Twice Ramanujan Sparsifiers

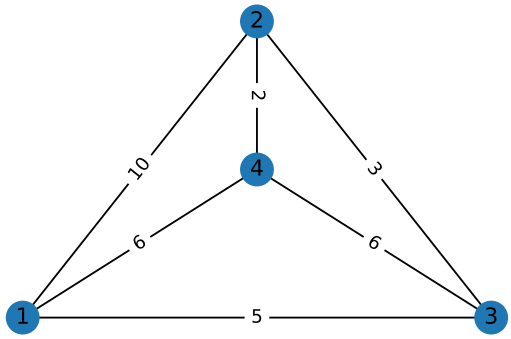
For any graph G a sparsifier H is a graph with far fewer edges that is similar to G in some useful way. While H is much easier to do computation on, it holds the same properties as G, and therefore, it is a reliable way of doing approximate computation on G. For example, if we are dealing with path-finding problems on a dense large graph G, the set of sparsifiers used in (Chew 1989) can be used because they are guaranteed to have almost the same shortest path properties as G.

For illustration, consider the following graph G with four vertices. The new graph obtained has far fewer edges but has the same set of shortest paths between any pair of vertices. This is a simple sparsifier that can be used for shortest path-finding problems and can be obtained via removing trivial edges $w(u, v)$ such that the shortest distance between u and v is smaller than $w(u, v)$.

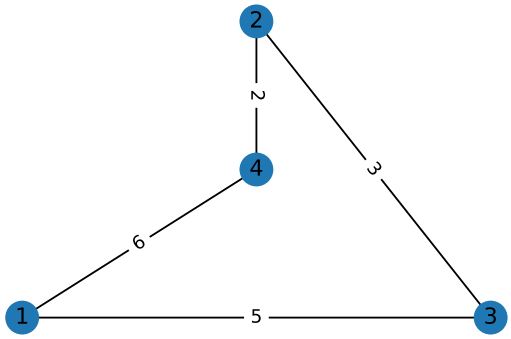
```
import networkx as nx
import matplotlib.pyplot as plt

# setup the graph
G = nx.Graph()
G.add_nodes_from([1, 2, 3, 4])
G.add_edges_from([
  (1, 2, {'w':10}),
  (1, 3, {'w':5}),
  (1, 4, {'w':6}),
  (2, 3, {'w':3}),
  (2, 4, {'w':2}),
  (3, 4, {'w':6})
])
# setup plotting position of all vertices
pos={
  1:(0,0),
  2:(0.5,1),
  3:(1, 0),
```

```
  4:(0.5, 0.5)
}

# a simple networkx plotting function
def plot_graph():
  nx.draw_networkx(G,pos)
  labels = nx.get_edge_attributes(G,'w')
  nx.draw_networkx_edge_labels(G,pos,edge_labels=labels)
  plt.axis('off')
  plt.show()

# before:
plot_graph()

# find the shortest path between any pair of vertices
shortest_paths = dict(nx.all_pairs_dijkstra_path(G, weight='w'))
for v in shortest_paths:
    for u in shortest_paths[v]:
      # if the edge from v to u has weight greater than the shortest path
      # between v and u, then remove it
      if v != u and len(shortest_paths[v][u]) > 2:
        # remove edge from v to u if it exists
        if G.has_edge(v, u):
          G.remove_edge(v, u)

# after:
plot_graph()
```

On the other hand, (Benczúr and Karger 1996) for example introduces the cut-sparsifiers which are a class of sparsifiers that have almost identical cut weights for any set $S \subset V$. In this report, we cover spectral graph sparsifiers which are a certain class of sparsifiers that have a tight connection with expander graphs and can approximate the Laplacian of a graph with high accuracy. Because of the close connection between graph spectral connectivity and edge connectivity introduced by Cheeger (Cheeger 1970) spectral sparsifiers were introduced by (Spielman and Teng 2004) and (Spielman and Teng 2011). Conventionally, these graphs are constructed using randomized algorithms where we pick a certain edge of an original graph with a probability. For example, if an edge is crucial to the connectivity of our graph, then it has high importance and should be picked with high probability. However, in this report, we will show that we can construct a sparsifier with a deterministic algorithm introduced in (Batson, Spielman, and Srivastava 2009) that has a tight connection with the Ramanujan bounds.



