gSparse: Library for Graph Sparsification

Thanaphon Chavengsaksongkram

Dissertation
Master of Science in Artificial Intelligence
School of Informatics
University of Edinburgh

2018

Abstract

This dissertation aims to close the gap between theoretical Computer Science and practical application in the field of Spectral Graph Theory. Thus, the first open-source, header-only C++ library for Graph Sparsification is developed. A literature review is also included, covering state of the art relating to Spectral Sparsification and best practices for developing a C++ software library. The initial build of the software features Spectral Sparsification by Effective Resistance with promising results. The implementation performs significantly faster than the existing state-of-the-art.

Acknowledgements

I would like to express my sincere gratitude to my supervisor Dr. He Sun for the continuous support of my dissertation and related research. Dr. He Sun goes extra mile in introducing the concepts of Spectral Graph Theory. His patience, motivation, and immense knowledge has guided me in performing a successful research.

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

Graphs are one of the most critical data abstraction in computer science because they can represent data that are often too complicated to describe adequately otherwise. In many practical applications, graphs are used to express data such as social networks, city maps, and bus routes. Graphs allow scientists and engineers to analyze large data automatically through the use of algorithms such as Dijkstra and Bellman-Ford [Cormen *et al.*, 1990].

Many graph algorithms perform in polynomial time in respect to the number of its nodes, for example, Floyd-Warshall algorithm [Cormen *et al.*, 1990] calculates the shortest path every pair of vertices in the graph at $O(n^3)$ time complexity, meaning that the execution time is expected to grow at an exponential rate compared to the number of vertices.

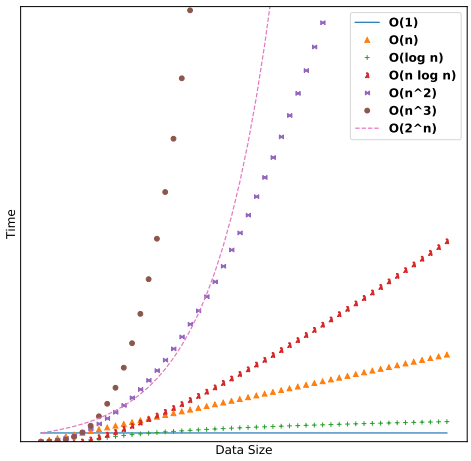

Figure 1.1: Space and Time Complexity of Algorithm

Figure 1.1 shows the runtime cost of the algorithm based on time complexity. Many polynomial algorithms may not be feasible for real-time industrial application once

the data size becomes enormous. For example, an algorithm that clusters millions of cell phone subscribers for capacity planning may not be so useful without real-time processing speed. Hence, optimizing graph algorithms have been a subject of interests for both scientific and industrial endeavors.

One of the many optimization techniques is to artificially reduce the amount of the data through approximation. This allows the existing algorithms to operate faster within a degree of error. For graphs related data, this process is referred to as Graph Sparsification. Graph Sparsification is a procedure that given a graph A, simplify it with another graph that approximately preserves some property of A.

Although there are many theoretical frameworks for Graph Sparsification available such as [Chu *et al.*, 2018], as far as we know, there is currently no existing open-source implementation that can be extended for practical applications. Hence, the primary motivation of this dissertation is to bridge the gap between theoretical and practical work of Graph Sparsification algorithms.

1.1 Objectives

The main objective of this dissertation is to build gSparse, a Library for Graph Sparsification. gSparse will initially based on Spectral Graph Sparsification by Effective Weight Resistance [Spielman & Srivastava, 2008]. The emphasis of this project puts on extensibility and performance of the Library.

The work of this dissertation can be summarized into the following. First, a literature review is performed to understand the state of Spectral Sparsification algorithms, to investigate potential issues, to identify and evaluate any existing works and implementations, and to research industry practices for constructing an extensible and high-performance software Library. Second, a software library for Graph Sparsification is constructed. And finally, experiments are performed to evaluate software implementation against the existing state of the art.

1.2 Dissertation Roadmap

The dissertation begins with Literature review to understand the state of Spectral Graph Sparsification and other resources required to implement this project. The software development process will be documented in the Requirement, Design, and Implementation chapters. Evaluation of the software will be done in the Quality Control and Experimentation chapters. Benchmark data will be discussed in the experiment results and analysis chapter. Finally, a summary of contribution and future work is discussed in the Conclusion chapter.

Chapter 2

Literature Review

2.1 Introduction

Prior to the development of any solutions, it is imperative to understand the problem domain. The literature review aims to capture any necessary background, identify state of the art, evaluate existing work, and research appropriate tool and processes required to produce gSparse.

2.2 Background Knowledge

2.2.1 Graph

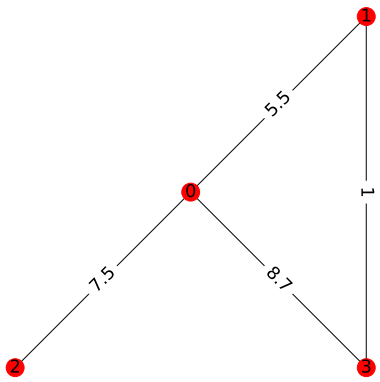

Figure 2.1: An undirected graph

In the context of discrete mathematics, a graph is a data structure containing sets of ordered pairs $G = (V, E, w)$ that are related [Weisstein, e]. Each node of a graph are called vertices, described as a set V. The relationship between each node are called edges and defined in the set E. Each edge may have an associated value called weight,

described as a set w. Edges without weight are generally defaulting to one. There are two types of graphs: directed graph and undirected graph.

For a directed graph, the order of the of vertices in the Edge sets matters [Weisstein, d]. Thus a node u is adjacent to a node v only if the pair (u,v) exists in the Edge set and not the vice versa. For an undirected graph, the order of the vertices does not matter [Weisstein, i]. Thus u is adjacent to v if either pair (u,v) or (v,u) exists in the Edge set.

Even though a set of edges and vertices can adequately represent a graph, it is also common to represent a graph as a matrix.

2.2.2 Adjacency Matrix

Adjacency matrix represents a graph by specifies which vertices in a graph contain an edge in between with associated weight value. Each row and column index of adjacency matrix refers to a vertex [Weisstein, a]. As a result, the adjacency is always a square matrix. For an undirected graph, the adjacency matrix is always symmetric. If a graph is a simple graph, a graph that contains no looping edges to itself, the diagonal entries of its adjacency matrix are all zero.

$$A_{uv} = \begin{cases} w_{uv} & if(V_u, V_v) \in E \\ \\ 0 & otherwise \end{cases}$$

2.2.3 Degree Matrix

A degree matrix is an all-zero matrix except for its diagonal entries, which stores the number of weighted edges attached to each vertex [Weisstein, c]. As the same with the adjacent matrix, the row and column index of degree matrix specifics its associated vertices.

$$D_{ij} = \begin{cases} Deg(i,j) & ifi = j \\ \\ 0 & otherwise \end{cases}$$

2.2.4 Incident Matrix

Incident Matrix defines the relationship of vertices within the graph. Often it is defined as a vertex-edge matrix: the row represents vertices and column representing an edge [Weisstein, f]. If there is an incoming edge to a particular vertice, a value -1 is assigned. Vice versa, if there an edge is outgoing from vertice, value 1 is assigned. In the case

of no connected edge, the value defaults to zero. For an undirected graph, there would be both incoming and outgoing edges for each particular vertice if they are connected.

$$B_{ev} = \begin{cases} 1 & if \text{ v equals head of e} \\ -1 & if \text{ v equals tail of e} \\ 0 & otherwise \end{cases}$$

2.2.5 Laplacian Matrix

A Laplacian matrix is often used in many Spectral Graph theory to characterizes the graph. A Laplacian matrix can be constructed by subtracting Diagonal matrix with Adjacency matrix [Weisstein, g].

$$L = D - A$$

For an undirected simple and connected graph, the Laplacian Matrix is always symmertric, diagonally dominant, its off-diagonal entries are non-positive.

2.3 Graph Sparsification

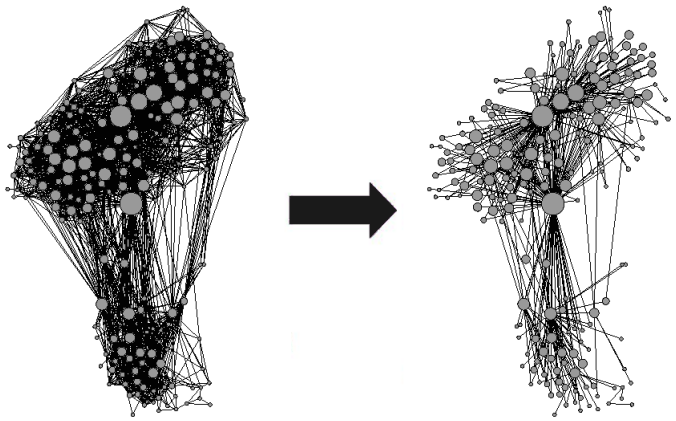

Figure 2.2: Structure Preserving Sparsification [Lindner *et al.*, 2015]

Graph sparsification is a process of approximating a big graph with a smaller counterpart to save storage and speed up processing speed. One of the easiest ways to sparsify a graph is to preserve each edge based on uniform sampling. However, this can be problematic because some edges are more important than the others. For example,

a graph that represents a map of multiple islands, removing the bridges that connect the islands can destroy the graph's connectivity properties. Thus, many graph sparsification algorithms, though share the same goals of reducing edges, are differentiated based on which properties it aims to retain. For example, [Chew, 1986]'s distance-based sparsifier preserves the distance between every pair of vertices, [Benczúr & Karger, 1996]'s cut-based sparsifier retains the weight of the boundary of every set of vertices, and lastly, [Spielman & Teng, 2008]'s spectral sparsifier preserves graphs' Laplacian quadratic form.

2.3.1 Spectral Sparsification

[Spielman & Teng, 2008] introduces the notion of Spectral Graph Sparsifier. For an undirected graph, the sparsifier generates a subgraph H from graph G which preserves graph Laplacian's quadratic form with the following guarantees:

$$(1-\varepsilon)x^T L_G x \leq x^T L_H x \leq (1+\varepsilon)x^T L_G x$$

Spectral Sparsifier has enjoyed popularity in many applications such as algorithm design and network analysis. It has become an active area of research with many new proposed algorithms. The initial Spectral Sparsifier leverages a recursive graph partitioning algorithm that computes approximate sparsest cuts in a nearly-linear time and generating $O(nlog^c n)$ edges, where n refers to the number of nodes and c is a large arbitrary constant. This is improved by [Spielman & Srivastava, 2008], an edge-efficient Spectral Sparsification algorithm based on physical approximation. The algorithm views a graph as a series of electrical circuit and sample edges based on probability proportional to the graph's effective resistance multiply the edge's weight. The algorithm generates the same sparsification guarantees using only $O(nlog(n)/\varepsilon^2)$ edges. [Koutis *et al.*, 2012] introduced the use of fixed size Johnson-Lindenstrauss random projection matrix to speed up the Sparsification algorithm in an unrelated work of building an efficient Symmetric Diagonally Dominant matrix solver.

