Spectral Sparsification and Spectrally Thin Trees

by

Rafael Oliveira

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Master of Engineering in Electrical Engineering and Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

September 2012

Signature redacted

Author ....
Department of Electrical Engineering and Computer Science
August 25, 2012

Signature redacted

Certified by ....
Michel Goemans
Leighton Family Professor of Applied Mathematics
Thesis Supervisor

Signature redacted

Accepted by ....
Dennis Freeman
Chairman, Department Committee on Graduate Theses

Spectral Sparsification and Spectrally Thin Trees

by

Rafael Oliveira

Submitted to the Department of Electrical Engineering and Computer Science
on August 25, 2012, in partial fulfillment of the
requirements for the degree of
Master of Engineering in Electrical Engineering and Computer Science

Abstract

We provide results of intensive experimental data in order to investigate the existence of spectrally thin trees and unweighted spectral sparsifiers for graphs with small expansion.

In addition, we also survey and prove some partial results on the existence of spectrally thin trees on dense graphs with high enough expansion.

Thesis Supervisor: Michel Goemans
Title: Leighton Family Professor of Applied Mathematics

1. Introduction

A sparsifier of a graph $G = (V, E)$ is a sparse subgraph $H = (V, E')$ of G that is similar to G according to some set of criteria. The criterion that defines a spectral sparsifier is the following: H is an ϵ-spectral approximation of G, if for all $x \in \mathbb{R}^V$,

$$(1-\epsilon)x^T L_G x \le x^T L_H x \le (1+\epsilon)x^T L_G x \qquad (1)$$

where L_G and L_H are the combinatorial Laplacian matrices of the (weighted) graphs G and H. This notion of sparsification was introduced by Spielman and Teng in [6, 7]. Another notion of sparsification, given by Benczur and Karger in [3], is the notion of cut-sparsifiers, where the weight of each cut in H is approximately the same as the weight of the same cut in G, for every cut of the graph. From both definitions, it is easy to see that the notion of spectral sparsification is strictly stronger than the notion of cut sparsification, since the latter notion is equivalent to satisfying condition (1) only for vectors $x \in \{0,1\}^V$.

A spanning tree $T = (V, F)$ of a graph $G = (V, E)$ is said to be *α-thin*, where $\alpha < 1$, if for every partition $(S, \overline{S})$ of the graph, we have $|\delta_F(S, \overline{S})| \le \alpha|\delta_E(S, \overline{S})|$, where $\delta_E(S, \overline{S})$ is the set of edges in the cut $(S, \overline{S})$ in G (we define $\delta_F(S, \overline{S})$ similarly). Thin trees are important combinatorial objects, and their existence for highly edge-connected graphs imply important results in combinatorics and in approximation algorithms. For instance, Goemans et al. [1] showed that if there is always an α-thin tree for k-edge-connected graphs, then the integrality gap for the Held-Karp relaxation for the Asymmetric TSP is bounded above by a constant, which would close the integrality gap for this problem. We can also formulate the definition of thin trees in the spectral sense. We say that a tree $T = (V, F)$ is spectrally α-thin with respect to $G = (V, E)$ if for all $x \in \mathbb{R}^V$, we have $x^T L_T x \le \alpha x^T L_G x$.

While it is again clear that the definition of spectral thinness is stronger than the cut definition, it is much easier to check for spectral thinness than to test whether a spanning tree is cut thin, for the latter is NP-hard, whereas the former can be easily done in polynomial time, by checking whether $\alpha L_G - L_T$ is positive semidefinite.

Therefore, it is natural to try to characterize which graphs have spectral thin trees and to construct these trees for these kinds of graphs. In 2009, Batson, Spielman and Srivastava provided a deterministic algorithm in [2] which, given an input graph $G = (V, E)$, produces a weighted spectral sparsifier $H = (V, F)$ where $|F| = O(|V|)$. With their algorithm, Goemans was able to show that any graph $G = (V, E)$ has a $\frac{4|V|}{|E|}$-spectrally thin subgraph $H = (V, F)$ with $|F| = n$ (not necessarily connected). It is also known that there exist k-edge-connected graphs (even for non constant $k = \sqrt{|V|}$) that have cut thin trees but which have no spectrally thin tree. Nevertheless, the categorization of which kinds of graphs have spectrally thin spanning trees is still an open problem, and the construction of spectrally thin trees for graphs where they exist is also an open and important problem. We consider the existence of spectrally thin trees to be an easier problem than to check the existence of spectral sparsifiers in which the weights of all the edges selected are all the same.

In this thesis, we present the results of a computationally intensive experimental investigation based on simulations done for random graphs with a number of vertices ranging from 200 to 600, and varying edge probability. More details will be presented in section 4.

2 Background

From this section on, we will assume that all graphs are connected, undirected, with no self loops and have n vertices, unless stated otherwise. We also assume that all matrices are square matrices and have dimension n, unless otherwise noted.

In this section we will define some technical terms and state some properties that will be used thoroughly in this thesis, such as the properties of the Laplacian matrix of a graph, describe the algorithm by Batson, Spielman and Srivastava (BSS), and define a special class of graphs called expander graphs, which will be a central object in our investigations.

2.1 Definitions and Properties of the Laplacian

Definition 1. The Laplacian matrix L_G of a weighted graph $G = (V, E, w)$ is defined as follows:

$$[L_G]_{ij} = \begin{cases} \sum\limits_{k\in\Gamma(i)} w_{ik}, & \text{if } i = j \\ -w_{ij}, & \text{if } (i,j) \in E(G) \\ 0, & \text{otherwise} \end{cases}$$

where $\Gamma(i)$ is the set of neighbors of vertex i in G and $w : E \to \mathbb{R}^+$.

With this definition, it is easy to see that $L_G = \sum\limits_{(i,j)\in E(G)} w_{ij}L_{ij}$, where L_{ij} is the Laplacian of the restriction of G to the graph on $V(G)$ having only the edge (i,j) with weight w_{ij}. With this observation, we can now show how to obtain cuts from the Laplacian matrix of G.

Proposition 1. If $G = (V, E, w)$ is a graph and $S \subset V(G)$, then we have that

$$|\delta_E(S, \overline{S})| = \chi_S^T L_G \chi_S$$

where $\chi_S \in \mathbb{R}^V$ is the characteristic vector of the set S.

Proof. From the observation above, we know that $\chi_S^T L_G \chi_S = \sum\limits_{(i,j)\in E(G)} w_{ij}\chi_S^T L_{ij}\chi_S = \sum\limits_{(i,j)\in E(G)} w_{ij}(\chi_i - \chi_j)^2$, where χ_i is the i^{th} coordinate of χ_S. Since $\chi_i \in \{0, 1\}$, for all $i \in V$, and since $\chi_i = 1$ if and only if $i \in S$, we have that $(\chi_i - \chi_j)^2 = 1$ if and only if $\chi_i \neq \chi_j$, which is equivalent to saying that exactly one element of $\{i, j\}$ is an element of S, which is equivalent to (i, j) being an edge across the cut $(S, \overline{S})$. Therefore the sum above is equal to $\sum\limits_{(i,j)\in\delta_E(S,\overline{S})} w_{ij} = |\delta_{E,w}(S, \overline{S})|$, as we wanted. $\square$

The proposition above formalizes the fact that the spectral condition defined in the previous section is stronger than the cut condition. Now, let's talk about positive semidefiniteness and the spectrum of a Laplacian matrix.