(a) The graph G that we intend to sparsify.



(b) The graph H that is obtained by removing trivial edges.

Figure 1: A simple illustration of a sparsifier that can help with shortest path problems.

Furthermore, we will cover an important reduction from the graph sparsification problem to a matrix approximation problem which has been further exploder in many follow-up papers (Tat Lee and Sun 2015) and (Lee and Sun 2017). Moreover, this will give us the first deterministic algorithm for obtaining sparsifiers with linear edge count. That said, we have implemented the algorithm in Python and have tested it on a few graphs for illustration purposes.

Finally, we will focus our attention on running the algorithm on complete graphs. The sparsifier obtained from the complete graph will have high connectivity which resembles similarities with the expander graphs. Although the graph obtained from the algorithm is not regular, we will show that it has a lot of expander-like properties and we will draw a close connection with Ramanujan graphs.

Recap and Preliminaries

Here we will cover some preliminaries on spectral sparsification and then we will discuss the effective resistance-based algorithm for spectral sparsification. We will also discuss an important reduction to the matrix problem that was first formalized in (Batson, Spielman, and Srivastava 2009) which lays the groundwork for the final algorithm.

Spectral Sparsification

Before everything, we should define what a spectral sparsifier is. A spectral sparsifier is a sparse graph that approximates the Laplacian of a graph with high accuracy. In other words, a sparsifier is a graph that has a lot of the same properties as the original graph.

Definition 0.1. A (k, ϵ)-spectral sparsifier of a graph $G = (V, E, w)$ is a graph H with k edges such that,

$$L_G \approx_\epsilon L_H : (1-\epsilon)L_G \preceq L_H \preceq (1+\epsilon)L_G$$

where L_G is the Laplacian of G and L_H is the Laplacian of H.

Reduction to the Matrix Problem

Here, we will present an analog problem for the sparsification of matrices that is tightly connected to the spectral sparsification problem. The problem is as follows:

Definition 0.2. (k, ϵ)**-approximation of matrices** Given a set of m vectors $v_1, \dots, v_m \in \mathbb{R}^n$ if $A = \sum_{i=1}^m v_i v_i^T$ is a positive semi-definite matrix, then we intend to find a subset of vectors $\mathcal{S} \subseteq \{1, \dots, m\}$ of size k and a set of coefficients $s_i \in \mathbb{R}^+$ such that $\hat{A} = \sum_{i \in \mathcal{S}} s_i v_i v_i^T \approx_\epsilon A$.

Now we will show that one can solve the (k, ϵ) problem in Definition **??** then it can plug into the graph sparsification problem and obtain a (k, ϵ)-spectral sparsifier. To do so, observe that if we set $A = L_G$ and $v_{ab} = \sqrt{w_G(a,b)}(\chi_a - \chi_b)$ and $s_{ab} = \frac{w_H(a,b)}{w_G(a,b)}$, then the problem in Definition **??** is equivalent to the spectral sparsification problem:

$$\begin{aligned}
A = L_G &= \sum_{(a,b)\in E(G)} w_G(a,b) L_{ab} \\
&= \sum_{(a,b)\in E(G)} \sqrt{w_G(a,b)}^2 (\chi_a - \chi_b)(\chi_a - \chi_b)^T \\
&= \sum_{ab \in E(G)} v_{ab} v_{ab}^T \\
\hat{A} = L_H &= \sum_{(a,b)\in E(H)} w_H(a,b) L_{ab} \\
&= \sum_{(a,b)\in E(H)} \frac{w_H(a,b)}{w_G(a,b)} \sqrt{w_G(a,b)}^2 (\chi_a - \chi_b)(\chi_a - \chi_b)^T \\
&= \sum_{(a,b)\in E(H)} s_{ab} v_{ab} v_{ab}^T
\end{aligned}$$

Sampling-based Sparsification

As alluded to previously, the problem of spectral sparsification can be approached from an edge-sampling perspective. In particular, one can assign importance weights to each edge and then come up with a sampling scheme that samples edges according to their importance. For example, an edge that is crucial for the connectivity of the graph has high importance for cut-sparsifiers or spectral sparsifiers. To that end, a set of edges can be independently sampled according to this scheme and after sampling each edge the graph becomes more and more similar to the original graph. However, since this sampling is done according to the measure of importance, even after sampling a small number of edges, the graph becomes a good approximation of the original graph.

