HARVARD UNIVERSITY
Graduate School of Arts and Sciences

DISSERTATION ACCEPTANCE CERTIFICATE

The undersigned, appointed by the

Harvard John A. Paulson School of Engineering and Applied Sciences

have examined a dissertation entitled:

"Sorting Shapes the Performance of Graph-Structured Systems"

presented by: Daniel Wyatt Margo

candidate for the degree of Doctor of Philosophy and here by certify that it is worthy of acceptance.

Signature [illegible]
Typed name: Professor M. Seltzer

Signature [illegible]
Typed name: Professor M. Mitzenmacher

Signature [illegible]
Typed name: Professor S. Idreos

Date: March 27, 2017

Sorting Shapes the Performance of Graph-Structured Systems

A DISSERTATION PRESENTED
BY
DANIEL WYATT MARGO
TO
THE DEPARTMENT OF COMPUTER SCIENCE

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY
IN THE SUBJECT OF
COMPUTER SCIENCE

HARVARD UNIVERSITY
CAMBRIDGE, MASSACHUSETTS
MARCH 2017

Thesis advisor: Professor Margo Seltzer Daniel Wyatt Margo

Sorting Shapes the Performance of Graph-Structured Systems

Abstract

Graphs are, historically, among the most important and useful abstractions in both theoretical and applied computer science. However, due to a recent abundance of scientific data, social data, and machine learning and analysis methods based on graphical models, we are now facing a radical expansion of graph computing into new research areas. Dozens of graph-structured computing systems were published in the last decade, and graph theory itself is flourishing due to major results, such as the graph structure theorem.

This dissertation presents several projects at the intersection of graph-structured systems, graph theory, and research practice, which are united by the common theme of vertex sort orders. We first present Sheep, a novel distributed graph partitioning algorithm based on vertex sorting. We show that Sheep finds balanced edge partitions with a low communication volume in near-linear time with excellent distributed scaling properties.

Afterwards, we investigate graph systems research practices. We construct a corpus of research papers and identify the most referenced benchmark algorithms and datasets. We discuss systematic biases in these benchmarks, and in particular the hidden performance impact of their "default" vertex sort orders. We show this impact is significant by benchmarking a gold standard system, Galois, while controlling the vertex order.

Finally, we address the role of synthetic data generators in graph systems research. We identify several features of the popular Kronecker graph model that inhibit its correct use as a standard benchmark. We propose and evaluate changes to the model to make it an appropriate benchmark.

Thesis advisor: Professor Margo Seltzer Daniel Wyatt Margo

This dissertation shows that a graph's input shape and the performance of graph-structured system are inseparable; that a good graph shape (e.g., a partitioning) can be found as quickly as sorting; and that, in general, one *must* control the input shape to correctly measure a system's performance.

Contents

for John

Acknowledgments

I have worked with and been supported by many people, and I am very grateful to all of them, including whomever I may fail to mention here.

I want to thank my adviser, Margo Seltzer. Margo has been incredibly supportive of my research, even when my topics were somewhat unorthodox. Most of my ideas could not have been realized without Margo's deep, practical knowledge of the scientific process. I also want to thank the members of my dissertation committee, Michael Mitzenmacher and Stratos Idreos.

I want to thank Peter Macko. Peter and I worked together on many projects, and I greatly admire his talent and especially his work ethic. I also want to thank David Holland, who has a truly impressive mastery of myriad topics in computer science and is always willing to share.

Thanks to Uri Braun for some truly good conversations. Thanks to the many other members of my research group, some of whom include Elaine Angelino, Robert Bowden, Marc Chiarini, Kiran-Kumar Muniswamy-Reddy, Nicholas Murphy, Jackson Okuhn, Robin Smogor, Amos Waterland, and Susan Welby.

Thank you all for being my friends.

I am grateful for the generosity of Oracle and the Siebel Foundation, who both supported my research for several years. I also want to thank Jonathan Zittrain for introducing me to the Berkman community, which has greatly expanded my ways of thinking about society.

I want to thank my mother, Wendy, and my father, Robert, for loving and supporting an odd child. And I especially want to thank my husband, John, for loving and supporting an odd adult.

1
Introduction

Graphs are a common data representation in all areas of computer science. Important graphical models include spatial meshes, Bayesian variable networks, neural networks, social networks, distributed computer networks, and the Web. Naturally, graphs are the subject of many decades of strong research from different fields with diverse methodologies. However, these ambitions are challenged by the theoretical and practical reality that it is hard to mine knowledge from graphs.

Questions about graphs that appear simple, such as whether a graph contains a given pattern, may reveal themselves to be extensive topics spanning four decades of ongoing algorithm research[122]. This has immediate, painful implications for every software engineer who simply wants to build a query engine on top of graph data. Practically, we need these software tools because raw, unprocessed graph data proves incomprehensible and useless. Sophisticated embedding models are required just to visualize small graphs[117]; extracting meaning from large networks might accurately be described as one of the major goals of the 21st century "big data" movement.

At the turn of the millenium and with the inception of the Web it became obvious that we were on the cusp of an explosion of big graph data. Major works in this period, such as Pagerank[97] and Barabasi and Albert's preferential attachment model[13], wrestled with this data and helped establish a research agenda for the following two decades. For system builders who work with graphs, the most influential idea of the past 6 years is certainly Pregel[85], a distributed model for computing parallel algorithms on graph-structured data. Pregel's model was adopted by many dozens of different research systems within a relatively short period of rapid graph systems development. These systems have matured, and with maturity has come a better understanding of their fundamental problems.

Data bandwidth is a fundamental problem for contemporary graph data processing systems. Graph algorithm access patterns are, in general, not serial, because a stream of edges may connect to arbitrary vertices, and vice versa. If the graph is distributed, then data bandwidth is further throttled by the network and the widespread use of message passing and consensus mechanisms in contemporary graph systems. Most importantly, both graph serialization and distribution translate directly to the well-known NP-hard problems of bandwidth minimization and partitioning, respectively. The consequence of all this is that many graph algorithms and systems are stalled waiting for data. This phenomenon is often misleadingly described as "random access," but in practice graphs are quite structured by e.g., spatial constraints or power laws.

The first contribution of this dissertation is my solution to the data bandwidth problem. I design and implement a low-communication distributed algorithm to "sort" and partition a graph, and thereby improve the data bandwidth of later computation stages. My algorithm, *Sheep*, finds competitive vertex cut-minimizing solutions several orders of magnitude more quickly than the next-fastest solvers.

During this research I became aware of many subtle issues in the assessment and measurement of graph systems' performance. Consequently, it seemed important to ask whether or not published systems are correctly measured, and if not, what implications this might have for the research field. *The second contribution of this dissertation is a quantitative metastudy of algorithms and datasets in graph systems research following Pregel.* I identify many systematic errors in published research that are potentially significant and may have impacted the recent direction of the field. I substantiate this claim by reevaluating the top algorithms and datasets while properly controlling for these factors.

Much of the error in graph systems research is the result of a crisis in the availability of representative datasets. Everyone agrees that big graph data is important, but precisely because it is valuable, no one wants to release it. Privacy concerns introduce legal liabilities, and anonymization proves to be yet another problem with deep graph-theoretical challenges. Research worsens this situation by fixating on ten or so famous datasets used in comparative studies, most of which possess unique features that inhibit generalization.

Most fields are rescued from this quandary by standardized benchmarks with synthetic data generators. To no one's surprise, synthetic graph generation also turns out to be a "hard open problem"[79]. Whatever the state of the art, my study shows that the overwhelming majority of published research uses the Kronecker model, a synthetic model with significant and well-documented statistical flaws[110]. Nevertheless, the Kronecker model is popular because it can efficiently generate large graphs in parallel and has convenient parameters. *The final contribution of this dissertation is a fix to the Kronecker generator's algorithm that corrects its flaws without otherwise disrupting the model's desirable benchmark properties.* My revised algorithm is a strict superset of the previous and is therefore compatible with the benchmark parameters already circulating in the literature.

In summary, this dissertation shows that:

1. You *can* sort and partition a graph well and quickly at scale. I describe Sheep, an algorithm that accomplishes this task.

2. You *must* experimentally control certain factors, such as vertex sort order, when you conduct research with graph data. I show the consequences if you do not.

3. Graph systems research suffers from a lack of diversity in benchmark algorithms and especially datasets. I show this by quantitatively studying the literature.

4. You *can* generate synthetic benchmark data with all of the conveniences of the Kronecker model and none of its known flaws. I present an algorithm that accomplishes this.

1.1 Overview

In the following Chapter, I review the research history of graph-structured data processing systems. I begin with the early history of the field through the 1990s, when spatial and physical graphical models dominated much of the research literature. I then descibe the explosion of new graphical models, such as social networks, that grew along with the Internet in the 90s and 2000s. This led to a new wave of graph processing systems, such as Pregel[85] in 2010, that grew up alongside the "big data" movement. I compare these systems with their traditional sparse matrix processing counterparts and discuss some shared concerns of these diverse systems as they relate to abstract problems such as graph partitioning and vertex sorting.

In Chapter 3, I present Sheep, a graph preprocessing system that addresses data bandwidth problems by reordering and partitioning a graph dataset. Sheep is parallelized, efficiently distributed, and tries to efficiently solve the partition problem *as quickly as possible* in order to scale to large datasets. I briefly survey the research literature of competing partitioners and reorderers in Section 3.2. In Section 3.3 I first review graph terminology and important concepts, and then explain a simplified serial version of my partitioning algorithm in reference to those concepts. I show that the quality of Sheep's result is loosely bounded in terms of its communication volume, which is a standard partitioning metric. I then show in Section 3.4 that the algorithm can be efficiently distributed by an elegant use of a union-find data structure. Finally, I offer some intuition as to the nature of the partitioning solutions that Sheep finds. In Section 3.5 I describe my implementation of the algorithm and how I evaluate it against competing algorithms in the literature; I show that Sheep produces partitioning results of comparable quality but in less time by several orders of magnitude.

Sheep addresses a recurring data bandwidth problem that persists in modern graph processing system designs. In Chapter 4, I discuss these designs in-depth. I review important and recent systems of the last decade organized by their research community and identify the novel contributions of each system, as well as features adopted from prior work. I then generalize these features to construct a taxonomy of contemporary graph systems research in Section 4.5.

In my experience working with these systems, I became conscious of systematic flaws in their evaluations and conclusions. Chapter 5 presents a quantitative metastudy of published graph systems research and evaluations in the period 2011 – 2015, inclusive. In Section 5.2 I discuss the methodology I used to select 65 relevant graph systems papers from a corpus of over 3236 conference publications. I then document and discuss the algorithms and datasets used as performance benchmarks by these papers. I place particular emphasis on relationships between the algorithms and statistical features in the datasets that function as hidden variables in a typical experimental setup. In Section 5.3 I show that these hidden variables have substantial impact on results via an evaluation in which I vary algorithms, datasets, and the hidden variables themselves.

In Chapter 6, I scrutinize the popular synthetic Kronecker graph model specifically as it relates to benchmarks and evaluations. In Section 6.2, I review some known problems with the model and discuss how these affect evaluation practices. One of the most blatant issues is the model's discretized degree distribution, and I argue that the current solution based on random noise is unused in practice, because it burdens evaluations with additional problems. In Section 6.3, I present my own fix to the degree distribution, which is based on averaging isomorphic variations of the Kronecker model by randomly permuting its operand order. My fix addresses these evaluation concerns and is also more effective than random noise, which I show in Section 6.5. Finally, in Chapter 7, I conclude with a summary of my work and some thoughts about future work on graph processing systems.

2 Background

We briefly survey some of the history of graph computing and give some context for the problems that we struggle with in our research. Graph computing is currently fashionable in the larger context of “Big Data” computing, especially as it relates to social network analysis. To outside observers this may appear strange, because graph computing never really went out of fashion and has always been an important niche in computer science. Thus, modern data science risks enthusiastically reinventing classic ideas published in SIAM Journals in the 1980s. Conversely, there are a great many software solutions from the 80s and 90s whose techniques truly cannot scale to modern “big” datasets. Recognizing which case is which requires some background and expertise.

“Big” is perhaps the greatest misnomer in all of graph computing; it is dangerously close to communicating nothing at all. A big PageRank computation for search ranking is different in scale and structure from a big subgraph matching query generated by SPARQL. Which is actually more difficult in practice may depend on features of the input data that have little to do with size, such as the presence of distinguishing vertex and edge labels. These problems are different enough that different communities build different system architectures to deal with them, and these architectures have different performance issues. We review some of these concerns with a particular emphasis on vexing issues in modern systems, especially those covered by this dissertation.

2.1 Graphs: Konigsberg to the 90s

Any historical account of graphs traditionally begins with Euler's "Seven Bridges of Konigsberg."[6] Euler proved that it was impossible to walk the city of Konigsberg and cross every bridge exactly once. To me, the most interesting feature of this problem is its spatial construction. Though Euler famously showed that geometry is not needed to solve this problem, the Konigsberg topology is greatly constrained by geometric principles. The bridges do not cross, so the graph is planar. And the bridges do not cross because of their physical and spatial construction.

Graphs with spatial constraints dominated much of 20th century graph computing. Konigsberg is a classic example of a *road network*, a class of spatial graphs associated with many important and practical problems such as reachability, shortest paths, tours, and the maximum flow of traffic. Spatial meshes, such as those used in computer-aided object design (CAD), are similarly constrained by domain-specific physical construction rules. When we decompose a CAD mesh into finite elements to solve physical equations, we are computing on a spatially-embedded mesh.

Just as spatial graphs prompt interesting questions, their constrained topologies admit interesting solutions. Shortest path problems admit faster solutions in planar graphs[53] and in graphs that obey the triangle inequality[44]. Many CAD objects have distinct parts, and thus their meshes admit distinct partitions that respond well to partition-based divide-and-conquer algorithms[114]. Broadly, the relationship between spatially constrained graphs and graph algorithms is governed by the relatively recent graph structure theorem[80], which implies the existence of special algorithms for certain graph families[63]. Though nascent in the 20th century this was nonethless implicitly true and admitted the possibility of many algorithms and heuristics for spatially embedded graphs.

Explicitly spatial graphs are not the only case of a physically constrained graph class. Materials scientists study the interaction between fluids and porous materials by using random graph models to decide e.g., whether one should expect a path through the material. Thus, this percolation theory[23] defines a class of graphs produced by random physical processes. These graphs correspond elegantly with the abstract random graph model pioneered by Paul Erdos and Alfred Reyni[37]. Erdos-Reyni graphs are defined by choosing edges from the set of all possible edges with uniform probability. This simple model is among the most fruitful of Erdos's career; it gives startling results regarding expected connectivity[38], clustering, and even NP-complete properties such as the maximum complete subgraph.

2.2 "Power Law" Graphs: The 00s to the 10s

In 1999, Barabasi and Albert analyzed many "complex networks" and observed that these graphs are dissimilar from the common meshes, etc. seen in other applications[13]. A famous observation is that the Web graph's degree-frequency plot is long-tailed in both directions: there are many vertices with a small degree and a few vertices with a large degree. This distribution is abnormal in physical graphs, because physical constraints, such as locality, contraindicate high-density constructs such as fully-connected subgraphs, which cannot trivially embed in three-dimensional space.Likewise, high degree vertices are vanishingly infrequent in the Erdos-Reyni generative model. Derek de Solla Price had previously made similar observations regarding scientific citations in 1976[101]. The similarity between hypertext links and citations suggests a common phenomenon: both groups proposed nearly identical generative models based on the "preferential attachment" of neighboring vertices in proportion to their degree.

The preferential attachment model ultimately proved too limited in scope and spawned a field of successors seeking to "correctly" model various graphs. Of these models, the most successful have been stochastic block models such as R-MAT[28] and the Stochastic Kronecker Model[75]. Underlying all these models is a common belief that we have encountered important new graph data that is meaningfully different from our old data. This new data has suffered from many misnomers; for consistency with the literature, we will adopt the term "power law graphs." This refers to the relationship between vertex degrees and frequency, which supposedly follows a power law $c^d = f$ where d is a degree, f is its frequency, and c is some graph-specific constant. In practice this "law" is rarely obeyed; both R-MAT and the Kronecker Model are actually log-normal[110]. Other misnomers include "scale-free networks," "skew networks," "complex networks," and "social networks."

The power law graphs typically model generative processes with many hidden variables. Friendship is not literally determined by a preference for people with more friends or a preference to form triangles; these are merely statistical observations of the results of hidden friendship processes. Even the gross physical process of neuron-synapse formation gives rise to a power law network, because synapse formation is precipated by hidden processes in the external sensory environment. The complexity and heterogeneity of these processes is reflected in the complexity of the network structure.

Unfortunately, these heterogeneous features can be adversarial to efficient graph processing. The mere existence of high-degree vertices presents a challenge for load balancing, as there is now a meaningful load distinction between one vertex and another. High-degree vertices are predisposed to forming star-like topologies and, insofar as they prefer to attach to one another, dense cores. In 2001, Tauro et al. visualized this observation as a "jellyfish"[118], which was further formalized by Fan Chung in 2002[31]. Both stars and dense cores are adversarial to divide-and-conquer strategies based on partitioning vertices into balanced sparsely-connected disjoint subsets. This is a painful reversal from three-dimensional CAD meshes with visually obvious partitions.

This is not to say that the power law graphs lack structure. For example, the Internet routing network is clearly organized around a multi-dimensional address structure. Many researchers, such as Boguná et al.[18], have confirmed that this network structure is embeddable and efficiently routable in hyperbolic space. Web URLs are similarly organized by a hierarchical principle; a classic folklore trick to improve locality in Web analysis is to sort URLs by their reverse domain and forward path so that their sort order expresses this hierarchy[15]. There are concrete exploitable structures in power law graphs, but the challenge is to recognize these structures, which are more domain-specific than physical structures.

There is some good work from this period that in our opinion is under-recognized. In 2004 Boldi and Vigna implemented WebGraph[21], an impressive graph computing framework that is still competitive with modern systems for many practical problems. WebGraph computes on a stream of source-sorted edges that are reordered and compressed using novel methods that consciously account for the power law model. For example, edges are expressed as intervals between two vertices in a universal compression code, but the code is tailored to the interval distribution predicted by a power law model. The interval lengths depend on the vertex sort order, and early versions of WebGraph exploited the folklore URL ordering trick described above. WebGraph is noteworthy for its early focus on the graph's structure as it relates to data bandwidth, which is a major issue in modern systems and a running theme throughout this dissertation.

2.3 Vertex Programming: 2010 to the Present

Malewicz et. al's Pregel[85] appeared at the end of a period in which data systems research was dominated by a prior Google system, MapReduce[32]. Like MapReduce, Pregel describes the high-level architecture and user interface of a system intended for distributed processing of large datasets.

However, MapReduce benefited from a great deal of prior art: the map and reduce idioms were well-established, to the point that some criticized the system's novelty[33]. In contrast, Pregel's authors intended to solve a hard open problem: "Despite the ubiquity of large graphs and their commercial importance, we know of no scalable general-purpose system for implementing arbitrary graph algorithms over arbitrary graph representations in a large-scale distributed environment."

Pregel's "vertex-centric" programming paradigm has been reproduced by dozens of systems and thus deserves a brief overview. In a vertex program, the user codes an update function that runs once per iteration for each active vertex in the graph. The update function can access per-vertex local data and per-edge local data from neighboring edges, and can pass messages to neighboring vertices. A vertex is active if it received a message in the previous iteration, and a limited set of global data aggregators are supported. The goal of this model is to decompose a graph algorithm into nearly-independent tasks that may be freely distributed and scheduled. In practice, the model differs somewhat across different systems with different distribution and scheduling idioms.

The absence of any clear reference implementation for Pregel's paradigm spawned an enormous number of competing systems. Apache produced Giraph as a layer on top of their Hadoop implementation of MapReduce, but were soon challenged by Guestrin et al.'s GraphLab[81] and especially its second version, PowerGraph[45]. PowerGraph was especially noteworthy for introducing the "edge partition" distribution model that was expanded on by later systems such as PowerLyra[29]. The authors' argument for this model makes extensive reference to the supposed power law data model governing large graphs of interest. Apache and some authors of GraphLab would later build another system, GraphX, on top of Apache Spark[46]. These systems are collectively discussed in-depth in Chapter 4.

2.4 Sparse Matrices

Another class of graph programming models that have enjoyed revived interest are generalized sparse vector and matrix (SpVM) operations over abstract algebraic semirings. SpVM systems for graph processing, such as Pegasus[59], actually predate Pregel, and computational linear algebra is, of course, a well-established area beyond the scope of this dissertation. However, these models are not as popular as vertex programming for processing graph datasets in industry. This situation may be changing as recent works such as GraphMat[116] have shown that a broad subset of vertex programs can be compiled to SpVM frameworks.

Graphs and sparse matrices may appear superficially different, but in the details of implementation the distinction is not so clear. The overwhelming majority of graph processing systems use compressed sparse row (CSR) and column (CSC) data structures that are virtually identical to their sparse matrix equivalents. Even the venerable edge list format is essentially identical to a sparse coordinate (COO) matrix. Furthermore, many popular graph metrics have clear realizations in the SpVM domain, such as the relationship between PageRank and the principle eigenvector of a stochastic walk matrix[97]. Graph systems mainly differ from SpVM systems in how these metrics are realized as algorithms and implementations. In particular, graph algorithms have a rich tradition of disjoint-set and priority queue data structures, whose realizations in SpVM systems are less obvious.

This graph-matrix equivalence works both ways, and many SpVM systems encounter graph-structured problems in their details of implementation. Notably, numerous sparse factorization algorithms reorder their matrices using a conceptual structure called an *elimination tree*[*] Elimination trees find their roots in Umberto and Francesco's 1972 work on nonserial dynamic programming[14], and were then rediscovered by graph theoretician Rudolf Halin in 1976[49], and again by Robertson and Seymour in 1984[103], and ultimately played a key role in their monumental proof of Wagner's

[*] It is difficult to informally describe elimination trees; see the next Chapter and specifically Section 3.3 for a formal definition, and for perspective from the sparse matrix community see Joseph Liu[78].

Conjecture in 2004[104]. Generally, many optimization problems in sparse matrix representations are reducible to graph problems such as balanced partitioning, bandwidth minimization, and the Traveling Salesman problem. Thus, it is useful and arguably inevitable to think about graphs and sparse martices as one heterogeneous domain.

2.5 Common Problems

Throughout this dissertation we will repeatedly refer to a closely related set of theoretically and empirically hard problems that frequently occur in graph-structured computing. These include graph partitioning and distribution, vertex ordering and its relationship with sparse matrix reordering, and bandwidth minimization. All of these problems admit no easy solutions and are the source of serious design and performance issues in the graph processing domain. We should review these problems before delving into our particular niche.

Graph partitioning problems are one of the best-known classes of NP-hard problems and arise in diverse domains. A general partitioning divides one of the graph's two sets of elements, either vertices or edges, into disjoint subsets while minimizing two functions. The disjoint subsets of one set (e.g., vertices) induce non-disjoint subsets on the other set (e.g., edges) in which some elements are shared between induced subsets and are said to be "cut" (see Figure 2.1). The "cost" function penalizes these cuts, and the "balance" function penalizes size differences between partitions. Without the cost function, the problem reduces to bin-packing, and without the balance function, the problem reduces to min-cut. Together, they form a problem that is, in most cases, difficult to even approximate[7].

Such broad language is necessary because variations of graph partitioning are common and almost all are similarly difficult. The cost of a particular cut may vary with the cut element's "weight," or it may be uniform. Similarly, the partition balance depends on the size of their elements, and

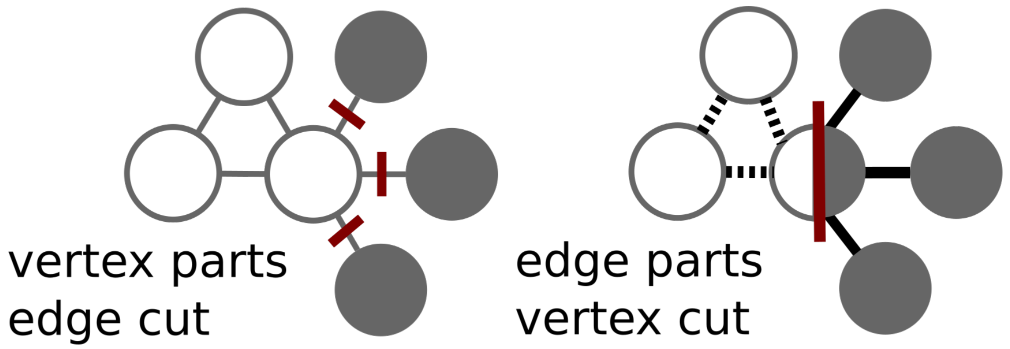


Figure 2.1: Vertex partitions with edge cuts, and edge partitions with vertex cuts, respectively.

the "size" of graph elements may vary with the element or may be uniform. One common variant admits any balanced partitionings within a particular margin of error (e.g., equal plus or minus a percent). Finally, one should note that in a vertex partitioning the cut edges are shared by exactly two partitions, whereas in an edge partitioning the cut vertices are shared by two or more partitions. In the edge-partitioned model, it is common to measure cut costs in proportion to the number of partitions that share a cut vertex, which is usually called the communication volume.

Bandwidth minimization is a closely related problem that replaces the partitions with a sort order and discards the balance function. The cost function is defined in terms of distances in the sort order, e.g., the distance between the endpoints of an edge in a vertex order. This problem is NP-hard even if the goal is just to minimize the longest distance. Naively, bandwidth minimization is hard because the space of orders is exponential, and there is no trick to simplify the search unless $P = NP$. Graph partitioning can be understood similarly, if we think about a k-part partitioning as a bandwidth minimization with a discrete cost function defined by k sort buckets of equal size.

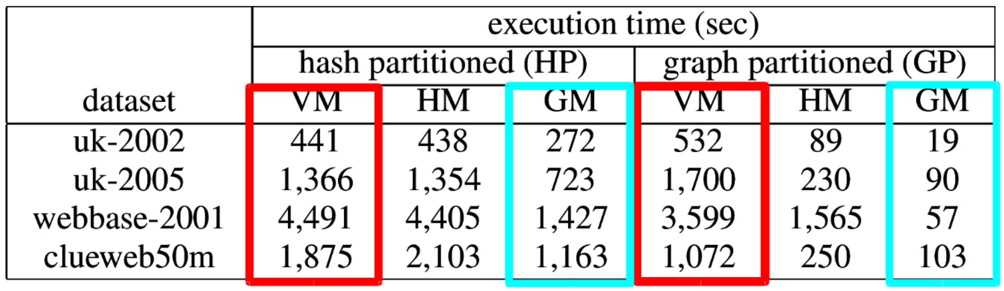
Both of these problems are difficult to avoid in graph processing and have a quantifiable and substantial impact on real graph systems. For example, if we want to distribute a graph computation by sharding the graph data across multiple machines, then the shards are partitions, the machine capacity is the balance function, and any inter-machine data references are cuts. Consequently, it is hard to minimize the number of inter-machine data references and the network traffic. Similarly, suppose the elements on each machine are stored in serial arrays. When we look up a vertex's neighbors, the prefetcher's hit rate depends on whether those neighbors follow the vertex in the array. This serial order and locality-based cost function give a bandwidth minimization problem. Thus, serializing a graph does not optimize your cache hit rate unless you also solve an NP-hard problem.

These examples are performance problems, so any proposed solution must itself run quickly if its goal is to improve end-to-end runtime. This means that practical solutions are domain, scale, and system-specific. Many historical solvers do not target the performance domain or do not improve end-to-end runtime on sufficiently large graphs. For example, METIS[61] is a gold standard partitioner for finite element meshes, but takes hours to run on social networks with billions of edges. Conversely, many modern solvers heuristically target "power law" graphs or are engineered for specific system designs, such as streaming graph partitioners[115]. Ultimately, many systems consider partitioning to be a pre-processing step outside of their scope.

It is normal to see published system results that are throttled by these problems. For example, Figure 2.2 reproduces a connected components benchmark on Giraph++, a system presented in VLDB'14 by Tian et al.[119] HP and GP compare partitions assigned by hashing and by a distributed METIS-like algorithm, respectively. VM, HM, and GM compare different implementations. For the best implementation, GM, the METIS-like partitions give up to a 25*x* performance improvement over hashing, and the paper shows similar results for other benchmarks. *However, these partitions take between* 1082 *and* 6891 *seconds to compute on these graphs.*

Here we see empirically that a large performance improvement is blocked by a graph partitioning problem. Note also that for the worst implementation, VM, the METIS-like partitions are, in some cases, a performance regression over hashing! Thus, these partitions are not generically "good." Rather, the valuation of partitions is problem-specific, as we discussed previously.

The sparse matrix community also struggles with these problems and has developed their own



dataset	execution time (sec)					
	hash partitioned (HP)			graph partitioned (GP)		
	VM	HM	GM	VM	HM	GM
uk-2002	441	438	272	532	89	19
uk-2005	1,366	1,354	723	1,700	230	90
webbase-2001	4,491	4,405	1,427	3,599	1,565	57
clueweb50m	1,875	2,103	1,163	1,072	250	103

Figure 2.2: A connected components benchmark, reproduced from Giraph++ [119], for two partitioning schemes (HP and GP) and three implementations (VM, HM, and GM).

tools to solve them. In general sparse matrices unify these problems under the framework of matrix reordering. A reordering maps the rows and columns of a matrix to new rows and columns in an isomorphic matrix. This new matrix minimizes a cost function, such as the number of cells that become non-zero during a particular algorithm, such as Cholesky decomposition. The problem is solved in the new matrix and then permuted back to the original input. Partitions are easily expressed as diagonal and off-diagonal cost functions; therefore, reordering is also NP-hard.

As we previously mentioned, elimination tree structures are an important matrix reordering tool. But they are also graph concepts and, because reordering and partitioning are closely related, one can adapt them to graph partitioning and bandwidth minimization problems. Doing so neatly simplifies the "sandwich" of matrix and graph theories and produces an elegent partitioning solution for graph systems. Our work on this approach is covered in Chapter 3 of this dissertation.

Most of the problems discussed in this section are "hard" only in the case of general graphs. Of course, concrete graph datasets for a particular application are rarely general. For example, a process might generate a graph dataset in its bandwidth-minimizing order and thus "solve" the cache prefetcher's performance problems. The research community is strongly focused on graph datasets and graph generating models that are large, publicly available, and appear in prior work for comparative analysis. Insofar as these graphs are not general (i.e., are not chosen at random from some general distribution), this has implications for the use of these graphs in evaluations and their generalizability as benchmarks. For example, if a widely referenced graph is distributed in a serial format correlated with its bandwidth-minimizing order, then cache hit rate may be better in these published results than in general practice. We quantify, discuss, and propose solutions to this problem in Chapter 5.

3
A Scalable Partitioner

We present Scalable Host-tree Embeddings for Efficient Partitioning (Sheep), a distributed graph partitioning algorithm capable of handling graphs that far exceed main memory.[*] Sheep produces high quality edge partitions an order of magnitude more quickly than both state of the art offline (e.g., METIS) and streaming partitioners (e.g., Fennel). Sheep's partitions are independent of the input graph's distribution, which means that graph elements can be assigned to processing nodes arbitrarily without affecting the partition quality.

Sheep transforms the input graph into a strictly smaller tree structure via a distributed map-reduce operation using a disjoint-sets data structure. By partitioning this tree, Sheep finds an upper-

[*] This chapter was originally published in VLDB 2015[86]. However, Sections 3.3 and 3.4 have been considderaby expanded to provide more background and context.

bounded communication volume partitioning of the original graph.

We describe the Sheep algorithm and analyze its space-time requirements, partition quality, and intuitive characteristics and limitations. We compare Sheep to contemporary partitioners and show that Sheep creates competitive partitions, scales to larger graphs, and has better runtime.

3.1 Introduction

Graph partitioning is an important problem that affects many graph-structured systems. For example, partitioning quality greatly impacts the performance of distributed graph analysis frameworks[91] such as Giraph[11] and PowerGraph[45]. PowerGraph even integrates a novel streaming partitioner into its loader. These designs have received considerable attention and invited much comparison[108].

METIS[61] is the gold standard for graph partitioning and remains competitive even after 15 years. Though METIS and related work “solve” the small graph partitioning problem, these approaches do not scale to today’s large graphs. Distributed systems, such as PowerGraph, have emerged to address graph-structured problems that exceed the main memory of a single machine, and METIS and similar approaches are unable to partition graphs of this scale in reasonable time and space. Additionally, there is growing interest in partitioning algorithms that minimize metrics other than edge-cut. For example, the minimum communication volume metric[22] has become attractive for the growing classes of graphs and analyses that do not partition well in edge-cut models.

Fundamentally, graph partitioning for distributed computing is a chicken and egg problem. We want to partition large graphs so we can process them at memory speed when they exceed the memory of a single machine. Distribution lets us handle larger graphs and parallelize computation, but it is only efficient when the partitions distribute the data well. Unfortunately, partitioning requires us to solve an NP-hard problem on an out-of-memory graph without an *a priori* well-partitioned data distribution! Streaming graph partitioners[115] and streaming graph analysis systems, such as

GraphChi[72], are recent approaches to this problem.