Definition 2. A real symmetric matrix A is positive semidefinite if:

1. the quadratic form $\mathbf{v^T}A\mathbf{v}$ is greater than or equal to 0 for all $\mathbf{v} \in \mathbb{R}^n$, or,

2. the eigenvalues of A are nonnegative real numbers, or,

3. A is a nonnegative linear combination of matrices of the type $\mathbf{vv^T}$,

the three conditions above being equivalent to one another.

From now on, we also make the following definition: we say that $A \preceq B$ if and only if $B - A$ is positive semidefinite.

Proposition 2. Laplacian matrices are positive semidefinite.

Proof. Since $x^T L_G x = \sum_{(i,j)\in E(G)} w_{ij} x^T L_{ij} x = \sum_{(i,j)\in E(G)} w_{ij}(x_i - x_j)^2$ and $w_{ij} \geq 0$ for all $(i,j) \in E$, we have that the last quantity is always greater than or equal to zero, which implies positive semidefiniteness, by the definition above. □

Hence, if L_G is the Laplacian of a graph G, we know that L_G is real symetric positive semidefinite and we obtain the following property:

Proposition 3. We can write L in the following form: $L = \sum_{i=2}^{n} \lambda_i \mathbf{v_i v_i^T}$, where the λ_i's are the nonzero eigenvalues of L and the $\mathbf{v_i}$'s are its corresponding eigenvectors. Moreover, the $\mathbf{v_i}$'s form an orthonormal basis of vectors in the space orthogonal to $\mathbf{v_1}$, where $\mathbf{v_1}$ is the eigenvector corresponding to $\lambda_1 = 0$ (the all 1's vector).

If the graph G is connected, as we are assuming throughout this thesis, we also know that all λ_i's are strictly greater than zero, for $i \geq 2$. Now we can define the pseudoinverse of a Laplacian.

Definition 3. For a Laplacian $L = \sum_{i=2}^{n} \lambda_i \mathbf{v_i v_i^T}$, the matrix $L^\dagger = \sum_{i=2}^{n} \frac{1}{\lambda_i} \mathbf{v_i v_i^T}$ is the Moore-Penrose pseudoinverse of L.

From the definition above, it is easy to see that $LL^\dagger = L^\dagger L = \sum_{i=2}^{n} \mathbf{v_i v_i^T}$. Based on the fact that $\lambda_i > 0$ for all $i \geq 2$, we can define the square roots of both matrices L and $L^\dagger$.

Definition 4. The square root of $L_G = \sum_{i=2}^{n} \lambda_i \mathbf{v_i} \mathbf{v_i}^T$ is defined as:

$$(L_G)^{1/2} = \sum_{i=2}^{n} \sqrt{\lambda_i} \cdot \mathbf{v_i} \mathbf{v_i}^T$$

the definition is analogous for the pseudoinverse $L_G^{\dagger}$.

With these definitions, we can now describe and analyze the BSS algorithm.

2.2 BSS Algorithm and Analysis

In [2], Batson, Spielman and Srivastava give a deterministic method to find weighted sparsifiers $H = (V, E, \tilde{w})$ with $O(|V|)$ edges of any weighted graph $G = (V, E, w)$. Their algorithm is as follows:

Algorithm 1: Batson, Spielman and Srivastava's algorithm

```
Input: Vectors $\mathbf{v_1}, \mathbf{v_2}, \ldots, \mathbf{v_m} \in \mathbb{R}^n$ such that $\sum_k \mathbf{v_k}\mathbf{v_k}^T = I_n$ and $d > 1$.
Output: Vector $s \in \mathbb{R}_+^m$ with $\leq \lfloor d(n-1) \rfloor$ positive entries, such that
    $I_n \preceq \sum_{i=1}^{m} s_i \mathbf{v_i}\mathbf{v_i}^T \preceq \left( \frac{d+1+2\sqrt{d}}{d+1-2\sqrt{d}} \right) I_n.$
Initialize $A_0 = 0$. Set parameters $u_0$, $l_0$, $\delta_L$ and $\delta_U$ as in [2] and $T = \lfloor d(n-1) \rfloor$.
Define potential functions $\Phi^u(A) = Tr(uI_n - A)^{-1}$ and $\Phi_l(A) = Tr(A - lI_n)^{-1}$;
for $t = 1, 2, \ldots, T$ do
    Find a vector $\mathbf{v_e}$ and a weight $s_e \geq 0$ such that:
    $\lambda_{max}(A_{t-1} + s_e\mathbf{v_e}\mathbf{v_e}^T) < u_t$ and $\lambda_{min}(A_{t-1} + s_e\mathbf{v_e}\mathbf{v_e}^T) > l_t$,
    $\Phi^{u_t}(A_{t-1} + s_e\mathbf{v_e}\mathbf{v_e}^T) \leq \Phi^{u_{t-1}}(A_{t-1})$ and $\Phi_{l_t}(A_{t-1} + s_e\mathbf{v_e}\mathbf{v_e}^T) \leq \Phi_{l_{t-1}}(A_{t-1})$;
    $A_t \leftarrow A_{t-1} + s_e\mathbf{v_e}\mathbf{v_e}^T$.
    $u_t \leftarrow u_{t-1} + \delta_U$;
    $l_t \leftarrow l_{t-1} + \delta_L$;
end
return $A = \frac{1}{l_T} A_T$
```

In the preprocessing stage, they compute the edge vectors given by $\mathbf{v_{ij}} = (L_G^\dagger)^{1/2}(e_i - e_j)$, for all $(i,j) \in E(G)$. Note that with these vectors, we have

$$\sum_{(i,j)\in E(G)} \mathbf{v_{ij}}\mathbf{v_{ij}^T} = \sum_{(i,j)\in E(G)} (L_G^\dagger)^{1/2}(e_i-e_j)(e_i-e_j)^T(L_G^\dagger)^{1/2} = \sum_{(i,j)\in E(G)} (L_G^\dagger)^{1/2}L_{ij}(L_G^\dagger)^{1/2} =$$

$$= (L_G^\dagger)^{1/2}\left(\sum_{(i,j)\in E(G)} L_{ij}\right)(L_G^\dagger)^{1/2} = (L_G^\dagger)^{1/2}L_G(L_G^\dagger)^{1/2} = I_{Im(L_G)}.$$

After precomputing these vectors, they pass these precomputed vectors to algorithm 1 above, along with a constant $d > 1$.

They can pass the precomputed vectors as the input to the algorithm because the vector space given by $Im(L_G)$ is isomorphic to $\mathbb{R}^{|V|-1}$. They initially start with the zero matrix A_0, which will be updated at each step by the chosen vector multiplied by its proper weight $s_e\mathbf{v_e}\mathbf{v_e^T}$ as the algorithm goes on.

To show that they can always choose a vector $\mathbf{v_e}$ and a weight s_e at each iteration, they prove the following lemmas and combine them, as will be explained:

Lemma 1. (Upper barrier shift) Suppose $\lambda_{max}(A) < u$ and $\mathbf{v}$ is any vector. If

$$\frac{1}{s} \geq \frac{\mathbf{v^T}((u+\delta_U)I_n - A)^{-2}\mathbf{v}}{\Phi^u(A) - \Phi^{u+\delta_U}(A)} + \mathbf{v^T}((u+\delta_U)I_n - A)^{-1}\mathbf{v} = U_A(\mathbf{v})$$

then $\lambda_{max}(A + s\mathbf{v}\mathbf{v^T}) < u + \delta_U$ and $\Phi^{u+\delta_U}(A + s\mathbf{v}\mathbf{v^T}) \leq \Phi^u(A)$.