One can also formulate the same thing for the matrix approximation problem. Assume that for each vector i, we have a corresponding matrix $X_i = s_i v_i v_i^T$ which will be picked with probability p_i and we will consider $\widehat{A} = \sum_{i \in \mathcal{S}} X_i$ where $\mathcal{S}$ is the set of indices of the sampled vectors. One can bound the number of sampled vectors by coming up with good probabilities p_i such that $\sum p_i$ is bounded by k and can also bound the error of the approximation by using matrix concentration bounds. However, these algorithms tend to have the following problems:

1. The algorithm is not deterministic meaning that there is a very low chance of producing a large set $\mathcal{S}$.
2. The algorithm is not deterministic meaning that there is a very low chance of producing an approximate $\widehat{A}$ which is not close to A.
3. Because these algorithms rely on exponential concentration bounds, typically they require to sample $\mathcal{O}(n \cdot polylog(n))$ vectors to achieve a good approximation.

Although flawed, these solutions are easy and using this idea, a set of sampling techniques have been proposed to tackle the problem of sparsification with the most famous among them being the **effective-resistance** based sparsifiers (Spielman and Srivastava 2008). We will briefly cover the main idea and intuition behind this and redirect the reader to other resources for further detailed reading.

The effective resistance between two nodes a and b is the equivalent resistance if we assume that the rest of the nodes are harmonic and only one external current is given to a and one external current is taken from b; then, the measured voltage difference between these two nodes will denote the effective resistance which can be written as $(\chi_a - \chi_b)^T L_G^+ (\chi_a - \chi_b)$ using Laplacians. Moreover, effective resistances have a combinatorial interpretation as well. If we assume we sample spanning trees proportional to their weight products, then the effective resistance between two nodes is proportional to the probability of the edge between those two nodes appearing. This means that a crucial edge in the connectivity, will have a high probability of appearing in the sampled spanning trees and thus will have a high effective resistance; that said, this will yield a high importance weight for that edge and thus it will be sampled more often:

Effective-resistance based sparsifier For each edge $(a, b) \in E$, sample (a, b) with probability $p(a, b) = \min\big(1, C \cdot (\log n)\epsilon^{-2} w(a, b) R_{eff}(a, b)\big)$. Where $R_{eff}(a, b)$ is the effective resistance between a and b. Using Rudelson concentration lemma (Rudelson 1999), (Spielman and Srivastava 2008) shows that for a certain constant C after picking $\mathcal{O}(n \log n/\epsilon)$ edges the resulting graph is a ϵ-spectral sparsifier with high probability.

Method

We will now discuss the deterministic algorithm for approximating the matrix A. The algorithm takes an iterative approach and follows k iterations. At each iteration, it will pick a vector v_i which corresponds to an edge and will add $s_i v_i v_i^T$ to the current accumulated matrix. After k iterations it will give a good approximate for the matrix A. But before we present the bulk of the algorithm, let's start by laying some groundwork by presenting some useful intuitions.

Geometric interpretation

Note that for any pair of matrices A and B, having the same null-space we have that $A \succeq B \iff I \succeq A^{+/2} B A^{+/2}$. Hence,

$$(1 - \epsilon)A \approx_\epsilon B \iff \Pi \approx_\epsilon A^{+/2} B A^{+/2}$$

where $\Pi = A^{+/2} A A^{+/2}$ is the identity in the subspace orthogonal to the null space of A and is an *idempotent* matrix. In other words, $\Pi^2 = \Pi$. Therefore, without loss of generality, we may assume that A in Definition **??** is an idempotent matrix Π via the transformation described where A is replaced by $A^{+/2} A A^{+/2}$ and $v_i = A^{+/2} v_i$ for all $1 \leq i \leq m$.