In addition to Speilman's algorithms, [Allen Zhu *et al.*, 2015] constructs a new and fast novel algorithm for computing linear sized spectral sparsifier by relating sparsification to a regret minimization problem. [Lee & Sun, 2017] introduces a nearly linear time algorithm to produce a linear-sized Sparsifier through a use of new potential function, efficient reduction, and one-sided Spectral Sparsifier by a semi-definite program.

As there are many novel constructions of Spectral Sparsifiers, the scope of this dissertation is limited to building an open-source implementation of Spectral Sparsification based on [Spielman & Srivastava, 2008]. The algorithm serves as a baseline implementation for future implementation of other Sparsifiers, which can is described in Algorithm 1.

Algorithm 1 Spectral Sparsification by Effective Resistance

$n \leftarrow$ number of vertices
$C \leftarrow$ hyper-parameter
$\varepsilon \leftarrow$ hyper-parameter
$Reff \leftarrow$ returns effective resistance per given edge
for `every edge` $u \sim v$ `in the graph` **do**
    `let` $q_{u,v} = w(u,v) * Reff(u,v)$
    `let` $p_{u,v} = min(1, C * (logn)\dot{w}(u,v) * Reff(u,v))$
t = 0;
H = (V,∅, ∅)
while $t \leq (nlogn/\varepsilon^2)$ **do**
    `t = t + 1;`
    `Sample an edge` $u \sim v$ `with probability` $p_{u,v}$
    `Add sampled edge` $u \sim v$ `to H with weight` $w(u,v)/p_{u,v}$
return graph H

2.3.2 Effective Resistance

The key part of [Spielman & Srivastava, 2008] algorithm is the effective resistance function $Reff(u,v)$. The effective resistance between two vertices is defined as the potential difference induced between the vertices when a unit current is injected at one vertex and extracted at the other [Vos, 2016]. Thus, computing the exact effective resistance of each edge requires solving a Laplacian system or calculating the determinant using Kirchoff max-graph theorem as described in [Vos, 2016]. Unfortunately, both of these methods are costly endeavors. [Spielman & Srivastava, 2008] introduces an algorithm that can approximate effective resistance to a constant factor by reducing the problem down to solving an $O(logn)$ linear system. The approach is to characterize the effective resistance as a square distance of vectors applied by the Johnson-Lindenstrauss random projection matrix. Furthermore, [Koutis *et al.*, 2012] shows that using a fixed sized Johnson-Lindenstrauss projection can speed up the calculation. The implementation of the approximate effective resistance can be described in Algorithm 2.

Algorithm 2 Approximate Effective Resistance

$n \leftarrow$ number of vertices
$m \leftarrow$ number of edges
$L \leftarrow$ Graph Laplacian Matrix
$W \leftarrow$ Graph Weight List
$B \leftarrow$ Graph Incident Matrix
$t = 0;$
$Scale = ceil(log_2(n))/\varepsilon$
$Reff = \emptyset$
while $t \leq Scale$ **do**
    `t = t + 1;`
    `let` $Q = 1xm$ Johnson-Lindenstrauss Random Projection
    $Q = Q/Scale$
    $Z = L^{-1}Q(diag(W^{1/2}))B$
    for `every edge` $u \sim v$ `in the Graph` **do**
        `let` $Reff(u,v) = Reff(u,v) + (Z(u) - Z(v))^2$
return $Reff$

2.4 Solving Laplacian System

Based on Algorithm 2, the performance of this algorithm heavily relies on the calculation of Z, which requires a linear system solver for graph Laplacian. Solving linear systems is currently an active area of research, because many graph related problems such as spectral clustering, partitioning, and drawing can all be expressed in terms of a sparse graph Laplacian matrices. Various solvers have been proposed for solving a large sparse system; many of them are built specifically for Graph Laplacian problems. They can be classified into the following.

2.4.1 Direct Solvers

There are many direct solvers available for solving a sparse linear system. Examples include Cholesky decomposition, LU factorization, and QR factorization. These direct solvers are easily available through many numerical analysis packages, but may not be suitable for a very large system.

2.4.2 Iterative Solvers

Iterative solvers such as Conjugated Gradient initially guess a solution and then sequentially generate improving approximate solutions to a system. Iterative solvers usually work well with a preconditioner which transforms the system into a more suitable form for the solver. Simple preconditioners such as incomplete Cholesky and

Jacobi are effective yet simple to implement. They are widely available in major linear algebra packages.

2.4.3 Theoretical Laplacian Solvers

There has been a variety of theoretical Laplacian solvers introduced in literature such as [DBL, 2003]. However, we are unable to find any practical implementations of these algorithms. Some of the new theoretical solvers, such as an unpublished approximate Cholesky, are implemented as work-in-progress by Daniel Speilman in his Laplacians.jil Julia package.

2.4.4 Practical Multigrid Solvers

There are several multigrid solvers introduced to solve Graph Laplacians. These are generally used as a preconditioner in conjunction with an iterative method such as Conjugated Gradient method. [Koutis *et al.*, 2011] introduces Combinatorial Multigrid (CMG) which construct hierarchies of modified spanning tree as a preconditioner. The existing implementation is still being maintained and continuously developed, but it is only available in MATLAB. Another notable multigrid solver is the Lean Algebraic Multigrid [Livne & Brandt, 2012], which is based on Algebraic Multigrid approach but customized for Graph Laplacians. There is an existing implementation in Python from the author, but it appears to be abandoned with unfinished documentation and previous updates dated back over four years at the time of this dissertation.

2.5 Existing Work

In this section, the existing implementations relating to Graph Sparsification by the effective resistance are listed.

2.5.1 Laplacians.jil

Developed by Daniel Speilman, Laplacians.jil is a Julia library package for graph algorithms related to Spectral and Algebraic graph theory. There is a Spectral Sparsification by effective resistance implementation available for Spectral Graph Theory called sparsify.jil. The code is captured and copied to Appendix section of this dissertation.

Upon reviewing the work, there is a notable difference than what is described in [Spielman & Srivastava, 2008]. Specifically in the following lines.

```
1 function sparsify(a; ep=0.3, matrixConcConst=4.0, JLfac=4.0)
2 ...
3 (ai,aj,av) = findnz(triu(a))
4 ...
5 ind = rand(Float64,size(prs)) .< prs
6 as = sparse(ai[ind],aj[ind],av[ind]./prs[ind],n,n)
7 ...
8 return as
```

The function takes in a as graph adjacency matrix. Line 3 creates an edge and weight list from the adjacency matrix a. Line 5 creates an edge selector *ind* based on sampling probability of every edge stored in *prs*. Line 6 rebuilds adjacency matrix based on edge selector *ind* to create a sparsifier.

This method is different than what is described in [Spielman & Srivastava, 2008]. The implementation suggests that the sampling is done at every edge, potentially generate $O(m)$ algorithm, instead of creating an $O(nlogn/ep^2)$ sparsifier, where m is the number of edges and n is the number of vertices. Another key difference is that any edge with sampling probability of 1.0 will always be preserved, however, this is not the case in [Spielman & Srivastava, 2008].

2.5.2 FastEffectiveResistance.mat

Developed by Richard Peng and Gary Miller, this is a MATLAB implementation of effective resistance calculation based on the work of [Koutis *et al.*, 2012] and [Koutis *et al.*, 2011]. The code takes in list of edges and weight as input and calculate graph's approximate effective resistance, presumably to be used with Spielman-Srivastava algorithm. Notably the algorithm leverages Combinatorial Multigrid as a preconditioner for Conjugated Gradient method to solve the Laplacian System. The code of this algorithm is captured and stored in the appendix section.

2.6 Development Language

Although there is no standard framework for building an extensible numerical software library, it is normally written in C or C++. For example, Python's most popular numerical computing library Numpy and Scipy are both written in C++ [Jones *et al.*, 2001–] Travis E [2006–], Google's high-performance Tensorflow is written in C ++ [Google, 2018–]. C++ allows objected oriented programming without sacrificing low-level control. C++ code compiles directly to target platforms without unnecessary layers of virtual machines like Java and C, allowing high optimization. Although C++ is a strongly-typed language, meaning that the data type must be defined as compile time, it supports the use of template metaprogramming which allows a dynamic definition of the data type. C++ can also be easily exposed to other languages such as

python and MATLAB through the use of a wrapper.

2.7 Framework for Building Library

There is no specific framework for building a C++ library. However, they can be classified into two categories: Header-only libray and Linked library.

2.7.0.1 Header-only Library

Header-only library is a C/C++ concept where the full definition of the libraries are defined and visible to the compiler only in a header form [Wilson, 2005]. There are no precompiled binaries for linkers such as Shared Object (.so) in Linux and Dynamic Linked Libraries (.dll) in Windows. Due to lack of standard package manager in C++, a header-only library has become popular as it can be easily distributed through source code sharing platforms such as GitHub.

Strength of Header-only Library:

1. No separate compilation required
2. Optimizer can potentially perform better when source code is available
3. Easy to distribute as there are no binary files
4. Supports the use of template meta-programming

Weakness of Header-only Library

1. Long compilation time
2. Hard to maintain as library becomes very big

2.7.0.2 Linked Library

Linked-library is standard practice for building a large C++ library. The interface definition and its implementation are segregated into a header file and files that are compiled into a linked library.

Strength of Dynamic Linked Library:

1. Faster compilation
2. Source code can be omitted
3. Code remains maintainable as code size scales

Weakness of Dynamic Linked Library:

1. Difficult to distribute as it is platform specific
2. No access to template meta-programming

2.7.1 Numerical Computing Library

Graph Sparsification involves matrices and linear algebra operations. There are many numerical libraries available for C++. The C++ programming language provides linear algebra functionality through its Boost.uBLAS library, but unfortunately, the project is no longer in development [Boost, 2008–]. In this dissertation, two of the most popular high-performance C++ numerical analysis packages are considered: Armadillo and Eigen3.

2.7.1.1 Armadillo

Armadillo is a linear algebra library for C++ that aims to provide a smooth transition for MATLAB programmer through balance of speed and ease of use. The key benefit of Armadillo is that it outsources its backend computing to an external optimized library such as PETsc and Intel MK [Armadillo, 2018–]. Armadillo also supports the use of template meta programming.

2.7.1.2 Eigen3

Eigen3 is a numerical analysis package for C++ with the goal of good integration and compatibility with other libraries. Eigen3 develops its own computing backend and features a very flexible but robust Dense and Sparse Matrix data type Guennebaud *et al.* [2010]. As a result, Eigen3 are often used by other high-performance libraries such as Google's Tensorflow [Google, 2018–].

Eigen3 features its own highly optimized matrix product kernels without requiring its user to link to other external computing backends. This makes Eigen3 the most appropriate solution for gSparse as it allows gSparse to minimize the number of dependencies.

2.7.2 Development Environment

2.7.2.1 Integrated Development Environment

GCC, Clang, and Microsoft Visual C++ are some of the most popular C++ compilers on the market. Microsoft Visual C++, together with Visual Studio, is known for its ease of use, which allows very fast prototyping. Modern Microsoft's C++ compiler is compliant with C++11 standards. Therefore, the development of the initial and prototype version will primarily be done in Microsoft Visual Studio with occasional migrations to GCC and Clang for testing.