Lemma 2. (Lower barrier shift) Suppose $\lambda_{min}(A) > l$, $\Phi_l(A) \leq 1/\delta_L$ and $\mathbf{v}$ is any vector. If

$$0 < \frac{1}{s} \leq \frac{\mathbf{v^T}(A - (l+\delta_L)I_n)^{-2}\mathbf{v}}{\Phi_{l+\delta_L}(A) - \Phi_l(A)} - \mathbf{v^T}(A - (l+\delta_L)I_n)^{-1}\mathbf{v} = L_A(\mathbf{v})$$

then $\lambda_{min}(A + s\mathbf{v}\mathbf{v^T}) > l + \delta_L$ and $\Phi_{l+\delta_L}(A + s\mathbf{v}\mathbf{v^T}) \leq \Phi_l(A)$.

Based on the lemmas above, they need to show that there is a vector $\mathbf{v_e}$ for which $0 < U_A(\mathbf{v_e}) \leq L_A(\mathbf{v_e})$, because only with this condition satisfied will they be able to choose a weight s_e for which $0 < U_A(\mathbf{v_e}) \leq 1/s_e \leq L_A(\mathbf{v_e})$ is satisfied, which would imply that they can shift both barriers and make progress in the algorithm.

The way they prove that there is always an edge e for which $0 < U_A(\mathbf{v_e}) \leq L_A(\mathbf{v_e})$ is by showing that, under a proper setting of the initial parameters u_0, l_0, δ_U and δ_L, the following inequality holds (lemmas 3.5 and 3.6 in [2]):

$$\sum_{e \in E(G)} U_A(\mathbf{v_e}) \leq \frac{1}{\delta_U} + \frac{n}{u_0} \leq \frac{1}{\delta_L} + \frac{n}{l_0} \leq \sum_{e \in E(G)} L_A(\mathbf{v_e}).$$

In the end of the for loop, they will have that $l_T I_n \preceq A_T \preceq u_T I_n$. Hence, by dividing this inequality by l_T (which will be positive at the end of the algorithm), they obtain $I_n \preceq \frac{1}{l_T} A_T \preceq \frac{u_T}{l_T} I_n$. Since $\frac{u_T}{l_T} \leq \frac{d+1+2\sqrt{d}}{d+1-2\sqrt{d}}$, by their choice of the initial parameters, they obtain the desired output. From the output matrix $\frac{1}{l_T} A_T$, they can obtain the graph sparsifier, since by multiplying the inequality

$$I_n \preceq \frac{1}{l_T} A_T = \frac{1}{l_T} \cdot \sum_{e \in E(G)} s_e \mathbf{v_e} \mathbf{v_e^T} \preceq \frac{u_T}{l_T} I_n$$

by $(L_G)^{1/2}$ on the left and on the right, they obtain that $L_G \preceq L_H(\tilde{w}) \preceq \frac{u_T}{l_T} I_n$, where the weights are defined as $\tilde{w_e} = \frac{s_e}{l_T}$, which implies that $\tilde{w_e} > 0$ for at most $\lfloor d(n-1) \rfloor = O(|V|)$ edges $e \in E(G)$. This finishes the analysis.

2.3. Expander Graphs

In this section we define a special class of graphs called expander graphs. Expander graphs are very important objects in Computer Science and Mathematics, having important applications in many areas of both fields. For a more complete view on expander graph properties, see [5].

Before we define what an expander is, we need to define the (edge) expansion of a graph.

Definition 5. The (edge) **Expansion Ratio** of G, denoted $h(G)$, is defined as:

$$h(G) = \min_{\{S \;||S| \leq n/2\}} \frac{|\delta_E(S, \overline{S})|}{|S|}.$$

With this definition of expansion, we can now define families of expander graphs.

Definition 6. A sequence of graphs $\{G_i\}_{i\in\mathbb{N}}$ of size increasing with i is a *family of expander graphs* if there exists $\epsilon > 0$ such that $h(G_i) \geq \epsilon$ for all i.

An easy (and uninteresting) example of a family of expander graphs is the family of complete graphs. It is easy to see that a complete graph with n vertices is an $\frac{n}{2}$-expander (and therefore it is easy to obtain the lower bound ϵ for this family). This example is not very interesting because every vertex in each graph of the family has a very large degree. More interesting examples of expander graphs are ones where the maximum degree of the graph is bounded. Some interesting examples of expander graphs are given in [5], page 453.

It turns out that one can derive bounds on the expansion of a graph G based on the maximum degree of the graph and also on the second smallest eigenvalue of L_G (λ_2). The precise statement of this fact is given in the following theorem:

Theorem 1. If $G = (V, E)$ is a graph where d_{max} is the maximum degree in G and $0 = \lambda_1 \leq \lambda_2 \leq \ldots \leq \lambda_n$, then

$$\frac{\lambda_2}{2} \leq h(G) \leq \sqrt{2d_{max}\lambda_2}.$$

Therefore, to get a lower bound on the expansion of a graph, it is enough to evaluate its second smallest eigenvalue.

3. Implementations of BSS's Algorithm, Simulations and Analysis of Results

In this section, we present our implementations of the adaptations of the BSS algorithm, and the results of our simulations for random graphs with large expansion. Our simulations were done mainly on random graphs $G(n,p)$ with $200 \leq n \leq 600$ (n being the number of vertices of G), based on the Erdős–Rényi model of random graphs, where each edge is independently included in the graph with probability p.

The values of the probability p that we chose for our simulations lie between 1/10 and 1/20. From now on, we denote by $G(n,p)$ a graph on n vertices that was constructed by using the Erdös–Rényi model with probability p. The code used for the experiments, with proper comments and details of implementation, is available at *web.mit.edu/rmendes/www/masters.*

3.1 Simulations

The simulations investigate four major questions. The first question is concerned with the range of values for the upper bounds ($L_A(\mathbf{v_e})$) and the lower bounds ($U_A(\mathbf{v_e})$) at every step, for all edges of the graph. Since Batson, Spielman and Srivastava proved that they can always take another step in their algorithm by proving that $\sum_{e\in E(G)} U_A(\mathbf{v_e}) \leq \sum_{e\in E(G)} L_A(\mathbf{v_e})$, it is natural to ask whether we have $U_A(\mathbf{v_e}) \leq L_A(\mathbf{v_e})$ for many edges in the graph, or in which cases this inequality fails for many edges. In addition, it is also natural to ask if many of the intervals $[U_A(\mathbf{v_e}), L_A(\mathbf{v_e})]$ have a common nonempty intersection at each iteration of the algorithm.

Our second question is related to their choice of potential function. A standard potential/barrier function that has been widely used and studied in the literature is $-\log(\det(uI-A))$, but in their paper, they chose to use $Tr(uI-A)^{-1}$ instead. Can we find a set of examples in which the latter works much better than the former? What empirical differences can we notice between these functions?

The third question that we investigated is related to the existence of spectrally thin trees. Goemans showed that graphs G having a spanning tree of maximum degree 3 (which is the case for d-regular d-edge-connected graphs) have a spectrally α-thin tree, with $\alpha = \dfrac{3+2\sqrt{2}}{\lambda_2}$ [4]. Hence if $\lambda_2(G) > 3+2\sqrt{2}$, then such graphs have spectrally thin trees. Therefore, it is natural to ask whether one can obtain counterexamples with high probability for graphs with $\lambda_2 < 3+2\sqrt{2}$, or if one can obtain spectrally thin trees with high probability for some graphs with $c < \lambda_2 < 3+2\sqrt{2}$, for some constant $c > 0$. We investigate this fact in our simulations. Another fact that we investigate is how to construct such trees. We devised two distinct algorithms to try to construct a thin tree from an input graph, and we want to see how often each of them succeeds in constructing a spectrally thin tree for a given input graph. The details of each algorithm will be explained in the next section.