But streaming graph systems pose two problems. First, they are sensitive to the stream order, which can affect performance (as in triangle counting[9]) or solution quality (as in PowerGraph or the more recent Fennel partitioner[120]). These results are unsurprising, because many graph ordering problems are NP-complete[16] and related to partitioning. The second problem is that streams cannot always take advantage of parallel scaling. Some streaming algorithms are difficult to parallelize (Fennel), while others support multiple streams (PowerGraph). However, if a multi-stream algorithm is sensitive to the input stream partitioning, it is yet another partitioning chicken and egg problem.

We present Sheep, a distributed graph partitioner that embraces the relationship between ordering and partitioning. Given an order or ranking on an undirected graph's vertices, Sheep finds partitions by a method that does not vary with how the input graph is distributed among tasks. Thus, Sheep can arbitrarily divide the input graph to exploit parallelism and fit tasks in memory. Using simple degree ranking, Sheep creates competitive edge partitions an order of magnitude more quickly than both offline and streaming partitioners. As a result, Sheep scales well on large graphs.

Sheep is founded on a synthesis of insights between sparse matrix and complex network theories. Sheep reduces the input graph to a small elimination tree[99], a venerable structure that expresses vertex separators of the input graph (defined in Section 3.3). Sheep then solves a partitioning problem on this tree that translates to an upper-bounded communication volume partitioning on the original graph. This "reduce and partition" technique is similar to METIS, but the theory and details are quite different. In particular, Sheep's tree transformation is a distributed map-reduce operation. This distributed reduction is itself an interesting avenue for future research.

The contributions of this work are:

- A new parallel and distributed partitioning algorithm that addresses the minimum communication volume partitioning problem on undirected graphs.

- A demonstration that Sheep scales well to graphs that exceed single machine memory and is faster than competing algorithms without sacrificing partition quality.

- A distributed elimination tree construction that avoids construction of a chordal graph.

- A novel theory that relates partitioning, complex networks, and sparse matrix theories.

The rest of this Chapter is structured as follows. In the following Section we present background material and related work. In Section 3.3 we present the high-level Sheep algorithm and show how it creates partitions. In Section 3.4 we go into detail on the distributed tree reduction step and give some theory and intuitions governing why Sheep works. We evaluate Sheep in comparison to other partitioners and against itself at various scales in Section 3.5. Section 3.6 addresses the limitations of Sheep and suggests future research.

3.2 Background and Related Work

There are four areas of research on which this work builds: graph partitioning algorithms, graph analytic systems that are impacted by partitioning, sparse matrix theory, and complex network analysis. We address the first three topics below, but we defer complex networks to Section 3.4.3, because its relevance to our algorithm is clearer in context.

3.2.1 Graph Partitioning

METIS[61], which has been a reliable standard for graph partitioning, is a *multi-level* graph partitioner that creates a sequence of "coarsened" graphs where each vertex represents a union of vertices in the previous graph. It computes partitions on the smallest graph and then projects them back to each larger graph in succession, refining the partitions as it does so. METIS can optimize edge cuts or communication volumes, but the solutions discovered in either case are sometimes similar[60].

Multi-level methods are frequently used for graphs with tens or hundreds of millions of elements. However, they consume memory and are an order of magnitude slower than streaming methods, so multi-level partitioners are challenged by the billion-element graphs becoming common today. ParMETIS[62] is a distributed version of METIS, but it suffers from a partitioning chicken and egg problem: each distributed task works on a subgraph and needs to communicate with other tasks in proportion to the edges between subgraphs. So the performance of this method is harmed without some *a priori* partitioning.

The streaming partitioning model[115] was created to address partitioning problems on large scale graphs. In this model each graph element arrives in sequence and must be immediately assigned to a partition. Streaming forbids partition refinement or any global introspection such as spectral analysis; it is also well suited for integration with the data loaders of graph analysis engines. Fennel[120] is a representative work in this area that interpolates between two established heuristics[100,115]. Bourse et al. extend Fennel's method to communication volume partitioning.[22] However, all streaming partitioners are sensitive to the stream order and random orders are pessimal approximations[120]. Thus, it is more difficult to quantify the practical quality of a streaming partitioner; but, in general, streaming partitioners produce worse partitions than METIS, though they operate much more quickly.

Sheep outperforms both METIS and Fennel in runtime, is competitive with METIS in quality for a small number of partitions (i.e., less than 5), and is competitive with Fennel for larger counts.

3.2.2 Partitioning in Graph Analytic Engines

PowerGraph[45] is representative of graph analysis frameworks that use stream partitioning to break a large graph into pieces small enough to run on individual nodes. It uses an edge placement partitioning model, assigning edges to machines and duplicating vertices on multiple nodes as necessary. PowerGraph integrates a novel multi-streaming partitioner into its data loader to minimize duplicates. This design has received much attention but is well known to have a problem with severe partition imbalances[22]. Pregel[85] and Giraph[11] are frameworks that partition vertices instead of edges, whereas GPS[107] and PowerLyra[29] are recent hybrid systems: they partition the edge sets of high-degree vertices but keep the edge sets of low-degree vertices together. Sheep is most effective for edge partitions, and it intuitively produces partitions that exploit this same property as GPS and PowerLyra, although it does so indirectly by a different method; we discuss this further in Section 3.4.4

GraphChi[72] is a single-machine graph analysis system that handles out-of-memory graphs by creating partitions that it processes as parallel streaming working sets. While in principle this system could benefit from well partitioned sets, out of memory partitioning is traditionally addressed by adding more memory; so GraphChi, which targets low-memory systems, does not feature any partitioner at all. X-Stream[106] is a similar streaming analysis system.

3.2.3 Sparse Matrix Theory

Sheep partitions a graph by partitioning a data structure called an *elimination tree*[99], as described in the following Section. Elimination trees are a famous data structure, but Sheep constructs the tree using a novel distributed method. Nested dissection[42] is an alternate method to construct elimination trees in parallel. It works by finding small vertex separators and then recursing into the remaining components of the graph. Because vertex separators are a form of partitioning (Section 3.3.4), this is a chicken and egg problem for partitioning applications. However, there is a sense in which Sheep "reverses" nested dissection by deriving partitions from an elimination tree constructed by other means. Ashcraft and Liu explored a similar idea[10] with an algorithm that extracts separators from an elimination tree and then sorts the separators to find a better tree, although they optimize for different parameters than does Sheep.

3.3 Overview

3.3.1 The Sheep Algorithm

Given an undirected graph, G, we partition it by:

1. Sorting the vertices,
2. Reducing the the graph to an elimination tree[99], T, according to the order in step 1,
3. Partitioning the elimination tree, and then
4. Translating the tree partitions into graph partitions.

Elimination trees are defined in detail in the rest of this Section, as is the partitioning method for T and the translation of partitions from T to G. The vertex sort and tree reduction are discussed in detail in Section 3.4.

3.3.2 Conventions

Graphs

A *graph* $G = (V, E)$ is an arbitrary set of *vertices* V and a subset of *edges* E from $V \times V$. To distinguish the element sets of different graphs, we may refer to G_V and G_E. By convention, $n = |V|$ and $m = |E|$. G is said to be *undirected* when (x, y) in E iff (y, x) in E, in which case we ignore the distinction between (x, y) and (y, x). Else, G is *directed*, and for (x, y), x is called a *source* and y a *target*. In either case we say that x and y are *adjacent* in G.

For the remainder of this dissertation, we assume graphs to be undirected unless stated otherwise; in particular, Sheep partitions undirected graphs. Where explicitly stated, we may sometimes treat a directed graph as if it were undirected by ignoring the distinction between (x, y) and (y, x). Broadly, we prefer this convention because most partitioning cost metrics are *symmetric*, i.e., independent of edge direction. Furthermore, many graph systems and algorithm implementations, such as pull-based PageRank, store and use a directed graph's *transpose*, where (x, y) in the transpose iff (y, x) in G. Thus, such systems effectively store undirected graphs.

We assume that a graph contains no self-edges (x, x) or multi-edges (x, y), (x, y) unless stated otherwise. It is trivial to extend our work to these cases, but they needlessly complicate notation. Given this assumptions, a graph is *complete* and called a *clique* if every possible edge in E exists.

The set of all y in V such that x is the source of some (x, y) in E is called the *out-neighbors* or *outputs* of x in G. Similarly, the set of all y in V such that x is the target of some (y, x) in E is called the *in-neighbors* or *inputs* of x in G. The union of these two sets is called the *neighbors* or *adjacencies* of x in G. For undirected graphs, all three sets are equivalent. The cardinalities of these sets are called the *out-degree*, *in-degree*, and *degree* of x in G, respectively.

A *path* in G is a sequence of edges $(a, b), (b, c) \ldots (y, z)$ in E such that for each sequential pair $(a, b), (b, c)$ the target of (a, b) is the source of (b, c). We say that x is *weakly connected* to y, or that y is *reachable* from x in G, if there exists a path in G whose first source is x and final target is y. If x and y are weakly connected and y and x are weakly connected, we say that x and y are *strongly connected* in G. For undirected graphs we ignore this distinction and say that x and y are *connected* in G. An entire graph is weakly connected if for all x and y in V, either x is weakly connected to y or vice versa. Similarly we may say a graph is strongly connected or just connected.

A *subgraph* of $G = (V, E)$ is a graph $G' = (V', E')$ such that V' is a subset of V and E' is a subset of E. An *induced subgraph* is a subgraph with the additional property that if x and y in V' and (x, y) in E, then (x, y) in E'. A weakly-connected *component* of G is a weakly-connected induced subgraph G' such that no vertex in G' is weakly connected in G to a vertex not in G' (that is, G' is maximal). Similarly we may refer to strongly connected or just connected components.

A set of vertices $S \subset V$ is called a *vertex separator* if the induced subgraph V/S (that is, V without S) has more components than G. Intuitively, there are connected components in G that are disconnected if S is "deleted" from G. We say that S *separates* these components. A separator is *minimal* if it does not contain a smaller separator as a subset. Vertex separators are one way of modeling graph cuts and partitions and will feature heavily in this Chapter.

Trees, DAGs, and Partial Orders

A *cycle* is a path that starts and ends at the same vertex. A cycle is *simple* if it repeats no edge and each vertex in the cycle appears as a source and a target exactly once. We will assume cycles are simple unless stated otherwise. An undirected graph is called a *forest* if it admits no simple cycles and a *tree* if it is a connected forest.

A directed acyclic graph (commonly called a *DAG*) may also be called a tree if it is a tree when treated as undirected. A directed tree is called an *in-tree* if each vertex x has at most one out-neighbor y, and an *out-tree* if each vertex x has at most one in-neighbor y. In either case, x is called a *child* of y, and y is the singular *parent* of x. If a vertex z has no children, then z is called a *leaf*. If z has no parent, then z is called a *root*. All trees have exactly one root. The distinction between in-trees and out-trees is sometimes unimportant, so we will collectively refer to them as *rooted trees*.

A *partial order* $P = (V, \leq)$ is a set V and a binary relation $\leq$ over V such that for all $x, y, z \in V$,

1. $x \leq x$ (reflexivity)
2. If $x \leq y$ and $y \leq x$ then $x = y$ (antisymmetry)
3. If $x \leq y$ and $y \leq z$ then $x \leq z$ (transitivity)

For clarity, we may refer to $\leq_P$ and $<_P$. We say that P is *total* when $x \leq y$ or $y \leq x$ for all x and $y \in P$. In this Chapter the symbol P usually represents a total order over a graph's vertex set V and is therefore a *permutation* of V. A *suborder* of $P = (V, \leq)$ is an order $P' = (V, \leq')$ such that $\leq'$ is a subset of $\leq$. Conversely, we say P is an *extension* of P'. A *linear extension* is an extension that is total.

Because $\leq$ is a binary relation over V, a partial order also defines a directed graph (V, E'). By convention if $x \leq y$ then $(x, y) \in E'$ where x is the source and y is the target. This graph is necessarily a DAG because a cycle would contradict the antisymmetry property. Conversely, the transitive closure (reachability graph) of any DAG is a partial order, so we may say that a DAG "defines a partial order". In particular, an elimination tree (following Section) is a DAG and therefore defines a partial order. This order is not a total order unless the DAG is a linear path graph.

3.3.3 Elimination Trees

Let $G = (V, G_E)$ be an undirected graph and $T = (V, T_E)$ be a directed rooted tree or forest with the same vertex set: that is, for any connected component $T_C \in T$ there is a unique $r \in T_C$ that is reachable from all $x \in T_C$. This implies that T is an in-tree where edges point from leaves towards the root. Then, T is an *elimination tree* of G iff T holds the following property:

Property 1 (Elimination Property). *For each $(x, y) \in G_E$, either x is reachable from y in T or vice-versa. Equivalently, x and y share an ancestor-descendant relation in T, and T defines a partial order where $x < y$ or $y < x$.*

If G is a complete graph (clique) then T must be a linear path graph. T is usually more interesting than a line, but in general T is not a well balanced tree. If G is connected then T must be a single tree, but for graphs with k connected components, T can be a forest with at most k trees. An elimination tree is deeply related to the components and connectivity of its associated graph via the following corollaries:

Corollary 1.1. *Let x and y be children of z in T. Let subt(x) be the set of vertices in the subtree rooted at x in T. Then, there does not exist any edge $(x', y') \in G_E$ between subt(x) and subt(y). This is a well known result*[52].

Corollary 1.2. *Let x and y be children of z in T. Let supr(z) be the set of z and reachable vertices from z in T. If we delete supr(z) from G, then in the resulting graph subt(x) is not reachable from subt(y) and vice-versa.*

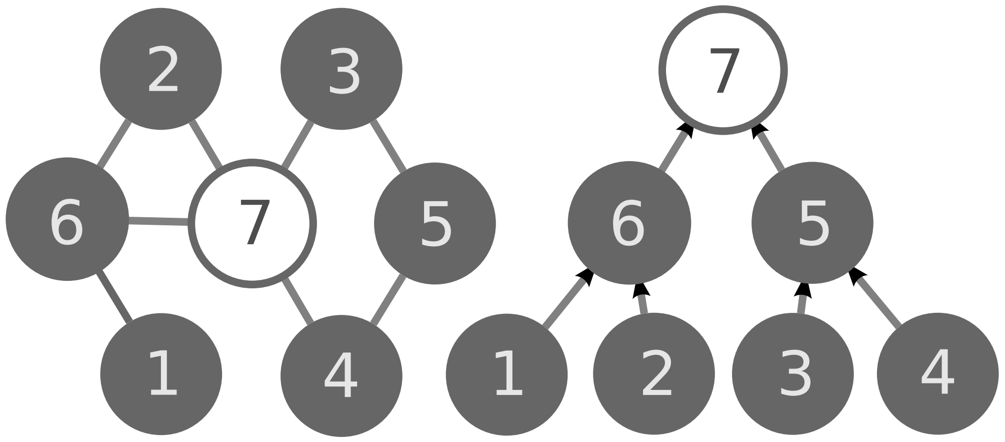


Figure 3.1: Graph G (left) and elimination tree T (right). For $(x, y) \in G_E$, x is reachable from y in T or vice-versa. Note that $supr(6) = \{6, 7\}$ is a vertex separator of 1 and 2, and similarly 7 is a separator of $subt(5)$ and $subt(6)$

Corollary 1.2 is true because any path from $subt(x)$ to $subt(y)$ must contain a vertex in $supr(z)$. Thus, $supr(z)$ is a *vertex separator* of $subt(x)$ and $subt(y)$, because it is a set of vertices whose removal *separates* sets of graph elements. Note that $supr(z)$ is not necessarily minimal, because it may contain vertices whose removal is unnecessary to separate $subt(x)$ and $subt(y)$. However, $supr(z)$ is bounded above by the depth of the elimination tree, which is called its *tree-depth*[17]. Figure 3.1 depicts an example tree; we discuss tree construction in Section 3.4.

3.3.4 Partitioning

An *edge k-partitioning* of G is an assignment $E \to \{1, ...k\}$ to k partition sets such that each edge is assigned to one partition and no partition is larger than $(1 + b)(m/k)$, where b is called the *balance factor*. Similarly, a *vertex k-partitioning* is an assignment $V \to \{1, ...k\}$ where no partition is larger than $(1 + b)(n/k)$. Partition optimizations usually minimize the balance factor and another constraint called the *cost*.

Sheep is a *communication volume*[22] minimizing partitioner, like PowerGraph and some versions of Fennel. Communication volume is a partition cost that counts the unique partitions adjacent to each vertex, minus 1 for normalization so that the cost of a 1-partitioned graph is 0.

$$\sum_{x \in V} |\{partition(element) : element \in adjacent(x)\}| - 1$$

Note that for edge partitions *element* is an edge, but for vertex partitions *element* is a vertex and $adjacent(x)$ is inclusive of x. In general, edge partitionings achieve lower volumes because there are more edges than vertices and therefore more degrees of freedom to assign partitions.

Informally, communication volume counts the number of "duplicate" vertices in some graph-structured systems. For example, PowerGraph partitions edges across machines and then instantiates adjacent vertices on each machine. Each vertex has one duplicate instance for each partition adjacent to it, minus one primary instance. Of course, which instance is primary does not affect the count. If we say the primary instance rests with the highest-ordered partition in some arbitrary order, then we can equivalently express communication volume by a summation over the partitions.

Let K be the set of partitions. Then, an equivalent expression of communication volume is:

$$\sum_{i \in K} |\{x : x \in V, x \in adjacent(i) \cap adjacent(j), i < j \in K\}|$$

That is, each vertex x adjacent to partition i is a duplicate iff x is also adjacent to a higher-ordered partition j; else, the primary instance of x rests with i. For vertex partitions, $adjacent(i)$ is inclusive of all $x \in i$. This duplicate set is trivially a vertex separator of $adjacent(i)$ from all $adjacent(j)$ in the subgraph induced on G by the union of $adjacent(i)$ and all $adjacent(j)$. Therefore, we can model a partitioning as a separator series: each partition in arbitrary order separates its primary vertices from the graph of "remaining" unclaimed vertices. The duplicate vertices are the separators, and the communication volume is the sum of the separators.

An elimination tree also expresses a series of separators that are upper bound by its tree-depth.

Recall that by Corollary 1.2, if x and y are children of z in T, then $supr(z)$ is a vertex separator of $subt(x)$ and $subt(y)$ in G. Furthermore, by recursion $supr(z)$ is a separator of $subt(x)$ and $subt(y')$ for all y' such that y' is a child of $z', z' \in supr(z)$. In effect, $supr(z)$ is a vertex separator of $subt(x)$ from the "remaining" vertices not in $subt(x)$ (see Figure 1).

Using these properties we establish a translation between the elements of T and G, such that a vertex partitioning of T translates to an edge or vertex partitioning of G, with an upper bound on the communication volume given by the tree-depth of T.

3.3.5 Translating Partitionings

First, we model a cut-minimizing vertex partitioning problem on T that will translate to a bounded communication edge partitioning problem on G. Let the partition of each edge $(x, z') \in G_E$ be the partition of $x \in T_V$, where $x \in subt(z')$ by the Elimination Property and x is a child of z in T. It follows that $z' \in supr(z)$, and that the adjacencies of (x, z') and the adjacencies of all (y', z') where $y' \notin subt(x)$ intersect in $supr(z)$, or else $supr(z)$ is not a vertex separator of x and y' and contradicts Corollary 1.2. More generally, the adjacencies of every edge mapped in $subt(x)$ and every edge *not* mapped in $subt(x)$ must intersect in $supr(z)$.

Therefore, let the weight of $x \in T_V$ be $|\{(x, z') \in G_E : x \in subt(z')\}|$, and let the cut cost of $(x, z) \in T_E$ be $|supr(z)|$. These costs decrease from leaf to root. It turns out that weighted tree partitioning is trivial for decreasing edge costs: there is a simple leaf to root dynamic program[69] that, given a decreasing edge-costed tree and a maximum subtree weight, finds a minimum cost partitioning of T into subtrees less than that weight. Each subtree maps to an edge partition in G, and its cut cost $|supr(z)|$ gives an upper bound on the intersection of the adjacencies of that partition and every partition that follows it in T. The sum of these cuts upper bounds the communication volume in G.

If the maximum subtree size is $(1 + b)(m/k)$ this will sometimes produce $k' \geq k$ partitions. To achieve exactly k partitions it may be necessary to bin-pack subtrees into partitions: this is a common feature of balanced tree partitioning algorithms. However, we can always achieve exactly k partitions by relaxing the balance factor. Since bin packing has a constant approximation factor, we know that the amount we may relax the balance is similarly bounded; in practice we achieve partitions with less than 3% imbalance. Packing partitions together can only decrease the communication volume, so this does not affect the correctness of the upper bound, which is at worst $O(k' \times depth(T))$ because $depth(T) \geq |supr(z)|$.

For a vertex partitioning of G, let the partition of each $x \in G_V$ be the partition of $x \in T_V$. However, in this case the cut cost of each edge $(x, z) \in T_E$ must be $|subt(x)| + |supr(z)|$, because the adjacencies in G of vertices $x' \in subt(x)$ and $z' \in supr(z)$ may intersect in $subt(x)$ as well as $supr(z)$. Intuitively, in an edge partitioning we have the freedom to map each edge to the "lower" vertex, and this lets us restrict the partition intersections to $supr(z)$, the set above the lower vertex. However, a vertex partitioning constrains edges to map to both endpoints. We could also construct an edge partitioning that assigns each edge to the higher vertex with a cut cost of $|subt(x)|$, but these costs do not decrease from leaf to root. If the costs do not decrease then we must partition T by some other algorithm. However, since T is small compared to G we could use a powerful tool such as METIS even for large G. For example, the UK Web dataset[20] is 44.8GB as a graph file, but only 841MB as a tree file. For now we focus on edge partitionings and leave this idea for future work.

Sheep produces better edge partitions, because its bounds are tighter in that case. This reflects the greater degree of freedom in edge partitioning problems; in particular, the freedom to divide up the edge sets of higher vertices. However, for both edge and vertex partitions, the cut cost is upper bound by the tree-depth of T, so in both cases Sheep's partitions improve with more shallow trees. Tree-depth minimization is NP-complete[17] but this goal opens up some new approaches. In particular, elimination tree construction is usually modeled as an ordering problem, so this lets us reason about partitioning in terms of the vertex order or ranking that would give an ideal tree. This leads to a novel observation regarding elimination trees and complex network theories (Section 3.4.3).

```
Require: $G$ is an undirected graph $(V, G_E)$
Require: $P$ is a total order $(V, \leq)$
  function Elimination Game($G$, $P$)
      $H \leftarrow G$
      $T \leftarrow (V, \emptyset)$
      for all $x \in V$ in order $P$ do
          $S \leftarrow \{y : (x, y) \in H_E, x <_P y\}$
          $H_E \leftarrow H_E \cup \{(y, z) : y, z \in S\}$
          $T_E \leftarrow T_E \cup (x, \min_P(y \in S))$
```

Algorithm 1: The Elimination Game

3.4 The Tree Construction

We present our tree construction in three parts. First, we review the elimination game, a classic elimination algorithm (Section 3.4.1). We then present our own elimination algorithm and prove that we can distribute it across arbitrary partitions of the graph(Section 3.4.2). Finally, we discuss how vertex orders affect the trees we build, and how we derive good orders from complex networks theory (Section 3.4.3). Finally, we give some intuition for these results (Section 3.4.4).

3.4.1 Elimination Games

The *elimination game* is a classic algorithm[98] that takes an undirected graph and produces an elimination tree; please see Algorithm 1. The following section describes Algorithm 1 in detail.

For each vertex x, we add edges to the graph between all y such that y is a P-greater neighbor of x. The parent of $x \in T$ is the P-minimum over all such neighbors. The Elimination Property that $(x, y) \in G_E$ share an ancestor descendant relationship in T holds because in iteration x either T_E gains (x, y) or some (x, x'), in which case H_E gains (x', y). Then, in iteration x' either T_E gains (x', y), or we continue until T_E gains some (x'', y). Then, $x, x' \ldots x'', y$ is a path from x to y in T. Note that T is not a subtree of G, because T_E gains (x, y) from H_E and H is a supergraph.

H is called a *chordal graph* or elimination graph. Chordal graphs are important generalizations of trees with many interesting properties[52], some of which are inherited by the elimination tree T, which is called a *host tree* of H. In particular, chordal graphs have $O(n)$ minimal separators (recall that $n = |V|$). However, H is an unbounded supergraph of G and, in practice, is often expensive to construct. P is called an *elimination order* of G and a "perfect" elimination order of H.

Our distributed tree construction algorithm requires a special observation about the elimination game. The proof is simple but is delegated to the appendix for brevity:

Theorem 1. *Let $G[V <_P z]$ be the subgraph induced on G by vertices less than z. Then, z is the parent in T of exactly the P-maximum vertices in the disjoint components of $G[V <_P z]$ that z joins together in $G[V \leq_P z]$.*

This gives an elegant characterization of the trees constructed by the elimination game as products of a union-find algorithm. Because union-find algorithms are easy to distribute, this leads to a distributed tree construction.

3.4.2 Distributed Reduction

Let $U = (V, P)$ be a union-find data structure over a set V that chooses as each subset's representative the maximum element in that subset according to a total order $P = (V, \leq)$.

In each outer iteration vertex z adopts the P-maximum vertex y in each disjoint component of $G[V \ <_P \ z]$ that z joins in $G[V \ \leq_P \ z]$. By Theorem 1, these are the same children of z as in the elimination game, so T is an elimination tree. Afterwards z is the new P-maximum representative for this set of components, which are now one and joined through z. We call this a "persistent" union-find because T captures the development of the union-find data structure.

This algorithm uses less space and time than an algorithm that creates an explicit chordal supergraph. Supergraphs are at best $o(n + m)$ but usually much worse; conversely, the union-find U and

```
Require: $G$ is an undirected graph $(V, G_E)$
Require: $P$ is a total order $(V, \leq)$
  function PERSISTENT UNION FIND($G, P$)
      $U \leftarrow (V, P)$
      $T \leftarrow (V, \emptyset)$
      for all $z \in V$ in order $P$ do
          for all $(x, z) \in G_E, x <_p z$ do
              $y \leftarrow U.find(x)$
              if $y \neq z$ then
                  $U.union(y, z)$
                  $T_E \leftarrow T_E \cup (y, z)$
      return $T$
```

Algorithm 2: The persistent union-find algorithm

tree T use $O(n)$ space and $O(n + a(n)m)$ time, where $a()$ is the near-constant inverse Ackermann function[40]. However, many elimination algorithms dynamically order themselves by inspecting the chordal graph, so this method is not clearly practical without a good order *a priori*.

The observation that this union-find algorithm can be efficiently distributed is at the core of Sheep. We prove that:

Theorem 2. *Let G_1 and G_2 be two subgraphs of G such that $G_1 \cup G_2 = G$. Let $t(G, P)$ be the elimination tree produced by union-find on G in order P. Then,*

$$t(t(G_1, P) \cup t(G_2, P), P) = t(G, P)$$

Note that though $t(G_1, P) \cup t(G_2, P)$ is a directed graph, it is interpreted as undirected when input to the union-find algorithm. This proof is given in the appendix.

We emphasize this creates the same exact tree as $t(G, P)$. By this theorem we can split G into any number of subgraphs, construct trees independently for each, and then union and reduce the intermediate trees in parallel to create a final tree for G. The result is insensitive to how the graph is split

```
function REDUCE TO TREE($G$, $P$)
    $G_1, G_2 \ldots G_w \leftarrow Split(G)$
    $T_1, T_2 \ldots T_w \leftarrow \{Mapper(G', P) : G' \in G_1, G_2 \ldots G_w\}$
    $T \leftarrow Reducer()$ over $T_1, T_2 \ldots T_w$ and fixed $P$
function MAPPER($G'$, $P$)
    return $PersistentUnionFind(G', P)$
function REDUCER($T_L$, $T_R$, $P$)
    $U \leftarrow Undirected(T_L \cup T_R)$
    return $PersistentUnionFind(U, P)$
```

Algorithm 3: Distributed reduction using persistent union-find.

and the space-time requirements remain nearly linear for each subgraph. This lets us reduce large graphs quickly. An implementation using the map-reduce idiom might look like:

Let $G_1, G_2 \ldots G_w$ be a set of subgraphs of G such that $G_1 \cup G_2 \cup \ldots G_w = G$. Let n' and m' be the maximum numbers of vertices and edges respectively in any such G_w. Then, the parallel runtime of this distributed tree construction is:

$$O(n' + a(n')m' + log(w)(n + a(n)n))$$

where the log expression is the reduce operation over intermediate trees. Because the log expression is essentially $O(n)$ for fixed w, and because m typically dominates n, it is more important to balance $m' = (m/w)$ than n'. This is achieved by evenly splitting an edge list of G_E. If we also want to balance n' we may do so by randomizing the list. In either case, this avoids the partitioning chicken and egg problem.

3.4.3 Ordering Vertices

Trees created by elimination algorithms are the result of a graph G in an order P. So far we have held P constant, but in real applications the graph is constant and the order can vary. Thus, the tree and the vertex separators it expresses are entirely determined by P, hence our earlier statement that Sheep embraces the relationship between ordering and partitioning: Sheep's partitions result from the separators expressed by an elimination order. Formally, if S is a minimal separator of components C_1 and C_2 in G, then any P such that $\forall x \in C_1 \cup C_2, \forall y \in S, x <_P y$ will express S in T.

Order sensitivity is a challenge faced by all streaming graph partitioners. Tsourakakis et al. proved that not only must every streaming partitioner have adversarial orders, but also that random input orders are approximately adversarial[120]. Therefore, for arbitrary input orders one cannot make guarantees as to the quality of streaming partitioning results. Sheep is subject to this if P is arbitrary.

However, Sheep accommodates its input order in its underlying theory, so we can define what is meant by a good order. As discussed in Section 3.3, tree-depth upper bounds the separators expressed by the elimination tree; therefore, a minimum tree-depth order will produce smaller bounds and better communication volumes. Unsurprisingly, tree-depth minimization is NP-complete. There are many depth heuristics in the literature, but, in general, these inspect the graph or chordal graph. We need a compatible heuristic for our distributed construction, since it cannot easily inspect the total graph and does not create a chordal graph.

We found a valuable resource in the complex networks community. Albert et al. pioneered the empirical use of attack plots that, for a given vertex order on the x-axis, plot the size of the largest remaining connected component when one deletes vertices in that *attack order*[5]. The purpose of these plots is to show how different networks dissassemble under different attack orders and to find orders that fully dissassemble networks in a minimum number of attacks. Figure 3.2 is an attack plot of several metrics by Iyer et al.[57]

Consider a walk on T from root to leaves. At each vertex z in the walk, the subtrees of the children of $z \in T$ represent components in the "remaining graph" $G[V <_p z]$: it is like we delete $supr(z)$ and examine the remaining components. The subtree with the most depth is the component that requires the most steps to fully disassemble. So, disassembling the graph in a minimum number of

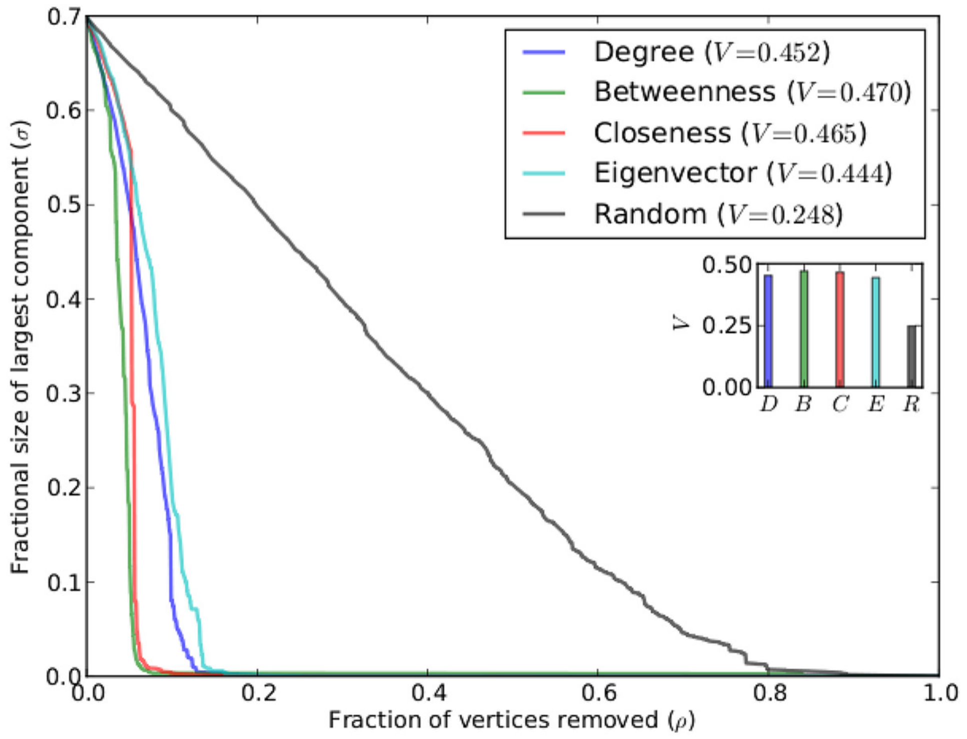


Figure 3.2: Attack plot on a High Energy Physics co-authorship graph, reproduced from Iyer et al.[57] Each line depicts the decaying size of the largest connected component as vertices are removed in a specific rank order.

attacks is exactly the same goal as minimizing its elimination tree-depth. Attack orders and elimination orders are simply opposite orders.