2.7.2.2 Memory Profiler

C++ allows full control of memory management, which can lead to human-error in dealing with dynamic memory allocations. Fortunately, Visual Leak detector (VLD) can be integrated into Visual Studio to profile the use of dynamic memory.

2.7.3 Source Control

A version control system is required when managing changes of software. One of the most popular hub for distributing open-source software is GitHub. Hence, the use of Git versioning control system is inevitable for gSparse as it is an open-source project. Git, as a distributed open-source control system, provides a key benefits is that it does not require a constant connection to the central repository. The developer can work seamless offline and publish a change only when is needed.

2.7.4 Test Framework

Software testing is an important part of the development process. Testing ensures that the software is working correctly and according to its specification. Although the C++ standard library does not provide a framework for software testing, there is an open-source solution called Catch.hpp which provides an alternative light-weight C++ framework for software testing [Catch2, 2018].

2.8 Conclusion

With a precise understanding of the literature in the domain and the development tools required to build gSparse. The development process can formally begin and can be summarized into the four consecutive phrases: requirement specification, software design, software implementation, and testing and evaluation.

Chapter 3

Requirement Specifications

3.1 Introduction

Even though the use of gSparse primarily limits to the Graph Sparsification, it is imperative to formalize the software requirements. This is to ensure that sufficient attention is paid to the implementation of gSparse and to give a clear overview of what gSpare will and will not include in its initial build.

3.2 Requirement Prioritization

Each requirement specification is classified based on priority: Critical, Essential, and Non-essential.

3.2.0.1 Critical

Critical requirements are the core functionality of gSparse. Without the critical requirements fulfilled, the development of gSparse is not fully completed.

3.2.0.2 Significant

Significant requirements are the functionality that will have a profound effect to gSparse. Without these requirements, gSparse would be a functional prototype only.

3.2.0.3 Insigificant

Non-vital requirements are the functionalities that provide niche features to gSparse. These are non-essential and are generally considered ”nice to have”.

3.2.0.4 Software Requirements

gSparse is a software library which will be consumed by software developers, who may integrate gSparse in their existing software environment. The software requirement specification outlines gSparse's compatibility obligation to external tools and libraries.

ID	Requirement	Importance
S1	gSparse compiles on GCC 4.8	Critical
S2	gSparse compiles on MSVC 19.0	Critical
S3	gSparse compiles on Clang 3.0 compiler	Significant
S4	gSparse compiles on Intel C++ 15.0	Non-vital
S5	gSparse compiles on Eigen 3.3.5	Critical
S6	gSparse supports CMake 3.0 build system	Non-vital

Table 3.1: Software Requirements of gSparse

3.3 Functional Requirements

The functional requirements specify what functionality gSparse will provide as a software library upon its initial release. The functional requirements are subjected to change throughout the development cycles of gSparse.

ID	Requirement	Importance
F1	gSparse implements Spectral Sparsification by Effective Resistance	Critical
F2	gSparse implements an extensible graph data type	Critical
F4	gSparse reads and writes graph edge and weight CSV data	Critical
F5	gSparse supports MPI	Significant
F6	gSparse exposes functionality to Python	Non-vital
F7	gSparse exposes functionality to MATLAB	Non-vital
F8	gSparse provides a GPU back-end to sparsify graphs	Non-vital
F9	gSparse can generate a complete graph	Critical
F3	gSparse can create a graph Laplacian	Critical
F10	gSparse can create a JL projection matrix	Critical
F11	gSparse can create a graph adjacency matrix	Critical
F12	gSparse supports undirected simple graphs	Critical
F13	gSparse supports directed simple graphs	Critical

Table 3.2: Functional Requirements of gSparse

3.4 Non-Functional Requirements

Non-functional requirements define the quality attributes that gSparse must oblige to as a software library.

ID	Requirement	Importance
N1	gSparse's Spectral Sparsification is faster than existing work	Critical
N2	gSparse does not leak any memory	Critical
N3	gSparse is thread-safe	Critical
N3	gSparse must have test cases for every functionality	Non-vital

Table 3.3: Non-Functional Requirements of gSparse

Chapter 4

Design

4.1 Introduction

In this chapter, an overview of the gSparse system is defined based on derived requirement specifications from the previous chapter. The chapter will discuss the design principle, the architecture, and key design decisions behind gSparse.

4.2 Design Principle

gSparse adopts the following five principles as its software design principal in order to make the library flexible and maintainable.

4.2.0.1 Single Responsibility Principle

In the single responsibility principle, each module should encapsulate the entire functionality provided by its class [Martin, 2008]. The service rendered by each class aligns with the class's responsibility in the system. In this case, a graph class should encapsulate the functionality of a graph representation but nothing more. A graph class that can sparsify itself or opens a CSV file would break the single responsibility principle.

4.2.0.2 Open/Closed Principle

The Open/Closed Principle states that the functionality of each class should be open for extension but closed for modification [Martin, 2008]. This principle does not imply never making changes to classes, but the existing class implementation must not alter its behavior to avoid breaking any applications that rely upon it. Modifications can be done through overloading function or object inheritance.

4.2.0.3 Liskov Substitution Principle

The Liskov Substitution Principle mainly concerns class variants. ”Functions that use pointers or references to base classes must be able to use objects of derived classes without knowing it” [Martin, 2008]. The Liskov principle suggests that the class and its children should be modeled based on behaviors instead of properties. For example, a square may have the same number of sides as a rectangle as a property, but square and rectangle behave quite differently. It makes sense to specify a width and a height of a rectangle, but it does not make sense to specify a height of a square.

4.2.0.4 Interface Segregation Principle

The Interface Segregation Principle states that subclass should not be forced to implement interfaces they don’t use [Martin, 2008]. For example, several graph sparsifier classes may share the same behavior (interface) but may leverage different algorithms that require different hyper-parameters. Enforcing hyper-parameter setting interface would break this segregation principle.

4.2.0.5 Dependency Inversion Principle

The Dependency Inversion Principle aims to decouple software module by stating that high-level modules should not depend on low-level modules; both should depend on abstractions [Martin, 2008]. This principle applies to a sizable software project with complicated inheritance hierarchies. Due to the small size of gSparse, this principle does not apply to the current state of the project. Nonetheless, as gSpare grows its functionality, this principle may become a critical consideration.

4.3 Unified Modelling Language

A Unified Modelling Language (UML) diagram is used to visualize the architecture of gSparse. The following UML class diagram features how the objects are designed and related.

4.3.1 Class Diagram

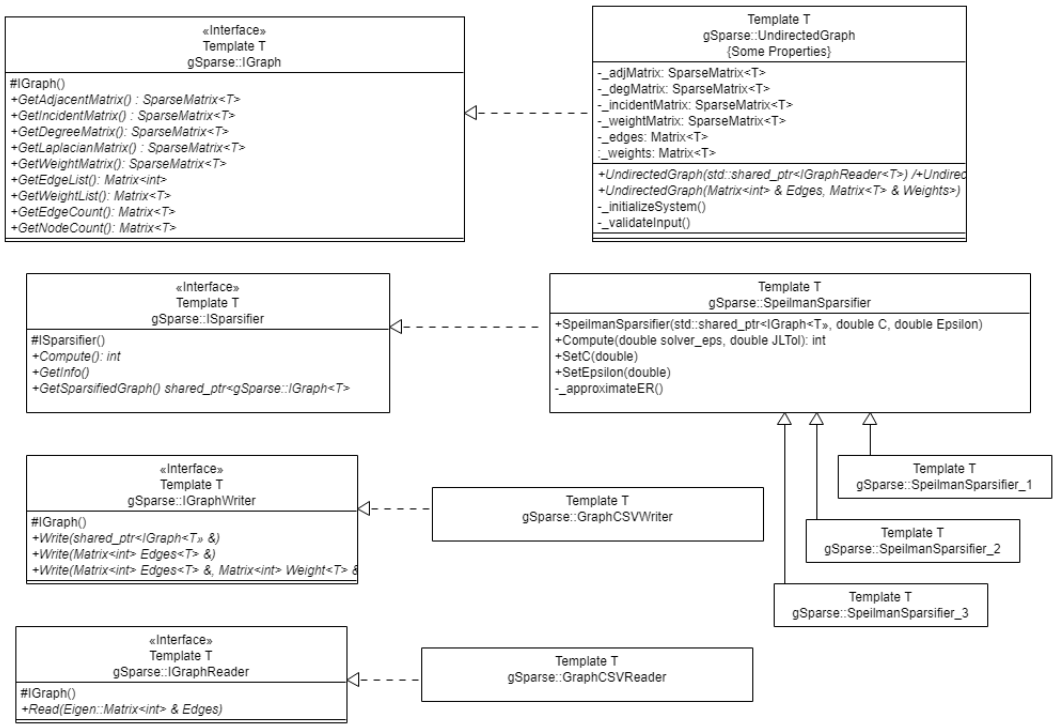

Figure 4.1: UML diagram outlining the architecture of gSparse

Figure 4.1 shows a simplified version of gSparse class diagram with necessary methods. Every core components of gSparse are derived from an interface class: IGraph, ISparsifer, IGraphWriter, and IGraphReader. In C++, the concept of interface classes does not exist, so these classes are defined as a pure abstract class. The interface classes define the interaction model between the users and gSparse. The interfaces also serve as a way to enforce conformance to the library developers not to omit any critical functionality as more features are added.

Extending the functionality of gSparse is done by deriving a new class from the base interfaces. The new class should automatically be compatible with the remaining of the system if its behaviors are properly implemented. For example, a cut-based sparsifier can be added by simply inhering the ISparsifier class and reusing the IGraph data type. A Graph can also be imported and exported to any format or source by creating a new object from IGraphReader and IGraphWriter respective.

The final key point is that every core classes of gSparse are C++ templated enable class except for Graph vertices remaining as an Integer type. At the initial release, gSparse aims to be fully compatible and tested with float and double data types.

4.4 Design Decisions and Guidelines

This section highlights some of the key design decisions and guidelines relating to the core components of gSparse, specifically, the Graph and the Sparsifier classes.

4.4.1 Graph DataType

The IGraph Data Type is an interface class detailing how a Graph data type should behave in gSparse. The Graph data type acts as a transaction material for the users to consume gSparse functionality. Although the implementation of IGraph Data Type is left for its subclass to decide, it plays a critical role in gSparse's extensibility and performance.

Ideally, the IGraph Data Type should leverage IGraphReader as a way to read Graph Data from any input source to comply with Liskov Substitution and Open/Close Principles. For performance reason, the Graph Data type should also construct all of its graph representation such as Laplacian and Adjacency Matrix at the initialization time, then save the results in separated matrices. Maintaining multiple copies of Matrix for each graph representation can make Graph Data type a constant and immutable because it is costly to make changes. More copies of Matrix representation also consume more memory. However, the key benefit of this approach is that it can significantly speed up read performance of the graph data type, an essential requirement for graph sparsification. Fortunately, Eigen3 supports the use of a Sparse Matrix which reduces a significant amount of memory usage without compromising much performance.