The fourth question that we investigated concerns the existence of unweighted spectral sparsifiers in highly connected graphs. In section 5 of [2], Batson, Spielman and Srivastava suggest that if one could prove the existence of a constant $\kappa > 0$ (dependent on the input) such that at every main step of algorithm 1 there would exist an edge $e \in E(G)$ for which $U_A(\mathbf{v_e}) \leq \kappa \leq L_A(\mathbf{v_e})$, then we could choose the weight $s_e = \kappa$ at each main step of algorithm 1, adding e to our sparsifier H, and in the end we would obtain the following inequalities:

$$l_T I_n \preceq A_T = \sum_{e \in E(G)} s_e \mathbf{v_e} \mathbf{v_e^T} = \sum_{e \in E(H)} \kappa \mathbf{v_e} \mathbf{v_e^T} \preceq u_T I_n \Rightarrow$$

$$\Rightarrow \frac{l_T}{\kappa} I_n \preceq \sum_{e \in E(H)} \mathbf{v_e} \mathbf{v_e^T} \preceq \frac{u_T}{\kappa} I_n \Rightarrow \frac{l_T}{\kappa} L_G \preceq L_H \preceq \frac{u_T}{\kappa} L_G$$

Hence, if $\kappa > u_T$, we will obtain an unweighted spectral sparsifier for G. In addition, we would not allow edge repetitions in this new algorithm, since we want to obtain an unweighted graph in the end (and allowing edge repetition would add multiplicities to edges of the graph, which we do not want).

3.2 Implementations

Due to the slow runtime of BSS's algorithm ($O(d|V|^3|E|)$), and the amount of matrix multiplications involved in it, we had to precompute matrices whenever possible, in order to significantly decrease the constant factor of the runtime. In addition, we could not afford to precompute and store all the edge vectors $\mathbf{v_e}$, because whenever we get a dense graph and $200 \leq |V| \leq 600$, these edges would require too much space to be stored and therefore MATLAB would run extremely slowly or not run at all. Therefore, we precomputed and stored $(L_G^\dagger)^{\frac{1}{2}}$ in the beginning of our algorithms, since this matrix is used to obtain the vectors $v_{i,j}$, for they are equal to $(L_G^\dagger)^{\frac{1}{2}}(e_i - e_j)$.

Algorithm 2: First Implementation For First Question

```
Input: $L_G$, $d > 1$ 'filename'
Output: A file named 'filename', containing the details of the ranges of
        $U_A(\mathbf{v_e})$ and $L_A(\mathbf{v_e})$ at each main step of BSS for the input graph $G$.
Precompute edge list $E$ and $(L_G^\dagger)^{1/2}$. Initialize an array $C$ of length equals to
the total numbers of edges of $G$ and having all entries equal to zero. Set
parameters $u_0$, $l_0$, $\delta_L$ and $\delta_U$ as in [2], $T = \lfloor d(n-1) \rfloor$ and $A_0 = 0$. Define
potential functions $\Phi^u(A) = Tr(uI_n - A)^{-1}$ and $\Phi_l(A) = Tr(A - lI_n)^{-1}$;
for $t = 1, 2, \ldots, T$ do
    $R \leftarrow [\,]$; $X \leftarrow [\,]$;
    for each edge $e \in E$ do
        if $U_A(\mathbf{v_e}) \leq L_A(\mathbf{v_e})$ then
            $C[e] += 1$;
            add tuple $(e, [U_A(\mathbf{v_e}), L_A(\mathbf{v_e})])$ to array $R$;
            add tuples $(U_A(\mathbf{v_e}), 0)$ and $(L_A(\mathbf{v_e}), 1)$ to array $X$.
        end
    end
    $U \leftarrow \frac{1}{|E(G)|} \sum_{e \in E} U_A(\mathbf{v_e})$;
    $L \leftarrow \frac{1}{|E(G)|} \sum_{e \in E} L_A(\mathbf{v_e})$;
    Get the fraction of pairs of intervals that intersect one another from array
    $X$ (details and explanation in next page).
    Through a linear scan over array $R$, check how many elements of $R$
    intersect interval $[U, L]$.
    Save all the information in 'filename'.
    Get a random element $(e, [U_A(\mathbf{v_e}), L_A(\mathbf{v_e})]) \in R$ and set
    $\frac{1}{s_e} = \frac{U_A(\mathbf{v_e}) + L_A(\mathbf{v_e})}{2}$;
    $A_t \leftarrow A_{t-1} + s_e \mathbf{v_e} \mathbf{v_e^T}$;
    $u_t \leftarrow u_{t-1} + \delta_U$; $l_t \leftarrow l_{t-1} + \delta_L$;
end
return $A = \frac{1}{l_T} A_T$ and write $C$ to 'filename'.
```

In algorithm 2, array C is created to keep track of how many times an edge e has $U_A(\mathbf{v_e}) \leq L_A(\mathbf{v_e})$ throughout the run of the algorithm – the number of times being given by $C[e]$. Array R is the array that keeps track of which edges at step t satisfy the inequality $U_A(\mathbf{v_e}) \leq L_A(\mathbf{v_e})$, and therefore can be chosen in order to choose a weight s_e that allows us to progress with the algorithm. Array X is created to obtain the number of pairs of intervals that intersect at each step t of the algorithm. The way we count the number of intersecting pairs of intervals is the following: sort the elements of X, which are tuples of the form $(U_A(\mathbf{v_e}), 0)$ or $(L_A(\mathbf{v_e}), 1)$, by their first coordinate – breaking ties with the second coordinate, so that when we break ties, we make sure that all ending times appear after the tied starting times. After performing this sorting, which takes $O(|E| \log |E|)$, we set up two counters: a counter c_1, which keeps track of the number of open intervals that we have so far (initially $c_1 = 0$) and a final counter int (initially $int = 0$), which will return the total number of intersecting intervals. We perform a linear scan over the entries of X, increasing c_1 by 1 if we find an entry of type $(U_A(\mathbf{v_e}), 0)$ – because this means that we hit the beginning of an interval – or (if we find an entry of type $(L_A(\mathbf{v_e}), 1)$) decreasing c_1 by 1 and then setting $int+ = c_1$, since the latter condition means that we hit the end of an interval, and therefore this interval that just ended must have intersected the $c_1 - 1$ intervals that are still open (that is why we need to decrement c_1 before adding it to int). This procedure counts the number of intersecting intervals in $O(|E| \log |E|)$ time, which is the best we can hope to achieve, since we need to sort the intervals' endpoints, and that takes $O(|E| \log |E|)$ time. (A good geometric description of this algorithm is the following: we put the intervals in the real line, and then scan the line counting the number of intersections).

Hence, for each step t, we loop through each edge e of the graph, and if $U_A(\mathbf{v_e}) \leq L_A(\mathbf{v_e})$ then we increase the counter $C[e]$, add the interval $[U_A(\mathbf{v_e}), L_A(\mathbf{v_e})]$ to R, since we can choose a good weight s_e from this interval, and update X as above. After counting the number of intersecting intervals (both with R and X), we get a random interval $[U_A(\mathbf{v_e}), L_A(\mathbf{v_e})]$ from R (since R contains only intervals $[U_A(\mathbf{v_e}), L_A(\mathbf{v_e})]$ that have $U_A(\mathbf{v_e}) \leq L_A(\mathbf{v_e})$) and update the matrix A_t, so that we can proceed.