With that in mind, thinking about idempotent matrices yields nice intuitions on how to think about the problem geometrically. Furthermore, for any positive semi-definite matrix M we can define an ellipsoid $\{x | x^T M x = 1\}$ and for $M = \Pi$ being an idempotent matrix the ellipsoid corresponds to the sphere in the linearly transformed subspace of Π:

$$x^T \Pi x = x^T \Pi \Pi x = ||\Pi x||_2^2 = 1.$$

Therefore, if we consider everything in the mapped subspace, i.e., replacing every vector x with Πx automatically, then we want to find a linear combination of their cross product such that the ellipsoid corresponding to that combination approximates a regular spherical shape.

In other words,

$$\begin{aligned}
&\hat{A} = \sum s_i v_i v_i^T \approx_\epsilon A \\
\iff &\hat{\Pi} = \sum s_i (A^{+/2}) v_i (A^{+/2} v_i)^T \approx_\epsilon A^{+/2} A A^{+/2} = \Pi \\
\iff &(1-\epsilon)\Pi \preceq \hat{\Pi} \preceq (1+\epsilon)\Pi \\
\iff &\forall x : (1-\epsilon)||\Pi x||_2^2 \leq [\Pi x]^T \hat{\Pi} [\Pi x] \leq (1+\epsilon)||\Pi x||_2^2
\end{aligned}$$

Therefore, the ellipsoid projected using Π is sandwiched between two spheres off by ϵ in their radius. Therefore, the algorithm takes an iterative approach to solve this geometric problem. It first starts of with $\hat{A}^{(0)} = \emptyset$ and then iteratively picks a vector v_i and assigns a weight s_i to it such that the ellipsoid $\hat{A}^{(i)} = \hat{A}^{(i+1)} + s_i v_i v_i^T$ becomes iteratively more like a sphere. To formalize this, the algorithm always bounds the corresponding ellipsoid between two spheres of radius $l^{(i)}$ and $u^{(i)}$. At the beginning of each iteration, the lower bound $l^{(i)}$ will be increased by some δ_l and the lower bound $u^{(i)}$ will be increased by some δ_u and the algorithm will try to find a vector v_i and a weight s_i such that the new ellipsoid $\hat{A}^{(i+1)}$. Moreover, the key idea here is to cleverly pick δ_l and δ_u values such that after k iterations the gap between the two spheres is off by ϵ. In other words, the following should hold:

$$\frac{u^{(0)} + k\delta_u}{l^{(k)} + k\delta_l} = \frac{u^{(k)}}{l^{(k)}} \leq \frac{1+\epsilon}{1-\epsilon}.$$

This will ensure that the shape of the ellipsoid becomes more and more spherical as the algorithm progresses, and finally, a simple scaling will yield an approximate unit sphere which is what we want.

For illustration, the following shows the algorithm in action. The algorithm starts with an initial ellipsoid that is not spherical and then iteratively picks a vector v_i and a weight s_i such that it still remains sandwiched between two spheres of radius $l^{(i)}$ and $u^{(i)}$. Note that in this example δ_l and δ_u are equal, therefore, for a large enough k the ellipsoid will become spherical because although the radius is growing the gap remains the same, further limiting the range of the ellipsoid.

Physical View and the Expected behavior

The fact that $\hat{A}^{(i)}$ should be bounded between two spheres translates into all the eigenvalues of $\hat{A}^{(i)}$ being bounded between the two radiuses. Therefore, an important observation is to monitor what happens to the eigenvalues of $\hat{A}^{(i)}$ when vv^T is being added at each iteration. To do so, we consider the characteristic polynomial of A at each iteration written as $p_A(\lambda) = \det(\lambda I - A)$. There are two important lemmas when analyzing $A + vv^T$ matrices, one is the Sherman-Morrison lemma which states that:

