Fast Algorithms For HodgeRank Problems

An Honors Thesis for the Department of Mathematics

submitted by

Zian Jiang

Adviser: Xiaozhe Hu

Tufts University, May 2019.

Abstract

Least squares problems on graphs are a category of numerical linear algebra and optimization problems that are prevalent in ranking problems arising in real world networks. It has been an active research area to identify algorithms with good scalability. A notable work is from Colley et al [2], where they investigate the effectiveness of the unsmoothed aggregation algebraic multigrid (UA-AMG) and show that the algorithm is able to preserve the property that the convergence rate is not tied to the condition number of the matrix, and thus is more appropriate for graph-related problems. Also, they solve the least squares problems by solving their normal equations. In this work, we try to solve the least squares problem directly without forming the normal equation by using subspace correction method, multigrid method and sampling method to transform the original problem into a sequence of sub-problems in order to achieve better performance, providing numerical experiments comparing against other iterative methods on large random graphs and one set of real world networks to demonstrate the effectiveness of our algorithm for solving least squares problems on graphs.

Dedicated to my grandparents

Acknowledgements

Foremost, I would like to express my sincere gratitude to my advisor Prof. Xiaozhe Hu for his continuous support, patience, motivation, enthusiasm, and immense knowledge. His guidance helped me in all the time of research and writing of this thesis. Working with him has been a privilege.

Besides my advisor, I would like to thank the rest of my thesis committee: Prof. James Adler and Prof. James Murphy for their encouragement, insightful comments, and hard questions.

Last but not least, I would like to thank my family: my parents for giving birth to me at the first place and supporting me spiritually throughout my life. I would also like to thank Michelle for her love and support.

Contents

List of Tables

List of Figures

Fast Algorithms For HodgeRank Problems

Chapter 1

Introduction

In this work, we are interested in solving the least squares optimization problems involving matrices arising from graphs. One particular application of the least squares optimization problems on graphs is statistical ranking. Since graphs derived from real world networks can be huge, one particular point of interest is to build a fast and robust least squares solver that scales well with respect to the size of the graph. In this paper, we are interested in solving the least squares graph problem whose normal equation involves the graph Laplacian, denoted by L, a well-studied matrix representation of a graph. However, it is computationally expensive and less ideal to solve the normal equation $L\boldsymbol{x} = \boldsymbol{b}$ directly using traditional iterative methods. In this work, we adopt space decomposition method and randomized row sampling method in order to solve the least squares problem directly, without forming the normal equation.

1.1 Statistical Ranking (HodgeRank)

This paper is largely motivated by the HodgeRank method proposed by Jiang et al. [9] for solving ranking problems on graphs. HodgeRank method applies the graph Laplacian to generate statistical ranking on data that are translated into graphs and also estimates the inconsistency in the ranking. More importantly, HodgeRank method can be solved easily as a linear least squares regression problem, thus building the bridge between statistical ranking and linear least squares problems.

1.2 Existing Work

Linear least squares problems in the form of

$$\min_{\boldsymbol{x}} \|A\boldsymbol{x} - \boldsymbol{b}\|_2^2 \tag{1.1}$$

have been studied extensively since the 18th century. For example, singular value decomposition (SVD) offers stable solution to the least squares problem via a factorization of the matrix. However, it does not scale well and thus it is not feasible for large-scale problems. On the other hand, iterative methods such as conjugate gradient (CG) offer practical computational cost, but their convergence behaviors may be negatively affected by ill-conditioned problems. In the recent work of Colley et al. [2], unsmoothed aggregation AMG (UA-AMG) is applied on the graph to create a hierachical structure of the graph, by repetitively partitioning the vertices and condensing each partition of vertices into one single vertex on the coarse graph. This is done repetitively until the computational cost of solving the normal equation directly on the coarse graph is negligible, then CG is applied. In other words, UA-AMG serves as a preconditioner to CG. Another notable method is successive subspace correction method (SSC) by Tai et al. [15], which is a general convex optimization algorithm that decomposes the original problem into a number of smaller optimization problems. Given its special properties that will be seen in the context of the graph problems, this is also the method we will adopt to solve the least squares problems.

1.3 Contribution of This Work

The focus of this paper is to apply SSC using different space decompositions to solve the least squares problem directly without forming the normal equation. The advantage of not forming the normal equation lies in the fact that we do not need to form L, which can be a potentially dense matrix that is expensive to store and compute with.

We start from well-conditioned sparse graphs, where we conclude that standard space decomposition leads to scalable convergence behavior. Thus, standard space decomposition is also our default choice of algorithm for most well-conditioned sparse graphs. On ill-conditioned sparse graphs however, standard space decomposition converges poorly, and we have to construct a coarse grid to aid each iteration. A coarse grid can be viewed as a coarse representation of our original graph.

Even though the two algorithms above converge well respectively for sparse well-conditioned and ill-conditioned graphs, we notice that on dense graphs computation is still expensive. This motivates us to develop one uniform and one weighted sampling algorithm to randomly sample edges out of a dense graph in order to sparsify and further minimize computational cost. We also investigate whether precise global ranking will still be persevered after the sampling.

To demonstrate the effectiveness of our algorithms, we will show that our algorithms pertain to all type of graphs with different density or condition number, such that they scale well, as the number of iterations increases slightly as size of the graph problem increases. Particularly, we will see that for sparse well-conditioned graphs, standard space decomposition scales well. And for sparse ill-conditioned graphs, coarse grid has to be constructed to preserve scalability. For dense well-conditioned graphs such as real world networks, our sampling algorithms are able to not only minimize the computational cost by sampling edges out of the graph but also preserve the ranking accuracy to a reasonable degree. Also, we provide experiment results comparing our algorithms and other algorithms that solve the least squares problems. The testing graphs we use include random graphs, square grid graphs, and real world network graphs.

We hope that this work will help establish our algorithms as a set of robust tools that scales well for solving not only large-scale least squares problems on graphs but also graph Laplacian linear systems.

1.4 Outline of This Work

The rest of the paper is organized as followed: In Section 2, we will first introduce the method we use to set up the graph and edge flow from a given dataset, then some tools from algebraic topology that are essential for us to set up the least squares problem on graph. We will then explain the HodgeRank method, and how it connects statistical ranking and least squares optimization. In Section 3, we will look at SSC method and explain how we will adopt SSC method to solve the least squares graph problem. We will also look at 2 sampling methods that will help sparsify our potentially dense graph. In Section 4, we discuss the real and synthetic data used to set up the graph least squares problem for our numerical experiments. In Section 5 we provide detailed numerical simulations and results followed by analysis and directions for future work in Section 6 and 7.

Chapter 2

HodgeRank and Least Squares on Graphs

Consider the case that we have collected a dataset of votes from voters who have rated a set of alternatives, and we want to form a global ranking that determines the importance of these user-rated objects. HodgeRank provides a systematic way of generating the global ranking based on a graph generated from the dataset. We will first present some notations and how to form the graph from the dataset.

2.1 Notations

Let V be a finite set. Denote $|V|$ as the cardinality of set V. The following notation is from combinatorics:

$$\binom{V}{k} \coloneqq \text{the set of all k-element subset of V.}$$

Also suppose we have a set of m voters, denoted by $\Lambda = \{1, 2, ..., m\}$, and a set of n alternatives to be rated, $V = \{1, 2, ..., n\}$. Now, consider one single voter $\alpha \in \Lambda$, we will investigate how to build a graph for one voter first, and then for all voters, combining m graphs together. Also, for a matrix A, both $A(i, j)$ and A_{ij} denote the entry on the ith row and jth column of A.

2.1.1 One Voter

For voter $\alpha \in \Lambda$, let $V_\alpha \subset V$ be the set of alternatives that this voter has voted on. Define pairwise comparison function $f^\alpha \in \mathbb{R}^{|V_\alpha| \times |V_\alpha|}$, where

$$f^\alpha(i, j) \coloneqq \text{voter } \alpha\text{'s preference between object } i \text{ and object } j \quad \text{for} \quad i, j \in V_\alpha$$

Particularly, given the nature of our data, we will measure voter's preference in the following way:

$$\text{Ratings Difference: } f^{\alpha}(i,j) = R(\alpha,i) - R(\alpha,j), \tag{2.1}$$

which is the subtraction between two cardinal scores.

If we use V_α as the set of vertices in the graph for voter α, denoted by G_α, then f^α can be regarded as the set of edges in the form of edge flows. Thus, we can create graph $G_\alpha = (V_\alpha, E_\alpha)$. Notice that this is a complete graph with $|V_\alpha|$ vertices and $\binom{|V_\alpha|}{2}$ edges.

2.1.2 All Voters

Given G_α, now we consider how to combine these graphs together to form $G = (V, E)$. Intuitively, V should include all the alternatives that each vote has voted on. In other words, $V = \underset{\alpha\in\Lambda}{\cup} V_\alpha$ and $E = \underset{\alpha\in\Lambda}{\cup} E_\alpha$. For edge flow f, we define

$$f(i,j) = \frac{1}{|\Lambda|}\sum_{\alpha} f^{\alpha}(i,j) = \frac{1}{|\Lambda_{ij}|}\sum_{\alpha\in\Lambda_{ij}} f^{\alpha}(i,j), \tag{2.2}$$

where Λ_{ij} is subset of voters who have particularly voted between alternatives i and j.

In other words, $\boldsymbol{f}$ takes the average of all the votes between each pair of alternatives. However, it is easy to see that if we combine all the edges together in this way, these edges have different levels of importance and thus extra consideration is needed. For example, suppose $i, j \in V$ are popular alternatives such that a lot of the voters have voted on them, while only one voter has voted for $k, l \in V$. Thus, $e = (i,j) \in E$ is more important than $e' = (k,l) \in E$ because e represents more votes; we need to introduce a "weight" to the edges in order to account for this difference in popularity.

For all $e = (i,j) \in E$, define $\omega_e = |\Lambda_{ij}|$. Thus, if more voters have voted between i and j, that particular edge will become more important in our graph. Formally, we

	A	B	C
α	5	4	3
β	3	5	-

Table 2.1: Ratings from voters α, β to alternatives A,B,C

form the weighted graph $G = (V, E, \omega)$.

Notice that the introduction of ω has already signaled the imbalanced nature of our data. Given any dataset of user-rated objects, there will always likely be some unpopular alternatives that are rarely voted on and others that are more frequently voted on (consider two Hollywood movies and two low-budget movies). We have to address this difference by creating a weighted graph.

2.2 A Vectorized Formulation

In our current setup, f is symmetric because $f^{\alpha}(i,j) = R(\alpha,i) - R(\alpha,j) = -(R(\alpha,j) - R(\alpha,i)) = -f^{\alpha}(j,i)$. Also, the edge flow f is a matrix that contains information of $|\Lambda|$ separate graphs. Hirani et al. [8] proposes a vectorized formulation $\boldsymbol{f} \in \mathbb{R}^{|E|}$ that replaces f.