Algorithm 3: Second Implementation For First Question

```
Input: $L_G$, $d > 1$ 'filename'
Output: A file named 'filename', containing the outcome of the algorithm
Precompute edge list $E$ and $(L_G^\dagger)^{1/2}$. Initialize $A_0 = 0$. Set parameters $u_0$, $l_0$,
$\delta_L$ and $\delta_U$ as in [2] and $T = \lfloor d(n-1) \rfloor$. Define potential functions
$\Phi^u(A) = Tr(uI_n - A)^{-1}$ and $\Phi_l(A) = Tr(A - lI_n)^{-1}$ ;
for $t = 1, 2, \ldots, T$ do
    for $i = 1, 2, \ldots, 30$ (that is, we will try a constant number of times) do
        Choose a random edge $e$ from edge list $E$
        if $U_A(\mathbf{v_e}) \leq L_A(\mathbf{v_e})$ then
            $\frac{1}{s_e} = \frac{U_A(\mathbf{v_e}) + L_A(\mathbf{v_e})}{2}$
            $A_t \leftarrow A_{t-1} + s_e \mathbf{v_e} \mathbf{v_e}^{\mathbf{T}}$
            break;
        end
        if $i = 30$ (this means we have found no edges in our attempt) then
            Write to 'filename' that the algorithm did not succeed;
            return fail
        end
    end
    $u_t \leftarrow u_{t-1} + \delta_U$;
    $l_t \leftarrow l_{t-1} + \delta_L$;
end
return $A = \frac{1}{l_T} A_T$ and write the eigenvalue gap of matrix $A$ on 'filename'
```

We devised algorithm 3 in order to address the following part of the first question: "at each step t of the algorithm, do we have $U_A(\mathbf{v_e}) \leq L_A(\mathbf{v_e})$ for many edges e in the graph?" If the answer to this question is yes, then we should expect algorithm 3 to return a correct answer at almost all times, and also to exhibit a runtime on the order of $\Theta(d|V|^3)$, since now we are only attempting to find an edge e (for which $U_A(\mathbf{v_e}) \leq L_A(\mathbf{v_e})$) in a constant number of random trials at every step t. If we fail in these constant number of attempts, then the algorithm will return that it failed, otherwise it will return to us a valid matrix, together with the eigenvalue gap of this matrix, so that we can compare this eigenvalue gap to the desired upper bond on the eigenvalue gap, which is $\frac{d+1+2\sqrt{d}}{d+1-2\sqrt{d}}$. Algorithm 3 differs from algorithm 2, since the former does not look at all the edges of the graph to try to find an edge which allows the algorithm to succeed, whereas that is the main point of algorithm 2.

Algorithms 2, 3 and 4 were devised in order to investigate the first two questions of the previous section, which are: to investigate the range of values of the upper bounds ($L_A(\mathbf{v_e})$) and the lower bounds ($U_A(\mathbf{v_e})$) given by the BSS algorithm, and also to inquire whether changing the potential functions to $-\log(\det(uI - A))$ and to $-\log(\det(A - lI))$ will have a different effect on the outcomes.

In these implementations, we precomputed the edge list, since we will need to look at elements of this list at every main step of the algorithm, when we want to add a new edge to the subgraph. Because we want to sample an edge from the graph, an edge list is much more efficient than trying to sample an edge from the Laplacian of the graph.

Because we have to look at all the edges at each main step of algorithm 2 and we had to sort list X, we are bound to the runtime of $\Theta(d|V|^3|E|\log|E|)$. Therefore, this is the best one can hope to achieve. However, we cannot afford the slow runtime of the BSS algorithm ($\Theta(d|V|^3|E|)$) for algorithms 3 and 4 – since we want to run them multiple times to produce relevant statistical data. Hence, at each main step, instead of looping through all the edges trying to find one edge which allows us to continue with the algorithm, we proceed as follows: we choose a random edge e of the edge list, check if we can choose a proper weight s_e which allows us to make progress, and in case this edge succeeds we proceed, else we try again. At each main step, we attempt to make progress only a constant number of times (30 in the algorithms above). If there are indeed many possible choices for an edge, then we should almost always be able to find an edge in a constant number of trials, which would reduce the running time of the algorithm to $O(d|V|^3)$ in practice and our algorithm would return a good sparsifier in almost all the times we run the algorithms. Moreover, we would be able to produce different spectral sparsifiers of G, in case we decide to run multiple experiments with the same graph.

Algorithm 4, which was designed to investigate the second question – concerning the choice of potential function – is very similar to algorithm 3, except that now the potential functions are given by $\Phi^u(A) = -\log(\det(uI - A))$ and $\Phi_l(A) = -\log(\det(A - lI))$. This also causes the formulas for the upper and lower barrier shifts (U_A and L_A in BSS) to change. Therefore the difference between algorithms 3 and 4 lies only in the initialization of the algorithm, where we choose our potential functions and hence define what the upper and lower barrier shifts (U_A and L_A for algorithm 3 and U'_A and L'_A for algorithm 3).

Algorithm 4: Implementation For Second Question

```
Input: $L_G$, $d > 1$ 'filename'
Output: A file named 'filename', containing the outcome of the algorithm
Precompute edge list $E$ and $(L_G^{\dagger})^{1/2}$. Initialize $A_0 = 0$. Set parameters $u_0$, $l_0$,
$\delta_L$ and $\delta_U$ as in [2] and $T = \lfloor d(n-1) \rfloor$. Define potential functions
$\Phi^u(A) = -\log(\det(uI - A))$, $\Phi_l(A) = -\log(\det(A - lI))$ and define

$$U'_A(\mathbf{v_e}) = \frac{\det((u+\delta_U)I - A)}{\det((u+\delta_U)I - A) - \det(uI - A)} \cdot \mathbf{v}^{\mathbf{T}}((u+\delta_U)I_n - A)^{-1}\mathbf{v},$$

$$L'_A(\mathbf{v_e}) = \frac{\det(A - (l+\delta_L)I)}{\det(A - lI) - \det(A - (l+\delta_L)I)} \cdot \mathbf{v}^{\mathbf{T}}(A - (l+\delta_L)I_n)^{-1}\mathbf{v}.$$

for $t = 1, 2, \ldots, T$ do
    for $i = 1, 2, \ldots, 30$ (that is, we will try a constant number of times) do
        Choose a random edge $e$ from edge list $E$
        if $U'_A(\mathbf{v_e}) \le L'_A(\mathbf{v_e})$ then
            $\frac{1}{s_e} = \frac{U'_A(\mathbf{v_e}) + L'_A(\mathbf{v_e})}{2}$;
            $A_t \leftarrow A_{t-1} + s_e \mathbf{v_e} \mathbf{v_e^T}$;
            break;
        end
        if $i = 30$ (this means we have found no edges in our attempt) then
            Write to 'filename' that the algorithm did not succeed;
            return fail
        end
    end
    $u_t \leftarrow u_{t-1} + \delta_U$;
    $l_t \leftarrow l_{t-1} + \delta_L$;
end
return $A = \frac{1}{l_T} A_T$ and write the eigenvalue gap of matrix $A$ on 'filename'
```