4.4.2 Sparsifier Object

The ISparsifier interface defines the behavior of a Graph Sparsifier. A graph sparsifier should read a graph as an input, compute any necessary information, and return a sparsifier. Referring to the UML diagram 4.1, the sparsification process is deliberately divided into two steps: Compute and GetSparsifiedGraph. The reason behind this decision is because many graph sparsification algorithms involve samplings. There are use-cases where the user might want to keep sampling a new graph without recalculating the sampling parameters for performance reasons.

gSparse initially provides an implementation of Spectral Sparsification by Effective Resistance with a class called SpeilmanSparsifier. This can quickly be done by extending the ISparsifier object. As there are potentially many implementations of this Spectral Sparsifier, such as the one that leverages specialized Laplacian solver, the alternative constructions of the algorithm can derive from SpeilmanSparisifer for fast prototyping that automatically takes advantage of existing benchmarks and test cases.

Chapter 5

Implementation

5.1 Introduction

The implementation chapter aims to highlight essential implementation consideration of gSparse and any associated challenges. The chapter begins with the implementation process, follows by implementation considerations related to specifications.

5.2 Test-Driven Development

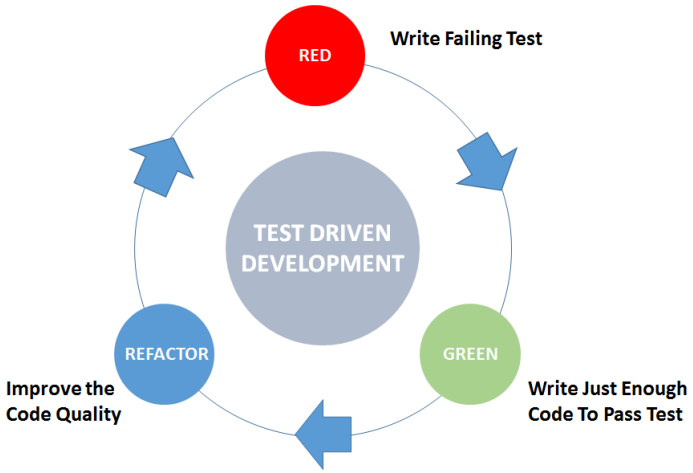

Figure 5.1: Test Driven Development Life Cycle [Bedelbaev, 2001]

The development process of gSparse adopts a Test-Driven Development process [Beck, 2003] highlighted in the figure 5.1. Catch.hpp C++ test framework is incorporated to facilitate the development cycle.

Initially, the requirements specifications are reviewed, then Catch.hpp test cases are then developed. The code repository is branched off to a feature branch with the new

test case to avoid modify existing build. Once the development of the new branch is completed and tested, the feature branch finally merges with the master branch.

5.3 Implementation Considerations

5.3.1 Performance Consideration

5.3.1.1 Unncessary Copy Operations

One of the key features that C++ provides is giving full control of how an object copies from one to another [Stroustrup, 1997]. This allows a programmer to explicitly avoid expensive and unnecessary copy operations when a subroutine is called. gSparse takes full advantage of this control where appropriate.

5.3.1.2 Memory Alignment

Although Matrix is a mathematical construct, its underlying architecture can be represented by arrays of memory addresses [Guennebaud *et al.*, 2010]. Thus, the layout of the matrix within the memory plays an important role in increasing performance. Matrix alignment divides into two categories: Row-Major and Column-Major. For Row-Major matrix, it is generally faster to access the elements within the same row before moving on to the next. It is because each element of the matrix row lies in consecutive memory addresses. The same concept applies to the column-major matrix but with column elements being in the consecutive memory instead.

In the implementation of gSparse, the majority of graph matrices are column major. This is because the default configuration of Eigen3 and C++ are to use column-major matrices. However, the matrices that store edge and weight lists are set to align in a row-major order as it is always accessed in a row order fashion.

5.3.1.3 Matrix Multiplication

In complicated Matrix Algebra, temporary objects to hold intermediate results during the calculation can significantly reduce performance. However, this is mostly outside the control of our gSparse implementation as it outsources Matrix manipulation to Eigen3. Fortunately, Eigen3 provides general guidelines in its documentation for dealing with this specific problem [Guennebaud *et al.*, 2010].

5.3.1.4 Sparse Data

Large graphs are usually quite sparse. A significant amount of performance and storage can be improved through the use of compressed Matrix. Eigen3 provides a seamless

solution to this through its Sparse Matrix data type. Eigen3's Sparse Matrix only stores the non-zero diagonal element and its associated indexes [Guennebaud *et al.*, 2010]. Eigen3 also provides a highly optimized Sparse Matrix algebra which can directly be used against a typical dense matrix [Guennebaud *et al.*, 2010].

5.3.1.5 Seperating Initialization and Computation

One of the bottlenecks identified during the implementation in Graph Sparsification is the construction of the Matrix system representing the graph. In gSparse, each representation of the same graph such as Adjacency Matrix and Laplacian Matrix are stored in multiple matrices for fast lookup time. These are constructed at the initialization time of the Graph Data Type. From an implementation standpoint, the separation of matrix system construction and graph sparsification allows the users to choose when to initiailize a graph's matrix system, potentially increase performance.

5.3.2 Linear System Solver

Implementing Effective Resistance calculation module based on Algorithm 2 requires a use of Linear System Solver. Unfortunately, after an extensive library search, none of the specialized Laplacian solvers researched in the Literature Review are available in C or C++. It is not feasible to implement another one due to the time constraint of this project. Thus, the initial version of gSparse leverages an existing but highly optimized Conjugated Gradient method with Jacobi preconditioner provided by Eigen3 to solve Laplacian systems.

5.3.2.1 Convergence issue

One of the major issues arises during the implementation of the Linear System Solver is a convergence issue. Convergence issues are common among iterative methods such as Conjugated Gradient method when the solver is unable to find a solution within a reasonable number of attempts. This comes as a surprise because of Laplacian Matrix for an undirected and simple graph, by definition, is a Symmetric Diagonally Dominant Matrix.

After extensive testing, empirical evidence suggests that the use of a random Johnson-Lindenstrauss projection to reduce the dimensionality of the Laplacian system can cause the solution not to converge. The workaround in the implementation that appears to be effective is to solve more linear systems with different random projections and aggregated the results. If the solver continues to fail, a calculation routine sets an error flag for the user to handle the scenario.

5.3.3 Parallelization

There are many parallelization opportunities for gSparse implementation, as it involves many data array aggregations. Parallel programming is time-consuming and is not part of the critical specifications. Therefore, the initial implementation of gSparse focuses on thread safety with an aim to parallelize trivial cases. Fortunately, Eigen3 seamlessly supports parallelization of Matrix multiplication and Conjugated Gradient method through compiler configuration [Guennebaud *et al.*, 2010].

5.3.3.1 Thread Safety and Random Generator

Graph Sparsification algorithms usually involve a sampling process. This requires the use of random number generators. Also, Spectral Sparsification leverages a random projection matrix. Therefore, in the initial implementation, gSparse leverages a commonly available Mersenne Twister 19937 random number generator provided by the C++ standard library [Matsumoto & Nishimura, 1998]. However, accessing the random number generator from multiple threads can lead to incorrect results due to contention issue. gSparse's workaround is to automatically create a separate Mersenne Twister engine for each running thread.

5.3.4 Resource Management

For performance reason, C++ allows full control of resource allocation and deallocation [Stroustrup, 1997]. This means the life cycle of a dynamically allocated object must be managed. gSparse follows an approach highlighted in Meyers [2005] by using Smart Pointers. A Smart Pointer is a templated class that keeps track of dynamic memory allocations. A Smart Pointer automatically deallocates resource once it detects that no reference to the memory remains.

5.3.5 Testing Stochastic Algorithm

Even though the unit test framework can ensure that the matrix system is constructed correctly, it is not sufficient to evaluate the Spectral Sparsification algorithm because it is stochastic. In this dissertation, the strategy is to build unit tests related only to the deterministic part of the algorithm. The final implementation will be evaluated against a test dataset during the experimentation phase and compare with the theoretical claims.

Chapter 6

Software Evaluation and Experiment Methodology

6.1 Introduction

This chapter begins with a requirement verification by comparing gSparse functionality against its original requirement. A memory test is also performed to validate the quality of the implementation. And finally, a benchmark dataset is introduced along the experiment setup and details to evaluate gSparse's implemetation against the existing work.

6.2 Requirement Verification

Table 6.1 highlights the requirements and its current status at the time of this dissertation. Incomplete requirements are attributed to the technical delays mostly in developing suitable solutions for the effective weight resistance routine. One of the critical requirements, support for directed graph, fails to implement due to feasibility issue relating to Graph Laplacian constructs.

ID	Importance	Status
S1	Critical	Partially Tested
S2	Critical	Completed
S3	Significant	Not Yet Implemented
S4	Non-vital	Not Yet Implemented
S5	Critical	Completed
S6	Non-vital	Not Yet Implemented
F1	Critical	Completed
F2	Critical	Completed
F4	Critical	Completed
F5	Significant	Partially Implemented
F6	Non-vital	Not Yet Implemented
F7	Non-vital	Not Yet Implemented
F8	Non-vital	Partially Implemented
F9	Critical	Completed
F3	Critical	Completed
F10	Critical	Completed
F11	Critical	Completed
F12	Critical	Completed
F13	Critical	Not Feasible
N1	Critical	Completed
N2	Critical	Completed
N3	Critical	Completed
N3	Non-vital	Completed

Table 6.1: Status of Requirements

6.3 Memory Leak Detection

Memory leak is a common issue in C++ because the language gives full control of how an object allocates and deallocates. For the gSparse project, Visual Leak Detector is attached to the Visual C++ a debugger to profile resource usage and object lifecycle. Fortunately, there's no memory leak detected in the implementation. Figure 6.1 also details memory usage from a profiler throughout an execution of a unit test. The result from the profiler also agrees with Visual Leak Detector, as resource appears to correctly allocate and deallocate.

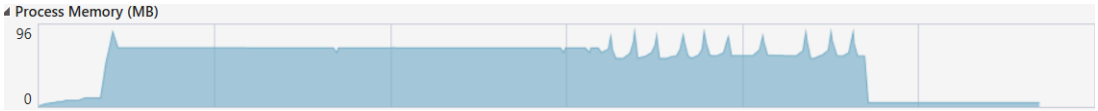

Figure 6.1: Memory Usage profile of a test gSparse application

6.4 Experiment Setup

6.4.1 Experiment Dataset

In this section, an evaluation dataset is introduced. This dataset is used to conduct benchmark and evaluation of gSparse against existing implementations. Three types of the dataset used in the experiments are the following.

6.4.1.1 Complete Graph

A complete graph is a simple undirected graph where every vertex is connected to one another by a unique edge [Weisstein, b]. Thus, the adjacency matrix for a unit complete graph is an all one matrix except the diagonal elements.

6.4.1.2 Random Graph

A random graph is constructed based on a variant of Erdős-Rényi Graph [Weisstein, h]. This random graph features three randomly generated communities with a number of edges randomly connecting the communities called bridges. The bridges of this graph should have a significantly higher effective resistance and should be preserved by the spectral Sparsifier. The code to generate this graph is attached on the Appendix section.