First, he introduces source vertices and sink vertices, which are ideas from flow network theory and graph theory. On the global graph G, for every edge $e = (i,j) \in E$, we will randomly assign these two vertices as source vertex and sink vertex respectively. Then, the new concise edge flow $\boldsymbol{f} \in \mathbb{R}^{|E|}$ will be constructed from the old edge flow f with $\boldsymbol{f}_e = f(i,j), \quad e = (i,j) \in E$. $f(j,i)$ can be neglected given the symmetry of f. Notice the importance of the change in dimensions. While condensing the graphs together, once we specify sink and source vertices in this manner, we are able to construct a vector edge flow that represents the global graph with $|E|$ number of edges. More importantly, the weight ω and $\boldsymbol{f}$ now belongs to the same dimension $\mathbb{R}^{|E|}$. They can be indexed accordingly so that each ω_e represents the weight of the corresponding edge flow $\boldsymbol{f}_e$. From now on, we will use $\boldsymbol{f} \in \mathbb{R}^{|E|}$ instead for further computation. Figure 2.1 is an example of constructing $\boldsymbol{f}$ for two voters and three alternatives given their votings in Table 2.1.

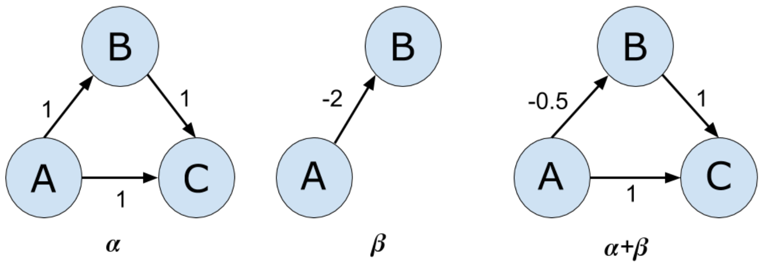

Figure 2.1: Edge flow for two voters individually and together given data in Table 2.1

2.3 Tools From Topology

To further understand ranking problem on graph, notations and tools from algebraic topology and graph theory need to be introduced.

Denote a set of triangles in graph by T. A vertex, an edge, and a triangle are called $0-$,$1-$, and $2-$simplex respectively on a graph. Recall that in Section 2.2 we introduced source vertices and sink vertices. This can be viewed as an **orientation** on the 1-simplex, where **orientation** is from source vertex to sink vertex. On the 0-simplex, an **orientation** is similar to a permutation of the vertices. On the 2-simplex, an **orientation** could be either counter-clockwise or clockwise. Let $C_i, i = 0, 1, 2$ be the space of real-valued functions on the oriented simplices of G, the i-th *boundary operator* $\partial_i : C_i \to C_{i-1}$ is the matrix with entries 0 or ±1. For the ranking problem we particularly consider the first boundary operator $\partial_1 : \mathbb{R}^{|E|} \to \mathbb{R}^{|V|}$ defined as following:

$$(\partial_1)_{ij} = \begin{cases} -1, & v_i \text{ is the source vertex in } e_j \\ +1, & v_i \text{ is the sink vertex in } e_j \\ 0, & \text{else} \end{cases}$$

Also, the second boundary operator is $\partial_2 : \mathbb{R}^{|T|} \to \mathbb{R}^{|E|}$ defined as following:

$$(\partial_2)_{ij} = \begin{cases} -1, e_i & \text{has same orietation as } T_j \\ +1, e_i & \text{has different orietation as } T_j \\ 0, & \text{else} \end{cases}$$

2.4 Hodge Decomposition

The Helmholtz decomposition of vector fields in Euclidean space states that every vector field on a compact simply connected domain can be decomposed into a gradient of a scalar potential and a curl of a vector potential. Hodge decomposition generalizes this idea to differential forms on manifolds. More importantly, this decomposition is orthogonal and thus unique, where the first part is curl-free and second-part is divergence-free. We refer to [8] for details and possible generalizations.

Now, consider the matrices ∂_1, ∂_2, $\triangle := \partial_1^T \partial_1 + \partial_2 \partial_2^T$ and vector space $\mathbb{R}^{|E|}$. According to Hodge decomposition, there exists a unique orthogonal decomposition of $\mathbb{R}^{|E|}$ such that

$$\mathbb{R}^{|E|} = \operatorname{im}(\partial_1^T) \oplus \ker(\triangle) \oplus \operatorname{im}(\partial_2^T), \tag{2.3}$$

where $\operatorname{im}(A)$ and $\ker(A)$ denotes the image and kernel of matrix A respectively. Furthermore, for any $\boldsymbol{f} \in \mathbb{R}^{|E|}$, there exists $\boldsymbol{r} \in \mathbb{R}^{|V|}$, $\boldsymbol{c} \in \mathbb{R}^{|T|}$, and $\boldsymbol{h} \in \ker(\triangle)$ such that

$$\boldsymbol{f} = \partial_1^T \boldsymbol{r} + \partial_2 \boldsymbol{c} + \boldsymbol{h}. \tag{2.4}$$

2.5 Least Squares Formulation

The least squares problems we consider here are related to the task of finding the optimal $\boldsymbol{r_*}$ and $\boldsymbol{c_*}$ in the Hodge decomposition of an edge flow $\boldsymbol{f} \in \mathbb{R}^{|E|}$ defined in equation (2.4). They are defined as the two following weighted least squares problems, where $W \in \mathbb{R}^{|E| \times |E|}$ is the diagonal matrix with each diagonal entry corresponds to the weight of the edge with the same index,

$$\min_{\boldsymbol{r}\in\mathbb{R}^{|V|}} \|\boldsymbol{f} - \partial_1^T \boldsymbol{r}\|_W^2 = \min_{\boldsymbol{r}\in\mathbb{R}^{|V|}} \|W^{\frac{1}{2}}\boldsymbol{f} - W^{\frac{1}{2}}\partial_1^T\boldsymbol{r}\|_2^2, \tag{2.5}$$

$$\min_{\boldsymbol{c}\in\mathbb{R}^{|E|}} \|\boldsymbol{f} - \partial_2\boldsymbol{c}\|_W^2. \tag{2.6}$$

which are respectively equivalent to the normal equations

$$\partial_1 W \partial_1^T \boldsymbol{r} = \partial_1 W \boldsymbol{f}, \tag{2.7}$$

$$\partial_2^T W \partial_2 \boldsymbol{c} = \partial_2^T W \boldsymbol{f}. \tag{2.8}$$

The focus of this work is solving equation (2.5) for the following reasons. First, the left-hand side of the normal equation of equation (2.5), namely the matrix in equation (2.7), is the graph Laplacian defined as $L = \partial_1 W \partial_1^T$. This is a well-known definition of the graph Laplacian besides $L = D - A$. More importantly, the solution $\boldsymbol{r}_*$ in equation (2.5) is essential to the task of finding the global ranking of all vertices, according to HodgeRank method. This will be explained in greater details once we construct the least squares framework in the context of HodgeRank in the next subsection.

2.6 Hodge Decomposition in Ranking Problems

HodgeRank, as its name suggests, applies Hodge decomposition in the setting of statistical ranking problems. Suppose we have a dataset and graph defined in Section 2.1.2, HodgeRank method solves the least squares equations (2.5) and (2.6) in order to find the unique Hodge decomposition of edge flow $\boldsymbol{f}$, namely to find $\boldsymbol{r}_* \in \mathbb{R}^{|V|}$, $\boldsymbol{c}_* \in \mathbb{R}^{|E|}$, and $\boldsymbol{h}_* \in \ker(\triangle)$. More importantly, Jiang et al. [9] proposed that $\boldsymbol{r}_*$ is acyclic and thus can be used in the task of ranking the vertices. Also, $\boldsymbol{c}_*$ measures the local inconsistency and $\boldsymbol{h}_*$ measures the global consistency of the data. In other words, if the residual term of equation (2.5), $\|\boldsymbol{f} - \partial_1^T \boldsymbol{r}_*\|_W^2$ is large, our data have serious inconsistencies and the global ranking we have obtained is of

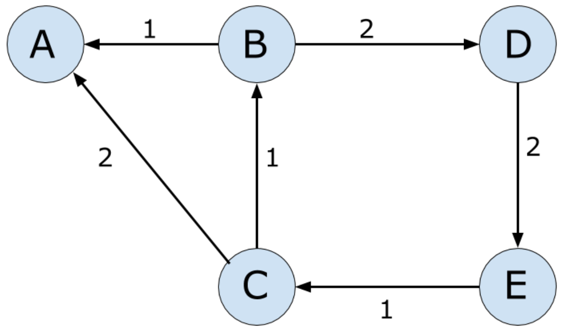

Figure 2.2: An example graph where inconsistency is big

low quality. Otherwise, our dataset is reasonably consistent and a global ranking over a set of alternatives makes sense. For example, for Figure 2.2, we would expect residual $\|\boldsymbol{f} - \partial_1^T \boldsymbol{r}_*\|_W^2$ to be large, because of the existing inconsistency. To see this, on Figure 2.2, vertices B, D, C, E form a cycle such that $B > D > E > C > B$, which contributes to the large inconsistency on this graph, while the triangle formed by A, B, C is well-consistent, as $C > B > A$ and $1 + 1 - 2 = 0$.

2.7 Existing Methods for Least Squares Problems

In Section 1.2 we briefly introduced the existing least squares solvers. In this section we will discuss them in further details. It turns out that their convergence behaviors depend strongly on the condition number of the problem. Here, we will first introduce the condition number of a graph.

2.7.1 Condition Number of Graph

It is a well-known property in the area of spectral graph theory that graph Laplacian L for a simple graph G with n vertices has eigenvalues

$$0 = \lambda_1(G) \leq \lambda_2(G) \leq \ldots \leq \lambda_n(G).$$

Moreover, $\lambda_2(G) > 0$ (the smallest non-zero eigenvalue) if and only if G is connected. This led Fiedler [6] to label $\lambda_2(G)$ the *algebraic connectivity* of a graph, and to conclude that better connected graphs have higher values of $\lambda_2(G)$. It has also been proved that there exists a lower bound of for $\lambda_2(G)$ on a connected graph. We refer to [12] and [14] for further details and generalizations. Here, we define the condition number of a graph Laplacian L to be

$$\mathbf{cond}(L) = \frac{\lambda_n}{\lambda_2}$$

and a high condition number will lead to an ill-conditioned problem, and vice versa.