To investigate the third question, we devised two distinct algorithms (algorithms 5 and 6). Algorithm 5 was very similar to the implementation of algorithm 3, with the exception that we now select an edge e to make progress if and only if e decreases the number of connected components and if $U_A(\mathbf{v_e}) < 1$. This allows us to try to obtain a tree in $n-1$ steps. To check whether the candidate edge e will decrease the number of connected components, we compared the number of zero eigenvalues of $A_{t-1} + \mathbf{v_e}\mathbf{v_e}^{\mathrm{T}}$ to the number of zero eigenvalues of A_{t-1}. If the number of zero eigenvalues is different, than it means that edge e indeed decreases the number of connected components. This is true because the multiplicity of the zero eigenvalue in a Laplacian matrix counts the number of connected components of the graph defined by this Laplacian matrix.

```
Algorithm 5: Randomly Constructing Spectrally Thin Trees
  Input: $L_G$, 'filename', $u_0$, $\delta_U$
  Output: A file named 'filename', containing the outcome of the algorithm
  Precompute edge list $E$ and $(L_G^{\dagger})^{1/2}$. Initialize $A_0 = 0$ and $T = n-1$. Define
  the upper potential function $\Phi^u(A) = Tr(uI_{Im(L_G)} - A)^{-1}$;
  for $t = 1, 2, \ldots, T$ do
      for $i = 1, 2, \ldots, 30$ (that is, we will try a constant number of times) do
          Choose a random edge $e$ from edge list $E$
          if $U_A(\mathbf{v_e}) \leq 1$ AND e decreases the number of connected components of
          our current forest then
              $A_t \leftarrow A_{t-1} + \mathbf{v_e}\mathbf{v_e}^{\mathrm{T}}$
              break;
          end
          if $i = 30$ (this means we have found no edges in our attempt) then
              Write to 'filename' that the algorithm did not succeed;
              return fail
          end
      end
      $u_t \leftarrow u_{t-1} + \delta_U$;
  end
  return $A_T$ and write the input and $A_T$ on 'filename'.
```

```
Algorithm 6: Finding Thin Trees By Growing One Connected Component
Input: $L_G$, 'filename', $u_0$, $\delta_U$
Output: A file named 'filename', containing the outcome of the algorithm
Precompute adjacency list $AL$ and $(L_G^\dagger)^{1/2}$. Initialize $A_0 = 0$, $C = \{w\}$, where
$w$ is a random vertex of $G$, and $T = n - 1$. Define the upper potential function
$\Phi^u(A) = Tr(uI_{Im(L_G)} - A)^{-1}$;
for $t = 1, 2, \dots, T$ do
    for $i = 1, 2, \dots, 30$ (that is, we will try a constant number of times) do
        Choose a random vertex $v$ from $C$ and a random edge $e \in AL(v)$
        if $U_A(\mathbf{v_e}) \leq 1$ AND $e \in \partial_G(C)$ then
            $A_t \leftarrow A_{t-1} + \mathbf{v_e}\mathbf{v_e^T}$
            break;
        end
        if $i = 30$ (this means we have found no edges in our attempt) then
            Write to 'filename' that the algorithm did not succeed;
            return fail
        end
    end
    $u_t \leftarrow u_{t-1} + \delta_U$;
end
return $A_T$ and write the input and $A_T$ on 'filename'.
```

Algorithm 6 was devised in order to try to obtain a tree by growing only one connected component. That is, in the first step we choose an arbitrary edge that allows us to make progress – forming our first non-trivial connected component, call it C – and from the first nonempty instance of C on we try to make progress only with edges of $\partial_G(C)$. In order to gain easy access to the edges in $\partial_G(C)$, we created an adjacency list of the graph in order to find these edges more easily. This way, we only loop through the vertices in C, checking which of their neighbors across the boundary allow us to make progress.

Algorithm 7 below was devised to test the construction and existence of spectral sparsifiers, following the approach from the pervious section. To choose the constant κ in the beginning, we decided to choose a random vector $\mathbf{v_e}$ from the edge vector list and calculate its upper and lower barrier. Then, we try to proceed with this value of κ until the end of the algorithm. We decided to approach this experiment as described in the pseudocode below in order to test the robustness of the algorithm, since the value of κ can vary according to the magnitude of $\mathbf{v_e}$. Hence, if we can often finish the algorithm with a good sparsifier, when we run this algorithm multiple times (sometimes even with the same input graph), this fact would suggest that there is a range of values for which the conjecture that there are spectral sparsifiers is true, in the case of random graphs.

And algorithm 8 is the variant of algorithm 7 when we change the potential functions from $Tr(uI - A)^{-1}$ to $-\log(\det(uI - A))$. (with the adjustments in the upper and lower barrier shift functions – i.e. U_A and L_A) We decided to test this variant in the case of spectral sparsifiers as well, in order to compare the efficiency of both potential functions in the task of finding spectral sparsifiers in which the weights of all the edges selected are all the same.

Algorithm 7: Randomly Constructing Spectral Sparsifiers

```
Input: $L_G$, $d > 1$ 'filename'
Output: A file named 'filename', containing the outcome of the algorithm
Precompute edge list $E$ and $(L_G^{\dagger})^{1/2}$. Initialize $A_0 = 0$. Set parameters $u_0$, $l_0$,
$\delta_L$ and $\delta_U$ as in [2] and $T = \lfloor d(n-1) \rfloor$. Define potential functions
$\Phi^u(A) = Tr(uI_n - A)^{-1}$ and $\Phi_l(A) = Tr(A - lI_n)^{-1}$ ;
Choose a random edge $e_0$ from edge list $E$ and set

    $$\frac{1}{\kappa} \leftarrow \frac{U_A(\mathbf{v}_{\mathbf{e}_0}) + L_A(\mathbf{v}_{\mathbf{e}_0})}{2}$$

for $t = 1, 2, \ldots, T$ do
    for $i = 1, 2, \ldots, 30$ (that is, we will try a constant number of times) do
        Choose a random edge $e$ from edge list $E$
        if $U_A(\mathbf{v}_\mathbf{e}) \leq 1/\kappa \leq L_A(\mathbf{v}_\mathbf{e})$ then
            $A_t \leftarrow A_{t-1} + \kappa \mathbf{v}_\mathbf{e} \mathbf{v}_\mathbf{e}^\mathbf{T}$
            break;
        end
        if $i = 30$ (this means we have found no edges in our attempt) then
            Write to 'filename' that the algorithm did not succeed;
            return fail
        end
    end
    $u_t \leftarrow u_{t-1} + \delta_U$;
    $l_t \leftarrow l_{t-1} + \delta_L$;
    If $u_t > 0.9\kappa$, break;
end
return $A = \frac{1}{l_T} A_T$ and write $\kappa$, the step $t$ in which the algorithm ended, $l_t$ and
$u_t$ on 'filename'
```

Algorithm 8: Randomly Constructing Spectral Sparsifiers With $-\log\det$ Potential Function

```
Input: $L_G$, $d > 1$ 'filename'
Output: A file named 'filename', containing the outcome of the algorithm
Precompute edge list $E$ and $(L_G^\dagger)^{1/2}$. Initialize $A_0 = 0$. Set parameters $u_0$, $l_0$,
$\delta_L$ and $\delta_U$ as in [2] and $T = \lfloor d(n-1) \rfloor$. Define potential functions
$\Phi^u(A) = -\log(\det(uI - A))$, $\Phi_l(A) = -\log(\det(A - lI))$ and define $U'_A(\mathbf{v_e})$ and
$L'_A(\mathbf{v_e})$ as in Algorithm 4. Choose a random edge $e_0$ from edge list $E$ and set

    $$\frac{1}{\kappa} \leftarrow \frac{U'_A(\mathbf{v_{e_0}}) + L'_A(\mathbf{v_{e_0}})}{2}$$

for $t = 1, 2, \ldots, T$ do
    for $i = 1, 2, \ldots, 30$ (that is, we will try a constant number of times) do
        Choose a random edge $e$ from edge list $E$
        if $U'_A(\mathbf{v_e}) \leq 1/\kappa \leq L'_A(\mathbf{v_e})$ then
            $A_t \leftarrow A_{t-1} + \kappa \mathbf{v_e} \mathbf{v_e^T}$
            break;
        end
        if $i = 30$ (this means we have found no edges in our attempt) then
            Write to 'filename' that the algorithm did not succeed;
            return fail
        end
    end
    $u_t \leftarrow u_{t-1} + \delta_U$;
    $l_t \leftarrow l_{t-1} + \delta_L$;
    If $u_t > 0.9\kappa$, break;
end
return $A = \frac{1}{l_T} A_T$ and write $\kappa$, the step $t$ in which the algorithm ended, $l_t$ and
$u_t$ on 'filename'
```