Suppose A is an invertible square matrix and u, v are column vectors. Then $A + uv^T$ is invertible iff $1 + v^T A^{-1} u \neq 0$. In this case,

$$(A + uv^T)^{-1} = A^{-1} - \frac{A^{-1} uv^T A^{-1}}{1 + v^T A^{-1} u}$$

The other is the matrix determinant lemma which states that:

Suppose A is an invertible square matrix and u, v are column vectors. Then

$$\det(A + uv^T) = \det(A)(1 + v^T A^{-1} u)$$

Moreover, plugging these into the characteristic polynomial of $A + vv^T$ yields the following:

$$\begin{aligned} p_{A+vv^T}(\lambda) &= \det(\lambda I - A - vv^T) \\ &= \det(\lambda I - A)(1 - v^T (\lambda I - A)^{-1} u) \\ &= \det(\lambda I - A) \left(1 - v^T \left[\sum_{i=1}^{n} \frac{1}{\lambda - \lambda_i} u_i u_i^T\right] v\right) \\ &= p_A(\lambda) \left(1 - \sum_{i=1}^{n} \frac{(v^T u_i)^2}{\lambda - \lambda_i}\right) \end{aligned}$$

Therefore, we can assume particles being set on each of the λ_i values with the ith one on λ_i with charge $v^T u_i$. The new set of equilibrium points for this particle set will entail the new eigenvalues of $A + vv^T$ which are the roots of $p_{A+vv^T}(\lambda)$. Note that for u_i values such that $v^T u_i = 0$ the charge is zero and therefore, the new eigenvalues will be the same as the old ones.

The following figure illustrates the matrix case A with three different vectors v_1, v_2 and v_3. Each color corresponds to the characteristic polynomial for different v values where,

$$A = \lambda_1 u_1 u_1^T + \lambda_2 u_2 u_2^T + \lambda_3 u_3 u_3^T = \begin{bmatrix} 1.6 & -0.2 & -0.33 \\ -0.2 & 3.4 & -0.33 \\ -0.33 & -0.33 & 1 \end{bmatrix}$$

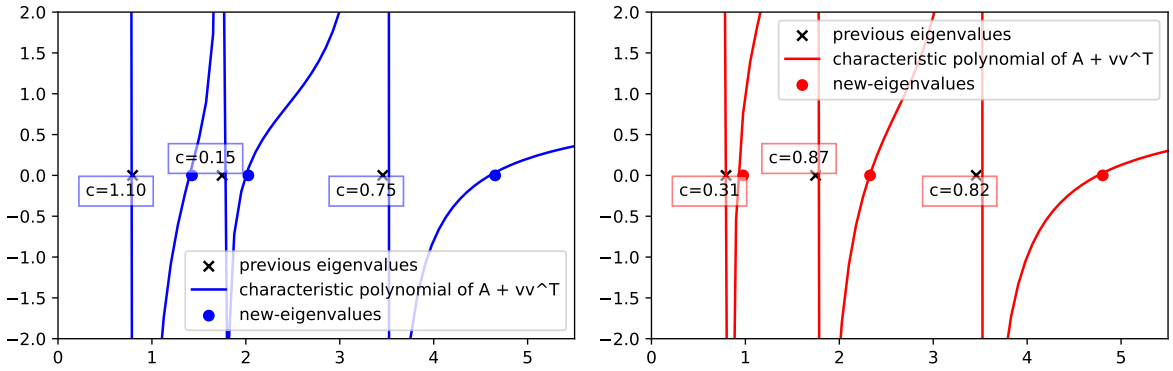
$$\begin{bmatrix} \lambda_1 \\ \lambda_2 \\ \lambda_3 \end{bmatrix} = \begin{bmatrix} 0.79 \\ 1.75 \\ 3.46 \end{bmatrix}, u_1 = \begin{bmatrix} -0.41 \\ -0.15 \\ -0.9 \end{bmatrix}, u_2 = \begin{bmatrix} -0.9 \\ -0.03 \\ 0.42 \end{bmatrix}, u_3 = \begin{bmatrix} 0.08 \\ -0.99 \\ 0.12 \end{bmatrix}$$