Complex network research repeatedly observes[5 57] that many natural networks, and, in particular, networks with skew degree distributions, are vulnerable to degree-ordered attacks. More sophisticated attack orders use centrality measures such as betweenness[57]. Intuitively, one can characterize degree order as a "greedy edge attack" and betweenness as a "greedy shortest path attack."

Elimination algorithms sometimes use a similar degree heuristic[43] , although this is usually applied online to the chordal graph to optimize related parameters called tree-width and matrix fill-in for matrix factorization. Tree-width is a strictly tighter separator bound than tree-depth[52], but because the chordal graph is a supergraph of the input graph, these parameters do not lead to graph size reductions. For matrix factorization this is not a concern, because factorization requires instantiation of the chordal graph (or "fill-in graph"). This research area arises from the application of graph theory to graphical models of matrices; Sheep reapplies some of these theories to the partitioning problems faced by large scale graph analysis frameworks.

It is well known that degree orders are sometimes "good" for elimination trees. However, so far as we know, it is a novel observation that complex networks research gives empirical characterizations of both the classes of graphs on which degree orders are tree-depth minimizing, and the orders that outperform degree orders on these graphs. By default, Sheep assumes a degree elimination order for input graphs; even in distributed graphs this is easy to compute by broadcasting local degree vectors. Note that Sheep *need not* physically sort the graph in degree order; it merely uses the order logically in its tree construction algorithm.

Our results show that degree orders on skew networks produce low cost partitions that are competitive with other partitioners; this method works extremely well for bipartitioning and often outperforms METIS. We also show that Sheep is improved when better rankings are available, e.g., across repeated analytics runs. This ability to improve the graph's data organization with purely analytic results may be an interesting technique for graph database cracking[56].

3.4.4 Intuition

Degree sorting is also a classic heuristic optimization for triangle counting algorithms, in part because it improves reference locality in networks with skew degree distributions. The many low-degree vertices tend to reference the few high-degree vertices, so clustering the high-degree vertices improves reference locality. However, this heuristic is topologically naive. In particular, many low-degree vertices are not clustered with their adjacencies. For example, if a 2-degree vertex is adjacent to another 2-degree vertex, they may not be clustered in the sort order even though this constraint may be easily fulfilled.

Intuitively, Sheep exploits a partially ordered tree that is more informative than its linear input order. Unlike a total order, the tree expresses *antichains*: sets of independent elements in the underlying graph. Independence is notably present in sparse topologies such as the low-degree vertices of a skew network. Shallow trees generally exhibit more antichains and a tighter bound on the set of vertices that other vertices may reference. Sheep clusters related elements better than a naive sorting heuristic, because it does *not* cluster *unrelated* elements in common cases where a sorting heuristic would. In dense graphs, Sheep devolves to a sort.

One consequence of the above is that for edge partitions, Sheep usually divides the edges of high-degree vertices and keeps the edges of low-degree vertices together. This is because Sheep maps $(x, y) \in G_E$ to the lower vertex $x \in subt(y)$ in T, which is also the lower-degree vertex if P is a degree order. Therefore, the edges of the high degree vertices are spread across T, whereas the edges of the low degree vertices are concentrated in the periphery and leaves of T. Due to the clustering property described above the peripherals are usually independent and well-localized. This result is intuitively similar to the high-degree vertex partitioning methods used by GPS[107] and PowerLyra[29].

3.5 Evaluation

We evaluate Sheep to demonstrate the following claims:

- Sheep scales in the following ways:
  - parallel processing on one machine (Section 3.5.2),
  - out-of-memory processing on one machine (Section 3.5.2), and
  - parallel processing in a distributed environment (Section 3.5.3).
- Sheep is faster than other partitioners on large graphs (Section 3.5.4).
- Sheep's partitions are competitive and are improved by better vertex orders (Section 3.5.5).

We compare Sheep to results published in KDD'14[22], which evaluated METIS, PowerGraph, and both vertex and edge streaming Fennel on the edge balanced minimium communication volume partitioning problem. Fennel[120] is a good representative for streaming partitioners, because it is a simple per-vertex or per-edge loop that considers each partition for each element and chooses the partition that minimizes a special cost function. This design is typical of streaming partitioners[100 115]. We contacted the authors to ensure that our results can be accurately compared. Fennel is called "IC" in Bourse's results, but we have confirmed this is a modification of Fennel to optimize communication volumes instead of edge cuts.

In addition to these graphs, we added a few others to cover interesting cases. The Twitter and UK Web graphs are popular billion-edge networks for graph systems evaluations, and the High Energy Physics coauthorship network is a well known complex networks dataset. We obtained most graphs through the Stanford Large Network Dataset Collection[77]. Table 3.1 summarizes these graphs.

3.5.1 Implementation and Setup

We implemented Sheep in C++ using LLAMA[83], an open-source graph storage library. LLAMA is based on the venerable compressed sparse row (CSR) representation, but allows mutability, and, for read-only algorithms like Sheep, it adds little overhead relative to conventional CSR implementations. It is not distributed, so for distributed tasks we simply open LLAMA subgraphs in different

name	$n = \|V\|$	$m = \|E\|$	file size	reason
HEphysics[95]	7,610	15,751	189KB	fig. 3.13
com-youtube	1,135k	2,988k	36MB	bourse
cit-patents	3,775k	16,519k	198MB	bourse
com-liveJ	3,998k	34,681k	416MB	bourse
soc-liveJ	4,848k	68,735k	828MB	fig. 3.3
com-orkut	3,072k	117m	1.4GB	bourse
twitter_rv	42m	1,468m	17.6GB	scale
uk_2007_05[20]	106m	3,739m	44.9GB	scale

Table 3.1: Graph datasets used in this evaluation.

processes. Sheep uses the MPI map-reduce library both for distributed sorting and for the tree reduction operation described in Section 3.4.2. However, Sheep can fall back on a parallel filesystem for inter-process communication if MPI is not available.

Sheep uses CSR for most variable-length data structures to reduce allocator overhead. However, a tree can be serialized as an array of parent pointers, so this representation is preferred for inter-process communication. As with many iterative graph algorithms, the inner loop of Sheep is performance sensitive, so it was important to optimize our union-find implementation and to use vertex isomorphisms between data structures for fast comparison in the order P.

Typically algorithms that process an out-of-memory graph in a given order should first sort and serialize the graph in that order. However, for Sheep it is more efficient to divide the graph into in-memory working sets and then process each subgraph as-is. Because each intermediate tree is a partial suborder of the total order P, and because these trees are merged and reduced in order P, in a sense Sheep implicitly conducts its own external merge sort.

For single machine experiments, we use a 6-core Intel i7-970 at 3.20GHz with 12GB of RAM and a Samsung 840 Pro SSD. For distributed processing, we use a cluster of Dell PowerEdge M915 servers; each has 64 AMD Abu Dhabi cores at 2.30GHZ with 256GB of RAMand 41.25GBps Infiniband. The local disk is not measured, because all our cluster benchmarks are hot cached. Graph input files are binary 96-bit edge lists, as in the Graph500 benchmark[92], but none of the graphs in this study exceed four billion (32-bit) vertices. LLAMA itself is a 64-bit CSR system. In distributed experiments we first copy the graph to local storage on each node, but we do not include the copy time because it is not a feature of the algorithms and may vary greatly between data center architectures. We do however include graph file ingest times, because distributed ingest is an important feature of distributed algorithms and is measured in Graph500.

Bourse et al. did not evaluate runtime, so we measured timings for several competing partitioners. There is no public Fennel implementation, but it is a simple algorithm, so we implemented the versions in Bourse's study. We implemented both edge streaming Fennel and vertex streaming Fennel with an initial CSR ingest. Despite the ingest time, Fennel's vertex streaming is an order of magnitude faster than its edge streaming, because the critical partitioning work is $O(kn + m)$ rather than $O(n + km)$, where k is the partition count. Because Bourse gives quality results for both versions of Fennel, we evaluated vertex Fennel to privilege Fennel's timing results. Vertex-streaming Fennel would be even faster with a pre-sorted vertex adjacency list, but so would Sheep, and we want to use the Graph500 standard input format.

We ran METIS with default settings. By default METIS is an edge cut partitioner, but it accepts a communication volume minimizing goal with some time overhead. Whether this actually improves the volume varies with the graph[60]. We timed METIS without this option because it slows down METIS and does not always improve its quality, and our quality results come from Bourse. METIS requires that adjacency lists have dense vertex IDs (i.e., $n = max(id)$), so we privileged METIS further by providing it this format.

The PowerGraph partitioner is deeply integrated with the PowerGraph bootstrap process, so it would not be fair to take timing results directly from PowerGraph. We do, however, include quality results for PowerGraph from Bourse.

3.5.2 Single-machine Scaling

Figure 3.3 plots the runtime of Sheep on our commodity machine for a variety of input graphs for one and six parallel workers. As expected, single-worker Sheep's runtime is linear in the size of the graph and additional workers speed it up. The speed up is relatively poor for small graphs, but it improves with larger graphs. We will see this trend in the cluster setting as well. This is a common pattern in parallel algorithms, but the cause here is especially interesting.

Figure 3.4 shows a detailed breakdown of runtime versus the number of workers for Orkut, the largest of the graphs shown in Figure 3.3. The load time represents the time to ingest into the in-

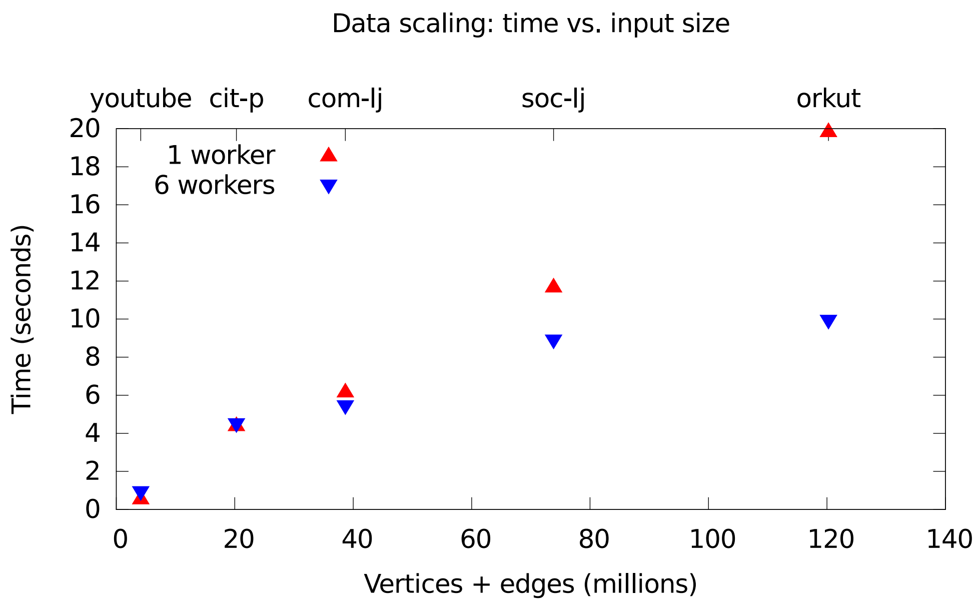


Figure 3.3: Sheep runtime on our commodity machine as a function of graph scale, expressed as the sum of the number of vertices and edges. We show results for both 1 and 6 workers on a variety of graphs.

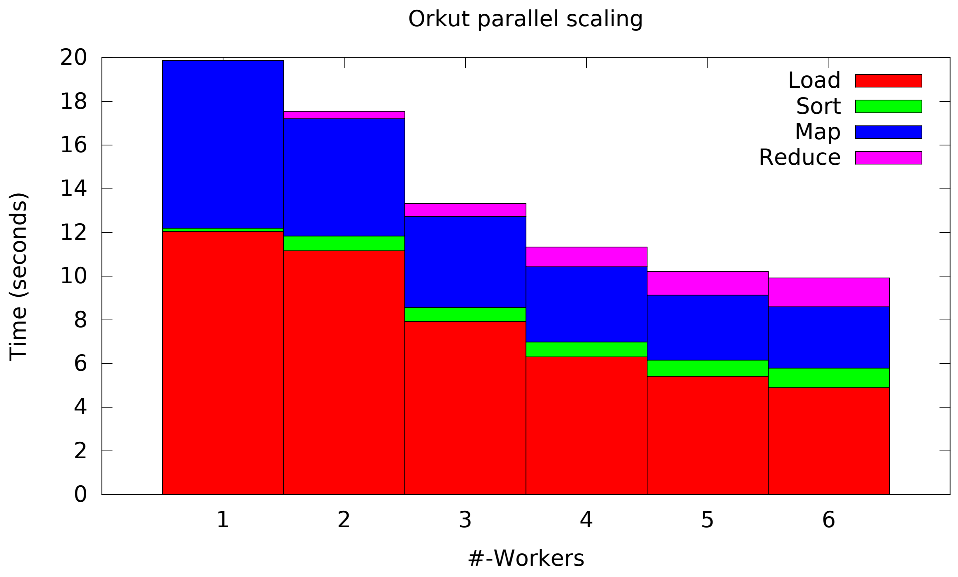


Figure 3.4: Parallel time scaling for Sheep on the Orkut data set and our commodity machine. Tree partitioning takes less than 200ms and is omitted.

memory CSR, the sort time measures computing the global degree order, the "map" time measures construction of the intermediate tree from the initial subgraphs, and the reduce time measures the distributed reduce that combines the intermediate trees into a final tree. The actual tree partitioning step takes less than 200ms and is not distributed, so we ellide it for visibility. We observe that, while the sort and reduce costs are not insignificant, scaling is limited because the load and reduce times do not scale linearly in the number of workers. This is surprising, because each worker processes m/k edges, and the algorithm is near-linear.

The cause of this limitation is imbalance in the underlying graph structure. If we divide a graph edge list into w random parts, then in expectation, each part contains m/w edges but some n' vertices, where n' is a function of the degree distribution and is generally greater than n/w. This is not an implementation detail but rather a fundamental property of distributed graph algorithms that divide the graph into subgraphs. Additionally, there is no guarantee that the input edge list is randomly distributed, and in fact, processes that produce edge lists typically exhibit locality. Therefore, there is some skew in the number of vertices represented in each subgraph, although this effect is less significant. This is not an instance of the partitioning chicken and egg problem – random hashing solves this. However, because this problem is specific to certain graphs and diminishes with scale, it is generally not worth randomizing the input.

When graphs exceed the memory of a single machine, Sheep scales by dividing the graph into memory-sized working sets. For example, the undirected Twitter graph is approximately 23GB in CSR, and therefore 1.9x the memory of our commodity machine. If we break the graph into 10 parts we can run 2 tree constructions simultaneously in memory and partition Twitter in just 7.5 minutes. Compare this result to our in-memory Orkut results. The Twitter graph is approximately 12.5 times the size of the Orkut graph, and 7.5 minutes is approximately 25x the runtime of Orkut with 2 workers, producing a factor of 2x overhead. Since Twitter is out of memory and Orkut is hot cached, this overhead is entirely expected and seems reasonable.

Twitter fits in memory on our 256GB cluster machines. Figure 3.5 plots the runtime of Sheep on Twitter versus the number of parallel workers on a single cluster machine. Twitter is large enough that we see serious parallel scaling in both our load and map times, such that the distributed reduction becomes the limiting factor, as predicted by the complexity equation in Section 3.4.2. In fact, for graphs as large as Twitter our ingest scales better than $1/k$, because CSR ingest requires a partial edge sort. Using 18 cores we load and partition this graph in just 2.8 minutes. In comparison, Fennel takes over 20 minutes, and METIS takes hours.

3.5.3 Distributed Scaling

Of course, for out of core graphs we are inevitably interested in distributed scaling. Figure 3.6 plots time versus increasing workers and nodes for Sheep on the 3.7 billion edge UK Web dataset[20]. While our single node time is quite reasonable (less than 8 minutes), adding more cores does not produce linear scaling. The ingest step does not reliably improve, and the reduction step is surprisingly expensive. The underlying cause of this is that we are badly thrashing the various caches of the 256GB NUMA node; the reduce, in particular, involves many inter-process buffer copies.

As Sheep is distributed, it can scale to more machines to relieve memory pressure. The addition of just one machine improves the runtime even when the total core count is held constant. The x-axis labels of *CxM* signify *C* cores on *M* machines. 3x2 cores is significantly faster than 6x1 cores because, unsurprisingly, the data ingest is faster. With 12x2 cores we get almost twice the performance of 24x1 cores, not only because ingest is faster, but also because we relieve the memory burden of buffer copying in the reduce step. Note also that 6x4 cores introduces no overhead over 12x2, so in our data center it is practical to simply ignore single-machine scaling and instead launch Sheep horizontally across available machines.

Sheep, like any distributed algorithm, has counter-scaling costs that eventually cause its performance to approach an asymptote. The eventual indivisibility of the graph and the growing cost of

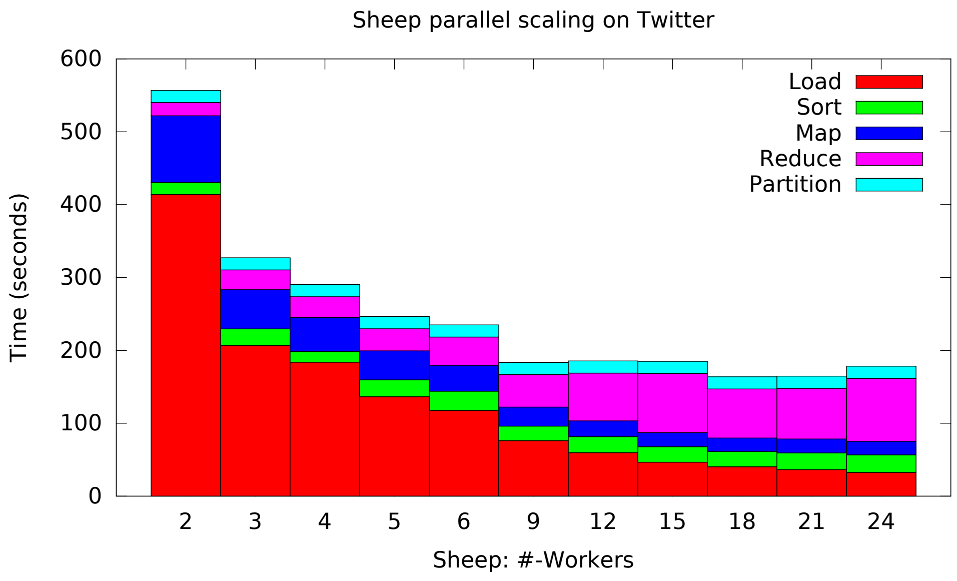


Figure 3.5: Parallel time scaling for Sheep on the Twitter data set on a single cluster node.

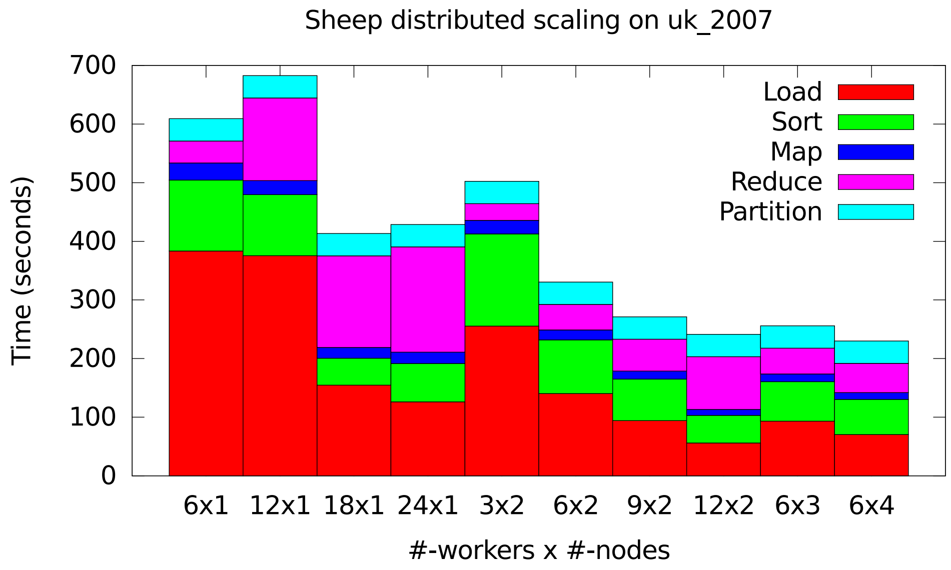


Figure 3.6: Time versus the workers and machines for Sheep on the UK Web dataset and multiple cluster nodes. 6x4 signifies 6 cores on 4 nodes, i.e. 24 cores. Sheep is data bound, so adding extra memory controllers improves its runtime at scale.

the reduce tree must eventually prevent Sheep from leveraging additional resources. It is also clear that optimal performance requires some parameter tweaking with respect to the graph size and features of the data center architecture. However, even with suboptimal parameters Sheep is extremely fast relative to other partitioners.

3.5.4 Comparative Time

Next we compare Sheep to other partitioning algorithms. Figure 3.7 returns to Orkut on our commodity machine and plots runtimes for the various partitioning algorithms as a function of different numbers of partitions. First, note that Sheep is invariant to the number of partitions; not only is our tree partitioning step invariant to the partition count, but the partitioning is a trivial fraction of the runtime. Other partitioners evaluate multiple partitions for each element, so their times scale with the number of partitions. Nevertheless, at the scale of the Orkut graph, Fennel is a reasonable competitor for Sheep, while METIS is comparatively slow.

We now reexamine Figure 3.5, which plots time for Sheep on Twitter versus the number of parallel workers on a single cluster machine. Compare to Figure 3.8, which plots time for Fennel on Twitter versus the partition count on a single cluster machine. Sheep is invariant to the partition count, and Fennel is not a distributed algorithm. We measure our own implementation of Fennel, but Tsourakakis et al. report a similar Twitter time of 40 minutes[120]. We were unable to get METIS to partition in-memory on our 256GB machines, but using 1TB, Tsourakakis reports 8.5 hours.

We observe that Sheep is many times faster than Fennel, primarily, but by no means entirely, because of rapid data ingest. But even if we exclude the ingest, Sheep is 2.4x faster than Fennel at 30 partitions, because Sheep is insensitive to the partition count. This ingest-free comparison is quite unfair to Sheep, because rapid ingest is a major advantage of a fully distributed partitioner. Note that our Fennel implementation's ingest supports multithreading, so this advantage is not simply due to parallelism. Fundamentally, it is cheaper to build several small CSRs than one large one.

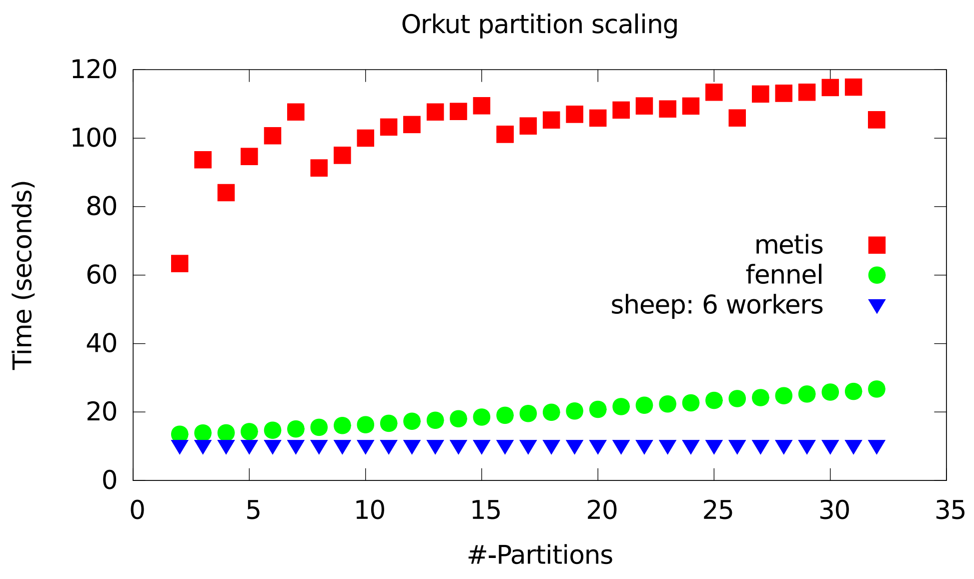


Figure 3.7: Runtime versus # of partitions on Orkut. Sheep is insensitive to the partition count, because Sheep spends less than 200ms on the partitioning step.

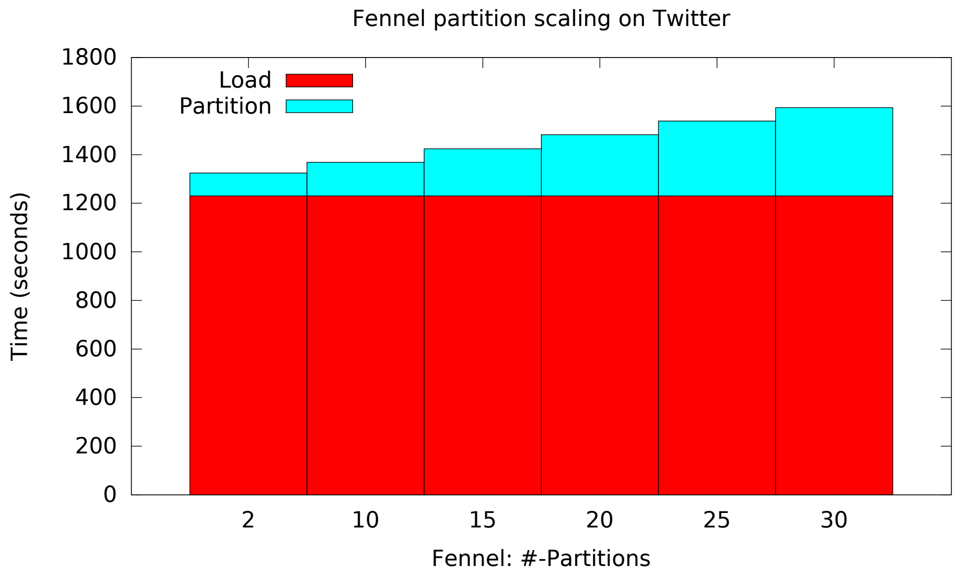


Figure 3.8: Time versus the partition count for Fennel on the Twitter dataset on a single cluster node. Note that the y-axis is 3x taller than Figure 3.5.

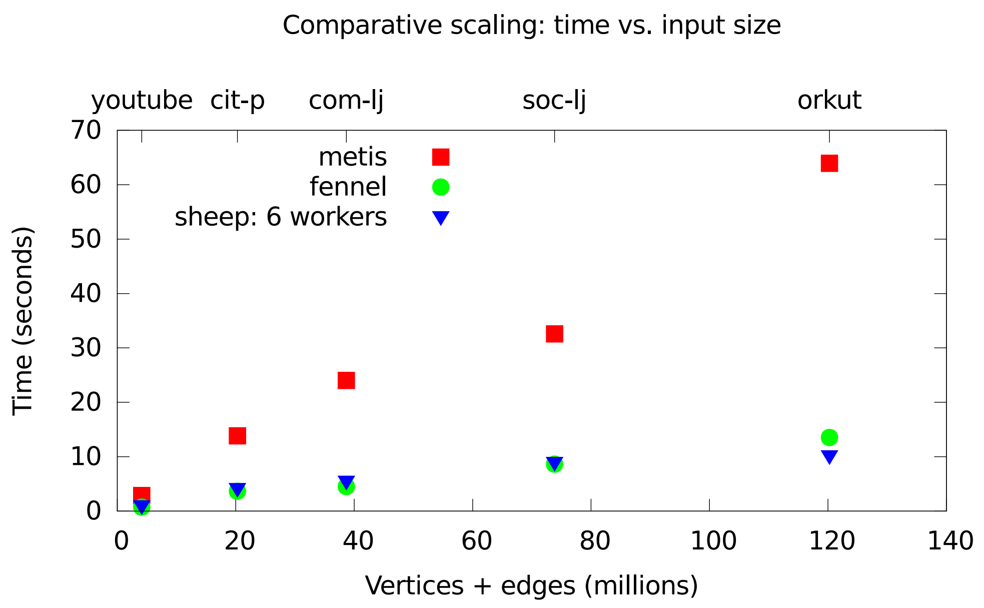


Figure 3.9: Partitioner runtime as a function of graph scale for 2-partitioning with various algorithms.

Figure 3.9 plots the runtimes of various 2-partitioning algorithms at different input scales; this is the least favorable comparison for Sheep. For smaller graphs Sheep and Fennel run in essentially the same time, suggesting that both algorithms are data-bound. We see the same result when 2-partitioning Twitter if we discount ingest times, but at no point does Fennel significantly outperform Sheep. However, Fennel is harmed by growing partition counts, and on larger graphs, Sheep benefits from rapid parallel ingest.

On small graphs, where METIS is a viable option, the partition quality of METIS and similar multi-level techniques are on average much better than other methods. Since METIS is almost two decades old, we think that small-graph partitioning is not currently an interesting problem. We created Sheep to target larger graphs, and on these graphs, Sheep is much faster than other partitioners.

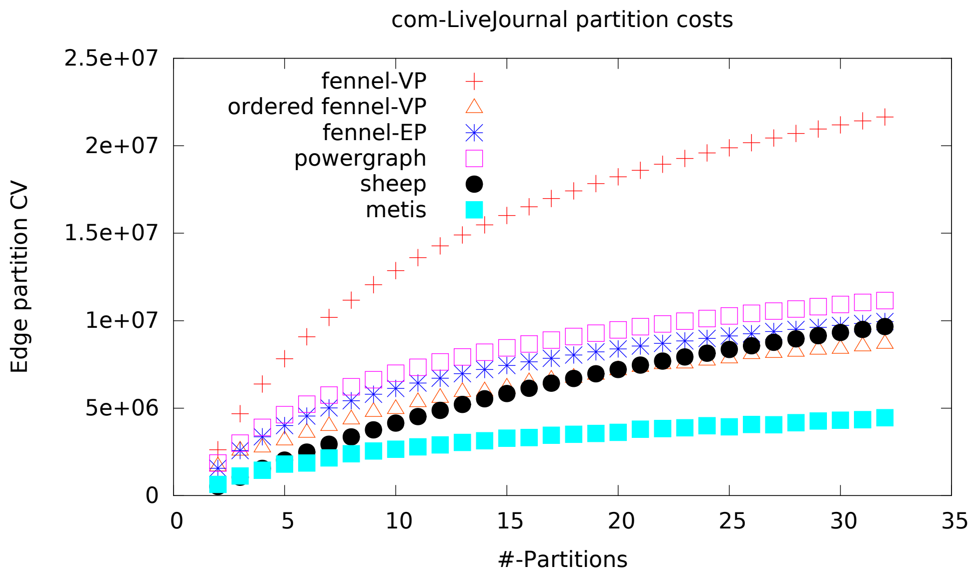


Figure 3.10: Edge-partitioned communication volumes (ECV) versus # partitions on com-LiveJournal. ECV may be thought of as a count of duplicate vertices.

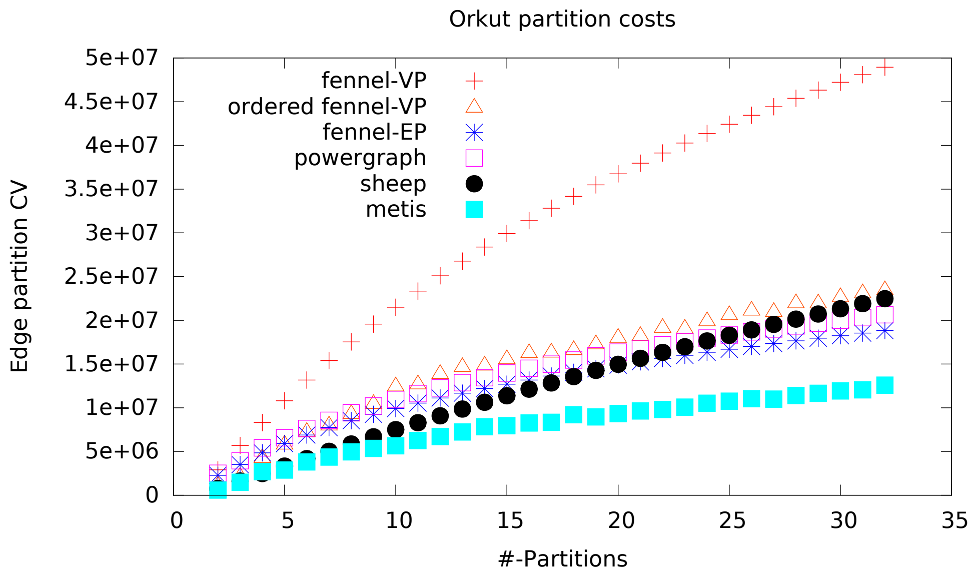


Figure 3.11: Edge-partitioned communication volumes (ECV) versus # partitions on Orkut.

3.5.5 Partition Quality

Figures 3.10 and 3.11 add Sheep results to partition quality results obtained from Bourse et al. The y-axes are edge partitioned communication volumes as defined in Section 3.3.4. In Bourse's plots the y-axes are normalized by m, but this compresses the curves and makes it difficult to compare partitioners within each plot. We do not reproduce balance factors from Bourse, because Sheep and every other partitioner except PowerGraph achieve balance factors of less than 3%, which is the default in METIS. PowerGraph's balance is unbounded, and Bourse reports factors as high as 50% on Orkut and over 200% on a Youtube dataset.

Sheep is clearly competitive with other partitioners. It is slightly better than even METIS on bipartitioning problems, competitive with METIS up to 5 partitions, significantly better than other streaming partitioners up to 10, and slightly worse than some partitioners at more than 20. Taken together with our timing results, this shows that Sheep achieves its goal of producing competitive partitions with significantly improved runtimes and scalability.

Fennel and Powergraph are both sensitive to the vertex input order, and the ideal orders for the two are not necessarily the same. Bourse et al. address this issue by using a randomized input order. However, this penalizes Fennel, as Tsourakakis et al. showed that a random order is approximately pessimal for Fennel[120]. To provide a more meaningful comparison, we include results from our own Fennel implementation in the graph's "natural" vertex ID order. While this may not be an optimal input order for Fennel (the optimal order is unknown), natural vertex orders are often correlated with some stochastic process such as a walk, and Tsourakakis argues that this often makes them appropriate for streaming. We tested Fennel with degree and reverse degree orders, but these gave worse results. Our Fennel implementation performs similarly to implementations reported elsewhere. It produces vertex partitions that are then transformed to edge partitions by Bourse's degree weight and random assignment method[22]. It may be possible to derive better partitions from

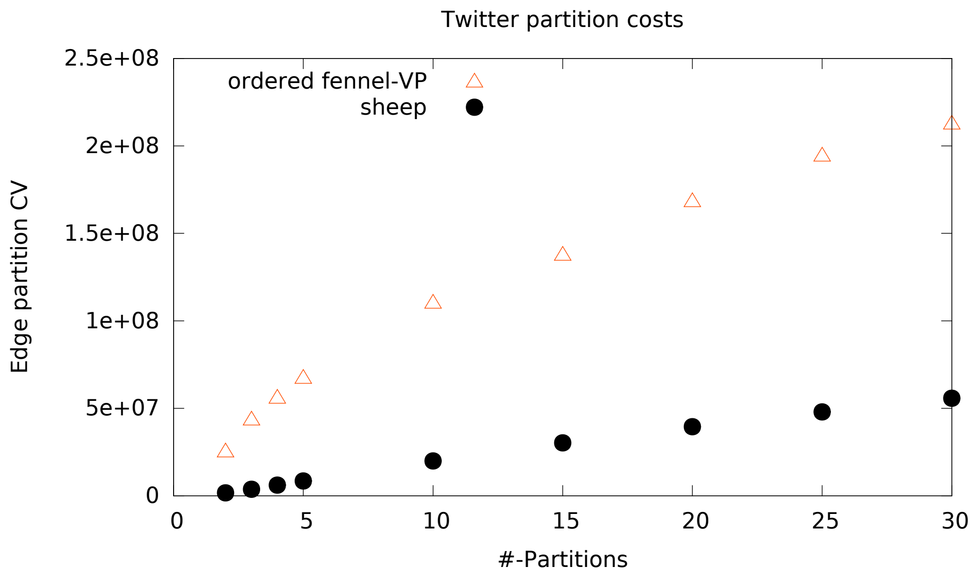
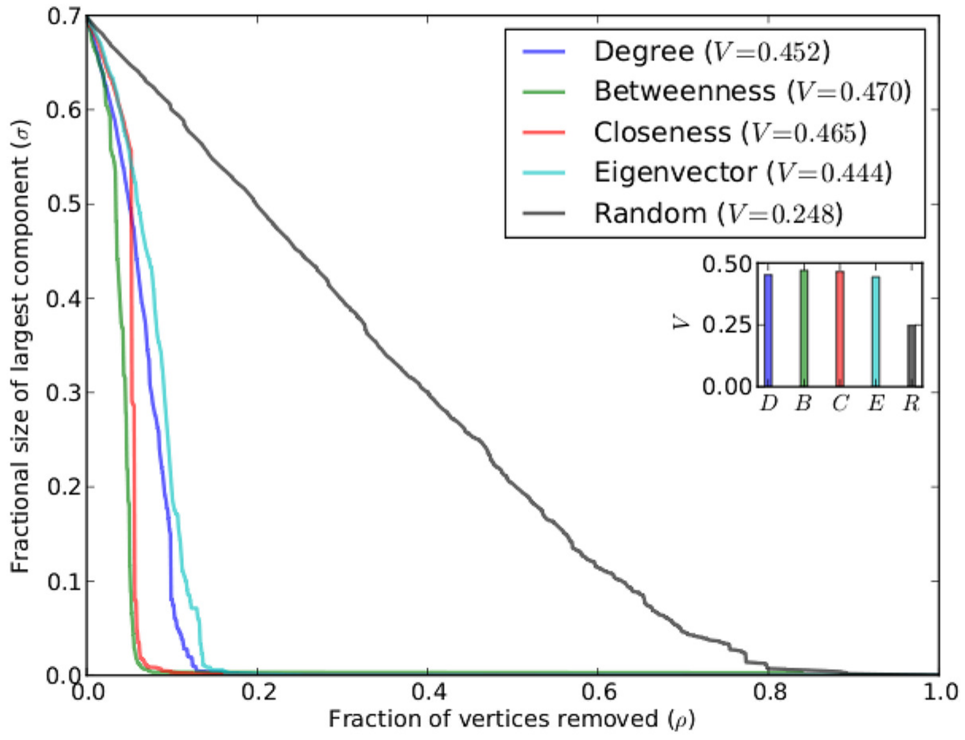


Figure 3.12: Edge-partitioned communication volumes versus # partitions on Twitter.

a well-ordered edge partitioned Fennel, but this is an order of magnitude slower than our vertex implementation.

Figure 3.12 plots Sheep against our Fennel implementation on Twitter; these results are not from Bourse. In this case the natural order of Twitter is unkind to Fennel, as Sheep outperforms it significantly. Nevertheless, this plot is similar to the Bourse plots if we assume that the natural order of Twitter is "random" for Fennel, and that our implementation should behave like Bourse's randomized fennel-VP. A well defined ideal vertex order is one of Sheep's key advantages.

Sheep's partition quality improves when it is provided with a lower tree-depth elimination order. Our observations regarding complex networks predict "sequential" betweenness centrality should improve over degree order (Section 3.4.3). Figure 3.13 shows how Sheep improves when using sequential betweenness order[57]. The input graph is a small complex networks dataset observed to disassemble more rapidly under betweenness attack than degree attack. As predicted, the tree-depth improves from 754 in degree order to 459 using sequential betweenness. The partition quality



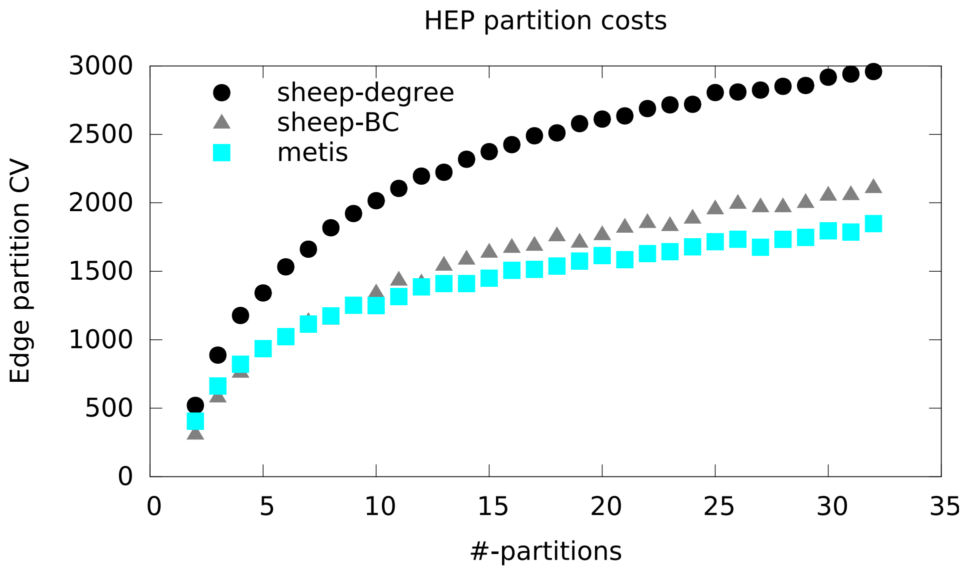


Figure 3.13: Above, attack plot on a High Energy Physics coauthorship graph. Below, edge-partitioned communication volumes versus # partitions on the same graph.

improvement is significant and roughly proportional to both the depth and the attack quality improvement reported by Iyer[57]. In fact, our results are comparable to METIS.

Unfortunately, sequential betweenness centrality is expensive to compute, so this exact method is impractical. However, there are methods to approximate and parallelize betweenness, and there are other efficiently-obtained centralities such as the k-core decomposition[35]. Conversely, it may be possible to derive useful centralities "in reverse" from low-depth elimination trees produced by e.g., METIS. While the importance of degree order is widely recognized in elimination trees, as far as we know the observation that non-local *analytic* centralities may reduce tree-depth is novel. This is an interesting line of research that we hope to address in future work.

3.6 Conclusion

Sheep is a graph partitioning algorithm that is many times faster than competing algorithms on large graphs without sacrificing partition quality. Sheep scales to effectively partition multi-billion edge graphs in as little as 4 minutes. Sheep's partition quality is comparable to, or even better than, METIS for small partition counts and competitive with streaming partitioners for larger partition counts. However, we see Sheep's most exciting contribution as the relationship it establishes between partition quality, tree decomposition theories, and analytic centralities. This is a rich space for innovative theories and system designs.

Sheep is free and open source software and is available at `https://github.com/dmargo/sheep`.

3.6.1 Limitations and Future Research

Sheep is an undirected communication volume partitioner, because its underlying theory works with vertex separators. There is a similar body of theory for edge-cut tree decompositions, called carving decompositions, so it may be possible to derive an edge-cut Sheep algorithm. The edge cut costs of our algorithm are unbounded and can be quite bad in practice, even though the communication volumes are consistently good. For edge-partitioned systems such as PowerGraph, an "edge cut" is not meaningfully defined, so this is not obviously a problem. However, the fact that high cut and low volume partitionings exist at all is interesting and a worthy topic for future research.

Sheep requires a good order to produce good partitions, but cannot easily create an order by introspection of the graph, because it splits the graph. Sheep works best for natural graphs with a skew degree distribution, such as "power law" graphs, because complex networks research shows that degree orders attack these graphs. Skew graphs are common in contemporary large graph analysis. For graphs with other distributions, such as finite element meshes, it may be possible to find an ordering heuristic by reviewing complex networks research, but we reserve this for future work. However, many of these graphs already have a rich history of successful partitioning methods such as planar bisection, whereas interest in skew networks is more recent.

Sheep creates a small partitioned tree that identifies the partition assignment of any graph element. This design is different from streaming partitioners, which place elements as they arrive, and affects best practices to integrate Sheep with existing systems. For example, an analysis system might ingest the graph, construct the tree, share it among nodes, and then direct each node's ingested data to the appropriate partition. If a placement step is necessary it should be trivially parallel, since the elements can be placed independently in consultation with the partitioned tree.

3.7 Acknowledgements

Thanks to Charalampos Tsourakakis, Florian Bourse, and Milan Vojnovic for helping us fairly compare to the results in their respective papers. Thanks to Peter Macko, both for LLAMA and for implementing several new features for our use.

4

Graph Systems: Review

In the previous chapter we presented Sheep, a graph partitioning algorithm and implementation that solves the partitioning "chicken and egg" problem. We motivated this problem with specific references to contemporary graph analysis engines, including Giraph, PowerGraph, and GraphChi. As it happens, these three systems are all examples of *vertex-centric programming* frameworks, a class of graph analysis systems that have enjoyed recent research interest. In this chapter, we discuss the research contributions and design principles of contemporary graph analysis systems. We review vertex-centric programming systems, dataflow systems, sparse matrix-vector (SpMV) systems, and graph databases in that order. The purpose of this review is to prepare for the systems metastudy in the following chapter.

Most graph processing systems are broadly derived from Leslie Valiant's Bulk Synchronous Parallel model[123]. BSP divides a parallel computation into supersteps. In each superstep, tasks compute in parallel, communicate with one another, and then arrive at a global synchronization barrier before the next superstep. In a typical graph processing system, the tasks are vertex updates, the communications are messages passed along edges, and the barrier is one iteration of a converging process. For example, one might implement a single-source shortest path algorithm by updating each vertex with the shortest path broadcast from its neighbors until convergence.

However, modern graph processing systems have diverged from and contributed to the BSP model in many significant ways, which we present throughout the chapter. Vertex programs (Section 4.1) are the most common task decomposition, but we will also see edge programs, subgraph programs, and systems that compose these tasks. Task decompositions imply data partitions and graph partitioning is not trivial, so we will see different partitioning idioms coupled with novel partition solvers. Because these inter-task data dependencies are not trivial, we will see new *asynchronous* scheduling idioms that relax the synchronous BSP barrier step. This "amorphous parallelism" explains why general-purpose parallel dataflow systems (Section 4.2) are also efficient graph analysis systems when fitted with an appropriate interface. Conversely, SpMV systems (Section 4.3) leverage the deep history of high-performance linear algebra via a logical interface from graphs and vertex programs to matrices and semirings. We summarize and discuss these themes in Section 4.5.

4.1 Vertex Programming Systems

Pregel[85] is a programming and computational model designed to be implemented on a distributed cluster. It introduced the concept of *vertex-centric programming*, in which a graph algorithm is expressed as a computation that runs iteratively on every vertex. In each iteration a vertex receives messages, computes a result, and then can send messages to its neighbors. The authors show that this model is sufficient to express many graph algorithms in clear, terse language. For example, a random walk algorithm receives messages containing the incoming walk probability from its neighbors, updates its own probability, and then sends messages that divide its probability among its neighbors. The path to a distributed implementation of Pregel seems straightforward, because the model is built up from BSP and message passing; however, the authors did not publish a reference implementation. In this respect, Pregel and Giraph (below) are analogous to MapReduce and Hadoop, in that Giraph/Hadoop are production systems built using Pregel/MapReduce as a guideline.

Giraph[11] is Apache's open-source implementation of Pregel. Giraph is a production system and not a research project, so it is fairly faithful to Pregel's specifications. One noteworthy caveat is that Giraph is implemented on Hadoop MapReduce, but Pregel's authors claim that MapReduce is an unsuitable backend for graph algorithms, citing performance concerns. Consequently Giraph is slow and, like Hadoop, served as a punching bag for early comparative evaluations. However, its reputation was rehabilitated when Facebook announced that their internal modifications to Giraph support computations over trillions of edges, an enormous scale by most standards. Most of these modifications are practical, such as multithreading support and efficient object serialization. However, a few address the same research issues as the systems described below, such as an edge-centric processing model and "splitting" vertices with many incoming messages.

GraphLab is a framework that was originally authored[82] for scalable parallel machine learning, because Hadoop was not sufficiently expressive for complex ML tasks, which often use graphical models. With the explosion of interest in Pregel, GraphLab's authors generalized their system to vertex-centric programming[81], and in time became an important competitor to Giraph. GraphLab is not strictly faithful to the Pregel model and explored different conventions as to how data is transferred between vertices, how vertex programs are scheduled, etc. In particular Graphlab supports *asynchronous* execution, in which vertex programs are not scheduled in iterative supersteps, but are instead scheduled whenever new messages are available. This scheduling policy introduces indeterminacy but, in practice, quickly converges to the correct result. Scheduling flexiblity is achieved by associating an RW lock with each vertex and then simply adopting different lock protocols. The authors use techniques such as vertex coloring to help produce schedules with less lock contention.

PowerGraph[45] is the second version of GraphLab, introducing several key research concepts that made it one of the first major alternatives to Giraph and a mainstay of comparative evaluations for several years. Prior vertex-centric systems distributed their work by assigning vertices to workers and then unicasting messages across cut edges between workers. In contrast, PowerGraph distributes itself by assigning disjoint sets of edges to workers and then duplicating and synchronizing state across cut vertices. The authors argue extensively that this model is better suited to social networks, which invariably feature some high-degree vertices that create imbalance if they are assigned to any one worker. This argument influenced the design of later systems, many of which adopt this model or some further extension thereof. PowerGraph also introduced Gather-Apply-Scatter (GAS), a new formal model for vertex-centric programs intended to aid program decomposition into discrete tasks for the benefit of the framework. However, the GAS model was more difficult to program and for that reason was not widely adopted in production systems.

Graph programming frameworks need to support the safe deployment of correct application-specific graph programs. Alternative scheduling idioms that violate well-understood BSP conventions can complicate this goal for both the system and the programmer, who has to learn new conventions. GRACE[125] is a vertex-centric programming framework that eases the transition between a synchronous prototype and a high-performance asynchronous deployment. GRACE's vertex programs are marked up with explicit isolation and consistency guarantees that impose restrictions on the schedule. The programmer can relax these guarantees to gain access to additional programming tools such as priority scheduling. GRACE then chooses a scheduler that is appropriate for the program's synchrony guarantees.

Ligra[112] is a parallel shared-memory graph processing framework with an unusual programming interface. It is based on simple map operations with support for vertex subsets and by extension induced edge subsets. This idiom is well-suited to traversal algorithms in which only a small subset of vertices are active in each step. Ligra is intentionally simple and not distributed, because its authors argue that modern servers will have enough memory for the foreseeable future. Ligra+[113] is a logical extension of this design that adds standard compression schemes, such as byte codes, to Ligra.

Giraph++[119] is a research fork of Giraph that introduces a more flexible subgraph-centric programming model. This model allows vertices in the same partition to bypass the message passing and scheduling subsystem by communicating through programs that operate over partitioned vertex sets. Giraph++ distinguishes between "boundary" vertices that communicate between partitions and "internal" vertices whose entire communication can be handled by subgraph programs. An important result for practical system building is that this model's performance strongly depends on a cut-minimizing partitioning. Giraph++'s runtime performance varies by as much as 25x between traditional hash partitions and cut-minimizing partitions; however, Giraph++'s distributed implementation of the METIS partitioner is itself a significant preprocesser overhead. This is a textbook example of the partitioning "chicken and egg" from Chapter 3.

Giraph++ is still "vertex-centric" in that subgraph programs operate over sets of vertices. Blogel[127] is a system that promotes subgraphs to first-class citizens of its programming model: subgraphs can store their own state, and address and communicate with each other. This model allows authors to express graph algorithms at a higher level of abstraction and still retain the advantages of a framework, such as distribution and fault tolerance. The flagship application for this model is Djikstra's shortest path algorithm, whose priority queue structure is difficult to express as a vertex program, but as a subgraph program is straightforward. Blogel is also sensitive to partition quality and implements several application-specific partitioners based on Voronoi diagrams and 2D spatial coordinates. These heuristic partitioners do not minimize cuts as well as METIS, but they introduce much less preprocessor overhead.

The disadvantage of PowerGraph-style edge partitioning is that a cut vertex spans several machines and is therefore a multicast synchronization, whereas a cut edge is merely a unicast. Therefore, cutting vertices amortizes traffic only for vertices with many edges. PowerLyra[29] is a system that hybridizes edge cut and vertex cut designs by explicitly distinguishing between low-degree and high-degree vertices according to a user-defined threshold. High-degree vertices are replicated as in PowerGraph and use the same GAS computation engine. However, low-degree vertices pass through a streaming partitioner to reduce replication, and their computation engine tries to aggregate GAS steps that are co-located on the same machine.

GraphTwist[128] further pushes the distinction between different kinds of vertices into the computation model. In an asynchronous scheduling model where vertices are run whenever new data is available, high-degree vertices are more likely to receive updates, and are therefore scheduled more frequently and emit more updates. GraphTwist formalizes this by quantifing the utility of each computation and pruning low-utility computations from the schedule. This affects the approximate result but, as with asynchronous scheduling, the result correctly converges in practice. In addition, GraphTwist aggressively indexes and partitions the graph along multiple dimensions, such as edge weight, and then exposes all of these partitions as potential subgraph program targets.

Asychronous vertex programs still encounter the occasional global synchronization point, such as when they test a stop condition that aggregates every vertex. Giraph Unchained[50] is a research fork of Giraph that tries to push asynchronous execution by removing as many global barriers as possible. This is achieved by simply replacing most global barriers with machine-local barriers that test the "logical superstep." Such a barrier can aggregate the local vertices and test whether a global barrier is necessary.

4.1.1 Streaming Systems

WebGraph[21] is an early but impressive system in which streaming is used to support efficiently computable graph compression techniques. The authors' key insight is that graph algorithms are frequently bound by data bandwidth, so compression improves performance by leveraging unused CPU cycles to speed up the data transfer rate. WebGraph adapts standard sparse compression techniques to graphs, such as universal integer codes whose statistics are fit to the input graph's power law constants. It also exploits community structure by coding vertex neighborhoods as differences relative to similar vertices within a streaming "window." These techniques work best when the graph's vertices are sorted such that connected and topologically similar vertices are near one another. This requirement is similar to partitioning, and WebGraph uses a label propagation sorting algorithm that is similar to a label propagation partitioner.

GraphChi[72] is a spin-off from the GraphLab group that emphasizes efficient single-machine streaming in place of distributed computing. It uses a "window" model like WebGraph, but its windows are instead discrete partitions that it manages as explicit working sets. GraphChi naively partitions the vertex range and assigns edges to the partition of their target vertex such that each working set fits in memory. Each partition is loaded in order, and a full pass over the partitions is equivalent to one iteration of a vertex-centric program. Conceptually GraphChi performs the role of every machine in a distributed vertex programming system, one machine at a time. The vertex programming model gives formalism and structure as to how data passes between working sets.

A major weakness of GraphChi is that the input graph requires substantial ingest processing to fit into the window model. X-Stream[106] is a GraphChi competitor that focuses specifically on arbitrarily-ordered edge streams without imposing any kind of vertex partition structure. This is made feasible by an *edge-centric* programming model that expresses graph algorithms in terms of how to pipe data between vertices. This model cannot express some operations such as per-iteration vertex initialization or finalization, so X-Stream supports mixed edge and vertex-programming idioms. In implementation both GraphChi and X-Stream are most concerned with how to parallelize intra-partition computation and how to prefetch the next working set.

4.2 General Dataflow Systems

GraphX[126, 46] is a spin-off from the GraphLab group that implements vertex programming on top of Apache Spark. Because GraphX exports a substantially different interface than its underlying system, it must exploit the vertex programming model to bring performance in line with specialized graph systems. The authors of GraphX reason about vertex programming in terms of optimizations to recursive joins and view maintenance. Recursive joins occur because, in each iteration, vertex programs pull and push values from and to their neighbors by joining the edge table to itself from targets to sources. View maintenance occurs because updates to vertex and edge value tables are only partial if some vertices are not activated or updated in every iteration. For many other problems GraphX adopts solutions from PowerGraph, such as edge partitioning with cut vertices.

The recursive join described above makes graph algorithms difficult to author and run efficiently in dataflow systems that do not support and optimize looping flow constructs. Naiad[93] is a general-purpose distributed dataflow system that supports efficient streaming through looping flows. Naiad's authors identify that their primary bottleneck for streaming loops is the time spent coordinating the end of a loop iteration across machines; this is analogous to the global barrier at the end of an iteration in synchronous graph engines. Naiad solves this problem by modeling dependencies between iterations and using virtual timestamps to synchronize dataflows between iterations.

Galois[68, 96] is a general dataflow system that, like Naiad, uses a dataflow graph programming model. Unlike Naiad, Galois was originally designed for large shared-memory machines and is focused on efficient parallelization instead of distribution. Galois is an "optimistic" paralllelization system that tries to speculatively extract parallelism from "amorphous" programs in which parallelism is not obvious. This model is unsurprisingly well-suited to graph-structured input data because parallelization opportunities vary with the input graph's structure. Galois performs well and has become a popular comparative evaluation target even for specialized graph systems.

4.3 Abstract Algebraic Systems

There are many useful equivalences between graphs and sparse matrices, as previously discussed in Chapter 2, Section 2.4 and throughout Chapter 3. Unsurprisingly, there are many similar equivalences between graph algorithms and abstract algebraic algorithms. For example, a shortest path algorithm is equivalent to matrix-vector multiplication in the semiring where $+$ is *min*, and $*$ is $+$. Pegasus[59] is a graph mining system for Hadoop based on generalized matrix-vector multiplication. The key component of Pegasus is an efficient Hadoop implementation of blockwise multiplication; the graph's edges are grouped by, mapped, and reduced in terms of blockwise submatrices. This brings a staple high-performance computing technique to Hadoop that would be difficult for a non-expert to realize in a hand-rolled implementation.

GBASE[58] pushes this block model further by explicitly partitioning the graph into dense and sparse blocks (e.g., by reordering and diagonalizing the adjacency matrix). Both the dense and sparse case are favorable for standard compression tools. Compression significantly reduces the data bandwidth of block transfers and thereby improves performance. GBASE also associates metadata indices with each compressed block so that targeted queries, such as neighborhood lookup, can search through a limited subset of blocks.

GraphMat[116] (now rebranded as Intel GraphPad) is a compiler that transforms vertex-centric programs into abstract algebraic operators that run on a dedicated sparse matrix-vector multiplication (SpMV) backend. This is made feasible by decomposing the vertex programming model into rigid send, process, reduce, and apply steps, in which the algebraic structure of the program is obvious. Unsurprisingly, a mature SpMV backend dramatically outperforms Hadoop and is competitive with some of the best systems, such as Galois, and even hand-optimized code. GraphMat was an important demonstration of the value that high-performance computing research into generalized SpMV has to offer to the graph analysis community.

Historically, the Basic Linear Algebra Subprograms (BLAS)[73] specification has been critical to the development of interoperable numerical linear algebra systems. In contrast, graph system interfaces vary greatly; even "vertex-centric programming" refers to a wide range of models whose frameworks impose different restrictions and guarantees. The Graph BLAS[87] is an evolving interface specification whose model is greatly influenced by generalized SPMV. Numerous projects implement and contribute to the GraphBLAS, such as GraphMat, CombBLAS[26], D4M[64], Graphulo[41], and GPI[36].

4.4 Graph Databases and Languages

Graph databases are distinguished from analysis engines in that they focus on the online point query and storage of dynamic graphs, typically with associated data properties, a query language, and some transaction support. These features come with high performance costs that must be optimized for point query workloads, often at the expense of big data analyses. For example, graph database partitioning schemes typically favor load balance at the expense of overall cut minimization, because load balance affects query throughput, and the cut cost of a point query is bounded. The traditional analysis workflow is to export a static database dump into some other framework. However, many recent analysis frameworks have tried to improve this workflow by supporting streaming or otherwise dynamic graph data. Insofar as this brings online query and dynamic storage support to graph analysis systems, the difference between analysis and database systems seems to be diminishing.

TAO[24] is a database caching layer that imposes graph semantics and graph-aware caching over Facebook's RDBMS backend. In operation, Facebook is prone to hotspot phenomena that follow from its social and temporal network structure, such as when a "celebrity" posts an update that is then read by their many friends. Therefore, TAO uses simple hash-based partitioning to equitably distribute this load across cache servers. Hash partitioning necessarily causes data to span many cache servers, so TAO implements a consistency model for those servers in terms of graph semantics.

The substitution of graph semantics for relational semantics allows programmers to reason about view consistency from the perspective of the social network.

GreenMarl[54] is a language for expressing graph algorithms coupled with its own compiler and implementation. It is distinct from GraphMat in that it compiles to its own graph primitives instead of generalized algebra. In particular, GreenMarl has efficient primitives for generalized BFS and DFS traversals, which are a typical weakness of algebraic systems.

LLAMA[83] is a dynamic graph storage layer, whose author's key insight is that the friction between dynamic graph storage and high-performance analysis is caused by the immutability of CSR representations. Therefore, LLAMA provides a mutable CSR by using mutable arrays that are similar to log-structured merge trees. An important observation is that deleting a vertex is particularly pernicious and requires special handling on merge operations to keep the whole graph representation in a valid state. Even though LLAMA is mutable and persistent, its analysis performance is competitive with in-memory streaming analysis frameworks such as GraphChi and XStream. Sheep's reference implementation is built on top of LLAMA (Chapter 3), and we also provided help with LLAMA's evaluation[83].

Gremlin[105] is a graph traversal language backed by a JDBC-esque interface called TinkerPop. TinkerPop generalizes graph databases that support a "property graph" model that assigns key-value properties to vertices and edges. These properties allow the interface to read and write generalized data such as vertex labels or edge weights. Gremlin is a functional language that programs a graph traversal virtual machine that operates on the property graph model. In principle Gremlin can program vertex-centric frameworks that export the property graph model, but only Giraph does so.

Some databases that implement the property graph model include Neo4j , DEX/SparkSee , HyperGraphDB , InfiniteGraph , OrientDB , and Titan . Neo4j is the most commercially successful of these, so it sometimes appears in the research literature as a comparative punching bag. Neo4j and its competitors are typically characterized as "NoSQL" databases whose traversal models solve the recursive join problem that is inherent to relational graph representations. Most of these are commercial production systems, and a review of their specific features is outside the scope of this dissertation.

The Resource Description Framework (RDF) was originally a metadata standard, but has since evolved into a more general model for conceptual knowledge management and data exchange. RDF records are *subject-predicate-object* triples, and it is conventional to depict subjects and objects as vertices, and predicates as edge labels, in a directed graph model. Thus, RDF storage and query engines can be thought of as graph databases, although the full field of RDF research is not generally graph-related[8]. Similarly SPARQL, the standard RDF query language, contains pattern and traversal syntax that can express e.g., subgraph isomorphism problems, although most SPARQL syntax operates on relational attributes. RDF engines began as adapters to relational systems, but over time have adopted more graph-aware query optimizations[111]. The RDF community is particularly noteworthy for creating many large publicly available datasets, and a few of these turn up as graph datasets in the metastudy in the following chapter.

4.5 Common Themes

Vertex-centric programming is more of an idiom than a proper model, and the actual programming models of graph analysis systems vary greatly. Researchers actively experiment with these models, because their restrictions and guarantees substantially influence the performance, parallelization, and distribution of the backing system. Vertex-centric models decompose a graph algorithm into many small task units with modest restrictions that are well-suited to fine-grained scheduling and message passing systems. If the model is more restrictive, as in PowerGraph and GraphMat, then the system can exploit more knowledge about the program and, in particular, can exploit high-performance computing techniques such as generalized SpMV. However, a coarser and less restrictive model such as subgraph programming lets the programmer author efficient algorithms that are difficult in the language of abstract algebra, such as Djikstra's algorithm.

Most importantly, the programming model's guarantees determine how tasks can be scheduled. Scheduling in graph analysis systems has trended strongly towards asynchronous execution, and consequently, models have relaxed their guarantees. Global barriers create stalls, and asynchronous execution relaxes the need for global barriers; systems such as Giraph Unchained and Naiad minimize these stalls by other means.

However, asynchrony is also an algorithmic technique that reduces the "real" work done by the analysis system. Asynchronous systems converge more quickly because they compute on fresh data and do not restrict themselves to stale data from the previous superstep. Systems such as GraphTwist try to generalize and formalize this, but Djikstra's algorithm can also help us understand the importance of using the "best" data available. Djikstra's algorithm uses priority to prune path computations that would be explored and later discarded by overly generic algebra methods. Extensions such as the A* algorithm improve this pruning by exploiting constraints on the graph's topology[51]. Insofar as asychronous scheduling is driven by edge messages and therefore topology, it may be viewed as topo-statistical prioritization and deprioritization of fresh and stale data.

At the other extreme, recursive joins on edge tables produce far too many intermediate values for scalable graph analysis. Insofar as such a join corresponds to one synchronous iteration, they are one of the best examples of an overly generic method that explores too much and prunes too little. Systems that use joins, such as GraphX and most RDF systems, must use graph-specific specializations to realize joins at scale. Graph exploration and traversal concepts, such as the programmable "traversers" seen in Gremlin and Neo4j, are a common method to surpass joins.

All graph analysis systems suffer from data bandwidth bottlenecks as a direct result of the "random" topology access patterns of unsorted graph data. Superficially the bottleneck's location and its solution may seem to vary with the system, but fundamentally it always reduces to a partitioning or equivalently a sorting problem. Systems such as PowerGraph, PowerLyra, Giraph++, and Blogel explicitly invoke a partitioner as part of their ingest phase. Other systems such as Ligra++, GBASE, and WebGraph improve data bandwidth by using compression, but in each case their compression ratio depends on a partitioning or sorting step. Some systems, such as GraphChi and Galois, view sorting as a wholly independent preprocessing step. However, it is crucial to understand that all graphs are "sorted" in the order they are naturally produced and distributed throughout the research community. Insofar as natural orders may be optimal or pessimal, it is not experimentally sound to disregard them; this observation is discussed in-depth in the following Chapter.

Differentation between vertices is a common cause of bandwidth problems and a common solution to them. High-degree vertices create hotspots that may imbalance workers or create obstructions for traditional cut-minimizing partitions. Low-degree vertices are often numerous relative to their computational importance and can overwhelm a fine-grained system that does not batch or prune them. If the graph follows a power law, then both kind of vertices will occur as a significant fraction of edge endpoints. Systems such as PowerGraph, PowerLyra, and GraphTwist explicitly differentiate vertices, but every partitioning or sorting system "differentiates" vertices insofar as a feasible cut-minimizing solution must do so.

With so many diverse systems, it is naive but also edifying to ask: which ideas are winning? In terms of performance, the best systems are algebraic systems such as GraphMat and dataflow systems such as Galois. Consequently, commentators have cast doubt on the success of vertex-centric programming systems, because they are outperformed by systems from more mature research fields. However, *the original Pregel paper specifically introduced an attractive programming model and not a reference system.* Systems researchers are not using vertex programs to pursue performance: they are pursuing performance so they can write vertex programs. Thus, the fact that other mature fields are now exploring the model is a testament to its success.

For better or worse, benchmarks shape a field.

David Patterson

5 Graph Systems: Metastudy

We will now study how researchers evaluate contemporary graph systems in published research. The review of systems in the previous chapter presents hypotheses, conclusions, and other claims as to the functions and roles of these systems in the research field. Ostensibly, these statements are supported by evaluations that affirm, e.g., whether a novel feature "is a performance improvement." We will formally investigate the contents of these evaluations, identify weaknesses, and then evaluate whether or not these weaknesses are real problems in practice.

First, we conduct a metastudy of benchmarks and datasets used by contemporary graph systems evaluations. We compile a corpus of evaluations, we disambiguate and enumerate references to benchmarks and datasets, and we determine which are most referenced. Then, we identify features of the most referenced benchmarks and datasets that present challenges for evaluations, and we also identify several recurring problems in evaluations throughout the corpus. We then evaluate whether or not these issues are significant enough to meaningfully affect published results. We show that, in a representative case, seemingly innocuous features, such as a graph's vertex IDs, can affect runtime up to a factor of two. We finish with a set of recommendations for future systems evaluations.

5.1 Introduction

The large number of contemporary graph systems (Chapter 4) is naturally accompanied by a large volume of comparative analysis; indeed, a "performance improvement" is the driving hypothesis behind many research systems. Performance can be quantitatively defined as runtime, scalability, power efficiency[39], or even user effort (e.g., lines of code). Improvement, however, is a more nebulous concept that entangles baselines, benchmarks, generalizability, and experiment design.

McSherry et al. prompted a great deal of conversation about "improvement" with their 2015 introduction of the COST metric[89]. COST, or Configuration that Outperforms a Single Thread, quantifies the relationship between the performance improvement and overhead of a parallel or distributed implementation relative to a single-threaded implementation. If a scalable system improves performance faster than it introduces overhead, then it must have a finite COST at which the parallel implementation outperforms the single-threaded one. However, the primary result of McSherry's study is that many published graph systems have infinite COST: *there is no configuration for which these systems outperform a single thread!*

COST is not presented as a robust benchmark survey so much as a disturbing demonstration of how evaluation practices can affect our perception, understanding, and the conclusions we draw from results. The issues raised by COST are simple and are widely known in principle, but in practice are not captured by a typical intra-comparative plot of runtime versus thread count. The demand for scalable systems drives a demand for comparable, positive scaling results and ultimately biases our evaluations in ways that may mislead us. This observation has been a major inspiration for the work we present in this chapter.

McSherry et al. addressed parallel scaling baselines and benchmarks, but said comparatively little about data and evaluated only two datasets. However, Sebastiano Vigna discussed dataset issues in an underappreciated opinion piece from 2007 titled "Stanford Matrix Considered Harmful."[124]. The target of Vigna's ire is a crawl of 300, 000 Stanford web pages that was used as a benchmark and testbed by "a considerable set of papers in the literature." Vigna presages McSherry when he notes that his laptop can easily compute PageRank on such a graph, but he also identifies numerous ways in which the Stanford graph differs radically from other web graphs.

Vigna argues that "the problem of computing PageRank is interesting...only if the size of the matrix is large and if the type of the matrix is a web graph." The graph's "type" is significant because it implicitly defines the graph's structure (e.g., in terms of a statistical power law). Algorithms may exploit these structures, and may be hindered by the absence of useful structures or the presence of obstructing structures; however, if nothing else, graph structure always affects cache hit rates. Vigna's own WebGraph system exploits cache-efficient compression of redundant power law graph structures, so with respect to his own research, the importance of graph structure is quite obvious.

In the following section we investigate how benchmarks and datasets are used in practice by graph systems research papers. Unlike Sherry and Vigna we quantify our claims about the field over a well-defined corpus, so we identify some "dark horse" issues that might otherwise go unnoticed. Though we found no instances of the Stanford graph in modern research, many of the same issues are present in popular datasets. Previous authors made no attempt to quantify the potential impact these parameters may have had on published results, but we show that the impact is significant and systematically biased.

We analyzed 65 papers describing graph systems, culled from a much larger corpus of thousands of papers published in seven major conferences from 2011 to 2015. We show the de facto benchmarks and datasets that have emerged in this area, and we show systematic errors that are regularly committed in evaluations. In particular there is widespread disregard for the performance relationship between popular benchmarks and the statistical properties of popular datasets, such as the cache effects induced in different benchmarks by non-random vertex IDs. We hope this review will aid future authors in devising more useful evaluations and better systems.

5.2 Systematic Paper Review

Any systematic review must begin with a research corpus. However, the volume of graph processing research and its generality (i.e., pertaining to the class of "graph problems" rather than a specific application) raises questions as to what criteria define a graph systems paper. For example, is an RDF query engine that effectively solves a subset of subgraph isomorphism problems a "graph processing system"? Our choice of venues in which we search for papers is similarly ambiguous.

We resolve that ambiguity by clearly stating our ideal corpus, which we methodically construct as best we can. *We want to review every paper that describes a graph processing system and was published after 2010's Pregel, which is generally regarded as the first "vertex programming" system*[85] *and a major motivator for recent research.* A "graph processing" system is an implementation that explicitly operates on graph-structured data and produces analysis or query results, and therefore graphs should be explicitly discussed in some associated paper. We impose no generality requirement on such a system, and therefore, an implementation of a single-purpose graph algorithm is a graph processing system. Conversely, a general-purpose processing framework that "can" implement graph algorithms is not within our scope unless some paper explicitly discusses those graph algorithms.

We have tried to adopt a consistent method in our paper search, but ultimately our judgement and expertise must play important roles in the search process. Inevitably, some distance will exist between our limited experience and the diverse experiences of our readers. Though this could be

Venue	Years	Total	Graph Related	Refers to System	Presents System
GRADES	2013-15	38	35	28	12
KDD	2011-15	1051	386	103	6
OSDI	2012,14	67	28	13	3
PODS	2011-15	143	77	13	1
SIGMOD	2011-15	730	330	102	16
SOSP	2011,13,15	89	40	17	5
VLDB	2011-15	1118	482	162	33

Figure 5.1: Venues used in this study. A paper is "graph related" if it mentions graph or networks and nodes, vertices or edges. A paper "refers to a system" if if contains a term from Figure 5.2.

construed as a sampling error or noise, the nature of such "error" changes with the expectations of the reader. Rather, we encourage you to read this paper as an autopsy and diagnosis of some limited subset of the graph processing community. The extent to which this diagnosis applies to the your own research community is left to your best judgement.

5.2.1 Method

We began with the complete proceedings of the venues described in Figure 5.1 (3236 papers). We chose those venues for their diversity, the availibility of their proceedings, and because we are familiar with all of them and therefore reasonably qualified to assess them. We filtered these proceedings for all papers that contained the terms "graph" or "network", and any one of "node", "vertex", or "edge" (1380 papers). This fits our requirement that the papers be explicitly about graph data.

We then created a list of popular graph processing systems (Figure 5.2) and searched for papers that mentioned those systems. We looked through these results manually and retained any paper that, in our personal assessment, met our definition of a graph processing system. Any additional systems we discovered were then added to our search terms, and we repeated the process until it returned no new results. Note that many papers discuss the same system, many papers name systems outside our corpus, and many systems go unnamed; therefore there is not a one-to-one mapping be-

APOLO	Giraph++	MADlib	Pregelix	Symple
Arabesque	GiraphUC	MapGraph	PSgL	SystemG
Aster	GPS	Medusa	RASP	SystemML
Asterix	GRACE	Mizan	REX	Titan
BIDMat	Graft	MOCGraph	Ringo	TriAD
Blogel	GraphBuilder	Myria	SAE	TriAL
CANDS	GraphChi	Naiad	SCAN++	TrinityRDF
Chaos	GraphGen	Neo4j	SEDGE	TurboGraph
CounterStrike	*GraphLab*	NOMAD	Shark	TwinTwigJoin
Dandelion	GraphMat	NSCALE	SimSQL	UDA-GIST
DEX	GraphTwist	OceanRT	SociaLite	UniAD
EAGr	*GraphX*	OptIQ	SOCRATES	UniQL
epiC	GreenMarl	PEGASUS	SparkSee	Vertexica
Galois	HelP	Petuum	SPARTex	Virtuoso
GBase	Horton+	*PowerGraph*	SpatialHadoop	WOO
GEMINI	KDT	PREDIcT	Stratosphere	XStream
Giraph	Ligra	*Pregel*	STwig	YZStack

Figure 5.2: Terms used in this study. The italicized terms are the initial "seed" terms. Other terms were found by recursive search (Figure 5.3).

tween papers and search terms. This process discovered 440 papers, of which we decided 77 are examples of graph systems. All of the raw data is available on request and will be published promptly.

This method is, ironically, a graph analysis that tries to find a connected reference component seeded by our initial search terms, and will wrongly exclude any graph systems paper that has no reference path to the initial terms. This method is necessary to practically reduce the number of papers we must personally review. There is some danger that our judgement imposes a subjective shape on the reference component by including or excluding certain references, but this error should be compared against the immense error that would be introduced by an impractical, shallow review of the full corpus.

For each of the 77 papers, we manually determined all of the benchmarks and datasets used in its evaluation. Some papers do not have a quantitative evaluation, so this further narrows us to 65

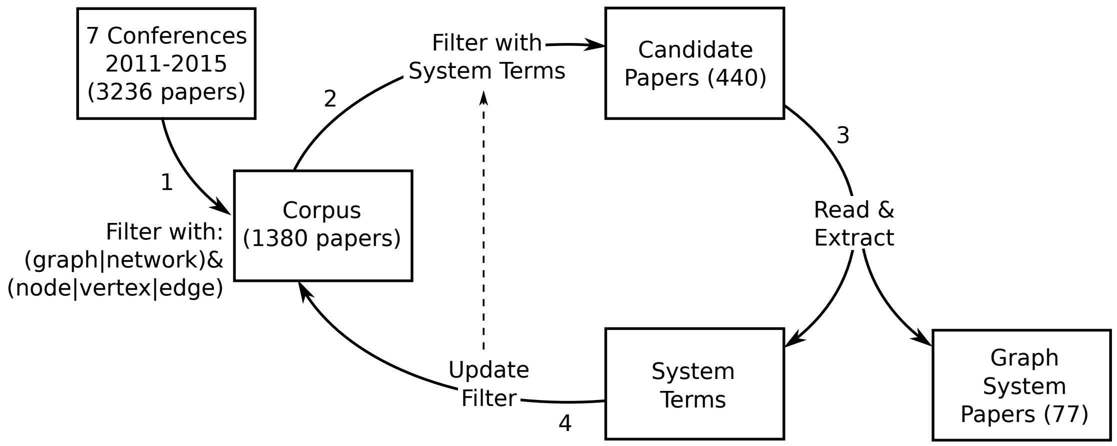


Figure 5.3: Our procedure to identify contemporary graph system papers. 1.) Our corpus consists of 7 conferences over 5 years pruned using common-sense graph terms. 2.) We filter our corpus with major graph system names to obtain candidate papers. 3.) We read the papers and extract more graph system terms from them. 4.) We update the filter with the new terms and repeat the process.

papers. This process must be manual because there are significant entity linkage concerns regarding benchmarks and datasets. For example, a paper might reference a “KONECT Wikipedia dataset”, but in fact there are numerous Wikipedia datasets on KONECT, and the specific dataset can be identified only by its vertex and edge counts. This is further complicated by variations in the reported vertex and edge counts of identical datasets! So we encounter both names that refer to non-specific datasets and specific datasets with varying features.

5.2.2 Benchmarks

In total, 74 distinct benchmarks appear in our corpus. The distribution is long-tailed, with only 36 benchmarks that appear in more than one paper; however, there is substantial reference mass spread among these 36 benchmarks (Figure 5.4), which suggests that the top benchmarks are widely agreed on and used in practice. The top 11 benchmarks are used 141 times out of 242 total benchmarks and are listed in Figure 5.5.

It will surprise no one that PageRank is by far the most widely used benchmark in graph pro-

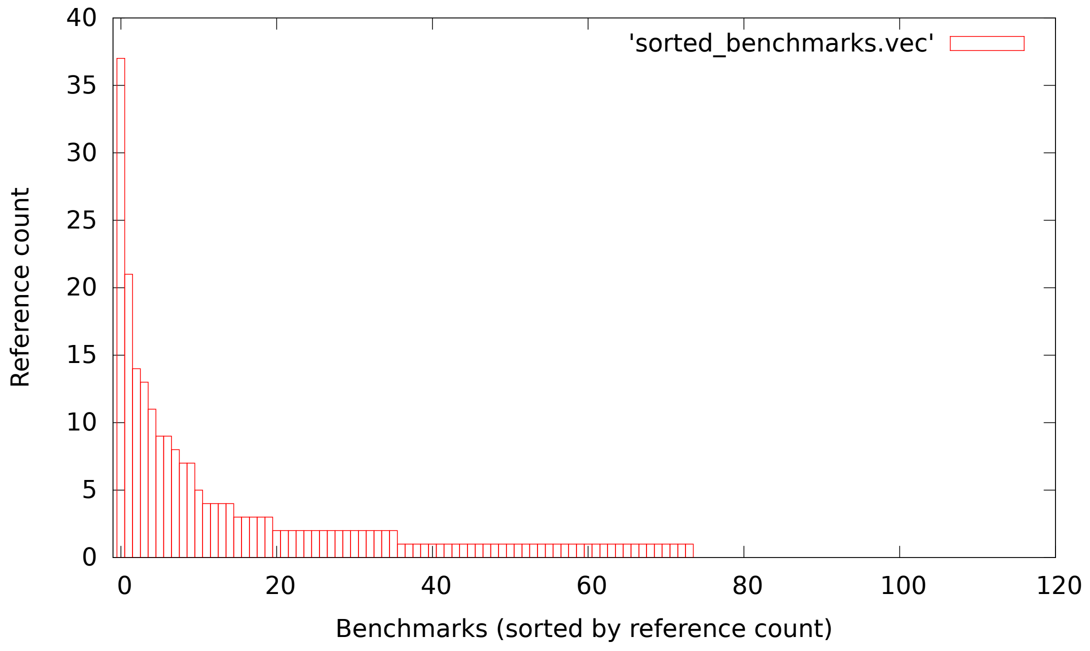


Figure 5.4: Distribution of benchmark reference mass.

cessing, followed in order by connected component detection, single-source shortest paths (SSSP), and triangle counting. Breadth-search first (BFS) is also popular and, if treated as a specific instance of SSSP (uniformly weighted), would boost SSSP to the second most popular benchmark. Similarly, subgraph matching is popular, and triangle counting may be viewed as an instance of 3-clique matching. We should note, however, that the specific features of BFS and triangle counting lead to improved algorithms, such as the "forward" triangle algorithm[109].

PageRank is most commonly implemented by using iterative methods until convergence, but is commonly benchmarked by the runtime of a fixed number of iterations. In each iteration each vertex is updated exactly once. However, some modern implementations exploit "asynchronous" methods that run every vertex as often as new data is available or prioritize vertices that are far from convergence. These techniques are deeply coupled to the vertex scheduler and its parallelism model, so they are compelling for systems research. However, the fixed iteration benchmark does not triv-

Benchmark	References
Pagerank	37
Connected Components	21
Single Source SP	14
Triangle Counting	13
Partitioning	11
Breadth-First Search	9
LUBM (SPARQL)	9
Subgraph Matching	8
Ingestion	7
Collab. Filters (ALS)	7
Get Neighbors	5

Figure 5.5: The top 11 benchmarks by reference.

ially apply to them. Instead, asynchronous implementations must be measured to within a target tolerance of convergence, or more generally, by plotting convergence over time.

The convergence rate of PageRank is not directly related to the input graph size but is instead a function of the random walk's mixing time, which may vary considerably between graphs with similar size and different structure. It is therefore difficult to compare convergence time between graphs, even if their sizes are controlled as in, e.g., a scale plot. In contrast, a fixed iteration count is in principle O(|V|+|E|) operations, which may explain the preference for fixed iteration benchmarks. But we must note that the PageRank *problem* is to find the converged distribution, whereas iterations are a feature of a particular *implementation*; see Section 5.3.2 for further discussion.

Connected component detection is often implemented using label propagation methods to convergence. Each vertex emits its label and then adopts the smallest label it has seen until no labels change. This naive algorithm was long ago supplanted by algorithms based on disjoint-set data structures. However, label propagation algorithms extend to more complex problems, such as community detection[102], so it may be that connected components via label propagation serves as a "simple label propagation" benchmark. Label propagation is also run to convergence, but unlike asynchronous PageRank, it deterministically converges with a bound given by the graph's diameter.

In vertex programming frameworks, single-source shortest path is commonly implemented by broadcasting the path length from the source vertex. Each vertex adopts the minimum path length it has seen and then broadcasts the new path only when the length changes. Compared to Djikstra's venerable algorithm, this method does more work due to the lack of prioritization but is easier to parallelize. Later "block-structured" graph systems, which organize the graph into subgraph partitions, run a local Djikstra's algorithm within each subgraph to try to get the best of both methods.[127] This method is one of several batch processing techniques commonly seen in parallel SSSP implementations, the most important of which is probably delta-stepping[90].

Since sorting the search queue in priority order is a central concern of Djikstra's algorithm, one might assume that breadth-first search is comparatively trivial, because it is the unit-weight case of shortest path. This is true for a single thread, because the breadth-first search queue implicitly sorts itself in the correct order. However, parallel BFS is non-trivial for essentially the same reason as SSSP – it is expensive to coordinate parallel searches – and it relies on similar techniques, such as batching and delta-stepping. Thus, parallel BFS is still an active research area[27]. Furthermore, BFS is a benchmark kernel in the Graph 500 supercomputing benchmark[92], whose synthetic data generator is the most popular in graph systems research.

In principle, triangle counting is simply a 3-way self join on the edge list. In practice however this produces a huge volume of intermediate data that different systems handle in different ways. Many systems adapt one of several algorithms by Thomas Schank[109] that count all of one vertex's triangles before moving to the next vertex. Another solution seen in GraphChi[72] is to stream over the graph, buffer incomplete triangles, and then search for missing edges in the remaining stream; this method takes multiple streaming passes in proportion to the buffer size. Both of these methods are especially sensitive to the vertex order, so frameworks sometimes preprocess and reorder the graph. Triangle counting algorithms are almost always run to completion, although some approximation algorithms offer parameterized error tolerance (such as EigenTriangle[121]).

Many practical subgraph isomorphism frameworks operate similarly to triangle counting. Their increased join complexity is typically offset by the pruning potential of vertex and edge properties, such as in RDF semantic web applications. In particular, the Lehigh University Benchmark[48] (LUBM) is a widely-used SPARQL benchmark in the RDF community. It is noteworthy that LUBM's query set contains several queries that are essentially analogues of other popular benchmarks; for example, Q2 and Q9 are labeled triangle counting queries. In general, LUBM's queries are subgraph matching queries, although some path queries are also represented. Unlike the majority of graph benchmarks, LUBM is tightly coupled with a synthetic RDF graph generator that we discuss in the next Section.

Among the remaining benchmarks, we note a few instances of popular knowledge discovery algorithms such as ALS collaborative filtering and belief propagation. These algorithms might be said to apply to "graphical models" rather than graph datasets, although of course that distinction is unclear. ALS, for example, is only applicable to k-partite graphical models, which are tremendously constrained compared to a social network. Nevertheless, these constrained algorithms and datasets co-occur alongside more general graph metrics in the literature.

5.2.3 Datasets

113 datasets appear across our corpus, but only 35 appear in more than one paper; the distribution is shown in Figure 5.6. It may seem surprising that the degree of de facto standardization in graph datasets is comparable to benchmarks, because unlike the benchmarks most of these datasets are not of general research or business interest. Rather, the standardization results from efforts towards comparability, and the limited availability of datasets with particular characteristics, such as large social networks. The top 11 datasets are used 106 times out of 239 total uses and are identified in Figure 5.7.

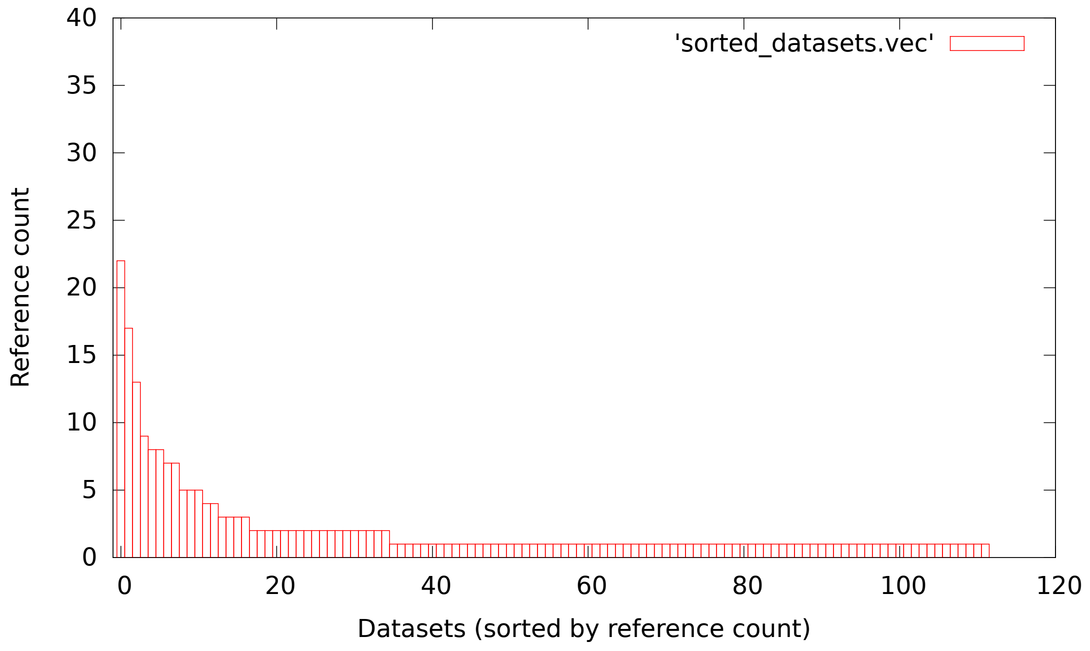


Figure 5.6: Distribution of dataset reference mass.

Compared to the benchmark distribution, the dataset distribution has less mass among the top datasets and more mass in its long tail. This difference reflects an important underlying difference in their reference structures – and who could resist the opportunity to draw a graph of graph datasets? Figures 5.8 and 5.9 visualize the co-citation networks of benchmarks and datasets respectively. Two benchmarks or datasets are connected if they appear together in a paper; therefore papers form cliques in this representation. If two groups of benchmarks or datasets are not joined by edges, this implies that there exists no paper that bridges the two groups.

Group structure is easily visible in the dataset visualization simply because it is sparse. There are many groups of datasets that occur in only a few papers and do not mix with other groups of datasets. Conversely, the benchmarks are largely mixed with the exception of the distinct RDF community. If we drop every dataset that has only one reference, then the dataset graph reveals a similar structure (Figure 5.10).

Dataset	References	Edges	Vertices	Max Vertex
twitter-2010	22	1.47b	41.7m	61.6m
soc-livejournal	17	69.0m	4.85m	4.85m
graph500	13	varies	varies	varies
LUBM	9	varies	varies	varies
cit-patents	8	16.5m	3.77m	6.01m
yahoo	8	6.64b	700m	1.41b
uk-2007-05	7	3.7b	106m	106m
com-friendster	7	1.81b	65.6m	125m
usa-roads	5	58m	23m	23m
netflix	5	99m	0.5m	2.67m
uk-2002	5	298m	18.5m	18.5m

Figure 5.7: The top 11 datasets by reference.

The well-known "What is Twitter?" dataset by Kwak et al. is a network of Twitter accounts and follower relationships. Its popularity may be attributed to its unusually large size for a publicly available social network dataset, the fortunate timing of its release (2010), and its use as a benchmark in Kyrola et al's GraphChi paper (2012), which is a frequent competitor for later systems. The authors of Naiad noted in 2013 that "several systems for iterative graph computation have adopted the computation of PageRank on a Twitter follower graph as a standard benchmark."[93] However, it is noteworthy that the dataset's authors concluded its topology is substantially different from other "social networks" in terms of its non power-law degree distribution, small diameter (maximum shortest path), and lack of reciprocity (correlation between edges X->Y and Y->X).

The remaining top ten datasets are an interesting mix of old friends and surprises. The Graph500 (Kronecker/RMAT) and LUBM synthetic generators, and the Yahoo and UK-200X web crawls, are well-known to domain experts. Conversely, the 2006 LiveJournal crawl by Backstrom et al. takes a surprising second place, perhaps owing to its availability as the largest social network in the popular Stanford Network Repository until 2012 (now superseded by Friendster, also in the top ten). Other surprises include the NBER patent citation network (from 2001!), the USA road network from the

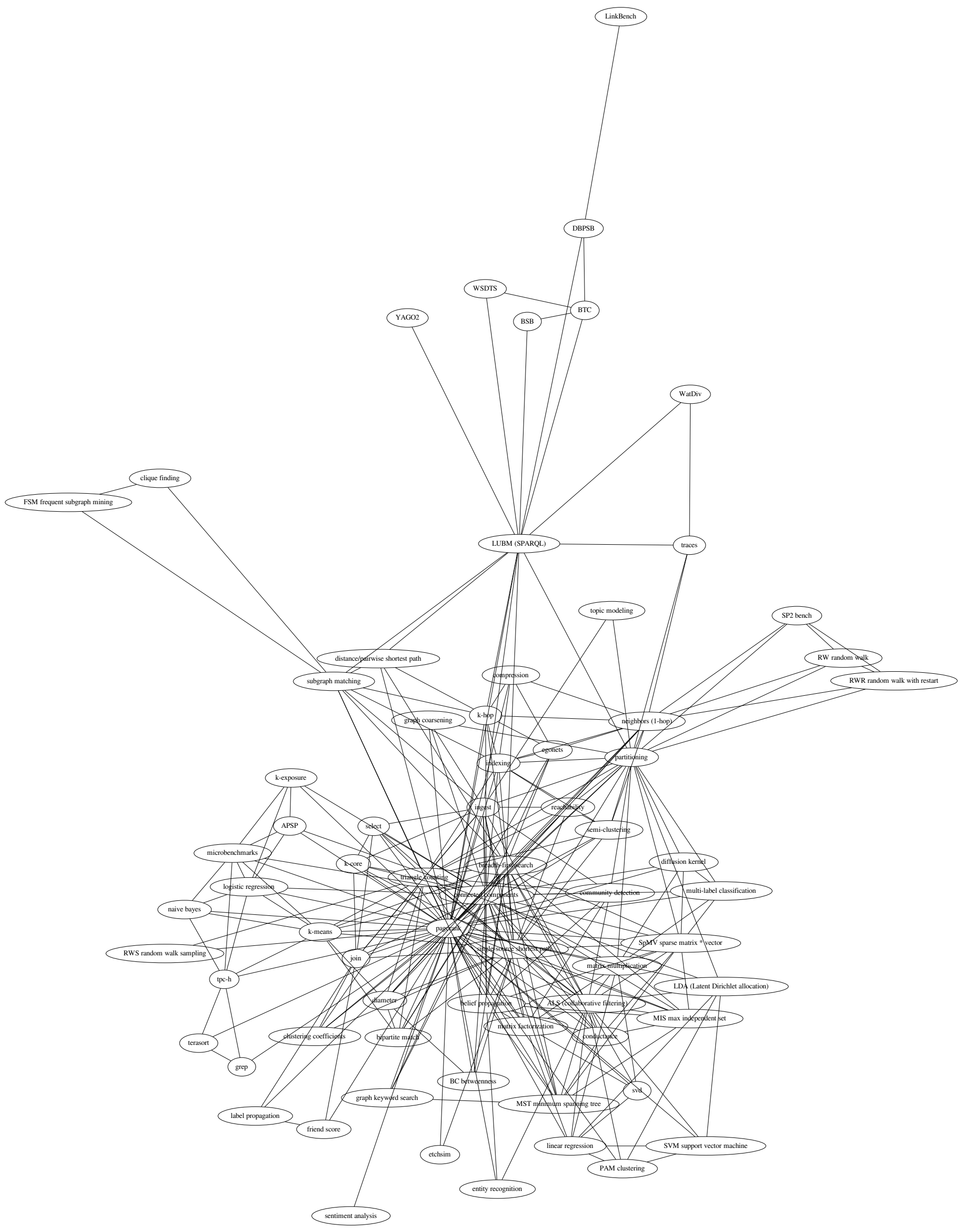


Figure 5.8: Benchmark co-citation network

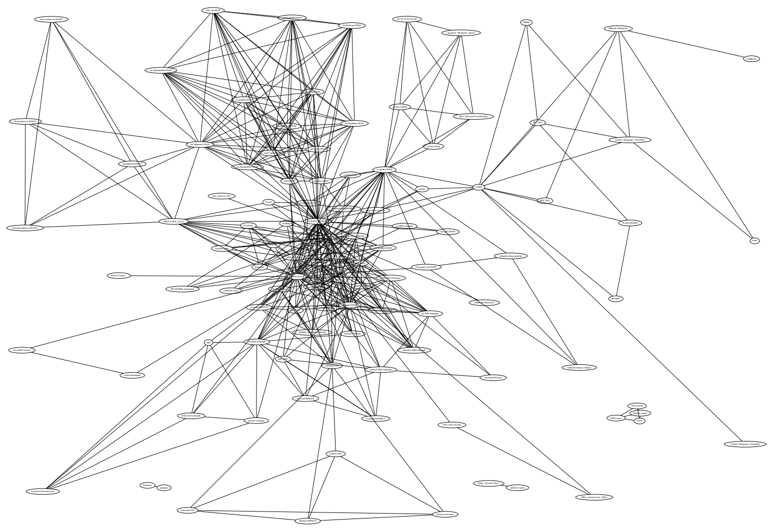


Figure 5.9: Dataset co-citation network

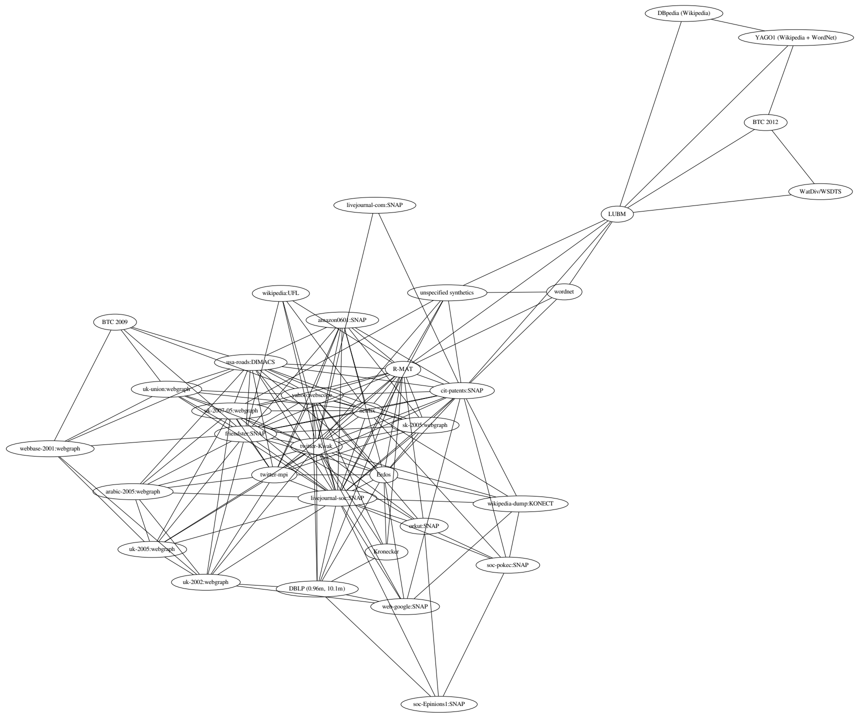


Figure 5.10: Co-citation network of datasets with more than one reference. The "handle of the Big Dipper" consists of RDF datasets with tightly coupled query benchmarks.

2006 DIMACS shortest path implementation challenge, and the Netflix Prize bipartite recommendation graph. These three graphs are tiny datasets by contemporary standards.

The Netflix prize is uniquely disturbing, because it is illegal to distribute and arguably unethical to experiment with. This dataset contains recommendations made by users who were later deanonymized by researchers Arvind Naranyanan and Vitaly Shmatikov in 2007[94]. This research was cited in both an FTC inquiry and a class action lawsuit that Netflix settled in 2010. Netflix cancelled further competitions and withdrew the Prize dataset, whose license clearly states that it cannot be redistributed without permission. Nevertheless, in 2015, three papers in our corpus used this dataset to benchmark recommendation algorithms! *We strongly recommend against any further use of this dataset.*

It is also unfortunate that the Yahoo web crawl is still in use, because it suffers from serious statistical issues described by the WebGraph Project[2]. About half of the graph's "vertices" have no edges, and unlike other web crawls the graph does not have a "giant" strongly connected component (less than %4 of the vertices are in the largest SCC). The combined effect of these issues is that many benchmarks run more quickly on this web crawl than its size would imply. This is a serious issue because the motivation for using this graph is almost certainly its perceived size, which is the largest among the top ten. WebGraph recommends UK-Union and ClueWeb-A as statistically sound web crawls of similar size, but unfortunately these datasets enjoy much fewer references in practice.

Researchers who obtain datasets such as UK-200X from WebGraph need to be aware of a few issues. WebGraph distributes graphs in a compressed format that benefits from a vertex reordering preprocess called Layered Label Propagation[19]. This process discards the natural order of the original dataset; therefore, WebGraph offers separate normal and "natural" datasets on their resource pages. Researchers who use WebGraph datasets specifically for benchmarking should arguably use the natural order, because LLP is a bandwidth-minimizing preprocess that can improve end-to-end performance (see Section 5.3 and Figure 5.12). Using LLP-sorted data lets systems benefit from reordering without including the preprocessing time, and discourages work on reordering, because it provides no further improvement. At minimum, researchers should clearly cite which dataset they used. If a use is undocumented, then it is likely that the LLP graph is being used, because LLP appears first on the Web page.

To be clear, WebGraph is an excellent resource with good data publishing standards; the issue is how graphs are handled and cited across the community. For example, the 10th DIMACS Implementation Challenge made both UK-2007-05 and UK-2002 graphs available as testbeds for graph partitioning and clustering algorithms. The DIMACS repository directs the reader to cite WebGraph for these datasets; however, the DIMACS datasets are not the same as those on WebGraph! They have been transformed into undirected graphs by adding their transpositions. Because DIMACS distributes these datasets in a standard compression format, whereas WebGraph distributes in a unique format that requires special tools, it is quite reasonable for researchers to choose the DIMACs dataset under the mistaken assumption that it is the same as WebGraph's, and to cite WebGraph as they are instructed to do.

This is one example of broad citation issues that plague the corpus. Undirecting UK-2007-05 nearly doubles its file size, but DIMACs claims the same edge count as WebGraph, because the reported edge count may be directed or undirected. Papers in our corpus do not always clarify this distinction, and there are several instances of datasets with varying reported edge counts. For example, Hong et al. claim Twitter contains 1.8 billion edges[55], but most sources claim 1.4 billion directed or 1.2 billion undirected. Bu et al. claim Yahoo contains 8 billion edges[25], but most sources claim 6.7 billion.

Similarly, in a graph distributed as an edge list where the maximum vertex ID differs from the unique vertex ID count, "vertex count" is ambiguous because it is not clear whether zero-degree vertices are counted. Twitter-2010 is often reported with 41.7 million vertices although its maximum vertex ID is 61.6 million; conversely, Yahoo is often reported with 1.41 billion vertices although its non-zero vertex count is 700 million.[2] This is most likely because these are the vertex counts as stated by their respective primary sources. *The community should obtain DOIs for its datasets and augment them with unambiguous metadata*, including the edge count and whether the edges are directed or undirected, the total number of vertices and non-zero-degree vertices, and, if possible, the procedure by which the vertices are numbered.

Synthetic graph models are dominated in practice by the Graph500 and LUBM generators, which is interesting because the concepts behind these two generators are strikingly different. The Graph500 generator is a straight-forward implementation of the popular Kronecker/R-MAT network model and produces unlabeled topology that is meant to *statistically* resemble a real-world network. In contrast, the LUBM generator produces labeled topology such that its *semantic* constraints, e.g., students per class, resemble a real-world RDF dataset. Unsurprisingly, LUBM is tightly bound to a particular benchmark query set (see end of Section 5.2.2), whereas the Graph500 generator, although bound to a connected component and BFS benchmark, is in practice used with many other numerical benchmarks such as PageRank.

5.2.4 Weaknesses

Benchmarks and datasets are in part the product of a citation process, so it is unsurprising that their reference counts follow long-tailed models. However, this natural process can have consequences for scientific and statistical reasoning. We would like some assurance that our benchmarks represent something other than the preferential attachments of our community. More importantly, if our test data is not randomly drawn, this has serious implications for the strength of inductive conclusions from empirical results. A hidden feature among the top 10 datasets could dramatically shape the plots of our research field.

For example, the default Graph500 Kronecker generates graphs whose edge density is strongly correlated with their vertex identities. If the edges are serialized in a vertex-sorted order, then the most commonly referenced edges and vertices will appear near the start of the serialization. This results in optimistic caching and prefetching behaviors relative to an unknown (e.g., random) vertex ID space; so, the Graph500 randomizes the vertex ID space to improve its generalizability. Equivalently, the Graph500 controls vertex identities by randomly sampling from all isomorphic vertex-labeled graphs; this may not be realistic, but it is controlled. Because vertex identities, vertex sort orders, and isomorphisms are so closely related, we will refer to these concepts interchangeably.

The overwhelming majority of evaluations in our corpus make no attempt to control the vertex identities of real datasets using randomization or otherwise. This would be fine, if enough datasets were sampled from some underlying distribution in which the vertex identities vary randomly or "realistically." However, our metastudy shows that so few datasets are used in practice that this cannot possibly be the case. The popular datasets may be labeled optimistically, pessimistically, or in some "average" way, but in any case there are not enough of them to be realistically sampled. Consequently, vertex identity is an uncontrolled variable with some unquantified influence over published evaluations, which we demonstrate in the following Section, e.g., in Figure 5.12.

Roughly the same argument could be made of any graph parameter, and there are infinitely many graph parameters; this parade of horribles could be dismissed as an unrewarding game of statistical whack-a-mole. From an evaluation perspective, such parameters are interesting only if they have a substantial impact on measured performance, and in principle, a synthetic generator that controlled every substantial parameter would resolve these issues. The question is whether such parameters exist and, if so, whether synthetic generators such as the Graph500 control these parameters. In particular, we ought to test the impact of vertex identities, because the Graph500 openly acknowledges these as important.

The Graph500's Kronecker model is the subject of much statistical inquiry, which can motivate our parameter search. For example, Seshadhri et al. have shown that in a Kronecker model of $|V|$ vertices, some fraction of vertices have no edges in expectation[110]. These zero-degree or *isolated* vertices manifest as gaps in the vertex identity space, as sparsity in vertex-keyed dense arrays, and therefore as potential prefetcher misses. The isolated vertex count is a complex function of $|V|$, $|E|$, and the Kronecker model's probability distribution (see Chapter 6, Section 6.2). As we discussed in the previous section, large zero-degree vertex counts are also common in real datasets. Do isolated vertices substantially affect benchmark results, and do evaluations account for this when using Kronecker graphs, or natural graphs such as the Twitter network?

Depending on the answers to these questions, the plots and conclusions of our field may be distorted. If expected behavior varies in a complex and substantial way across real graph datasets, or with the tunable parameters of synthetic models, then experimenters who are unaware of these effects will find alternative explanations as to why their observed performance varies with the input. A sub-linear distortion in a scale plot may be welcomed as a sign of desirable scaling properties and evidence of good system design.

Most pernicious is that these distortions motivate or discriminate against different system designs. For example, the choice between dense and sparse vertex arrays partially depends on how many isolated vertices are in common test graphs and synthetic models. Two systems with different array implementations might exhibit cache and performance differences according to the isolated vertex model embedded in the test data. But more likely, because system builders are fully aware of the standard test data and profile with it regularly, the "inferior" choice will never pass the prototype stage. Similarly, if popular test data ships with optimistic vertex identities, then *optimizing* vertex identities to improve caching and prefetching will seem unrewarding, when in reality it may be an important contributor to good performance.

5.3 Evaluation

In this Section we show that some of these graph parameters substantially impact the performance of the most popular benchmark algorithms, and that they are not adequately controlled in the most popular test graphs or the Graph500 Kronecker implementation. Furthermore, different algorithm implementations, and consequently systems, respond to these parameters in different ways. Given so many dimensions, the scope of such an analysis is potentially massive, so we will limit ourselves to a small subset of systems, algorithms, datasets, and parameters. This is sound, because we want to show that problems occur in popular and plausible cases and not necessarily in every possible case.

5.3.1 Setup

We investigated the performance impact of vertex identities and isolated vertices in popular datasets and Kronecker graphs on PageRank, breadth-first search, and triangle counting using the Galois system. We use Galois because a recent high-quality study by Satish et al.[108] found that it performs most closely to hand-optimized native code. Giraph and GraphLab are more popular, but both underperform Galois by orders of magnitude on most benchmarks. Galois is nevertheless a comparative target in several evaluations, so an impact on Galois should have broader implications than one publication. Furthermore, Galois is an older system, so its library of algorithms and implementations is extensive and includes "best performance" implementations specifically indicated for benchmarking[67]. This helps guarantee that comparisons between implementations are "fair" and include systemic best practices.

PageRank, breadth-first search, and triangle counting are all popular benchmarks that are worth evaluating. We prefer breadth-first search to shortest paths, because the most popular datasets do not have natural edge weights, and because BFS is the Graph500's flagship benchmark. We omit connected components, because common benchmark implementations have already faced heavy criticism from McSherry's COST study[89]. In the following Sections we first discuss each benchmark individually, and defer discussion of datasets, vertex identities and isolated vertices, etc., to the Conclusions. This is because each benchmark has unique concerns, whereas datasets and vertex parameters are best understood as overarching themes.

For datasets, our primary constraint is that Galois is not distributed. Fortunately, the five most referenced datasets either fit in the memory of a commodity server or are synthetic generators. We omit the LUBM generator, because in principle and in practice it is tightly coupled with the LUBM queries and not normally used for other benchmarks. Four datasets is the mean average in our corpus, but as we have noted, this is not enough samples to properly generalize. We reiterate that the purpose of our evaluation is not to generalize, but instead to show that problems can occur in plausible cases.

Default order and random order are the most important vertex identities to consider. Both occur frequently in published work; the former as the de facto standard for real graphs, and the latter as a correction for the Kronecker model's "optimistic" default order. Correspondingly, we also look at random orders of real graphs to test whether their defaults are optimistic. A random order is certainly "pessimistic" from the cache prefetcher's perspective and from a theoretical bandwidth and cut-minimizing perspective, although it can do a good job balancing implicit parallel loads.

Optimistic orders are, in general, the product of a correlation between the graph's structure and the order in which the graph is generated; this is obviously the case in the Kronecker model (see Chapter 6, Section 6.2). Web crawls and similar processes also exhibit this correlation if the crawler assigns identities as vertices are discovered, because discovery in a walk is correlated with graph structure. The WebGraph project specifically exploits this by using LLP order and distributes compressed datasets in this order, so we will also look at LLP order for datasets hosted on WebGraph, such as Kwak's Twitter. We also include degree orders, because these are common pre-processor optimizations in e.g., triangle counting implementations (including Galois) and have an interesting history (see Section 5.3.4). We favor ascending degree order because it is more common than descending in practice, but we also include descending in triangle counting for reasons delinated in that section. Our degree sorts are stable with respect to each graph's default order and are therefore not wholly independent of it; this is consistent with degree sorts in the wild.

Many graph datasets are distributed with isolated vertices, so our most important test cases are the default distribution and the "packed" case where we ellide every isolated vertex. We ensure a stable packing, i.e., we preserve the order between non-isolated vertices. Isolated vertices are not strictly separable from vertex order because their position affects their impact; for example, randomly distributed isolated vertices have a greater prefetcher impact than if they were all located at the end of an array. Our random order is intended to be pessimistic, so it randomly distributes isolated vertices. Conversely, our optimistic orders (LLP and degree orders) identify and ellide isolated vertices at no additional cost, so they are effectively packed orders. For real datasets in default order, the isolated vertices' positions are unique to that dataset and may be optimistic or pessimistic depending on that particular dataset.

For natural graphs, "missing" vertices may arise from anonymized, redacted, or otherwise hidden data such as an incomplete Web crawl process. The isolated vertex counts of the top 11 datasets are implicitly shown in Figure 5.7 by the difference between the vertex count and maximum vertex identity. In the case of Kronecker model graphs, the edge distribution simply gives some zero-degree vertices in expectation. Using Graph500 settings, from 51 to 74% of a Kronecker graph's vertices are isolated as the scale parameter increases from 26 to 42[110].

The combination of three benchmarks, four datasets, five orders, and packed variants produces many data points, which are further complicated by benchmark-specific parameters and metrics. Many of these combinations are uninteresting, redundant, or impossible (e.g., packed variants of datasets without zero-degree vertices), so we are highly selective as to which data we choose to present. The data presented in the following experiments are summarized in Figure 5.11.

Problem	Datasets	Varies	Measures
PageRank	twitter-2010, soc-livejournal	algorithms, isomorphisms	runtime
Breadth-First Search	twitter-2010	isomorphisms, 0-degree vertices	runtime, l2 and l3 cache miss ratios
Triangle Counting	soc-livejournal, cit-patents, kronecker models	isomorphisms, 0-degree vertices	runtime, work, load balance

Figure 5.11: Summary of data presented in the following experiments.

Our benchmark machine is the same 6-core Intel i7-970 at 3.20GHz that we used to evaluate Sheep in Chapter 3, Section 3.5. However, we upgraded to 24GB of RAM, to support the Twitter graph in-memory, and to a Samsung 850 Pro SSD. All graphs were converted beforehand to Galois' binary input format, and all benchmarks were run with a hot disk cache, so, in principle, the SSD should have no significant impact. We used the most recent Galois 2.3.0-beta release, because it has the most algorithm implementations, and we did not encounter any correctness or obvious performance issues. We used 12 threads, because it gave the best end-to-end runtime, and otherwise we used default settings wherever they are not explicitly specified.

5.3.2 PageRank

Figure 5.12 depicts PageRank running on Galois to within 0.01 tolerance for four different vertex orders (or equivalently, labeled isomorphisms) of the Twitter-2010 dataset. The y-axis is runtime and the x-tics are various PageRank implementations that differ primarily by their scheduling idioms. In particular, "synchronous" schedules update each vertex once between barriers (as in the Bulk Synchronous Parallel model), whereas "asynchronous" schedules update any vertex when new data is available. We use tolerance instead of fixed iterations, because it gives a definition of end-to-end runtime that is comparable between synchronous and asynchronous implementations.

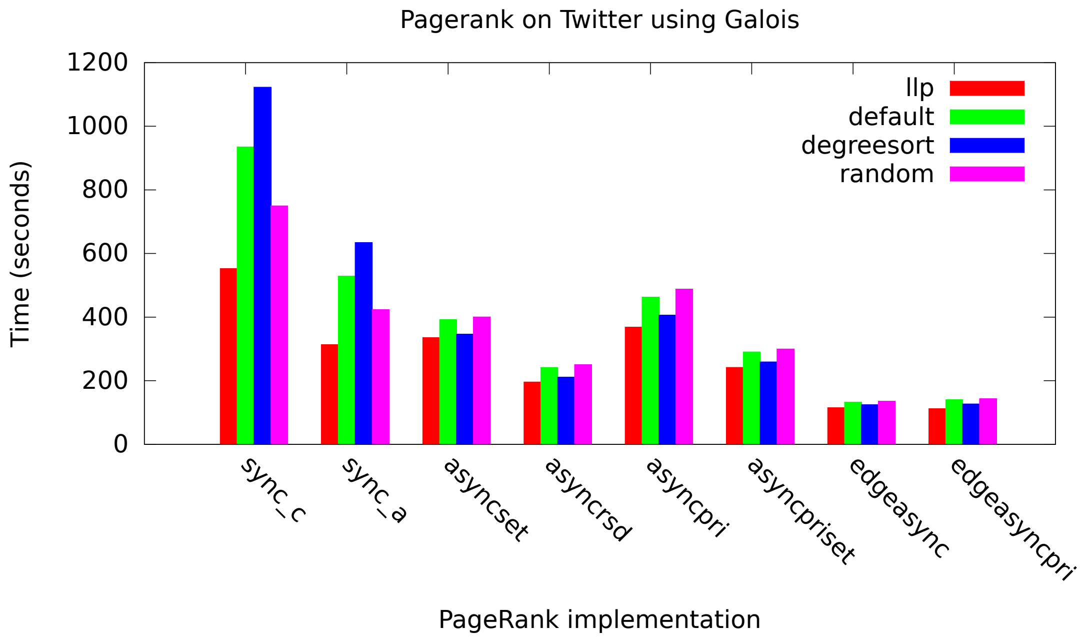


Figure 5.12: Performance of PageRank on Twitter using Galois with different implementations and isomorphisms. Synchronous implementations (left) wait on a global barrier between each PageRank iteration. Asynchronous implementations (right) update vertex ranks as soon as new data is available.

Many critical facts about graph systems benchmarking are visible in this figure. First, the scheduling idiom has an enormous performance impact, so researchers' interest in this is justified. Second, the isomorphisms' impact ranges with the implementation between a factor of 2.03 and 1.18, worst isomorphism over best. This is a wide range, but even on the low end it is a substantial impact. Finally, the *relative* performance of the isomorphisms differs between synchronous and asynchronous implementations.

Synchronous PageRank implementations (the leftmost two x-tics) are so common in contemporary graph systems evaluations that *the majority of evaluations in our metastudy define and measure PageRank in terms of a fixed synchronous iteration count*, even though this is an implementation-specific detail. 21 out of 37 PageRank evaluations in our metastudy explicitly measure a fixed count of at most twenty iterations. This discourages comparison with systems that research asynchronous execution models (the rightmost six x-tics), even though they may converge more quickly.

For example, PowerGraph (2012) was one of the first vertex programming systems to support asynchronous execution, and its authors discuss and evaluate asynchronous PageRank[45]. However, in their comparative evaluation they only benchmark synchronous PageRank, because prior works measured and published "per-iteration" runtimes that are defined only for synchronous implementations. Similarly, the authors of Galois discuss and evaluate "data-driven" asynchronous execution, and even reimplement PowerGraph's asynchronous engine (2013)[96]. However, they compare only synchronous per-iteration PageRank results.

The authors of GraphMat (2015), an iterative SpMV system, compare against PowerGraph and Galois-2.2.0 using synchronous PageRank and conclude GraphMat is 2.6 faster per-iteration than Galois[116]. This factor is comparable to the difference between the best synchronous and asynchronous implementations in the more recent Galois-2.3.0-beta [*]. Galois outperforms GraphMat on every end-to-end runtime measure but underperforms on every per-iteration measure, from which GraphMat's authors conclude that, on average, they outperform Galois; this claim is almost entirely due to synchronous PageRank results. To be absolutely clear, GraphMat's evaluation followed conventional best practices and was consistent with prior work. Our concern is that *consistency with prior evaluations has brought the field to a state where the most popular comparative benchmark discriminates against alternative and arguably superior solutions.*

It takes 71 iterations for the synchronous PageRank implementations to converge within 0.01 tolerance in Figure 5.12. However, as previously mentioned, the majority of PageRank evaluations in our metastudy benchmark a constant iteration count that is never more than twenty. PowerGraph's authors note that their system needs *at least* twenty iterations of PageRank on Twitter before their most sophisticated partitioner produces any end-to-end runtime improvement. So the field-wide

[*] Galois did not include a *native* asynchronous PageRank in version 2.2.0, but it did support PowerGraph's asynchronous implementation. The authors of GraphMat similarly evaluate against synchronous PowerGraph results.

perception of PageRank as a per-iteration benchmark, rather than an end-to-end benchmark, also misrepresents preprocessing steps (such as vertex sorting) that in practice give real speed-ups (as shown here). For these reasons, *we strongly recommend that future work should, at minimum, measure end-to-end PageRank runtime for a target tolerance, instead of per-iteration runtime or a fixed iteration count*. It would be even better to measure tolerance over runtime and error against a canonical solution.

When we consider vertex orders, the situation becomes even more complicated. Varying the order also varies the performance by as much as 2.03*x* for Galois' synchronous PageRank implementations, whereas its asynchronous implementations vary by up to 1.18*x*. For synchronous implementations, the "default" isomorphism runs up to 1.69*x* slower than the "LLP' isomorphism, and both are publicly available from online sources. The default isomorphism comes from the primary source[70], whereas LLP is the default isomorphism of the LAW repository[3] (which also offers the primary source isomorphism as another download). LLP is a bandwidth-minimizing sort algorithm, so one could argue that Twitter's primary source isomorphism is pessimal for PageRank relative to LLP's optimistic baseline.

Most importantly, for the synchronous implementations, randomizing vertices looks like an optimization and ascending degree sort looks pessimal, whereas for asynchronous implementations the reverse is true. In fact, for asynchronous implementations, ascending degree sort is almost as good as LLP's much more complicated topology-aware sort. Depending on the PageRank implementation, either randomization or degree sorting might look like a clever preprocessing step. However, for different graphs, this result differs considerably; Figure 5.13 depicts the same measurements on soc-LiveJournal, the second-most popular benchmark graph. On soc-LiveJournal, randomization is consistently bad, and degree sorting interpolates between random in the synchronous cases and *no better than the default* in the asynchronous cases. In this scenario, LiveJournal's default isomorphism is "good enough" such that simple degree sorting never give an improvement.

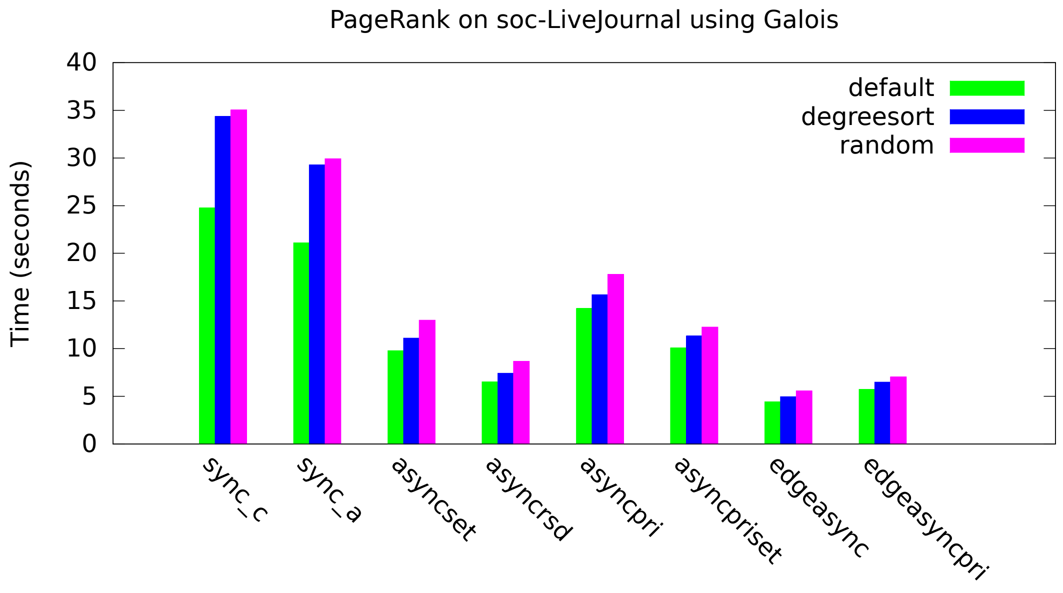


Figure 5.13: Performance of PageRank on LiveJournal using Galois with different implementations and isomorphisms.

Whatever the scenario, it's clear that isomorphisms control PageRank performance in a complicated but substantial way. It is not difficult to imagine how this might affect system design in practice. Consider the case of a system designer who must decide whether or not to hash vertex identities to e.g., improve load balance. At present, if her benchmark is PageRank and her test graph is Twitter in its default order, then hashing looks like a performance improvement for the most common algorithms, and no great loss in any case. But in an alternate reality where Kwak et al. published their Twitter dataset using WebGraph's tools and therefore LLP order, hashing looks like a major performance regression for every implementation! And in another reality where Kwak et al. stored and enumerated their crawl in a degree-sorted data structure, the costs and benefits of hashing are highly variable and implementation dependent. All three cases are misleading, because the vertex isomorphisms are not explicitly controlled in the experiment. We will see more examples of this in

isomorphism	l2 miss ratio	l3 miss ratio	fall-through ratio	runtime
LLP	0.860	0.613	0.527	6.328
degree	0.924	0.605	0.559	6.704
random-pack	0.949	0.636	0.604	6.882
packed	0.933	0.646	0.603	7.003
default	0.916	0.639	0.585	7.211
random	0.925	0.667	0.617	7.643

Figure 5.14: Performance of Breadth-First Search on Twitter using Galois. The "fall-through" ratio is the combined ratio of L2 and L3 misses.

the following experiments.

5.3.3 Breadth-First Search

Figure 5.14 depicts single-source breadth-first search (BFS) running on Galois and averaged over 100 random sources for six different isomorphisms of the Twitter-2010 dataset. BFS runtimes are usually benchmarked end-to-end, so unlike PageRank we focus exclusively on Galois' best BFS implementation. As was the case with PageRank, Twitter's primary source isomorphism is hardly better than random, but simple degree sorting gives an isomorphism that performs almost as well as LAW's LLP algorithm. The two new isomorphisms, "stablepack" and "random-stablepack," ellide zero-degree vertices from the default and random isomorphisms respectively; for Twitter these are 32% of the vertices. *Zero-degree vertices account for about half of the performance difference between the best and worst isomorphisms.*

Since these workloads ought to be nearly identical, their performance differences are best explained by data bandwidth effects. However, if we look at cache metrics we find that the details are complicated. In general, cache miss ratios are high, because graph topology is not trivially serializable, so the cache is the bottleneck and small differences in the end-to-end miss ratio are roughly proportional to end-to-end runtimes. However, for any given specific level there are inversions between its miss ratio and the end-to-end runtime of different isomorphisms. For example, the packed default isomorphim has a lower L2 miss ratio than packed random, but it has a higher miss ratio in L3 and is slower end-to-end. Conversely, degree sort has a slightly lower L3 miss ratio than LLP, but it has a higher miss ratio in L2 and is slower end-to-end. One plausible explanation is that degree sort "fits" in L3 but not in L2 (specifically, that vertex neighborhoods fit in the prefetcher's implicit window), whereas because LLP "fits" in L2, a miss in L2 is likely to also miss in L3.

Zero-degree vertices have a significant performance impact and also distort our perception of cache measurements. One might hypothesize (like all of our office mates) that zero-degree vertices would fragment data structures and disrupt the cache's prefetcher, but for the default order removing zero-degree vertices actually increases the miss ratio. This is because zero-degree vertices are accessed on routine serial scans of vertex arrays, such as initialization, where they inflate the prefetcher's hit ratio with trivial work (e.g., in BFS these values are initialized to zero, but are never encountered in the search). So the miss ratio goes down, but only because we are successfully prefetching meaningless work. *If you want to meaningfully measure the cache hit ratio of a graph algorithm, it is important to remove zero-degree vertices, or else busywork will poison the cache measurements.* RMAT/Kronecker generators such as the Graph500 generator produce graphs with a large number of zero-degree vertices (as much as 74%), so RMAT users need to be particularly aware of this effect.

5.3.4 Triangle Counting

Triangle counting is unique among graph benchmarks in that largely by accident it sometimes has an isomorphic control standard. The most popular triangle counting algorithms implement fast edge set intersection by pre-sorting the graph's edges in some vertex order. Chiba and Nishizeki[30], and later Thomas Schank[109], both describe algorithms that use degree order. Schank also gives a complexity bound for degree order, and his "forward" algorithm is implemented and cited by many

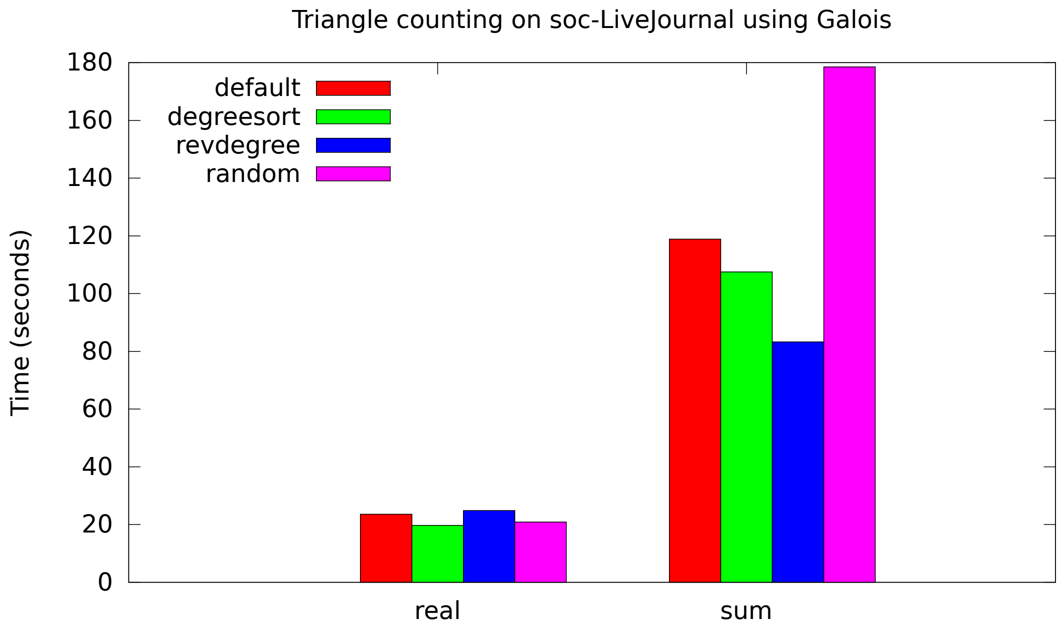


Figure 5.15: Triangle counting on soc-LiveJournal using Galois. The "real" time is end-to-end, whereas the "user" time is the sum of all unhalted thread runtimes (i.e. a measure of total work).

systems such as PowerGraph, GraphChi, Galois, GraphX, and LLAMA. Oddly, Schank's proof relies on descending degree order, but in practice this convention is reversed and systems sort by ascending degree. PowerGraph and GraphX notably reuse the graph's input isomorphism instead of degree sorting.

We must emphasize that descending degree sorting gives a loose upper bound and is not an "optimization." If a graph's natural order is well-suited to triangle counting then degree sorting is a waste of time, producing a performance regression. In this sense, triangle counting and degree sorting are no different than any other algorithm and isomorphism in this study, except that Shank's proof establishes an explicit relationship between the two in the standard cache-oblivious complexity model. This explains why some real-world frameworks abandon the degree sorting step.

Figure 5.15 shows triangle counting running on Galois for four different isomorphisms of the soc-LiveJournal dataset; packed isomorphisms are not included because soc-LiveJournal is already packed. In addition to previous isomorphisms, we included descending (reverse) degree for consistency with Schank's dissertation. In this context it seems that ascending degree sort is the wiser convention, as it outperforms descending sort by 1.26*x*. However, these results also invert our prior experience with soc-LiveJournal: the default isomorphism is poor and random performs well, whereas the opposite is true for PageRank and BFS.

An explanation emerges if we look at the sum of individual threads' unhalted runtimes, i.e., time spent executing in user mode. This is a simple measure of the total amount of work done by the thread group. Descending degree order substantially reduces this work compared to ascending, and the default isomorphism is substantially better than random. Random order uses 1.5*x* more thread time than the default isomorphism even though it is 1.13*x* faster end-to-end. Ascending degree order just happens upon a sweet spot that balances efficiency and parallelism in practice. Schank's order may be best for his serial algorithm, but the parallel framework introduces new concerns, and in this implementation they are partially controlled by the isomorphism. This is a concrete example of system builders gravitating towards an algorithmically sub-optimal sort order because of the isomorphism's practical system effects.

Triangle counting is essentially a read-only workload, so load balance is the main factor that controls parallelism. Each thread inspects the 2-hop neighborhood of some disjoint vertex set, and this induces per-thread subgraphs of varying size. If the vertex array is partitioned in blocks (as is the case in most loop-parallelizing systems), then the induced subgraphs are strongly determined by the isomorphism. Reverse degree order places the vertices with the most neighbors in the first block and consequently induces a large subgraph for the first thread. Conversely, ascending degree order places these vertices last and minimizes their associated subgraph. Unsurprisingly, random partitions also improve the load balance.

Note that if one's goal is to ”maximize parallel work” then the random isomorphism is consid-

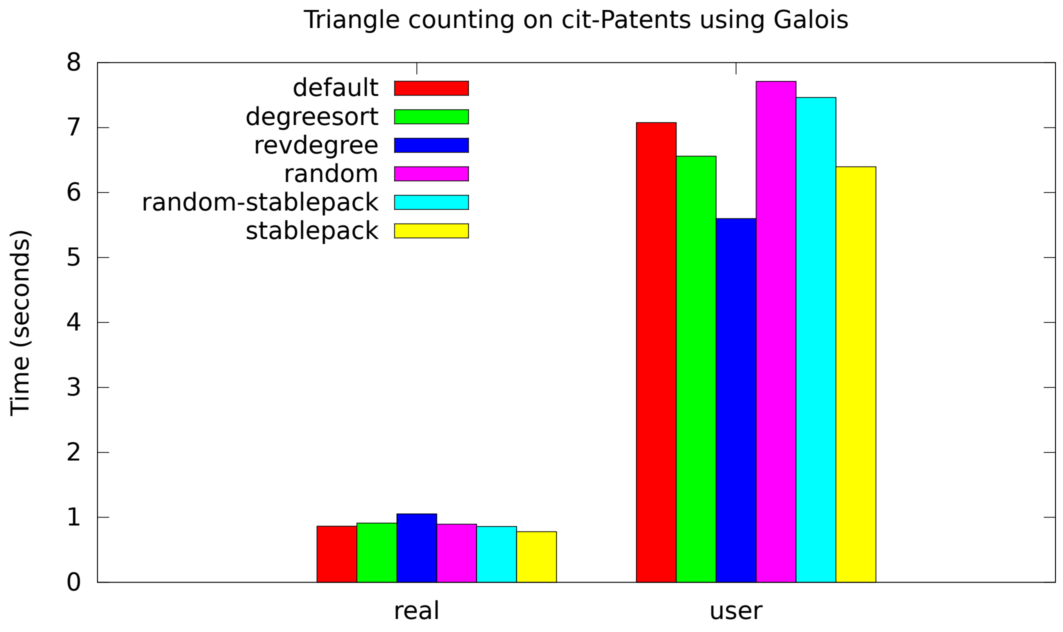


Figure 5.16: Triangle counting on cit-Patents using Galois.

erably better, but only because it does considerably more work to solve the same problem in the same end-to-end time. These kinds of consequences and pitfalls are normally associated with system and algorithm designs, yet we are only considering different input vertex numbers! In the field we see explicitly optimistic isomorphisms (LLP), unknown-but-not-random isomorphisms (e.g. "natural" IDs, which may be optimistic or pessimistic), genuinely random isomorphisms (synthetic graphs and anonymized datasets), and in the case of triangle counting, degree sorted isomorphisms for some implementations (but not all, e.g. GraphX). If your evaluation doesn't control these many variations, then you will have to find some other explanation for the variation in your results.

For example, GraphMat's authors claim that Galois outperforms them at triangle counting on soc-LiveJournal by about 20%; but GraphMat uses the default isomorphism, Galois uses ascending degree, and the difference between these two isomorphisms running on Galois is also about 20%. GraphMat and Galois are very different systems running necessarily different triangle counting im-

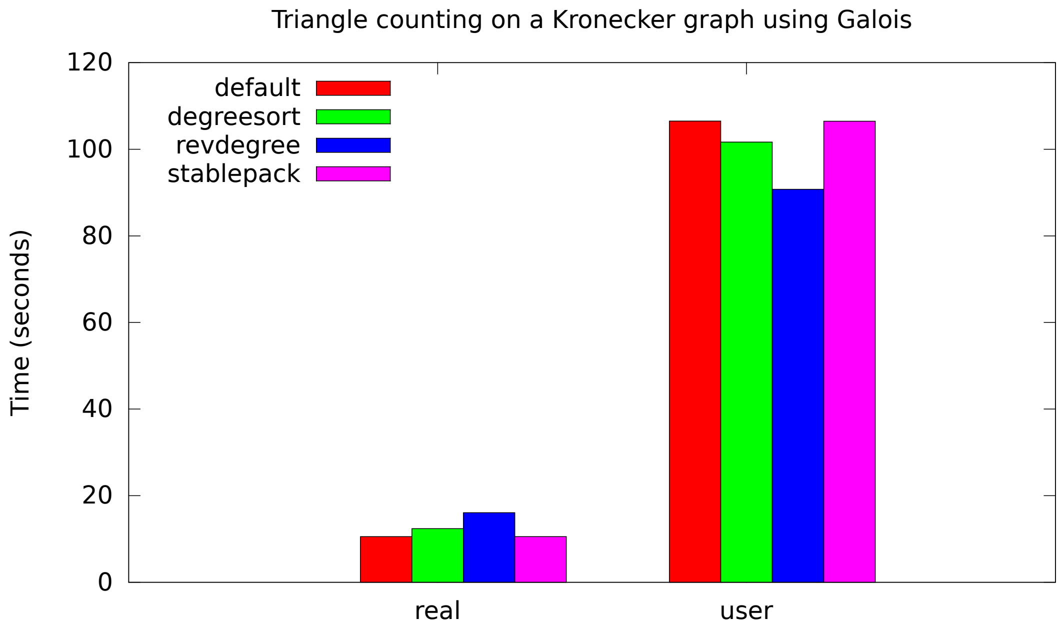


Figure 5.17: Triangle counting on a Kronecker model using Galois, where $s = 22$, $e = 16$, $A = 0.45$, $B = C = 0.15$

plementations, but it's unclear whether we can attribute their performance difference to the system, the implementation, or soc-LiveJournal's isomorphisms. We want to emphasize that this comparison is only possible because GraphMat and Galois are neck-and-neck as two of the fastest systems in this field; if their performance differed by an order of magnitude, then the isomorphic impact would also differ. The impact might be buried, or it might balloon, as was the case with synchronous vs. asynchronous PageRank.

Ascending degree sort is not strictly a triangle counting "optimization." As was the case with PageRank, different graphs vary these results. Figure 5.16 depicts triangle counting on the cit-Patents dataset; this plot includes packed isomorphisms because cit-Patents is not packed by default. Here we see that degree sorting is a waste of time; the same overall work reduction is achieved by simply elliding the zero-degree vertices. That is, excepting the zero-degree vertices, the natural isomorphism is better in every way than ascending degree.

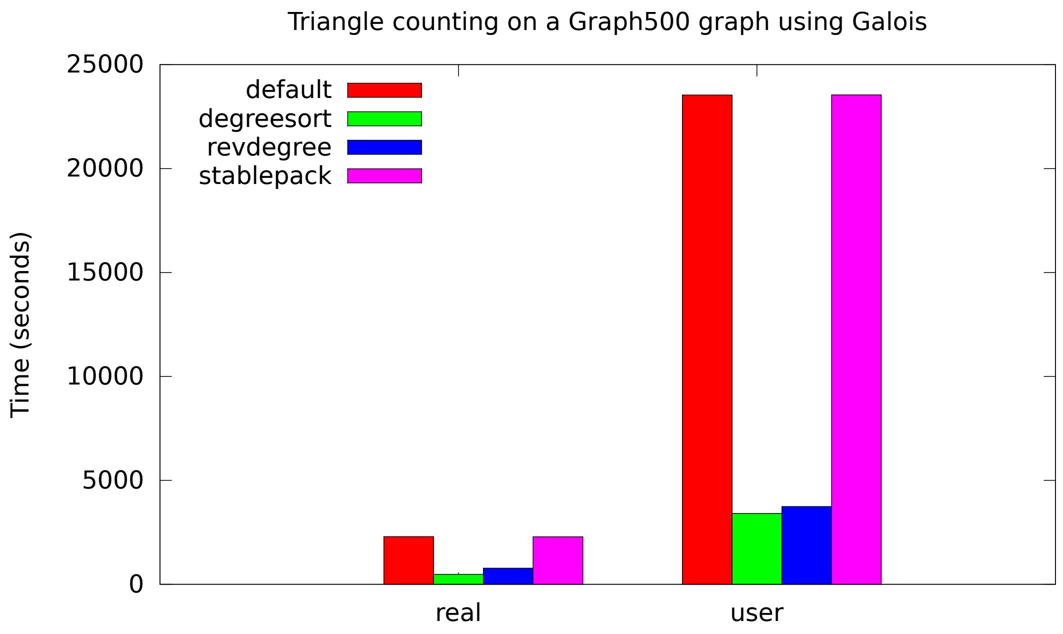


Figure 5.18: Triangle counting on Kronecker model using Galois, with Graph500 default parameters ($s = 22, e = 16, A = 0.57, B = C = 0.19$.

Random can also outperform ascending degree in common scenarios; Figure 5.17 depicts triangle counting on a Kronecker graph with parameters $s = 22$, $e = 16$, $A = 0.45$, $B = C = 0.15$. These parameters are used by Sundaram et al., because the default Graph500 parameters generate graphs with enormous triangle counts. Here we see that ascending and reverse degree are both performance regressions compared to the synthetic graph's default random order. This is not true for all Kronecker graphs; if we use the Graph500's default parameters, then both degree sorts give a tremendous performance improvement (Figure 5.18). Note also that 50% of the vertices in this graph are isolated, but packing them has no performance impact at all; because the isolated vertices are randomly distributed by the isomorphism, they do not imbalance the induced thread partitions, which are the dominant factor in parallel triangle counting performance.

Given that the isomorphism induces partitions, we must accept that partitioning is hard and de-

scalable system	cores	twitter	uk-2007-05
GraphLab	127	242s	714s
GraphX	128	251s	800s
Single thread (SSD)	1	153s	417s
Union-Find (SSD)	1	15s	30s

Figure 5.19: Performance of connected component algorithms: label propagation (GraphLab, GraphX, and single-threaded) versus union-find. Reproduced from "Scalability! But at what COST?"[89]

gree sorting is a naive heuristic. Without an isomorphic standard, what we're measuring in triangle counting is, to a large extent, the quality of an unspecified partitioning. Therefore, if you want to measure any other effect, it is important to control the isomorphism. Systems that degree sort are partially controlled for their own intra-comparison, but there is no inter-comparative standard for published results. However, this control is only partial because degree order is not a strict order.

5.3.5 Connected Components

In our corpus, the connected components benchmark is frequently implemented using label propagation. The problems with this benchmark are discussed extensively by McSherry[89], so we will only summarize his findings. Union-find algorithms are over 40 years old and dramatically outperform label propagation both in theory and in practice (see Figure 5.19, reproduced from McSherry's paper). In particular, this is a major reason that the Sheep partitioner, which is based on a modified union-find, is faster than other partitioners that are based on explicit partition labels (Chapter 3). However, label propagation is more intuitive than union-find when written as a vertex program. McSherry's proposal is that distributed systems should compare end-to-end with a single-threaded hand-optimized baseline implementation, which they should obviously outperform. This call for an objective *end-to-end* standard is similar to our own proposal for PageRank.

5.4 Conclusions and Advice

The previous Section covers a small sample of many different cases in which the underlying principles are complex. This is, in fact, what we want to show: evaluations that do not control these factors are chaotic, and all we've done is expose that fact. Nevertheless there are some common principles that can guide us towards better practices for the field.

5.4.1 Vertex Isomorphisms

As we have shown, uncontrolled vertex isomorphisms have a performance impact of up to 2*x* on the most popular benchmark (synchronous PageRank) on the most popular dataset (Kwak's Twitter) using a popular high-performance comparative target (Galois). In other cases about 1.2*x* is common, which is well within the range of reported differences between targets in comparative evaluations. Papers in our corpus use less than four datasets on average, and those datasets are not randomly chosen. Given these numbers it is hard to imagine that uncontrolled vertex isomorphisms have not shaped reported results.

It is tempting to dismiss vertex order as a "feature" of the dataset and sorting as an "optimization" conditioned on that feature. Certainly, graphs have other features (such as their mixing rate in a random walk) that are difficult to control but may be optimized by graph systems (e.g. by prioritized vertex scheduling). One key difference is that prioritized scheduling is a novel and interesting optimization to measure, whereas vertex sorting is decades old. But more importantly, the existence of countless uncontrolled variables is not an excuse for failing to control for isomorphism, which is easy to do and has a quantifiable impact.

When a hidden variable is uncontrolled, its effects may appear linked to explicit variables such as the competing systems. In the case of triangle counting this effect is obvious: different implementations adopt different input sort orders and some of their performance differences are caused by this. However, conceptually and quantitatively the exact same thing is true of every algorithm in this study, even though the mechanisms are less obvious. Different implementations adopt practices that favor some isomorphisms over others and ultimately cause some of their performance differences.

The field needs standard isomorphisms. Ideally one would draw a significant number of samples from "realistic" isomorphisms, but as present we have no model to define this. The Graph500 adopts random as its standard, but insofar as random is pessimal this may exagerate the impact of sorting as a preprocessing step. Conversely, optimistic standards such as LLP discourage preprocessor research by delivering an *a priori* preprocessed graph.

In our opinion *random and ascending degree are the best isomorphisms for comparative benchmarks.* Our study shows some cases where random is not strictly pessimal, and absent this pessimal-optimal distinction, random is close to an ideal control. Degree sort, precisely because it is an established trick, is simple enough that any system can do it but naive enough that it can be improved upon. Most importantly, two different standards would create an opportunity for systems to measure and show how they respond to different isomorphisms. But in practice studies may not want to benchmark multiple isomorphisms; in this case we recommend random for studies that use Kronecker/R-MAT data and ascending degree sort for studies with a triangle counting benchmark. At minimum, *studies should declare their isomorphisms when they describe their experiments.*

This proposal discriminates against systems that use the "natural" isomorphism under the assumption that it is optimistic. This seems to be true of e.g., PageRank on soc-LiveJournal, and in specific applications it may be a good practice. However, as previously mentioned the pessimal-optimal framework breaks down across different cases, e.g., soc-LiveJournal's default is poor for triangle counting. This assumption also does not hold for other datasets, such as Twitter. Therefore, for benchmarks using the default isomorphism would only be reasonable if the field had a large sample of realistic benchmark datasets.

5.4.2 Isolated Vertices

In terms of performance, isolated vertices are not as dramatic as vertex isomorphims, although in some cases such as breadth-first search they have a meaningful impact. A 10% difference is not utterly trivial and will become large, e.g., if we sample many BFS trees to approximate betweenness centrality. The more important consequence of isolated vertices is that the reported vertex counts of graphs are unreliable. Because isolated vertices impose little work relative to non-isolated vertices, they mask the "real" vertex count and therefore the scale and edge density of the graph.

For example, soc-LiveJournal's vertex and edge counts are closely approximated by Graph500 graphs with a scale parameter of 22. However, soc-LiveJournal has no isolated vertices, whereas Kronecker graphs have isolated vertices determined by a complex formula of their parameters. With the Graph500's default parameters, 43% of the vertices are isolated and the average degree is roughly doubled at a scale of 22. A more appropriate scale for comparison with soc-LiveJournal would be 23 with half the edge factor. Therefore, even when papers follow a consistent standard for vertex counts, if it includes isolated vertices it leads to incomparable counts between datasets. In practice, though, it is not difficult to find inconsistent vertex counts. For example, Sundaram reports Twitter including its isolated vertices, but reports Friendster without[116].

"Scale" parameters, such as the vertex and edge count, do not wholly determine performance and, like isomorphisms, are only one of an infinite number of graph parameters. However, like isomorphisms they can and should be controlled, and this is particularly important because "scalability" is such a popular idiom. Isolated vertices in the Kronecker model are particularly tricky because they are nonlinear with respect to the model's "scale" parameter; therefore, a plot of Kronecker graphs with scale on the x-axis is nonlinear with respect to functions of the vertex count, such as the average degree. We ourselves made this error in our own published research[83], and we discuss it further in the next Chapter.

5.4.3 Datasets

The field does not use enough datasets. If there exist other graph parameters that are as substantial as isomorphism, but more subtle to express or difficult to control, then we have no obvious remedy. A healthy sample of graphs from a robust population should be the panacea for any such hidden variables, but in practice we are far from achieving that standard. This is a social problem, because datasets are obtained through social processes. The social nature of this problem is reflected in the structure of the dataset citation graph (Figure 5.9).

It is conspicious that many of the top datasets are over a decade old. To some extent this is due to citation pressures, but we would argue that this also predates the modern "Big Data" movement and the widespread perception that such data is valuable. It also predates the Netflix Prize lawsuit and other indicators that releasing such data entails risks. Mitigating these concerns is not obviously possible and is certainly outside the scope of this dissertation.

In the absence of a robust data corpus we must rely on synthetic data generators such as the Kronecker model. Such results generalize only to the model itself, so we in turn depend on the model to generalize across "realistic" graphs. However, the most popular generators and also their parameters are subject to the same citation pressures as real data. The Kronecker model has seen numerous extensions (e.g., to arbitrary $k \times k$ seeds[75] and mixtures of different seeds[66]), but the field still uses the same 2×2 model that Chakrabarti pioneered in 2004[28]. Similarly, the parameter space is dominated by the Graph500's settings and a few alternatives when these settings are intolerable, such as in triangle counting. As we've shown, these parameters can vary performance characteristics (of e.g., isomorphisms) in the same manner as varying real datasets. So at present, our synthetic data situation seems almost as dire as our real data. Nevertheless we think this is the best route forward, so in the next Chapter we discuss improvements to the Kronecker model with a specific focus on generating benchmark graphs.

6

Smooth Kronecker Models

As shown in the previous chapter, graph systems evaluations are concentrated on a handful of publicly available datasets. This causes generalization problems, because "hidden" graph parameters, such as the vertex isomorphism, are badly undersampled. These variations would wash out as noise if significantly more benchmark data were available and in use. However, stakeholders are unlikely to publish data due to its business value and the significant liability risks. Fortunately, we have one inexhaustible source from which to draw benchmark datasets: synthetic graph generators.

The Graph500 and LUBM graph generators are by far the most popular generators for research evaluations, as revealed by our metastudy. Of these, the Graph500's Kronecker generator is more popular and more widely used by a variety of research fields, in contrast with LUBM which is tightly bound to the RDF query model. This prompts a critical question: *is the Kronecker generator a suffi-*

cient source of benchmark datasets for research evaluations? Unfortunately, the answer is no: we will show in Section 6.2 that there are several features of the Kronecker generator that inhibit its correct use as a benchmark.

Seshadhri et al. identified some of these problems in 2011[110], proposing a fix that was never integrated into the Graph500 specification and was implemented incorrectly in the generator's reference code. This is inconvenient for benchmarks, because it adds noise on a per-graph basis, so benchmarks must run on a series of graphs to reach statistical convergence. We show that this noise paradoxically improves the graph's partitions and therefore reshapes the relative value of partitioning schemes.

We present Smooth Kronecker, a Kronecker generator that fixes the problems identified by Seshadhri et al. without adding per-graph noise. Smooth Kronecker works by resampling the generator's discrete distribution parameters into several distributions whose dimensions are relatively prime to one another. By randomly substituting one relative prime distribution for another, we smooth the generator function in the same manner that adding noise would blur it. However, our distributions are deterministic samples from the same underlying distribution, so unlike noise, they do not vary the graph model and preserve its properties.

6.1 Background

Generative graph models occupy prestigious roles in the history of graph theory, but in practice these roles have changed over time. As mentioned in Chapter 2, Erdos' random graph model was enormously productive both in theory and in practice; for example, Erdos' proved a random graph is almost certainly connected if its edge probability is greater than $ln|V|/|V|$[38], and because this is the mean-field case for bond percolation theory[23] it has implications for e.g., materials science. In contrast, although later models such as Barabasi's preferential attachment model successfully describe select properties of contemporary datasets, such as centralization and fault tolerance[5], they have failed to produce a holistic "complex network" theory with widely-applicable conclusions.

This is in part a consequence of opposing research directions. Erdos defined a powerful but general mathematical model that logically must have real-world consequences, whereas Barabasi et al. observed empirical features of contemporary networks and then quantified those features in a formal model. Though both models are statistical in the sense that they describe a class of graphs using random variables, in Erdos' model these variables are random *constructions*, whereas in Barabasi's model they're parameters that fit real *observations*.

Chakrabarti et al.'s seminal "R-MAT: A Recursive Model for Graph Mining"[28] takes the same approach as Barbasi et al. The purpose of this work is clearly stated in its title: R-MAT extracts knowledge from real graphs by fitting them to a stastical model's parameters. R-MAT models the graph's square adjacency matrix as a probability distribution over its four square quadrants. To generate an edge, R-MAT chooses a quadrant from this distribution and then recursively reapplies the distribution to that quadrant. Naturally, k such recursions model a graph with 2^k vertices, and each recursion generates one bit of the source and target vertices. Intuitively, the diagonal quadrants model two independent vertex partitions, and the anti-diagonal quadrants model the edge cut between those partitions. When the diagonal has more mass than the anti-diagonal, then R-MAT is a "planted partition" or community model.

R-MAT is a simple example of a *stochastic block model*, a broad field of logical extensions to Erdos' model that have enjoyed much attention in recent years. In practice R-MAT is widely used precisely because it is simple: the distribution is intuitively meaningful and its parameters are easily published, exchanged, and reproduced. Furthermore, when the distribution is derived from real graph data (as its authors intended and demonstrated), then researchers can claim that synthetic graphs drawn from the distribution are "like" the real graph and therefore valid experimental data. In particular, this gives a credible method to vary the size of graphs in an experiment and maintain their inter-comparability. R-MAT can generate very large graphs because its large adjacency matrix is not explicitly materialized, and each edge may be independently generated in parallel. Given these features, it is unsurprising that DARPA'S HPCS supercomputing group chose R-MAT as the input generator for its SSCA#2 scalable graph analysis benchmark specification in 2005[12]. This specification evolved into the "HPC Scalable Graph Analysis Benchmark" in 2009[34] and then the Graph500 supercomputing benchmark from 2010 onwards[92].

A prestigious group including Leskovec, Chakrabarti, and Kleinberg refined R-MAT in several publications between 2005 and 2010[74 76 75]. Their new *Kronecker model* characterized R-MAT's recursive step as the Kronecker product of the distribution matrix with itself, and the recursive series as a Kronecker exponentiation process; the distribution of graphs generated by this process are the Kronecker graphs. These extensions provide for the use of distributions with dimensions other than 2×2, and also give a more rigorous procedure to fit the model's parameters to real graph data. Most importantly, the Kronecker model is a strict superset of R-MAT and so existing R-MAT applications could adopt the Kronecker model without changing their behavior. Hence, the Graph500 now uses a "Kronecker" model even though its implementation is fundamentally similar to SSCA#2's original R-MAT model. Figure 6.1 depicts a 2×2 Kronecker model, which is equivalent to the R-MAT model.

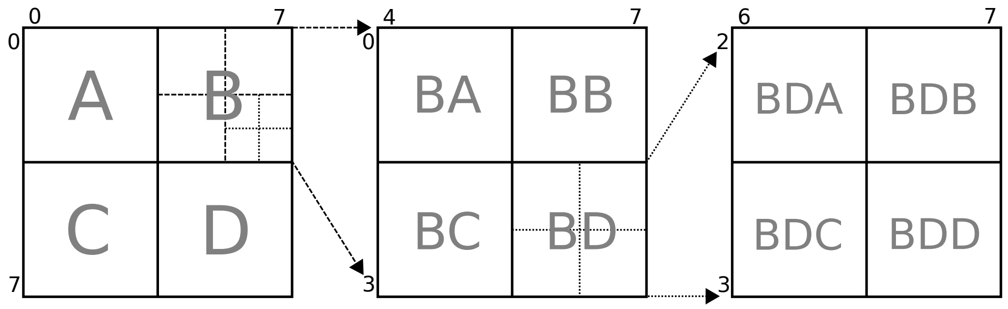


Figure 6.1: A Kronecker model using a 2×2 seed with 3 recursions. To generate an edge, we draw from the 2×2 distribution 3 times to uniquely determine a source and target in the range $[0 : 2^3 - 1]$. Each iteration of a $d \times d$ seed generates one d-ary bit of the source and target.

Source	A	B	C	D	k	$\lvert E\rvert$
Graph500[92]	0.57	0.19	0.19	0.05	varies	$16 * 2^k$
CAHepPh co-authors[75]	0.42	*0.19	*0.19	0.20	14	237010
WEBNotreDame web graph[75]	0.48	0.20	0.21	0.11	18	1497134
Triangle benchmark[108]	0.45	0.15	0.15	0.25	20	$16 * 2^k$
SSSP benchmark[96]	0.50	0.10	0.10	0.30	24	$16 * 2^k$

Figure 6.2: A sample of Kronecker parameters from various evaluations. In general benchmark Kronecker graphs are characterized by a strongly connected upper-left vertex partition (A) and a weak anti-diagonal (B and C). Therefore, depending on the strength of the lower-right partition (D) they are either planted partition (strong) or planted community (weak) models.

Notably, Leskovec also maintains the Stanford Large Network Dataset Collection, which is de facto among the most popular sources of real graph data used in research evaluations[77]. Much of Leskovec's research mines knowledge from these graphs, and R-MAT was originally advertised as a graph mining model; but Leskovec's graphs are also widely used as benchmarks, and the R-MAT/Kronecker models are widely used as benchmark generators. In practice graph mining and graph construction are quite close because "realistic" graph construction depends on a definition of what is "real," which graph mining provides.

In particular, graph mining provides parameters for the Kronecker model's distribution. When reseachers use the Kronecker model to generate experimental data, they must control its parameters. It is reasonable to vary and plot size parameters, such as vertex and edge counts, but the distribution's parameters are more challenging. It's not clear whether every choice of parameters is "valid" in the sense that they produce graphs "like some real graph," or that the parameters vary in a principled manner that one could feasibly map to a plot axis. In practice, researchers choose fixed parameters from published results, which provides legitimacy and enhances comparability. The most popular parameters are overwhelmingly the Graph500's defaults, but several other parameters are also in circulation (see Figure 6.2). Notably these are all 2×2 distributions, so the original R-MAT model is very much alive in 2017.

6.2 Benchmarking Issues

Unfortunately, the Kronecker graphs exhibit several features that affect their validity as benchmark datasets. In the previous Chapter we identified two features – isomorphisms and isolated vertices – that are commonly mishandled in graph systems evaluations. With regards to both these features, the Kronecker model presents unique experimental challenges. The model's default isomorphism is highly optimistic, and for that reason the Graph500 generator explicitly outputs a random isomorphism, which is instead highly pessimistic relative to the default isomorphisms of real datasets. The model also generates isolated vertices in proportion to a complex function of its parameters, which means the real vertex count is poorly represented when those parameters are e.g., a plot axis. Most importantly, the model produces a degree distribution that is dramatically unlike any real dataset, with implications for benchmarks that depend on the degree such as triangle counting (Figure 6.3).

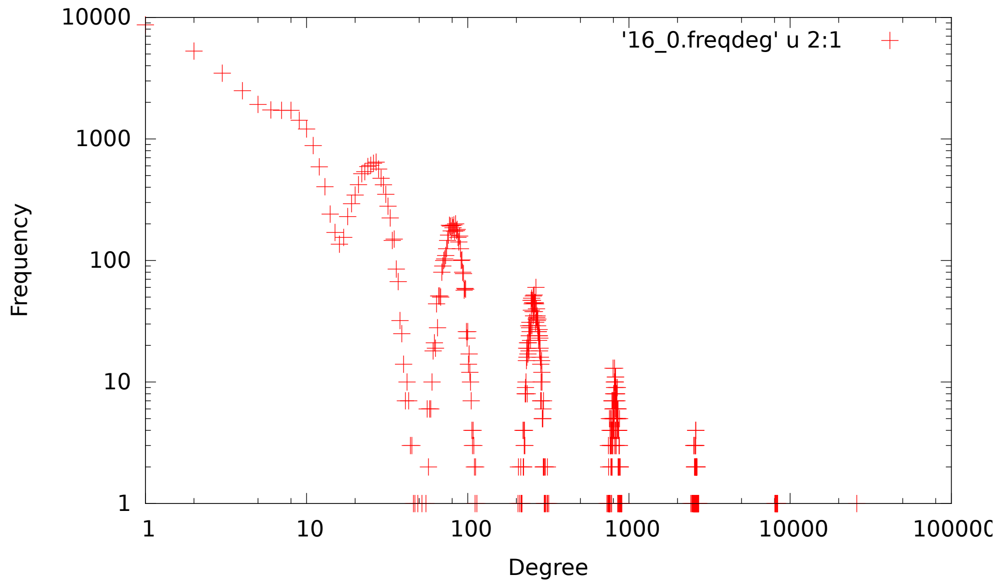


Figure 6.3: Degree-frequency of a Kronecker graph using Graph500 parameters ($s = 16, e = 16, A = 0.57, B = C = 0.19$). Observe the exaggerated combing, which is the product of a few normal distributions around expected degree (see Figures 6.4, 6.5, and 6.6).

It is readily apparent that a Kronecker graph's default isomorphism is totally correlated with its edge structure. Indeed, this defines the model: the first iteration of the distribution gives the most significant bit of the source and target vertices, the second iteration gives the next-most significant bit, and so on such that the vertices necessarily sort according to the edge distribution. As mentioned previously, the Kronecker model is a planted partition model when the distribution's diagonal (which corresponds with vertex partitions) has more mass than its anti-diagonal (which corresponds with edge cuts). Such a Kronecker graph is essentially pre-partitioned in its default isomorphism.

The Graph500 specification requires that a random isomorphism be applied to the graph's vertices and edge list. This decision is correct and reasonable within the context of the benchmark, but when the Kronecker generator is used in other evaluations it creates serious comparability issues. As discussed in the previous Chapter, the default isomorphisms of real datasets are often optimistic, so it is not common practice to randomize them. Therefore, "out of the box" Graph500 graphs and real graphs measure differently in dimensions determined by their isomorphisms, such as cache hit rates. For example, the evaluation of "Graph Prefetching using Data Structure Knowledge" by Ainwsworth and Jones[4] measures less than a 40% L1 cache hit rate on Graph500 graphs and more than 80% on real graphs, which the authors attribute to register spills. But since this evaluation does not explicitly control for isomorphism, the difference could be due to the Graph500's randomization. The problem is that the field has not defined any standard control isomorphism to impose on any graph, synthetic or otherwise; but this problem is felt accutely when benchmarking with Kronecker graphs.

In principle, a 2×2 Kronecker seed produces a graph with $|V| = 2^{scale}$ vertices and $|E|$ edges where $|E| = O(|V|)$ for very large sparse graphs. For example, the Graph500 specifies that the scale varies and the edge count covaries by a constant factor $2^{scale} * edgefactor$. Naively, one might think that this would plot a log-scale vertex axis with a proportional edge count and a constant average degree. However, we must account for isolated (i.e., zero-degree) vertices, just as in real datasets.

Seshadhri et al[110] show that *in Kronecker graphs a large fraction of vertices are isolated in expectation.* Furthermore, the fraction of isolated vertices grows with increasing scale and a fixed edge factor. This means that in a plot of Graph500 graphs with the scale parameters on the x-axis, *the real vertex count is sublinear with respect to the x-axis, the edge count is linear, and the average degree is superlinear!*

The approximate isolated vertex count is given by Seshadhri et al. as:

$$d = |E|/|V|$$

$$o = (A + B) - 1/2$$

$$t = (1 + 2o)/(1 - 2o)$$

$$g = d(1 - 4o^2)^{k/2}$$

$$isolated\_vertex\_count = \sum_{r=-k/2}^{r=k/2} \binom{k}{k/2 + r} exp(-2gt^r)$$

This is obviously a complicated relationship between the scale axis and the non-isolated vertex count. For Graph500 parameters, to obtain the "expected" vertex count of 2^k, one must usually request two to four times as many vertices (i.e., increase scale by one to two) and decrease the edge factor proportionally.

This is not an error in the Kronecker model so much as a misunderstanding between the graph modeling and benchmarking communities. The graph modeling community regards densification "with growth" as a key feature of power law models, and Leskovec et al. present it as one of the their model's strengths[75]. However, this definition of scale gives a complex x-axis with a non-trivial relationship to simple benchmark parameters, such as the vertex count. If a benchmark algorithm depends on the vertex count then its relationship with this x-axis is similarly complicated.