3.3 Analysis of Simulations and Conjectures

From the simulations that were done over the course of this year, we randomly generated 1000 examples of each random graph $G(n,p)$ that we present in tables 2, 4 and 5, and we ran each algorithm that we described above on these 1000 examples. The results suggest the following:

For the first question, as we ran Algorithm 2, we observed that a significant fraction of pairs of edges (e_1, e_2) have intervals where $U_A(\mathbf{v_{e_i}}) \leq L_A(\mathbf{v_{e_i}})$, for $i \in \{1,2\}$, and also have nonempty intersection, that is,

$$[U_A(\mathbf{v_{e_1}}), L_A(\mathbf{v_{e_1}})] \bigcap [U_A(\mathbf{v_{e_2}}), L_A(\mathbf{v_{e_2}})] \neq \emptyset.$$

From now on, we will call a pair (e_1, e_2) a *good pair* whenever (e_1, e_2) satisfies the conditions above. This observation was true in all of our test cases, as we can see some examples of our simulations in table 1. In addition, as we can also see in table 1, as we increase parameter p, the percentage of *good pairs* tend to increase.

We also noticed that the average values of the U_A's, which we denote by $\overline{U_A}$, tend to stay in a very limited range throughout the algorithm (and similarly for L_A), as we can see on table 1 below. Moreover, we observed that the maximum value of $\overline{U_A}$ tend to always be less than the least value taken by $\overline{L_A}$ throughout the run of algorithm 2. Notice also that in this table, the fraction of *good pairs* – calculated over all pairs of edges – is always around the square of the fraction of edges e that have $U_A(\mathbf{v_e}) \leq L_A(\mathbf{v_e})$ and nonempty intersection with $[\overline{U_A}, \overline{L_A}]$. This last fact implies that a significant fraction of *good pairs* (e_1, e_2) occur when both intervals $[U_A(\mathbf{v_{e_i}}), L_A(\mathbf{v_{e_i}})]$, for $i \in \{1,2\}$, intersect $[\overline{U_A}, \overline{L_A}]$. Hence, all the facts in this paragraph suggest that we should expect a high success rate of algorithm 7, since the intervals $[\overline{U_A}, \overline{L_A}]$ tend to overlap during the whole execution of the algorithm and also we have many edges e for which $U_A(\mathbf{v_e}) \leq L_A(\mathbf{v_e})$ and $[U_A(\mathbf{v_e}), L_A(\mathbf{v_e})] \bigcap [\overline{U_A}, \overline{L_A}] \neq \emptyset$.

Conversely, another fact that suggests the behavior that we see in table 1 is the high rate of success of algorithm 7 on table 5, since this algorithm only succeeds if at every main step we are able to find an edge e for which $U_A(\mathbf{v_e}) \leq \kappa \leq L_A(\mathbf{v_e})$ in less than 30 random attempts, where κ is a constant that is precomputed in algorithm 7. Since the success rate is high, this implies that there exists a range of values of κ for which at every step t of the algorithm, there is a significant fraction of edges whose intervals contain κ, and therefore have nonempty intersection.

Table 1: Statistics for First Investigation

$G(n,p)$	ranges of $\overline{U_A}$	ranges of $\overline{L_A}$	num of intersecting intervals at 90% of the rounds	intersections with $[\overline{U_A}, \overline{L_A}]$ at 90% of the rounds
$G(200, 1/10)$	$[2.06, 2.07] \cdot 10^{-4}$	$[2.10, 2.11] \cdot 10^{-4}$	between 20% and 60%	between 40% and 80%
$G(300, 1/20)$	$[1.88, 1.89] \cdot 10^{-4}$	$[1.90, 1.92] \cdot 10^{-4}$	between 21% and 29%	between 35% and 50%
$G(300, 1/20)$	$[1.94, 1.96] \cdot 10^{-4}$	$[1.98, 2.00] \cdot 10^{-4}$	between 18% and 24%	between 30% and 45%

If we look at the runtimes that we observe in table 2 and compare them to the fact that it took longer than one day to finish running the naive implementation of BSS (given by algorithm 2) on a graph $G(300, 1/20)$, and if we also take into account the success rate of the algorithm 3 shown on table 5 (it always returned a spectral weighted sparsifier) we conclude that the runtimes of the two implementations (algorithms 2 and 3) also suggest that there are many edges available at every main step of the algorithm. For if this were not the case, then algorithm 3 would halt early and fail for some instances, since at each main step of algorithm 3 we are only allowed 30 random attempts to try to add an edge to the sparsifier and make progress – otherwise we halt.

Table 2: Running Times of algorithm 3

G(n,p)	Average Runtime (in seconds)
G(200, 1/10)	110
G(300, 1/20)	250
G(400, 1/20)	720
G(500, 1/20)	1490
G(600, 1/20)	2750

The significant amount of edges e for which $U_A(\mathbf{v_e}) \leq L_A(\mathbf{v_e})$ at most steps of algorithm 2, as we can see in table 1, causes us to ask the following question: how many times does a specific edge e has $U_A(\mathbf{v_e}) \leq L_A(\mathbf{v_e})$ throughout algorithm 2? The answer to this question can give us a better sense on the distribution of the edges that have $U_A(\mathbf{v_e}) \leq L_A(\mathbf{v_e})$ at each step of algorithm 2.

According to table 3, which shows some common examples of our simulations, every edge tends to have $U_A(\mathbf{v_e}) \leq L_A(\mathbf{v_e})$ at least once throughout the algorithm (and usually all of them satisfy $U_A(\mathbf{v_e}) \leq L_A(\mathbf{v_e})$ in the first round). From the second column of table 3, we see that most edges satisfy $U_A(\mathbf{v_e}) \leq L_A(\mathbf{v_e})$ in more than 10% of the steps of algorithm 2. From the third column of table 3, we see that at least half of the edges tend to have $U_A(\mathbf{v_e}) \leq L_A(\mathbf{v_e})$ in at least 70% of the steps of the algorithm. From analyzing the data more closely, we see that the set of all the edges which have $U_A(\mathbf{v_e}) \leq L_A(\mathbf{v_e})$ in more than 70% of the steps is incident on all vertices, which implies that for each vertex v we often have, at each step of the algorithm, at least one edge e satisfying $U_A(\mathbf{v_e}) \leq L_A(\mathbf{v_e})$ such that $v \in e$. We also noticed that the more we increased parameter p – i.e., the probability that an edge is chosen to be in the graph – the higher the fractions would become, as is evident when we compare the vales obtained for $G(300, 1/20)$ and $G(300, 1/10)$ in table 3 below.