We note that $\langle v_i, u_j \rangle^2$ is the charge of particle j when adding v_i to A and we can summarize all the charged particles in the following matrix:

$$v_1 = \begin{bmatrix} 0 \\ 1 \\ 1 \end{bmatrix}, v_2 = \begin{bmatrix} 1 \\ 1 \\ 0 \end{bmatrix}, C = \begin{bmatrix} 1.10 & 0.15 & 0.75 \\ 0.31 & 0.87 & 0.82 \end{bmatrix}, C_{ij} = \langle v_i, u_j \rangle^2$$

```
import numpy as np
import matplotlib.pyplot as plt

def plot_characteristic_polynomial(A, v, color):
  x = np.linspace(-25, 50, 1000)
  w, u = np.linalg.eig(A)
  # plot the determinant of xI - A
  y = []
  roots = []
  prv = 0
  for i in x:
    val = 1 - np.sum(1/(i - w) * (v @ u)**2)
    if prv < 0 and val > 0:
      roots.append(i)
    prv = val
    y.append(val)
  plt.plot(x, y, color = color, label='characteristic polynomial of A + vv^T')
  plt.scatter(roots, np.zeros(len(roots)), color = color, marker = 'o', label='new-eigenva

# create an orthonormal 3 by 3 matrix U
U = np.array([[1, 1, 1], [1, 1, -2], [1, -1, 0]])
U = U / np.linalg.norm(U, axis = 0)
A = U @ np.diag([1 , 2, 3]) @ U.T



vz = [[0, 1, 1], [1, 1, 0]]

colors = ['blue', 'red']

# plot the eigenvalues of A
w, u = np.linalg.eig(A)
# sort according to w and reorder u
w, u = zip(*sorted(zip(w, u.T)))
u = np.array(u).T
```

```
w = np.array(w)

for col, v in zip(colors, vz):
  plt.scatter(w, np.zeros(w.shape), color = 'black', marker = 'x', label='previous eigenva
  # add text with textbox equal to np.sum(w) on top of each eigenvalue
  for i, wi in enumerate(w):
    t = plt.text(wi - 0.5, 0.1 * (4 * (i % 2) - 2.5), f"c={(v @ u[:,i])**2:.2f}", color =
    t.set_bbox(dict(facecolor='white', alpha=0.5, edgecolor=col))

  plot_characteristic_polynomial(A, v, col)
  plt.xlim(0, 5.5)
  plt.ylim(-2, 2)
  plt.legend()
  plt.show()
```



(a) The characteristic polynomial after adding v_1 to A.

(b) The characteristic polynomial after adding v_2 to A.

Figure 2: The characteristic polynomial of $A + vv^T$ for different v values, the higher the charge the more it will repel the new eigenvalues from the old ones.

The goal is to pick a v such that all the eigenvalues are uniformly pushed forward so that they can stay between the new ranges $l^{(i+1)}$ and $u^{(i+1)}$. To get a sense, let's pick one of the m vectors with uniform probability and add it to A. In that case, the expected charges can be written as:

$$E[\langle v, u_j \rangle^2] = \frac{1}{m} \sum_{i=1}^{m} \langle v_i, u_j \rangle^2 = \frac{1}{m} u_j^T \left(\sum_{i=1}^{m} v_i v_i^T \right) u_j = \frac{||\Pi u_j||_2^2}{m} = \frac{1}{m}$$

Hence, on expectation all the particles have charge $1/m$ and the expected deterministic polynomial is:

$$
\begin{aligned}
E[p_{A+v}(\lambda)] &= p_A(\lambda) E\left[1 - \sum_{i=1}^{m} \frac{\langle u_i, v\rangle^2}{\lambda - \lambda_i}\right] = p_A(\lambda)\left(1 - \sum_{i=1}^{m} \frac{E\langle u_i, v\rangle^2}{\lambda - \lambda_i}\right) \\
&= p_A(\lambda)\left(1 - \sum_{i=1}^{m} \frac{1/m}{\lambda - \lambda_i}\right) = p_A(\lambda) - \frac{1}{m}\sum_{i=1}^{m} \frac{p_A(\lambda)}{\lambda - \lambda_i} \\
&= p_A(\lambda) - \frac{1}{m}\sum_{i=1}^{m} \prod_{1=j\neq i}^{m} (\lambda - \lambda_j) \\
&= p_A(\lambda) - \frac{1}{m} p'_A(\lambda)
\end{aligned}
$$

