<td>2.0376</td>
<td>17943</td>
<td>1.2579</td>
<td>1.2783</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>15374</td>
<td>1.8394</td>
<td>1.6380</td>
<td>23490</td>
<td>1.1406</td>
<td>1.1314</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>30116</td>
<td>1.3682</td>
<td>1.2615</td>
<td>30947</td>
<td>1.0897</td>
<td>1.0635</td>
</tr>
<tr>
<td>Random p=1/√n</td>
<td>1</td>
<td>7223</td>
<td>3.0206</td>
<td>1.5832</td>
<td>13769</td>
<td>1.5069</td>
<td>1.1581</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>14459</td>
<td>1.3677</td>
<td>1.1262</td>
<td>16009</td>
<td>1.0</td>
<td>1.0</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>15867</td>
<td>1.0</td>
<td>1.0</td>
<td>16009</td>
<td>1.0</td>
<td>1.0</td>
</tr>
</tbody>
</table>

The accuracy results for Algorithm 6 are very similar to the results of Algorithm 2. For \( d = 1 \), the \( a \) and \( b \) values produced by Algorithm 6 are about 2-3.5 times larger than the corresponding values produced by the Benczúr-Karger algorithm, while the numbers of edges in Benczúr-Karger sparsifications are about 2-4 times as many as the number of edges in Algorithm 6-sparsifications. For larger values of \( d \), both the edge-bound advantage of Algorithm 6 and the accuracy advantage of Benczúr-Karger decrease.
5 Conclusion

We have presented two algorithms that seem to improve upon the Benczúr-Karger algorithm for graph sparsification in practice. The Benczúr-Karger algorithm guarantees that cut values are preserved and that the resulting sparsification has $O(n \log n/\epsilon^2)$ edges. Furthermore, it runs in time $O(m \log n)$ when taking into account conjecture 1. However, the high constants in the running time and edge bound limit the usefulness of Benczúr-Karger sparsifications.

Running a constant number of trials of Algorithm 2 takes time $O(m)$ and improves slightly upon the constant factor in the edge bound in our tests (though the results are somewhat unpredictable). Running a constant number of trials of Algorithm 6 takes time $O(m \log n)$, and greatly improves upon the constant factor in the edge bound with the graphs we tried. Furthermore, in our tests, the edge bound for Algorithm 6 was a decreasing function of the number of trials we ran our algorithm. This is a useful property, as it allows one to substitute running time for better sparsifications. Even Algorithm 2 allows one to substitute running time for better sparsifications (though the number of edges produced is not always a strictly decreasing function of the number of trials). The Benczúr-Karger algorithm for computing the sampling probabilities is deterministic, so it does not have this property. While both Algorithm 2 and Algorithm 6 produce sparsifications with somewhat worse accuracy than Benczúr-Karger, the sparsifications we generated were still accurate enough to make them useful in applications such as solving linear systems quickly. Finally, Algorithm 6 is simpler to describe and implement than the Benczúr-Karger algorithm.

We have not been able to prove that any number of trials of Algorithm 2 or Algorithm 6 are guaranteed to result in a sparsification of $O(n \log n/\epsilon^2)$ edges. However, our experimental evidence indicates that it is probably true for the families of graphs we tested, and it may be true for a large number of graphs.

6 Future work

One open question is the actual edge-bound guarantee for Algorithms 2 and 6. While we have a trivial guarantee of $O(n \log^2 n/\epsilon^2)$ edges for unweighted graphs (the result of a single trial), we do not yet have a proof that running multiple trials and taking the maximum height for each edge is guaranteed to improve the edge bound.

There is a process similar to Algorithm 6 that we do know more about. All of our algorithms effectively peel off spanning forests without replacement. The similar process is sampling spanning trees at random with replacement, and setting the height values equal to the number of the spanning tree that the edge first appears in.

If we repeatedly sample single random spanning trees (with replacement) as we did in section 4.1, the expected “height” of an edge $e$ is simply $1/p_e$, where $p_e$ is the probability that edge $e$ appears in any spanning tree from the distribution in our algorithm. We know two useful facts about $p_e$ values:
Theorem 8. \( p_e \geq 1/\lambda_e \) for all \( e \in E \).

Proof. For edge \( e = (u, v) \), consider any minimum \( u - v \) cut of \( \lambda_e \) edges. If edge \( e \) comes before all \( \lambda_e \) edges of this minimum cut in the random permutation in the algorithm, \( u \) and \( v \) will be disconnected when edge \( e \) is examined. This means that edge \( e \) will be added to the tree. The probability of any \( \lambda_e \) edges coming before edge \( e \) in a random permutation is simply \( 1/\lambda_e \). Because this is a sufficient (but not necessary) condition for edge \( e \) including edge \( e \) in this type of random spanning tree, we have \( p_e \geq 1/\lambda_e \). \( \square \)

Theorem 9. \( \sum_{e \in E} p_e = n - 1 \)

Proof. This follows from the fact that there are exactly \( n - 1 \) edges in any spanning tree \( \sum_{e \in E} p_e = E \left[ \sum_{e \in E} X_e \right] = n - 1 \).

These two theorems together imply that we can substitute \( 1/p_e \) into conjecture 1 and get an \( O(n \log n/\epsilon^2) \) sparsifier that preserves cut values (assuming conjecture 1 is true). The cut values would be preserved because \( p_e \ln n/\epsilon^2 \geq \ln n/\epsilon^2 \lambda_e \), and the edge bound would hold because \( \sum_{e \in E} X_e = \Theta \left( \frac{n \ln n}{\epsilon^2} \right) \sum_{e \in E} p_e = \Theta(n \log n/\epsilon^2) \).

However, at this time, we have no efficient way of computing \( p_e \) values directly. Standard Monte-Carlo sampling requires \( \Omega(n \log n) \) trials (each taking time \( O(m) \)) to get sufficient accuracy, since \( p_e \) can be as low as \( O(1/n) \). \(^3\) But even if we can’t compute \( p_e \) directly, the quantity is still useful if we can relate it to the heights calculated by peeling off spanning trees without replacement. We know of no such relation at this time.

Another open problem is extending Algorithm 6 to support weighted-graphs. Algorithm 2 can be easily modified to work on weighted graphs: we can simply use the weighted-graph version of the Nagamochi-Ibaraki algorithm instead of the unweighted version. However, we do not yet know of an efficient weighted-graph analogue for Algorithm 6.

7 References


\(^3\)One way to improve this would to sample for \( \Omega(m/n \log n) \) trials, and set any remaining edges not sampled to \( p_e = n/m \). This works because we are only looking for lower bounds on \( p_e \), and a lower bound of \( n/m \) is sufficient to get an \( O(n \log n/\epsilon^2) \)-edge sparsifier. However, the total running time of all \( \Omega(m/n \log n) \) trials would still be prohibitive for our purposes.