2.7.2 Existing Solvers

Consider the general least squares problem

$$\min_{\boldsymbol{x} \in \mathbb{R}^n} \|A\boldsymbol{x} - \boldsymbol{b}\|_2^2. \tag{2.9}$$

2.7.2.1 Factorization Methods

One traditional approach is to solve equation (2.9) via a factorization of A. Singular Value Decomposition (SVD) factorizes $A \in \mathbb{R}^{m \times n}$ and there exist orthogonal matrices $U \in \mathbb{R}^{m \times m}$, $V \in \mathbb{R}^{n \times n}$ and diagonal matrix $\Sigma \in \mathbb{R}^{m \times n}$ such that $A = U\Sigma V$. Then using the orthogonality of U, V,

$$\|A\boldsymbol{x} - \boldsymbol{b}\|_2^2 = \|U^T(AVV^T\boldsymbol{x} - b)\|_2^2 = \|\Sigma V^T\boldsymbol{x} - U^T b\|_2^2 \tag{2.10}$$

, and (2.9) can be solved easily this way. Another approach is QR decomposition, which factorizes A into an orthogonal matrix Q and upper triangular matrix R, then it is well-studied that equation (2.9) can be solved easily by exploiting the orthogonality of Q and upper triangular structure of R. However, the biggest problem with these factorization methods is that though they offer stable solutions to linear least squares problems, the tasks of computing these factorizations do not scale well and thus they are not suitable for large-scale problems.

2.7.2.2 Conjugate Gradient Method for Least Squares (CGLS)

Given equation (2.9), we can solve its normal equation $A^T A\boldsymbol{x} = A^T\boldsymbol{b}$ using iterative methods. One of them is Conjugate Gradient (CG). Using CGLS to solve equation (2.9) is mathematically equivalent to using CG to solve its normal equation. However, it is well-known that the convergence rate of CG depends on the condition number of the matrix. Thus in applications usually the matrix is preconditioned first by a preconditioner in order to reduce the condition number. However in our problem this means designing a preconditioner for ∂_1, which is more difficult than preconditioning L.

2.7.2.3 Algebraic Multigrid Methods

Algebraic Multigrid (AMG) methods create a hierarchy of problems of decreasing sizes from the given problem. The larger system is called a fine grid and the smaller one is called a coarse grid. The solutions in the different hierarchies are related via prolongation. Coarsening is done via an aggregation of vertices that have a strong connection. Particularly, in smooth aggregation, a vertex can belong to several aggregates. In [8], Smoothed Aggregation Algebraic Multigrid (SA-AMG) is used where Hirani et al. concludes that AMG methods such as SA-AMG performs poorly on graph Laplacians. However, according to Colley et al. [2], the conclusion of [8] is due to that smooth aggregation destroys the graph structure and causes every L_{i+1} on the coarse level not to be a graph Laplacian anymore. Instead, Colley et al. [2] uses Unsmoothed Aggregation Algebraic Multigrid (UA-AMG) as a preconditioner to CG and it outperforms traditional iterative methods as a scalable algorithm, demonstrating UA-AMG methods as a class of effective scalable linear least squares solvers on graphs. For further details in algebraic multigrid, smoothed/unsmoothed aggregation, we refer to [8], [19] and [17].

Chapter 3

Algorithms

In this chapter, we will mainly discuss two types of algorithms that we have developed, deterministic and randomized. The deterministic algorithms mainly involve SSC with different space decomposition methods, while the randomized algorithms will solve from a randomly sampled graph.

3.1 Deterministic Method

We will first look at the deterministic algorithms, which are the most useful for sparse graphs. Though they will also work on dense graphs, they are not the computationally cheapest algorithms for solving this problem. In this section we will develop algorithms using Successive Subspace Correction (SSC) but with different setups in order to pertain to all types of graphs.

SSC is an iterative domain decomposition method that reduces the original minimization problem into a number of smaller minimization problems and solves them successively to achieve a monotone decreasing of the original objective function.

Suppose we have objective function $F : \mathbb{R}^n \to \mathbb{R}$ and we want to solve $\min\limits_{\boldsymbol{v}\in\mathbb{R}^n} F(\boldsymbol{v})$. Given solution space $V = \mathbb{R}^n$, decompose V into a sum of closed subspaces such that there are closed subspaces

$$V_i \subset V, i = 1, 2, ..., m \quad \text{and} \quad V = V_1 + V_2 + ... + V_m.$$

Also, for any $\boldsymbol{v} \in V$,

$$\exists \boldsymbol{v_i} \in V_i, i = 1, 2, ..., m \quad \text{such that} \quad \boldsymbol{v} = \sum_{i=1}^{m} \boldsymbol{v_i}.$$

Also, let P_i be the matrix representation of V_i such that range$(P_i) = V_i$.

Now, suppose we have least squares objective function $F(\boldsymbol{x}) = \frac{1}{2}\|A\boldsymbol{x} - \boldsymbol{b}\|_2^2$, then

Algorithm 1 General Successive Subspace Correction Algorithm [15]

```
1: choose initial guess $\boldsymbol{v^0} \in V$;
2: for every $n \geq 0$ do
3:   if $\boldsymbol{v^n} \in V$ is defined then
4:     for $i = 1 \to m$ do
5:       find $\boldsymbol{e_i^n} \in V_i$ such that $F(\boldsymbol{v^{n+(i-1)/m}} + \boldsymbol{e_i^n}) \leq F(\boldsymbol{v^{n+(i-1)/m}} + \boldsymbol{u_i})$, $\forall \boldsymbol{u_i} \in V_i$;
6:       $\boldsymbol{v^{n+i/m}} \leftarrow \boldsymbol{v^{n+(i-1)/m}} + \boldsymbol{e_i^n}$;
7:     end for
8:   end if
9: end for
```

we will be able to construct the Successive Subspace Correction Algorithm for Least Squares. Also, after some simply matrix algebra, it is easy to find out that for a

Algorithm 2 Successive Subspace Correction Algorithm for Least Squares

```
1: choose initial guess $\boldsymbol{x^0} \in V$;
2: for every $n \geq 0$ do
3:   if $\boldsymbol{x^n} \in V$ is defined then
4:     for $i = 1 \to m$ do
5:       $\boldsymbol{x^{n+i/m}} \leftarrow \boldsymbol{x^{n+(i-1)/m}} + \frac{\langle \boldsymbol{b} - A\boldsymbol{x^{n+(i-1)/m}}, AP_i \rangle}{\langle AP_i, AP_i \rangle} P_i$;
6:     end for
7:   end if
8: end for
```

least squares objective function $F(\boldsymbol{x}) = \frac{1}{2}\|A\boldsymbol{x} - \boldsymbol{b}\|_2^2$, the minimizer for each subspace V_i defined in 2 is $\boldsymbol{e_i^n} = \frac{\langle b - A\boldsymbol{x^{n+(i-1)/m}}, AP_i \rangle}{\langle AP_i, AP_i \rangle} P_i$, where $\langle \ , \ \rangle$ defines an inner product.

3.1.0.1 Convergence of SSC

Assume there exist constants $K, L > 0$ $p \geq q > 1$ such that

$$\|F'(w) - F'(v)\|_{V'} \leq L\|w - v\|_V^{q-1} \quad \text{and} \quad \langle F'(w) - F'(v), w - v \rangle \geq K\|w - v\|_V^p \quad \forall w, v \in V. \tag{3.1}$$

Notice that the least squares objective function satisfies (3.1). Let C_1 be the least constant satisfying the following property: for any $v \in V$, there exists $v_i \in V_i$ such that

$$v = \sum_{i=1}^{m} v_i \quad \text{and} \quad (\sum_{i=1}^{m} \|v_i\|_V^{\sigma})^{\frac{1}{\sigma}} \leq C_1 \|v\|_V \tag{3.2}$$

and let C_2 be the least constant satisfying the following property: for any $w_{ij} \in V, u_i \in V_i$ and $v_j \in V_j$ the following inequality holds:

$$\sum_{i,j=1}^{m} \langle F'(w_{ij} + u_i) - F'(w_{ij}), v_j \rangle \leq C_2 (\sum_{i=1}^{m} \|u_i\|_V^p)^{\frac{q-1}{p}} (\sum_{j=1}^{m} \|v_j\|_V^\sigma)^{\frac{1}{\sigma}}. \tag{3.3}$$

If (3.1),(3.2) and (3.3) are all satisfied, then according to the convergence analysis in [15], algorithm 1 achieves geometric convergence if $p = q$.

3.1.1 Vertex-based Decomposition

Notice that in the proposed SSC algorithm for least squares problems, there are both an outer loop, and an inner loop. For the canonical decomposition of the solution space, $V = \text{span}\{\boldsymbol{e_i}\}_{i=1}^m$, in one iteration of the inner loop the algorithm will go through every $\boldsymbol{e_i}$. However, going through $\text{span}\{\boldsymbol{e_i}\}_{i=1}^m$ is exactly one iteration of Gauss-Seidel method on the normal equation $A^T A\boldsymbol{x} = A^T\boldsymbol{b}$. In equation (2.5), $A^T A = L$ and $A^T\boldsymbol{b} = \partial_1 W^{\frac{1}{2}}\boldsymbol{f}$. The connection can be easily checked via some matrix algebra.

From a implementation-wise standpoint, if $V = \text{span}\{\boldsymbol{e_i}\}_{i=1}^m$, for every outer iteration k, we are able to substitute the inner loop of m iterations with only one iteration of Gauss-Seidel (L_* denotes the lower triangular matrix of L):

$$\boldsymbol{x^{k+1}} = \boldsymbol{x^k} + L_*^{-1}(A^T\boldsymbol{b} - L\boldsymbol{x^k}), \tag{3.4}$$

thus making the implementation simpler. Here, we list the full algorithm for reference. Note that this is only valid for canonical space decomposition $V = \text{span}\{\boldsymbol{e_i}\}_{i=1}^m$.

Extra insight may be gained by looking at the space decomposition from the perspective of a graph problem: the canonical decomposition of our solution space, $V = \mathbb{R}^n = \text{span}\{\boldsymbol{e_i}\}_{i=1}^m$, represents m vertices in our graph, where each $\boldsymbol{e_i}$ represents one vertex. By solving the sub-problem in the direction of $\boldsymbol{e_i}$, we are adding more information to the corresponding vertex v_i. Thus, in our graph problem, space decomposition is done in a vertex-based way, and in the canonical space decomposition, each vertex corresponds to one subspace.