Therefore, if we start off with the matrix $p_{A^{(0)}}(\lambda) = \lambda^n$, after nd iterations the expected polynomial is a set of associate Laguerre polynomials that are well studied (Dette and Studden 1995), and in particular, it has been proven that the ratio between the largest and smallest root for these polynomials is bounded by the value below:

$$
\frac{d + 1 + 2\sqrt{d}}{d + 1 - 2\sqrt{d}} \xrightarrow{\epsilon = \frac{2\sqrt{d}}{d+1}} \frac{1+\epsilon}{1-\epsilon}
$$

Although this is just speculation and no v_i values will necessarily exist with the expected behavior, we can still get an idea of the goal ϵ and come up with the following proposition:

Proposition 0.1. *For any matrix $A = \sum_{i=1}^{m} v_i v_i^T$ we can choose a subset $\mathcal{S}$ of v_i and a set of coefficients s_i with size nd such that:*

$$
\hat{A} = \sum_{i \in \mathcal{S}} s_i \cdot v_i v_i^T, \quad (1 - \frac{2\sqrt{d}}{d+1})A \preceq \hat{A} \preceq (1 + \frac{2\sqrt{d}}{d+1})A
$$

The graph formulation of Proposition **??** is as follows:

Corollary 0.1. *For any graph G and any ϵ we can choose a subset of $\mathcal{O}(n/\epsilon^2)$ edges with arbitrary edge weights to obtain H such that H is an ϵ-sparsifier of G: $L_G \approx_\epsilon L_H$.*

This is set using $\epsilon = \frac{2\sqrt{d}}{d+1}$ where $\frac{n}{\epsilon^2} = \mathcal{O}(nd)$. In the next section, we will see how we can choose v_i and s_i at each step such that after nd iterations this happens.

Potential Functions

The big question is, how can we quantize the boundedness of the matrix A at each step? We want $A^{(i)}$ to have eigenvalues that are bounded by $l^{(i+1)}$ and $u^{(i+1)}$; and so, we use a family of **potential functions** that explode when the eigenvalues approach the bounds. A set of such potentials can be chosen using the fact that $uI - A$ or $A - lI$ will have infinitely small eigenvalues when the eigenvalues of A approach u or l respectively; therefore, their inverse will be ill-conditioned and have infinitely large eigenvalues. We can use the following potential functions:

$$\Phi_l^u(A) = \Phi^u(A) + \Phi_l(A) = Tr[(uI - A)^{-1}] + Tr[(A - lI)^{-1}]$$

In summary, the main idea is to choose v_i and s_i such that the potential for the matrix $A^{(i)}$ in the next iteration does not explode. To do so, we ensure that the potentials remain monotonically decreasing:

$$\infty \gg \Phi^{u^{(0)}}(A^{(0)}) \geq \Phi^{u^{(1)}}(A^{(1)}) \geq ... \geq \Phi^{u^{(nd)}}(A^{(nd)})$$
$$\infty \gg \Phi_{\ell^{(0)}}(A^{(0)}) \geq \Phi_{\ell^{(1)}}(A^{(1)}) \geq ... \geq \Phi_{\ell^{(nd)}}(A^{(nd)})$$

With that in mind, let's assume we are going to assign s_k to any vector v_k such that after the increase in our upper and lower bound, the potential remains non-increasing. Now let us separately consider the upper and lower bound potentials.

When increasing $l^{(i)}$, the eigenvalues come closer to the lower bound, and hence, the potential of the lower bound will increase; therefore, for any vector v_k, the coefficient s_k should be bounded by some value $L_{A^{(i)}}(v_k)$ such that after adding $s_k \cdot v_k v_k^T$ to $A^{(i)}$, spectrum shifts forward and the increase in the potential cancels out. That said, for any matrix A and any vector v we have: