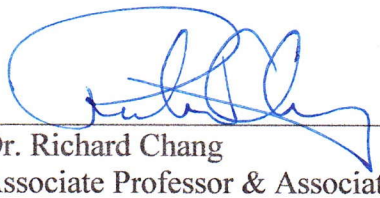
APPROVAL SHEET

Title of Thesis: An Empirical Study of Expander Graphs and Graph Expansion

Name of Candidate: Mark Lotts
Master of Science, 2016

Thesis and Abstract Approved: 

Dr. Richard Chang
Associate Professor & Associate Chair
Department of Computer Science & Electrical Engineering

Date Approved: 4/26/2016

ABSTRACT

Title of thesis: AN EMPIRICAL STUDY OF EXPANDER GRAPHS AND GRAPH EXPANSION

Mark Allen Lotts, Master of Science, 2016

Thesis directed by: Professor Richard Chang
Department of Computer Science and Electrical Engineering

Expander graphs are commonly studied objects in computer science and mathematics that are found in the proofs of many important theorems. The vast majority of these theoretical uses of expanders rely on probabilistic statements of existence and do not grapple with the challenge of creating expander graphs or validating their expansion properties. In this paper, we will define expander graphs and describe different ways their expansion can be measured. We will discuss applications of expander graphs and provide empirical evidence of how they can be used in practice. We will also outline the difficulties of computing exact expansion rates and the hardness of estimating these rates, relating these problems to well-known results and conjectures in complexity theory. Using our own implementation of graph creation and verification algorithms, we will gain an empirical understanding of expander graphs, utilizing high-performance computing resources and repurposing well-known statistical methods to analyze expansion. We will show that, given an arbitrary graph, its potential to be used as an expander can be measured and bounded by employing community detection algorithms that seek to maximize modularity.

AN EMPIRICAL STUDY OF EXPANDER GRAPHS AND GRAPH EXPANSION

by

Mark Allen Lotts

Thesis submitted to the Faculty of the Graduate School of the
University of Maryland, Baltimore County in partial fulfillment
of the requirements for the degree of
Master of Science
2016

Advisory Committee:
Professor Richard Chang, Chair/Advisor
Professor Charles Nicholas
Professor Samuel Lomonaco

Acknowledgments

The author would like to thank his advisor, Dr. Richard Chang, for his guidance, support, and advice. The author would also like to thank his friends and family for their support and encouragement.

The hardware used in the computational studies is part of the UMBC High Performance Computing Facility (HPCF). The facility is supported by the U.S. National Science Foundation through the MRI program (grant nos. CNS–0821258 and CNS–1228778) and the SCREMS program (grant no. DMS–0821311), with additional substantial support from the University of Maryland, Baltimore County (UMBC). See `hpcf.umbc.edu` for more information on HPCF and the projects using its resources.

Table of Contents

List of Tables v

List of Abbreviations vii

1 Introduction 1
- 1.1 History . . . 1
- 1.2 In Practice . . . 2
- 1.3 Purpose . . . 3
- 1.4 Organization . . . 4

2 Expander Graph Basics 6
- 2.1 Definitions . . . 6
- 2.2 The Second Eigenvalue . . . 8
- 2.3 Random Walks on Expander Graphs . . . 10

3 Expander Graph Applications 14
- 3.1 Probabilistic Amplification . . . 14
  - 3.1.1 Background and Motivation . . . 15
- 3.2 Cuckoo Hashing . . . 16
  - 3.2.1 Definition and Background . . . 17
  - 3.2.2 Random Walk Cuckoo Hashing . . . 19

4 Calculating and Bounding Expansion 21

5 Hardness of Approximating Graph Expansion 27
- 5.1 Unique Games Conjecture . . . 28
- 5.2 Approximating Small Set Expansion . . . 30
- 5.3 Unique Games, Small Set Expansion, and Graph Expansion . . . 31

6 Algorithms for Approximating Graph Expansion 33
- 6.1 Cheeger's Inequality and the Second Eigenvalue . . . 33
- 6.2 `SparsestCut` Relaxation . . . 34

7 Graph Modularity 38
- 7.1 Definition and Motivation . . . 39
- 7.2 Algorithms . . . 41
- 7.3 Relationship with Graph Expansion . . . 44

8 Generating Expander Graphs 47
- 8.1 Random Generation . . . 47
- 8.2 Explicit Generation . . . 50
  - 8.2.1 Early Work . . . 50
  - 8.2.2 The Zig-Zag Graph Product: Preliminaries . . . 53
  - 8.2.3 The Zig-Zag Graph Product: Definition and Recursion . . . 55

8.2.4 The Zig-Zag Graph Product: In Practice . . . 57
8.3 Expander Construction Summary . . . 58

9 Graph Generation, Eigenvalue Testing, and Expansion Evaluation 60
9.1 Graph Generation and Evaluation . . . 60
9.1.1 Data . . . 63
9.1.2 Analysis . . . 63
9.2 Vertex Expansion Testing . . . 65
9.3 Subset Testing . . . 69
9.3.1 Expansion Evaluation Methodology . . . 71
9.3.2 Data . . . 72
9.3.3 Singleton Subset Testing Analysis . . . 75
9.3.4 Subset Sampling Testing Analysis . . . 77
9.4 Conclusion . . . 79

10 Graph Modularity Testing and Results 81
10.1 Methodology . . . 81
10.2 Data . . . 89
10.3 Results . . . 90

11 Expander Graph Application Testing and Results 95
11.1 Probabilistic Amplification . . . 95
11.1.1 Testing Methodology . . . 95
11.1.2 Data . . . 98
11.1.3 Results . . . 98
11.2 Cuckoo Hashing and Expander Graphs . . . 101

12 Conclusions 105

13 Future Work 108

A Tables and Charts 110

Bibliography 136

List of Tables

9.1 Graph Generation Results . . . 63
9.2 Graph Generation Results (Cont'd) . . . 63
9.3 Neighborhood Sizes – Theoretical Expectation . . . 66
9.4 Neighborhood Sizes – Empirical; $n = 100$ . . . 67
9.5 Neighborhood Sizes – Empirical; $n = 200$ . . . 67
9.6 Singleton, $|S| = 1$; $n = 100$ . . . 73
9.7 Singleton, $|S| = 1$; $n = 200$ . . . 73
9.8 Subset Sampling, $|S| = 25$; $n = 100$ . . . 74
9.9 Subset Sampling, $|S| = 25$; $n = 200$ . . . 74

10.1 Community Detection – Algorithm Timing . . . 84
10.2 Community Detection – Greedy Algorithm Timing . . . 86
10.3 Community Detection – Greedy Tests for $n = 100$ random graph . . . 89
10.4 Community Detection – Algorithm Comparison . . . 94

11.1 Probability Amplification – Test Results; $m = 17$ . . . 99
11.2 Probability Amplification – Test Results; $k = 15\%$ . . . 99

A.1 Neighborhood Sizes – Theoretical Expectation . . . 110
A.2 Graph Generation Results . . . 110
A.2 Graph Generation Results (Cont'd) . . . 110
A.3 Singleton, $|S| = 1$ searching; $n = 100$ . . . 111
A.3 Singleton, $|S| = 1$ searching; $n = 100$ (continued from previous page) 112
A.4 Singleton, $|S| = 1$ searching; $n = 200$ . . . 113
A.4 Singleton, $|S| = 1$ searching; $n = 200$ (continued from previous page) 114
A.5 Subset sampling, $|S| = 25$; $n = 100$ . . . 115
A.5 Subset sampling, $|S| = 25$; $n = 100$ (continued from previous page) . 116
A.5 Subset sampling, $|S| = 25$; $n = 100$ (continued from previous page) . 117
A.6 Subset sampling, $|S| = 25$; $n = 200$ . . . 118
A.6 Subset sampling, $|S| = 25$; $n = 200$ (continued from previous page) . 119
A.6 Subset sampling, $|S| = 25$; $n = 200$ (continued from previous page) . 120
A.6 Subset sampling, $|S| = 25$; $n = 200$ (continued from previous page) . 121
A.7 Community Detection – Algorithm Timing . . . 122
A.8 Community Detection – Greedy Algorithm Variant Timing . . . 123
A.9 Community Detection – Random Graph Testing Results . . . 124
A.9 Community Detection – Random Graph Testing Results (continued from previous page) . . . 125
A.10 Community Detection – Explicit Graph Testing Results . . . 126
A.10 Community Detection – Explicit Graph Testing Results (continued from previous page) . . . 127
A.11 Probability Amplification – Test Results . . . 128
A.11 Probability Amplification – Test Results (continued from previous page) . . . 129

A.11 Probability Amplification – Test Results (continued from previous page) . . . 130
A.11 Probability Amplification – Test Results (continued from previous page) . . . 131
A.11 Probability Amplification – Test Results (continued from previous page) . . . 132
A.11 Probability Amplification – Test Results (continued from previous page) . . . 133
A.11 Probability Amplification – Test Results (continued from previous page) . . . 134
A.11 Probability Amplification – Test Results (continued from previous page) . . . 135

List of Abbreviations

CSPRNG	Cryptographically Secure Pseudo-random Number Generator
GG	Gabber-Galil
MFP	Multi-commodity Flow Problem
NP	Nondeterministic polynomial time (complexity class)
P	Polynomial time (complexity class)
PCP	Probabilistically Checkable Proof
PRNG	Pseudo-random Number Generator
SSE	Small Set Expansion
UGC	Unique Games Conjecture

Chapter 1

Introduction

Expander graphs are highly connected graphs that are also, in some sense, "sparse." These two competing notions give rise to graphs with extremely interesting properties. As such, expander graphs have been used to tackle many problems in both theoretical computer science and mathematics, including network design, coding theory, complexity theory, derandomization, cryptography, number theory, and geometry [14, 24, 31, 45].

1.1 History

The first appearance of expander graphs was in a paper by Pinsker, who gave a probabilistic existence argument for bipartite graphs with specific properties [42]. Pinsker, like many of the mathematicians and computer scientists studying expanders at the time, was interested in applications of expanders to switching networks used for communication. In his seminal paper, Pinsker was primarily focused on constructing graphs called "concentrators" with as few edges as possible. The definition of a concentrator is as follows:

Definition 1. *An (n, m)-concentrator is a bipartite graph with n inputs and m outputs (where $m < n$) and where any $k \leq m$ inputs can be simultaneously connected to some k outputs by non-intersecting paths. [42]*

Pinsker was able to construct a concentrator with $29n$ edges, the proof of which relied on the existence of what came to be known as expander graphs [42]. In the late 1970s, Valiant was interested in determining how many logic gates it would take to build a circuit that could compute linear transformations over a finite field [48]. He conjectured that the graph layout of any circuit that computes such a transformation would have to be a "super-concentrator," defined as follows:

Definition 2. *An n-superconcentrator is a graph with n inputs and n outputs such that, for any set of inputs and any set of outputs of the same size, there exists a set of vertex-disjoint paths that connect the inputs to the outputs in a one-to-one manner. [11]*

Valiant erroneously conjectured that any superconcentrator must have more than $\Omega(n)$ edges, a belief which he later disproved using expander graphs [48]. As these two examples illustrate, expanders are useful in a variety of unexpected fields and applications.

1.2 In Practice

Although expander graphs enjoy wide usage in many disciplines, they are almost never constructed explicitly; instead, probabilistic results about their existence are invoked. From a theoretical point of view, this is perfectly natural; however, many of the theoretical applications of expanders also have quite practical analogues. For example, Dinur uses expander graphs in her construction of probabilistically checkable proofs (PCPs), but, if one were to attempt to construct

such a proof for a real-world problem, one would quickly realize that the numerous and varied expander graphs needed for the PCP construction are not easy to find, create, or validate [14].

Many of these theoretical results that rely on the aforementioned probabilistic arguments are frequently constrained when it comes to using random bits; thus, in a practical scenario, using randomly generated expanders would simply not be an option. As a result, there has been a rich history of literature focused on finding ways to efficiently generate infinite families of expander graphs. These constructions vary wildly in complexity and in the characteristics of the resulting graphs, and as such, not all are suitable in all circumstances. The downside to many of these explicit graph construction algorithms is that the resulting graphs often have expansion rates less than that of their randomly-generated counterparts, and on top of that, many of them only work well for extremely large graphs.

1.3 Purpose

As discussed above, there exist probabilistic guarantees that, with very high likelihood, a random d-regular graph is an expander with nice expansion properties. However, that is far short of a guarantee. Furthermore, even if a randomly generated graph were an expander, there is no efficient way to determine exactly (or even approximately, to within a sub-logarithmic factor) how good the expansion is. Thus, one of the primary goals of this work was to provide methods and techniques for validating that a given graph does indeed have a high expansion rate. We examined

methods for computing estimates of expansion rates, methods for computing values that are correlated with expansion rate, and methods for making improvements over existing theoretical bounds on graph expansion. We also explored specific applications of expander graphs and provided empirical evidence for how they can be used to recycle random bits and to implement hashing algorithms.

In order to perform the tests necessary to develop and validate these methods and techniques, we have created a software suite for creating, manipulating, and testing expander graphs. The majority of the software is written in Python and makes extensive use of the numpy and scipy packages. We have also written a testing framework for community detection algorithms testing in the statistical software R, in which we make extensive use of the modMax package. Using a combination of our software, open-source code, and high-performance computing resources, we were able to explore and evaluate graph expansion and applications from an empirical standpoint. Ultimately, we will show that, given an arbitrary graph, its expansion rates can be measured and bounded by employing community detection algorithms that seek to maximize modularity.

1.4 Organization

This paper is organized into a series of chapters. We will start by formally defining expander graphs and talking about some of their characteristics, after which we will provide a brief overview of a few applications of expander graphs. Next, we discuss how different expansion measures are computed, the hardness of those

computations, and different bounds for various types of expansion. After that, we will analyze various approximation algorithms for graph expansion and assess their complexity. Following that, we will talk about graph modularity and community detection algorithms, detailing their relationship with graph expansion. We will then describe methods for creating expander graphs, to include both random and explicit constructions. We then outline the software we have written for analyzing expander graphs, describe the various subset tests we have conducted, and assess the empirical data that we have collected using our software. Next, we will discuss the software we have written for performing community detection by maximizing modularity and outline how we have used those algorithms to improve bounds on expansion rates. Afterwards, we will provide empirical evidence of the usefulness of expanders in various applications. Finally, we will discuss the implications that our results have for practical uses of expander graphs and suggest some possible directions for future work in this area.

Chapter 2

Expander Graph Basics

Strictly speaking, every finite, connected graph (which can contain parallel edges and/or self-loops) is an expander graph. However, not every graph will have good expansion parameters, and not every graph is part of an expander family. There are three primary notions of expansion that are regularly studied, which we will describe in the subsequent section. Throughout this paper, all the graphs that we will consider are undirected, unless explicitly stated otherwise.

2.1 Definitions

We will start by defining the edge and vertex boundary sets. These sets will play an important role in calculating expansion rates.

Definition 3 (Edge Boundary)**.** *Given a graph $G = (V, E)$, the edge boundary ∂S of a given set of vertices $S \subseteq V$ is the number of edges in E with exactly one endpoint in S.*

$$\partial S = \{(u, v) \in E : u \in S, v \in V \setminus S\}. \quad [31]$$

Definition 4 (Outer Vertex Boundary)**.** *Given a graph $G = (V, E)$, the outer vertex boundary $\partial_{\text{out}} S$ of a set $S \subseteq V$ is the number of vertices in $V \setminus S$ with at least one neighbor in S.*

$$\partial_{\text{out}} S = \{v \in V \setminus S : \exists (u, v) \in E \text{ such that } u \in S\}. \quad [31]$$

Definition 5 (Inner Vertex Boundary)**.** *Given a graph $G = (V, E)$, the inner vertex boundary $\partial_{\text{in}} S$ of a set $S \subseteq V$ is the number of vertices in S with at least one neighbor in $V \setminus S$.*

$$\partial_{\text{in}} S = \{v \in S : \exists (u, v) \in E \text{ such that } u \in V \setminus S\}. \quad [31]$$

Now, with our boundary sets defined, we can outline the three expansion notions mentioned in the chapter introduction, starting with edge expansion.

Definition 6 (Edge Expansion)**.** *Given a graph $G = (V, E)$ on n vertices, the edge expansion of G, $h(G)$ is*

$$h(G) = \min_{0<|S|\leq \frac{n}{2}} \frac{\partial S}{|S|}. \quad [31]$$

Intuitively speaking, we can understand edge expansion to represent the worst-case ratio of the size of a subset's edge boundary (the number of edges with exactly one edge in the subset) to the size of the subset. Thus, in order for a graph to have high edge expansion, every subset of vertices must have lots of edges in its boundary. The definition of vertex expansion is very similar to that of edge expansion; however, instead of using the edge boundary when calculating the ratio, we use the vertex boundary instead.

Definition 7 (Vertex Expansion)**.** *Given a graph $G = (V, E)$ on n vertices, the outer vertex expansion of G, $h_{\text{out}}(G)$ is*

$$h_{\text{out}}(G) = \min_{0<|S|\leq \frac{n}{2}} \frac{\partial_{\text{out}} S}{|S|}. \quad [31]$$

Similarly, the inner vertex expansion of G, $h_{\text{in}}(G)$ is

$$h_{\text{in}}(G)) = \min_{0<|S|\leq \frac{n}{2}} \frac{\partial_{\text{in}} S}{|S|}. \quad [31]$$

If G is d-regular, we can see from these definitions that the two values, vertex expansion and edge expansion, are related to one another in the sense that

$$h_{\text{out}}(G) \leq h(G) \leq d \cdot h_{\text{out}}(G).$$

In later chapters, we will see that there are even more complex relationships between these quantities and other graph expansion measures.

2.2 The Second Eigenvalue

While edge and vertex expansion provide a precise measurement of a graph's expansion properties, the eigenvalues of the adjacency matrix of an expander can also shed some light on a graph's expansion potential. Eigenvalues of the adjacency matrix are considered almost exclusively in cases where the graph G is d-regular, which means that each vertex in G is an endpoint of exactly d different edges. This is mainly due to the fact that, with non-regular graphs, the situation is more complicated and a modified version of the Laplacian matrix is required, as opposed to simply using the adjacency matrix of the graph directly [47].

Using the adjacency matrix to estimate a graph's expansion is known as computing the spectral gap of the graph. Given a d-regular graph G, consider its adjacency matrix A. Since A is symmetric, the spectral theorem tells us that there exist n eigenvalues $\lambda_1 > \lambda_2 > \cdots > \lambda_n$ of A such that $\lambda_i \in \mathbb{R}$. Since G is regular, we also know that the value of each eigenvalue falls between $-d$ and d, inclusive, and that the largest eigenvalue λ_1, is equal to d [47]. Now, let $\lambda(G)$ be defined as the second largest eigenvalue (in absolute value) of A, which gives us $\lambda(G) = |\lambda_2|$. We compute

the spectral gap as follows:

Definition 8 (Spectral Gap)**.** *Given a d-regular graph $G = (V, E)$ on n vertices, the spectral gap of G is the difference between d and $\lambda(G)$*

$$d - \lambda(G),$$

which is also sometimes referred to as the spectral expansion of G. Note that this is the difference between the first and second eigenvalues of G's adjacency matrix.

There are a number of results that show that the size of this spectral gap can provide a good estimate for a graph's expansion properties, but we will save the details for a later chapter. As the size of the gap increases, the better the graph's expansion could potentially be. There has also been a significant amount of work done on bounding the potential sizes of the spectral gap for various graph families, which we will also discuss in a later chapter. Clearly, though, since we know that the largest eigenvalue of a d-regular graph is d, we want expander graphs whose second eigenvalue is as far away from d as possible. There are even some definitions of expander graphs that require this value to be bounded away from d for explicitly constructed sets of graphs. For instance, Reingold et al. define an "expander family" as follows:

Definition 9 (Expander Family)**.** *An infinite family $\{G_n\}$ of d-regular graphs is an expander family if λ_2 is bounded uniformly from above by d such that $\lambda_2 < d$. Equivalently, the normalized version of λ_2 must be bounded away from 1. [45]*

While this definition is not particularly useful for randomly generated expanders and does not necessarily ensure good expansion, it should be noted that

there is significant importance in developing and proving tight eigenvalue bounds, a theme which we will revisit in Chapter 6.

Before we move onto talking about other characteristics of expander graphs, we will present one final result that ties the second eigenvalue to the "randomness" of a graph. This result is known as the Expander Mixing Lemma:

Lemma 1 (Expander Mixing Lemma)**.** *For all $S, T \subseteq V$:*

$$\left| |E(S,T)| - \frac{d|S||T|}{n} \right| \leq \lambda(G)\sqrt{|S||T|},$$

where $E(S,T)$ is the number of edges between subset S and subset T. [31]

Examining this inequality, we see that the left side is simply comparing the actual number of edges between S and T with the number of edges that would be expected to exist between those two subsets in a random graph. Thus, when the graph's structure is similar to the what would be expected in a random graph, the left side of the inequality is very small. Thus, when $\lambda(G)$ is small (when the spectral gap is large), the graph's structure is very similar to that of a random graph [31]. We will see later that random graphs generally do, as this result suggests, have nice expansion properties.

2.3 Random Walks on Expander Graphs

One important property of expander graphs is that taking a random walk of length t looks very similar, probabilistically-speaking, to selecting t vertices from the graph uniformly at random. At first, this might not sound very useful, but

consider that taking such a random walk requires much fewer random bits than selecting vertices at random, especially for graphs with lots of vertices. Thus, the expansion property of the graph reduces the number of random bits that need to be used to create specific probabilistic scenarios. We will formalize these notions in the subsequent paragraphs.

First, we will start with the definition of a random walk as presented by Linial and Wigderson [31].

Definition 10. *A random walk over the vertices of a graph G is a stochastic process defining a series of vertices $(v_1, v_2, \dots)$ in which the initial vertex is selected by some initial distribution, and vertex v_{i+1} is selected from the neighbors of v_i uniformly at random.*

Thus, since all but the first vertex are selected from the neighbors of the preceding vertices, we see the process of selecting subsequent vertices is, in fact, a Markov process. Furthermore, the transition matrix of the Markov chain that represents a random walk of length t on G is precisely the normalized adjacency matrix $\hat{A}^t$ of G [31]. Clearly then, we see that the stationary distribution of the random walk is, in fact, the uniform distribution. Linial and Wigderson also prove the following theorem:

Theorem 2.3.1. *Let $\hat{A}^t$ be the normalized adjacency matrix of a graph G, let $\vec{u}$ denote the uniform distribution, and let α denote the maximum of the absolute value of the second-largest normalized eigenvalue and the absolute value of the smallest normalized eigenvalue of the graph G. Then, $||\hat{A}^t\vec{p} - \vec{u}||_1 \leq \sqrt{n} \cdot \alpha^t$ for any distribution*

vector $\vec{p}$. [31]

Thus, regardless of the initial distribution vector $\vec{p}$, after taking a logarithmic number of steps, if G is an expander graph (meaning that α would necessarily be bounded away from 1 for non-bipartite expanders) we would arrive at a distribution that is within a polynomial factor of the uniform distribution [31]. Furthermore, as α shows, the size of the second largest eigenvalue is actually a measure of how quickly the random walk on G converges to a uniform-looking distribution [45].

Linial and Wigderson go on to show that, through the use of random walks on expander graphs, the success probability of randomized algorithms can increase [31]. One example of this being used in another theoretical application is in Dinur's proof of the probabilistically checkable proof (PCP) theorem. Instead of selecting t vertices uniformly at random from a graph and checking to see whether or not any of their associated constraints are violated, Dinur samples a t-step walk and shows with high probability that such a walk will, in fact, pass through at least one rejecting edge of the constraint graph, if one exists [14]. This example from Dinur is also instructive because, in this case, the reason she does not simply select t vertices uniformly at random is because the PCP theorem limits the number of random bits that the PCP verifier can use. Thus, taking advantage of the uniform distribution of a random walk on an expander graph is essential to her proof. Furthermore, because the number of random bits that can be used is limited, constructing random expander graphs on-the-fly would not be an option for Dinur. There are also many other applications for expander graphs where random constructions simply cannot

be used. We will explore methods for generating expander graphs in Chapter 8.

Now that we have explained some of the fundamental properties of expander graphs, we will describe some common applications of these graphs. These examples will provide some motivation regarding our goal to accurately assess graph expansion.

Chapter 3

Expander Graph Applications

As outlined in the introduction, there are a number of different applications for expander graphs in a wide variety of fields. Some are extremely theoretical, like their use in Dinur's proof of the PCP Theorem, while others are quite concrete, like their use in building efficient communications networks [14, 42, 48]. In this paper, we will discuss two specific applications of expander graphs in some detail. The first is the use of expander graphs for probabilistic amplification, and the second is the use of expander graph in cuckoo-style hashing algorithms.

3.1 Probabilistic Amplification

Randomness is a cornerstone in both the theory and practice of computer science. In fact, randomized algorithms are frequently faster or much more simple than their deterministic counterparts. There are also many situations that require randomly sampling from a probability distribution, such as cryptographic applications and scientific modeling [25]. Unfortunately, generating random bits is quite slow, so computer scientists have developed various methods for preserving the usage of random bits.

3.1.1 Background and Motivation

One of the most frequently used means for recycling randomness is a pseudo-random number generator. These pseudo-random number generators (PRNGs) take a random "seed" and then, via a deterministic process, produce a much longer sequence of numbers that serves as a substitute for truly random numbers. Thus, these PRNGs are one way that random bit usage can be reduced [25]. There has been a very large body of work in this area, and it has been shown that cryptographically secure pseudo-random number generators (CSPRNGs) exist and can produce sequences of arbitrary lengths such that the next bit produced cannot be predicted with probability of success better than 50% by any polynomial-time algorithm [7, 50].

Building on this work, Impagliazzo and Zuckerman were able to create a PRNG that can be constructed by taking random walks on expander graphs [25]. In fact, they were able to create these generators by using random walks on the explicitly constructed Gabber-Galil expanders (which will discuss in detail later) that strike an almost ideal balance between the error probability and the amount of randomness required by the generator; using r random bits and an error probability less than $1/2$, their generator can reduce the error probability to 2^{-k} using $\mathcal{O}(r+k)$ random bits [20, 25]. Intuitively, the fact that taking random walks on expander graphs can be used to produce good strings of random bits is not surprising. In fact, we previously discussed results from Linial and Wigderson that show that random walks on expanders converge quickly to the uniform distribution [31]. However,

those proofs relied on the fact that G not be bipartite, which leads to a slightly different analysis than that done by Impagliazzo and Zuckerman [21, 25, 31].

Expander graphs have also played a key role in the creation of randomness extractors. These extractors are essentially functions that can take sources of random numbers with low minimum entropy and transform them into a source of random numbers with minimum entropy arbitrarily close to 1 [23]. Much like the PRNG's, these extractors frequently also require a seed, but since the seeds can be as small as $\mathcal{O}(\log n)$ in length, it is usually feasible to simply enumerate all of the possibilities. Thus, these extractors can simulate strong random sources using only weak random sources and a very short seed [23].

We have performed empirical testing of probabilistic amplification in expander graphs, the results and analysis of which can be found in Chapter 11.

3.2 Cuckoo Hashing

In recent years, there has been renewed interest in the hashing algorithms, which was brought on by a paper by Azar et al., which showed that remarkably low maximum loads could be achieved using multiple-choice hashing [6, 35]. In multiple-choice hashing schemes, the available memory is divided into n buckets, and each item is hashed d times, which provides d different potential locations for the item to be stored. The item is then placed in the bucket with the smallest load. The analysis of this scheme by Azar et al. showed that, for $d \geq 2$, with high probability, the maximum load grows like $\log \log n / \log d + \mathcal{O}(1)$, which, for all

practical purposes, means that a bucket will never overflow [6]. Furthermore, only a constant number of locations need to be checked when searching, and these searches can run in parallel. From this work emerged the concept of cuckoo hashing.

3.2.1 Definition and Background

Cuckoo hashing was first introduced by Pagh and Rodler in 2001, and represents a natural extension to multiple-choice hashing techniques [41]. In the original formulation of cuckoo hashing, there are exactly two hash functions, each of which is applied to every item to be hashed, giving two possible locations for each item. On insertion of an item, if one of its two possible locations is empty, it is simply stored at that location. If both locations are already occupied, instead of simply failing to insert, the hash table removes one of the items in the two locations, stores the new item at that location, and then checks the other valid location (remember, there are two possible locations for each item) for the item that was removed. If the other location for that item is also full, then the algorithm again replaces that item with the item that was pushed out of its location initially, and the process continues until an empty position is found. In the event that the algorithm finds itself back in the position it originally started when the insertion began with the same item it needs to store, then the algorithm terminates, and the hash tables are re-built in place with new hash functions. It has been shown that, when the load of cuckoo hash tables is less than $1/2$, the lengths of these insertion paths are $\mathcal{O}(\log n)$. Thus in most practical implementations, the algorithm simply terminates after $c \log n$ steps

for some constant c if no open bucket can be found [35]. It is also easy to see that a lookup using this type of cuckoo hashing requires checking exactly two places.

Since its introduction, cuckoo hashing has been generalized and extended in a number of ways. One extension was to increase the number of hash functions (and thus, the number of potential locations in the hash table for a single item) from two to arbitrarily many. There are also variants of cuckoo hashing that allow for a given location to house more than one item, which combines the concept of cuckoo hashing with chaining to help resolve collisions [35]. If only one item can be stored in a given location, in the case of cuckoo hashing with two hash functions, if there are more than three items that all hash to the same location, the hash functions must be thrown out and all the keys rehashed with new hash functions. Thus, building in some form of collision detection is useful for avoiding costly rebuilds in real-world situations, though the performance of searches drops. Of course, the lookups still require checking only a constant number of positions in the hash table, but now, each position cannot necessarily be searched in constant time.

It is also worth noting that, in the case of two hash functions, cuckoo hashing has a very clear connection to random graphs. For instance, we can consider the vertices of the graph to be the buckets where items can be stored, and then each edge represents an item, with the two endpoints of that edge representing the two possible locations where that item can be stored [35]. We could also have the edges be directed to represent which of an edge's two endpoints is where the item is stored. Due to this neat correspondence, cuckoo hashing with two hash functions is very well understood, since random graphs are well-studied. This relationship between

random graphs and cuckoo hashing also extends to cases where there are more than two hash functions, but instead of sharing similarities with random graphs where each edge has exactly two endpoints, these hashing schemes correspond to random hypergraphs, where a single edge can be adjacent to as many vertices as there are hash functions. This scenario is much more complicated from a theoretical point of view, and is currently the subject of significant research [35].

3.2.2 Random Walk Cuckoo Hashing

One aspect of cuckoo hashing that becomes much more complicated when there are $d > 2$ has functions is the decision that needs to be made when an item is being inserted into the table, but all d of its potential locations are filled. In that scenario, one of the d items needs to be selected so that the new item can take its place, and then that item's d positions are examined for insertion. In the case where there are only two hash functions, there is only one item that can be moved during each iteration. However, if there are d possibilities, then there must be some mechanism for deciding which item to move [35]. One option would be to perform a search in the graph to find the shortest path that leads to an empty position that is then filled, but that would obviously be extremely inefficient, since it would have to be performed for every insertion.

Instead, a much more efficient strategy is to simply randomly select one of the d items to be removed at each step, and then continuing to make random selections until an empty position is found. In this way, the choices for which item to kick

out and move can be seen as a random walk on the underlying graph structure of the hash table. Experimental results have shown that this scheme might very well have logarithmic performance, but due to the complexities of having to consider how items' placements are affected by other items, the current tightest proven bound is polylogarithmic [35]. Frieze, Melsted, and Mitzenmacher were able to show that, over the choices of the hashing algorithm, with high probability, an insertion will take polylogarithmic time under reasonable loads and choices for d [19]. Work to tighten this bound is currently ongoing.

Both of these expander graph applications rely on having graphs with good expansion properties in order for them to work. Thus, developing a method for evaluating a graph's expansion rates is essential to using expander graphs in practice.