Table 3: Fraction of steps where each edge is available

G(n,p)	edge that appeared least	10 percentile	median	edge that appeared most
G(200, 1/10)	0.5%	16%	66%	100%
G(300, 1/20)	1.5%	13%	75%	100%
G(300, 1/20)	0.7%	15%	76%	100%
G(300, 1/10)	3%	22%	82%	100%
G(300, 1/10)	5%	26%	85%	100%

As for the second question, algorithm 4 with the new potential functions ($-\log\det(uI-A)$ for the upper potential and $-\log\det(A-lI)$ for the lower potential) proved to be rather inefficient if compared to the results obtained for algorithm 3. As we can clearly see in table 4, the output of this modification of the algorithm almost always returned a matrix with a spectral ratio (that is, the ratio $\lambda_{max}(A)/\lambda_{min}(A)$) that would be much larger than our $\dfrac{d+1+2\sqrt{d}}{d+1-2\sqrt{d}}$ bound. According to table 4, since we started with the input $d = 3$, we would expect to find a spectral gap that is $< \dfrac{d+1+2\sqrt{d}}{d+1-2\sqrt{d}} < 14$. However, in most of the simulations done, where we set $d = 3$, we found that this algorithm would output a matrix with spectral ratio in the range $[20, 40]$. And as we increased the size of the graph, the spectral ratio of the output matrix A would increase as well, as we can see in table 4. This result was very surprising, and it suggests that the original potential functions might be more robust than the ones that we chose above for algorithm 4. Although the reasons for the observed behavior with the new potential functions are not understood, one possible explanation for this behavior is based on the fact that with the new potential functions, we are allowed to have many eigenvalues close to the barriers, without having a huge discrepancy in the potential function, whereas the same is not possible with the potential function used in BSS, for they explain it in section 3.2 of [2].

Table 4: Eigenvalue Ranges for Outputs of Algorithm 4, with input $d = 3$

$G(n,p)$	$< u_T$	in $[14, 20]$	in $[20, 40]$	> 40
$G(250, 1/10)$	0%	30%	61%	9%
$G(300, 1/20)$	3.3%	29.4%	57.9%	9.4%
$G(400, 1/20)$	0%	22.5%	56.3%	21.2%
$G(500, 1/20)$	0%	4.8%	27.4%	67.8%
$G(600, 1/20)$	0%	0%	63%	37%

For the question of constructing a spectrally thin spanning tree, from table 5 we observe that algorithm 5 (the one that creates a thin tree by randomly trying to add edges of the edge set E at each main step) almost always returned a spectral thin tree when we had $2.7 < \lambda_2 < 3 + 2\sqrt{2}$ and set the initial parameters $u_0 = 1/2$ and $\delta_U = 1/4n$. The algorithm would also run fast – a runtime which would be on average 80 seconds – a third of the average runtime of algorithm 3 on $G(300, 15)$ graphs (which is = 250 seconds, from table 2). When we tried smaller initial parameters – $u_0 = 1/3$ and $\delta_U = 1/3n$ – algorithm 5 would almost never return a spectrally thin tree. For values of $\lambda_2 < 2$, the algorithm would almost never return a spanning tree, independently of the initial parameters. Therefore, we conjecture that it is possible to find spectrally thin trees for graphs with $2.7 < \lambda_2 < 3 + 2\sqrt{2}$, given the following initial parameters: $u_0 = 1/2$ and $\delta_U = 1/4n$. Table 5 shows only the rates of success of algorithm 5 for initial parameters $u_0 = 1/2$ and $\delta_U = 1/4n$.

Whenever algorithm 5 would fail in returning a spectrally thin tree, it would return a forest with one giant component (of size greater than $n/2$) and the rest of the forest would be made out of small components. This behavior is as we expected, because at each iteration of our algorithm, we have many available edges to add to the

Table 5: Rates of Success of the Experiments (in percentage)

$G(n,p)$	Algorithm 3	Algorithm 4	Algorithm 5	Algorithm 6	Algorithm 7	Algorithm 8
$G(200, 1/10)$	100	5	90	53	83	68
$G(250, 1/10)$	100	0	96	60	95	59
$G(300, 1/10)$	100	0	93	46	97	68
$G(300, 1/20)$	100	3	76	12	60	43
$G(400, 1/20)$	100	0	92	15	84	45
$G(500, 1/20)$	100	0	94	14	91	54

graph, since for the initial parameters chosen above, we have that $\sum_{e \in E(G)} U_A(\mathbf{v_e}) \leq 5n$, which implies that for any set of $5n$ edges, we must have one for which $U_A(\mathbf{v_e}) \leq 1$, and therefore we can add it to our graph. Due to the fact that we have a total of $dn/2$ edges in $G(n,d)$, in expectation, at each step of the algorithm we obtain a large number of available edges. Moreover, the fact that our graphs are good expanders with high probability implies that we would obtain a giant component – otherwise, by the expansion property, there would be more than $5n$ edges going across boundaries of connected components.

The surprising part of the investigation of the this third question was that when we tried to grow a tree by forming a single component and taking only edges from its boundary (algorithm 6), our simulations failed considerably more often than they failed in algorithm 5, according to the data in table 5. This suggests that the right way to construct these thin trees is by some sort of load balancing procedure, where we try to keep the sizes of the components as similar as possible, until we cannot help but getting a giant component – at which stage all of the remaining components will be of a large enough size. And at this stage, due to the expansion properties of the graph (if the components all have sizes $\geq 5n/\lambda_2$), we would always be able to connect all of the components, forming a spanning tree. Unfortunately, we were not able to prove any positive result in this approach.

For the question concerning the construction of spectral sparsifiers, from table 5 we can observe that algorithm 7 returned in most cases an unweighted spectral sparsifier. However, from the data in table 4, algorithm 8 (where we used the log det barrier functions) was not as efficient in returning an unweighted sparsifier for the same set of graphs. In both algorithms 7 and 8, the average runtime was similar to the runtime found for algorithm 3 in table 2. Since algorithm 7 chooses the candidate edges randomly at each main step, we can also infer that it should be relatively easy to find an instance of a spectral sparsifier for a random graph, otherwise our implementation would halt for a significant fraction of our simulations, which is not what we observed in table 4.

4. Conclusion

Throughout this year, we performed extensive numerical simulations in order to empirically investigate the existence and frequency of unweighted spectral sparsifiers and of spectrally thin trees among random graphs. Although the results of the experiments seem to indicate that most sufficiently dense random graphs have unweighted spectral sparsifiers (and hence spectrally thin trees), it still seems very difficult to directly prove, by using the methods introduced by Batson, Spielman and Srivastava, that we can even obtain spectrally thin spanning trees when we assume that the graph has large enough expansion.

Bibliography

[1] Asadpour, Arash and Goemans, Michel X. and Madry, Aleksander and Gharan, Shayan Oveis and Saberi, Amin.
An $O(\log n/\log\log n)$-approximation algorithm for the asymmetric traveling salesman problem. In *Proceedings of the Twenty-First Annual ACM-SIAM Symposium on Discrete Algorithms*, 2010, 379–389.

[2] Batson, Joshua D. and Spielman, Daniel A. and Srivastava, Nikhil.
Twice-Ramanujan Sparsifiers. In *Proceedings of the 41st annual ACM symposium on Theory of computing*, 2009, 255–262.

[3] Benczúr, András A. and Karger, David R.
Approximating s-t minimum cuts in $\tilde{O}(n^2)$ time. In *Proceedings of the twenty-eighth annual ACM symposium on Theory of computing*, 1996, 47–55.

[4] Goemans, Michel X. Personal communication.

[5] S. Hoory, N. Linial, and A. Wigderson.
Expander graphs and their applications. In *Bulletin of the American Mathematical Society*, 2006, 43:439–561.

[6] Spielman, Daniel A. and Teng, Shang-Hua.
Nearly-linear time algorithms for graph partitioning, graph sparsification, and solving linear systems. In *Proceedings of the thirty-sixth annual ACM symposium on Theory of computing*, 2004, 81–90.

[7] Spielman, Daniel A. and Teng, Shang-Hua.
Spectral Sparsification of Graphs Available at *http://arxiv.org/abs/0808.4134*.