Algorithm 3 Modified Successive Subspace Correction Algorithm for Least Squares with Canonical Space Decomposition

```
1: choose initial guess $\boldsymbol{x^0} \in V$;
2: for every $n \geq 0$ do
3:     $\boldsymbol{x^{n+1}} \leftarrow \boldsymbol{x^n} + L_*^{-1}(A^T\boldsymbol{b} - L\boldsymbol{x^n})$;
4: end for
```

3.1.2 Alternative Space Decomposition Using Graph Matching

Recall in Section 3.1.1 that while using the canonical basis $\{\boldsymbol{e_i}\}_{i=1}^m$ for space decomposition, we are essentially solving each sub-problem in each basis in order from $\boldsymbol{e_1}$ to $\boldsymbol{e_m}$, which respectively respresent vertices v_1 to v_m. Though this canonical space decomposition grants us the implementation-wise advantage of using Gauss-Seidel on the normal equation to skip the inner loop, it is far from the only available method of decomposing the solution space.

It follows naturally that we can also take account of the relationships between vertices, namely edges, when decomposing the solution space. One approach is to find a greedy graph matching of the graph, where we treat the vertices at the ends of every edge, v_i and v_j, as one sub-problem in span$\{\boldsymbol{e_i}, \boldsymbol{e_j}\}$. Let $V_{ij} = \text{span}\{\boldsymbol{e_i}, \boldsymbol{e_j}\}$, find $\boldsymbol{\alpha}_* \in \mathbb{R}^2$ such that it will optimize our solution in subspace V_{ij}, which is spanned by two canonical subspaces. V_{ij} in matrix form can be represented by $P_{ij} = [\boldsymbol{e_i}, \boldsymbol{e_j}]$ so that range$(P_{ij}) = V_{ij} = \text{span}\{\boldsymbol{e_i}, \boldsymbol{e_j}\}$.

Intuitively graph matching poses as a better space decomposition than span$\{\boldsymbol{e_i}\}_{i=1}^m$, because it takes advantage of the inner-structure of the graph by looking at vertices in terms of the edges they are connected to, as opposed to individually. For Figure 3.1 (left), according to the graph matching in red, we will construct P_{15}, P_{26} and

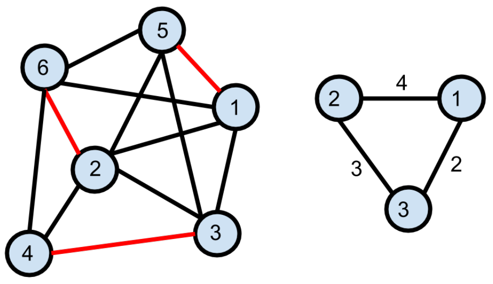

Figure 3.1: Left: example of a greedy graph matching with aggregates $\{1,5\},\{2,6\},\{3,4\}$. Right: coarse graph obtained by condensing each aggregate into one vertex and edges

P_{34} as follows:

$$P_{15} = \begin{pmatrix} 1 & 0 \\ 0 & 0 \\ 0 & 0 \\ 0 & 0 \\ 0 & 1 \\ 0 & 0 \end{pmatrix}, \quad P_{26} = \begin{pmatrix} 0 & 0 \\ 1 & 0 \\ 0 & 0 \\ 0 & 0 \\ 0 & 0 \\ 0 & 1 \end{pmatrix}, \quad P_{34} = \begin{pmatrix} 0 & 0 \\ 0 & 0 \\ 1 & 0 \\ 0 & 1 \\ 0 & 0 \\ 0 & 0 \end{pmatrix} \tag{3.5}$$

In general, graph matching can be done in different ways. Since in our setup each subspace V_{ij} is spanned by 2 vectors that respectively represent vertices v_i, v_j, we would want the matching to include important edges but without common vertices. The importance of edges can be interpreted in terms of weights. Thus, in our effort to decompose the solution space using graph matching, we would first sort the weights in a descending order, then repeatedly pick the edges with the largest weights into the matching set until a **maximal matching** is reached (every vertex is picked). This can be implemented as a simple greedy algorithm on graph matching. Here we list it in Algorithm 4 for reference.

Algorithm 4 Greedy Graph Matching Algorithm for Space Decomposition

```
1: $V, E, \omega \leftarrow G$;
2: sort $\omega$ in a descending order;
3: for $i = 1 \rightarrow |E|$ do
4:     if a maximal matching is not reached then
5:         $v_{i_m}, v_{i_n} \leftarrow \omega_i$
6:         $P_{i_m i_n} \leftarrow [\boldsymbol{e_{i_m}}, \boldsymbol{e_{i_n}}]$
7:     end if
8: end for
```

Algorithm 5 Successive Subspace Correction Algorithm for Least Squares with Graph Matching

```
1: $P_1, P_2, ..., P_m \leftarrow$ GraphMatching($G$);
2: choose initial guess $\boldsymbol{x^0} \in V$;
3: for every $n \geq 0$ do
4:     if $\boldsymbol{x^n} \in V$ is defined then
5:         for $i = 1 \rightarrow m$ do
6:             $\boldsymbol{\alpha_*^n} \leftarrow \frac{\langle AP_i, b - Ax^n \rangle}{\langle AP_i, AP_i \rangle}$
7:             $\boldsymbol{x^{n+i/m}} \leftarrow \boldsymbol{x^{n+(i-1)/m}} + \boldsymbol{\alpha_*^n} P_i$;
8:         end for
9:     end if
10: end for
```

3.1.3 Coarse Grid For Ill-conditioned Problems

In this section, we introduce another use of graph matching such that we are able to aid span$\{\boldsymbol{e_i}\}_{i=1}^m$ by a coarse grid, instead of directly using graph matching as the space decomposition. The reason for constructing a coarse grid is due to the condition number of the graph. It turns out that coarse grid is especially important for ill-conditioned problems. We will later see that, for an ill-conditioned problem, SSC can only scale well with respect to the size of the problem if a coarse grid is added. In other words, Algorithm 3 and Algorithm 5 will not converge nicely for ill-conditioned problems. Thus, in the next subsection we will discuss how to construct the coarse grid given graph G.

3.1.3.1 Coarse Grid Construction

We want to create a coarse grid V_c, a coarse version of the graph, to aid Algorithm 3. Let P be a matrix representation of V_c. To create V_c, we first condense the two associated vertices of every matching edge into one aggregated vertex, so that for a graph G with matching number $v(G)$, there will be $v(G)$ aggregated vertices $\{V^i\}$ such that

$$V = \bigcup_{j-1}^{v(G)} V^j \quad \text{and} \quad V^i \cap V^j = \varnothing, \quad \text{if} \quad i \neq j.$$

Recall that when using graph matching as the space decomposition, we let $P_{ij} = [\boldsymbol{e_i}, \boldsymbol{e_j}]$, which is a matrix concatenated by two vectors. Here we let $\boldsymbol{P_{ij}} = \boldsymbol{e_i} + \boldsymbol{e_j} \in \mathbb{R}^{|V|}$, which is a vector with two nonzero entries. Then, P is created by concatenating all P_{ij} together. Generally, we construct P as follows:

$$P_{ij} = \begin{cases} 1, & \text{if vertex } v_i \in V \text{ is in aggregate } V^j, \\ 0, & \text{otherwise.} \end{cases}$$

For example, let L_f denote the graph Laplacian corresponding to the graph shown in Figure 3.1 (left).

$$L_f = \begin{pmatrix} 4 & -1 & -1 & 0 & -1 & -1 \\ -1 & 5 & -1 & -1 & -1 & -1 \\ -1 & -1 & 4 & -1 & -1 & 0 \\ 0 & -1 & -1 & 3 & 0 & -1 \\ -1 & -1 & -1 & 0 & 4 & -1 \\ -1 & -1 & 0 & -1 & -1 & 4 \end{pmatrix} \tag{3.6}$$

One possible set of aggregates are given in Figure 3.1 (left) so we define P as the matrix representation of V_c,

$$P = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \\ 0 & 0 & 1 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \end{pmatrix} \tag{3.7}$$

We can then accordingly calculate the graph Laplacian of the coarse graph,

$$L_c = P^T L_f P = \begin{pmatrix} 6 & -4 & -2 \\ -4 & 7 & -3 \\ -2 & -3 & 5 \end{pmatrix} \tag{3.8}$$

However, L_c is exactly the graph Laplacian of Figure 3.1 (right), obtained by condensing each aggregate into one vertex. Here, we can see the connection between aggregates and the graph structure are preserved. Thus, we are able to maintain the graph structure on a coarse level, which aids the computation by solving on the coarse graph and adding that to the original solution.

Now, we have the algorithm for SSC with coarse grid construction. Notice that we can even construct coarse grid from an existing coarse grid, in other words, constructing coarse grid recursively to build a multi-leveled linear systems. This idea corresponds to that of UA-AMG, because here we are essentially constructing matrix P using unsmoothed aggregation and P is called the prolongation matrix in the AMG setting. Thus, in this perspective Algorithm 6 can be regarded as a two-level UA-AMG solver that solves the least squares equation (2.5) instead of its normal equation (2.7). After similar matrix algebra done in the derivation of Algorithm 2, the minimizer for coarse grid V_c defined in Algorithm 6 is $\boldsymbol{e}^{\boldsymbol{n}}_{\boldsymbol{V_c}} = \frac{\langle b - A\boldsymbol{x}^{\boldsymbol{n}}, AP\rangle}{\langle AP, AP\rangle} P$.

Algorithm 6 Modified Successive Subspace Correction Algorithm for Least Squares with Canonical Space Decomposition and Coarse Grid

```
1: choose initial guess $\boldsymbol{x}^{\boldsymbol{0}} \in V$;
2: $P \leftarrow$ CreateCoarseGrid$(G)$;
3: for every $n \geq 0$ do
4:     $\boldsymbol{x}^{\boldsymbol{n+\frac{1}{2}}} \leftarrow \boldsymbol{x}^{\boldsymbol{n}} + L_*^{-1}(A^T\boldsymbol{b} - L\boldsymbol{x}^{\boldsymbol{n}})$;
5:     $\boldsymbol{x}^{\boldsymbol{n+1}} \leftarrow \boldsymbol{x}^{\boldsymbol{n+\frac{1}{2}}} + \frac{\langle b - A\boldsymbol{x}^{\boldsymbol{n+\frac{1}{2}}}, AP\rangle}{\langle AP, AP\rangle} P$;
6: end for
```

So far, we have discussed all of the deterministic algorithms. They are SSC with different space decomposition methods and setups. Algorithm 3 is SSC with canonical space decomposition. Algorithm 5 is SSC with space decomposition according to graph matching. Algorithm 6 is SSC with canonical space decomposition. Next, we will discuss the randomized algorithms.