6.4.1.3 Social Network Graph

Lastly, a Social Network Graph is used to evaluate gSparse. Social Network Graph represents a very good evaluation dataset due to its availability and property as a simple and undirected graph. In this dissertation, Facebook friendship circle is collected from the SNAP datbase is used [Leskovec & Krevl, 2014].

6.4.1.4 Summary of Experiment Data

Table 6.3 highlight details of the dataset used for the experiments.

Graph ID	Graph Type	Node Count	Edge Count
C-1000	Completed Graph	1,000	499,500
C-2000	Completed Graph	2,000	1,999,000
C-3000	Completed Graph	3,000	4,498,500
C-4000	Completed Graph	4,000	7,998,000
C-5000	Completed Graph	5,000	12,497,500
C-6000	Completed Graph	6,000	17,997,000
C-7000	Completed Graph	7,000	24,496,500
C-8000	Completed Graph	8,000	31,996,000
C-9000	Completed Graph	9,000	40,495,500
R-3000	Random Graph with 3 communities	3,000	1,501,500
R-6000	Random Graph with 3 communities	6,000	6,000,000
R-9000	Random Graph with 3 communities	9,000	13,498,500
R-12000	Random Graph with 3 communities	12,000	23,997,000
R-15000	Random Graph with 3 communities	15,000	37,495,500
SN	Social Network Graph	4,038	88,234

Table 6.2: Details of Benchmark Data

6.4.2 Experiment Environment

The following table details the hardware used to run all of the experiments.

Category	Information
Laptop Vender	Dell Alienware 15 R2
Central Processing Unit	Intel Core i7-6700HQ CPU @2.60 GHz 4 Core(s)
Physical Memory	16 GB DDR3
Disk Storage	1 TB Solid State Drive
Graphic Processing Unit	EVGA Nividia Geforce 1070
Operating System	Windows 10 Home 64-bit Edition
BIOS Version	Alienware 1.2.8 1/29/2016

Table 6.3: Details of Benchmark Data

6.4.3 Experiment Artifacts

There are several implementation of Spectral Sparsification algorithms. This section formalizes which exact implementations are being tested.

6.4.3.1 gSparse's Single-Threaded

This is a single thread implementation of gSparse's Spectral Sparsification. The implementation follows the algorithm details in Algorithm 1 and Algorithm 2 in the Literature Review chapter. The implementation features a Conjugated Gradient Method with Jacobi preconditoner to solve the Laplacian System. The code is compiled using MSVC 14.1 64-bit with O2 optimization option.

6.4.3.2 gSparse's 8 Threads

This is the same implementation as the gSparse's Single-Threaded, but with multi-thread enabled. The code is compiled using MSVC 14.1 64-bit with Microsoft MPI enabled set to 8 threads.

6.4.3.3 gSparse's Laplacian.jil

This is a single-threaded C++ implementation in gSparse to replicates Dan Speilman's Spectral Sparsification in his Laplacian.jil Library. The algorithm feature O(m) runtime where m represents the number of edges. The key difference is that the implementation only features sampling method and does not include Dan Speilman's effective weight resistance routine, which leverages an experimental and unpublished theoretical Linear System Solver only available in Laplacian.jil. The code compiles using the same configuration as gSparse's single threaded.

6.4.3.4 FastEffectiveResistance.mat

This is a MATLAB implementation of effective resistance calculation featuring a specialized Laplacian Solver, the Combinatorial Multi-Grid (CMG). Thus, it is expected to run faster than gSparse's initial implementation. The experiment will run on an Academic version of MATLAB2018a. Unfortunately, FastEffectiveResistance.mat does not provide any routines for Spectral Sparsification.

6.4.4 Experiment Details

This section details each experiment conducted to evaluate and benchmark gSparse Graph Sparsification library.

6.4.4.1 Experiment 1: Evaluating gSparse's Spectral Sparsification

The first experiment is to evaluate gSparse Spectral Sparsification capability and to generate empirical data of the algorithm. Spectral Sparsification preserves the graph Laplacian quadratic form, which means the maximum eigenvalue obtained in the sparse

system should remain approximately within $1 \pm \varepsilon$ from the original graph. Thus, the error ratio is calculated using the following formula.

$$ErrorRatio \approx \frac{\lambda_{\max}(L_{\text{dense}} - L_{\text{sparse}})}{\lambda_{\max}(L_{\text{dense}})}$$

The experiment is evaluated over three datasets. C-1000 is the first dataset to be evaluated. The remaining dataset: a complete graph with 30 nodes and a random graph with 90 nodes are used only for visualization purposes. The goal of Experiment 1 is to visualize Spectral Sparsification and to investigate the effect of C and ε hyper-parameter highlighted in the Algorithm 1.

6.4.4.2 Experiment 2: Effective Resistance Benchmark

The second experiment is simple: compare gSparse's effective weight resistance implementations against the existing MATLAB implementation. Three implementations of Effective Weight resistance are considered: gSparse's 8 Threads, gSparse's Single-Threaded, and FastEffectiveResistance.mat. The experiment is performed ten times for each dataset and average is calculated. The execution time is measured in a global clock time. Visual C++ provides access to Windows API with functionality to measure clock time. MATLAB also provides the same functionality through its built-in timeit function.

6.4.4.3 Experiment 3: Spectral Sparsification Benchmark

The third experiment is similar to the second one. The experiment benchmark three implementations of Spectral Sparsification: gSparse's Laplacian.jil, gSparse's 8 Threads, and gSparse's Single-Threaded. Benchmark information is calculated using the same methodology as the second experiment.

6.4.4.4 Experiment 4: Sparsification of Social Network Graph

The last experiment is conducted only to illustrate the effectiveness of gSparse's Spectral Sparsification routine over a Social Network Graph.

Chapter 7

Experiment Result and Analysis

7.1 Introduction

In this section, experiment results are shown and discussed. The raw data of all the experiments can be obtained in the Appendix.

7.2 Experiment 1: Spectral Sparsification Results

7.2.1 Effect of C Hyper-parameter

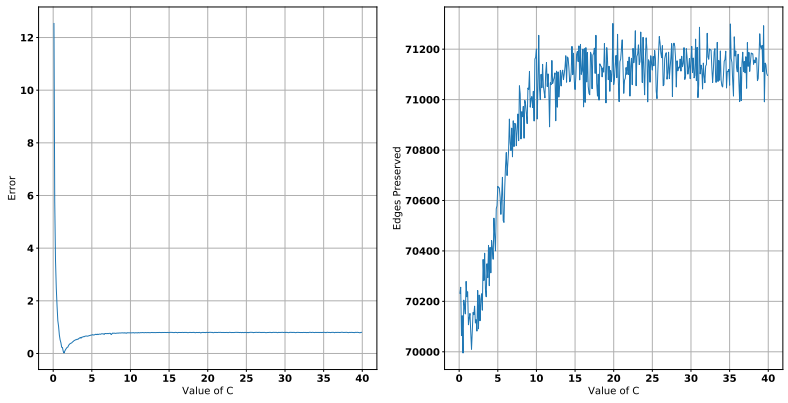

Figure 7.1: The effect of C hyper-parameter for $\varepsilon = 0.5$. The left figure compares the value of C over error ratio. The right figure compares the value of C over the number of edge in the Sparsifier.

Figure 7.1 illustrates the impact of C-hyper parameter over the C-1000 data set. The data suggests the optimal value for C is around 1.3 based on Error ratio and edges

preserved. This empirical data slightly disagree with [Spielman & Srivastava, 2008], which states that the optimal C value should be a large constant greater than 2.

7.2.2 Effect of ε Hyper-parameter

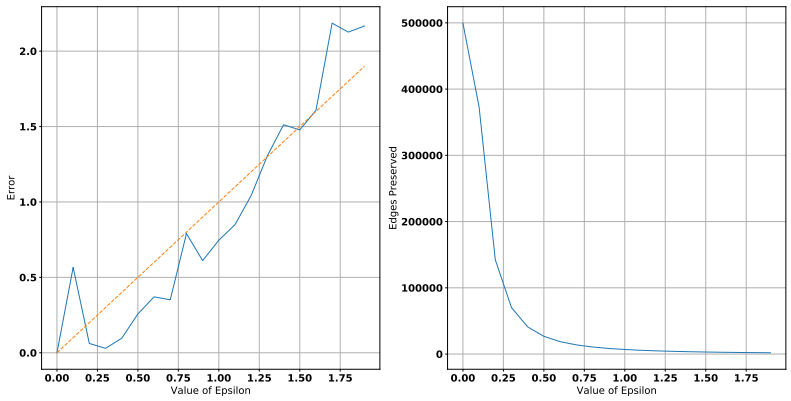

Figure 7.2: The effect of ε hyper-parameter for $C = 1.5$. The left figure compares the value ε over error ratio. The right figure compares the value of ε over the number of edge in the Sparsifier.

Figure 7.2 demonstrates the impact of ε parameter over the C-1000 data set. There appears to be a positive correlation between ε hyper-parameter and the error ratio. The results suggest that gSparse's implementation of Spectral Sparsifier agrees with [Spielman & Srivastava, 2008]. Another notable result is despite a small value of ε and a relatively large number of edges preserved, the Sparisifer sees a high error ratio. This could be explained by the fact that Spectral Sparsification is a stochastic algorithm with a high probability of success.

7.2.3 Visualization of Spectral Sparsifier

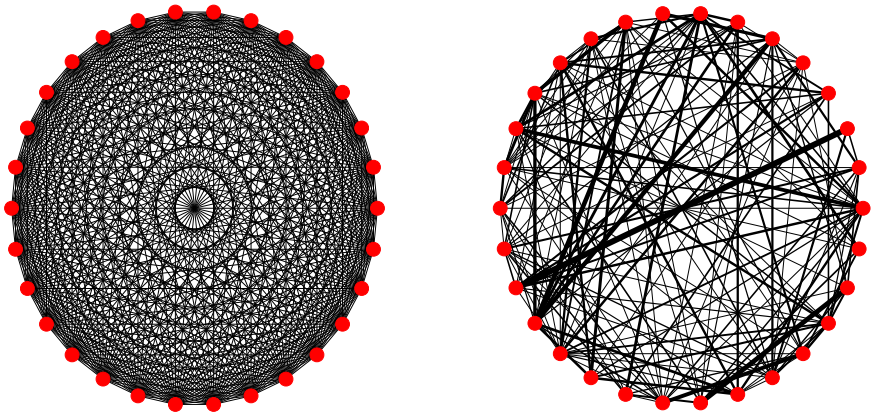

Figure 7.3: gSparse's Spectral Sparsification of a Complete Graph. The left figure shows a Complete Graph with 30 nodes. The right figure shows its Sparsifier. The line width of the edge represents relative weight value of the graph.

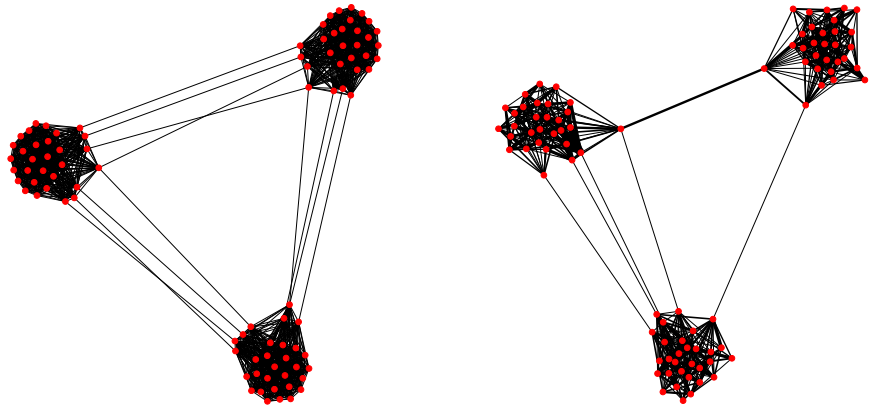

Figure 7.4: gSparse's Spectral Sparsification of a Complete Graph. The left figure shows a Random Graph with 90 nodes. The right figure shows its Sparsifier. The line width of the edge represents relative weight value of the graph.

Figure 7.3 and 7.4 visualize the effectiveness of Spectral Sparsifier. The visual results confirm that gSparse's Spectral Sparsifier is implemented correctly. Notably, for the random graph, the bridges are preserved as its effective weight resistance should be higher than those within the communities.

7.3 Experiment 2: Effective Resistance Benchmarks

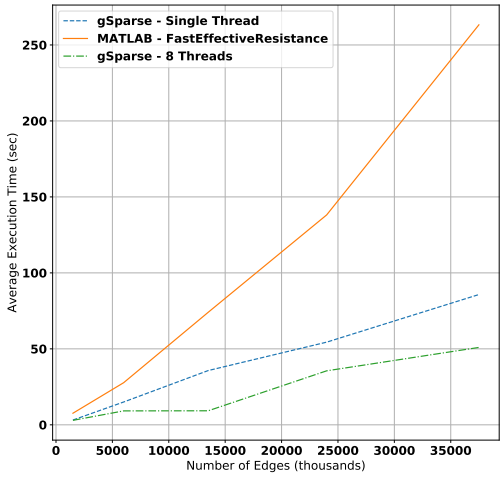

Figure 7.5: Run time cost for Effective Resistance calculations over Complete Graph datasets.

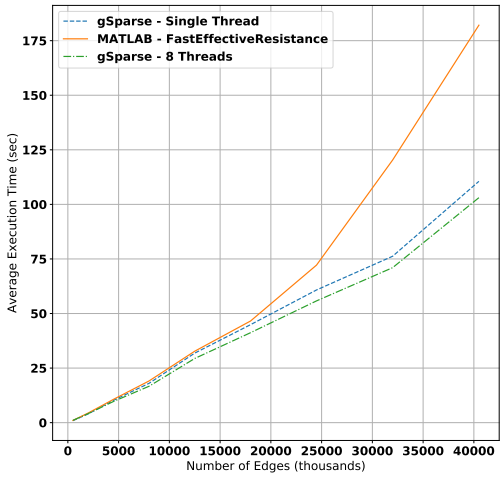

Figure 7.6: Run time cost for Effective Resistance calculations over Random Graph datasets.

Surprisingly, the MATLAB implementation is significantly slower than gSparse's C++ implementation. The superior performance of gSparse is likely due to the efficiency of C++ programming language. Nonetheless, it is worth noting that a simple but highly optimized Conjugated Gradient Method with a Jacobi preconditioner works well in solving Laplacian Systems. Another notable result is that the parallelization does not

significantly improve performance despite using eight threads. This implies that the administrative cost of setting up the multi-thread environment is significant comparing to the benefits it provides. The parallelization architecture of gSparse should be revisited.

7.4 Experiment 3: Spectral Sparsification Benchmarks

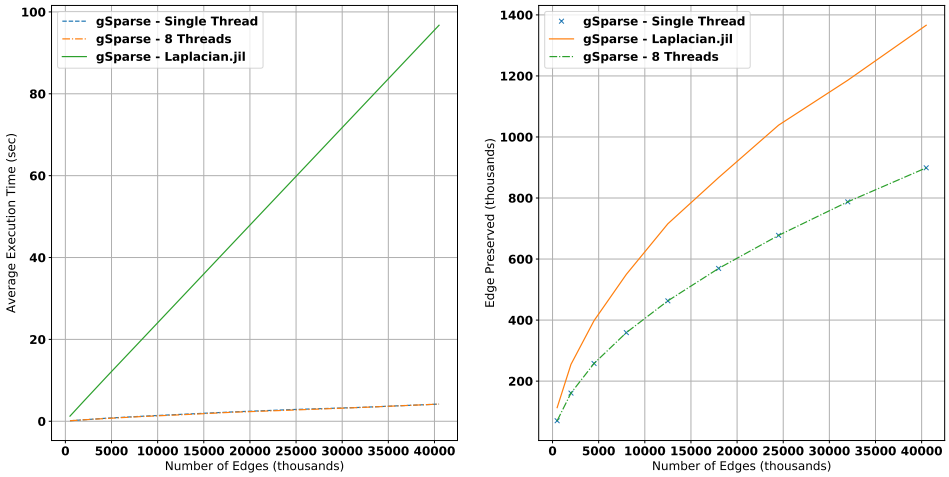

Figure 7.7: Spectral Sparsification over Complete Graph datasets with hyper-parameter: $C = 1.5, \varepsilon = 0.5$. The left figure shows the run time cost. The right figure shows the number of edge preserved.

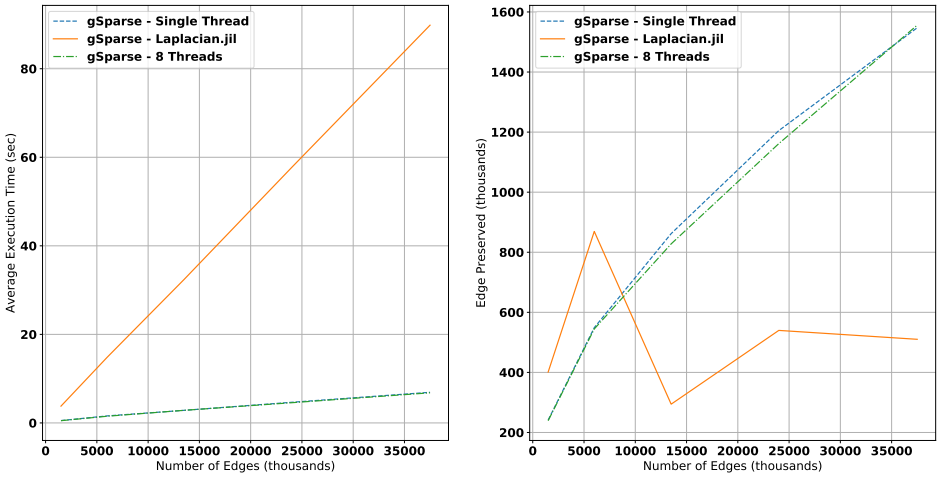

Figure 7.8: Spectral Sparsification over Random Graph datasets with hyper-parameter: $C = 1.5, \varepsilon = 0.5$. The left figure shows the run time cost. The right figure shows the number of edge preserved.

As expected, gSparse's Single-Thread and gSparse's 8 Threads algorithm which features $O(nlogn)$ time complexity runs significantly faster than gSparse's Laplacian.jil's algorithm which features $O(m)$ time complexity. The preserved edges of gSparse's Laplacian.jil implementation appears to be inconsistent. This suggests Laplacian.jil library may contain an incorrect implementation of Spectral Sparisifier. A fix should be proposed to the author of the library.

7.5 Experiment 4: Sparsification of Social Network Graph

Figure 7.9: Spectral Sparsification of Facebook social network data. The left figure shows the original graph. The right figure shows its Sparsifier.

The result shows the effectiveness of gSparse's Spectral Sparsification algorithm over the SN dataset. Sparsifying the SN dataset requires a negligible amount of processing time (less than 0.01 second) in the experiment machine. The Sparsifier preserved 43,290 edges out of 88,234. The acceptable performance of gSparse suggests that gSparse may soon be suitable for future practical applications.

Chapter 8

Conclusion

8.1 Introduction

In the concluding chapter, key findings from each section of the dissertation are outlined before the project reflection. Some of the key findings can be identified as a potential area of future work and research.

8.2 Overview

In the first chapter of this dissertation, an introduction is given to explain why Graph Sparsification is important. And there is a need for building an extensible open-source library for Graph Sparsification. Due to its popularity, Spectral Sparsification by Effective Weight resistance is chosen as an initial algorithm to implement.

To build a Graph Sparsification library, a literature review was conducted in both academic and industrial space to identify any existing work and the best approaches to complete the project. The research identified two existing implementations relating to Spectral Sparsification. One is a MATLAB module implemented by Gary Miller and Richard Peng to calculate effective weight resistance and the other is a full implementation of Spectral Sparsification within a Julia package developed by Daniel Speilman.

Once the problem domain is precisely understood, the construction of the library formally started. The development of the library was highlighted in Chapter 3, 4, and 5. The result is a construction of a header-only C++ Library called gSparse. gSparse leverages Eigen3 for Matrix computation to facilitate its Graph Sparsification algorithms. gSparse features its own flexible Graph Data Type and an extensible Graph Sparsifier class.

In Chapter 6 and 7, the quality of gSparse was evaluated. Benchmark was performed on gSparse against other existing implementations on a benchmark dataset. The result shows that gSparse is performing in accordance with what is claimed in the litera-

ture and also running significantly faster than existing implementations despite using a relatively simple Linear System Solver.

Overall, the development of gSparse successfully sets a solid foundation in bringing Graph Sparsification algorithms into practice with a promising empirical evidence.

8.3 Future Work

8.3.1 Laplacian System Solver

The next logical step for gSparse is to implement specialized Laplacian Solvers in C++ such as those featured in the existing implementations. More literature reviews should be performed because some of these solvers involve many advanced concepts.

8.3.2 Benchmark

gSparse was benchmarked on Complete and Random graphs. More benchmarks should be performed in order to claim the significant of gSparse library.

8.3.3 Extending Functionality

At the current state, the functionality of gSparse is quite limited. Adding more algorithms and extending compiler supports should increase the exposure and usability of gSparse.

8.3.4 Parallelization and GPU Support

gSparse does not appear to take full advantage of parallelization. There are many data aggregations in graph sparsification algorithm which can benefit from a parallel reduction. Future work should revisit parallelization architecture of gSparse. Additionally, an experimental build of gSparse is working towards outsourcing computation to GPU devices. The work in this area could potentially increase the performance of gSparse.

Bibliography

Nearly-linear time algorithms for graph partitioning, graph sparsification, and solving linear systems. *CoRR*, cs.DS/0310051, 2003. URL http://arxiv.org/abs/cs.DS/0310051. Withdrawn.

Z. Allen Zhu, Z. Liao, and L. Orecchia. Spectral sparsification and regret minimization beyond matrix multiplicative updates. *CoRR*, abs/1506.04838, 2015. URL http://arxiv.org/abs/1506.04838.

Armadillo. Armadillo c++ library for linear algebra scientific computing, 2018–. URL http://arma.sourceforge.net/. [Online; accessed ¡today¿].

K. Beck. *Test-driven development : by example*. Addison-Wesley, Boston, 2003. ISBN 978-0-321-14653-3.

Z. Bedelbaev. Tdd example, 2001. URL https://github.com/zismailov/TDD_example/. [Online; accessed ¡today¿].

A. A. Benczúr and D. R. Karger. Approximating s-t minimum cuts in Õ(n2) time. In *Proceedings of the Twenty-eighth Annual ACM Symposium on Theory of Computing*, STOC '96, pages 47–55, New York, NY, USA, 1996. ACM. ISBN 0-89791-785-5. doi: 10.1145/237814.237827. URL http://doi.acm.org/10.1145/237814.237827.

Boost. Boost basic linear algebra, 2008–. URL https://www.boost.org/doc/libs/1_66_0/libs/numeric/ublas/doc/index.html. [Online; accessed ¡today¿].

Catch2. A modern, c++-native, header-only, test framework for unit-tests, tdd and bd, 2018. URL https://github.com/catchorg/Catch2. [Online; accessed ¡today¿].

P. Chew. There is a planar graph almost as good as the complete graph. In *Proceedings of the Second Annual Symposium on Computational Geometry*, SCG '86, pages 169–177, New York, NY, USA, 1986. ACM. ISBN 0-89791-194-6. doi: 10.1145/10515.10534. URL http://doi.acm.org/10.1145/10515.10534.

T. Chu, Y. Gao, R. Peng, S. Sachdeva, S. Sawlani, and J. Wang. Graph sparsification, spectral sketches, and faster resistance computation, via short cycle decompositions. *CoRR*, abs/1805.12051, 2018. URL http://arxiv.org/abs/1805.12051.

T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to algorithms*. The MIT press, 1990.

Google. Computation using data flow graphs for scalable machine learning, 2018–. URL https://github.com/tensorflow/tensorflow. [Online; accessed ¡today¿].

G. Guennebaud, B. Jacob, *et al*. Eigen v3. http://eigen.tuxfamily.org, 2010.

E. Jones, T. Oliphant, P. Peterson, *et al*. SciPy: Open source scientific tools for Python, 2001–. URL http://www.scipy.org/. [Online; accessed ¡today¿].

I. Koutis, G. L. Miller, and D. Tolliver. Combinatorial preconditioners and multilevel solvers for problems in computer vision and image processing. *Comput. Vis. Image Underst.*, 115(12):1638–1646, December 2011. ISSN 1077-3142. doi: 10.1016/j.cviu.2011.05.013. URL http://dx.doi.org/10.1016/j.cviu.2011.05.013.

I. Koutis, A. Levin, and R. Peng. Faster spectral sparsification and numerical algorithms for SDD matrices. *CoRR*, abs/1209.5821, 2012. URL http://arxiv.org/abs/1209.5821.

Y. T. Lee and H. Sun. An sdp-based algorithm for linear-sized spectral sparsification. *CoRR*, abs/1702.08415, 2017. URL http://arxiv.org/abs/1702.08415.

J. Leskovec and A. Krevl. SNAP Datasets: Stanford large network dataset collection. http://snap.stanford.edu/data, June 2014.

G. Lindner, C. Staudt, M. Hamann, H. Meyerhenke, and D. Wagner. Structure-preserving sparsification of social networks. *2015 IEEE/ACM International Conference on Advances in Social Networks Analysis and Mining (ASONAM)*, pages 448–454, 2015.

O. E. Livne and A. Brandt. Lean algebraic multigrid (lamg): Fast graph laplacian linear solver. *SIAM J. Scientific Computing*, 34(4), 2012. URL http://dblp.uni-trier.de/db/journals/siamsc/siamsc34.html#LivneB12.

R. C. Martin. *Clean Code: A Handbook of Agile Software Craftsmanship*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 1 edition, 2008. ISBN 0132350882, 9780132350884.

M. Matsumoto and T. Nishimura. Mersenne twister: A 623-dimensionally equidistributed uniform pseudo-random number generator. *ACM Trans. Model. Comput. Simul.*, 8(1):3–30, January 1998. ISSN 1049-3301. doi: 10.1145/272991.272995. URL http://doi.acm.org/10.1145/272991.272995.

S. Meyers. *Effective C++: 55 Specific Ways to Improve Your Programs and Designs (3rd Edition)*. Addison-Wesley Professional, 2005. ISBN 0321334876.

D. Speilman. Laplacians.jl graph library. https://github.com/danspielman/Laplacians.jl.

D. A. Spielman and N. Srivastava. Graph sparsification by effective resistances. *CoRR*, abs/0803.0929, 2008. URL http://arxiv.org/abs/0803.0929.

D. A. Spielman and S. Teng. Spectral sparsification of graphs. *CoRR*, abs/0808.4134, 2008. URL http://arxiv.org/abs/0808.4134.

B. Stroustrup. *The C++ programming language*. Addison-Wesley, 3rd edition, 1997. ISBN 978-0-201-88954-3.

O. Travis E. A guide to numpy, 2006–. URL http://www.numpy.org/. [Online; accessed ¡today¿].

V. S. S. Vos. Methods for determining the eective resistance, 2016.

E. W. Weisstein. Adjacency matrix, a. URL http://mathworld.wolfram.com/AdjacencyMatrix.html. Visited on 8/01/18.

E. W. Weisstein. Complete graph, b. URL http://mathworld.wolfram.com/CompleteGraph.html. Visited on 8/01/18.

E. W. Weisstein. Degree matrix, c. URL http://mathworld.wolfram.com/DegreeMatrix.html. Visited on 8/01/18.

E. W. Weisstein. Directed graph, d. URL http://mathworld.wolfram.com/DirectedGraph.html. Visited on 8/01/18.

E. W. Weisstein. Graph, e. URL http://mathworld.wolfram.com/Graph.html. Visited on 8/01/18.

E. W. Weisstein. Incidence matrix, f. URL http://mathworld.wolfram.com/IncidenceMatrix.html. Visited on 8/01/18.

E. W. Weisstein. Laplacian matrix, g. URL http://mathworld.wolfram.com/LaplacianMatrix.html. Visited on 8/01/18.

E. W. Weisstein. Random graph, h. URL http://mathworld.wolfram.com/RandomGraph.html. Visited on 8/01/18.

E. W. Weisstein. Undirected graph, i. URL http://mathworld.wolfram.com/UndirectedGraph.html. Visited on 8/01/18.

M. Wilson. *Imperfect C: practical solutions for real-life programming*. Addison-Wesley, 2005.

Appendices

Appendix A

Laplacian.jil/sparsify.jil

This appendix chapter captures the existing implementation of Spectral Sparsification in the Laplacian.jil open source library [Speilman].

```
#=
Code for Sparsification.
Right now, just implements Spielman-Srivastava
=#

"""
as = sparsify(a; ep=0.5)
Apply Spielman-Srivastava sparsification:
sampling by effective resistances.
`ep` should be less than 1.
"""
function sparsify(a; ep=0.3, matrixConcConst=4.0, JLfac=4.0)

  if ep > 1
    @warn "Calling sparsify with ep > 1 can produce a
    disconnected graph."
  end

  f = approxCholLap(a,tol=1e-2);

  n = size(a,1)
  k = round(Int, JLfac*log(n)) # number of dims for JL

  U = wtedEdgeVertexMat(a)
  m = size(U,1)
  R = randn(Float64, m,k)
  UR = U'*R;

  V = zeros(n,k)
  for i in 1:k
```

```
    V[:,i] = f(UR[:,i])
  end

  (ai,aj,av) = findnz(triu(a))
  prs = zeros(size(av))
  for h in 1:length(av)
      i = ai[h]
      j = aj[h]
      prs[h] = min(1,av[h]* (norm(V[i,:]-V[j,:])^2/k) *
      matrixConcConst * log(n)/ep^2)
  end

  ind = rand(Float64,size(prs)) .< prs

  as = sparse(ai[ind],aj[ind],av[ind]./prs[ind],n,n)
  as = as + as'

  return as

end
```

Appendix B

FastEffectiveResistance.mat

This appendix chapter captures the existing implementation of Effective Resistance calculation in MATLAB [Koutis *et al.*, 2012].

```
function [eff_res,Z] = EffectiveResistances(elist,e,w,tol,epsilon,
    ↪ type_,pfun_)
% Function [eff_res,Z] = EffectiveResistances(e,w,tol,epsilon,type_
    ↪ ,pfun_)
% calculate the effective resistance (ER) in a given graph, the
    ↪ graph
% is view as an electrical network in where the weight of the edges
% in the graph are the capacitances.
%
%
% Input:
% elist: list of the effective resistance to compute (edges) [K x
    ↪ 2]
% e: edges of the graph [M x 2]
% w: weights in the graph [M x 1]
% [tol]: optional string, specify the tolerance of the solver.
% 1e-4 <default>
% [epsilon]: optional string, control the number of systems that
    ↪ solve
% 1 <default>
% [type_]: optional string
% 'slm' <default>: this version use less memory.
% 'spl': this is the exact version of the Spielman Sirvastava
% 2008 paper.
%
% 'org': find the exact effective resistances.
% [pfun_]: cmg_sdd(L), if already computed.
%
% Output:
% eff_res: the effective resistance between v and u
```

```
% Z: the [s x N] system described in Spielman Sirvastava 2008.
% Available only with the 'spl' version.
%
% User Recomendations:
% 1) Use 'slm' version when the elist is know and there are no
% more ER to compute.
% 2) Use 'slm' version when the graph is bigger relative to the
% amount of memory in the machine (RAM).
% 3) Use 'spl' versioin when the elist is not know a future ER
% computations are needed. For progressive ER computations.
% See Example 2.
% 4) Using the 'spl' version in machines with small amount of RAM
% may hang the system due to RAM demand.
%
% Example 1: The path graph.
% E = [(1:49)' (2:50)']; w = ones(length(E),1); elist = [1 50];
% [er] = EffectiveResistances(elist,E,w,1e-5,1,'org')
%
% Example 2: The path graph alternative version.
% E = [(1:49)' (2:50)']; w = ones(length(E),1); elist = [1 50];
% [er,Z] = EffectiveResistances(elist,E,w,1e-5,1,'spl');
% % Compute other ER.
% o_elist = [2 49;3 48];
% o_er = sum(((Z(:,o_elist(:,1))-Z(:,o_elist(:,2))).^2),1)';
%
% Richard Garcia-Lebron
%
   %% input Validation
   [m,two] = size(e);
   [~,two1] = size(elist);
   % Check input
   if nargin > 7
      error('Too many arguments, see help EffectiveResistancesJL
         ↪ .');
   elseif nargin < 3
      error('More arguments needed, see help
         ↪ EffectiveResistancesJL.');
   elseif nargin == 5
      type_ = 'slm';
   elseif nargin == 4
      type_ = 'slm';
      tol = 10^-4; %tolerance for the cmg solver
      epsilon = 1;
   elseif nargin == 3
      type_ = 'slm';
      tol = 10^-4; %tolerance for the cmg solver
```

```
    epsilon = 1;
elseif two ~= 2 || two1 ~= 2
    estring = ['The first or the second input have wrong' ...
               ' column dimension should be [M x 2].'];
    error(estring);
end
% Check output
if nargout > 2 || nargout < 1
    error('Wrong number of outputs: see help
       ↪ EffectiveResistances.');
elseif nargout == 2 && strcmp(type_,'slm')
    error('Second output not available with this version: see
       ↪ help EffectiveResistances.');
elseif nargout == 2 && strcmp(type_,'org')
    error('Second output not available with this version: see
       ↪ help EffectiveResistances.');
end

%% Creating data for effective resitances
numIterations = 300; %iteration for the cmg solver
tolProb = 0.5;%use to create the johnson lindestraus projection
    ↪  matrix
n = max(max(e));%number of node in the graph
A = sparse([e(:,1);e(:,2)],[e(:,2);e(:,1)],[w;w],n,n); %
    ↪ adjacency matrix
L = diag(sum(abs(A),2)) - A; %laplacian matrix
clear 'A' %adjacensy matrix no needed
B = sparse([1:m 1:m],[e(:,1) e(:,2)],[ones(m,1) -1*ones(m,1)]);
    ↪  % bidirectional incident matrix?
[m,n] = size(B);
W = sparse(1:length(w),1:length(w),w.^(1/2),length(w),length(w)
    ↪ ); %
scale = ceil(log2(n))/epsilon;
%% Finding the effective resitances
if strcmp(type_,'slm')
    if nargin == 7 % Creating the preconditioned function
        pfun = pfun_;
    elseif (length(L) > 600)
        pfun = cmg_sdd(L);
    end
    [rows,~,~] = size(elist);
    eff_res = zeros(1,rows);
    eff_res1 = cell(1,rows);
    optimset('display','off');
    if (length(L) > 600) % bigger graphs
        for j=1:scale
```

```
            ons = (rand(1,m) > tolProb);
            ons = ons - not(ons);
            ons = ons./sqrt(scale);
            [Z_ flag]= pcg(L,(ons*(W)*B)',tol,numIterations,pfun)
               ↪ ;
            if flag > 0
               error(['PCG FLAG: ' num2str(flag)])
            end
            Z_ = Z_';
            eff_res1{j} = (abs((Z_(elist(:,1))-Z_(elist(:,2))))
               ↪ .^2)';
         end
         eff_res = sum(cell2mat(eff_res1),2);
         Z = Inf;
      else % smaller graphs
         for j=1:scale
            ons = (rand(1,m) > tolProb);
            ons = ons - not(ons);
            ons = ons./sqrt(scale);
            [Z flag]= pcg(L,(ons*(W)*B)',tol,numIterations);
            if flag > 0
               error(['PCG FLAG: ' num2str(flag)])
            end
            Z = Z';
            eff_res = eff_res + (abs((Z(elist(:,1))-Z(elist(:,2))
               ↪ )).^2); % the second "loop"
         end
      end
      eff_res = eff_res';
   elseif strcmp(type_,'spl')
      Q = sparse((rand(scale,m)) > tolProb);
      Q = Q - not(Q);
      Q = Q./sqrt(scale);
      SYS = sparse(Q*(W)*B); % Creation the system
      if nargin == 7 % Creating the preconditioned function
         pfun = pfun_;
      elseif (length(L) > 600)
         pfun = cmg_sdd(L);
      end
      optimset('display','off');
      %% Solving the systems
      if (length(L) > 600) % bigger graphs
         for j=1:scale
            [Z(j,:) flag] = pcg(L,SYS(j,:)',tol,numIterations,
               ↪ pfun);
            if flag > 0
```

```
                error(['PCG FLAG: ' num2str(flag)])
            end
        end
    else % smaller graphs
        for j=1:scale
            [Z(j,:) flag] = pcg(L,SYS(j,:)',tol,numIterations);
            if flag > 0
                error(['PCG FLAG: ' num2str(flag)])
            end
        end
    end
    eff_res = sum(((Z(:,elist(:,1))-Z(:,elist(:,2))).^2),1)';

elseif strcmp(type_,'org')
    [m,~] = size(elist);
    B = sparse([1:m 1:m],[elist(:,1) elist(:,2)],[ones(m,1) -1*
        ↪ ones(m,1)],m,length(L));
        if length(L) > 600 % bigger graphs
            if nargin == 7
                pfun = pfun_;
            else
                pfun = cmg_sdd(L);
            end
            optimset('display','off');
            for j=1:m
                [Z flag]= pcg(L,B(j,:)',1e-10,numIterations,pfun)
                    ↪ ;
                if flag > 0
                    error(['PCG FLAG: ' num2str(flag)])
                end
                eff_res(j) = B(j,:)*Z;
            end
        else % smaller graphs
            opts.type = 'nofill'; opts.michol = 'on';
            for j=1:m
                [Z,flag] = pcg(L,B(j,:)',1e-10,numIterations);
                eff_res(j) = B(j,:)*Z;
            end
        end
    eff_res = eff_res';
end
end
```

Appendix C

Random Graph Generator

This appendix chapter details the implementation of random graph generator written in C++.

```
void buildRandomGraph(int total_community = 3, int bridges = 4, int
    ↪ community_size = 30)
{
        std::default_random_engine generator;

        for (int i = 1; i != 2; ++i)
        {
                stringstream ss;
                ss << "random-" << i * community_size << ".csv";
                ofstream myfile;
                myfile.open(ss.str());

                //Each community is currently a complete graph to
                    ↪ represent a dense graph. This can be sparsify
                    ↪ safety through uniform sampling.
                std::shared_ptr<gSparse::Graph> g = gSparse::Util::
                    ↪ buildUnitCompletedGraph(i * community_size);

                //Connecting the bridges via uniform sampling
                int offset = g->GetNodeCount();
                std::uniform_int_distribution<int> distribution(0,
                    ↪ offset);

                for (int community_id = 0; community_id !=
                    ↪ total_community; ++community_id)
                {
                        stringstream ss;
                        // Build communities of complete graph
                        myfile << g->GetEdgeList().array() + offset *
```

```
                        ↪ community_id << endl;
                }
                // connect the communities
                for (int j = 0; j != total_community; ++j)
                {
                        // Connect the graphs
                        for (int k = 0; k != total_community; ++k)
                        {
                                if (k > j)
                                {
                                        for (int l = 0; l != bridges; ++
                                            ↪ l)
                                        {
                                                myfile << j * offset +
                                                    ↪ distribution(
                                                    ↪ generator) << " "
                                                    ↪ << k * offset +
                                                    ↪ distribution(
                                                    ↪ generator) << endl
                                                    ↪ ;
                                        }
                                }
                        }
                }
                myfile.close();
        }
}
```

Appendix D

Raw Experiment Data

Graph ID	EffectiveResistance.mat	gSparse - Single Thread	gSparse - 8 Threads
C-1000	1.015 ± 0.005	0.897 ± 0.030	1.063 ± 0.0148
C-2000	4.408 ± 0.015	4.011 ± 0.068	4.122 ± 0.0379
C-3000	10.686 ± 0.014	10.234±0.118	9.708 ± 0.02444
C-4000	19.099 ± 0.024	18.015±0.2379	16.671 ± 0.03430
C-5000	32.724 ± 0.437	31.847± 0.314	29.395 ± 0.08072
C-6000	46.563 ± 1.438	44.887± 0.2466	41.209 ± 0.063406
C-7000	72.168 ± 2.082	60.784 ± 0.439	55.780 ± 0.35702
C-8000	120.314 ± 1.569	76.235 ± 0.211	71.003 ± 0.343
C-9000	182.101 ± 2.760	110.561 ± 0.453	103.069 ± 0.149
R-3000	7.63 ± 0.02	3.149 ± 0.533	2.948 ± 0.002
R-6000	27.731 ± 1.621	15.023 ± 0.912	9.170 ± 0.051
R-9000	74.077 ± 0.451	35.723± 0.214	9.244 ± 1.031
R-12000	138.134 ± 1.823	54.409 ± 0.291	35.536 ± 0.253
R-15000	263.249 ± 3.132	85.739 ± 0.341	50.933 ± 0.139

Table D.1: Effective Resistance Runtime

Graph ID	gSparse - Laplacian.jil	gSparse - Single Thread	gSparse - 8 Threads
C-1000	113,107.8 ± 279.096	70,165.5 ± 64.708	70,142.8 ± 62.745
C-2000	254,954.2 ± 531.070	160,792 ± 117.518	160,798.9 ± 45.191
C-3000	398,191 ± 532.437	257,887.8 ± 138.515	25,7875.7 ± 106.092
C-4000	549,825.3 ± 607.348	358,898.6 ± 86.911	358,939.4 ± 89.430
C-5000	715,563.9 ± 461.097	463,004.4 ± 88.783	463,039.9 ± 114.8
C-6000	866,923.2 ± 623.395	569,266.9 ± 121.439	569,321.3 ± 91.285
C-7000	1,038,661.3 ± 787.265	677,626.3 ± 88.989	677,581.4 ± 110.038
C-8000	1,185,931.7 ± 916.388	787,406.1 ± 89.417	787,532.4 ± 121.110
C-9000	1,366,110.7 ± 973.842	898,959.7 ± 128.668	898,899.7 ± 154.451
R-3000	401,159.9 ± 572.952	239,246.1 ± 139.264	241,345.4 ± 99.789
R-6000	869,273.3 ± 751.231	544,688.6 ± 165.938	548,949.4 ± 148.623
R-9000	294,233.8 ± 497.670	827,808.8 ± 131.823	861,982.6 ± 205.305
R-12000	539,802.8 ± 503.067	1,161,687.7 ± 359.405	1,204,969.2 ± 299.460
R-15000	510,194.5 ± 614.203	1,555,974.5 ± 219.389	1,547,344.1 ± 233.932

Table D.2: Spectral Sparsification: Edge Preserved

Graph ID	gSparse - Laplacian.jil	gSparse - Single Thread	gSparse - 8 Threads
C-1000	1.249 ± 0.007	0.126 ± 0.008	0.124 ± 0.005
C-2000	4.917 ± 0.0218	0.360 ± 0.004	0.352 ± 0.0033
C-3000	10.952 ± 0.014	0.749 ± 0.053	0.688 ± 0.007
C-4000	19.285 ± 0.062	1.185 ± 0.027	1.119 ± 0.003
C-5000	30.013 ± 0.015	1.683 ± 0.034	1.579 ± 0.009
C-6000	43.141 ± 0.025	2.265 ± 0.022	2.190 ± 0.011
C-7000	58.626 ± 0.018	2.851 ± 0.029	2.722 ± 0.013
C-8000	76.497 ± 0.030	3.389 ± 0.017	3.365 ± 0.009
C-9000	96.745 ± 0.055	4.193 ± 0.021	4.190 ± 0.055
R-3000	3.792 ± 0.007	0.535 ± 0.013	0.505 ± 0.005
R-6000	14.812 ± 0.017	1.584 ± 0.030	1.509 ± 0.007
R-9000	32.382 ± 0.028	2.842 ± 0.011	2.842 ± 0.0136
R-12000	57.689 ± 0.146	4.662 ± 0.072	4.562 ± 0.036
R-15000	89.836 ± 0.046	6.919 ± 0.341	6.800 ± 0.119

Table D.3: Spectral Sparsification: Runtime