Chapter 4

Calculating and Bounding Expansion

Although the definitions of the different types of expansion measures are fairly straightforward, computing exact edge and vertex expansion rates for arbitrary graphs is hard. The difficulty, of course, comes from the fact that the minimum of the ratio of the boundary size to the subset size is taken over all subsets of size $\frac{n}{2}$ or smaller. Thus, the number of subsets whose boundaries must be considered is $\binom{n}{\frac{n}{2}} + \binom{n}{\frac{n}{2}-1} + \cdots + \binom{n}{2} + \binom{n}{1} = \Omega(2^n)$, which is exponential in the size of the graph. It is a well-known result that computing edge expansion is NP-hard, as was shown by Kaibel via a reduction from `MaximumCut`, a known NP-hard problem [26]. Proving that vertex expansion is NP-hard can be accomplished with a fairly straightforward reduction from edge expansion where all parallel edges are removed.

Now, we will prove that the problem of finding the subset with minimum edge expansion (which allows us to calculate edge expansion) can be reduced to the problem of finding the sparsest cut of a multi-commodity flow problem, which is known as the `SparsestCut` problem. Later, we will use this reduction to show that we can approximate edge expansion by using methods that approximate `SparsestCut`. Before we describe `SparsestCut`, we will first define a set of problems known as multi-commodity flow problems (MFPs), which are a generalization of the typical single-commodity network flow problem. In a MFP, there are k different commodi-

ties, each of which has an associated demand D_i, a source vertex s_i and a sink vertex t_i. Much like the normal single-commodity flow problem, each edge e has an associated capacity $c(e)$, such that no more than $c(e)$ total units of any commodity can traverse edge e. The goal of an MFP is to find a flow such that D_i units of each of the k commodities can be routed from s_i to t_i without any of the edge capacity constraints being violated [30].

A maximum flow for an MFP is slightly different than a maximum flow for a single-commodity network. For example, if there is only enough capacity on a given edge for one commodity's demand to be satisfied, one would have to somehow rank the commodities to decide which one would be part of a "max flow." Instead, the max flow of an MFP is usually defined by a fraction f, such that, for each demand D_i, fD_i units of flow are able to travel from s_i to t_i. The goal, then, is to find the the flow that maximizes that fraction [30]. The minimum cut of an MFP is also slightly different than the minimum cut of a standard flow problem. In fact, the MFP equivalent of a minimum cut is the sparsest cut, which we can now define formally.

`SparsestCut`: Let $G = (V, E)$ be a weighted graph with edge weights $c_e \in \mathbb{R}^+$ for all $e \in E$, and let P be a set containing k 2-tuples of vertices

$$\{(u_1, v_1), (u_2, v_2), \ldots, (u_k, v_k)\}, \text{where } u_i, v_i \in V$$

with demand D_i between the two vertices in tuple i. Find the cut S^* with minimum sparsity, which is the cut that minimizes

$$\Phi(S) = \frac{c(S)}{D(S)},$$

where

$$c(S) = \sum_{e \in V \text{ s.t. } e \text{ crosses } S} c(e)$$

and

$$D(S) = \sum_{(u_i, v_i) \in P \text{ s.t. } S \text{ separates } u_i \text{ and } v_i} D_i.$$

Now, the first thing to notice is that this definition of `SparsestCut` places no restrictions on the demand values D_i between vertices. However, for the reduction, we will be reducing instances of edge expansion to a special case of `SparsestCut`, known as uniform `SparsestCut`, where there is unit demand between every pair of distinct vertices and where edge weights $c_e \in \{0, 1\}$. Note that, when we have unit demand between all pairs of distinct vertices, we know that there are $|S|$ vertices on one side of the cut and $|V - S|$ vertices on the other side of the cut for a total of $|S||V - S|$ vertex pairs with unit demand that are separated by the cut. Thus, we can re-examine our sparsity equation and see that the denominator, $D(S)$, can be rewritten as $|S||V - S|$. This gives us

$$\Phi(S) = \frac{c(S)}{|S||V - S|}.$$

Without loss of generality, we can write this fraction as a minimization over all cuts $|S| \leq \frac{n}{2}$ such that

$$\min_{0 < |S| \leq \frac{n}{2}} \frac{c(S)}{|S|}.$$

This minimization is equal to the true minimum sparsity up to a constant factor, since $\frac{n}{2} \leq |V - S| \leq n$, but more importantly, the same cut that leads to the minimum sparsity using the first form is the cut that will lead to the minimum in the second form [30].

Now, we can proceed with our reduction; we will present our own proof of the hardness of `SparsestCut`.

Theorem 4.0.1. *`SparsestCut` is at least as hard as computing edge expansion.*

Proof. We will prove that finding the `SparsestCut` of a MFP is at least as hard as computing a graph's edge expansion rate by providing a polynomial-time reduction from edge expansion to `SparsestCut`. The reduction will transform a graph into an MFP; the vertices in the smaller half of the cut that leads to the minimum sparsity of the MFP will be identical to the subset of vertices in the graph for which the ratio of the edge boundary to the size of the subset (the graph's edge expansion) is minimized.

Given an undirected graph G, we are interested in finding its edge expansion by identifying the subset R where $\partial(R)/|R|$ is minimized over all subsets of size $\frac{|V|}{2}$.

We will transform G into an MFP as follows. First, the underlying graph of the MFP will be identical to G, containing the same vertices and edges. The set P will contain a tuple for each pair of distinct vertices in the graph for a total of $\binom{|V|}{2}$ tuples. Thus, there will be $\binom{|V|}{2}$ different commodities. Each of these commodities will have unit demand, and the capacity of each edge will be 1. Clearly this construction takes polynomial time.

Now that we have an MFP, we will let S^* represent the cut of the MFP that corresponds to the minimum sparsity of the network. We will show that this minimum sparsity value is precisely equal to the edge expansion of the graph G.

First, we will let S' and V' be the sets of vertices into which S^* divides the

vertices of G. Without loss of generality, we assume that $|S'| < |V'|$, thus, $|S| \leq \frac{|V|}{2}$. Now, since all the edges have unit capacity, we see that the numerator in the sparsity fraction, $c(S^*)$, is equal to the number of edges that cross the cut. Also, since there is unit demand between all pairs of vertices, we can use the second form of `SparsestCut`, meaning that, since we know that S^* is the sparsest cut, we know that S' minimizes

$$\min_{0<|S|\leq\frac{n}{2}} \frac{c(S)}{|S|}.$$

In this form, the relationship between `SparsestCut` and edge expansion is clear. Since we have identified the set that minimizes

$$\min_{0<|S|\leq\frac{n}{2}} \frac{c(S)}{|S|},$$

we know that the numerator is equivalent to the number of edges that cross the cut, and the denominator is the size of the smaller half of the cut, this minimization is identical to the edge expansion minimization; the subset S' is precisely the subset R that we need to find the graph's edge expansion. The size of the edge boundary of S' is precisely equal to $c(S')$, since the edges that leave the subset S' are the same edges that cross the cut S^* and all edges have unit capacity. Edge expansion then divides that number by the size of the subset, which we also do in the case of the second form of `SparsestCut`.

Thus, after creating a MFP from our original graph and finding the sparsest cut, we then take the subset of vertices in the smaller half of the sparsest cut, and we have shown that this subset is the subset for which the edge expansion of G is minimized. Clearly, this identity mapping of the subset from the MFP back to the

original graph G takes polynomial time. □

Thus, we have now shown that finding the `SparsestCut` is at least as hard as finding the subset of a graph which minimizes its edge expansion, which means that `SparsestCut` is NP-hard. More importantly, however, this reduction from edge expansion to `SparsestCut` means that, any method that can be used to estimate `SparsestCut` can also be used to estimate edge expansion. Since we have shown that there is no efficient way to compute a graph's expansion rate, the natural next step is to determine how difficult it would be to create an accurate estimate for graph expansion. We will use this fact that edge expansion reduces to `SparsestCut` to develop one method for approximating edge expansion. For additional information on reductions to and from expansion-related problems, see Raghavendra et al. [44].

Chapter 5

Hardness of Approximating Graph Expansion

In the previous chapter, we proved that exact expansion rates for arbitrary graphs are difficult to compute. Thus, the natural question that arises is whether or not there are polynomial-time algorithms that can provide good approximations of expansion rates. Ideally, we would like a polynomial-time approximation scheme that will allow us to compute expansion rates to within arbitrarily small constant factors greater than 1. Unfortunately, no such approximation algorithms are currently known [30, 32, 43]. Furthermore, if P = NP, then the Unique Games Conjecture (UGC) can be used to show that it is NP-hard to approximate these expansion rates to within **any** constant factor [32, 43].

However, all is not lost. Although there is likely no way to approximate edge expansion efficiently up to a constant factor, there are algorithms that will allow us to approximate these values to within a logarithmic factor [30]. In this chapter, we will describe the Unique Games Conjecture (UGC) and connect the hardness of the UGC with the hardness of approximating graph expansion. We will also describe and explain the logarithmic approximation algorithm for graph expansion.

5.1 Unique Games Conjecture

The Unique Games Conjecture (UGC) was published by Khot in 2002 and continues to be one of the most important open problems related to the hardness of approximation [27, 28]. Khot's original conjecture was in the form of examining the power of 2-prover systems; we will give a slightly more concrete formulation. Consider a graph G where each vertex must be assigned a color and where each edge has a set of ordered pairs of colors that constrain which colorings are valid. The constraints themselves must also be unique, meaning that there cannot be more than one ordered pair constraint at a given edge that represents assigning the same color to the same node. This problem is known as the *label cover with unique constraints* problem [27, 36]. More formally, an instance of such a problem can be represented by an alphabet of size m, an undirected graph, and a set of permutations $\Pi_e : [m] \to [m]$, one for each edge $e \in E$.

Note that these constraints are extremely robust. When trying to find a satisfying assignment of colors to the vertices in the graph, after selecting an initial vertex and a color, the colors of all of the other vertices are immediately fixed, due to the uniqueness of the constraints on each edge. Thus, if a satisfying assignment exists, finding such an assignment can easily be computed in polynomial time. However, if the problem instance, or "game," is unsatisfiable, it is very difficult to compute the maximum fraction of constraints, the *value* of the game, that can be satisfied.

The gap-version formulation of the UGC is the problem of distinguishing between the following two cases, given a pair of constants (ϵ, δ) and an instance of the

unique label cover problem:

- $(1-\epsilon)$ satisfiability - There exists an assignment of colors to the vertices of G such that a $(1-\epsilon)$ fraction of edges have satisfied constraints.
- δ non-satisfiability - There does not exist an assignment of colors to the vertices of G such that more than a δ fraction of edges have satisfied constraints. [27]

Now, with the gap-version of the problem completely defined, we can state the conjecture directly.

Conjecture 1 (Unique Games Conjecture [27])**.** *For all constants ϵ and δ, there exists some constant R such that the gap-version of the label cover with unique constraints problem over an alphabet of size R is NP-hard to compute.*

Intuitively, the conjecture is simply stating that, regardless of how one select the constants ϵ and δ, there is a version of the label cover with unique constraints problem for which the two cases above cannot be distinguished, namely, when the alphabet (i.e. the number of colors) is of size at least R.

Since the UGC was postulated, it has been successfully shown to imply optimal inapproximability results for a number of problems, including `MaxCut`, `VertexCover`, and `SparsestCut`, among others [28]. One of the most interesting aspects of the UGC is that there is no clear consensus as to the likelihood of the UGC being true or false. Recently, Arora, Barak, and Steurer demonstrated subexponential algorithms that improved the best known approximations for a number of problems, including `MaxCut`, `SparsestCut`, and `SmallSetExpansion` [4]. As we will see in the next

section, the hardness of these problems is directly related to the hardness of UG. This result does not refute the UGC outright, but does suggest that it is significantly easier than other NP-hard problems like `3-SAT` and `Vertex Cover` [4].

Regardless of the ultimate resolution of the UGC, there has been a significant amount of literature in recent years that has been used to prove results about the relationship between the UGC and the hardness of approximating various graph expansion measures.

5.2 Approximating Small Set Expansion

Recall the definition of edge expansion, $h(G)$, that we stated earlier. This optimal measurement is quite coarse in the sense that it represents only the worst case edge expansion of the graph over subsets of all different sizes since it is a global minimum. For instance, consider a "typical" graph, such as a random d-regular graph (which we will discuss at great length in subsequent chapters) and the effect that the size of the subset considered has on its expansion rate. All sets S of size δn in a random, d-regular graph have expansion of approximately $1 - \frac{2}{d}$, whereas the conductance Ξ_G of the whole graph is about $\frac{1}{2}$ [43]. Thus, instead of trying to approximate the value of $h(G)$ directly, there has been significant work done in describing the difficulty of approximating the *expansion profile* of a regular graph G, given as

$$\Xi_G(\delta) = \min_{\mu(S)=\delta} \Xi(S) \;\; \forall \delta \in \left[0, \frac{1}{2}\right],$$

where $\mu(S)$ is the size, or volume, of set S. This gave rise to what is known as the `SmallSetExpansion` Hypothesis (SSE):

Hypothesis 1 (`SmallSetExpansion` Hypothesis, [43])**.** *For every constant $\eta > 0$, there exists sufficiently a small $\delta > 0$ such that, given a graph $G = (V, E)$, it is NP-hard to distinguish between the two cases:*

- *Yes: There exists a set $S \subseteq V$ with volume $\mu(S) = \delta$ and expansion $\Xi(S) \leq \eta$*
- *No: There does not exist a set $S \subseteq V$ with volume $\mu(S) = \delta$ that has expansion $\Xi(S) < 1 - \eta$*

Although the SSE might seem initially like a more esoteric problem than simple edge expansion, it actually has very close ties to the UGC. For instance, Raghavendra and Steurer were able to prove that the SSE problem actually reduces to the UGC, illustrating that small set expansion approximation plays a central role in the combinatorial inner workings of Unique Games problems [43]. Moreover, this also means that a refutation of the UGC would provide new algorithms for approximating edge expansion. Raghavendra and Steurer also proved that, under a modified UGC, approximating small set expansion is UG-hard [43].

5.3 Unique Games, Small Set Expansion, and Graph Expansion

There has also been a significant amount of work done in the area of trying to reduce the UGC to edge and vertex expansion approximation. However, due to the nature of the UGC instances (namely, their lack of expansion), reducing

directly from the UGC is impractical. Instead, a slightly stronger statement of the UGC is used, one which comes from the SSE hypothesis and assumes that fairly reasonable expansion occurs (by having sufficiently small sets have conductance close to 1) [32, 43]. This slightly modified version of the UGC has, in fact, been shown by Louis, Reghavendra, and Vempala to reduce to vertex expansion by way of an intermediate reduction through a problem called `Balanced Analytic Vertex Expansion` [32]. Their work showed that it is SSE-hard to differentiate between the cases where a graph has vertex expansion $< \epsilon$ for some $\epsilon > 0$ or whether the vertex expansion is at least an absolute constant [32]. This result is interesting in that it suggests that approximating vertex expansion is harder than approximating edge expansion, since Cheeger's inequality can be used to determine whether a graph has constant edge expansion [1].

Thus, we have shown that approximating edge and vertex expansion is at least as hard as the UGC, which is, in turn, at least as hard as SSE. If the UGC is true, then this would prove that it is NP-hard to find good (within a constant factor) approximations for edge and vertex expansion; however, despite the difficulty of finding a good approximation for these values, there has been significant work in the area of creating approximation algorithms for these problems.

Chapter 6

Algorithms for Approximating Graph Expansion

6.1 Cheeger's Inequality and the Second Eigenvalue

As was outlined in the second chapter, one of the first (and simplest) methods for approximating edge expansion was examining the size of the spectral gap of a graph's adjacency matrix. Alon and Milman, using this fact, were able to prove a discrete form of the Cheeger inequality that involves using the second largest eigenvalue to bound edge expansion [1, 2, 9]. They proved the following:

Theorem 6.1.1 (Cheeger's Inequality [1, 2])**.** *If λ_2 denotes the second largest eigenvalue of the adjacency matrix of a d-regular graph G then,*

$$\frac{d-\lambda_2}{2} \leq h(G) \leq \sqrt{2d(d-\lambda_2)}.$$

Thus, since the second eigenvalue can be computed quickly, this inequality provides an approximation for edge expansion. However, the bounds provided by the Cheeger inequality are not very tight (as we will see explicitly in subsequent chapters), especially when the expansion rate is low. Relationships between the spectral gap and vertex expansion have also been shown, such as

$$h_{\text{out}}(G) \leq (\sqrt{4(d-\lambda_2)}+1)^2 - 1$$

and

$$h_{\text{in}}(G) \leq \sqrt{8(d-\lambda_2)} \quad [8].$$

Using the second eigenvalue is useful for getting a rough idea of the expansion rates of a graph, but it does not provide a good means for approximating edge expansion.

6.2 `SparsestCut` Relaxation

In 1999, Leighton and Rao revolutionized the approximation algorithm for edge expansion by introducing a relaxation to the `SparsestCut` problem (which we showed before was NP-hard) that provides a $\mathcal{O}(\log n)$ approximation algorithm for computing the value of the true sparsest cut in a multi-commodity flow problem. As we saw in our reduction from `SparsestCut` to edge expansion, the sparsest cut of an MFP is itself an approximation for $h(G)$ to within a constant factor [30]. Thus, this relaxation also allows for edge expansion to be approximated to within a logarithmic factor. In order to understand how this relaxation works, we must first introduce the concept of a cut metric.

Definition 11 (Cut Metric)**.** *Given a cut $S \subseteq V$ of a graph G, the cut metric associated with S is δ_S, where*

$$\delta_S(x, y) = \begin{cases} 0, & \text{if } x, y \in S \text{ or } x, y \in V \setminus S \\ 1, & \text{otherwise} \end{cases}$$

Note that we can associate vectors in $\mathbb{R}^{\binom{n}{2}}$ with any n-point metric, where each coordinate represents a pair of vertices in the corresponding metric space. Now, we can restate `SparsestCut` in terms of cut metrics and dot products:

$$\min_{\text{all cut metrics } S} \frac{\overline{c} \cdot \overline{\delta_S}}{\overline{D} \cdot \overline{\delta_S}}$$

where $\overline{c}$ is the vector with one coordinate per 2-tuple of vertices in V, which represents the capacity of the edge between the two vertices in the 2-tuple. Similarly, $\overline{D}$ represents the demand vector of the set of vertices. From this point, it is fairly straightforward to see that the set of all cut metrics is a subset of the ℓ_1 metrics. Of course, since we know that `SparsestCut` is NP-hard, taking the minimum over only the set of ℓ_1 metrics is intractable. However, the observation made by Leighton and Rao was that this problem can be relaxed by taking the minimum over all possible metrics, since a cut S already defines a semi-metric over V since d_S is symmetric, since the distance between a vertex and itself is 0, and since the triangle inequality holds [30]. Thus, the relaxed form minimizes over the set $\{d \in \{V \times V \to \mathbb{R}\} : d \text{ is a semi-metric}\}$. Now, we can approximate sparsest cut as follows:

$$\min_{d \text{ semi-metric}} \frac{\overline{c} \cdot \overline{\delta_S}}{\overline{D} \cdot \overline{\delta_S}}$$

Now, we can re-write this as a linear program where the $c_{i,j}$ values are elements of $\overline{c}$ and the $d_{i,j}$ values are elements of $\overline{\delta_S}$. Moreover, since we are computing edge expansion and not `SparsestCut`, we know that there is uniform demand between all pairs of vertices, which further allows us to simply our program. We now have:

$$\begin{aligned} \min \quad & c_{i,j} d_{i,j} \\ \text{subject to} \quad & d_{i,j} \leq d_{i,k} + d_{k,j} \quad \forall i, j, k \in V \\ & \sum\nolimits_{i,j} d_{i,j} = \frac{|V|^2}{2|E|} \\ & d_{i,j} \geq 0 \quad \forall i, j \in V \end{aligned}$$

There are a few things to notice about this linear program. First, we can arbitrarily scale any semi-metric by multiplying all distances by a fixed constant.

Hence, we are able to remove the denominator by scaling $\overline{\delta_S}$ such that the sum of its components is $\frac{|V|^2}{2|E|}$ [30]. Second, although the formulation is clean and the program can be solved in polynomial time, the triangle inequality constraints alone represent $3\binom{n}{3}$ unique constraints, which for $n = 1000$ is already approximately 10^8 constraints. Thus, using this method for approximating the edge expansion of very large graphs is infeasible from a memory and time point of view, which was acknowledged by Leighton and Rao themselves [30]. Finally, the last point is about the accuracy of the estimation for edge expansion produced by this program. Leighton and Rao were able to show that the solution to this linear program, $LR(G)$, is within a logarithmic factor of the true sparsest cut such that

$$LR(G) \leq \Phi(G) \leq \mathcal{O}(\log |V|) \cdot LR(G). \quad [30,\ 47]$$

In our reduction of edge expansion to `SparsestCut`, we proved that the minimum sparsity is within a constant factor of $h(G)$, the graph's edge expansion. In subsequent sections, we will use this linear program to estimate the edge expansion rates of graphs. A proof of this logarithmic bound on the approximation can be found in Leighton and Rao's paper, which also includes a discussion about the dual of the preceding linear program [30].

One goal of this work was to develop and use an implementation of the Leighton-Rao relaxation linear program to compute actual estimates for the expansion rates of graphs. We wrote code that implements this linear program and uses the scipy.optimize package to find the optimal solution. A list of pairs of distinct vertices and a list of 3-tuples of vertices (which are used to formulate the

triangle inequality constraints) are generated using the itertools Python package. Unfortunately, due to the relaxation relying on the aforementioned triangle inequality constraints, $3\binom{n}{3}$ different constraints are required for a graph on n vertices. For a graph with 100 vertices, that is approximately $160,000$ constraints, which was beyond the computational resources we had available. This difficulty is not surprising, as, in their paper, Leighton and Rao themselves note that "linear programming is not very fast for large multi-commodity flow problems in practice" [30].

There are also newer algorithms that improve upon the $\mathcal{O}(\log n)$ approximation. For example, in 2004, Arora, Rao, and Vazirani used semidefinite programming to find an $\mathcal{O}(\sqrt{\log n})$ approximation algorithm [5]. More pessimistically, Ambühl, Mastrolilli, and Svensson showed that both vertex expansion and edge expansion have no polynomial-time approximation schemes if SAT does not have a sub-exponential time algorithm [3]. Thus, although approximations for edge and vertex expansion do exist, it is very unlikely that there will ever be a way to approximate these values to within a constant factor.

We have now shown that, given a graph, it is very difficult to determine how good its expansion properties are by either trying to compute the expansion rates directly or by trying to estimate the rates to within a constant factor. In fact, we have seen that estimating the rates to within a logarithmic factor is also intractable in most cases. Thus, we will now explore some other methods for beating the theoretical bounds on expansion rates, which then narrows the bounds on the possible expansion rates of a given graph.

Chapter 7

Graph Modularity

Many interesting datasets can be represented as a graph consisting of vertices and edges, including communication networks, the Internet, and ecological data, among countless others. Thus, the study of these networks and their structure has been the subject of significant research in statistics, mathematics, and computer science [39, 40]. In an attempt to understand the structure of these graphs and derive meaning from their topological properties, the concept of community detection has come to the forefront. In general terms, the goal of community detection is to find clusters of vertices within a network that have dense connections with other members of the cluster, but which are much more sparsely connected to vertices outside of the cluster [10, 13, 22, 39, 40]. This problem of detecting communities is similar to various graph partitioning algorithms studied by computer scientists, but the major difference is that many graph partitioning algorithms rely on being given a fixed number of communities into which the graph should be divided. However, if the goal of detecting communities is to derive meaning from the graph structure, then it makes little sense to forcibly divide the graph into some arbitrary number of pieces [39]. Furthermore, if a graph cannot be divided into well-defined communities, graph partitioning algorithms will still find a valid partitioning, but a more sensible approach would be to use an algorithm that could identify cases where there is no

"good" separation of the network. To resolve this dilemma, most modern community detection algorithms work by maximizing a value known as the graph's modularity, which was first introduced by Newman in 2003 [40].

7.1 Definition and Motivation

Intuitively, maximizing a graph's modularity is equivalent to finding a division of the graph into communities such that, from a probabilistic point of view, the number of edges traversing communities is minimized when compared to what would be expected in a randomly generated graph with vertices of the same degree. As Newman puts it, "true community structure in a network corresponds to a statistically surprising arrangement of edges, [which can] be quantified using the measure known as modularity" [39].

There are a few different, but equivalent, ways to precisely express a modularity measure. The first is a sum over the communities detected by the modularity maximizing algorithm, and can be written as

$$Q = \sum_i (e_{ii} - a_i^2),$$

where the sum is taken over each community i, e_{ii} represents the fraction of edges in the graph that are internal to community i (that is, those edges whose endpoints are both in community i), and a_i is the fraction of edges that have at least one endpoint in community i [10, 39].

Alternatively, we can write a formula for modularity using indicator variables that sum over all of the unordered pairs of vertices. We will define A to be the

adjacency matrix of the network, k_i to be the degree of vertex i, and m as the total number of edges in the network [10, 39]. We will also let δ_{c_i,c_j} be the Kronecker delta symbol, which is 0 if c_i and c_j are distinct communities, and 1 otherwise. Then, we can write modularity as

$$Q = \frac{1}{2m} \sum_{ij} \left(A_{ij} - \frac{k_i k_j}{2m} \right) \delta_{c_i,c_j}. \tag{7.1}$$

In a random graph, it is easy to see that the expected number of edges between vertices i and j would be precisely $\frac{k_i k_j}{2m}$. Thus, the quantity being summed is, again, the difference between the actual number of edges between vertices i and j and the expected number of edges between those two vertices. This definition can be extended to handle arbitrarily many communities, but the notation is somewhat unwieldy.

As these two formulations indicate, the actual modularity value can be positive or negative with a maximum value of 1. Modularity values around 0 indicate that the community structure found by the algorithm is no more statistically significant than a corresponding structure in a random graph with similar vertex degrees. On the other hand, modularity values that approach 1 indicate that the graph can be divided well into distinct communities, and negative modularity values indicate a worse-than-random community structure, given the degrees of the vertices in the network [10, 39]. This definition and comparison of edge counts within communities to expected edge counts is very intuitive, but it is not necessarily clear that an algorithm that divides a network with the goal of maximizing modularity will find good communities. However, in empirical studies performed by Guimerá and

Amaral and by Danon et al., it was shown that community detection using modularity maximization performed significantly better than other community detection methods, and since then, has been the de-facto means for detecting communities [13, 22]. These tests used simulated annealing to maximize modularity, which is simply too computationally intensive to be used in practice [39]. Thus, a wide range of algorithms have been developed that aim to maximize the modularity of a network using a variety of different approaches.

7.2 Algorithms

As mentioned in the previous section, the first modularity maximization algorithms used simulated annealing, which is a well-known procedure for locating good approximations of local optima [10, 13]. This method starts with some arbitrary partitioning of the vertices into communities, and then, at each step, it selects a node and a (possibly empty) community, and recalculates the modularity of the network if the node were moved from its current community into the new community. If the modularity increases, the move is accepted. Unfortunately, due to the relatively slow convergence rates of simulated annealing algorithms, it simply was not suitable to be used for maximizing modularity in most practical cases [39].

There are also a number of greedy algorithms that can be used for maximizing modularity [37]. These algorithms also start with an initial partitioning of the vertices into communities, which can then be combined in numerous ways. In the most basic version of these algorithms, each vertex starts in its own community,

but there are also versions where random walkers traverse some percentage of the graph, and the initial communities are populated based on the vertices traversed in those walks [10]. There are also versions that use subgraph similarity, versions that use refinement methods, and versions that allow for the initial community structure to be specified if there is pre-existing knowledge of the network topology [46]. At a high level, these algorithms work by, at each step, simply merging pairs of communities together, computing the modularity of the new community structure, and then selecting the merge that results in the highest change in modularity.

A slightly different variant of the traditional greedy algorithm for maximizing modularity, the Louvain algorithm, is also frequently used in practice. This algorithm is a divided into two stages that repeat iteratively until changes to the structure no longer improve the modularity. The algorithm starts with each vertex in its own community, and in the first phase, merges between neighboring communities are considered in some canonical ordering [10]. The merge that leads to the highest modularity improvement is then selected, and the algorithm transitions to the second phase. During this phase, the algorithm constructs a new network based on the community structure found in the first phase, where each node in the new meta-network represents an entire community in the structure of the original network. Vertices in this meta-network are connected by an edge if there are edges that connect nodes in the corresponding communities of the original network. After this new network is constructed, the algorithm transitions back to the first phase and attempts to maximize the modularity of the synthetically constructed network. Once no more improvements can be made, the algorithm terminates. This algorithm

typically requires only a few iterations, and thus, its complexity grows linearly in the number of edges of the graph [10].

Another class of modularity maximization algorithms uses a technique known as spectral optimization. Rather than using the spectrum of the adjacency matrix directly, most of these algorithms use the spectrum of a matrix known as the modularity matrix [10, 38]. In order to understand these methods, we must first take a slight detour in order to properly define the modularity matrix.

First, we wil re-write the formulation for modularity in equation 7.1 using an index vector $\mathbf{s}$ where s_i is 1 if vertex i is in community s_i, and -1 otherwise. Then, we see that

$$\frac{1}{2}(s_i s_j + 1)$$

is equal to 0 if s_i and s_j are different communities (that is, when vertex i and vertex j are in different communities), and 1 otherwise [38]. Thus, we can write

$$\begin{aligned} Q &= \frac{1}{4m}\sum_{ij}[A_{ij} - \frac{k_i k_j}{2m}](s_i s_j + 1) \\ &= \frac{1}{4m}\sum_{ij}[A_{ij} - \frac{k_i k_j}{2m}](s_i s_j) \end{aligned}$$

Now, we can simply rewrite this formulation using matrices as follows:

$$Q = \frac{1}{4m} s^T \mathbf{B}\mathbf{s}.$$

Here, we have that $\mathbf{s}$ is the index vector and $\mathbf{B}$ is the matrix whose entries are

$$B_{ij} = A_{ij} - \frac{k_i k_j}{2m},$$

which we refer to as the modularity matrix. Now that the modularity matrix has been defined, we can use its spectrum to attempt to maximize the modularity of

the network. The most simple way to do this is to simply find the eigenvector of the matrix with the largest positive eigenvalue, and then use the signs of the elements of the eigenvector to divide the network into two groups. Thus, the division of the network into communities is accomplished by choosing an index vector $\mathbf{s}$ that is proportional to the leading eigenvector of the modularity matrix [10, 38]. After dividing the initial network into two groups, the algorithm continues to divide the newly identified communities until the change in modularity resulting from the division is no longer positive. There are also other variations of spectral optimization algorithms that use the Laplacian matrix, but the approach is essentially the same [10].

In addition to simulated annealing, greedy, and spectral optimization algorithms, there are also extremal optimization, sampling, and genetic algorithms for performing community detection through maximizing modularity. The work in this paper is heavily focused on the greedy algorithm and its variants, but further details on other algorithms can be found in Chen et al., Schelling and Hui, and various papers by Newman [10, 38, 39, 46].

7.3 Relationship with Graph Expansion

Now that we have some intuition about graph modularity and how maximizing modularity is an effective means of detecting communities, we can examine the relationship between community detection and graph expansion. Recalling the definition of edge and vertex expansion, we see that the problem of directly computing

graph expansion requires finding the set of vertices (of size less than or equal to half the total number of vertices) where the ratio of the size of the set's boundary to the set's size is smallest. Of course, then, this set would necessarily have a very low number of edges (or vertices, depending on the type of expansion) that have one endpoint within the set and one endpoint outside the set, since those vertices are precisely those in the set's boundary. So, if we want to find the set with the smallest boundary to size ratio, it seems logical that community detection algorithms using modularity maximization could be useful, since it will identify sets of vertices with very small boundaries. In fact, that is precisely what a graph's modularity represents; it is a measure of how many external connections a set has as compared to what would be expected in a randomly generated graph. Thus, the communities found by a modularity maximization algorithm should be good exemplar subsets for the worst-case scenario subset that we are looking for when computing expansion. One potential drawback, however, is that modularity calculations are not weighted based on the sizes of the communities it finds. In other words, the modularity of a graph is based solely on comparing actual percentages of external connections to expected percentages of external connections; the size of the communities found by the algorithm do not affect the modularity calculations directly. In the cases of edge and vertex expansion, we want communities that have both small boundaries *and* large size. Thus, finding communities that maximize a network's modularity is not exactly the same as finding communities with the worst expansion, but certainly, having much fewer connections between a given community and the vertices outside that community is a necessary, but not sufficient, condition for having a low

expansion rate.

Having identified one potential method for narrowing the bounds of expansion rates of graphs, we will now describe some of the ways expander graphs are generated. Afterwards, we will describe and discuss our empirical testing of expansion estimation and evaluation.

Chapter 8

Generating Expander Graphs

Due to the wide applicability of expander graphs, it is natural to want to find ways of efficiently constructing graphs with good expansion properties. In general, there are two main classes of methods for creating expander graphs. The first way is through randomly generating graphs that are, with high likelihood, expanders. This strategy is the fastest and most straightforward, but the downside is that many applications of expander graphs in theoretical computer science have a limited amount of randomness available and wish to conserve the number of random bits used. Thus, exhausting precious random bits to generate a large expander graph defeats the purpose of creating it in the first place. The second class of methods for generating expander graphs is through explicit construction. There are a number of different expander constructions, but many of them are limited to only being able to produce expander graphs that fall within a narrow range of certain characteristics.

8.1 Random Generation

If there is no restriction on the amount of randomness that can be used in a particular application, randomly generating expander graphs is the fastest and easiest way for producing graphs with good expansion rates. In 1973, Pinsker showed, using probabilistic arguments, that almost all d-regular graphs are expander graphs,

in the sense of vertex expansion [42]. Alon then extended this work to conjectures about eigenvalue bounds for random regular graphs, suggesting that, for any $\epsilon > 0$ and d, the second largest eigenvalue of the vast majority of random regular graphs is less than or equal to $2\sqrt{d-1}+\epsilon$ [1]. Friedman was later able to prove this result, and was also able to show what remains to this day as the best known bound on the eigenvalues of random d-regular graphs with d even, which is that the second largest eigenvalue is at most $\frac{2}{\sqrt{d}} + \mathcal{O}(\frac{\log d}{d})$ [16, 18]. Alon and Boppana later showed that the best possible eigenvalue bound for an infinite family of d-regular expander graphs is $2\sqrt{d-1}$ [2]. Graphs whose second eigenvalue is less than $2\sqrt{d-1}$ are known as Ramanujan graphs.

It should also be noted that there are different methods for producing a random d-regular graph. Perhaps the most common construction is that used by Friedman, which involves creating $\frac{d}{2}$ permutations on $\{1 \dots n\}$ uniformly and independently [16]. Each of these $\frac{d}{2}$ permutations represents the creation of n edges. Given one such permutation $P = \{p_1, p_2, \dots, p_n\}$, the following edges are added to the graph: $\{(1, p_1), (2, p_2), \dots, (n, p_n)\}$, where 1 to n represent the n vertices in the graph under some canonical ordering. One thing that should be noted about this method is that, when one of the random permutations matches a vertex to itself, a single self-loop is added. This increases the degree of the vertex with the self-loop by two instead of by one, which is expected. However, in order for the adjacency matrix of an undirected graph to have d as an eigenvalue (which occurs if and only if a graph is d-regular), self-loops should only add one to the corresponding position in the matrix diagonal.

Another method for generating random regular graphs was introduced by Kim

and Vu [29]. Their method greedily selects "suitable" pairs of vertices and creates edges based on the pairs that were selected. Their algorithm proceeds as follows:

Random Regular Graph Generation Algorithm [29]

(I) Start with a set U of nd points (nd even) partitioned into n groups of size d.

(II) Repeat the following until no suitable pair can be found: Choose two random points i and j in U and if ij is suitable, pair i with j and delete them from U.

(III) Create a graph G with an edge from r to s if and only if there is a pair containing points in the r^{th} and s^{th} groups. If G is regular, output it, otherwise return to step (I).

Essentially, this algorithm provides a "bin" of d half-edges for each vertex in the graph that are then matched up with other half-edges. Much like the previous algorithm, if half-edges from the same bin are selected, two half-edges for that vertex are removed, but only 1 is added to the corresponding position in the graph's adjacency matrix. This will ensure that the degree of every vertex is d, and that the eigenvalues of the adjacency matrix are representative of a d-regular graph.

Either of these methods can be used to generate random regular graphs. While there is no guarantee that the resulting graph will always have good expansion, the probabilistic eigenvalue bounds proved by Pinsker, Friedman, and Alon can be used to show that, with high probability, they will have good expansion rates [2, 17, 42].

8.2 Explicit Generation

As was previously mentioned, many of the applications that use expander graphs have a limited amount of randomness to work with. Thus, generating expander graphs randomly is simply not an option. This has lead to a significant amount of research focused on developing methods for generating families of expander graphs with nice expansion properties (and frequently also with constant degree).

8.2.1 Early Work

The first explicit construction was published by Margulis, who used the following definition of bipartite expander. [Note: In this section, and this section only, we will use the traditional (n, k, w) notation for bipartite expanders, where n is half the total number of vertices in a graph (since there are n inputs and n outputs), as opposed to the total number of vertices.]

Definition 12 (Bipartite Expander [34])**.** *A bipartite (n, k, w) expander is a graph with n inputs, n outputs, at most kn edges, and where, for every subset X of inputs, $|\Gamma_X| \geq [1 + w(1 - |X|/n)]|X|$, where Γ_X is the set of outputs connected to X.*

Margulis used group representation theory to construct explicit bipartite expanders $\{G_n\}$ for $n = m^2,\ m = 1, 2, \ldots$, and proved the following:

Theorem 8.2.1 (Margulis [34])**.** *There exists a constant $w > 0$ such that for $m = 1, 2, \ldots,$ and $n = m^2$, G_n is a bipartite $(n, 5, w)$ expander.*

One huge downside to Margulis's result was that the constant w was not known. Years later, Gabber and Galil improved the construction and simplified its analysis. They were able to generate a family of expanders with similar properties with a constant value of $w = (2 - \sqrt{3})/4$ [20]. Their construction is exceedingly simple and proceeds as follows:

Definition 13 (Gabber and Galil Construction [20])**.** *Let $n = m^2$ and let A_m be $\{0, 1, \ldots, m-1\} \times \{0, 1, \ldots, m-1\}$. The bipartite graphs G_n are obtained from five permutations on A_m. The permutations are:*

$$\sigma_0(x, y) = (x, y),$$

$$\sigma_1(x, y) = (x, x + y),$$

$$\sigma_2(x, y) = (x, x + y + 1),$$

$$\sigma_3(x, y) = (x + y, y),$$

$$\sigma_4(x, y) = (x + y + 1, y),$$

where the $+$ is modulo m.

Thus, using this straightforward construction, it is possible to create expander graphs with $2n = 2m^2$ vertices, at most $5n$ edges, and with $|\Gamma_X| \geq [1 + ((2 - \sqrt{3})/4)(1 - |X|/n)]|X|$ [20]. Although the methodology it simple, it is also clear that the types of expanders that can be created using these permutations are somewhat limited.

It is also important to note that, since these graphs are not randomly generated, Alon's proof of the upper bound of the second eigenvalue being $2\sqrt{d-1}+\epsilon$ no longer holds [1]. However, Alon also has another proof that shows that the upper bound of the second eigenvalue of the adjacency matrix for a regular, bipartite (n, k, w) expander is

$$nk - \frac{w^2}{1024 + 2w^2} \quad [1]$$

For the Gabber-Galil graphs, we know that the expanders produced are $(n, 5, w_0)$ expanders, where $w_0 = (2-\sqrt{3})/4$ [20]. Unfortunately, since this value of d is so small, the upper bound on this eigenvalue is not very tight. Thus, doing eigenvalue testing on these graphs would likely not be very interesting. Thankfully, the bounds derived from the Cheeger inequalities for edge and vertex expansion rates are still valid for these graphs, so we can simply use those figures to benchmark expansion [9].

Later, families of Ramanujan graphs were able to be constructed explicitly due to work by Lubotzky, Phillips, and Sarnak [33]. For many of these constructions, the neighbors of all the vertices in the resulting graphs can be computed in constant time due to the explicitness of their generation. However, the eigenvalue bound analysis for the resulting graphs was extremely complicated, and thus, it is difficult to get an intuitive understanding of why the graphs are expanders [45]. Recently, a breakthrough in explicitly constructing expander graphs, the zig-zag graph product, has been published; we will describe this product in detail in the next section.

8.2.2 The Zig-Zag Graph Product: Preliminaries

In 2001, Reingold, Vadhan, and Wigderson introduced a new method for combining graphs, which they called the zig-zag product [45]. The method computes the product of two input graphs to generate a single output graph that inherits many of the properties, including expansion rates, of the inputs. Thus, by performing multiple iterations of zig-zag products, new constant-degree expanders can be generated from existing expanders.

In order to understand the construction of zig-zag products, we must first provide a few definitions used by Reingold et al.

Definition 14. *An (n, d, λ)-graph is any d-regular graph on n vertices, whose normalized adjacency matrix has second largest (in absolute value) eigenvalue at most λ. [45]*

Here, the term "normalized adjacency matrix" simply corresponds to the standard adjacency matrix of a d-regular graph having all of its entries divided by d. Reingold et al. also use objects called rotation maps to keep track of canonical orderings of edges at each vertex, which helps provide the maximum amount of notational generality to their final construction [45].

Definition 15. *For a d-regular, undirected graph G, the rotation map $Rot_G : [N] \times [d] \to [N] \times [d]$ is defined as follows: $Rot_G(v, i) = (w, j)$ if the i'th edge incident to v leads to w, and this edge is the j'th edge incident to w. [45]*

These rotational maps are used to specify graphs in the graph operations that

we will describe next. The zig-zag product itself is composed of two fairly standard graph operations: graph squaring and graph tensoring.

The first operation, graph squaring (or powering) works as follows. The t'th power of a d-regular graph G is a d^t-regular graph G^t, whose rotation map is $\mathrm{Rot}_{G^t}(v_0, (k_1, k_2, \ldots, k_t)) = (v_t, (\ell_t, \ell_{t-1}, \ldots, \ell_1))$, where $(v_i, \ell_i) = \mathrm{Rot}_G(v_{i-1}, k_i)$ [45]. From this definition, it is easy to see that, if G is an (n, d, λ) graph, then G^t is an (n, d^t, λ^t) graph. This is because the powering operation simply multiplies copies of G's adjacency matrix by itself, thus powering the number of edges in the graph, as well as scaling the eigenvalues by the same power, but keeping the number of vertices the same.

The second operation, graph tensoring, is slightly more complicated. In order to define it, we will start with two graphs, G_1 and G_2. Let G_1 be a d_1-regular graph on vertex set $[n_1]$ and let G_2 be a d_2-regular graph on vertex set $[n_2]$. Now we define the tensor product $G_1 \otimes G_2$ to be the $(d_1 d_2)$-regular graph on vertex set $[n_1] \times [n_2]$ given by $\mathrm{Rot}_{G_1 \otimes G_2}((v, w), (i, j) = ((v', w'), (i', j'))$, where $(v', i') = \mathrm{Rot}_{G_1}(v, i)$ and $(w', j') = \mathrm{Rot}_{G_2}(w, j)$ [45]. From this definition, we see that, if G_1 is an (n_1, d_1, λ_1)-graph and G_2 is a (n_2, d_2, λ_2)-graph, then the tensor product $G_1 \otimes G_2$ is a $(n_1 n_2, d_1 d_2, \max(\lambda_1, \lambda_2))$-graph [45]. Thus, the number of vertices in the tensor product is the sum of the number of vertices in the two factor graphs. Furthermore, the total number of edges increases from $\frac{n_1 d_1}{2} + \frac{n_2 d_2}{2}$ to $\frac{n_1 n_2 d_1 d_2}{2}$.

8.2.3 The Zig-Zag Graph Product: Definition and Recursion

Now that we have defined graph powering and graph tensoring, we can present the definition of the zig-zag product itself, which we will reproduce directly from Reingold et al. [45].

Definition 16. *If G_1 is a d_1-regular graph on $[n_1]$ with rotation map Rot_{G_1} and G_2 is a d_2-regular graph on $[d_1]$ with rotation map Rot_{G_2}, then their zig-zag product $G_1 \textcircled{z} G_2$ is defined to be the d_2^2-regular graph on $[n_1] \times [d_1]$ whose rotation map $Rot_{G_1 \textcircled{z} G_2}$ is as follows:*

$Rot_{G_1 \textcircled{z} G_2}((v,k),(i,j)) :$

1. *Let $(k', i') = Rot_{G_2}(k, i)$.*
2. *Let $(w', \ell') = Rot_{G_1}(v, k')$.*
3. *Let $(\ell, j') = Rot_{G_2}(\ell', j)$.*
4. *Output $((w, \ell), (j', i'))$. [45]*

Essentially, the zig-zag product is taking the vertices of G_1 and expanding them into a set of d_1 vertices, one corresponding to each edge of G_1 that is incident on v in the product graph. Furthermore, every edge in G_1 is associated with two vertices in the zig-zag product graph, one in each of the two vertex clouds that correspond to the endpoints of the edge in G_1.

Now, the main theorem proved by Reingold regarding these maps is as follows:

Theorem 8.2.2. *If G_1 is an (n_1, d_1, λ_1)-graph and G_2 is a (d_1, d_2, λ_2) graph, then*

$G_1 ⓩ G_2$ is a $(n_1 d_1, d_2^2, f(\lambda_1, \lambda_2))$-graph, where

$$f(\lambda_1, \lambda_2) = \frac{1}{2}(1 - \lambda_2^2)\lambda_1 + \frac{1}{2}\sqrt{(1 - \lambda_2^2)^2\lambda_1^2 + 4\lambda_2^2}$$

and $f(\lambda_1, \lambda_2) < 1$ when $\lambda_1, \lambda_2 < 1$ [45].

This theorem shows that the second eigenvalue of the zig-zag product is bounded above by reasonable parameters. Thus, although the normalized spectral gap will be smaller for zig-zag product graphs than their factor graphs, they still retain much of the expansion properties of those factor graphs.

Now, the question that remains is how this zig-zag product can be used to explicitly generate expander graphs. The answer lies in a recursive construction that involves all three graph operations that we have seen thus far: graph powering, tensor products, and the zig-zag product. The recursive method starts with an initial graph H which must be a (d^8, d, λ) graph for some d and some λ that is sufficiently small (to ensure good expansion) [45]. Reingold et al. suggest picking an H with $\lambda \leq \frac{1}{5}$. With that graph in hand, the recursive construction for generating a set of graphs $\{G_t\}$ starts by setting $G_1 = H^2$ and $G_2 = H \otimes H$. Then, for $t > 2$,

$$G_t = (G_{\lceil \frac{t-1}{2} \rceil} \otimes G_{\lfloor \frac{t-1}{2} \rfloor})^2 ⓩ H.$$

Using this construction, Reingold et al. were able to prove that, for every $t \geq 0$, G_t is an (d^{8t}, d^2, λ_t)-graph with $\lambda_t = \lambda + \mathcal{O}(\lambda^2)$ [45]. This shows that by using the recursive construction outlined above and starting with a sufficiently good H, it is possible to use the zig-zag graph product to generate an infinite family of expander graphs that get larger in size but whose spectral gaps only decrease by a

modest amount. The best eigenvalue bounds on the graphs constructed using the zig-zag method are $\mathcal{O}(1/d^{1/3})$, which is good, but still does not meet the Ramanujan bound [45].

8.2.4 The Zig-Zag Graph Product: In Practice

Although the recursive formulation of the expander graph generation using the zig-zag product is fairly straightforward, using the zig-zag product for real-world scenarios is much more difficult. For instance, consider the third graph produced by the zig-zag recurrence, G_3. From the definition of the recursive construction, we know that G_3 is a (d^{24}, d^2, λ_3) graph. Thus, even for a small value of d, such as $d = 3$, G_3 will have approximately 282 billion vertices. Clearly, the computational power and storage space needed to generate and manipulate these graphs, even for modest values of d and t, are simply too great for the zig-zag construction to be useful in practical applications.

Reingold et al. also provide a means for constructing a starting graph H directly [45]. Unfortunately, this construction is also computationally infeasible. It starts with a graph over the vertex set of $\mathbb{F}_q^2$, where q is a prime power, which is the set of all 2-tuples over the finite field F_q. This alone is manageable, but constructing H from that initial graph requires one tensor product of the graph with itself, followed by five rounds of recursively using the zig-zag product, which again will cause the number of vertices to increase enormously [45].

We implemented code that computes zig-zag products of graphs, as described

in Reingold et al. [45]. To our knowledge, this is one of the first implementations of the zig-zag product. It uses array functions from the numpy Python package to provide the adjacency matrix powering, and a helper script is used to compute tensor products of adjacency matrices. The zig-zag process itself, as well as the tensoring, are handled using generalized rotation maps, as outlined by Reingold et al. [45]. However, instead of imposing an arbitrary canonical ordering on the edges in order to generate a rotation map, we simply use a given edge's position in the row representing the source vertex as its index in the rotation map. For instance, if a vertex v is connected to vertices $4, 5,$ and 7, and if vertex 5 is connected to vertices $1, 2, 6,$ and v, then $(v, 1) \rightarrow (5, 3)$ would be in the rotation map for the graph, since, the second edge in the row of the adjacency matrix corresponding to vertex v is 5 (the rotation map is zero-indexed), and that edge corresponds to the fourth edge in the row of the adjacency matrix corresponding to vertex 5. Despite the fact that we have implemented this algorithm, since the construction itself is not useful for creating expander graphs with a relatively small number of vertices (in the hundreds), we were not able to use it for testing graph expansion.

8.3 Expander Construction Summary

Although randomly generated expander graphs are not useful for many theoretical applications, they result in graphs with better expansion parameters than their explicitly constructed counterparts. There is simply not yet an explicit construction that can match the upper bound on the second largest eigenvalue that

has been proven probabilistically for randomly generated d-regular graphs. Many of the methods for explicitly creating expander graphs are fairly straightforward, but either lead to creating a hamstrung family of expanders or are not able to be used in practice. In the next chapter, we will describe one potential method for developing a measure that can be used to compare the expansion rates of graphs.

Chapter 9

Graph Generation, Eigenvalue Testing, and Expansion Evaluation

As the previous chapters have illustrated, expander graphs are extremely important to many different fields, but they are also difficult to use in practice. While probabilistic existence proofs show that they are abundant, it is hard to verify, or even approximate, a graph's expansion rate. In order to help make it easier to understand and use expander graphs in practical applications, we have written a software suite in Python using the numpy and scipy packages that provides tools for expander graph generation (both random and explicit constructions), eigenvalue testing, empirical expansion testing, and expansion approximation. We used this software to perform significant testing and analysis of random graph generation and graph expansion. We will provide a brief discussion of the software, after which will describe our experiments and methodology, provide our data, and analyze our findings.

9.1 Graph Generation and Evaluation

The first pieces of software in the suite are graph generation scripts. The script for generating random graphs implements the graph generation algorithm described in Friedman [17]. This algorithm creates random d-regular graphs on n vertices by selecting $\frac{d}{2}$ permutations on the set $\{1, 2, \ldots n\}$ uniformly at random.

Each permutation p can be thought of as the image of a map from the set of vertices in some canonical order $\{1, 2, \ldots, n\}$ to p; thus, each permutation defines n edges. The script accepts n and d as parameters, and will generate a graph for all valid parameter values. The graph itself is stored in a numpy array in adjacency matrix form, which is then written in binary format (as required by numpy) to a file. For the cases where d is odd, we use a combination of Friedman's method and the half-edge method found in Kim and Vu [17, 29]. Those graph files can then be loaded and read by numpy in follow-on programs.

In addition to random graph generation, we also have written code that implements the explicit construction found in Gabber and Galil [20]. As is required by their algorithm, n must be a perfect square. Similar to the random generation script, the output graph is stored in a numpy array that is written to disk. For uses of the Gabber-Galil (GG) graphs that do not require the adjacency matrix, we also have utility functions that can be used to calculate vertex adjacencies for GG graphs in constant time.

We have also written additional scripts to generate "bad" graphs that we created that have known bad expansion. One example of this is a script that generates a cycle graph on n vertices, which also accepts a d parameter (where $d \geq 2$). If d is larger than two, then a self-loop is added to each vertex, but the vertices themselves are only connected to the two neighbors next to them in the cycle.

After writing programs for generating graphs, we also wrote a utility module that computes the eigenvalues of a graph's adjacency matrix. After computing the eigenvalues, it sorts them in descending order and then takes their absolute

values, showing the ten largest eigenvalues. As mentioned extensively throughout the paper, the absolute value of the second largest eigenvalue of a graph's adjacency matrix is a good way to estimate the graph's expansion capabilities. This script was used in combination with the random graph generation script (it takes a binary file containing a numpy array as input) to enable our testing, which we will discuss in the analysis section.

Before describing the various tests we performed, we will first outline how we used our software to generate expanders. We started by first generating graphs for various values of n and d, saving a handful at each combination for further testing. For $n = 100$, we generated $5,000$ graphs each for $d = 2$, $d = 4$, and $d = 6$. For each of these three values, we computed the second largest eigenvalue for every graph generated and computed the median and mean of the second eigenvalues. We also saved the graphs corresponding to the lowest and highest eigenvalue for each combination, as well as three other randomly selected graphs, for a total of 15 graphs. We then repeated this process for $n = 200$; we generated $15,000$ graphs total and saved another 15. The results of this generation can be found in Tables 9.1 and 9.2. After generating the graphs and using their calculated eigenvalues to save the best graph, worst graph, and three other graphs for each n and d combination, we started performing some empirical expansion testing.

Table 9.1: Graph Generation Results

d	n	#Graphs	Min	Max	Average	Median	$2\sqrt{d-1}$
4	100	5000	3.28804	4.00000	3.45528	3.44783	3.46410
6	100	5000	4.19232	5.02061	4.44508	4.43706	4.47214
8	100	5000	4.94387	6.37714	5.24823	5.24280	5.29150
4	200	5000	3.36599	4.00000	3.45932	3.45442	3.46410
6	200	5000	4.29392	4.99664	4.45581	4.45105	4.47214
8	200	5000	5.08165	6.37413	5.26645	5.26107	5.29150

Table 9.2: Graph Generation Results (Cont'd)

d	n	Min (norm)	Max (norm)	$\frac{2\sqrt{d-1}}{d}$	Ramanujan Diff. (norm)
4	100	0.82201	1.00000	0.86603	0.04402
6	100	0.69872	0.83677	0.74536	0.04664
8	100	0.61798	0.79714	0.66144	0.04345
4	200	0.84150	1.00000	0.86603	0.02453
6	200	0.71565	0.83277	0.74536	0.02970
8	200	0.63521	0.79677	0.66144	0.02623

9.1.1 Data

9.1.2 Analysis

Tables 9.1 and Table 9.2 contain statistics related to the collections of random graphs that we generated. The first two columns of both tables show the values of d and n that were used to generate the graphs. For each combination of d and n, we generated $5,000$ graphs and recorded information about the second eigenvalues of those graphs. In addition to tracking the minimum second eigenvalue (which, intuitively, should correspond to higher expansion), we also tracked the maximum, average and median eigenvalue. The column on the far right of Table 9.1 contains the value of the expression $2\sqrt{d-1}$ for the d that represents the graphs in each

row. This is the value that was proved by Friedman to bound the second largest eigenvalue of most random regular graphs [18]. Thus, comparing the Min column to the far right column gives us an idea of how far the best second eigenvalue was from this upper bound. It might seem surprising initially that the rows for the $n = 200$ graphs have slightly worse eigenvalues, but that is likely due to the fact that, for relatively small values of d, a graph with 200 vertices is simply too large for the edge boundaries of vertex sets to scale in size with the sizes of those sets.

In the second of these two generation-related tables, Table 9.2, we normalized the minimum, maximum, and Friedman bound for each set of graphs by dividing them by d, which conveniently maps them into the interval [0,1], since we know the second eigenvalue can be at most d. The far right column is simply the difference between the normalized minimum and the normalized Friedman bound. As we discussed earlier, a graph whose second eigenvalue is less than the normalized Friedman bound is said to be a Ramanujan graph; thus, this difference represents the distance the optimal eigenvalue for a given set of graphs is from this bound. As the first table showed, the $n = 200$ graphs are slightly closer to the bound than their $n = 100$ counterparts, due to the increase in the number of vertices without a corresponding increase in d. Also, as we would expect, as the value of d increased, for a given value of n, the minimum and maximum eigenvalues decreased, indicating improved expansion.

9.2 Vertex Expansion Testing

One of the initial goals for using our software to empirically analyze expander graphs was to simply sample various subsets of randomly generated graphs and compare the observed vertex expansion rates of those subsets to the theoretical expectations for vertex expansion. Instead of computing vertex expansion directly, we simply computed the size of the neighborhood of a given subset, which consists of all the vertices in the subset, along with any other vertices in the graph that are adjacent to a vertex in the subset.

For a given vertex in a d-regular graph with n vertices, it is straightforward to see that the likelihood that it will not be adjacent to another vertex, s, is $\left(\frac{n-1}{n}\right)^d$. In the more general case, we are interested in the likelihood that a given vertex will not be adjacent to any vertex in some subset $S \subseteq V$. We will let ϵ represent the size of the subset S as a percentage of n such that $|S| = n\epsilon$. Thus, we now have that, for a given vertex v and a given subset S in the graph G, the odds that v is not adjacent to any vertex in S is

$$\left(\frac{n-1}{n}\right)^{d\epsilon n}.$$

From here, we see that

$$\lim_{n\to\infty}\left(1-\frac{1}{n}\right)^{d\epsilon n} = e^{-d\epsilon}.$$

Now, for each vertex $V_i \notin S$, we can create a random variable X_i such that

$$X_i = \begin{cases} 1 & \text{if vertex } v_i \text{ is adjacent to some vertex in } S. \\ 0 & \text{otherwise.} \end{cases}$$

Thus, the number of vertices outside of S that are connected to S is simply $\sum_{v_i \notin S} X_i$. We can then use the linearity of expectation as follows:

$$\begin{aligned} E\left(\sum_{v_i \notin S} X_i\right) &= \sum_{V_i \notin S} E(X_i) \\ &\approx \sum_{V_i \notin S} 1 - e^{-d\epsilon} \\ &= n(1-\epsilon)(1-e^{-d\epsilon}) \end{aligned}$$

Now, we see that we can calculate the expected size of a neighborhood, which is simply $|S| + n(1-\epsilon)(1-e^{-d\epsilon})$, or, the size of the neighborhood plus the size of the expected number of vertices outside of S that are adjacent to a vertex in S. Table 9.3 shows these expected sizes for the values of d, n, and ϵ that were also tested empirically. The empirical results can be found in Tables 9.4 and 9.5.

Table 9.3: Neighborhood Sizes – Theoretical Expectation

D	N	ϵ	Expected Neighb. Size
4	100	0.25	72.409
6	100	0.25	83.265
8	100	0.25	89.850
4	200	0.125	93.857
6	200	0.125	117.336
8	200	0.125	135.621

As these results show, the average neighborhood size observed in our testing was slightly larger than the theoretically expected neighborhood size. This discrepancy can likely be attributed to a couple different factors. First, since the random graphs we generated were d-regular, the possible configurations of the graphs are much more restricted than if a graph were generated truly randomly. Although

Table 9.4: Neighborhood Sizes – Empirical; $n = 100$

D	N	ϵ	$\lambda(G)$	Trials	Min	Max	Average	Median
4	100	0.25	3.288	250000	61	90	76.716	77
4	100	0.25	4.000	250000	60	90	76.170	76
4	100	0.25	3.475	250000	61	90	76.214	76
4	100	0.25	3.420	250000	60	90	76.366	76
4	100	0.25	3.373	250000	62	90	76.727	77
6	100	0.25	4.192	250000	73	98	86.933	87
6	100	0.25	5.021	250000	71	98	86.373	87
6	100	0.25	4.486	250000	73	98	86.344	87
6	100	0.25	4.290	250000	73	98	86.974	87
6	100	0.25	4.496	250000	73	98	87.017	87
8	100	0.25	4.944	250000	80	100	92.794	93
8	100	0.25	6.377	250000	80	100	92.233	93
8	100	0.25	5.373	250000	80	100	92.090	92
8	100	0.25	5.289	250000	79	100	92.431	93
8	100	0.25	5.297	250000	80	100	92.578	93

Table 9.5: Neighborhood Sizes – Empirical; $n = 200$

D	N	ϵ	$\lambda(G)$	Trials	Min	Max	Average	Median
4	200	0.125	3.366	250000	80	115	97.882	98
4	200	0.125	4.000	250000	77	115	96.710	97
4	200	0.125	3.419	250000	79	114	97.221	97
4	200	0.125	3.436	250000	78	114	97.410	98
4	200	0.125	3.436	250000	78	114	97.553	98
6	200	0.125	4.294	250000	103	143	122.191	122
6	200	0.125	4.997	250000	101	144	121.178	122
6	200	0.125	4.402	250000	101	142	121.848	122
6	200	0.125	4.478	250000	101	141	121.118	121
6	200	0.125	4.392	250000	100	143	121.744	122
8	200	0.125	5.082	250000	118	162	140.545	141
8	200	0.125	6.374	250000	117	159	139.449	140
8	200	0.125	5.283	250000	119	161	139.924	140
8	200	0.125	5.282	250000	119	160	139.739	140
8	200	0.125	5.423	250000	119	160	139.994	140

we took the d-regularity into account from the point of view of calculating adjacency probabilities to develop the expected values, there is no straightforward way

to tweak the expected values to account for the more restricted and uniform graph possibility space. Second, although the methods we used for generating random graphs are those most commonly found in the literature, it is possible that using the $\frac{d}{2}$ permutations method yields a slightly different distribution of possible graphs than what would be expected from a truly random d-regular graph generator [17].

Examining our empirical data in Tables 9.4 and 9.5, it is also interesting to note that higher observed neighborhood sizes are closely correlated with smaller second eigenvalues. As we have described before, the size of the second eigenvalue determines the size of the spectral gap of the graph's adjacency matrix. As the size of this gap increases, we expect the corresponding graphs to have better expansion properties. Thus, our results show that this is indeed the case in practice. Graphs that demonstrated, on average, the best vertex expansion (based on the observed neighborhood sizes), also tended to have smaller second eigenvalues. Thus, we can conclude that the distribution of random graphs produced by Friedman's method is not quite flat, but is relatively close. We have also shown that sampling graph subsets and averaging their vertex expansion rates provides a set of values that is correlated with the second eigenvalue of the graph's adjacency matrix, which means that it is also correlated with the true overall expansion rate of the graph. After completing this testing, we performed additional empirical tests that sought to find an alternative way to evaluation graph expansion rates, which we discuss in the next section.

9.3 Subset Testing

Since, as we have discussed in the preceding chapters, it is NP-hard to compute exact edge and vertex expansion rates, as well as UG-hard to approximate expansion to within a constant factor, we developed a novel method of assessing graph expansion. Rather than simply randomly selecting subsets and determining their expansion rates, we identify a set of "marked" vertices, and then randomly select subsets and determine the size of the intersection between the neighborhoods of the subsets and the marked vertices. In this way, we have significantly reduced the chance of selecting only "good" subsets for testing expansion, since we are no longer concerned with just the size of a given subset's neighborhood. Since the marked vertices are selected at random (and since we repeat the experiments with many different sets of marked vertices), a graph that only expands well in certain places will have a much smaller intersection in cases where vertices in the poorly expanding areas are marked. Thus, the average size of this intersection, or, alternatively, the expected number of marked vertices in the neighborhood of a subset, can be used to compare one graph's expansion to another. This method does not provide an actual estimation for edge or vertex expansion; rather, it provides a value that can be used to compare expansion behavior.

After developing this model, we wrote Python scripts to enable the empirical testing of graphs using this method in an attempt to differentiate good expanders from poor expanders. This method works as follows: first, a certain number (user-specified as an argument) of vertices are marked as "red" vertices (we will use

"marked" and "red" interchangeably to refer to these vertices). Next, a random subset of vertices of a user-defined size is selected from amongst all the graph vertices (red and otherwise). The script then calculates the number of unique red vertices amongst the neighbors of the vertices in the selected subset, as well as the number of red vertices in the subset itself. The number of red vertices found in this manner is recorded for each subset sample across multiple selections of marked vertices. After all of the testing is performed, the program prints, for each possible count of red vertices in a sample, the percentage of the sampled subsets with that count. It then calculates the expected number of red vertices by taking a dot product of these percentages with the set of natural numbers.

This empirical testing might seem slightly unintuitive at first, but by tweaking the number of marked vertices and the size of the subsets we sample, we can essentially test how much expansion we can get from a random subset of the graph. Checking all the subsets is what makes this problem NP-hard in the first place, so we simply sample a portion of subsets of various sizes and for various percentages of marked vertices. Since expander graphs are supposed to be well-connected, we would expect, for instance, that the neighborhoods of subsets of expander graphs would have a larger intersection with the set of marked vertices than neighborhoods of subsets of a non-expanders. We will discuss the results of this testing in subsequent sections.

9.3.1 Expansion Evaluation Methodology

In the first type of empirical test, we used a red vertex count size of 20 for the $n = 100$ graph and size 40 for the $n = 200$ graphs. We randomly selected 500 different red vertex subsets, and for each subset, we sampled all of the size 1 subsets and determined how many red neighbors were contained in the neighbors of the selected vertex and the vertex itself. For example, for the $n = 100$ graphs, we calculated how many red vertices were found in the neighborhoods of all n possible single-vertex subsets. The results of this testing can be found in Table A.3 in Appendix A; Table 9.6 contains a summarization of the results. For the $n = 200$ graphs, we performed the same singleton subset checking, except now 200 single-vertex subsets had to be sampled for each of the 500 red vertex subsets. The data from these tests can be found in Table A.4 in Appendix A; Table 9.7 contains the summarized results. After tabulating all of the counts from all of the subset checking, we calculated the observed percentage for each possible number of red vertices found. We then used those percentages to calculate the expected number of red vertices that we would see in the neighborhood of a single vertex subset.

In addition to checking single-vertex subsets, we also performed checking of multi-vertex subsets. Specifically, for the $n = 100$ graphs, we again selected 500 subsets of 20 red vertices, and then, for each of those 500 subsets, we sampled 500 subsets of size 25 and counted how many red vertices were in that set or the neighbors of that set. Much like the single-vertex cases, we calculated the percentages for each possible count of red vertices found, and then used that to calculate the expected

number of red vertices (as a weighted average) in the neighborhood of a 25 vertex subset. We were also able to use the theoretical results about neighborhood sizes (in Table 9.3) to compute the theoretically expected number of red vertices that would be found in each neighborhood. The full testing results can be found in Table A.5 in Appendix A; an abbreviated version of the results is presented in Table 9.8. For the $n = 200$ graphs, we performed the exact same subset sampling tests using the same subset sample size of 25 vertices. Those results can be found in Table A.6 in Appendix A; abbreviated results are presented in Table 9.9. In the next section, we provide the data collected during our testing. Following that, we provide an analysis and discussion of the data.

9.3.2 Data

Table 9.6: Singleton, $|S| = 1$; $n = 100$

D	N	# TRIALS	# RED	$\lvert S\rvert$	$\lambda(G)$	$E(\text{REDS})$
4	100	500	20	1	3.288	0.99220
4	100	500	20	1	4.000	0.98440
4	100	500	20	1	3.475	0.98000
4	100	500	20	1	3.420	0.98350
4	100	500	20	1	3.373	0.99220
6	100	500	20	1	4.192	1.37362
6	100	500	20	1	5.021	1.35840
6	100	500	20	1	4.486	1.34764
6	100	500	20	1	4.290	1.37216
6	100	500	20	1	4.496	1.37814
8	100	500	20	1	4.944	1.75554
8	100	500	20	1	6.377	1.72698
8	100	500	20	1	5.373	1.70852
8	100	500	20	1	5.289	1.72878
8	100	500	20	1	5.297	1.74006

Table 9.7: Singleton, $|S| = 1$; $n = 200$

D	N	# TRIALS	# RED	$\lvert S\rvert$	$\lambda(G)$	$E(\text{REDS})$
4	200	500	40	1	3.288	0.99604
4	200	500	40	1	4.000	0.98193
4	200	500	40	1	3.475	0.98841
4	200	500	40	1	3.420	0.98984
4	200	500	40	1	3.373	0.99178
6	200	500	40	1	4.192	1.39205
6	200	500	40	1	5.021	1.37607
6	200	500	40	1	4.486	1.38618
6	200	500	40	1	4.290	1.37306
6	200	500	40	1	4.496	1.38385
8	200	500	40	1	4.944	1.78098
8	200	500	40	1	6.377	1.76715
8	200	500	40	1	5.373	1.76237
8	200	500	40	1	5.289	1.76839
8	200	500	40	1	5.297	1.75842

Table 9.8: Subset Sampling, $|S| = 25$; $n = 100$

D	N	# TRIALS	# RED	# SUBSETS	$\|S\|$	$\lambda(G)$	E(REDS)	THEOR. REDS
4	100	500	20	500	25	3.288	15.347	14.482
4	100	500	20	500	25	4.000	15.250	14.482
4	100	500	20	500	25	3.475	15.247	14.482
4	100	500	20	500	25	3.420	15.276	14.482
4	100	500	20	500	25	3.373	15.339	14.482
6	100	500	20	500	25	4.192	17.389	16.653
6	100	500	20	500	25	5.021	17.283	16.653
6	100	500	20	500	25	4.486	17.270	16.653
6	100	500	20	500	25	4.290	17.398	16.653
6	100	500	20	500	25	4.496	17.402	16.653
8	100	500	20	500	25	4.944	18.557	17.970
8	100	500	20	500	25	6.377	18.446	17.970
8	100	500	20	500	25	5.373	18.414	17.970
8	100	500	20	500	25	5.289	18.483	17.970
8	100	500	20	500	25	5.297	18.519	17.970

Table 9.9: Subset Sampling, $|S| = 25$; $n = 200$

D	N	# TRIALS	# RED	# SUBSETS	$\|S\|$	$\lambda(G)$	E(REDS)	THEOR. REDS
4	200	500	40	500	25	3.366	19.572	18.771
4	200	500	40	500	25	4.000	19.354	18.771
4	200	500	40	500	25	3.419	19.440	18.771
4	200	500	40	500	25	3.436	19.498	18.771
4	200	500	40	500	25	3.436	19.515	18.771
6	200	500	40	500	25	4.294	24.431	23.467
6	200	500	40	500	25	4.997	24.227	23.467
6	200	500	40	500	25	4.402	24.371	23.467
6	200	500	40	500	25	4.478	24.213	23.467
6	200	500	40	500	25	4.392	24.356	23.467
8	200	500	40	500	25	5.082	28.107	27.124
8	200	500	40	500	25	6.374	27.897	27.124
8	200	500	40	500	25	5.283	27.988	27.124
8	200	500	40	500	25	5.282	27.951	27.124
8	200	500	40	500	25	5.423	27.992	27.124

9.3.3 Singleton Subset Testing Analysis

Table A.3 and Table A.4 contain the statistics related to checking all n single-vertex subsets for each graph tested. Here, the number of trials refers to the number of different red vertex selections that were made. For each such selection, all n subsets were checked, for a total of $50,000$ subsets for the $n = 100$ graphs in Table A.3. The $\lambda(G)$ column is the second largest eigenvalue (in absolute value) of the graph's adjacency matrix. To the right of that column, the "1 red," "2 red," etc. columns denote the percentage of the $50,000$ subsets tested where the number of distinct red vertices present in the singleton subset and amongst the neighbors of that vertex equaled the number at the top of the column. Thus, for these singleton subsets, there could be at most $d + 1$ red vertices for a given subset, which would represent the case where the selected vertex and all of its neighbors were red. The column on the far right, $E(\text{REDS})$, is the expected number of reds we would find amongst a single vertex and its neighbors. The values in this column are calculated by simply taking the cross product of the percentages in the preceding columns with set of non-negative integers, $[0, 1, 2, \dots]$.

Now, examining Table A.3 more closely, we will consider each set of five rows with the same d value. For the $d = 4$ graphs, we see that the graph with the best (smallest) second eigenvalue has the highest expected number of reds out of the five, although one of the three randomly selected graphs has the same expectation. The graph with the worst second eigenvalue does not have the lowest expected number of reds, although it does have the highest percentage of 0 red subsets.

Likewise, with the $d = 6$ graphs, the best second eigenvalue has a significantly higher expected number of reds than the worst, but it is beaten slightly by one of the randomly selected graphs. For the $d = 8$ graphs, the graph with the best second eigenvalue does have the highest expected number of reds, but the lowest number of expected reds does not come from the graph with the worst second eigenvalue. Now, examining the table as a whole, we see that the expected number of reds increased as d increased, which is to be expected, since it increases the maximum number of neighbors that a given vertex can have. This data suggests that there is a correlation between the second eigenvalue and the expansion rate (which we are measuring using the expected number of reds based on our sampling), but clearly, there is not a direct correlation. The best eigenvalue does not necessarily always correspond to the highest expected number of reds. This is likely due to the fact that, since we are only examining subsets of size 1, we are not able to observe the true effects of the graphs' expansion properties.

Table A.4 has similar contents to Table A.3, except now we are dealing with the $n = 200$ graphs. Much like the $n = 100$ graphs, we see here that, for a given value of d, the graph with the best eigenvalue also has the highest number of expected reds, what is exactly what we would anticipate. For $d = 4$, we see an even stronger relationship, which is that the five graphs' expectations are ordered inversely to the size of their eigenvalue. Again, like the $n = 100$ graphs, we see that the number of expected reds rises as d increases. One very interesting thing to note is that, when comparing the expected number of reds in Table A.4 with the expected number of reds in Table A.3, we see that the $n = 200$ graphs have slightly higher expectations

than their $n = 100$ counterparts. For instance, the highest expectation for $d = 4$ amongst the $n = 200$ graphs is 0.99604, whereas the highest expectation for $d = 4$ amongst the $n = 100$ graphs is 0.99220. For $d = 6$, we have 1.39205 as compared to 1.37362, and for $d = 8$, we have 1.78098 versus 1.75554. These results are somewhat interesting in that, even though the percentage of red vertices are the same in both the $n = 100$ and $n = 200$ graphs, the $n = 200$ graphs appear to have observably better expansion. This is somewhat surprising due to the fact that, as Table 9.1 and Table 9.2 indicated, the $n = 200$ graphs have a smaller Ramanujan difference and smaller normalized minimum than their $n = 100$ counterparts. Thus, we might expect that the number of expected reds would be slightly higher in the $n = 100$ graphs; however, our empirical testing clearly indicates that there are more factors at play. It is possible that the discrepancy can be attributed to the fact that the size of the subset we are testing is smaller relative to the size of the graph for $n = 200$.

9.3.4 Subset Sampling Testing Analysis

The next table, Tables A.5, illustrates the results of our subset sampling tests for graphs with 100 vertices. For these graphs, we performed 500 subset samplings, each of size 25 for each of 500 different selections of 20 red vertices, for a total of $250,000$ samples per graph. Since the size of the sampled subset was larger, the maximum number of red vertices that could be found within the subset and the neighbors of the subset is $|S| + d|S| + 1$, which, for the case of $|S| = 25$, is obviously larger than 20. Note that, since all of the results could not fit onto a single page,

Table A.5 is spread across two pages.

Now, examining the first five rows of the table, which represent the $d = 4$ case, we see that the expected number of reds appears to be well-correlated with the value of the second eigenvalue. The graph with the smallest eigenvalue has the highest expected number of reds, and that number goes down as the eigenvalues increase. For the $d = 6$ cases, the graph with the smallest eigenvalue again has a higher number of expected reds than the graph with the worst eigenvalue, but there is slightly more fluctuation in the correlation. The same is true for the $d = 8$ graphs; in fact, the graph with the smallest second eigenvalue actually has the highest number of expected reds out of all the graphs that were tested. Also, examining the table as a whole, we again see that the number of expected reds increases as d increases, which is exactly what we would expect. Having a larger value of d increases the size of the neighborhood of the subset we are sampling, thus increasing the potential number of other vertices that are checked.

The next table, Tables A.6, contains the results of subset sampling tests for the $n = 200$ graphs. For these tests, we again used the same number of red vertex subsets, but instead of marking 20 vertices as red, we marked 40 vertices as red in order to keep the percentage of red vertices constant. This table also spans multiple pages.

For the $d = 4$ graphs, the graph with the lowest eigenvalue again had the highest number of expected reds. The same also held true for the $d = 6$ and $d = 8$ graphs, with an overall trend of having lower eigenvalues associated with higher expectations, though the relationship is not perfect. We also see that, as with the

$n = 100$ graphs, the number of expected reds is increasing as d increases, which is what we expect to happen. We also see that the number of expected reds is higher, for the same values of d, than the expectation of their $n = 100$ counterparts, though the percentage of reds is significantly lower. This indicates that, for relatively small values of d, increasing the size of the graph does not necessarily lead to a higher expectation.

It is also interesting to note that, for both the $n = 100$ and $n = 200$ graphs, the empirically expected number of red vertices were very close to the theoretical number of red vertices that we would expect to find in neighborhoods of similar size. This shows that our method for randomly generating graphs works reasonably well, and also demonstrates the existence of some expansion properties.

9.4 Conclusion

In the previous sections, we have described multiple methods for empirically analyzing expander graphs. In the first section, we showed how our testing indicated that the distribution of random graphs generated by the Friedman method is likely not quite flat due to the discrepancies between the observed neighborhood sizes and expected neighborhood sizes [17]. We also saw that the average neighborhood sizes correlated strongly with the second eigenvalue of graph adjacency matrices, illustrating that neighborhood size testing is a viable means for assessing graph expansion. It is also worth noting, that, if we were to subtract the size of the subset from the size of its neighborhood and then divide by the size of the subset, we

could use the minimum value over all of the tests as a new lower bound on vertex expansion. We will see a similar idea used in the next chapter to improve lower bounds on expansion rates.

In the second section, our results also indicated that we developed another measure that is well-correlated with the second eigenvalue of graph adjacency matrices. Thus, we can also use this expected number of red vertices to compare one graph's expansion rate to another. However, this method is far from perfect, and most importantly, it does not provide an actual estimation for the graph's true expansion rate. In the following chapter, we will discuss another method for assessing graph expansion that does provide actual bounds on the true expansion rates of graphs.

Chapter 10

Graph Modularity Testing and Results

As outlined in Chapter 7, community detection algorithms that maximize modularity are potentially useful in identifying subsets with poor expansion. If low expansion subsets can be identified by modularity maximization algorithms, then they can be used to improve upper bounds on the overall vertex and edge expansion of the graph in question. In order to test this theory, we performed empirical testing of these algorithms on a number of different expander graphs, both those that are randomly generated and those that are created via explicit constructions. We computed the expansion of the subsets identified by the community detection algorithms and then compared the lowest expansion rates found with the theoretical upper and lower bounds for expansion.

10.1 Methodology

The first step in performing testing of modularity maximization algorithms was to generate the graphs that would serve as input to the community detection algorithms. First, we used our random graph generation algorithm (which uses the $\frac{d}{2}$ random permutations on $\{1, \ldots n\}$ for even d, and combines that method with the half-edge method for odd d) to generate ten 5-regular graphs, one for each multiple of 100 from 100 up to 1000 [16, 29]. After creating the randomly generated graphs,

we used the Gabber-Galil (GG) construction to create nine additional graphs [20]. The values of m used for these graphs were the odd numbers from 7 through 23, inclusive. For a given value of m, the corresponding GG graph contains $2m^2$ vertices, so the graphs we generated ranged in vertex count from 98 to 1058 to provide a similar range between the randomly generated and explicitly constructed graphs. It should also be noted that we selected all of the randomly generated graphs to be 5-regular since the GG construction we utilized can only be used to generate 5-regular graphs. These graphs were created as numpy adjacency matrix objects, which were then written to disk so that they could be read into follow-on scripts.

The actual community detection algorithms were run using the statistical software R. The open-source modMax package in R contains implementations of all of the major community detection algorithms that use modularity maximization and, when applicable, includes the ability to tweak the parameters of the algorithms [46]. In order for the graphs to be loaded into R, we wrote a helper script in Python that loaded numpy matrices and then wrote them back to files as comma-separated values, which could then be loaded into R as an adjacency matrix, where they were then transformed into a graph object from the igraph R package [12]. These objects were then passed as parameters to the various community detection algorithms.

After generating the graphs and developing a workflow for loading the graphs into R, we started our testing by evaluating the various modularity maximization algorithms that are available in the modMax package to get an idea of their run times and their potential for identifying bad communities. For this testing, we used the small, 100-vertex randomly generated graph, since we figured that even the slow

algorithms would be able to converge to a network clustering fairly quickly.

For this research, we used the UMBC High Performance Computing Facility's maya cluster, which contains over 300 nodes, seventy-two of which have two eight-core 2.6 GHz Intel E5-2650v2 Ivy Bridge CPUs and 64 GB memory. Nineteen of the seventy-two are hybrid nodes that contain two NVIDIA K20 graphics processing units, and another ten of which contain two 60-core Intel Xeon Phi 5110P accelerators. All of the cluster's nodes are connected to over 750 TB of storage via InfiniBand.

The algorithms that we benchmarked using the 100-vertex graph were extremalOptimization, geneticAlgorithm, greedy, simulatedAnnealing, and spectralOptimization [46]. For each algorithm, we kept track of the total number of communities the algorithm discovered and the average size of the communities. We also calculated the vertex and edge expansion of each community using functions from the igraph package, and we kept track of the worst expansion rates over the entire community structure and the number of vertices in that worst community. It should also be noted that, in the cases where the algorithm found a community that contained more than half of the vertices in the graph, instead of processing the community itself, we inverted the community and considered all the vertices not in the original community to be their own community. This was done to ensure that the expansion rates of the communities could be valid edge and vertex expansion rates, and the definitions of those rates only considers communities that contain no more than half of the vertices in a graph. Additional statistics, like the median expansion rates, and calculated values, like the Cheeger upper bound for vertex

expansion and the spectral bounds for vertex expansion were also computed and output by our testing script. Finally, we calculated the percentage improvement that the worst community's edge expansion provides over the spectral upper bound. Since the actual edge expansion rate is a minimum taken over all subsets of vertices that contains at most half of the total number of vertices in the graph, if one of the community detection algorithms finds a community whose edge expansion is worse than the theoretical spectral bound, then we have improved that upper bound since the edge expansion of the graph cannot be more than the edge expansion of the community that was identified.

Table 10.1: Community Detection – Algorithm Timing

N	GEN TYPE	λ_2	ALGORITHM	# COMMS	AVERAGE COMM. SIZE	MIN V. EXP.	MIN E. EXP.	TIME (SECS)
100	RANDOM	3.7618	EXTREMALOPTIMIZATION	23	4.35	3.00	3.33	4082.37
100	RANDOM	3.7618	GENETICALGORITHM	42	2.38	2.60	2.67	1569.67
100	RANDOM	3.7618	GREEDY	8	12.50	1.81	1.94	12.88
100	RANDOM	3.7618	SIMULATEDANNEALING	100	1.00	3.00	3.00	2.53
100	RANDOM	3.7618	SPECTRALOPTIMIZATION	20	5.00	2.6	3.67	116.46

The results of these initial benchmarks can be found in Table 10.1, from which we made a number of interesting observations. First, the extremalOptimization and geneticAlgorithm experiments took an extremely long time, 68 minutes and 26 minutes, respectively, especially when considering the modest size of the graph and the power of the nodes on which these experiments were running. Second, of the five algorithms, the greedy and geneticAlgorithm tests performed significantly better than the others when it came to finding communities with low vertex and edge expansion. Third, even though the simulated annealing algorithm ran very quickly,

an examination of the number and size of the communities it found reveals that the algorithm simply terminated without merging any of the communities from the initial structure where every vertex is in its own community. The alpha and beta parameters for the simulatedAnnealing run were 1.005 and half the number of vertices, respectively, which are the standard values. We did some further exploration and tweaking of these algorithms, but the results remained unimpressive and their run times quickly increased. Thus, based on these results, we decided to focus on using the greedy method of modularity maximization since it performed the best of all the algorithms we tested and it ran in a reasonable amount of time.

Once we settled on greedy algorithms, we began benchmarking various greedy algorithm variants that were also contained in the modMax package [46]. These variants included greedy, which is the standard Clauset-Newman-Moore algorithm; rgplus, which identifies core groups to create an initial partitioning; msgvm, which performs multiple merges in a given step and uses greedy refinement; cd, which performs complete greedy refinement iteratively and then moves vertices based on a provided probability to avoid landing in local optima; Louvain, which was described earlier in a previous section; and mome, which uses a combination of coarsening and uncoarsening phases to reconstruct community structure [46]. Much like the initial algorithm testing, these greedy algorithm benchmark tests were performed using the 100-vertex randomly generated graph. The statistics for these tests can be found in Table 10.2.

Examining Table 10.2, we see that the best-performing algorithms were greedy, rpgplus, and Louvain. It is also interesting to note that the mome algorithm con-

Table 10.2: Community Detection – Greedy Algorithm Timing

N	GEN TYPE	λ_2	ALGORITHM	# COMMS	AVERAGE COMM. SIZE	MIN V. EXP.	MIN E. EXP.	TIME (SECS)
100	RANDOM	3.7618	GREEDY	8	12.5	1.8095	1.9412	9.4609
100	RANDOM	3.7618	RGPLUS	7	14.2857	1.5000	1.7273	29.5869
100	RANDOM	3.7618	MSGVM	6	16.6667	1.4583	1.7500	270.4000
100	RANDOM	3.7618	CD	10	10	2.0909	2.2727	2505.1517
100	RANDOM	3.7618	LOUVAIN	2	50	0.7959	1.1633	110.2937
100	RANDOM	3.7618	MOME	1	100	NA	NA	20.0359

verged to selecting only a single community that contained all 100 vertices. Behavior like this is not uncommon with modularity maximization algorithms and is known as the resolution limit problem [10]. We also note that the cd algorithm took an order of magnitude longer than the other variants, likely because the time required to perform a complete greedy refinement on every partition along its path to convergence is quite significant. Based on these results, we settled on using the greedy, rgplus, and Louvain variants for our testing; however, after doing some more benchmarking tests, we began to run into issues where the function call to the rgplus algorithm would be issued properly and then return an error code internal to the modMax library, so we were forced to restrict our testing to the greedy and the Louvain algorithms.

Next, we set out to explore the parameter space of the greedy algorithm and then, after developing a sufficient number of tests, running all of the tests over all of our graphs. The two main parameters to the greedy algorithm we examined were q and *initial*. The q parameter is used to tell the algorithm which variant of the standard modularity measure to use, if any. If q is set to "general',' then the normal

modularity measure is used. If it is set to "danon," then the value that is maximized is a normalized version of the traditional modularity measure based on the number of edges in a a community, and if q is set to one of "wakita1," "wakita2," or "wakita3," the algorithm maximizes the product of the consolidation ratio with the standard modularity measure [46]. The *initial* parameter is used to tell the algorithm how the initial partition of the graph into communities should be performed. The five options available for this parameter are "general," "prior," "walkers," "adclust," "subgraph," and "own." The "general" version has every vertex start in its own community, whereas the "prior" and "own" options allow the user to direct the algorithm to perform the clustering based on previous knowledge about the structure of the network. The "walkers" option directs the algorithm to place random walkers on the graph and create the initial clustering based on the vertices traversed by these random walkers; the "adclust" option performs fast greedy refinement on the general structure and then applies the refinement again after every merge of communities. Finally, the "subgraph" option directs the algorithm to find an initial community structure based on similarities of subgraphs of the network structure [46]. For purposes of our testing, since we have no prior expectations about the structure of networks in these graphs (in fact, we expect that identifying any good communities at all would be difficult), we did not use the "prior" or "own" options in our testing. Thus, with five options for q and four options for *initial*, we had twenty total tests for the greedy algorithm. We also had a single test for the Louvain variant, which does not take any additional command line arguments.

After narrowing down our testing to these twenty-one tests, we began running

each of the tests on each of the graphs. Thanks to size of the maya cluster, we were able to run tests on each graph in parallel, which greatly reduced the time it took for the tests to be performed. Unfortunately, after queueing up the tests on each graph, we noticed them some tests were starting to fail. Digging into the R error logs, we discovered that, despite running these tests on maya batch nodes, we were running into out-of-memory errors when using the "adclust" option on moderately-sized graphs (around 300 vertices or more) and when using the "subgraph" option on large graphs (800 vertices or more). Since the "adclust" variant adds refinement steps at each merging event, as the sizes of the communities and graphs get larger, it is understandable that this particular option could lead to memory errors; similarly, with sufficiently large graphs, attempting to use subgraph similarity to create initial partitions was simply too computationally intensive. Some of the smaller graphs were able to make it through all twenty-one tests, however. Table 10.3 shows these results for the 100-node randomly generated graph. As this table illustrates, the results for the "wakita1," "wakita2," and "wakita3" tests were identical. The consolidation ratios in the wakita variants are used to control the sizes of the communities being merged, and since we do not wish to restrict the algorithm's ability to find the best communities, regardless of size, we made the decision to remove the "wakita2" and "wakita3" tests [10, 49]. Furthermore, for regular graphs, "wakita1" and "wakita3" will produce identical results, since one uses a heuristic based on the number of vertices in a community, and the other is based on the sum of the degrees of the vertices in a community. Thus, before running our tests over all nineteen graphs again, we removed those tests that used the

"adclust" option, those that used the "subgraph" option, and those that used the "wakita2" or "wakita3" options. This left us with a total of seven tests: a "general" and a "walkers" version of each of the "general," "danon," and "waikita1" variants, and a single Louvain algorithm test. With these changes in place, we were able to start out tests in parallel.

Table 10.3: Community Detection – Greedy Tests for $n = 100$ random graph

ALGORITHM	*q*	*initial*	# COMMS	AVERAGE COMM. SIZE	# VERTICES IN COMM. WITH WORST EXP.	MIN V. EXP.	MIN E. EXP.	MEDIAN V. EXP.	MEDIAN E. EXP.	CHEEGER UPPER BOUND	SPECTRAL LOWER BOUND	SPECTRAL UPPER BOUND	% IMPROV. OVER UPPER BOUND	% ABOVE LOWER BOUND	TIME (SECS)
LOUVAIN	N/A	N/A	2	50	49	0.7959	1.1633	0.7959	1.1633	9.4036	0.6191	3.5188	66.9410	87.9012	111.1219
GREEDY	GENERAL	GENERAL	8	12.5	21	1.8095	1.9412	2.1381	2.2500	9.4036	0.6191	3.5188	44.8335	213.5565	9.7352
GREEDY	GENERAL	WALKERS	8	12.5	23	1.6957	1.8696	2.2898	2.3750	9.4036	0.6191	3.5188	46.8686	201.9892	21.4187
GREEDY	GENERAL	SUBGRAPH	9	11.1111	20	1.8000	2.0000	2.0833	2.3333	9.4036	0.6191	3.5188	43.1618	223.0583	46.7899
GREEDY	GENERAL	ADCLUST	9	11.1111	15	1.7333	1.9333	2.2222	2.3333	9.4036	0.6191	3.5188	45.0564	212.2897	2323.5021
GREEDY	DANON	GENERAL	8	12.5	15	1.8667	2.0667	2.1538	2.3077	9.4036	0.6191	3.5188	41.2672	233.8269	14.3571
GREEDY	DANON	WALKERS	9	11.1111	16	1.7500	2.0000	2.3333	2.5000	9.4036	0.6191	3.5188	43.1618	223.0583	22.6941
GREEDY	DANON	SUBGRAPH	9	11.1111	20	1.8000	2.0000	2.0000	2.3333	9.4036	0.6191	3.5188	43.1618	223.0583	48.5408
GREEDY	DANON	ADCLUST	8	12.5	17	1.5294	1.7059	2.1364	2.2727	9.4036	0.6191	3.5188	51.5203	175.5497	2164.7540
GREEDY	WAKITA1	GENERAL	8	12.5	21	1.8095	1.9412	2.1381	2.2500	9.4036	0.6191	3.5188	44.8335	213.5565	13.7886
GREEDY	WAKITA1	WALKERS	8	12.5	23	1.6957	1.8696	2.2898	2.3750	9.4036	0.6191	3.5188	46.8686	201.9892	23.0443
GREEDY	WAKITA1	SUBGRAPH	9	11.1111	20	1.8000	2.0000	2.0833	2.3333	9.4036	0.6191	3.5188	43.1618	223.0583	49.1804
GREEDY	WAKITA1	ADCLUST	9	11.1111	15	1.7333	1.9333	2.2222	2.3333	9.4036	0.6191	3.5188	45.0564	212.2897	2184.6411
GREEDY	WAKITA2	GENERAL	8	12.5	21	1.8095	1.9412	2.1381	2.2500	9.4036	0.6191	3.5188	44.8335	213.5565	11.3278
GREEDY	WAKITA2	WALKERS	8	12.5	23	1.6957	1.8696	2.2898	2.3750	9.4036	0.6191	3.5188	46.8686	201.9892	23.0482
GREEDY	WAKITA2	SUBGRAPH	9	11.1111	20	1.8000	2.0000	2.0833	2.3333	9.4036	0.6191	3.5188	43.1618	223.0583	49.2500
GREEDY	WAKITA2	ADCLUST	9	11.1111	15	1.7333	1.9333	2.2222	2.3333	9.4036	0.6191	3.5188	45.0564	212.2897	2189.7384
GREEDY	WAKITA3	GENERAL	8	12.5	21	1.8095	1.9412	2.1381	2.2500	9.4036	0.6191	3.5188	44.8335	213.5565	14.1587
GREEDY	WAKITA3	WALKERS	8	12.5	23	1.6957	1.8696	2.2898	2.3750	9.4036	0.6191	3.5188	46.8686	201.9892	31.9514
GREEDY	WAKITA3	SUBGRAPH	9	11.1111	20	1.8000	2.0000	2.0833	2.3333	9.4036	0.6191	3.5188	43.1618	223.0583	48.4612
GREEDY	WAKITA3	ADCLUST	9	11.1111	15	1.7333	1.9333	2.2222	2.3333	9.4036	0.6191	3.5188	45.0564	212.2897	2146.2282

10.2 Data

The results of running these seven tests on each of our nineteen graphs can be found in Tables A.9 and A.10 in Appendix A, both of which occupy multiple pages. We have also created Table 10.4, which illustrates the overall performance of each of the seven algorithms on the nineteen graphs.

10.3 Results

We will start our discussion of the data in Tables A.9 and A.10 by examining how the community detection algorithms performed on the randomly generated graphs. Generally speaking, all of the algorithmic variants provided some improvement over the spectral upper bound on edge expansion for each of the eleven random graphs. That is, the community with the worst expansion in each of the algorithms' partitionings consistently had a lower expansion rate than the theoretical upper bound on the expansion rate. The greedy trials with $q =$"general" and *initial*="walkers" were frequently the best, and their performance was often mirrored by the "wakita1" with "walkers" trials. This shows that the optimal communities to join based on a modularity delta were proportional to the the size of the communities involved in the join, which is not surprising for regular graphs. The dominance of the variants that chose an initial partitioning of the vertices using random walkers is also unsurprising. Since these graphs do not have a strong underlying community structure, when the greedy algorithm starts with each vertex in its own community, the modularity improvements that result from merging such small communities are all quite small, and the algorithm is simply unable to escape local optima and reach a more globally optimal partitioning. However, by using random walkers, the initial partitioning generally consists of larger communities of vertices that are within a short distance of one another, which provides a much better starting point for the algorithm and prevents it from being bogged down trying to merge tiny communities. We also noticed some very interesting results with the Louvain algorithm; as

the size of the graph got bigger, the algorithm tended to lump all of the vertices together in a single community. The Louvain method is known for having resolution limit issues that cause it to fail to detect small communities, which is likely what happened in our testing [15].

Generally speaking, as the sizes of the graphs got larger, the percentage improvement (the difference between the theoretical bound and the empirical bound, divided by the theoretical bound) of the empirical edge expansion upper bound that we observed using community detection decreased slightly, but remained close to 40%. In fact, the average percentage improvement of the edge expansion upper bound for the randomly generated graphs was 41.66%, meaning that the new upper bounds were a little less than 60% of the old bounds. These improvements are quite large, and illustrate the fact that the theoretical upper bounds on edge expansion are extremely loose. That said, the empirical observations were still an average of 255.47% above the theoretical lower bound. This means that the observed expansion rate is over twice as large as the theoretical lower bound. Thus, while we have made a significant improvement over the previous upper bound, the gap between the lower bound and the new upper bound is still quite large.

The results for the explicitly generated GG graphs are similar to those of the randomly generated graphs, with a few notable differences. Once again, we notice that the variants that use the random walker initializations vastly outperform their counterparts. In the GG graphs, this difference is even more pronounced, likely attributable to the fact that having each vertex start in its own community in a bipartite graph is a significantly worse starting position than using a similar strat-

egy in the randomly generated case. In fact, for some of the large GG graphs, the communities with the worst expansion found by some of the algorithms were not able to beat the spectral upper bound, which provides a clear indication of just how poorly the general initialization works for these graphs. The Louvain algorithm had slightly more success with the GG graphs than with the random graphs, but even in the cases where it found non-trivial partitionings, the communities found by the other algorithms were superior. It is also worth noting that the variants of the algorithm that maximized the "danon" version of modularity performed slightly better on some of the smaller GG graphs than what was observed with the randomly generated graphs. Since the "danon" modularity is normalized based on the number of edges internal to communities, those variants are closer to the "general" modularity variants for the GG graphs since the bipartite nature of the graph means that communities will typically have fewer internal edges than in a randomly generated graph.

Unlike the randomly generated graphs, as the GG graphs got larger, there was not a clear decline in the percentage improvement provided by the edge expansion rates of the communities over the theoretical upper bounds on edge expansion. For the GG graphs, the average percentage improvement was 52.98%, which is significantly higher than the improvements we saw with the random graphs. However, the empirical observations of the GG graphs were an average of 558.72% above the theoretical lower bound, which is much, much higher than the 255.47% of the random graphs. A further inspection of the theoretical bounds shows that the spectral lower bounds for the GG graphs are much, much smaller than their random graph counter-

parts. The average second largest eigenvalue of the GG graphs is 4.77, whereas the random graphs have an average second largest eigenvalue of 3.91, which means that the spectral gap of the GG graphs is much smaller than that of the random graphs. Since the upper and lower bounds on edge expansion are based solely on d (which was 5 for both the GG and random graphs) and the second largest eigenvalue, this difference in the spectral gap explains the large difference in the theoretical bounds. Thus, even though, on a percentage basis, larger improvements were made with the upper bounds on edge expansion for the GG graphs than the random graphs, both the theoretical upper and lower bounds for the GG graphs are inherently less tight, which also explains the gap between our observed expansion rates and the theoretical lower bounds.

In terms of comparing the greedy algorithmic variants themselves, Table 10.4 contains information about average improvements and average modularity values for each of the algorithms. As the table shows, the greedy variants that used random walker community initialization far outperformed their general counterparts, especially for the GG graphs. The greedy algorithm that maximized the standard modularity value and used random walker initialization also was the best algorithm (of the ones tested) on the most number of graphs, both for the random graphs and for the GG graphs. In examining the modularity values themselves, we see that they were fairly consistent across all the graphs. The average modularity value seen for the random graphs was 0.4369, and the average modularity for the GG graphs was 0.4943. The higher modularities for the GG graphs is expected, since, by virtue of being explicitly constructed, they are structured less like random graphs than

the randomly generated networks. It is also interesting to note that the algorithms whose community structure had the highest (or close to the highest) average modularity were also the algorithms that saw the most improvement. For example, the "greedy: general, walkers" and the "greedy: wakita1, walkers" algorithms had the highest average modularity for the GG graphs amongst all of the algorithms, and those variants were also the best, in terms of improving the upper bound on edge expansion, for eight out of the nine GG graphs. Thus, it is clear that those algorithms that are the best at maximizing modularity are also the best at detecting communities with poor expansion, which justifies our use of this method for improving upper bounds on edge expansion.

Table 10.4: Community Detection – Algorithm Comparison

Algorithm	Avg. % Improv. Rand.	Avg. % Improv. GG	Avg. % Improv.	Avg. Mod Rand.	Avg. Mod GG	Avg. Mod	Best Rand.	Best GG	Best
Louvain	66.941	24.974	33.368	0.280	0.413	0.386	1	0	1
greedy: general, general	37.981	6.499	23.068	0.442	0.462	0.452	2	0	2
greedy: general, walkers	39.446	52.101	45.440	0.439	0.542	0.488	7	8	15
greedy: danon, general	34.051	0.060	17.950	0.437	0.459	0.447	0	0	0
greedy: danon, walkers	33.895	46.331	39.786	0.438	0.534	0.484	0	4	4
greedy: wakita1, general	37.981	6.499	23.068	0.442	0.462	0.452	2	0	2
greedy: wakita1, walkers	39.446	52.101	45.440	0.439	0.542	0.488	7	8	15

Now, with the capability to use modularity maximization algorithms to assess graph expansion, we will revisit some of the expander graph applications discussed earlier. We will provide some empirical evidence for how expanders can be used to recycle random bits, as well as provide an outline for how expanders can be used to implement cuckoo-like hashing.

Chapter 11

Expander Graph Application Testing and Results

As described in Chapter 3, two applications of expander graphs that we have studied are probabilistic amplification and implementing cuckoo-like hashing algorithms. In this chapter, we will outline our testing methodology for assessing probabilistic amplification, as well as provide and analyze the data from our tests. We will also describe one possible implementation of cuckoo-like hashing using expander graphs.

11.1 Probabilistic Amplification

11.1.1 Testing Methodology

The first step in testing the amount of probabilistic amplification we could achieve with random walks was generating the graphs on which we would be performing the walks. As outlined in Chapter 2, since showing that random walks on randomly generated graphs could amplify probability would not be an interesting result, we used explicit Gabber-Galil expanders for experimentation. We tested a total of twenty-one graphs, one for each odd integer from 7 to 49. These integers represent the m value in the Gabber-Galil construction, where $|V| = 2m^2$. Thus, the sizes of the graphs we tested ranged from 98 vertices to $4,802$ vertices. Since the neighbors of a given vertex in a Gabber-Galil graph can be computed in constant

time, we did not actually have to store an adjacency matrix, or even adjacency lists for the graph. Rather, for a given value of m, we could simply walk through the graph, computing neighbors as we progressed from one vertex to the next.

After identifying and creating the graphs we wanted to test, we determined that we would compare random walks to the traditional use case of random bits, which is using $\log |V|$ bits to select a graph vertex uniformly at random, and then keeping track of how frequently each vertex in the graph is selected. For purposes of our testing, we also looked at the neighboring vertices of the nodes selected in this manner and tracked their counts as well. In order for the comparison of random samples to random walking to be fair, the same amount of random bits must be used in each case; otherwise, one method would have a clear advantage over the other. Thankfully, we were able to accomplish this in a rather straightforward manner. First, note that, in order to select t vertices from a graph uniformly at random using vertex sampling, it would take $t \cdot \log |V|$ random bits. Second, for the case of random walking, we see that it takes $\log d$ steps at each vertex to randomly determine the next vertex to visit in a d-regular graph. Thus, if we take $\log_d |V|$ steps, we use

$$\log_d |V| \log d = \frac{\log |V|}{\log d} \log d = \log |V|$$

bits of randomness. It also takes $\log |V|$ bits to choose the starting point for the random walk. Thus, since it takes $t \cdot \log |V|$ random bits to sample t vertices, we can take $(t-1) \log_d |V|$ steps in a random walk in order to use the same number of random bits. Since we wanted to compare the performance of the sampling method to the

performance of the random walk method across a variety of percentages of random bit usage, we performed eighteen different pairs of tests on each of the twenty-one graphs, each test sampling a different percentage of the vertices. Each pair consisted of a sampling test that sampled $t = k|V|$ vertices uniformly at random, and a random walk that took $(t-1)\log_d |V|$ steps. The eighteen different values of k we chose were $5\% - 15\%$, inclusive, with 1% increments, along with $20\%, 25\%, 30\%, 40\%, 50\%, 60\%$, and 70%.

After selecting the percentages of vertices to sample, we began running the tests. For each value of t for each of the twenty-one graphs, we ran $1,000$ trials of sampling t vertices and $1,000$ trials of taking a random walk of length $(t-1)\log_d |V|$. As previously mentioned, for the sampling, we selected t vertices uniformly at random, and then also visited all of the neighbors of that vertex. Similarly, during the random walk, we would visit each vertex along the walk, as well as all the neighbors of each selected vertex. Since the graph is explicitly constructed, we can do this without fear of gaining an "unfair" advantage over the sampling methods, since there is no randomness inherent in the graph's structure, combining the idea of neighbor sampling with random walks [31].

Thus, in the case of taking t random samples on a d-regular graph, the maximum number of unique vertices that could be seen is $\max\{t(d+1), |V|\}$. In the case of the a random walk with $(t-1)\log_d |V|$ steps, the math is somewhat more complicated. For each step, a maximum of $(d-1)$ new vertices can be seen, since one of the d vertices (namely, the vertex the walk is currently at), was a neighbor of the previous vertex on the walk, and thus should not be double-counted. However,

the vertex chosen at the start point and its d neighbors must also be accounted for, so the maximum number of unique vertices that could be seen by a random walk of $(t-1)\log_d |V|$ steps is $\max\{1+d+(d-1)(t-1)\log_d |V|, |V|\}$. Thus, theoretically speaking, the random walk method has the potential to find more unique vertices.

Now, for each set of $1,000$ trials, we computed the median and average number of unique vertices that were explored by both the sampling and random walk tests. We then computed these as percentages of the number of vertices, and then computed the difference between the two percentages to gauge the relative performance of the two algorithms.

11.1.2 Data

The complete results from the testing can be found in Table A.11 in Appendix A. A sample of the output, representing the tests for just one graph, can be found below in Table 11.1. A second chart, Table 11.2, contains results for all of the $k = 15\%$ tests for each of the twenty-one graphs, providing a horizontal cross-section of the results.

11.1.3 Results

The results of our probability amplification tests were extremely interesting. As the rightmost column indicates, for all twenty-one graphs, every one of the eighteen tests resulted in the random walk method visiting more unique vertices. The difference percentages also followed a similar pattern for each of the graphs; typically,

Table 11.1: Probability Amplification – Test Results; $m = 17$

m	$\|V\|$	TRIALS	t	k (%)	# STEPS	WALK AVG	WALK MEDIAN	SAMPLE AVG	SAMPLE MEDIAN	% WALK	% SAMPLE	% DIFF
17	578	1000	29	5	110	220.57	223	145.61	146	0.382	0.252	0.130
17	578	1000	35	6	134	255.48	258	170.79	171	0.442	0.295	0.147
17	578	1000	41	7	158	285.42	290	194.58	195	0.494	0.337	0.157
17	578	1000	47	8	181	310.03	314	217.16	217	0.536	0.376	0.161
17	578	1000	53	9	205	334.31	337	237.69	238	0.578	0.411	0.167
17	578	1000	58	10	225	355.14	360	254.50	255	0.614	0.440	0.174
17	578	1000	64	11	248	375.28	379	273.17	273	0.649	0.473	0.177
17	578	1000	70	12	272	394.05	398	291.50	292	0.682	0.504	0.177
17	578	1000	76	13	296	410.03	413	307.40	307	0.709	0.532	0.178
17	578	1000	81	14	316	422.14	425	320.98	321	0.730	0.555	0.175
17	578	1000	87	15	339	436.45	441	335.56	336	0.755	0.581	0.175
17	578	1000	116	20	454	487.03	490	396.76	397	0.843	0.686	0.156
17	578	1000	145	25	569	518.81	523	441.70	442	0.898	0.764	0.133
17	578	1000	174	30	683	538.13	540	476.66	477	0.931	0.825	0.106
17	578	1000	232	40	912	559.78	562	520.27	520	0.968	0.900	0.068
17	578	1000	289	50	1138	568.99	571	545.08	545	0.984	0.943	0.041
17	578	1000	347	60	1367	572.81	575	559.32	560	0.991	0.968	0.023
17	578	1000	405	70	1596	575.13	577	567.48	568	0.995	0.982	0.013

Table 11.2: Probability Amplification – Test Results; $k = 15\%$

m	$\|V\|$	TRIALS	t	k (%)	# STEPS	WALK AVG	WALK MEDIAN	SAMPLE AVG	SAMPLE MEDIAN	% WALK	% SAMPLE	% DIFF
7	98	1000	15	15	39	60.80	62	56.01	56	0.620	0.572	0.049
9	162	1000	25	15	75	108.66	110	94.17	94	0.671	0.581	0.089
11	242	1000	37	15	122	168.68	171	140.32	140	0.697	0.580	0.117
13	338	1000	51	15	180	242.78	245	195.68	196	0.718	0.579	0.139
15	450	1000	68	15	254	334.22	336	261.33	261	0.743	0.581	0.162
17	578	1000	87	15	339	436.45	441	335.56	336	0.755	0.581	0.175
19	722	1000	109	15	441	556.86	561	420.94	421	0.771	0.583	0.188
21	882	1000	133	15	556	690.90	694	514.22	514	0.783	0.583	0.200
23	1058	1000	159	15	683	838.48	843	617.56	618	0.793	0.584	0.209
25	1250	1000	188	15	828	1002.60	1008	730.31	731	0.802	0.584	0.218
27	1458	1000	219	15	986	1178.85	1183	853.03	853	0.809	0.585	0.223
29	1682	1000	253	15	1163	1377.56	1383	984.90	985	0.819	0.586	0.233
31	1922	1000	289	15	1353	1585.52	1591.5	1126.22	1126	0.825	0.586	0.239
33	2178	1000	327	15	1556	1809.53	1815.5	1275.97	1275.5	0.831	0.586	0.245
35	2450	1000	368	15	1779	2051.87	2056	1438.44	1439	0.837	0.587	0.250
37	2738	1000	411	15	2016	2306.50	2310	1606.28	1606.5	0.842	0.587	0.256
39	3042	1000	457	15	2272	2577.09	2582	1786.39	1787	0.847	0.587	0.260
41	3362	1000	505	15	2542	2865.92	2873	1976.14	1977	0.852	0.588	0.265
43	3698	1000	555	15	2827	3165.81	3172	2174.52	2175	0.856	0.588	0.268
45	4050	1000	608	15	3132	3483.83	3491	2381.75	2381	0.860	0.588	0.272
47	4418	1000	663	15	3452	3814.15	3819	2597.98	2599	0.863	0.588	0.275
49	4802	1000	721	15	3792	4168.62	4174	2826.91	2828	0.868	0.589	0.279

for low values of $k\%$, the random walk method would be about 10-15 percentage points higher than the sampling method in terms of the proportion of unique vertices visited. As the value of k increased, so would the differences in the percentages, up to

the point where k was about $15\% - 20\%$. At those sampling percentages, the random walk method was significantly more effective than the sampling method, reaching about 30 percentage points of difference. Generally speaking, as the graphs got larger, this performance gap also tended to get bigger. Thus was somewhat unexpected, since it is clear that the sampling method would have much fewer collisions when the graphs are larger; however, due to the nice expansion properties of the GG graphs, there are also fewer collisions with the random walks on larger graphs, which more than made up for the difference.

For values of k that are around 40% and higher, the differences between the two methods began to shrink. Since all of the GG graphs are 5-regular, and since we are examining the neighbors of the sampled points and the neighbors of the points on the walk, by the time k got into the 40%+ range, our data shows that over 95% of all the vertices in the graph were being visited by both algorithms. Thus, even though the random walk method significantly out performs random sampling when the percentage of vertices sampled is fairly small, as that percentage grows, the returns diminish greatly.

Thus, these results show that it is indeed possible, and in fact, very wise, to use random walks on expander graphs to amplify probability. Using the same number of random bits, it is possible to reach significantly more vertices from the graph using this method. For instance, if one were to use enough random bits to select 7% of the vertices as samples, using the random walk method would allow for over half of the graph's vertices (for graphs of moderate to large size) to be visited, as opposed to only about a third of the vertices if using random sampling alone.

11.2 Cuckoo Hashing and Expander Graphs

As described by Mitzenmacher, cuckoo hashing with two hash functions is similar to a random graph where the edges represent the items stored in the hash [35]. However, cuckoo hashing on regular expander graphs can also be viewed in another way. Instead of considering the edges to be items, we simply consider the vertices of a graph to be positions in a hash table, and we have a single hash function whose image is the entire set of vertices. The valid locations for an item to be stored in the hash are the vertex to where it is hashed, in addition to the d neighbors of that vertex. Thus, insertion into the hash would work as follows: first, a hash function is applied to the key being stored, and that hash produces an index that corresponds to a vertex in the graph. If no other object already occupies that node, then the item is simply stored there. If there is another object, then the neighbors of that vertex are checked in some canonical ordering. If any one of the neighboring vertices is empty, the item is stored at that vertex. If all of the neighboring vertices are already occupied by an item, then one of the neighbors is selected at random, the item being inserted is placed at that neighbor's position, and the item that was previously stored at that neighboring vertex is then hashed, and the process continues. Thus, similar to cuckoo hashing, keys being inserted can push out previously placed keys, which is then attempted to be placed back in the hash itself. Similar to traditional cuckoo hashing, searching for a key can be done in $\mathcal{O}(d)$ time by simply hashing the key and then checking the resulting vertex and its d neighbors. In this scheme, similar to the traditional cuckoo hashing method, if

there are $d+2$ keys that all map to the same vertex, then a new hash function will have to be used and all the keys will need to be rehashed.

In terms of implementing the algorithm itself, the information stored at each vertex would consist of the following: one bit to indicate whether an item is currently stored at that vertex, d bits to indicate whether any item that has hashed to the vertex has migrated to a neighboring vertex, one bit to indicate whether or not the item stored at this vertex hashed to this vertex, and d bits to indicate which neighboring vertex (if any) the item in the current vertex came from (hashed to originally). Then, when searching for a key, the d bits indicating whether anyone has migrated can be shifted d times, and if a set bit shifts into the carry, then that neighbor will have to be checked for the item in question (assuming it's not stored at the first vertex checked). The bit indicating whether or not the item stored at a vertex actually hashed to that vertex, in addition to the d bits for migrations, can also be used to quickly determine whether or not a new hash functions needs to be selected; if that bit is set for a given vertex and the d bits for the outgoing edges are all set, then it is clear that there are more than $d+1$ keys that all hashed to the same vertex.

Now that we have described how our cuckoo-like hashing algorithm can be implemented using expander graphs, we can discuss the merits of using random expanders versus explicitly constructed expanders.

One of the main benefits to using explicitly constructed expanders, such as the Gabber-Galil expanders, is that the graph itself does not need to be stored in memory [20]. This is because the neighbor relations of any vertex in the graph can

be computed on the fly in $\mathcal{O}(d)$ without having any additional information stored. For the GG construction, and many other explicit expander constructions, the edge relationships often involve computations that are performed modulo the number of vertices. Knowing this, we can speed up the modular arithmetic process by simply choosing the number of vertices in the graph to be a power of two. In that case, calculating the mod of a number could be performed simply by bit-shifting the number the proper number of digits.

Conversely, using a randomly generated expander would yield better performance under similar loads as compared to explicit expanders, but that increased performance comes at the cost of having to generate the entire graph ahead of time and then storing that graph so that the neighbors of each vertex can be retrieved when needed. However, there is a also a middle option, which is where the underlying expander graph is pseudo-random. In that case, the hashing algorithm could use pre-defined seeds for each vertex that would be used to prime a pseudo-random number generator to compute the neighbors of a given vertex. By using these seeds on a per-vertex basis, it would be possible to guarantee that the algorithm would compute the same neighbors for a vertex every time, which would mean that the entire graph itself would not have to be stored. Instead, of course, the seeds themselves would need to be stored, and, in order for this method to be a replacement for storing the entire graph, the pseudo-randomness properties of the algorithm that uses the seeds would have to be thoroughly verified. This a clear case of a time versus space trade-off, where, as more information about the graph is stored, fewer computations would be required to compute the neighboring vertices. Some form of

pseudo-random graphs with seeds would likely yield better performance than using this cuckoo-like hashing on explicitly constructed expanders, simply due to better expansion rates.

To get an idea of how this hashing scheme might work in practice, we can simply consider a hash table created using this scheme that has a load factor less than 1/2. In that case, we see that the set of vertices that currently store an item would have size less than $\frac{1}{2}n$, meaning that the community would have a vertex expansion rate at least as big as the lower bound on vertex expansion. Throughout our testing of randomly generated graphs, the lowest vertex expansion rate we came across was 0.79 for a 100-vertex graph. Thus, on average, close to 80% of the vertices in our hash that have items stored in them will have an edge that connects to a neighbor that does not have any item stored at it. Thus, if a new item is being stored in the table, there is a very high likelihood that there will be a place for it, either at the position to which it hashed or at one of the neighbors of that position. If all of the positions are filled, then, as previously hashed items are pushed out, it is clear that, with high probability, an available space would be found within three of four iterations.

As this analysis shows, d-regular expander graphs can be used to implement d-choice cuckoo hashing using only a single hash function. If the load factor of the hash table is less than $\frac{1}{2}$, bounds on the vertex expansion of graph subsets can guarantee that we would not expect many collisions. That said, much like traditional cuckoo hashing, if more than $d+1$ items hash to the same vertex, a new hash function will have to be used, and all of the existing keys will need to be rehashed.

Chapter 12

Conclusions

Expander graphs are extremely versatile objects with many applications, both practical and theoretical, across mathematics and computer science. We have described and discussed two such applications, probability amplification and hashing. We performed empirical tests to measure probabilistic amplification, the results of which showed that using random walks on explicitly constructed expanders can help preserve a substantial number of random bits as compared to simply drawing samples uniformly at random from a population. We also outlined how expanders can be used to implement cuckoo-like hashing algorithms, where the vertices of a random graph represent possible locations where items can be stored in a hash table.

Unfortunately, we have also shown that, given a random regular graph, it is very difficult to get a good idea of that graph's expansion rate. We've shown theoretical results proving that calculating exact expansion rates is NP-hard, as well as theoretical results that prove that it is NP-hard to provide a good estimation of a graph's expansion rate. We have also discussed the deep ties that the difficulty of estimating a graph's expansion rate has to the Unique Games Conjecture and other problems related to the hardness of approximation.

Despite these computational difficulties, we also know that, with high probability, any random d-regular graph is an expander. Thus, in order to assure that

a given graph is a good expander, it becomes necessary to develop methods to try and assess expansion rates. Methods like the linear programming approximation of `SparsestCut` can provide some information about a graph's expansion rate, but due to the computational infeasibility of running the linear programming relaxation on large graphs and the constants hidden in front of the approximations, it is not a practical solution for the general case. The spectral gap of a graph is also correlated to its expansion rate, but that relationship is inconsistent, and, as we have shown, the bounds on expansion that the spectral gap provides are very loose.

To address this shortcoming in evaluating graph expansion, we have developed multiple methods for empirically evaluating a graph's expansion. One method was to randomly sample subsets of a fixed size and compute the average observed expansion rate. These averages were closely correlated with the second eigenvalue of the corresponding graphs' adjacency matrices, which illustrates that they are a good measure for comparing the expansion rates of graphs. Other methods for evaluating expansion used community detection algorithms that maximize a graph's modularity in order to find well-defined communities of vertices. These communities are subsets of vertices that have very few connections to vertices not in the community, and thus, correspond to clusters of vertices with bad expansion.

Based on the data from our community detection experiments, we have clearly shown that the spectral upper bound on edge expansion is an extremely loose bound for randomly generated d-regular graphs, and for graphs that were explicitly constructed using the Gabber-Galil method. Similarly, we believe that the lower spectral bound is also quite loose, especially in the case of the GG graphs and other

graph families that have a relatively small spectral gap. Thus, we believe the actual expansion rates of randomly generated graphs to be much closer to the minimum expansion found via community detection testing than to the spectral lower bound. Our data clearly illustrates that, given an expander graph, using community detection algorithms can help significantly narrow the range of the graph's true expansion rate and provide some evaluation of a graph's true expansion rate.

Chapter 13

Future Work

There are a number of different directions for future work in the areas of expander graphs and approximating expansion rates. First, performing more modularity maximization testing with larger graphs would help make the relationship between expansion rates and modularity more clear. It would also be interesting to test large non-random expanders that were created using a different explicit construction than that of Gabber and Galil. There are also additional modularity maximization algorithms that could be tested, as well as methods that we did test whose parameter spaces could be more fully explored in testing. More generally, there are also other community detection algorithms besides modularity maximization algorithms that could be tested. For example, some sort cosine similarity algorithm that uses the network's adjacency matrix could potentially be used to find communities with low expansion. In fact, some of these alternative measures might be more closely aligned with finding communities with low expansion, as opposed to finding communities with significantly more internal connections that external connections, which is the primary goal of modularity maximization.

Repeating the vertex expansion sampling outlined in Section 9.2 using larger graphs, more trials, and testing subsets of different sizes would help create a clearer connection between the second eigenvalue and the expected vertex expansion of an

arbitrarily chosen subset. Our results established a link between the two, but until a larger number of sizes of subsets are tested, it is not clear how reliably the two values are correlated. With more empirical observations, statistical methods could be used to quantify the strength of this relationship.

Implementing the cuckoo-like hashing algorithms described in Section 11.2 and then performing empirical tests to see how its performance compares to existing hashing implementations would be very interesting. This would also enable the exploration of the space versus time tradeoff that was discussed regarding using random regular graphs as the underlying structure of the hash.

Given enough compute power and time, it would also be worthwhile to re-examine the zig-zag product and potentially create expanders using the zig-zag construction (and other explicit construction methods). Once the graphs have been constructed, the same modularity maximization and community detection algorithms could be used to empirically measure their expansion rates. Additional computing resources would also make it possible to revisit the linear programming implementation of the `SparsestCut` approximation algorithm and run that program over dozens of both randomly generated and explicitly constructed expander graphs.

Appendix A

Tables and Charts

Table A.1: Neighborhood Sizes – Theoretical Expectation

d	n	ϵ	Expected Neighb. Size
4	100	0.25	72.409
6	100	0.25	83.265
8	100	0.25	89.850
4	200	0.125	93.857
6	200	0.125	117.336
8	200	0.125	135.621

Table A.2: Graph Generation Results

d	n	#Graphs	Min	Max	Average	Median	$2\sqrt{d-1}$
4	100	5000	3.28804	4.00000	3.45528	3.44783	3.46410
6	100	5000	4.19232	5.02061	4.44508	4.43706	4.47214
8	100	5000	4.94387	6.37714	5.24823	5.24280	5.29150
4	200	5000	3.36599	4.00000	3.45932	3.45442	3.46410
6	200	5000	4.29392	4.99664	4.45581	4.45105	4.47214
8	200	5000	5.08165	6.37413	5.26645	5.26107	5.29150

Table A.2: Graph Generation Results (Cont'd)

d	n	Min (norm)	Max (norm)	$\frac{2\sqrt{d-1}}{d}$	Ramanujan Diff. (norm)
4	100	0.82201	1.00000	0.86603	0.04402
6	100	0.69872	0.83677	0.74536	0.04664
8	100	0.61798	0.79714	0.66144	0.04345
4	200	0.84150	1.00000	0.86603	0.02453
6	200	0.71565	0.83277	0.74536	0.02970
8	200	0.63521	0.79677	0.66144	0.02623

Table A.3: Singleton, $|S| = 1$ searching; $n = 100$

D	N	# TRIALS	# RED	$\|S\|$	$\lambda(G)$	0 RED	1 RED	2 RED	3 RED	4 RED	5 RED	6 RED
4	100	500	20	1	3.288	32.38000	41.74600	20.68800	4.65800	0.51600	0.01200	
4	100	500	20	1	4.000	32.76800	41.57600	20.60800	4.57600	0.44000	0.03200	
4	100	500	20	1	3.475	32.68200	42.07800	20.29800	4.45800	0.46800	0.01600	
4	100	500	20	1	3.420	32.64000	41.88800	20.47800	4.48400	0.49600	0.01400	
4	100	500	20	1	3.373	32.28800	41.80400	20.80600	4.61800	0.47000	0.01400	
6	100	500	20	1	4.192	20.99400	37.27000	28.12800	10.92400	2.37400	0.29200	0.01800
6	100	500	20	1	5.021	21.11000	37.98800	27.75400	10.54400	2.32800	0.25600	0.02000
6	100	500	20	1	4.486	21.51400	38.05200	27.45000	10.42200	2.27800	0.27000	0.01400
6	100	500	20	1	4.290	20.32400	38.10800	28.48400	10.50200	2.29200	0.27400	0.01600
6	100	500	20	1	4.496	20.41200	37.68200	28.50200	10.78200	2.34400	0.26200	0.01600
8	100	500	20	1	4.944	13.19000	31.19400	31.28800	17.15400	5.75400	1.21800	0.19800
8	100	500	20	1	6.377	13.69600	31.50200	31.39000	16.66600	5.47000	1.12400	0.14600
8	100	500	20	1	5.373	13.83000	32.20000	30.96200	16.63800	5.18600	1.03600	0.14600
8	100	500	20	1	5.289	13.70600	31.76200	30.75800	16.96400	5.53200	1.10600	0.15400
8	100	500	20	1	5.297	13.37200	31.53800	31.34800	16.71400	5.70800	1.14800	0.14600

Table A.3: Singleton, $|S| = 1$ searching; $n = 100$
(continued from previous page)

D	N	# TRIALS	# RED	$\|S\|$	$\lambda(G)$	7 RED	8 RED	9 RED	E(REDS)
4	100	500	20	1	3.288				0.99220
4	100	500	20	1	4.000				0.98440
4	100	500	20	1	3.475				0.98000
4	100	500	20	1	3.420				0.98350
4	100	500	20	1	3.373				0.99220
6	100	500	20	1	4.192	0.00000			1.37362
6	100	500	20	1	5.021	0.00000			1.35840
6	100	500	20	1	4.486	0.00000			1.34764
6	100	500	20	1	4.290	0.00000			1.37216
6	100	500	20	1	4.496	0.00000			1.37814
8	100	500	20	1	4.944	0.00400	0.00000	0.00000	1.75554
8	100	500	20	1	6.377	0.00600	0.00000	0.00000	1.72698
8	100	500	20	1	5.373	0.00200	0.00000	0.00000	1.70852
8	100	500	20	1	5.289	0.01800	0.00000	0.00000	1.72878
8	100	500	20	1	5.297	0.02600	0.00000	0.00000	1.74006

Table A.4: Singleton, $|S| = 1$ searching; $n = 200$

D	N	# TRIALS	# RED	$\|S\|$	$\lambda(G)$	0 RED	1 RED	2 RED	3 RED	4 RED	5 RED	6 RED
4	200	500	40	1	3.288	32.81500	40.91500	20.73700	4.93700	0.57600	0.02000	
4	200	500	40	1	4.000	33.36100	41.14500	20.07800	4.79700	0.59400	0.02500	
4	200	500	40	1	3.475	32.85100	41.44300	20.32000	4.81200	0.54800	0.02600	
4	200	500	40	1	3.420	32.73200	41.48900	20.43700	4.77700	0.53500	0.03000	
4	200	500	40	1	3.373	32.57900	41.64300	20.39100	4.82800	0.52600	0.03300	
6	200	500	40	1	4.192	20.74000	37.11900	27.80700	11.28100	2.66800	0.35400	0.03000
6	200	500	40	1	5.021	21.30100	37.29800	27.33000	11.03800	2.65800	0.34700	0.02800
6	200	500	40	1	4.486	20.98700	36.96100	27.83600	11.24900	2.61900	0.32700	0.02000
6	200	500	40	1	4.290	21.27500	37.31200	27.56500	10.92500	2.55000	0.35000	0.02200
6	200	500	40	1	4.496	20.80500	37.43000	27.66300	11.19500	2.52100	0.35700	0.02800
8	200	500	40	1	4.944	13.18900	30.65300	30.82200	17.47000	6.20500	1.41700	0.22200
8	200	500	40	1	6.377	13.19100	30.94800	31.09600	17.23700	6.00500	1.30900	0.19900
8	200	500	40	1	5.373	13.60000	30.75900	30.95300	17.02700	6.05300	1.38800	0.20100
8	200	500	40	1	5.289	13.42800	30.90900	30.68400	17.24700	6.08500	1.42700	0.19700
8	200	500	40	1	5.297	13.80700	30.81100	30.55400	17.21700	6.02200	1.36800	0.20400

Table A.4: Singleton, $|S| = 1$ searching; $n = 200$
(continued from previous page)

D	N	# TRIALS	# RED	$\|S\|$	$\lambda(G)$	7 RED	8 RED	9 RED	E(REDS)
4	200	500	40	1	3.288				0.99604
4	200	500	40	1	4.000				0.98193
4	200	500	40	1	3.475				0.98841
4	200	500	40	1	3.420				0.98984
4	200	500	40	1	3.373				0.99178
6	200	500	40	1	4.192	0.00100			1.39205
6	200	500	40	1	5.021	0.00000			1.37607
6	200	500	40	1	4.486	0.00100			1.38618
6	200	500	40	1	4.290	0.00100			1.37306
6	200	500	40	1	4.496	0.00100			1.38385
8	200	500	40	1	4.944	0.02200	0.00000	0.00000	1.78098
8	200	500	40	1	6.377	0.01500	0.00000	0.00000	1.76715
8	200	500	40	1	5.373	0.01900	0.00000	0.00000	1.76237
8	200	500	40	1	5.289	0.02000	0.00300	0.00000	1.76839
8	200	500	40	1	5.297	0.01600	0.00100	0.00000	1.75842

Table A.5: Subset sampling, $|S| = 25$; $n = 100$

D	N	# TRIALS	# RED	# SUBSETS	$\lvert S\rvert$	$\lambda(G)$	0 RED	1 RED	2 RED	3 RED	4 RED	5 RED
4	100	500	20	500	25	3.288	0.000	0.000	0.000	0.000	0.000	0.000
4	100	500	20	500	25	4.000	0.000	0.000	0.000	0.000	0.000	0.000
4	100	500	20	500	25	3.475	0.000	0.000	0.000	0.000	0.000	0.000
4	100	500	20	500	25	3.420	0.000	0.000	0.000	0.000	0.000	0.000
4	100	500	20	500	25	3.373	0.000	0.000	0.000	0.000	0.000	0.000
6	100	500	20	500	25	4.192	0.000	0.000	0.000	0.000	0.000	0.000
6	100	500	20	500	25	5.021	0.000	0.000	0.000	0.000	0.000	0.000
6	100	500	20	500	25	4.486	0.000	0.000	0.000	0.000	0.000	0.000
6	100	500	20	500	25	4.290	0.000	0.000	0.000	0.000	0.000	0.000
6	100	500	20	500	25	4.496	0.000	0.000	0.000	0.000	0.000	0.000
8	100	500	20	500	25	4.944	0.000	0.000	0.000	0.000	0.000	0.000
8	100	500	20	500	25	6.377	0.000	0.000	0.000	0.000	0.000	0.000
8	100	500	20	500	25	5.373	0.000	0.000	0.000	0.000	0.000	0.000
8	100	500	20	500	25	5.289	0.000	0.000	0.000	0.000	0.000	0.000
8	100	500	20	500	25	5.297	0.000	0.000	0.000	0.000	0.000	0.000

Table A.5: Subset sampling, $|S| = 25$; $n = 100$
(continued from previous page)

D	N	$\lambda(G)$	6 RED	7 RED	8 RED	9 RED	10 RED	11 RED	12 RED	13 RED	14 RED	15 RED
4	100	3.288	0.001	0.004	0.021	0.126	0.462	1.562	4.188	9.034	15.527	20.663
4	100	4.000	0.000	0.005	0.026	0.135	0.556	1.793	4.656	9.717	16.029	20.792
4	100	3.475	0.000	0.002	0.026	0.147	0.509	1.804	4.673	9.770	15.966	20.819
4	100	3.420	0.000	0.003	0.023	0.141	0.523	1.746	4.539	9.493	15.910	20.816
4	100	3.373	0.000	0.003	0.028	0.120	0.465	1.578	4.184	9.153	15.513	20.756
6	100	4.192	0.000	0.000	0.000	0.000	0.004	0.029	0.141	0.678	2.454	6.998
6	100	5.021	0.000	0.000	0.000	0.001	0.007	0.045	0.210	0.879	2.940	7.893
6	100	4.486	0.000	0.000	0.000	0.001	0.004	0.028	0.212	0.862	3.032	7.934
6	100	4.290	0.000	0.000	0.000	0.000	0.003	0.024	0.143	0.678	2.486	6.977
6	100	4.496	0.000	0.000	0.000	0.002	0.003	0.021	0.128	0.692	2.481	7.017
8	100	4.944	0.000	0.000	0.000	0.000	0.000	0.000	0.003	0.014	0.164	0.888
8	100	6.377	0.000	0.000	0.000	0.000	0.000	0.000	0.004	0.034	0.216	1.146
8	100	5.373	0.000	0.000	0.000	0.000	0.000	0.001	0.002	0.040	0.234	1.268
8	100	5.289	0.000	0.000	0.000	0.000	0.000	0.001	0.003	0.032	0.193	1.066
8	100	5.297	0.000	0.000	0.000	0.000	0.000	0.000	0.005	0.029	0.172	0.981

Table A.5: Subset sampling, $|S| = 25$; $n = 100$
(continued from previous page)

d	n	$\lambda(G)$	16 red	17 red	18 red	19 red	20 red	E(reds)	Theor. Reds
4	100	3.288	21.212	15.867	8.306	2.646	0.381	15.347	14.482
4	100	4.000	20.691	15.178	7.639	2.434	0.348	15.250	14.482
4	100	3.475	20.860	15.144	7.572	2.360	0.347	15.247	14.482
4	100	3.420	20.768	15.376	7.771	2.524	0.367	15.276	14.482
4	100	3.373	21.160	15.871	8.145	2.635	0.389	15.339	14.482
6	100	4.192	15.169	24.414	26.714	17.826	5.572	17.389	16.653
6	100	5.021	16.140	24.624	25.600	16.641	5.019	17.283	16.653
6	100	4.486	16.360	24.674	25.557	16.502	4.834	17.270	16.653
6	100	4.290	15.172	24.025	26.745	18.052	5.694	17.398	16.653
6	100	4.496	15.065	23.959	26.774	18.078	5.781	17.402	16.653
8	100	4.944	3.744	12.029	26.203	35.268	21.688	18.557	17.970
8	100	6.377	4.627	13.616	27.482	33.760	19.116	18.446	17.970
8	100	5.373	5.010	14.062	27.447	33.418	18.518	18.414	17.970
8	100	5.289	4.432	13.057	27.022	34.051	20.143	18.483	17.970
8	100	5.297	4.090	12.593	26.554	34.684	20.890	18.519	17.970

Table A.6: Subset sampling, $|S| = 25$; $n = 200$

d	n	# trials	# red	# subsets	$\lvert S\rvert$	$\lambda(G)$	0 red	1 red	2 red	3 red	4 red	5 red	6 red	7 red	8 red
4	200	500	40	500	25	3.366	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.004
4	200	500	40	500	25	4.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.002	0.007
4	200	500	40	500	25	3.419	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.001	0.008
4	200	500	40	500	25	3.436	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.004
4	200	500	40	500	25	3.436	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.002	0.002
6	200	500	40	500	25	4.294	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000
6	200	500	40	500	25	4.997	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000
6	200	500	40	500	25	4.402	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000
6	200	500	40	500	25	4.478	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000
6	200	500	40	500	25	4.392	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000
8	200	500	40	500	25	5.082	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000
8	200	500	40	500	25	6.374	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000
8	200	500	40	500	25	5.283	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000
8	200	500	40	500	25	5.282	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000
8	200	500	40	500	25	5.423	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000

Table A.6: Subset sampling, $|S| = 25$; $n = 200$
(continued from previous page)

D	N	$\lambda(G)$	9 RED	10 RED	11 RED	12 RED	13 RED	14 RED	15 RED	16 RED	17 RED	18 RED	19 RED	20 RED
4	200	3.366	0.013	0.062	0.178	0.500	1.148	2.292	4.128	6.477	9.309	11.658	13.297	13.298
4	200	4.000	0.016	0.082	0.220	0.594	1.323	2.643	4.552	7.159	9.853	12.268	13.382	13.042
4	200	3.419	0.024	0.080	0.220	0.542	1.206	2.540	4.384	6.871	9.592	11.972	13.298	13.295
4	200	3.436	0.021	0.066	0.193	0.493	1.161	2.440	4.293	6.778	9.520	11.832	13.248	13.281
4	200	3.436	0.021	0.072	0.215	0.532	1.165	2.357	4.191	6.673	9.479	11.810	13.243	13.412
6	200	4.294	0.000	0.000	0.000	0.001	0.008	0.027	0.091	0.241	0.568	1.184	2.434	4.316
6	200	4.997	0.000	0.000	0.000	0.002	0.010	0.031	0.085	0.296	0.680	1.440	2.773	4.745
6	200	4.402	0.000	0.000	0.000	0.002	0.008	0.027	0.082	0.251	0.581	1.333	2.555	4.447
6	200	4.478	0.000	0.001	0.001	0.003	0.010	0.041	0.110	0.296	0.678	1.431	2.754	4.753
6	200	4.392	0.000	0.000	0.001	0.004	0.009	0.034	0.096	0.260	0.618	1.356	2.571	4.482
8	200	5.082	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.002	0.008	0.032	0.101	0.237
8	200	6.374	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.004	0.012	0.038	0.113	0.304
8	200	5.283	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.002	0.012	0.032	0.102	0.287
8	200	5.282	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.002	0.008	0.048	0.118	0.272
8	200	5.423	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.004	0.016	0.033	0.102	0.266

Table A.6: Subset sampling, $|S| = 25$; $n = 200$
(continued from previous page)

D	N	$\lambda(G)$	21 RED	22 RED	23 RED	24 RED	25 RED	26 RED	27 RED	28 RED	29 RED	30 RED	31 RED
4	200	3.366	11.935	9.702	6.913	4.440	2.527	1.241	0.558	0.208	0.078	0.025	0.006
4	200	4.000	11.550	9.054	6.367	3.904	2.159	1.090	0.458	0.188	0.055	0.020	0.009
4	200	3.419	11.787	9.309	6.546	4.130	2.283	1.122	0.504	0.205	0.059	0.017	0.004
4	200	3.436	11.876	9.472	6.666	4.232	2.356	1.212	0.548	0.207	0.071	0.020	0.005
4	200	3.436	11.838	9.453	6.738	4.345	2.372	1.238	0.526	0.214	0.076	0.020	0.004
6	200	4.294	6.861	9.365	11.940	13.494	13.510	11.966	9.568	6.568	4.051	2.178	1.027
6	200	4.997	7.319	10.036	12.429	13.541	13.197	11.496	8.983	6.083	3.599	1.914	0.870
6	200	4.402	6.889	9.773	11.924	13.480	13.349	11.915	9.340	6.440	3.926	2.140	0.962
6	200	4.478	7.360	10.096	12.460	13.655	13.130	11.555	8.912	5.949	3.552	1.905	0.845
6	200	4.392	7.067	9.557	11.932	13.428	13.505	11.828	9.272	6.384	3.973	2.076	0.984
8	200	5.082	0.573	1.333	2.556	4.714	7.421	10.532	13.031	14.426	13.872	11.870	8.765
8	200	6.374	0.755	1.562	3.104	5.185	8.123	10.888	13.407	14.293	13.667	11.201	8.128
8	200	5.283	0.661	1.437	2.840	4.965	7.750	10.846	13.370	14.411	13.722	11.400	8.364
8	200	5.282	0.681	1.508	2.967	5.156	7.840	10.982	13.182	14.341	13.584	11.428	8.266
8	200	5.423	0.670	1.501	2.883	5.022	7.830	10.702	13.109	14.287	13.652	11.548	8.513

Table A.6: Subset sampling, $|S| = 25$; $n = 200$
(continued from previous page)

d	n	$\lambda(G)$	32 red	33 red	34 red	35 red	36 red	37 red	38 red	39 red	40 red	E(Red)	Theor. Reds
4	200	3.366	0.001	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	19.572	18.771
4	200	4.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	19.354	18.771
4	200	3.419	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	19.440	18.771
4	200	3.436	0.002	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	19.498	18.771
4	200	3.436	0.001	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	19.515	18.771
6	200	4.294	0.415	0.132	0.038	0.011	0.003	0.000	0.000	0.000	0.000	24.431	23.467
6	200	4.997	0.324	0.106	0.027	0.011	0.001	0.000	0.000	0.000	0.000	24.227	23.467
6	200	4.402	0.403	0.131	0.030	0.010	0.001	0.000	0.000	0.000	0.000	24.371	23.467
6	200	4.478	0.350	0.114	0.032	0.005	0.001	0.000	0.000	0.000	0.000	24.213	23.467
6	200	4.392	0.385	0.126	0.040	0.010	0.002	0.000	0.000	0.000	0.000	24.356	23.467
8	200	5.082	5.550	2.934	1.378	0.478	0.144	0.036	0.006	0.001	0.000	28.107	27.124
8	200	6.374	4.882	2.597	1.158	0.412	0.132	0.029	0.004	0.000	0.000	27.897	27.124
8	200	5.283	5.185	2.728	1.260	0.456	0.131	0.033	0.005	0.000	0.000	27.988	27.124
8	200	5.282	5.085	2.714	1.210	0.433	0.130	0.037	0.005	0.001	0.000	27.951	27.124
8	200	5.423	5.238	2.763	1.246	0.453	0.118	0.034	0.007	0.001	0.000	27.992	27.124

Table A.7: Community Detection – Algorithm Timing

N	GEN TYPE	λ_2	ALGORITHM	# COMMS	AVERAGE COMM. SIZE	# VERTICES IN COMM. WITH WORST EXP.	MIN V. EXP.	MIN E. EXP.	MEDIAN V. EXP.	MEDIAN E. EXP.	CHEEGER UPPER BOUND	SPECTRAL LOWER BOUND	SPECTRAL UPPER BOUND	% IMPROV. OVER UPPER BOUND	% ABOVE LOWER BOUND	TIME (SECS)
100	RANDOM	3.7618	EXTREMALOPTIMIZATION	23	4.3478	4	3	3.3333	3.6667	4	9.4036	0.6191	3.5188	5.2696	438.4304	4082.3736
100	RANDOM	3.7618	GENETICALGORITHM	42	2.381	5	2.6	2.6667	5	5	9.4036	0.6191	3.5188	24.2157	330.7443	1569.6727
100	RANDOM	3.7618	GREEDY	8	12.5	21	1.8095	1.9412	2.1381	2.25	9.4036	0.6191	3.5188	44.8335	213.5565	12.8817
100	RANDOM	3.7618	SIMULATEDANNEALING	100	1	1	3	3	5	5	9.4036	0.6191	3.5188	14.7427	384.5874	2.5343
100	RANDOM	3.7618	SPECTRALOPTIMIZATION	20	5	20	2.6	3.6667	4.3333	4.7292	9.4036	0.6191	3.5188	-4.2034	492.2735	116.4646

Table A.8: Community Detection – Greedy Algorithm Variant Timing

N	GEN TYPE	λ_2	ALGORITHM	# COMMS	AVERAGE COMM. SIZE	# VERTICES IN COMM. WITH WORST EXP.	MIN V. EXP.	MIN E. EXP.	MEDIAN V. EXP.	MEDIAN E. EXP.	CHEEGER UPPER BOUND	SPECTRAL LOWER BOUND	SPECTRAL UPPER BOUND	% IMPROV. OVER UPPER BOUND	% ABOVE LOWER BOUND	TIME (SECS)
100	RANDOM	3.7618	GREEDY	8	12.5	21	1.8095	1.9412	2.1381	2.2500	9.4036	0.6191	3.5188	44.8335	213.5565	9.4609
100	RANDOM	3.7618	RGPLUS	7	14.2857	22	1.5000	1.7273	1.8125	2.1250	9.4036	0.6191	3.5188	50.9125	179.0049	29.5869
100	RANDOM	3.7618	MSGVM	6	16.6667	24	1.4583	1.7500	1.9786	2.0362	9.4036	0.6191	3.5188	50.2666	182.6760	270.4000
100	RANDOM	3.7618	CD	10	10	11	2.0909	2.2727	2.4222	2.6000	9.4036	0.6191	3.5188	35.4111	267.1117	2505.1517
100	RANDOM	3.7618	LOUVAIN	2	50	49	0.7959	1.1633	0.7959	1.1633	9.4036	0.6191	3.5188	66.9410	87.9012	110.2937
100	RANDOM	3.7618	MOME	1	100	0	N/A	N/A	N/A	N/A	9.4036	0.6191	3.5188	N/A	N/A	20.0359

Table A.9: Community Detection – Random Graph Testing Results

n	algorithm	q	*initial*	λ_2	# comms	Average comm. size	# vertices in comm. with worst exp.	modularity	min v. exp.	min e. exp.	median v. exp.	median e. exp.	Cheeger upper bound	spectral lower bound	spectral upper bound	% improv. over upper bound	% above lower bound	time (secs)
100	Louvain	N/A	N/A	3.7618	2	50	49	0.2800	0.7959	1.1633	0.7959	1.1633	9.4036	0.6191	3.5188	66.9410	87.9012	109.0601
100	greedy	general	general	3.7618	8	12.5	21	0.4100	1.8095	1.9412	2.1381	2.2500	9.4036	0.6191	3.5188	44.8335	213.5565	7.7694
100	greedy	general	walkers	3.7618	8	12.5	23	0.4000	1.6957	1.8696	2.2898	2.3750	9.4036	0.6191	3.5188	46.8686	201.9892	21.7350
100	greedy	danon	general	3.7618	8	12.5	15	0.4100	1.8667	2.0667	2.1538	2.3077	9.4036	0.6191	3.5188	41.2672	233.8269	8.1405
100	greedy	danon	walkers	3.7618	9	11.1111	16	0.4000	1.7500	2.0000	2.3333	2.5000	9.4036	0.6191	3.5188	43.1618	223.0583	22.1263
100	greedy	wakita1	general	3.7618	8	12.5	21	0.4100	1.8095	1.9412	2.1381	2.2500	9.4036	0.6191	3.5188	44.8335	213.5565	8.5010
100	greedy	wakita1	walkers	3.7618	8	12.5	23	0.4000	1.6957	1.8696	2.2898	2.3750	9.4036	0.6191	3.5188	46.8686	201.9892	22.3027
200	Louvain	N/A	N/A	3.8805	1	200	0	N/A	N/A	N/A	N/A	N/A	8.7100	0.5597	3.3458	N/A	N/A	319.3290
200	greedy	general	general	3.8805	8	25	33	0.4500	1.6364	1.9062	1.9030	2.0770	8.7100	0.5597	3.3458	43.0262	240.5655	16.3802
200	greedy	general	walkers	3.8805	9	22.2222	35	0.4400	1.6000	1.8000	2.2222	2.3636	8.7100	0.5597	3.3458	46.2018	221.5832	87.9230
200	greedy	danon	general	3.8805	9	22.2222	30	0.4400	1.8000	2.0000	2.1200	2.2000	8.7100	0.5597	3.3458	40.2242	257.3147	21.1902
200	greedy	danon	walkers	3.8805	11	18.1818	27	0.4400	1.8148	1.8889	2.2308	2.3529	8.7100	0.5597	3.3458	43.5451	237.4639	91.8731
200	greedy	wakita1	general	3.8805	8	25	33	0.4500	1.6364	1.9062	1.9030	2.0770	8.7100	0.5597	3.3458	43.0262	240.5655	20.9406
200	greedy	wakita1	walkers	3.8805	9	22.2222	35	0.4400	1.6000	1.8000	2.2222	2.3636	8.7100	0.5597	3.3458	46.2018	221.5832	92.2740
300	Louvain	N/A	N/A	3.9131	1	300	0	N/A	N/A	N/A	N/A	N/A	8.5180	0.5435	3.2969	N/A	N/A	499.6951
300	greedy	general	general	3.9131	11	27.2727	42	0.4400	1.9048	2.1190	2.1935	2.3333	8.5180	0.5435	3.2969	35.7255	289.9123	28.8947
300	greedy	general	walkers	3.9131	12	25	35	0.4300	2.0571	2.1724	2.2703	2.4390	8.5180	0.5435	3.2969	34.1068	299.7318	218.2100
300	greedy	danon	general	3.9131	13	23.0769	31	0.4300	2.0968	2.2564	2.3684	2.4737	8.5180	0.5435	3.2969	31.5590	315.1875	43.4904
300	greedy	danon	walkers	3.9131	13	23.0769	30	0.4400	2.0000	2.1852	2.3571	2.4783	8.5180	0.5435	3.2969	33.7194	302.0818	231.8493
300	greedy	wakita1	general	3.9131	11	27.2727	42	0.4400	1.9048	2.1190	2.1935	2.3333	8.5180	0.5435	3.2969	35.7255	289.9123	42.5169
300	greedy	wakita1	walkers	3.9131	12	25	35	0.4300	2.0571	2.1724	2.2703	2.4390	8.5180	0.5435	3.2969	34.1068	299.7318	231.0450
400	Louvain	N/A	N/A	3.9204	1	400	0	N/A	N/A	N/A	N/A	N/A	8.4745	0.5398	3.2857	N/A	N/A	753.2198
400	greedy	general	general	3.9204	12	33.3333	46	0.4500	1.8696	2.0217	2.1909	2.3448	8.4745	0.5398	3.2857	38.4689	274.5362	43.5309
400	greedy	general	walkers	3.9204	12	33.3333	47	0.4500	1.8723	1.9787	2.1775	2.2866	8.4745	0.5398	3.2857	39.7781	266.5674	445.1850
400	greedy	danon	general	3.9204	12	33.3333	50	0.4400	1.9600	2.1064	2.1644	2.3727	8.4745	0.5398	3.2857	35.8928	290.2169	76.3948
400	greedy	danon	walkers	3.9204	14	28.5714	30	0.4400	2.1000	2.2667	2.2969	2.4183	8.4745	0.5398	3.2857	31.0146	319.9102	468.9196
400	greedy	wakita1	general	3.9204	12	33.3333	46	0.4500	1.8696	2.0217	2.1909	2.3448	8.4745	0.5398	3.2857	38.4689	274.5362	71.2310
400	greedy	wakita1	walkers	3.9204	12	33.3333	47	0.4500	1.8723	1.9787	2.1775	2.2866	8.4745	0.5398	3.2857	39.7781	266.5674	470.3922
500	Louvain	N/A	N/A	3.9217	1	500	0	N/A	N/A	N/A	N/A	N/A	8.4671	0.5392	3.2838	N/A	N/A	1052.7817
500	greedy	general	general	3.9217	12	41.6667	80	0.4400	1.7750	2.0000	2.2752	2.4220	8.4671	0.5392	3.2838	39.0951	270.9401	60.0907
500	greedy	general	walkers	3.9217	11	45.4545	71	0.4400	1.8873	2.0141	2.2564	2.3478	8.4671	0.5392	3.2838	38.6662	273.5523	779.1507
500	greedy	danon	general	3.9217	14	35.7143	76	0.4500	1.8289	2.0526	2.4131	2.4643	8.4671	0.5392	3.2838	37.4924	280.7017	119.0351
500	greedy	danon	walkers	3.9217	15	33.3333	48	0.4400	2.0000	2.2500	2.3750	2.4815	8.4671	0.5392	3.2838	31.4820	317.3076	806.3766
500	greedy	wakita1	general	3.9217	12	41.6667	80	0.4400	1.7750	2.0000	2.2752	2.4220	8.4671	0.5392	3.2838	39.0951	270.9401	101.2741
500	greedy	wakita1	walkers	3.9217	11	45.4545	71	0.4400	1.8873	2.0141	2.2564	2.3478	8.4671	0.5392	3.2838	38.6662	273.5523	730.3919

Table A.9: Community Detection – Random Graph Testing Results
(continued from previous page)

N	ALGORITHM	q	*initial*	λ_2	# COMMS	AVERAGE COMM. SIZE	# VERTICES IN COMM. WITH WORST EXP.	MODULARITY	MIN V. EXP.	MIN E. EXP.	MEDIAN V. EXP.	MEDIAN E. EXP.	CHEEGER UPPER BOUND	SPECTRAL LOWER BOUND	SPECTRAL UPPER BOUND	% IMPROV. OVER UPPER BOUND	% ABOVE LOWER BOUND	TIME (SECS)
600	LOUVAIN	N/A	N/A	3.9428	1	600	0	N/A	N/A	N/A	N/A	N/A	8.3414	0.5286	3.2514	N/A	N/A	1277.8239
600	GREEDY	GENERAL	GENERAL	3.9428	12	50	79	0.4400	1.8734	2.0886	2.1727	2.3297	8.3414	0.5286	3.2514	35.7627	295.1369	78.9809
600	GREEDY	GENERAL	WALKERS	3.9428	12	50	89	0.4400	1.6854	1.8764	2.2616	2.3981	8.3414	0.5286	3.2514	42.2892	254.9909	1266.0886
600	GREEDY	DANON	GENERAL	3.9428	16	37.5	71	0.4400	1.9718	2.1268	2.4259	2.5180	8.3414	0.5286	3.2514	34.5893	302.3549	183.1218
600	GREEDY	DANON	WALKERS	3.9428	13	46.1538	55	0.4400	2.0364	2.2364	2.2955	2.4182	8.3414	0.5286	3.2514	31.2183	323.0904	1382.3912
600	GREEDY	WAKITA1	GENERAL	3.9428	12	50	79	0.4400	1.8734	2.0886	2.1727	2.3297	8.3414	0.5286	3.2514	35.7627	295.1369	167.8138
600	GREEDY	WAKITA1	WALKERS	3.9428	12	50	89	0.4400	1.6854	1.8764	2.2616	2.3981	8.3414	0.5286	3.2514	42.2892	254.9909	1347.2672
700	LOUVAIN	N/A	N/A	3.9301	1	700	0	N/A	N/A	N/A	N/A	N/A	8.4170	0.5349	3.2709	N/A	N/A	1500.2534
700	GREEDY	GENERAL	GENERAL	3.9301	12	58.3333	86	0.4500	1.8372	2.0353	2.2193	2.3988	8.4170	0.5349	3.2709	37.7761	280.4672	102.0198
700	GREEDY	GENERAL	WALKERS	3.9301	12	58.3333	95	0.4400	1.8632	2.0244	2.3168	2.4339	8.4170	0.5349	3.2709	38.1095	278.4289	1930.4046
700	GREEDY	DANON	GENERAL	3.9301	16	43.75	68	0.4400	2.0735	2.2308	2.4553	2.5174	8.4170	0.5349	3.2709	31.8000	317.0083	263.0628
700	GREEDY	DANON	WALKERS	3.9301	17	41.1765	67	0.4400	2.0746	2.2090	2.4545	2.5385	8.4170	0.5349	3.2709	32.4669	312.9305	2050.2926
700	GREEDY	WAKITA1	GENERAL	3.9301	12	58.3333	86	0.4500	1.8372	2.0353	2.2193	2.3988	8.4170	0.5349	3.2709	37.7761	280.4672	234.7615
700	GREEDY	WAKITA1	WALKERS	3.9301	12	58.3333	95	0.4400	1.8632	2.0244	2.3168	2.4339	8.4170	0.5349	3.2709	38.1095	278.4289	2062.4834
800	LOUVAIN	N/A	N/A	3.9483	1	800	0	N/A	N/A	N/A	N/A	N/A	8.3089	0.5259	3.2430	N/A	N/A	2333.7424
800	GREEDY	GENERAL	GENERAL	3.9483	15	53.3333	106	0.4400	1.9151	2.0805	2.3958	2.4792	8.3089	0.5259	3.2430	35.8476	295.6365	129.4126
800	GREEDY	GENERAL	WALKERS	3.9483	16	50	90	0.4500	1.9333	2.0667	2.3358	2.4610	8.3089	0.5259	3.2430	36.2729	293.0135	2701.4665
800	GREEDY	DANON	GENERAL	3.9483	16	50	81	0.4400	1.9877	2.2346	2.3638	2.4484	8.3089	0.5259	3.2430	31.0956	324.9429	375.7258
800	GREEDY	DANON	WALKERS	3.9483	15	53.3333	85	0.4400	2.0706	2.1529	2.3462	2.4483	8.3089	0.5259	3.2430	33.6126	309.4201	3077.3684
800	GREEDY	WAKITA1	GENERAL	3.9483	15	53.3333	106	0.4400	1.9151	2.0805	2.3958	2.4792	8.3089	0.5259	3.2430	35.8476	295.6365	307.8351
800	GREEDY	WAKITA1	WALKERS	3.9483	16	50	90	0.4500	1.9333	2.0667	2.3358	2.4610	8.3089	0.5259	3.2430	36.2729	293.0135	2855.7331
900	LOUVAIN	N/A	N/A	3.9561	1	900	0	N/A	N/A	N/A	N/A	N/A	8.2625	0.5220	3.2309	N/A	N/A	2551.9475
900	GREEDY	GENERAL	GENERAL	3.9561	13	69.2308	108	0.4500	1.9537	2.1368	2.2237	2.3385	8.2625	0.5220	3.2309	33.8633	309.3950	159.6610
900	GREEDY	GENERAL	WALKERS	3.9561	15	60	94	0.4500	1.9362	2.0532	2.3226	2.4151	8.2625	0.5220	3.2309	36.4524	293.3684	3883.7847
900	GREEDY	DANON	GENERAL	3.9561	17	52.9412	71	0.4400	2.1972	2.3333	2.3913	2.4762	8.2625	0.5220	3.2309	27.7818	347.0405	497.7236
900	GREEDY	DANON	WALKERS	3.9561	17	52.9412	70	0.4500	2.2143	2.3143	2.3750	2.4545	8.2625	0.5220	3.2309	28.3713	343.3912	4577.0117
900	GREEDY	WAKITA1	GENERAL	3.9561	13	69.2308	108	0.4500	1.9537	2.1368	2.2237	2.3385	8.2625	0.5220	3.2309	33.8633	309.3950	413.5222
900	GREEDY	WAKITA1	WALKERS	3.9561	15	60	94	0.4500	1.9362	2.0532	2.3226	2.4151	8.2625	0.522	3.2309	36.4524	293.3684	4290.0652
1000	LOUVAIN	N/A	N/A	3.9563	1	1000	0	N/A	N/A	N/A	N/A	N/A	8.2612	0.5218	3.2306	N/A	N/A	3299.9272
1000	GREEDY	GENERAL	GENERAL	3.9563	15	66.6667	104	0.4500	1.9712	2.0865	2.3594	2.4773	8.2612	0.5218	3.2306	35.4137	299.8378	196.2924
1000	GREEDY	GENERAL	WALKERS	3.9563	13	76.9231	130	0.4500	1.8692	2.0769	2.1757	2.3647	8.2612	0.5218	3.2306	35.7114	297.9952	5181.5289
1000	GREEDY	DANON	GENERAL	3.9563	16	62.5	91	0.4400	2.1538	2.3000	2.3935	2.4766	8.2612	0.5218	3.2306	28.8063	340.7429	630.9505
1000	GREEDY	DANON	WALKERS	3.9563	16	62.5	74	0.4500	2.1216	2.2500	2.3473	2.4513	8.2612	0.5218	3.2306	30.3540	331.1615	5769.7194
1000	GREEDY	WAKITA1	GENERAL	3.9563	15	66.6667	104	0.4500	1.9712	2.0865	2.3594	2.4773	8.2612	0.5218	3.2306	35.4137	299.8378	552.6951
1000	GREEDY	WAKITA1	WALKERS	3.9563	13	76.9231	130	0.4500	1.8692	2.0769	2.1757	2.3647	8.2612	0.5218	3.2306	35.7114	297.9952	5580.5825

Table A.10: Community Detection – Explicit Graph Testing Results

n	algorithm	q	*initial*	λ_2	# comms	Average comm. size	# vertices in comm. with worst exp.	modularity	min v. exp.	min e. exp.	median v. exp.	median e. exp.	Cheeger upper bound	spectral lower bound	spectral upper bound	% improv. over upper bound	% above lower bound	time (secs)
98	Louvain	N/A	N/A	4.5647	1	98	0	N/A	N/A	N/A	N/A	N/A	4.3805	0.2177	2.0865	N/A	N/A	74.3549
98	greedy	general	general	4.5647	9	10.8889	18	0.3700	0.8889	1.1111	2.0000	2.2000	4.3805	0.2177	2.0865	46.7468	410.4619	6.6807
98	greedy	general	walkers	4.5647	5	19.6	28	0.4800	0.7143	0.7143	1.3913	1.6957	4.3805	0.2177	2.0865	65.7658	228.1541	10.4485
98	greedy	danon	general	4.5647	8	12.25	12	0.3700	1.1667	1.5000	1.7054	2.0000	4.3805	0.2177	2.0865	28.1082	589.1235	7.3523
98	greedy	danon	walkers	4.5647	5	19.6	28	0.4800	0.7143	0.7143	1.4091	1.7273	4.3805	0.2177	2.0865	65.7658	228.1541	10.6526
98	greedy	wakita1	general	4.5647	9	10.8889	18	0.3700	0.8889	1.1111	2.0000	2.2000	4.3805	0.2177	2.0865	46.7468	410.4619	7.5768
98	greedy	wakita1	walkers	4.5647	5	19.6	28	0.4800	0.7143	0.7143	1.3913	1.6957	4.3805	0.2177	2.0865	65.7658	228.1541	10.5747
162	Louvain	N/A	N/A	4.6766	3	54	80	0.3900	0.6750	0.8750	1.1667	1.3333	3.5687	0.1617	1.7985	51.3474	441.0464	165.3681
162	greedy	general	general	4.6766	11	14.7273	11	0.4200	1.0000	1.3636	1.8750	2.0000	3.5687	0.1617	1.7985	24.1777	743.1892	12.3224
162	greedy	general	walkers	4.6766	7	23.1429	28	0.5100	0.7143	0.7143	1.8333	1.9167	3.5687	0.1617	1.7985	60.2836	341.6705	29.0774
162	greedy	danon	general	4.6766	9	18	16	0.4300	1.0000	1.2500	1.8000	1.8000	3.5687	0.1617	1.7985	30.4962	672.9234	14.6975
162	greedy	danon	walkers	4.6766	8	20.25	28	0.5100	0.7143	0.7143	1.6795	1.9231	3.5687	0.1617	1.7985	60.2836	341.6705	29.5313
162	greedy	wakita1	general	4.6766	11	14.7273	11	0.4200	1.0000	1.3636	1.8750	2.0000	3.5687	0.1617	1.7985	24.1777	743.1892	14.5764
162	greedy	wakita1	walkers	4.6766	7	23.1429	28	0.5100	0.7143	0.7143	1.8333	1.9167	3.5687	0.1617	1.7985	60.2836	341.6705	29.7984
242	Louvain	N/A	N/A	4.7385	3	80.6667	90	0.4100	0.8333	1.2000	0.9286	1.2857	3.0914	0.1307	1.6171	25.7910	817.8292	249.1278
242	greedy	general	general	4.7385	12	20.1667	18	0.4600	1.1111	1.3333	1.8818	2.0000	3.0914	0.1307	1.6171	17.5456	919.8103	20.2857
242	greedy	general	walkers	4.7385	9	26.8889	28	0.5300	0.7143	0.7143	1.8333	1.9333	3.0914	0.1307	1.6171	55.8280	446.3269	70.1923
242	greedy	danon	general	4.7385	10	24.2	36	0.4500	1.0556	1.2778	1.8661	1.9286	3.0914	0.1307	1.6171	20.9812	877.3182	26.4625
242	greedy	danon	walkers	4.7385	11	22	28	0.5200	0.7143	0.7143	2.0357	2.1200	3.0914	0.1307	1.6171	55.8280	446.3269	71.3528
242	greedy	wakita1	general	4.7385	12	20.1667	18	0.4600	1.1111	1.3333	1.8818	2.0000	3.0914	0.1307	1.6171	17.5456	919.8103	26.2353
242	greedy	wakita1	walkers	4.7385	9	26.8889	28	0.5300	0.7143	0.7143	1.8333	1.9333	3.0914	0.1307	1.6171	55.8280	446.3269	71.7402
338	Louvain	N/A	N/A	4.7772	3	112.6667	144	0.4000	0.7222	1.0417	0.9262	1.2131	2.7794	0.1114	1.4927	30.2157	835.0102	384.2354
338	greedy	general	general	4.7772	14	24.1429	42	0.4600	1.1429	1.4286	2.0000	2.0909	2.7794	0.1114	1.4927	4.2959	1182.2998	32.0778
338	greedy	general	walkers	4.7772	9	37.5556	49	0.5400	0.7755	0.9184	1.6667	1.9048	2.7794	0.1114	1.4927	38.4759	724.3356	163.4707
338	greedy	danon	general	4.7772	14	24.1429	16	0.4700	1.0000	1.2500	1.8819	1.9375	2.7794	0.1114	1.4927	16.2589	1022.0123	51.7382
338	greedy	danon	walkers	4.7772	11	30.7273	30	0.5300	0.7000	0.8000	1.9286	2.1818	2.7794	0.1114	1.4927	46.4057	618.0879	171.9044
338	greedy	wakita1	general	4.7772	14	24.1429	42	0.4600	1.1429	1.4286	2.0000	2.0909	2.7794	0.1114	1.4927	4.2959	1182.2998	49.5417
338	greedy	wakita1	walkers	4.7772	9	37.5556	49	0.5400	0.7755	0.9184	1.6667	1.9048	2.7794	0.1114	1.4927	38.4759	724.3356	173.6838
450	Louvain	N/A	N/A	4.8034	1	450	0	N/A	N/A	N/A	N/A	N/A	2.5601	0.0983	1.4022	N/A	N/A	572.2092
450	greedy	general	general	4.8034	17	26.4706	46	0.4900	1.2826	1.4348	1.9286	2.0000	2.5601	0.0983	1.4022	-2.3251	1359.5145	47.6842
450	greedy	general	walkers	4.8034	8	56.25	70	0.5500	0.6714	0.8286	1.4138	1.6064	2.5601	0.0983	1.4022	40.9083	742.8539	341.2379
450	greedy	danon	general	4.8034	14	32.1429	48	0.4700	1.2083	1.4167	1.8711	2.0839	2.5601	0.0983	1.4022	-1.0331	1341.0863	94.6713
450	greedy	danon	walkers	4.8034	11	40.9091	40	0.5400	0.8000	0.9000	1.8868	2.0500	2.5601	0.0983	1.4022	35.8142	815.5137	354.5796
450	greedy	wakita1	general	4.8034	17	26.4706	46	0.4900	1.2826	1.4348	1.9286	2.0000	2.5601	0.0983	1.4022	-2.3251	1359.5145	86.5155
450	greedy	wakita1	walkers	4.8034	8	56.25	70	0.5500	0.6714	0.8286	1.4138	1.6064	2.5601	0.0983	1.4022	40.9083	742.8539	356.7103

Table A.10: Community Detection – Explicit Graph Testing Results
(continued from previous page)

n	algorithm	q	*initial*	λ_2	# comms	Average comm. size	# vertices in comm. with worst exp.	modularity	min v. exp.	min e. exp.	median v. exp.	median e. exp.	Cheeger upper bound	spectral lower bound	spectral upper bound	% improv. over upper bound	% above lower bound	time (secs)
578	Louvain	N/A	N/A	4.8222	1	578	0	N/A	N/A	N/A	N/A	N/A	2.3976	0.0889	1.3333	N/A	N/A	871.4070
578	greedy	general	general	4.8222	11	52.5455	74	0.4800	1.1757	1.4000	1.7500	1.8611	2.3976	0.0889	1.3333	-5.0015	1475.0435	71.7547
578	greedy	general	walkers	4.8222	7	82.5714	117	0.5700	0.5641	0.6581	1.3913	1.6327	2.3976	0.0889	1.3333	50.6403	640.4051	662.8636
578	greedy	danon	general	4.8222	15	38.5333	46	0.4900	1.1522	1.5217	1.8889	1.9444	2.3976	0.0889	1.3333	-14.1320	1612.0038	165.7118
578	greedy	danon	walkers	4.8222	13	44.4615	40	0.5400	0.8000	0.9000	1.6829	1.8800	2.3976	0.0889	1.3333	32.4991	912.5280	671.0379
578	greedy	wakita1	general	4.8222	11	52.5455	74	0.4800	1.1757	1.4000	1.7500	1.8611	2.3976	0.0889	1.3333	-5.0015	1475.0435	141.2943
578	greedy	wakita1	walkers	4.8222	7	82.5714	117	0.5700	0.5641	0.6581	1.3913	1.6327	2.3976	0.0889	1.3333	50.6403	640.4051	658.6821
722	Louvain	N/A	N/A	4.8364	1	722	0	N/A	N/A	N/A	N/A	N/A	2.2725	0.0818	1.2791	N/A	N/A	1557.1319
722	greedy	general	general	4.8364	15	48.1333	84	0.4700	1.5357	1.7600	2.0455	2.1250	2.2725	0.0818	1.2791	-37.5938	2051.3684	102.2198
722	greedy	general	walkers	4.8364	7	103.1429	150	0.5600	0.5267	0.6133	1.5625	1.7500	2.2725	0.0818	1.2791	52.0507	649.7193	1145.3087
722	greedy	danon	general	4.8364	15	48.1333	52	0.4700	1.3462	1.5000	2.0625	2.1250	2.2725	0.0818	1.2791	-17.2674	1733.5526	270.0365
722	greedy	danon	walkers	4.8364	10	72.2	100	0.5600	0.5900	0.7400	1.6526	1.8405	2.2725	0.0818	1.2791	42.1481	804.5526	1225.1693
722	greedy	wakita1	general	4.8364	15	48.1333	84	0.4700	1.5357	1.7600	2.0455	2.1250	2.2725	0.0818	1.2791	-37.5938	2051.3684	223.5113
722	greedy	wakita1	walkers	4.8364	7	103.1429	150	0.5600	0.5267	0.6133	1.5625	1.7500	2.2725	0.0818	1.2791	52.0507	649.7193	1177.1231
882	Louvain	N/A	N/A	4.8474	4	220.5	270	0.4500	0.8778	1.3274	1.1193	1.4398	2.1730	0.0763	1.2353	-7.4565	1639.7318	1937.0702
882	greedy	general	general	4.8474	15	58.8	112	0.5000	1.0446	1.3571	1.9375	2.0000	2.1730	0.0763	1.2353	-9.8614	1678.6686	149.4956
882	greedy	general	walkers	4.8474	6	147	198	0.5700	0.5051	0.6061	1.1989	1.4920	2.1730	0.0763	1.2353	50.9391	694.3018	2165.2580
882	greedy	danon	general	4.8474	17	51.8824	60	0.4900	1.3500	1.6333	1.9342	2.1290	2.1730	0.0763	1.2353	-32.2192	2040.6433	477.1742
882	greedy	danon	walkers	4.8474	10	88.2	115	0.5700	0.6000	0.7739	1.4645	1.7871	2.1730	0.0763	1.2353	37.3513	914.2888	2650.0855
882	greedy	wakita1	general	4.8474	15	58.8	112	0.5000	1.0446	1.3571	1.9375	2.0000	2.1730	0.0763	1.2353	-9.8614	1678.6686	400.8865
882	greedy	wakita1	walkers	4.8474	6	147	198	0.5700	0.5051	0.6061	1.1989	1.4920	2.1730	0.0763	1.2353	50.9391	694.3018	2355.2239
1058	Louvain	N/A	N/A	4.8562	1	1058	0	N/A	N/A	N/A	N/A	N/A	2.0919	0.0719	1.1991	N/A	N/A	2637.1348
1058	greedy	general	general	4.8562	11	96.1818	214	0.5100	0.7897	0.9533	1.4841	1.8571	2.0919	0.0719	1.1991	20.5026	1225.9250	209.8204
1058	greedy	general	walkers	4.8562	7	151.1429	214	0.5700	0.4766	0.5514	1.4040	1.6225	2.0919	0.0719	1.1991	54.0162	666.9566	3919.4795
1058	greedy	danon	general	4.8562	16	66.125	142	0.4900	1.2535	1.5667	1.9958	2.1125	2.0919	0.0719	1.1991	-30.6510	2079.1100	775.3061
1058	greedy	danon	walkers	4.8562	11	96.1818	158	0.5600	0.5506	0.7089	1.5840	1.8320	2.0919	0.0719	1.1991	40.8851	885.9695	4667.6648
1058	greedy	wakita1	general	4.8562	11	96.1818	214	0.5100	0.7897	0.9533	1.4841	1.8571	2.0919	0.0719	1.1991	20.5026	1225.9250	577.6686
1058	greedy	wakita1	walkers	4.8562	7	151.1429	214	0.5700	0.4766	0.5514	1.404	1.6225	2.0919	0.0719	1.1991	54.0162	666.9566	4312.958

Table A.11: Probability Amplification – Test Results

m	$\|V\|$	TRIALS	t	k (%)	# STEPS	WALK AVG	WALK MEDIAN	SAMPLE AVG	SAMPLE MEDIAN	% WALK	% SAMPLE	% DIFF
7	98	1000	5	5	11	27.35	28	24.20	25	0.279	0.247	0.032
7	98	1000	6	6	14	32.55	33	28.35	29	0.332	0.289	0.043
7	98	1000	7	7	17	36.36	37	32.17	32	0.371	0.328	0.043
7	98	1000	8	8	19	39.72	41	35.92	36	0.405	0.367	0.039
7	98	1000	9	9	22	43.42	44	39.15	39	0.443	0.399	0.044
7	98	1000	10	10	25	47.31	49	42.34	43	0.483	0.432	0.051
7	98	1000	11	11	28	50.84	52	45.69	46	0.519	0.466	0.053
7	98	1000	12	12	31	53.45	55	48.36	48	0.545	0.493	0.052
7	98	1000	13	13	34	56.32	58	51.23	51	0.575	0.523	0.052
7	98	1000	14	14	37	59.21	60	53.60	54	0.604	0.547	0.057
7	98	1000	15	15	39	60.80	62	56.01	56	0.620	0.572	0.049
7	98	1000	20	20	54	71.46	73	66.18	66	0.729	0.675	0.054
7	98	1000	25	25	68	78.03	79	74.28	74	0.796	0.758	0.038
7	98	1000	30	30	82	83.20	84	79.79	80	0.849	0.814	0.035
7	98	1000	40	40	111	89.84	91	87.70	88	0.917	0.895	0.022
7	98	1000	49	50	136	92.61	94	91.59	92	0.945	0.935	0.010
7	98	1000	59	60	165	94.68	96	94.29	94	0.966	0.962	0.004
7	98	1000	69	70	193	95.79	97	95.80	96	0.977	0.978	0.000
9	162	1000	9	5	25	54.54	56	43.65	44	0.337	0.269	0.067
9	162	1000	10	6	28	58.51	60	47.66	48	0.361	0.294	0.067
9	162	1000	12	7	34	67.63	69	55.16	55	0.417	0.340	0.077
9	162	1000	13	8	37	71.25	73	58.91	59	0.440	0.364	0.076
9	162	1000	15	9	44	79.62	82	65.61	66	0.491	0.405	0.086
9	162	1000	17	10	50	86.75	89	72.33	73	0.535	0.446	0.089
9	162	1000	18	11	53	88.84	91	75.01	75	0.548	0.463	0.085
9	162	1000	20	12	60	96.04	98	80.94	81	0.593	0.500	0.093
9	162	1000	22	13	66	101.32	103	86.45	87	0.625	0.534	0.092
9	162	1000	23	14	69	104.72	107	89.06	89	0.646	0.550	0.097
9	162	1000	25	15	75	108.66	110	94.17	94	0.671	0.581	0.089
9	162	1000	33	20	101	124.13	125	110.20	110	0.766	0.680	0.086
9	162	1000	41	25	126	134.56	136	122.80	123	0.831	0.758	0.073
9	162	1000	49	30	151	141.80	143	132.24	132	0.875	0.816	0.059
9	162	1000	65	40	202	150.48	152	144.65	145	0.929	0.893	0.036
9	162	1000	81	50	252	155.37	157	151.93	152	0.959	0.938	0.021
9	162	1000	98	60	306	158.30	159	156.14	156	0.977	0.964	0.013
9	162	1000	114	70	357	159.62	161	158.63	159	0.985	0.979	0.006
11	242	1000	13	5	40	83.79	86	63.69	64	0.346	0.263	0.083
11	242	1000	15	6	47	95.56	97	71.82	72	0.395	0.297	0.098
11	242	1000	17	7	54	103.86	106	79.81	80	0.429	0.330	0.099
11	242	1000	20	8	64	115.29	119	90.78	91	0.476	0.375	0.101
11	242	1000	22	9	71	125.75	127.5	97.74	98	0.520	0.404	0.116
11	242	1000	25	10	81	134.89	137	107.74	108	0.557	0.445	0.112
11	242	1000	27	11	88	141.88	145	113.59	114	0.586	0.469	0.117
11	242	1000	30	12	98	152.59	154.5	122.30	123	0.631	0.505	0.125
11	242	1000	32	13	105	157.16	159	128.16	128	0.649	0.530	0.120
11	242	1000	34	14	112	161.97	164	133.04	134	0.669	0.550	0.120
11	242	1000	37	15	122	168.68	171	140.32	140	0.697	0.580	0.117
11	242	1000	49	20	163	191.47	193	165.29	165	0.791	0.683	0.108
11	242	1000	61	25	204	206.68	208	183.78	184	0.854	0.759	0.095
11	242	1000	73	30	245	216.37	218	197.89	198	0.894	0.818	0.076
11	242	1000	97	40	327	228.42	230	216.98	217	0.944	0.897	0.047
11	242	1000	121	50	409	234.68	236	227.50	228	0.970	0.940	0.030
11	242	1000	146	60	494	237.57	239	233.66	234	0.982	0.966	0.016
11	242	1000	170	70	576	239.38	241	237.16	237	0.989	0.980	0.009

Table A.11: Probability Amplification – Test Results
(continued from previous page)

m	$\|V\|$	TRIALS	t	k (%)	# STEPS	WALK AVG	WALK MEDIAN	SAMPLE AVG	SAMPLE MEDIAN	% WALK	% SAMPLE	% DIFF
13	338	1000	17	5	57	118.08	121	84.69	85	0.349	0.251	0.099
13	338	1000	21	6	72	139.47	143	101.30	101	0.413	0.300	0.113
13	338	1000	24	7	83	154.88	158	112.97	113	0.458	0.334	0.124
13	338	1000	28	8	97	172.56	176	127.83	128	0.511	0.378	0.132
13	338	1000	31	9	108	182.81	186	138.27	138	0.541	0.409	0.132
13	338	1000	34	10	119	194.61	197	148.12	148	0.576	0.438	0.138
13	338	1000	38	11	133	208.81	212	160.49	161	0.618	0.475	0.143
13	338	1000	41	12	144	216.39	219	168.80	169	0.640	0.499	0.141
13	338	1000	44	13	155	225.81	229	177.46	177	0.668	0.525	0.143
13	338	1000	48	14	170	237.08	238	188.14	188	0.701	0.557	0.145
13	338	1000	51	15	180	242.78	245	195.68	196	0.718	0.579	0.139
13	338	1000	68	20	242	274.05	276	230.35	230	0.811	0.682	0.129
13	338	1000	85	25	303	295.02	298	257.69	258	0.873	0.762	0.110
13	338	1000	102	30	365	307.20	309	277.25	277	0.909	0.820	0.089
13	338	1000	136	40	488	322.44	325	303.61	304	0.954	0.898	0.056
13	338	1000	169	50	607	329.51	331	317.94	318	0.975	0.941	0.034
13	338	1000	203	60	730	332.90	335	326.43	327	0.985	0.966	0.019
13	338	1000	237	70	853	335.15	337	331.54	332	0.992	0.981	0.011
15	450	1000	23	5	83	169.05	172	114.74	115	0.376	0.255	0.121
15	450	1000	27	6	98	188.54	192	131.14	132	0.419	0.291	0.128
15	450	1000	32	7	117	214.00	217	151.05	152	0.476	0.336	0.140
15	450	1000	36	8	132	232.54	236	166.46	167	0.517	0.370	0.147
15	450	1000	41	9	151	251.27	254	183.19	183	0.558	0.407	0.151
15	450	1000	45	10	167	267.93	270	196.95	197	0.595	0.438	0.158
15	450	1000	50	11	185	284.85	288	213.11	213	0.633	0.474	0.159
15	450	1000	54	12	201	295.99	299.5	224.64	225	0.658	0.499	0.159
15	450	1000	59	13	220	312.15	316	238.55	239	0.694	0.530	0.164
15	450	1000	64	14	239	324.37	326	251.68	252	0.721	0.559	0.162
15	450	1000	68	15	254	334.22	336	261.33	261	0.743	0.581	0.162
15	450	1000	90	20	337	372.31	375	307.25	307	0.827	0.683	0.145
15	450	1000	113	25	425	398.08	401	343.15	344	0.885	0.763	0.122
15	450	1000	135	30	508	414.99	418	369.68	370	0.922	0.822	0.101
15	450	1000	180	40	679	433.36	436	404.43	404.5	0.963	0.899	0.064
15	450	1000	225	50	850	441.12	443	423.87	424	0.980	0.942	0.038
15	450	1000	270	60	1021	445.12	447	435.20	435	0.989	0.967	0.022
15	450	1000	315	70	1191	447.26	449	441.57	442	0.994	0.981	0.013
17	578	1000	29	5	110	220.57	223	145.61	146	0.382	0.252	0.130
17	578	1000	35	6	134	255.48	258	170.79	171	0.442	0.295	0.147
17	578	1000	41	7	158	285.42	290	194.58	195	0.494	0.337	0.157
17	578	1000	47	8	181	310.03	314	217.16	217	0.536	0.376	0.161
17	578	1000	53	9	205	334.31	337	237.69	238	0.578	0.411	0.167
17	578	1000	58	10	225	355.14	360	254.50	255	0.614	0.440	0.174
17	578	1000	64	11	248	375.28	379	273.17	273	0.649	0.473	0.177
17	578	1000	70	12	272	394.05	398	291.50	292	0.682	0.504	0.177
17	578	1000	76	13	296	410.03	413	307.40	307	0.709	0.532	0.178
17	578	1000	81	14	316	422.14	425	320.98	321	0.730	0.555	0.175
17	578	1000	87	15	339	436.45	441	335.56	336	0.755	0.581	0.175
17	578	1000	116	20	454	487.03	490	396.76	397	0.843	0.686	0.156
17	578	1000	145	25	569	518.81	523	441.70	442	0.898	0.764	0.133
17	578	1000	174	30	683	538.13	540	476.66	477	0.931	0.825	0.106
17	578	1000	232	40	912	559.78	562	520.27	520	0.968	0.900	0.068
17	578	1000	289	50	1138	568.99	571	545.08	545	0.984	0.943	0.041
17	578	1000	347	60	1367	572.81	575	559.32	560	0.991	0.968	0.023
17	578	1000	405	70	1596	575.13	577	567.48	568	0.995	0.982	0.013

Table A.11: Probability Amplification – Test Results
(continued from previous page)

m	$\|V\|$	TRIALS	t	k (%)	# STEPS	WALK AVG	WALK MEDIAN	SAMPLE AVG	SAMPLE MEDIAN	% WALK	% SAMPLE	% DIFF
19	722	1000	37	5	147	288.89	294	185.59	186	0.400	0.257	0.143
19	722	1000	44	6	175	326.89	333	215.25	216	0.453	0.298	0.155
19	722	1000	51	7	204	364.76	369	242.84	243	0.505	0.336	0.169
19	722	1000	58	8	233	395.04	399	269.35	270	0.547	0.373	0.174
19	722	1000	65	9	261	426.10	430	293.39	294	0.590	0.406	0.184
19	722	1000	73	10	294	456.89	461	320.15	320	0.633	0.443	0.189
19	722	1000	80	11	323	479.59	484	342.24	342	0.664	0.474	0.190
19	722	1000	87	12	351	501.04	504	362.71	363	0.694	0.502	0.192
19	722	1000	94	13	380	522.47	526	383.15	383	0.724	0.531	0.193
19	722	1000	102	14	413	540.86	544.5	404.32	405	0.749	0.560	0.189
19	722	1000	109	15	441	556.86	561	420.94	421	0.771	0.583	0.188
19	722	1000	145	20	588	617.59	621	496.84	498	0.855	0.688	0.167
19	722	1000	181	25	736	654.92	658	552.15	552	0.907	0.765	0.142
19	722	1000	217	30	883	678.02	681	594.79	594	0.939	0.824	0.115
19	722	1000	289	40	1177	701.24	705	650.05	650	0.971	0.900	0.071
19	722	1000	361	50	1472	712.21	714	681.41	682	0.986	0.944	0.043
19	722	1000	434	60	1770	716.80	719	699.31	700	0.993	0.969	0.024
19	722	1000	506	70	2065	719.19	721	709.04	709	0.996	0.982	0.014
21	882	1000	45	5	185	362.69	368	226.94	227	0.411	0.257	0.154
21	882	1000	53	6	219	405.73	412	260.90	261	0.460	0.296	0.164
21	882	1000	62	7	257	453.04	458	296.15	297	0.514	0.336	0.178
21	882	1000	71	8	294	497.26	502	330.08	330	0.564	0.374	0.190
21	882	1000	80	9	332	535.30	540	361.63	362	0.607	0.410	0.197
21	882	1000	89	10	370	569.33	574	391.79	392	0.645	0.444	0.201
21	882	1000	98	11	408	600.86	604	420.54	420	0.681	0.477	0.204
21	882	1000	106	12	442	623.93	628	443.34	444	0.707	0.503	0.205
21	882	1000	115	13	480	647.64	652	469.19	469	0.734	0.532	0.202
21	882	1000	124	14	518	669.34	674	493.06	493	0.759	0.559	0.200
21	882	1000	133	15	556	690.90	694	514.22	514	0.783	0.583	0.200
21	882	1000	177	20	741	763.90	768	607.05	607	0.866	0.688	0.178
21	882	1000	221	25	927	807.54	811	675.99	676	0.916	0.766	0.149
21	882	1000	265	30	1112	833.34	837	727.68	727	0.945	0.825	0.120
21	882	1000	353	40	1483	860.63	864	795.52	796	0.976	0.902	0.074
21	882	1000	441	50	1854	871.86	874	832.70	832.5	0.989	0.944	0.044
21	882	1000	530	60	2229	876.73	879	854.53	855	0.994	0.969	0.025
21	882	1000	618	70	2600	878.89	881	866.45	867	0.996	0.982	0.014
23	1058	1000	53	5	225	438.68	444	267.97	268	0.415	0.253	0.161
23	1058	1000	64	6	272	502.85	509	314.45	315	0.475	0.297	0.178
23	1058	1000	75	7	320	563.65	568.5	358.50	359	0.533	0.339	0.194
23	1058	1000	85	8	363	607.26	612	396.14	397	0.574	0.374	0.200
23	1058	1000	96	9	411	653.42	656	434.37	434	0.618	0.411	0.207
23	1058	1000	106	10	454	694.94	699	468.21	468	0.657	0.443	0.214
23	1058	1000	117	11	501	726.94	731	503.15	503	0.687	0.476	0.212
23	1058	1000	127	12	545	760.69	765	533.16	533	0.719	0.504	0.215
23	1058	1000	138	13	592	790.89	794.5	563.56	564	0.748	0.533	0.215
23	1058	1000	149	14	640	816.12	821	592.90	593	0.771	0.560	0.211
23	1058	1000	159	15	683	838.48	843	617.56	618	0.793	0.584	0.209
23	1058	1000	212	20	913	924.96	931	728.60	729	0.874	0.689	0.186
23	1058	1000	265	25	1142	974.51	978	811.97	812	0.921	0.767	0.154
23	1058	1000	318	30	1371	1005.59	1008	874.15	874	0.950	0.826	0.124
23	1058	1000	424	40	1830	1035.70	1039	954.72	955	0.979	0.902	0.077
23	1058	1000	529	50	2284	1047.69	1050	999.87	1000	0.990	0.945	0.045
23	1058	1000	635	60	2743	1052.74	1055	1025.16	1025	0.995	0.969	0.026
23	1058	1000	741	70	3202	1055.20	1057	1039.45	1040	0.997	0.982	0.015

Table A.11: Probability Amplification – Test Results
(continued from previous page)

m	$\|V\|$	TRIALS	t	k (%)	# STEPS	WALK AVG	WALK MEDIAN	SAMPLE AVG	SAMPLE MEDIAN	% WALK	% SAMPLE	% DIFF
25	1250	1000	63	5	274	531.59	538	319.08	319	0.425	0.255	0.170
25	1250	1000	75	6	327	602.42	607	369.99	370.5	0.482	0.296	0.186
25	1250	1000	88	7	385	670.92	677	422.27	423	0.537	0.338	0.199
25	1250	1000	100	8	438	729.95	735	467.00	467	0.584	0.374	0.210
25	1250	1000	113	9	496	784.85	791	513.66	514	0.628	0.411	0.217
25	1250	1000	125	10	549	831.55	836	553.65	554	0.665	0.443	0.222
25	1250	1000	138	11	607	872.28	877	594.11	595	0.698	0.475	0.223
25	1250	1000	150	12	660	908.40	915	630.93	631	0.727	0.505	0.222
25	1250	1000	163	13	717	944.49	950	666.88	667	0.756	0.534	0.222
25	1250	1000	176	14	775	979.17	986	701.25	702	0.783	0.561	0.222
25	1250	1000	188	15	828	1002.60	1008	730.31	731	0.802	0.584	0.218
25	1250	1000	250	20	1103	1101.77	1106	860.65	861	0.881	0.689	0.193
25	1250	1000	313	25	1382	1159.32	1163	960.87	961	0.927	0.769	0.159
25	1250	1000	375	30	1657	1193.52	1197	1033.22	1033	0.955	0.827	0.128
25	1250	1000	500	40	2210	1227.14	1230	1128.23	1128	0.982	0.903	0.079
25	1250	1000	625	50	2764	1239.44	1242	1181.38	1182	0.992	0.945	0.046
25	1250	1000	750	60	3318	1244.47	1247	1211.55	1211.5	0.996	0.969	0.026
25	1250	1000	875	70	3872	1247.00	1249	1228.10	1228	0.998	0.982	0.015
27	1458	1000	73	5	325	629.99	636	370.20	370	0.432	0.254	0.178
27	1458	1000	88	6	393	714.99	720	434.64	435	0.490	0.298	0.192
27	1458	1000	103	7	461	801.61	807	494.04	495	0.550	0.339	0.211
27	1458	1000	117	8	525	869.34	872	546.86	547	0.596	0.375	0.221
27	1458	1000	132	9	592	932.91	938.5	599.40	599	0.640	0.411	0.229
27	1458	1000	146	10	656	978.96	986.5	646.89	647	0.671	0.444	0.228
27	1458	1000	161	11	724	1035.88	1039	694.32	695	0.710	0.476	0.234
27	1458	1000	175	12	787	1076.59	1084	735.77	736	0.738	0.505	0.234
27	1458	1000	190	13	855	1117.38	1123	778.40	778	0.766	0.534	0.232
27	1458	1000	205	14	923	1152.43	1158	817.69	818	0.790	0.561	0.230
27	1458	1000	219	15	986	1178.85	1183	853.03	853	0.809	0.585	0.223
27	1458	1000	292	20	1317	1294.81	1300	1006.31	1007	0.888	0.690	0.198
27	1458	1000	365	25	1647	1360.10	1364	1121.14	1122	0.933	0.769	0.164
27	1458	1000	438	30	1977	1399.41	1403	1205.83	1206	0.960	0.827	0.133
27	1458	1000	584	40	2638	1433.94	1437	1316.46	1317	0.983	0.903	0.081
27	1458	1000	729	50	3295	1447.32	1450	1378.64	1379	0.993	0.946	0.047
27	1458	1000	875	60	3955	1452.92	1455	1414.15	1414	0.997	0.970	0.027
27	1458	1000	1021	70	4616	1455.34	1457	1433.06	1433	0.998	0.983	0.015
29	1682	1000	85	5	387	745.08	749	431.96	432	0.443	0.257	0.186
29	1682	1000	101	6	461	843.41	849	499.38	499	0.501	0.297	0.205
29	1682	1000	118	7	539	930.93	938	567.45	567	0.553	0.337	0.216
29	1682	1000	135	8	618	1013.16	1018	631.97	632	0.602	0.376	0.227
29	1682	1000	152	9	696	1086.03	1093	691.96	692	0.646	0.411	0.234
29	1682	1000	169	10	775	1152.51	1158	748.16	749	0.685	0.445	0.240
29	1682	1000	186	11	853	1210.79	1214.5	802.71	803	0.720	0.477	0.243
29	1682	1000	202	12	927	1253.97	1259	850.09	850	0.746	0.505	0.240
29	1682	1000	219	13	1006	1301.49	1308	897.64	898	0.774	0.534	0.240
29	1682	1000	236	14	1084	1340.29	1345	942.89	943	0.797	0.561	0.236
29	1682	1000	253	15	1163	1377.56	1383	984.90	985	0.819	0.586	0.233
29	1682	1000	337	20	1550	1505.07	1511	1161.19	1161	0.895	0.690	0.204
29	1682	1000	421	25	1938	1574.91	1580	1293.53	1294	0.936	0.769	0.167
29	1682	1000	505	30	2326	1617.60	1621	1392.09	1392	0.962	0.828	0.134
29	1682	1000	673	40	3101	1657.46	1661	1519.58	1520	0.985	0.903	0.082
29	1682	1000	841	50	3876	1671.39	1674	1591.01	1591	0.994	0.946	0.048
29	1682	1000	1010	60	4656	1676.91	1679	1631.06	1631	0.997	0.970	0.027
29	1682	1000	1178	70	5431	1679.19	1681	1652.92	1653	0.998	0.983	0.016

Table A.11: Probability Amplification – Test Results
(continued from previous page)

m	$\|V\|$	TRIALS	t	k (%)	# STEPS	WALK AVG	WALK MEDIAN	SAMPLE AVG	SAMPLE MEDIAN	% WALK	% SAMPLE	% DIFF
31	1922	1000	97	5	451	863.98	870	493.30	494	0.450	0.257	0.193
31	1922	1000	116	6	540	981.51	987	574.19	574	0.511	0.299	0.212
31	1922	1000	135	7	629	1083.88	1090	650.28	650	0.564	0.338	0.226
31	1922	1000	154	8	718	1176.09	1184	721.80	722	0.612	0.376	0.236
31	1922	1000	173	9	808	1255.23	1262	789.38	790	0.653	0.411	0.242
31	1922	1000	193	10	902	1332.32	1339	856.51	857	0.693	0.446	0.248
31	1922	1000	212	11	991	1391.14	1394	916.72	916	0.724	0.477	0.247
31	1922	1000	231	12	1080	1450.06	1457	972.05	973	0.754	0.506	0.249
31	1922	1000	250	13	1169	1504.07	1507.5	1025.83	1025	0.783	0.534	0.249
31	1922	1000	270	14	1263	1549.00	1553	1078.69	1079	0.806	0.561	0.245
31	1922	1000	289	15	1353	1585.52	1591.5	1126.22	1126	0.825	0.586	0.239
31	1922	1000	385	20	1804	1733.22	1738	1328.79	1329	0.902	0.691	0.210
31	1922	1000	481	25	2255	1808.34	1812	1477.78	1478	0.941	0.769	0.172
31	1922	1000	577	30	2706	1856.03	1859	1591.65	1592	0.966	0.828	0.138
31	1922	1000	769	40	3608	1896.46	1900	1737.15	1737	0.987	0.904	0.083
31	1922	1000	961	50	4510	1911.04	1914	1818.74	1819	0.994	0.946	0.048
31	1922	1000	1154	60	5416	1916.93	1919	1864.20	1864	0.997	0.970	0.027
31	1922	1000	1346	70	6318	1919.37	1921	1889.52	1890	0.999	0.983	0.016
33	2178	1000	109	5	515	986.50	994	555.60	556	0.453	0.255	0.198
33	2178	1000	131	6	620	1124.26	1132.5	648.58	649	0.516	0.298	0.218
33	2178	1000	153	7	725	1245.67	1252	736.97	737	0.572	0.338	0.234
33	2178	1000	175	8	830	1350.01	1358	820.15	820	0.620	0.377	0.243
33	2178	1000	197	9	936	1444.62	1450	898.42	898.5	0.663	0.412	0.251
33	2178	1000	218	10	1036	1523.68	1532	968.96	969	0.700	0.445	0.255
33	2178	1000	240	11	1141	1597.23	1603	1038.61	1039	0.733	0.477	0.256
33	2178	1000	262	12	1246	1662.18	1668.5	1103.37	1104	0.763	0.507	0.257
33	2178	1000	284	13	1351	1713.65	1718	1166.02	1165	0.787	0.535	0.251
33	2178	1000	305	14	1451	1763.91	1771	1222.88	1222	0.810	0.561	0.248
33	2178	1000	327	15	1556	1809.53	1815.5	1275.97	1275.5	0.831	0.586	0.245
33	2178	1000	436	20	2077	1971.11	1976.5	1506.13	1506	0.905	0.692	0.213
33	2178	1000	545	25	2597	2058.25	2062	1676.56	1676.5	0.945	0.770	0.175
33	2178	1000	654	30	3118	2108.15	2112	1804.21	1803	0.968	0.828	0.140
33	2178	1000	872	40	4159	2151.49	2155	1969.29	1970	0.988	0.904	0.084
33	2178	1000	1089	50	5195	2167.25	2170	2061.67	2062	0.995	0.947	0.048
33	2178	1000	1307	60	6237	2172.61	2175	2112.66	2113	0.998	0.970	0.028
33	2178	1000	1525	70	7278	2175.26	2177	2141.18	2141	0.999	0.983	0.016
35	2450	1000	123	5	591	1130.80	1135	626.51	627	0.462	0.256	0.206
35	2450	1000	147	6	707	1273.24	1281	729.62	730	0.520	0.298	0.222
35	2450	1000	172	7	829	1412.77	1421	830.32	831	0.577	0.339	0.238
35	2450	1000	196	8	945	1530.04	1536	920.47	920	0.625	0.376	0.249
35	2450	1000	221	9	1066	1635.90	1641	1009.20	1009	0.668	0.412	0.256
35	2450	1000	245	10	1183	1730.19	1735.5	1090.26	1090	0.706	0.445	0.261
35	2450	1000	270	11	1304	1810.21	1817	1169.21	1169	0.739	0.477	0.262
35	2450	1000	294	12	1420	1882.39	1891	1241.30	1242	0.768	0.507	0.262
35	2450	1000	319	13	1541	1946.46	1951	1310.68	1310	0.794	0.535	0.260
35	2450	1000	344	14	1663	2007.35	2014	1378.42	1378	0.819	0.563	0.257
35	2450	1000	368	15	1779	2051.87	2056	1438.44	1439	0.837	0.587	0.250
35	2450	1000	490	20	2371	2227.73	2233	1694.55	1694	0.909	0.692	0.218
35	2450	1000	613	25	2967	2322.07	2327	1886.44	1887	0.948	0.770	0.178
35	2450	1000	735	30	3559	2375.46	2380	2028.84	2029	0.970	0.828	0.141
35	2450	1000	980	40	4746	2422.55	2426	2215.74	2216	0.989	0.904	0.084
35	2450	1000	1225	50	5934	2438.47	2441	2319.04	2319	0.995	0.947	0.049
35	2450	1000	1470	60	7122	2444.82	2447	2377.24	2378	0.998	0.970	0.028
35	2450	1000	1715	70	8310	2447.20	2449	2409.19	2409	0.999	0.983	0.016

Table A.11: Probability Amplification – Test Results
(continued from previous page)

m	$\|V\|$	TRIALS	t	k (%)	# STEPS	WALK AVG	WALK MEDIAN	SAMPLE AVG	SAMPLE MEDIAN	% WALK	% SAMPLE	% DIFF
37	2738	1000	137	5	668	1278.02	1288	699.07	699	0.467	0.255	0.211
37	2738	1000	165	6	806	1445.37	1455	818.78	819	0.528	0.299	0.229
37	2738	1000	192	7	939	1598.70	1606	926.99	928	0.584	0.339	0.245
37	2738	1000	220	8	1077	1729.95	1737	1033.39	1034	0.632	0.377	0.254
37	2738	1000	247	9	1209	1843.93	1851	1129.90	1130	0.673	0.413	0.261
37	2738	1000	274	10	1342	1946.05	1952	1220.06	1221	0.711	0.446	0.265
37	2738	1000	302	11	1480	2039.90	2047	1308.40	1309	0.745	0.478	0.267
37	2738	1000	329	12	1613	2116.13	2124	1388.46	1388	0.773	0.507	0.266
37	2738	1000	356	13	1745	2188.20	2195	1465.86	1466	0.799	0.535	0.264
37	2738	1000	384	14	1883	2252.65	2259	1539.67	1539	0.823	0.562	0.260
37	2738	1000	411	15	2016	2306.50	2310	1606.28	1606.5	0.842	0.587	0.256
37	2738	1000	548	20	2690	2497.90	2504	1896.53	1897	0.912	0.693	0.220
37	2738	1000	685	25	3363	2604.14	2609	2110.12	2110	0.951	0.771	0.180
37	2738	1000	822	30	4037	2659.43	2663	2270.10	2270	0.971	0.829	0.142
37	2738	1000	1096	40	5385	2710.00	2714	2477.27	2478	0.990	0.905	0.085
37	2738	1000	1369	50	6727	2726.82	2729.5	2592.63	2593	0.996	0.947	0.049
37	2738	1000	1643	60	8075	2732.58	2735	2656.86	2657	0.998	0.970	0.028
37	2738	1000	1917	70	9422	2735.13	2737	2692.45	2693	0.999	0.983	0.016
39	3042	1000	153	5	757	1436.24	1444	781.88	782	0.472	0.257	0.215
39	3042	1000	183	6	906	1623.62	1631	909.27	910	0.534	0.299	0.235
39	3042	1000	213	7	1056	1794.54	1802	1029.33	1030	0.590	0.338	0.252
39	3042	1000	244	8	1210	1938.39	1944	1146.71	1147	0.637	0.377	0.260
39	3042	1000	274	9	1360	2070.28	2077	1254.85	1255	0.681	0.413	0.268
39	3042	1000	305	10	1514	2188.17	2196	1358.29	1359	0.719	0.447	0.273
39	3042	1000	335	11	1664	2285.50	2295.5	1453.35	1453	0.751	0.478	0.274
39	3042	1000	366	12	1818	2370.94	2379	1545.18	1546	0.779	0.508	0.271
39	3042	1000	396	13	1968	2450.78	2455	1630.23	1630	0.806	0.536	0.270
39	3042	1000	426	14	2117	2514.30	2519.5	1710.79	1711	0.827	0.562	0.264
39	3042	1000	457	15	2272	2577.09	2582	1786.39	1787	0.847	0.587	0.260
39	3042	1000	609	20	3029	2786.92	2793	2107.30	2109	0.916	0.693	0.223
39	3042	1000	761	25	3787	2899.55	2904	2343.72	2345	0.953	0.770	0.183
39	3042	1000	913	30	4544	2961.19	2967	2523.08	2524	0.973	0.829	0.144
39	3042	1000	1217	40	6059	3013.89	3018	2752.91	2753	0.991	0.905	0.086
39	3042	1000	1521	50	7574	3031.23	3034	2880.39	2881	0.996	0.947	0.050
39	3042	1000	1826	60	9094	3037.14	3039	2952.26	2952	0.998	0.970	0.028
39	3042	1000	2130	70	10609	3039.59	3041	2992.00	2992	0.999	0.984	0.016
41	3362	1000	169	5	847	1602.35	1610	863.65	864	0.477	0.257	0.220
41	3362	1000	202	6	1014	1814.40	1824	1004.42	1005.5	0.540	0.299	0.241
41	3362	1000	236	7	1185	2000.89	2010.5	1140.47	1141	0.595	0.339	0.256
41	3362	1000	269	8	1352	2163.60	2170	1266.47	1266	0.644	0.377	0.267
41	3362	1000	303	9	1523	2305.25	2310	1385.74	1386	0.686	0.412	0.274
41	3362	1000	337	10	1695	2435.34	2441	1501.31	1501	0.724	0.447	0.278
41	3362	1000	370	11	1861	2539.78	2546.5	1605.88	1606	0.755	0.478	0.278
41	3362	1000	404	12	2033	2637.91	2644	1708.09	1708	0.785	0.508	0.277
41	3362	1000	438	13	2204	2728.12	2731	1803.71	1804	0.811	0.536	0.275
41	3362	1000	471	14	2371	2798.26	2804	1892.77	1893	0.832	0.563	0.269
41	3362	1000	505	15	2542	2865.92	2873	1976.14	1977	0.852	0.588	0.265
41	3362	1000	673	20	3390	3091.90	3098	2329.43	2329	0.920	0.693	0.227
41	3362	1000	841	25	4238	3211.99	3217	2593.98	2595	0.955	0.772	0.184
41	3362	1000	1009	30	5085	3276.90	3282	2788.51	2789	0.975	0.829	0.145
41	3362	1000	1345	40	6781	3332.88	3337	3042.60	3043	0.991	0.905	0.086
41	3362	1000	1681	50	8476	3351.04	3354	3184.70	3185	0.997	0.947	0.049
41	3362	1000	2018	60	10176	3356.85	3359	3262.93	3263	0.998	0.971	0.028
41	3362	1000	2354	70	11871	3359.55	3361	3306.77	3307	0.999	0.984	0.016

Table A.11: Probability Amplification – Test Results
(continued from previous page)

m	$\|V\|$	TRIALS	t	k (%)	# STEPS	WALK AVG	WALK MEDIAN	SAMPLE AVG	SAMPLE MEDIAN	% WALK	% SAMPLE	% DIFF
43	3698	1000	185	5	939	1778.61	1785	946.64	947	0.481	0.256	0.225
43	3698	1000	222	6	1128	2013.62	2021	1104.70	1106	0.545	0.299	0.246
43	3698	1000	259	7	1316	2218.76	2228	1253.91	1254	0.600	0.339	0.261
43	3698	1000	296	8	1505	2399.17	2402	1393.40	1394	0.649	0.377	0.272
43	3698	1000	333	9	1694	2555.00	2564	1524.45	1524	0.691	0.412	0.279
43	3698	1000	370	10	1883	2695.74	2702	1649.18	1650	0.729	0.446	0.283
43	3698	1000	407	11	2072	2812.42	2822	1768.51	1768	0.761	0.478	0.282
43	3698	1000	444	12	2261	2919.75	2925	1877.55	1878	0.790	0.508	0.282
43	3698	1000	481	13	2450	3011.72	3020	1982.15	1983	0.814	0.536	0.278
43	3698	1000	518	14	2639	3096.03	3102	2081.50	2081	0.837	0.563	0.274
43	3698	1000	555	15	2827	3165.81	3172	2174.52	2175	0.856	0.588	0.268
43	3698	1000	740	20	3772	3414.73	3420	2561.82	2562	0.923	0.693	0.231
43	3698	1000	925	25	4716	3542.15	3547	2852.63	2853.5	0.958	0.771	0.186
43	3698	1000	1110	30	5661	3611.41	3616	3068.45	3069	0.977	0.830	0.147
43	3698	1000	1480	40	7549	3668.79	3672	3348.12	3349	0.992	0.905	0.087
43	3698	1000	1849	50	9433	3686.49	3689	3503.27	3503	0.997	0.947	0.050
43	3698	1000	2219	60	11322	3692.85	3695	3588.89	3589	0.999	0.970	0.028
43	3698	1000	2589	70	13210	3695.63	3697	3637.65	3638	0.999	0.984	0.016
45	4050	1000	203	5	1042	1966.22	1973	1038.49	1039	0.485	0.256	0.229
45	4050	1000	243	6	1248	2222.55	2230	1209.95	1210	0.549	0.299	0.250
45	4050	1000	284	7	1460	2452.11	2459	1375.50	1375	0.605	0.340	0.266
45	4050	1000	324	8	1667	2644.20	2651	1526.05	1526	0.653	0.377	0.276
45	4050	1000	365	9	1878	2816.30	2824	1672.83	1673	0.695	0.413	0.282
45	4050	1000	405	10	2085	2966.28	2974	1808.29	1807.5	0.732	0.446	0.286
45	4050	1000	446	11	2296	3103.39	3108	1937.81	1938	0.766	0.478	0.288
45	4050	1000	486	12	2503	3215.93	3222.5	2056.98	2057	0.794	0.508	0.286
45	4050	1000	527	13	2714	3319.92	3323	2174.02	2173	0.820	0.537	0.283
45	4050	1000	567	14	2921	3400.76	3408	2279.92	2280	0.840	0.563	0.277
45	4050	1000	608	15	3132	3483.83	3491	2381.75	2381	0.860	0.588	0.272
45	4050	1000	810	20	4175	3745.60	3752	2808.43	2808	0.925	0.693	0.231
45	4050	1000	1013	25	5223	3886.08	3892.5	3125.82	3126	0.960	0.772	0.188
45	4050	1000	1215	30	6265	3958.68	3962	3359.64	3360.5	0.977	0.830	0.148
45	4050	1000	1620	40	8355	4019.97	4023	3667.69	3669	0.993	0.906	0.087
45	4050	1000	2025	50	10446	4038.37	4041	3837.12	3837	0.997	0.947	0.050
45	4050	1000	2430	60	12536	4045.32	4047	3931.91	3932	0.999	0.971	0.028
45	4050	1000	2835	70	14626	4047.68	4049	3984.24	3984	0.999	0.984	0.016
47	4418	1000	221	5	1147	2164.33	2173.5	1132.37	1133	0.490	0.256	0.234
47	4418	1000	266	6	1382	2454.72	2464	1323.28	1324	0.556	0.300	0.256
47	4418	1000	310	7	1611	2696.70	2706	1500.79	1502	0.610	0.340	0.271
47	4418	1000	354	8	1840	2905.74	2914.5	1667.55	1668	0.658	0.377	0.280
47	4418	1000	398	9	2070	3095.80	3104.5	1825.32	1824.5	0.701	0.413	0.288
47	4418	1000	442	10	2299	3259.03	3263	1973.19	1974	0.738	0.447	0.291
47	4418	1000	486	11	2529	3401.19	3408	2112.77	2113	0.770	0.478	0.292
47	4418	1000	531	12	2764	3530.32	3537	2247.77	2248	0.799	0.509	0.290
47	4418	1000	575	13	2993	3634.55	3642	2370.78	2371	0.823	0.537	0.286
47	4418	1000	619	14	3222	3728.82	3738	2487.55	2487	0.844	0.563	0.281
47	4418	1000	663	15	3452	3814.15	3819	2597.98	2599	0.863	0.588	0.275
47	4418	1000	884	20	4604	4097.41	4103	3063.42	3063	0.927	0.693	0.234
47	4418	1000	1105	25	5757	4244.94	4251	3408.80	3409	0.961	0.772	0.189
47	4418	1000	1326	30	6910	4323.99	4329	3667.63	3669	0.979	0.830	0.149
47	4418	1000	1768	40	9215	4386.84	4391	4002.40	4004	0.993	0.906	0.087
47	4418	1000	2209	50	11515	4406.10	4409	4186.64	4187	0.997	0.948	0.050
47	4418	1000	2651	60	13820	4412.99	4415	4288.84	4289	0.999	0.971	0.028
47	4418	1000	3093	70	16125	4415.68	4417	4345.83	4346	0.999	0.984	0.016

Table A.11: Probability Amplification – Test Results
(continued from previous page)

m	$\|V\|$	TRIALS	t	k (%)	# STEPS	WALK AVG	WALK MEDIAN	SAMPLE AVG	SAMPLE MEDIAN	% WALK	% SAMPLE	% DIFF
49	4802	1000	241	5	1264	2375.36	2384	1234.41	1235	0.495	0.257	0.238
49	4802	1000	289	6	1516	2685.68	2691.5	1439.70	1439	0.559	0.300	0.259
49	4802	1000	337	7	1769	2950.07	2955	1633.39	1634	0.614	0.340	0.274
49	4802	1000	385	8	2022	3187.51	3194	1813.41	1814	0.664	0.378	0.286
49	4802	1000	433	9	2275	3384.66	3392.5	1986.99	1987	0.705	0.414	0.291
49	4802	1000	481	10	2528	3558.65	3565	2147.50	2147	0.741	0.447	0.294
49	4802	1000	529	11	2780	3712.07	3720	2300.38	2300	0.773	0.479	0.294
49	4802	1000	577	12	3033	3853.75	3864	2443.89	2443	0.803	0.509	0.294
49	4802	1000	625	13	3286	3971.12	3979	2579.30	2580	0.827	0.537	0.290
49	4802	1000	673	14	3539	4069.91	4074	2705.31	2706	0.848	0.563	0.284
49	4802	1000	721	15	3792	4168.62	4174	2826.91	2828	0.868	0.589	0.279
49	4802	1000	961	20	5056	4467.31	4472	3331.66	3331	0.930	0.694	0.236
49	4802	1000	1201	25	6320	4623.62	4629	3706.82	3706	0.963	0.772	0.191
49	4802	1000	1441	30	7584	4703.37	4709	3986.92	3987	0.979	0.830	0.149
49	4802	1000	1921	40	10112	4770.80	4774	4350.09	4351	0.994	0.906	0.088
49	4802	1000	2401	50	12640	4790.69	4793	4550.39	4551	0.998	0.948	0.050
49	4802	1000	2882	60	15174	4797.24	4799	4662.55	4663	0.999	0.971	0.028
49	4802	1000	3362	70	17702	4799.56	4801	4723.98	4724	0.999	0.984	0.016

Bibliography

[1] N. Alon. Eigenvalues and expanders. *Combinatorica*, 6(2):83–96, 1986.

[2] N. Alon and V. D. Milman. λ_1, isoperimetric inequalities for graphs, and superconcentrators. *Journal of Combinatorial Theory*, Series B(38):73–88, 1985.

[3] C. Ambühl, M. Mastrolilli, and O. Svensson. Inapproximability results for sparsest cut, optimal linear arrangement, and precedence constrained scheduling. In *Proc. of the IEEE 48th Annual Symposium on Foundations of Computer Science (FOCS)*, pages 329–337, Providence, Rhode Island, 2007.

[4] S. Arora, B. Barak, and D. Steurer. Subexponential algorithms for unique games and related problems. In *Proc. of the IEEE 51st Annual Symposium on Foundations of Computer Science (FOCS)*, Las Vegas, Nevada, 2010.

[5] S. Arora, S. Rao, and U. Vazirani. Expander flows, geometric embeddings and graph partitioning. In *Proc. of the Thirty-Sixth Annual ACM Symposium on Theory of Computing (STOC)*, Chicago, Illinois, 2004.

[6] Y. Azar, A. Broder, A. Karlin, and E. Upfal. Balanced allocations. *SIAM Journal on Computing*, 29(1):180–200, 1999.

[7] M. Blum and S. Micali. How to generate cryptographically strong sequences of pseudo-random bits. *SIAM Journal on Computing*, 13:270–299, 1984.

[8] S. Bobkov, C. Houdre, and P. Tetali. $\lambda\infty$, vertex isoperimetry, and concentration. *Combinatorica*, 20(2):153–172, 2000.

[9] J. Cheeger. A lower bound on smallest eigenvalue of a laplacian. In *Problems in Analysis (Papers dedicated to Salomon Bochner, 1969)*, pages 195–199, 1970.

[10] M. Chen, K. Kuzmin, and B. K. Szymanski. Community detection via maximization of modularity and its variants. *IEEE Transactions on Computational Social Systems*, 1(1):46–65, 2014.

[11] F. Chung. On concentrators, superconcentrators, generalizers, and nonblocking networks. *The Bell System Technical Journal*, 58(8):1765–1777, 1978.

[12] G. Csardi and T. Nepusz. The igraph software package for complex network research. *InterJournal*, Complex Systems:1695, 2006.

[13] L. Danon, J. Duch, A. Diaz-Guilerá, and A. Arenas. Comparing community structure identification. *Journal of Statistical Mechanics*, page P09008, 2005.

[14] I. Dinur. The pcp theorem by gap amplification. *Journal of the ACM*, 54(3), 2007.

[15] S. Fortunato and M. Barthelemy. Resolution limit in community detection. *Proceedings of the National Academy of Sciences of the United States of America*, 104(1):36–41, 2007.

[16] J. Friedman. On the second eigenvalue and random walks in random d-regular graphs. *Combinatorica*, 11(4):331–362, 1991.

[17] J. Friedman. A proof of alon's second eigenvalue conjecture and related problems. In *Proc. of the Thirty-Fifth Annual ACM Symposium on Theory of Computing (STOC)*, San Diega, California, 2003.

[18] J. Friedman. *Memoirs of the American Mathematical Society: A Proof of Alon's Second Eigenvalue Conjecture and Related Problems*. American Mathematical Society, Providence, Rhode Island, 2008.

[19] A. Frieze, P. Melsted, and M. Mitzenmacher. An analysis of random walk cuckoo hashing. In *13th International Workshop on Randomized Techniques in Computation*, pages 490–503, 2009.

[20] O. Gabber and Z. Galil. Explicit constructions of linear-sized superconcentrators. *Journal of Computer and System Sciences*, 22(3):407–420, 1981.

[21] D. Gillman. A chernoff bound for random walks on expander graphs. *SIAM Journal on Computing*, 27(4):1203–1220, 1998.

[22] R. Guimerá and L. A. N. Amaral. Functional cartography of complex metabolic networks. *Nature*, 433:895–900, 2005.

[23] V. Guruswami, C. Umans, and S. P. Vadhan. Unbalanced expanders and randomness extractors from parvaresh-vardy codes. In *Proceedings of the IEEE Conference on Computational Complexity*, pages 96–108, 2007.

[24] S. Hoory, N. Linial, and A. Wigderson. Expander graphs and their applications. *Bulletin of the American Mathematical Society*, 43(4):439–561, 2006.

[25] R. Impagliazzo and D. Zuckerman. How to recycle random bits. In *Proceedings of the 30th Annual IEEE Symposium on the Foundations of Computer Science*, pages 248–253, 1989.

[26] V. Kaibel. On the expansion of graphs of 0/1-polytopes. In M. Grøtschel, editor, *The Sharpest Cut: The Impact of Manfred Padberg and His Work*, pages 199–216. SIAM, 2004.

[27] S. Khot. On the power of unique 2-prover 1-round games. In *Proc. of the Thirty-Fourth Annual ACM Symposium on Theory of Computing (STOC)*, Montreal, Quebec, Canada, 2002.

[28] S. Khot, G. Kindler, E. Mossel, and R. O'Donnell. Optimal inapproximability results for max-cut and other 2-variable csps? *SIAM Journal on Computing*, 37(1):319–357, 2007.

[29] J. Kim and V. Vu. Generating random regular graphs. In *Proc. of the Thirty-Fifth Annual ACM Symposium on Theory of Computing (STOC)*, San Diego, California, 2003.

[30] F. T. Leighton and S. Rao. Multicommodity max-flow min-cut theorems and their use in designing approximation algorithms. *Journal of the ACM*, 46(6):787–832, 1999.

[31] N. Linial and A. Wigderson. Expander graphs and their applications. Lecture notes from a course. 2003.

[32] A. Louis, P. Raghavendra, and S. Vempala. The complexity of approximating vertex expansion. In *Proc. of the IEEE 54th Annual Symposium on Foundations of Computer Science (FOCS)*, pages 360–369, Berkeley, California, 2013.

[33] A. Lubotzky, R. Phillips, and P. Sarnak. Ramanujan graphs. *Combinatorica*, 8(3):261–277, 1988.

[34] G. Margulis. Explicit constructions of expanders. *Problemy Peredachi Informatsii*, 9(4):71–80, 1973.

[35] M. Mitzenmacher. Some open questions related to cuckoo hashing. In *Proceedings of the 17th Annual European Symposium on Algorithms*, pages 1–10, 2009.

[36] D. Moshkovitz and R. Raz. Two query pcp with sub-constant error. In *Proc. of the IEEE 49th Annual Symposium on Foundations of Computer Science (FOCS)*, pages 314–323, 2008.

[37] M. E. J. Newman. Fast algorithm for detecting community structure in networks. *Physical Review E*, 69(066133), 2004.

[38] M. E. J. Newman. Finding community structure in networks using the eigenvectors of matrices. *Physical Review E*, 74(036104), 2006.

[39] M. E. J. Newman. Modularity and community structure in networks. In *Proceedings of the National Academy of Sciences*, volume 103, pages 8577–8582, 2006.

[40] M.E. J. Newman and M. Girvan. Finding and evaluating community structure in networks. *Physical Review E*, 69(026113), 2004.

[41] A. Pagh and F. Rodler. Cuckoo hashing. In *Proceedings of the 9th Annual European Symposium on Algorithms*, pages 121–133, 2001.

[42] M. Pinsker. On the complexity of a concentrator. In *Proc. of the 7th Annual Teletraffic Conference*, pages 318/1–318/4, Stockholm, Sweden, 1973.

[43] P. Raghavendra and D. Steurer. Graph expansion and the unique games conjecture. In *Proc. of the Forty-Second Annual ACM Symposium on Theory of Computing (STOC)*, Cambridge, Massachusettes, 2010.

[44] P. Raghavendra, D. Steurer, and M. Tulsiani. Reductions between expansion problems. *Computing Research Repository (CoRR)*, abs/1011.2586, 2010.

[45] O. Reingold, S. Vadhan, and A. Wigderson. Entropy waves, the zig-zag graph product, and new constant-degree expanders. *Annals of Mathematics*, 155(1):157–187, 2002.

[46] M. Schelling and C. Hui. *modMax: Community Structure Detection via Modularity Maximization*, 2015. R package version 1.1.

[47] L. Trevisan. Expansion, sparsest cut, and spectal graph theory. Lecture notes from a course. 2014.

[48] L. Valiant. Graph-theoretic properties in computational complexity. *Journal of Computer and System Sciences*, 13(3):278–285, 1976.

[49] K. Wakita and T. Tsurumi. Finding community structure in mega-scale social networks. In *16th International Conference on World Wide Web (WWW '07)*, pages 1275–1276.

[50] A. Yao. Theory and applications of trapdoor functions. In *23rd Annual IEEE Symposium on the Foundations of Computer Science*, 1982.