3.2 Random Sampling

In this problem, for a graph Laplcian with n vertices, we notice that a sparse graph usually has number of edges $m = O(n)$, and a dense graph will have $m = O(n^\alpha)$ for $1 < \alpha \leq 2$. Real world networks thus usually fall into the category of well-conditioned dense graphs. For a real network network graph with $m = O(n^\alpha)$, its graph Laplacian will have few nonzero entries and thus becomes a highly dense matrix to store and compute with. This fact at first motivates us to not solve the normal equation $L\boldsymbol{x} = \boldsymbol{b}$ directly but to solve the least squares optimization problem directly. However, we also notice that even for Algorithm 3, in every iteration of Gauss-Seidel, equation (3.4) still involves the lower triangular matrix of L. To avoid forming L, another option is to approximate L with a less dense matrix $\tilde{L}$. In this section, on well-conditioned dense graph of $O(n^\alpha), 1 < \alpha \leq 2$, we will first apply a famous sampling algorithm for computing an approximate solution to a set of linear equations defined by a Laplacian left-hand side, namely $L\boldsymbol{x} = \boldsymbol{b}$, and the ways we can modify this algorithm to solve equation (2.5). It turns out that in this algorithm, approximating L by $\tilde{L}$ is equivalent to randomly sampling edges out of the graph to form a sparse graph with graph Laplacian $\tilde{L}$. Note that this is especially important for a dense graph G. Since sampled graph $\tilde{G}$ is sparser than G, computations on $\tilde{G}$ will be much cheaper than on G. Our goal here is to modify this algorithm so that solving on $\tilde{G}$ will result in an accurate ranking of the vertices on the orginal graph G.

3.2.1 Row Sampling

Proposed in the works of Mahoney et al. [4], for graph Laplacian L associated with graph G, L can be approximated by $\tilde{L}$, which is the graph Laplacian associated with graph $\tilde{G}$. $\tilde{G}$ is a sparsified version of G, constructed by randomly sampling edges out of G with probability proportional to the “importance" of each edge. The idea of row sampling can be generalized to a row/column sampling algorithm that can be used to approximate large-scale matrix multiplication [3].

Notice that each edge of G corresponds to a row in the edge-incidence matrix ∂_1^T, where each row of ∂_1^T has only two non-zero entries. Since edges are weighted, the norm of each row of $A = W^{\frac{1}{2}}\partial_1^T$ shows the importance of each edge: the larger the edge weight, the larger the norm of the row corresponding to that edge. Thus, we can randomly sample edges of graph G by randomly sampling the rows of A, with probability proportional to the norm of each row of A. In this sampling method, it is convenient to construct the probability distribution, because the sum of row norm of A equals to the square of the Frobenius norm of A, $\|A\|_F^2$. Thus, the idea of this sampling algorithm is to randomly sample s rows from A to form A_s, and then construct $\tilde{L} = A_s^T A_s$.

Algorithm 7 Weighted Row Sampling Algorithm [4]

```
1: for $i = 1 \to |E|$ do
2:     $p_i \leftarrow \frac{\|A_k\|_2^2}{\|A\|_F^2}$;
3: end for
4: for $t = 1 \to s$ do
5:     pick $i_t \in \{1, 2, ..., |E|\}$ with probability $P_r\{i_t = k\} = p_k$ in identical and inde-
       pendent distributed (i.i.d.) trials;
6: end for
7: define $S \in \mathbb{R}^{s\times|V|}$ with $S_{t,i_t} \leftarrow \frac{1}{\sqrt{sp_{i_t}}}$;
8: $A_s = SA$;
```

According to the convergence analysis of the algorithm in [1], for matrix A of size $m \times n$ with $m > n$ and given $\epsilon \in (0,1), \delta \in (0,1)$, let $C = \frac{2}{3}(6\|A\|_2^2 + 2\epsilon)(1 - \log_n(\delta/2))$ and $s = C\epsilon^{-2} n \log n$. Let A_s be the sampled matrix obtained by Algorithm 7 with sample size s, then

$$\|A_s^T A_s - A^T A\|_2 \leq \epsilon \quad \text{with probability at least} \quad 1 - \epsilon.$$

Also, [1] mentions that in practice $s = 4n \log n$ is a good choice for a reasonably good sampling matrix. In other words, in order to use this sampling scheme, our graph needs to have at least $4n \log(n)$ edges. For real world networks, this requirement is easy to fulfill, as the number of edges is way larger than that of vertices. This is also the reason that we will only use Algorithm 7 on well-conditioned dense graphs.

In Algorithm 7, we sample edges out of the graph according to their weights, which are calculated in the first loop that goes through every edge on the graph. Since we have a dense graph with number of edges as possibly many as $O(n^2)$, generating the probabilities of all edges can be an expensive task. This motivates us to assign the same probability to all edges and result in a uniform sampling algorithm. However, we will investigate later if uniform sampling preserves good accuracy compared to weighted sampling. Remark: the task of generating all probabilities may be $O(\log^2(n))$ if A is stored in the Binary search tree data structure (BST). For further topics on this subject, we refer to [10] and Section 4 of [16].

Algorithm 8 Uniform Row Sampling Algorithm

```
1: $p \leftarrow \frac{1}{|E|}$;
2: for $t = 1 \rightarrow s$ do
3:     pick $i_t \in \{1, 2, ..., |E|\}$ with probability $P_r\{i_t = k\} = p$ in identical and independent distributed (i.i.d.) trials;
4: end for
5: define $S \in \mathbb{R}^{s \times |V|}$ with $S_{t,i_t} \leftarrow \frac{1}{\sqrt{sp}}$;
6: $A_s = SA$;
```

3.2.2 Solvers on Sampled Graph

Given the sampling algorithm given in Section 3.2.1, we are able to form the sampled graph $\tilde{G}$. In this section, we will discuss two methods of solving on $\tilde{G}$.

3.2.2.1 Using SSC on Sampled Graph

Recall that in equation (3.4), we have to take the inverse of the lower triangular matrix L_*. One natural thought would be replacing L_* by the sampled $\tilde{L}_*$, or creating a coarse grid according to $\tilde{G}$ instead of G. By using this scheme, we will modify Algorithm 3 and 6 respectively to get Algorithm 9 and 10 described below.

Algorithm 9 Modified Successive Subspace Correction Algorithm for Least Squares with Canonical Space Decomposition and Sampling

```
1: $As \leftarrow$ WeightedRowSampling($G$);
2: $\tilde{L} \leftarrow As^T As$;
3: choose initial guess $\boldsymbol{x^0} \in V$;
4: for every $n \geq 0$ do
5:     $\boldsymbol{x^{n+1}} \leftarrow \boldsymbol{x^n} + \tilde{L}_*^{-1}(A^T\boldsymbol{b} - L\boldsymbol{x^n})$;
6: end for
```

Algorithm 10 Modified Successive Subspace Correction Algorithm for Least Squares with Canonical Space Decomposition, Coarse Grid and Sampling

```
1: $As, \tilde{G} \leftarrow$ WeightedRowSampling($G$);
2: $P \leftarrow$ CreateCoarseGrid($\tilde{G}$);
3: choose initial guess $\boldsymbol{x^0} \in V$;
4: for every $n \geq 0$ do
5:     $\boldsymbol{x^{n+\frac{1}{2}}} \leftarrow \boldsymbol{x^n} + L_*^{-1}(A^T\boldsymbol{b} - L\boldsymbol{x^n})$;
6:     $\boldsymbol{x^{n+1}} \leftarrow \boldsymbol{x^{n+\frac{1}{2}}} + \frac{\langle b - A\boldsymbol{x^{n+\frac{1}{2}}}, AP\rangle}{\langle AP, AP\rangle} P$;
7: end for
```

3.2.2.2 Solving Normal Equation on Sampled Graph

Recall that directly solving the normal equation $L\boldsymbol{x} = \boldsymbol{b}$ was deemed too computational expensive. In the same paper where they propose Algorithm 7, Mahoney and et al. [4] also propose that we can simply solve $\tilde{L}\tilde{\boldsymbol{x}} = \boldsymbol{b}$ instead using direct methods such as CG and $\tilde{\boldsymbol{x}}_*$ will approximate $\boldsymbol{x}_*$ under quality-of-approximation

$$\|\boldsymbol{x}_* - \tilde{\boldsymbol{x}}_*\|_L \leq \epsilon \|\boldsymbol{x}_*\|_L, \tag{3.9}$$

for accuracy parameter $\epsilon \in (0, 1)$.

Algorithm 11 Solving Normal Equation on Sampled Graph [4]

```
1: $As \leftarrow$ WeightedRowSampling($G$);
2: $\tilde{L} \leftarrow As^T As$;
3: $\tilde{\boldsymbol{x}} \leftarrow \tilde{L}^{\dagger}\boldsymbol{b}$ via CG;
```

Remark: Given a well-conditioned dense graph Laplacian L, the weighted sample graph Laplacian $\tilde{L}$ will also be well-conditioned so that we can use CG in Algorithm 11. Its proof is given in Corollary 2 in [1]. As mentioned earlier in Section 2.7.2.2, convergence rate of CG will be negatively affected by an ill-conditioned graph Laplacian.

The apparent advantage of Algorithm 11 is that after sampling the graph, we ignore L totally and solve the normal equation with new left-hand side matrix $\tilde{L}$, which is sparse enough to allow us to solve the normal equation cheaply using traditional iterative methods such as CG. However, notice that in this scheme, though we replace L by $\tilde{L}$, the right-hand side b is still the same as in equation (2.7), which may affect the accuracy of the solution. Notice that after solving for $\boldsymbol{x}_* = L^\dagger \boldsymbol{b}$ or $\tilde{\boldsymbol{x}}_* = \tilde{L}^\dagger \boldsymbol{b}$, we still need to sort the entries in $\boldsymbol{x}_*$ and $\tilde{\boldsymbol{x}}_*$ to get the rankings of the vertices. Granted that the convergence result proves that $\boldsymbol{x}_*$ and $\tilde{\boldsymbol{x}}_*$ approximate each other under equation (3.9), this does not mean the sorting of the entries will not be affected. As later confirmed in numerical experiments, using Algorithm 11 to solve normal equation directly will not give out the exact ranking, though the solutions satisfy equation (3.9). However, using Algorithm 9 or 10 will give us the exact solution, by using a more complicated computation. We may investigate further whether this trade-off is worthy and ways to improve Algorithm 11.

3.2.3 Accuracy Improvement via Averaging and Multiple Samplings

So far, we have seen this tradeoff between SSC and the sampling algorithm. While SSC gives us the exact ranking, it is computing on the original graph G, a dense graph with at most $O(n^2)$ complexity. As shown in equation 3.4, the dense graph Laplacian L is still involved in every iteration. On the other hand, the sampling algorithms proposed in Section 3.2.1 trim the graph down to $O(n \log n)$, but solving the normal equation on the sampled graph gives out imprecise ranking.

Given the definition $L = A^T A$ for $A = W^{\frac{1}{2}} \partial_1^T \in \mathbb{R}^{|E| \times |V|}$, we can also write L as a sum of rank 1 matrices:

$$L = \sum_{i=1}^{|E|} a_i a_i^T, \tag{3.10}$$

where a_i is the ith row of A. Then it follows from Algorithm 7 that we can write $\tilde{L} = A_s^T As$ as

$$\tilde{L} = \frac{1}{s} \sum_{t=1}^{s} \frac{1}{p_{i_t}} a_{i_t} a_{i_t}^T, \tag{3.11}$$

and it follows naturally that for any choice of sampling density,

$$\mathbb{E}[\tilde{L}] = L, \tag{3.12}$$

so that $\tilde{L}$ is an unbiased estimator for L. According to analysis in [11] and [1], weighted row sampling will minimize the variance in Frobenius norm and keep the spectral norm in a small variance with high probability. Because of this property, we deem it reasonable to sample the graphs multiple times to improve the ranking by averaging the results. Given this idea, we can then modify Algorithm 7 and 8.

Algorithm 12 Modified Sampling Approach With Uniform Sampling and Averaging

```
1: for $t = 1 \to m$ do
2:     $A_{s_t} \leftarrow \textsc{UniformRowSampling}(G)$;
3:     Solve for $\tilde{\boldsymbol{x}}_t$ using SSC;
4: end for
5: $\tilde{\boldsymbol{x}} \leftarrow \frac{1}{m}\sum_{t=1}^{m}\tilde{\boldsymbol{x}}_t$;
```

Algorithm 13 Modified Sampling Approach With Weighted Sampling and Averaging

```
1: for $t = 1 \to m$ do
2:     $A_{s_t} \leftarrow \textsc{WeightedRowSampling}(G)$;
3:     Solve for $\tilde{\boldsymbol{x}}_t$ using SSC;
4: end for
5: $\tilde{\boldsymbol{x}} \leftarrow \frac{1}{m}\sum_{t=1}^{m}\tilde{\boldsymbol{x}}_t$;
```

Remark: We can tune the number of times to sample m freely. We expect that the larger it is the more accurate the ranking will be. However, to keep the complexity to $O(n\log^2 n)$, we want m to be at most $O(\log n)$ complexity.

Chapter 4

Data

For the numerical experiments, we will consider both synthetic and real world graphs. In order to test the effectiveness of our algorithms, we will increase the size of the graph gradually, make the graphs more ill-conditioned, and pay close attention to the increase in the number of iterations and CPU time.

4.1 Synthetic Graphs

4.1.1 Random Graphs

For numerical experiments conducted on random graphs we consider Erdos-Reyni [5] and Watts-Strogatz [18] random graphs, which are notable random graph models used to simulate real-world networks. For Erdos-Reyni graphs, we set $p = 10\log(|V|)/|V|$ to ensure the strong connectivity of the graph. For Watts-Strogatz random graphs, the rewiring probability of each vertex, β, is set to 1 to create a Watts-Strogatz random graph ($\beta = 0$ will create a non-random ring lattice graph instead). Also, we set mean-vertex degree parameter $K \geq 4\log|V|$.

4.1.2 Square Grid Graphs

We will also consider the square grid graph, which is an $n \times n$ lattice graph $G_{n,n}$ that is the graph Cartesian product $P_n \times P_n$ on two sets of n vertices. As size increases, square grid graph leads to an ill-conditioned problem because of its sparsity. To test the effectiveness of our algorithms, we hope to see its number of iterations to be only slightly affected by the increasing size of the graph. Figure 4.1 is an example of a square grid graph with 16 vertices and 24 edges.

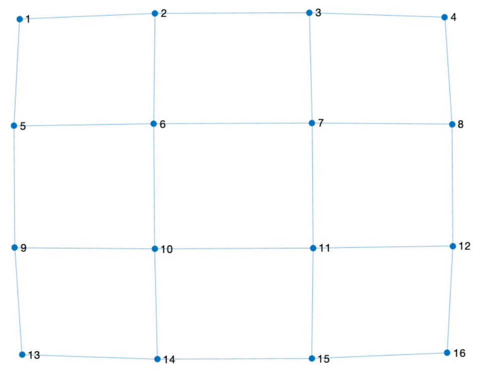

Figure 4.1: Square grid graph with 16 vertices and 24 edges

4.1.3 Real World Data: MovieLens

The MovieLens dataset [7] is a dataset of movies rated by voters collected by the Grouplens team at University of Minnesota. It contains over 20 million ratings for $27,278$ movies by $138,493$ users. Given the graph construction method in Section 2.1.2, the graph of this dataset would have $27,278$ vertices and at most $\binom{27,278}{2}$ edges. However, for the purpose of testing the scalability of our algorithms, we would test the connected components of the graph with gradually increasing sizes. Particularly, we will first arbitrarily determine the orientation, and then use equation (2.1) and equation (2.2) to create edge flow $\boldsymbol{f}$. Since the subset of the votings may not form a connected graph, we have to find its largest connected component before solving.

Chapter 5

Numerical Experiments and Results

For numerical experiments, we want to compare the performances between not only different methods referred in Section 3, but also traditional iterative method such as unconditioned CG for solving normal equation (2.7), and LSQR [13] for solving least squares equation (2.5).

For all iterative methods solving least squares equation in the form of $\min_{x}\|\boldsymbol{Ax}-\boldsymbol{b}\|_2^2$ or normal equation $A^T A\boldsymbol{x} = A^T\boldsymbol{b}$, we use the stopping criterion in terms of relative error $\|A^T(\boldsymbol{b}-A\boldsymbol{x})\|/\|A^T\boldsymbol{b}\| \leq 10^{-6}$. The CPU time (in seconds) reported in all tables includes the solving phase. Moreover, "-" denotes that the iterative method fails to convergence within $3,000$ iterations. For Algorithm 11, 12 and 13, we will compare the ranking of our approximate solution to the exact solution to equation (2.7) and compute the accruacy of the top 100 vertices. All numerical experiments are conducted in MATLAB on a mid-2014 Macbook Pro with 2.2 GHz Intel(R) Core i7 CPU and 16 GB RAM. Each set of parameters will be repeated 5 times and the results will be averaged.

The results of the random graphs are reported in Table 6.1 and 6.2 respectively. The results of square grid graphs are reported in Table 6.3. Moreover, the results of MovieLens dataset are reported in Table 6.7 and 6.6. For Algorithm 12 and 13, on a graph with n vertices, we will initially set to sample and solve on $25\log n$ graphs and take the average of these $25\log n$ solutions as the final solution. The solving duration will be the sum of the solving durations for solving these graphs individually using SSC with coarse grid (Algorithm 6). Since the number of times to sample is a hyper-parameter that we can tune freely, $25\log n$ is a value we choose heuristically. The results for Algorithm 12 and 13 are shown respectively in Table 6.4 and Table 6.5.

Chapter 6

Analysis

In this chapter, we will analyze the convergence behavior of our deterministic and randomized algorithms and existed solvers on three types of graphs: well-conditioned sparse graphs, ill-conditioned sparse graphs, and well-conditioned dense graphs. Particularly, well-conditioned sparse graphs will be set up according to Section 4.1.1, and ill-conditioned sparse graphs will be the square grid graphs described in Section 4.1.2. For well-conditioned dense graphs, we will use MovieLens graphs and Erdos-Reyni graphs with $p = 0.99$ to ensure high density.

Since our algorithms are implemented in MATLAB, which is known for being slow in loops, we believe that the increase in number of iterations is a better and more important indicator of the effectiveness and scalability of our algorithms than the actual CPU elapse time. We also believe that our algorithms would be much faster if implemented in another language. Thus, we deem our algorithms to be effective if they converge in number of iterations fewer than that of the existing iterative methods, though the latter may have a shorter elapse time than our algorithms in a MATLAB implementation.

6.1 Well-conditioned Sparse Graphs (Random Graphs)

From the results, we see that all algorithms converged in all cases. Algorithm 5 has a relatively large elapse time, which is due to the slow implementation of the inner loop, though it will converge eventually. As the size of the graph increases, the number of iterations is steady for all algorithms, though Algorithm 3 takes more iterations to converge. This can be explained by the fact that Algorithm 5 and 6 take account the inner structure of the graph, respectively by creating a edge-based decomposition and a coarse grid. Also, we see that the number of iterations holds steady for Algorithm 9 and 10. For Algorithm 11, the accuracy is around 70%. This is because we are solving two different equations, as discussed in Section 3.2.2.2. On the other hand, plain CG and LSQR took more iterations to converge, though the number of iterations also only increases moderately, which indicates that the graph problem is well-conditioned. To conclude, for sparse random graphs that lead to well-conditioned problems, we would expect that the canonical space decomposition (Algorithm 3) to be enough for scalable and stable convergence, thus coarse grid or sampling is not required.

Also, we recommend not using Algorithm 9 and 10 because we believe they are mathematically similar, respectively, to Algorithm 3 and 6. Recall that Algorithm 3 is equivalent to Gauss-Seidel on the normal equation $L\boldsymbol{x} = \boldsymbol{b}$. In Algorithm 9, the sampled graph Laplacian $\tilde{L}$ actually serves as a preconditioner on L, and

$$\boldsymbol{x_{k+1}} = \boldsymbol{x_k} + \tilde{L}_*^{-1}(\boldsymbol{b} - L\boldsymbol{x}) \tag{6.1}$$

is mathematically equivalent to one iteration of Gauss-Seidel on

$$(\tilde{L}L)\boldsymbol{x} = \tilde{L}\boldsymbol{b}, \tag{6.2}$$

where $\tilde{L}_*$ is the lower triangular matrix of $\tilde{L}$.

6.2 Ill-conditioned Sparse Graphs (Square Grid Graphs)

From the results on Table 6.3, we see that only Algorithm 6 and 10 converge in all cases within $3,000$ iterations. The advantage of the coarse grid can be seen clearly in Figure 6.1, the log-log plot of matrix size versus number of iterations. As the size of the graph increases, for all solvers that include coarse grid, namely Algorithm 6 and 10, the number of iterations increases only moderately. CG and LSQR do not even converge within 3000 iterations for graphs with more than 100 vertices.

We can also see the ill-conditionedness and sparsity of square grid graphs. For a graph with only 400 vertices and 760 edges, SSC with canonical space decomposition

SSC on Erdos-Reyni random graphs														
Graphs		Algorithm												
Info		3		5		6		9		10		11		
vertices	edges	nIter	time(s)	nIter	time(s)	nIter	time(s)	nIter	time(s)	nIter	time(s)	nIter	time(s)	Accuracy(%)
1,000	34,454	7	0.02	4	0.60	5	0.06	4	0.02	4	0.09	31	0.05	81
2,000	76,143	7	0.05	4	2.73	5	0.2	4	0.06	4	0.25	32	0.01	72
4,000	166,088	7	0.17	4	15.71	5	0.75	4	0.19	4	0.88	32	0.03	64
8,000	359,916	7	0.56	4	72.25	5	2.43	4	0.74	4	2.96	33	0.08	60

CG and LSQR on Erdos-Reyni random graphs					
Info		CG		LSQR	
vertices	edges	nIter	time(s)	nIter	time(s)
1,000	34,354	27	0.004	24	0.01
2,000	76,143	28	0.004	25	0.05
4,000	166,088	28	0.004	25	0.05
8,000	359,916	29	0.004	26	0.1

Table 6.1: Numerical results for Erdos-Reyni random graphs

SSC on Watts-Strogatz random graphs														
Graphs		Algorithm												
Info		3		5		6		9		10		11		
vertices	edges	nIter	time(s)	nIter	time(s)	nIter	time(s)	nIter	time(s)	nIter	time(s)	nIter	time(s)	Accuracy(%)
1,000	35,000	7	0.02	4	0.60	5	0.006	4	0.002	4	0.09	58	0.01	80
2,000	76,000	7	0.05	4	2.84	5	0.19	4	0.07	4	0.25	67	0.02	58
4,000	164,000	7	0.15	4	13.90	5	0.67	4	0.20	4	0.83	71	0.02	65
8,000	360,000	7	0.57	4	77.05	5	2.78	4	0.75	4	3.01	72	0.02	58

CG and LSQR on Watts-Strogatz random graphs					
Info		CG		LSQR	
vertices	edges	nIter	time(s)	nIter	time(s)
1,000	35,000	48	0.003	43	0.01
2,000	76,000	58	0.002	51	0.04
4,000	164,000	60	0.02	54	0.11
8,000	360,000	61	0.08	50	0.23

Table 6.2: Numerical results for Watts-Strogatz random graphs

(Algorithm 3) would converge in more than $1,000$ iterations, while the same algorithm would take only 7 iterations to converge on a well-conditioned random graph with $8,000$ vertices and $360,000$ edges. Thus, we have to introduce coarse grid to build a stable and robust alternative to traditional methods like CG.

Using sampling to solve on normal equations (Algorithm 11) fails completely in terms of number of iterations and accuracy. We believe the increasing number of iterations is due to not constructing coarse grid, and inaccuracy due to the sparsity of the square grid graphs. To see this, notice how disconnected the square grid graph is: no matter the size of the graph, the degree of each vertex is either 2,3,or 4. Sampling edges out of the original graph will further decrease the average degree of each vertex, resulting in a sampled graph that does not resemble the original graph. Also, while using sampling on the square grid graph, we have already violated the convergence condition in equation (3.9), which requires sample size $s \geq 4|V|\log|V|$. In square grid graphs, given number of vertices $|V|$, the number of edges would not

SSC on square grid graphs														
Graphs		Algorithm												
Info		3		5		6		9		10		11		
vertices	edges	nIter	time(s)	nIter	time(s)	nIter	time(s)	nIter	time(s)	nIter	time(s)	nIter	time(s)	Accuracy(%)
100	180	292	0.007	116	0.07	31	0.003	146	0.02	17	0.008	110	0.0004	100
400	760	1,170	0.07	484	1.66	31	0.01	549	0.35	21	0.004	453	0.002	27
900	1,740	2,472	0.33	1,075	13.12	35	0.02	1,208	2.2	25	0.32	753	0.005	10
1,600	3,120	-	-	1,887	616.10	38	0.04	2,165	7.83	25	0.32	948	0.01	9
6,400	12,640	-	-	-	-	42	0.22	-	-	25	3.1	1,622	0.16	2
25,600	50,880	-	-	-	-	41	0.97	-	-	25	37.7	-	-	-

CG and LSQR on square grid graphs					
Info		CG		LSQR	
vertices	edges	nIter	time(s)	nIter	time(s)
100	180	671	0.002	660	0.002
400	760	-	-	-	-
900	1,740	-	-	-	-
1,600	3,120	-	-	-	-
6,400	12,640	-	-	-	-
25,600	50.880	-	-	-	-

Table 6.3: Numerical results for square grid graphs

exceed $4|V|\log|V|$.

To conclude, for all ill-conditioned problems, we have to construct coarse grid for good scalability. For even better convergence behavior, we may incorporate both coarse grid and sampling scheme into our algorithm.

6.3 Well-conditioned Dense Graphs

6.3.1 Well-conditioned Dense Erdos-Reyni Graphs

Firstly, from Figure 6.2, Algorithm 12 achieves time complexity $O(n\log^2 n)$ if we bound the number of samples to the scale of $O(\log n)$. Algorithm 13 takes more time to elapse, but theoretically it should have complexity $O(\log n + n\log^2 n)$. To show this, we use a dense Erdos-Reyni graph G with 6400 vertices and approximately 20 millions edges and sample it down to a sparse $\tilde{G}$ with approximately 897 thousands edges. Between the real solution on G and the approximate solutions on $\tilde{G}$, we compare the ranking of the first 10 important vertices, using the analogy of top 10 movies.

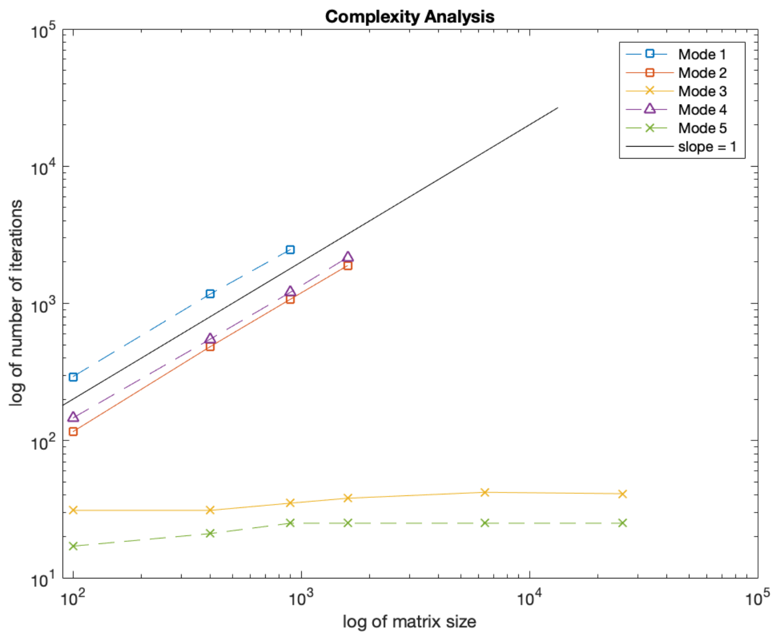

Figure 6.1: Computational complexity for square grid graphs. Mode 1: Algorithm 3. Mode 2: Algorithm 5. Mode 3: Algorithm 6. Mode 4: Algorithm 9. Mode 5: Algorithm 10

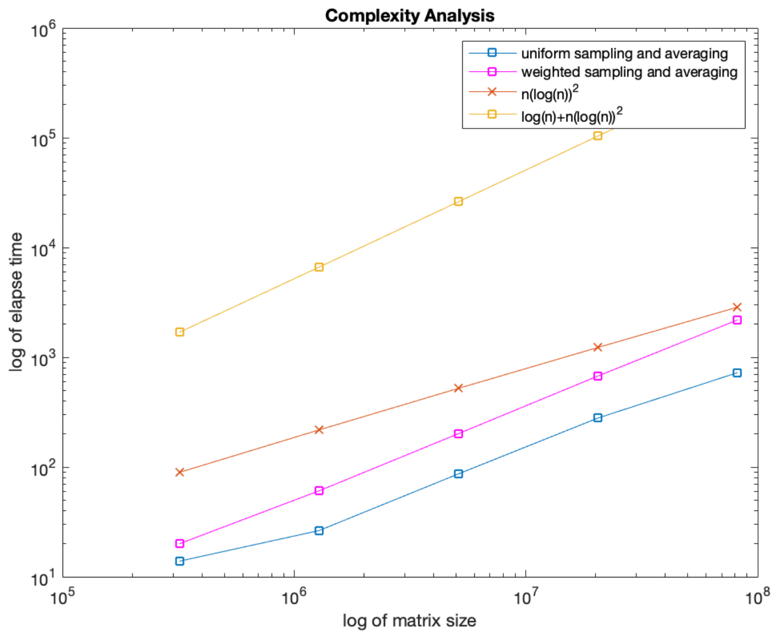

Figure 6.2: Computational complexity for randomized solvers on dense Erdos-Reyni graphs

Algorithm 12 on Erdos-Reyni graphs			
Info			Result
vertices	edges	edges after sampling	elapse time(s)
800	316,353	85,563	13.88
1,600	1,266,551	188,871	26.36
3,200	5,067,916	413,230	85.90
6,400	20,272,932	897,439	279.50
12,800	81,092,542	1,936,835	724.42

Table 6.4: Numerical results for Algorithm 12 on dense Erdos-Reyni graphs

Algorithm 13 on Erdos-Reyni graphs			
Info			Result
vertices	edges	edges after sampling	elapse time(s)
800	316,404	85,563	20.02
1,600	1,266,570	188,871	60.65
3,200	5,067,456	413,230	201.53
6,400	20,271,630	897,439	675.50
12,800	81,093,701	1,936,835	2,187.20

Table 6.5: Numerical results for Algorithm 13 on dense Erdos-Reyni graphs

$$\boldsymbol{r}_* = \begin{pmatrix} 140 \\ 11 \\ 51 \\ 56 \\ 47 \\ 29 \\ 31 \\ 101 \\ 17 \\ 85 \end{pmatrix}, \quad \tilde{\boldsymbol{r}}_{\text{uniform}} = \begin{pmatrix} 51 \\ 11 \\ 140 \\ 31 \\ 101 \\ 56 \\ 98 \\ 29 \\ 47 \\ 39 \end{pmatrix}, \quad \tilde{\boldsymbol{r}}_{\text{weighted}} = \begin{pmatrix} 140 \\ 11 \\ 51 \\ 56 \\ 31 \\ 47 \\ 101 \\ 29 \\ 17 \\ 3 \end{pmatrix} \tag{6.3}$$

We can clearly see that Algorithm 13 takes much longer time to elapse. This is mainly due to the extra computation involved in locating the edge to sample in each iteration of the loop, according to the pre-calculated cumulative density function. Also, we can clearly see that both Algorithm 13 (right) and Algorithm 12 (middle) have included most of the correct vertices of interest, though Algorithm 12 has a more accurate ranking. Given the extra computational cost in Algorithm 13, in practice, we propose that if precise accuracy is less of a priority than computational cost, and we only want to know the vertices that are relatively more important, Algorithm 12 is a viable alternative that preserves not only $O(n \log^2 n)$ complexity but also an accurate inclusion of vertices of interest, though their order, namely ranking, may be inaccurate. If for this purpose, weighted sampling is not necessary. To achieve better accuracy, we can further increase sample size or the number of samples, which are currently $16n \log n$ and $25n \log n$ respectively, or we can use Algorithm 13, paying an extra computational cost of calculating the density function of the edge weights at the start. On the other hand, if we pursue absolute accuracy, SSC works better than these randomized algorithms with an extra cost in computation, though we have also shown that coarse grid and sampling can be applied to achieve scalable convergence behavior.

6.3.2 Well-conditioned Dense MovieLens Graphs

In this section, we will follow the similar analysis done in Section 6.3.1. Firstly, from Figure 6.3, we can see that on MovieLens graphs Algorithm 12 and 13 also follow similar convergence behaviors as described in Section 6.3.1. Comparing the accuracy between Algorithm 12 and 13 in equation (6.4), we use a MovieLens graph G with $8,907$ vertices and approximately 10 million edges and sample it down to a sparse $\tilde{G}$ with approximately 1.3 million edges. The result we get is similar to Section 6.3.1. Algorithm 13 achieves higher accuracy. Algorithm 12 has gotten the correct vertices but some of the rankings between them are wrong. Thus, we follow the same conclusion as in Section 6.3.1.

$$\boldsymbol{r}_* = \begin{pmatrix} 27 \\ 61 \\ 87 \\ 216 \\ 94 \\ 445 \\ 201 \\ 50 \\ 26 \\ 157 \end{pmatrix}, \quad \tilde{\boldsymbol{r}}_{\text{uniform}} = \begin{pmatrix} 27 \\ 161 \\ 61 \\ 347 \\ 445 \\ 157 \\ 87 \\ 216 \\ 201 \\ 26 \end{pmatrix}, \quad \tilde{\boldsymbol{r}}_{\text{weighted}} = \begin{pmatrix} 27 \\ 61 \\ 87 \\ 94 \\ 216 \\ 26 \\ 201 \\ 50 \\ 44 \\ 193 \end{pmatrix} \tag{6.4}$$

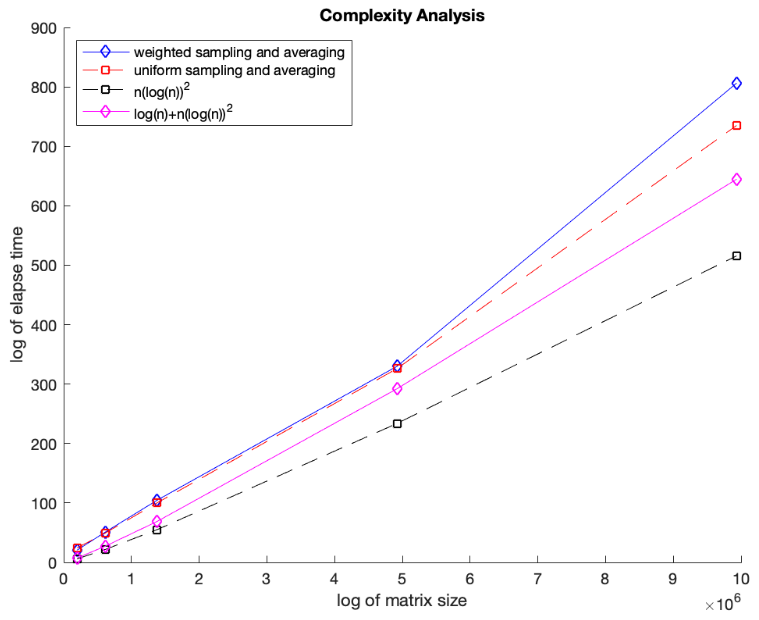

Figure 6.3: Computational complexity for randomized solvers on MovieLens graphs

Algorithm 12 on MovieLens graphs			
Info			Result
vertices	edges	edges after sampling	elapse time(s)
870	197,896	94,217	24.36
1,613	614,883	190,614	49.49
2,412	1,378,926	300,563	100.15
4,893	4,923,331	665,100	326.16
8,907	9,934,406	1,296,089	735.10

Table 6.6: Numerical results for Algorithm 12 on MovieLens graphs

Algorithm 13 on MovieLens graphs			
Info			Result
vertices	edges	edges after sampling	elapse time(s)
870	197,896	94,217	20.68
1,613	614,883	190,614	51.04
2,412	1,378,926	300,563	104.63
4,893	4,923,331	665,100	330.10
8,907	9,934,406	1,296,089	806.34

Table 6.7: Numerical results for Algorithm 13 on MovieLens graphs

Chapter 7

Conclusion and Future Work

We have shown that SSC routines are a class of scalable least squares algorithms for the least squares graph problems in equation (2.5). We tested these routines for a variety of random graphs and one real world network dataset, which led to well-conditioned and ill-conditioned least squares problems. The numerical results have shown that SSC with coarse grid and sampling (Algorithm 10) is robust in both ill-conditioned and well-conditioned graphs. For well-conditioned sparse graphs, we would recommend Algorithm 3 for scalable steady convergence behavior, and Algorithm 6 or 10 for ill-conditioned problems, where a coarse grid has to be constructed to achieve scalable steady convergence. We believe that this is a promising approach and has potential on other real world applications. We also hope that this paper would bring SSC attention as an important graph algorithm. On the other hand, we also have developed a uniform and a weighted sampling algorithm, denoted as Algorithm 12 and 13, that achieve complexity $O(n \log^2 n)$ and reasonably good accuracy. We would recommend using these algorithms when the graph is well-conditioned and dense, and accuracy is not of the highest priority, because sampling destroys the graph structure on sparse and ill-conditioned graphs.

For future work, there is one important direction that can be worked on. We conjecture that our randomized algorithms can be improved if we design a more complex sampling scheme that can further sample the graph down to a spanning tree so that solving on this spanning tree gives out the same ranking as the original graph. This conjecture is based on an observation from a simplified example.

Suppose we have Figure 7.1, a fully connected graph G with only 3 vertices and 3 equally weighted edges. We can then remove the edge between v_1 and v_3 and call this sampled graph $\tilde{G}$, which is also a spanning tree. Now, it is easy to see that if we solve the ranking problem on both G and $\tilde{G}$, their solutions are both $A > C > B$.

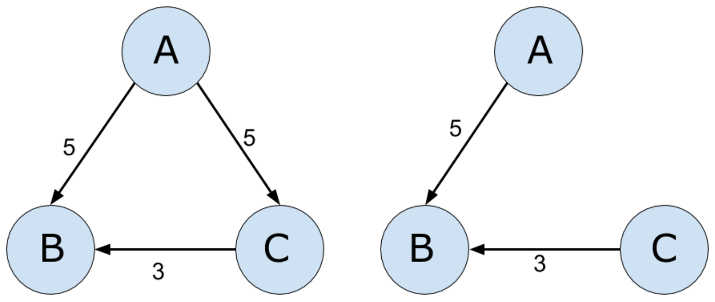

Figure 7.1: Left: fully connected graph G. Right: sampled spanning tree $\tilde{G}$

From hindsight, we know that sampling that edge out of the graph does not change the final ranking. The question now turns to that, when we look at any edge, how can we know beforehand that sampling it out will not affect the final ranking? If we can answer this question, then we can sample any graph down to a spanning tree, solve the ranking problem on this spanning tree and get the precise solution on the original graph.

Bibliography

[1] Long Chen and Wu Huiwen, *A preconditioner based on non-uniform row sampling for linear least squares problems*, 2018.

[2] C. Colley, J. Lin, X. Hu, and S. Aeron, *Algebraic multigrid for least squares problems on graphs with applications to hodgerank*, in 2017 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW), May 2017, pp. 627–636.

[3] Petros Drineas, Ravi Kannan, and Michael W Mahoney, *Fast monte carlo algorithms for matrices i: Approximating matrix multiplication*, SIAM Journal on Computing, 36 (2006), pp. 132–157.

[4] Petros Drineas and Michael W. Mahoney, *Effective resistances, statistical leverage, and applications to linear equation solving*, CoRR, abs/1005.3097 (2010).

[5] P ERDdS and A R&wi, *On random graphs i*, Publ. Math. Debrecen, 6 (1959), pp. 290–297.

[6] Miroslav Fiedler, *Algebraic connectivity of graphs*, Czechoslovak mathematical journal, 23 (1973), pp. 298–305.

[7] F Maxwell Harper and Joseph A Konstan, *The movielens datasets: History and context*, Acm transactions on interactive intelligent systems (tiis), 5 (2016), p. 19.

[8] Anil Nirmal Hirani, Kaushik Kalyanaraman, and Seth Watts, *Graph laplacians and least squares on graphs*, in Proceedings - 2015 IEEE 29th International Parallel and Distributed Processing Symposium Workshops, IPDPSW 2015, United States, 9 2015, Institute of Electrical and Electronics Engineers Inc., pp. 812–821.

[9] Xiaoye Jiang, Lek-Heng Lim, Yuan Yao, and Yinyu Ye, *Statistical ranking and combinatorial hodge theory*, Mathematical Programming, 127 (2011), pp. 203–244.

[10] Iordanis Kerenidis and Anupam Prakash, *Quantum recommendation systems*, in ITCS, 2017.

[11] Michael W. Mahoney, *Lecture notes on randomized linear algebra*, CoRR, abs/1608.04481 (2016).

[12] Bojan Mohar, *The laplacian spectrum of graphs*.

[13] Christopher C Paige and Michael A Saunders, *Lsqr: An algorithm for sparse linear equations and sparse least squares*, ACM Transactions on Mathematical Software (TOMS), 8 (1982), pp. 43–71.

[14] Daniel A Spielman, *Spectral graph theory and its applications*, in 48th Annual IEEE Symposium on Foundations of Computer Science (FOCS'07), IEEE, 2007, pp. 29–38.

[15] Xue-Cheng Tai and Jinchao Xu, *Global and uniform convergence of subspace correction methods for some convex optimization problems*, Math. Comput., 71 (2002), pp. 105–124.

[16] Ewin Tang, *A quantum-inspired classical algorithm for recommendation systems*, CoRR, abs/1807.04271 (2018).

[17] John C Urschel, Xiaozhe Hu, Jinchao Xu, and Ludmil T Zikatanov, *A cascadic multigrid algorithm for computing the fiedler vector of graph laplacians*, arXiv preprint arXiv:1412.0565, (2014).

[18] Duncan J Watts and Steven H Strogatz, *Collective dynamics of ?small-world?networks*, nature, 393 (1998), p. 440.

[19] Jinchao Xu and Ludmil Zikatanov, *Algebraic multigrid methods*, Acta Numerica, 26 (2017), pp. 591–721.