In a typical Kronecker benchmark plot the x-axis is scale and the y-axis is a quanity of interest such as time. If the quantity of interest is controlled by the graph's edge count $|E|$, then the plot preserves the shape of that relationship. However, if the quantity is controlled by the graph's vertex count or density then the relationship is reshaped by Equation 6.2. For most system and algorithms the true relationship may not be precisely known, which is a major reason we want to quantify it with respect to a known axis.

For example, the inner loop of a simple PageRank implementation iterates the edge list, but its cache behavior depends on a vertex-rank map, and its convergence rate is partially a function of the graph's density. A novel PageRank implementation, such as the one in GraphTwist[128] that exploits values near convergence to ellide insignificant work, has a novel and possibly unknown relationship with these parameters. The consequence of this is that it's extremely difficult to use a plot with Kronecker scale on the x-axis to correctly explain the behavior of a system or algorithm, particularly if you are not aware of the isolated vertex problem. We have committed this error in our own research by generating a Kronecker scale plot and incorrectly referring to the x-axis as a log-scale vertex count[83].

However, isolated vertices in the Kronecker model are only a symptom of a more serious problem. Figure 6.3 plots the frequency distribution of vertex degrees in a Kronecker graph drawn from the Graph500's default parameters. The graph's distribution is "combed" at regular geometric intervals, between which vertices with a given degree are highly improbable. Again, this will clearly affect an algorithm whose expected runtime depends on the degree distribution, such as triangle counting. Moreover, the same combing appears in other fundamental parameters, such as the k-core distribution (Figures 6.13). This combing problem is so pervasive that it calls into question whether Kronecker ought be treated as representative datasets for benchmarking.

Combing and isolated vertices share the same basic cause. The cause is most easily understood in the one dimensional case, and then generalized to two dimensional matrices. Consider the one

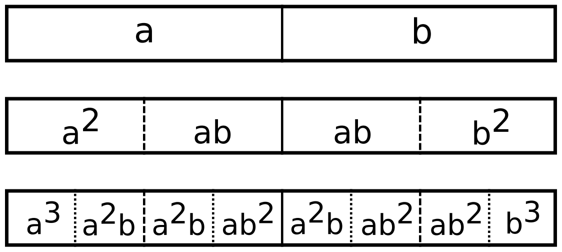


Figure 6.4: A one-dimensional Kronecker series with two distribution parameters. At iteration k the number of samples is 2^k but the number of unique sample probabilities is only $k+1$.

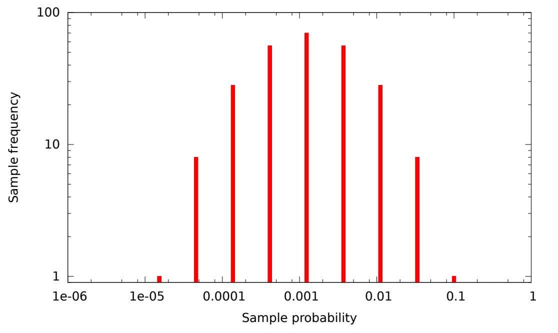


Figure 6.5: Because probability corresponds to expected degree, and degree ranges over the number of samples, the frequency of degrees is always logarithmically under-sampled.

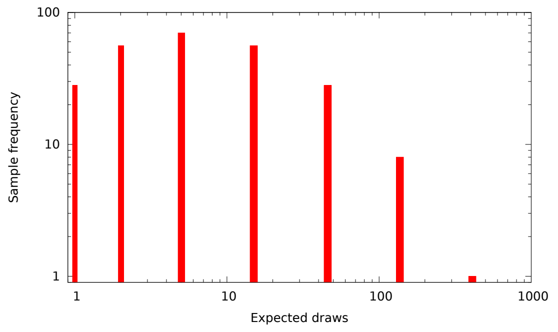


Figure 6.6: Note also that several of the samples have an expected degree of zero (i.e. are isolated) when $a = 0.75, b = .25, |E| = 16 * 2^k$ as in the Graph500.

dimensional Kronecker seed with two distribution parameters a and b, depicted in Figure 6.4. The first Kronecker product of this seed with itself is $[aa|ab|ab|bb]$, the second is

$$[aaa|aab|aab|abb|aab|abb|abb|bbb]$$

And so on. Observe that each sample's probability corresponds to the *sum* of its binary representation. Therefore, each iteration produces 2^k samples that share only $k + 1$ distinct probabilities, corresponding to $a^p b^{k-p}$ for p from 0 to k (see Figures 6.5 and 6.6). If we draw from this distribution and plot the sample frequency, we'll see $k + 1$ normal distributions at geometric intervals of a/b.

Fundamentally, there are exponentially more vertices in a Kronecker model than there are expected degrees. The out-degree distribution of a two dimensional 2×2 Kronecker seed is just the one dimensional distribution with parameters $A + B$ and $C + D$. Similarly, the in-degree distribution is the one dimensional distribution with parameters $A + C$ and $B + D$. Each Kronecker iteration produces 2^k vertices but only $k + 1$ expected out-degrees and in-degrees, which correspond to the sum of each vertex's binary representation. When we draw edges from the model and plot the out-degree or in-degree frequency, we see $k + 1$ normal distributions at geometric intervals, as in the one dimensional case. Furthermore, when $B = C$ as in the most popular Kronecker seeds (Table 6.2), the source and target distributions are symmetric and therefore fully aligned, so the total degree distribution also has $k + 1$ combs (as in Figure 6.3).

The vertex frequencies under each degree distribution follow a binomial distribution, because they correspond to the *count* of binary vertex representations with a fixed *sum*. Therefore, in a simplified sense the holistic Kronecker model follows a log-normal out and in-degree distribution, which is a key reason that it produces networks with skew degree distributions. However, this relationship is always logarithmically undersampled, and as a result, it is badly combed. Leskovec[75], Groer[47], and Seshadhri et al.[110] fully formalize this relationship.

Isolated vertices exist because, in a sparse graph, many of the normal distributions correspond to an expected degree approaching zero. In the one dimensional case, if we assume without loss of generality that $a > b$, then since $0.5 > b$ clearly b^k shrinks faster than 2^k grows. Therefore, as k goes to infinity the expected frequency of a sample with probability b^k approaches zero if the number of edge draws from the distribution is $O(|V| = 2^k)$. Similarly, for a sufficiently large graph if $A + B > C + D$ then the expected out-degree of a vertex with associated probability $(C + D)^k$ is less than one if the graph is sparse, and likewise in-degree. Totally isolated vertices must exist in expectation when $A + B > C + D$ and $A + C < B + D$ or vice-versa, which in particular is true if $B = C$, as in the Graph500 and many other Kronecker seeds.

Seshadhri proposes to smooth the distribution by independently blurring each Kronecker iteration with uniform random noise. This breaks up the normal peaks to fill the surrounding valleys by adding variation that depends on each vertex's binary *permutation*, rather than its sum. For example, in the one dimensional case two samples with coincident probabilities ab and ba become distinct probabilities $(a + n_1)(b - n_2)$ and $(b - n_1)(a + n_2)$. Consequently there are as many expected degrees as there are vertices, and the degree distribution is less obviously clustered, although in principle the distribution still varies around the same combs. This implies that the noise parameters must be chosen once per *graph* and not per edge draw, or else the model converges back to the combed distribution. Essentially the *graph model* is drawn from around the combed model as defined by the noise parameters.

This is one of several problems with Seshadhri's solution. A 2×2 Kronecker graph consists of $o(2^k)$ edge draws, so we do not expect much variation in graphs drawn from the same Kronecker model due to the law of large numbers; this is why combing always occurs. But "noisy" Kronecker models depend on only k additional random variables, and in fact it is crucial that this is not large or else the noisy model converges to the same combs as the original. Benchmarks that use noisy

Kronecker should control these random variables by running on a sufficiently large series of random graph models. The Graph500 reference implementation simply fixes the "noise" parameters at uniform intervals along their range (and slightly alters the noise equation); *it is not obvious that Seshadhri's proofs hold in this fixed case*.

This decision exposes another problem with the noisy model. In a series of k random draws from the uniform noise range $[-n : n]$, the minimum draw in expectation is $-n + 2n/k$; the Graph500's method guarantees the minimum is always $-n$. Let p be the Kronecker iteration with minimum noise; this iteration generates the *pth* most significant bit of each vertex. If we sort the vertices by this bit, this gives a bi-partitioning of the graph in which the anti-diagonal is weakened by the minimum noise. If the noise magnitude is significant, this will substantially reduce the expected edge cut cost of the planted bi-partition. More generally, noise reduces the planted edge cut of any $2^{k/2}$-way partitioning in expectation, which realistically covers all practical partitionings for any large graph. So adding noise has topological and spectral ramifications in proportion to the size of the noise.

This inevitably leads to the question: how much noise do we need to add? Unfortunately, the answer is "a lot" in practice. The correct noise depends on the Kronecker model's parameters and how one defines a "smooth enough" distribution; Seshadhri declares that $0.05 - 0.1$ is reasonable, and the Graph500 reference uses 0.1. The anti-diagonals B and C in Seshadhri's study and the Graph500 are approximately 0.2, so 0.1 is a relatively large variation; at one extreme, it halves the expected edge cut of the graph's planted bi-partition! All these problems are complimentary: a large noise magnitude more substantially alters the graph's planed bipartition and makes proper control of the random noise variables an important evaluation concern. This may explain why noisy Kronecker has not been adopted by the Graph500 *specification* (and is compiled out of the reference code) despite being implemented in the reference code for 5 years. The Graph500 and research evaluations that use the default graph generator are still based on the original combed Kronecker model.

6.3 Proposed Solution

Several key ideas from the previous Section will help us fix combing in the Kronecker model.

1. The model generally follows a reasonable degree distribution; it is just undersampled.
2. The distribution is undersampled because each vertex's expected degree is determined by the non-unique sum of its binary representation.
3. Noisy Kronecker substitutes a "Kronecker-like" graph model in which each vertex's expected degree varies according to its unique binary representation.
4. The problem with noisy Kronecker is that the vertex degrees vary normally around the undersampled model, instead of sampling the correct underlying model.

Our goal is to get each vertex's expected degree to sample the correct underlying model in a manner uniquely determined by its binary representation.

The "correct underlying model" is the log-normal distribution that the Kronecker model's binomial distribution follows as it approaches infinity. So naively, we could try to find the result of an infinite series of Kronecker products and then downsample a $|V|x|V|$ matrix from it. But if our seed is $d \times d$ and $|V| = d^k$, then this sample is just the *kth* Kronecker product, so we are right back where we started. However, this prompts another key idea: what if the vertex count is not an integer power of the seed size?

A non-integer Kronecker exponent may seem strange, but for small cases the solution is straightforward. Once again, the one dimensional case is illustrative, because the two dimensional case is just its bilinear extension. Consider the two parameter seed, $[a|b]$, from whose infinite series we want to downsample a 3-parameter distribution $[x|y|z]$. Figure 6.7 shows that x is just the geometric series whose initial term is aa and ratio is ab; this corresponds to the geometric series $1/3 =$

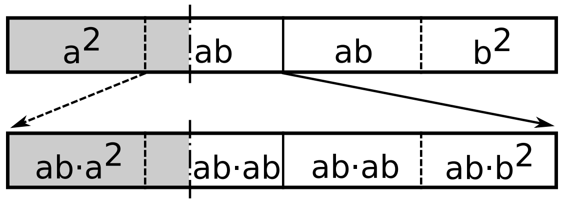


Figure 6.7: Sampling the leftmost third of the one-dimensional, two-parameter Kronecker series as it approaches infinity. The sample is clearly $aa + ab * aa + (ab)^2 * aa...$, i.e. the geometric series with initial aa and ratio ab. To sample the rightmost third swap a and b, and to sample the middle subtract the left and right from the unit.

$1/4 + 1/16 + 1/64 + ...$ y and z are similarly trivial, as is the 5-parameter distribution; in general any downsampled distribution can be explicitly solved as a geometric series of d-ary fractions. The extension to the two dimensional case is straightforward and is demonstrated in Figure 6.8

This explicit method clearly cannot scale to the septillion samples of a large Graph500 adjacency matrix. To realistically generate large Kronecker models, we need to express them as the Kronecker product of a series of small distributions. Logically, the 6-parameter distribution should be the Kronecker product of the 2 and 3-parameter distributions. However, this poses an interesting problem: the Kronecker product is not commutative. Depending on the operand order we can generate $[ax|ay|az|bx|by|bz]$ or $[xa|xb|ya|yb|za|zb]$.

Both solutions are isomorphic in that they contain the same probabilities, but the probabilities are associated with different samples. The first distribution is in “binary and then ternary” bit order, the second is in “ternary and then binary” order, and the isomorphism reorders these bits. In the two dimensional graph case, the Kronecker product of a 2×2 and a 3×3 seed is a 6×6 Kronecker model. The seeds uniquely determine the unlabeled graph model, but the operand order determines its vertex isomorphism. This distinction is unimportant in the original Kronecker model because all of the operands are equal, and because when generating Kronecker graphs the default isomorphism is usually destroyed afterwards by randomization.

However, these isomorphisms are useful to us because samples with equal probabilities are not

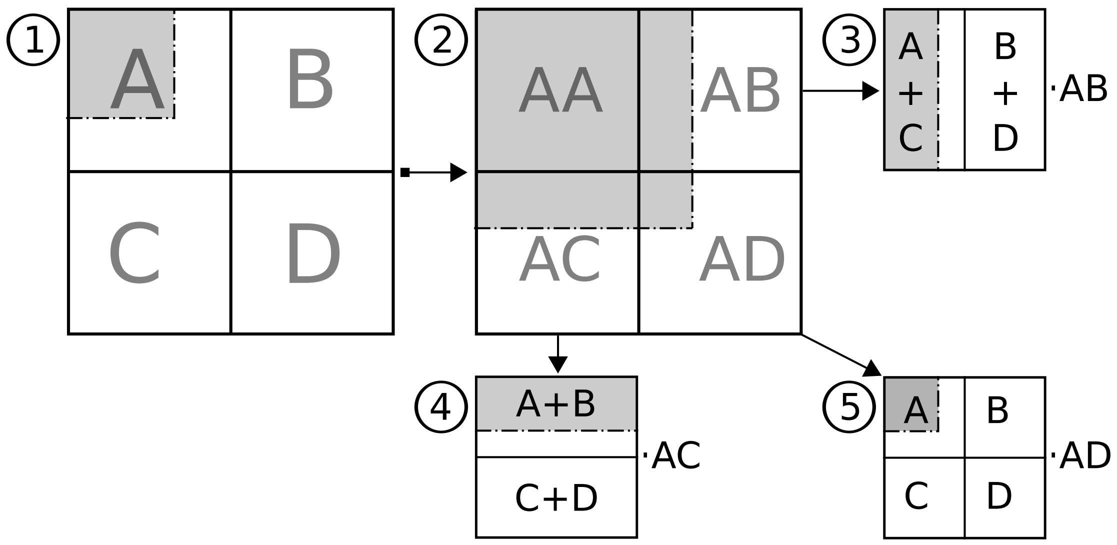


Figure 6.8: 1.) Sampling the top-left ninth of the two-dimensional, 4-parameter Kronecker series as it approaches infinity. 2.) The sample consists of AA, the leftmost third of AB, the topmost third of AC, and the top-left ninth of AD. 3.) The left and top (4) thirds are just instances of the one-dimensional case from Figure 6.7. 5.) The remaining ninth is recursive and implies AD is the ratio of a geometric series around the whole expression. Four corners, four thirds, and the unit sum give us a solvable system of 9 equations for the 3×3 downsampled seed.

necessarily preserved across different operand orders. Equivalently, vertices with equal expected degrees are not necessarily preserved across different isomorphisms. So varying the operand order/isomorphism lets us differentiate between vertices while we hold constant the actual operands of the Kronecker model. The only catch is that to meaningfully permute the operand order we need at least one alternative operand. Fortunately, geometric downsampling lets us generate alternative seeds that are consistent with the original Kronecker seed as it goes to infinity (see Figure 6.8).

6.4 Smooth Kronecker Algorithm

Our simplest algorithm works as follows:

1. Given any $d \times d$ Kronecker seed, scale k and edge count e...

2. Using geometric downsampling, generate an alternative $d_1 \times d_1$ seed such that d_1 is not a

```
function One-Dimensional Third($A, B$)
    return $A * A/(0 - A * B)$
function Two-Dimensional Ninth($A, B, C, D$)
    $Right \leftarrow A * B * $1D Third$(A + C, B + D)$
    $Bottom \leftarrow A * C * $1D Third$(A + B, C + D)$
    $Initial \leftarrow A * A + Right + Bottom$
    return $Initial/(0 - A * D)$
function 3x3 Resample($A, B, C, D$)
    $a \leftarrow$ 2D Ninth$(A, B, C, D)$
    $b \leftarrow$ 2D Ninth$(B, A, D, C)$
    $c \leftarrow$ 2D Ninth$(C, A, D, B)$
    $d \leftarrow$ 2D Ninth$(D, B, C, A)$

    $ab \leftarrow$ 1D Third$(A + B, C + D) - a - b$
    $ac \leftarrow$ 1D Third$(A + C, B + D) - a - c$
    $bd \leftarrow$ 1D Third$(B + D, A + C) - b - d$
    $cd \leftarrow$ 1D Third$(C + D, A + B) - c - d$

    $x \leftarrow 0 - a - b - c - d - ab - ac - bd - cd$
    return $[a, ab, b, ac, x, bd, c, cd, d]$
```

Algorithm 4: Explicit function to resample a $2x3$ seed distribution from a 2×2 seed using the method shown in Figure 6.8.

power of d.

3. For each edge, choose an operand order r uniformly at random from $[0 : k)$.
4. Iterate $k - 1$ draws from the initial seed as normal, but on draw r use the alternative seed.
5. Repeat from Step 3 for e edges to generate a graph with $d^{k-1} * d_1$ vertices.

An intuitive explanation for this algorithm is readily apparent. The alternative $d_1 \times d_1$ seed is a smoothing filter that pushes the finite Kronecker product towards a better approximation of the infinite Kronecker product. At any point in a given series there is a $1/k$ probability that the distribution is locally blurred by an alternative but faithful downsample from the infinite series. The finite

```
function SMOOTH KRONECKER GENERATOR($A, B, C, D, k, e$)
    $2 \times 2_{S}eed \leftarrow [A, B, C, D]$
    $3 \times 3_{S}eed \leftarrow$ 3x3 RESAMPLE($A, B, C, D$)
    for $e$ edges do
        $source \leftarrow 0$
        $target \leftarrow 0$
        $base \leftarrow 1$

        $r \leftarrow$ RANDOM INT($0, k$)
        for all $i \in$ RANGE($0, k$) do
            if $\not i = r$ then
                $cell \leftarrow$ RANDOM CHOICE($2 \times 2_{S}eed$)
                $source \leftarrow source + (cell/2) * base$
                $target \leftarrow target + (cell\%2) * base$
                $base \leftarrow base * 2$
            else
                $cell \leftarrow$ RANDOM CHOICE($3 \times 3_{S}eed$)
                $source \leftarrow source + (cell/3) * base$
                $target \leftarrow target + (cell\%3) * base$
                $base \leftarrow base * 3$
        Yield $source, target$
```

Algorithm 5: Smooth Kronecker Algorithm with a 2×2 input seed. Essentially, when we substitute the 3×3 seed we generate a trinary bit. This method generalizes to arbitrary mixtures of d-ary seeds.

Kronecker product gives an undersampled binomial degree distribution, but the infinite Kronecker product gives a correct log normal degree distribution, so improving this approximation improves the distribution. Geometric downsampling and permuting the operand order are just clever tools to efficiently approximate the infinite series without explicitly materializing a large distribution. Figure 5 gives more explicit code for 2×2 input with an alternative 3×3 seed, which is the case that covers all the popular Kronecker benchmark parameters.

The most critical distinction between our algorithm and noisy Kronecker is that we draw a new isomorphism for every edge and therefore converge over many edge draws. In contrast, noisy Kronecker draws a new graph model only once per graph, because it converges to the very same combed distribution that it tries to avoid. Another key distinction is that our algorithm is smoothed by a *d1xd1* filter that we sample from the Kronecker model as it approaches infinity, because the infinite Kronecker model gives a correct log-normal degree distribution. In contrast, noisy Kronecker applies a strong uniform blur that alters the model's properties.

The single $d_1 \times d_1$ seed increases the model's vertex count, so graphs drawn from this model are not directly comparable to graphs drawn from a pure $d \times d$ model because the explicit scale is different. Fortunately it is trivial to extend this algorithm to arbitrary mixtures of $d_1 \times d_1$ and $d \times d$ seeds. For example, 3^5 and 2^8 are approximately equal, so one can substitute five 3×3 seeds for eight 2×2 seeds and preserve the approximate scale. In fact this gives our scale parameters more degrees of freedom than the pure $d \times d$ model, because we are no longer restricted to purely logarithmic scales.

This algorithm and its sampling method are our own research, but Daniel Alabi and Dimitris Kalimeris have independently pursued a formula for the degree distribution of the Smooth Kronecker model and a proof that it follows a log-normal distribution. The proof is not fully complete, but a proof sketch is available as a preprint. This is par for the course: six years passed between the initial publication of the Kronecker model[74] and a full characterization of its degree distribution[110].

6.5 Empirical Demonstration

Figure 6.9 compares the degree-frequency of a traditional Kronecker graph, previously shown in Figure 6.3, to an equivalent graph that substitutes five resampled 3×3 seeds for eight of the original 2×2 seeds. The combing is almost entirely corrected by the addition of the alternative seeds. In comparison, noisy Kronecker exhibits much of the original combing even with a relatively large noise parameter. Figure 6.10 reproduces a plot from Seshadhri et al. that compares noisy Kronecker to the original Kronecker model using the same Graph500 parameters. Even $+-0.1$ noise leaves a considerable amount of combing towards the end of the degree distribution; and note that the noise

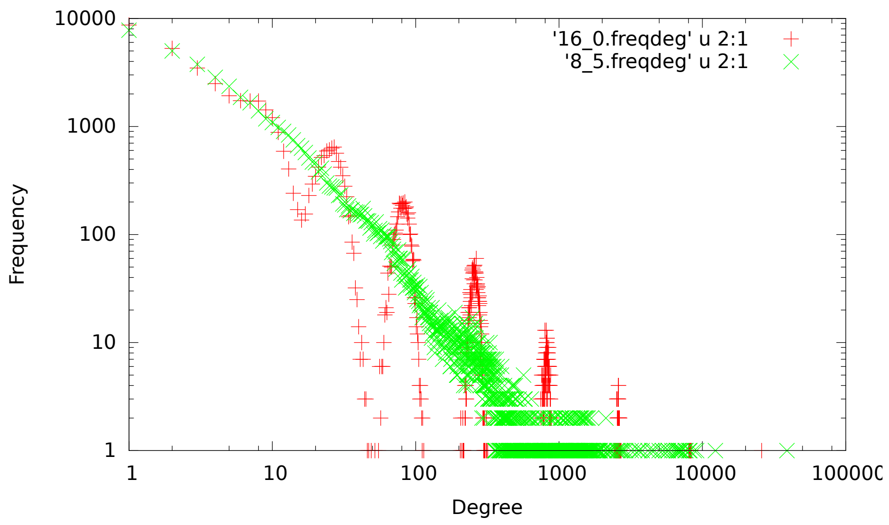


Figure 6.9: Degree-frequency plot of a Smooth Kronecker graph superimposed over the traditional Kronecker graph from Figure 6.3.

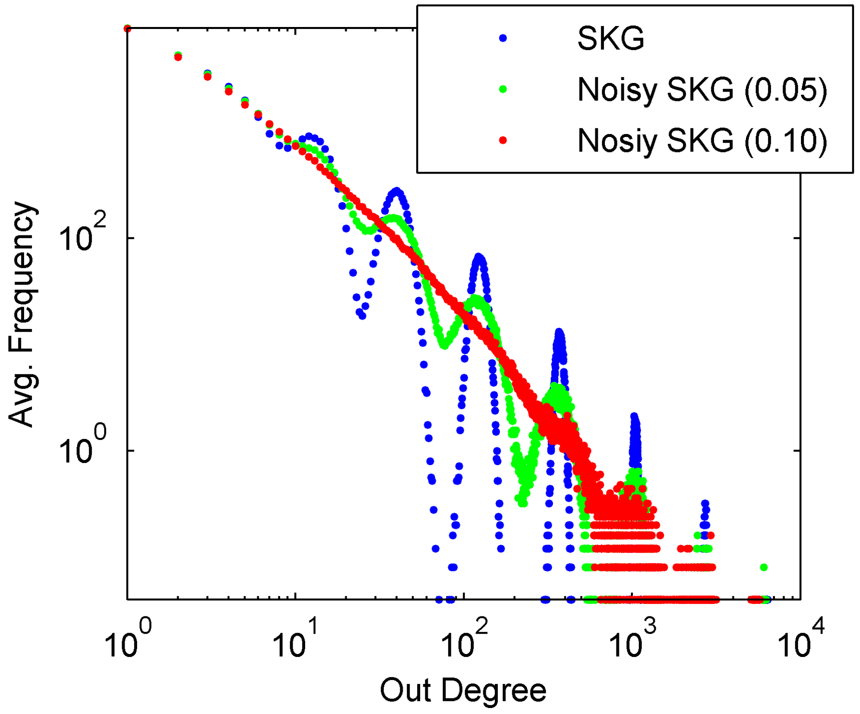


Figure 6.10: Average degree-frequency plot of 25 noisy Kronecker graphs for each of two noise parameters, superimposed over traditional Kronecker.

is added directly to each of the anti-diagonals *B* and *C*, which in the Graph500 are 0.19, so to achieve this result the anti-diagonal varies by more than a factor of two. Consequently, the reproduced plot is averaged over 25 graphs because in any one graph the realized noise parameters may vary considerably. Like the original Kronecker model, our algorithm is not random per-graph but only per-edge; so individual graphs converge over their many edge draws. This makes our algorithm more suitable for generating canonical benchmark graphs.

Note that combing is only plottable when the graph's scale is relatively small. Because the comb count is equal to the scale plus one, increasing the scale visually pushes the combs closer together in a fixed-width log-scale plot. Thus, at large scales the combs appear to blend together and vanish, but this is strictly a visual illusion because the plot width does not increase in proportion to the scale. In reality combing is *more* exaggerated at large scales, because each new comb is separated by a geometric factor from previous combs.

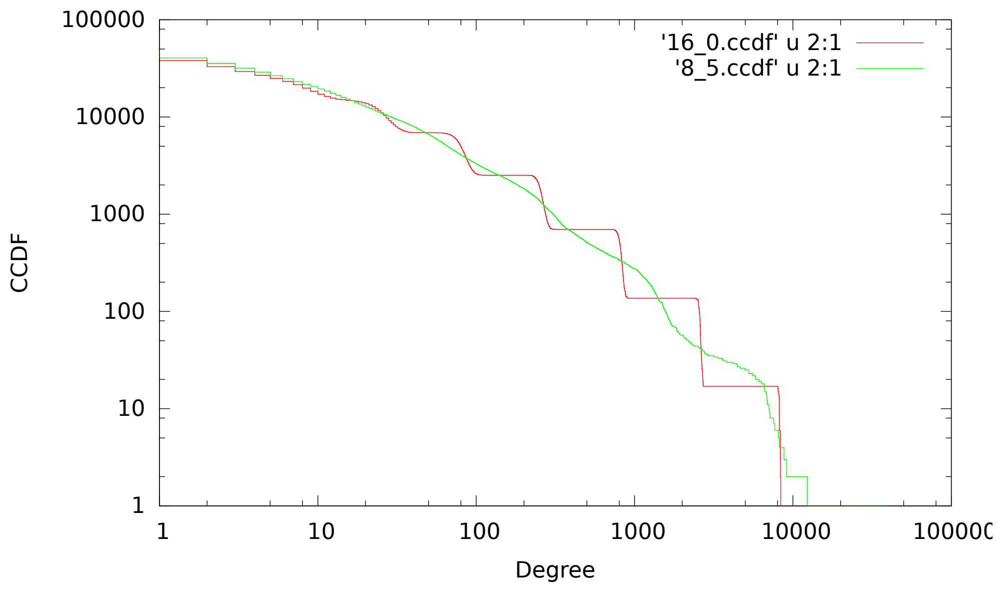


Figure 6.11: CCDF of a smoothed Kronecker graph superimposed over the traditional Kronecker graph from Figure 6.3.

It is much easier to see details in the degree's complementary cumulative distribution (CCDF); in this representation the combs appear as "steps." Figure 6.11 compares the CCDF of the same traditional and smoothed Kronecker graphs. Here we see that the smoothed graph actually follows the same combing pattern as the traditional graph, but it is dampened by orders of magnitude (note that the y-axis is log scale). This is evidence of our intuition that the 3×3 resampled seed acts as a smoothing filter over the pure 2×2 series.

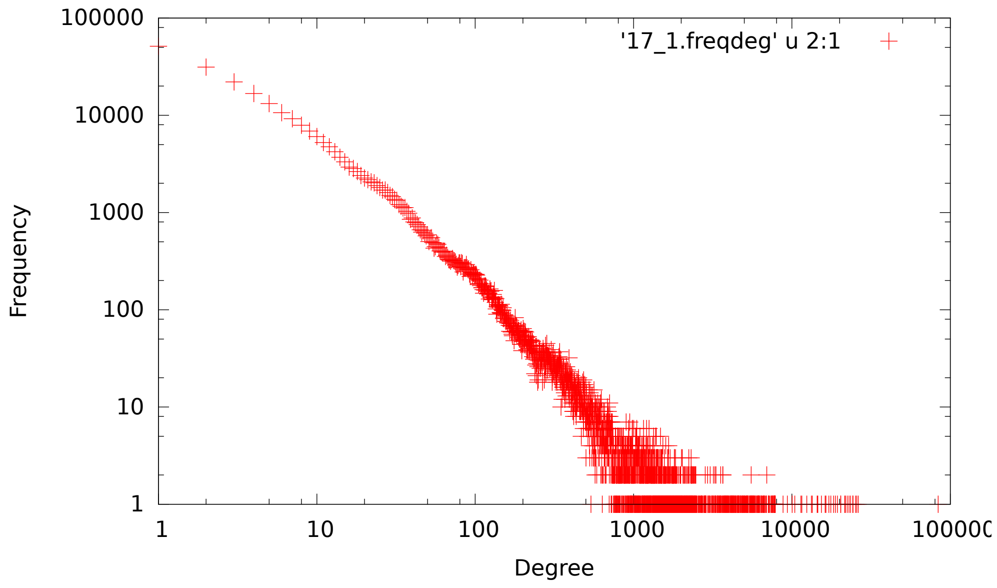
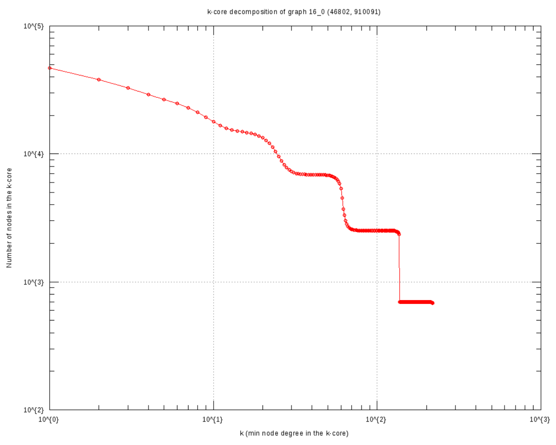


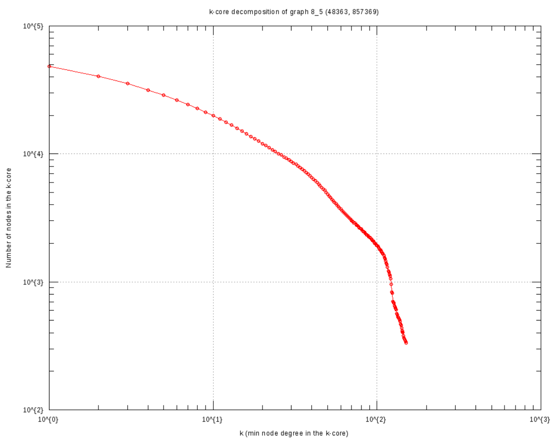
Figure 6.12: A smoothed Kronecker graph with seventeen 2×2 seeds and only one resampled 3×3 seed.

One seed is enough to produce a smooth degree-frequency curve. Figure 6.12 plots the degree-frequency and CCDF of a graph with only one resampled 3×3 seed. Broadly this means that many more vertex scales are possible in the mixed-seed model; in principle one could realize any vertex scale by factoring the desired scale and generating appropriate seeds. However, recall that one motivation for the seed-based Kronecker model is to avoid materializing very large distributions. Therefore it is not efficient to realize vertex scales that are not the products of small factors. And of course, one must avoid powers of primes, or else the model degenerates to the original Kronecker model.

Like noisy Kronecker, smooth Kronecker does not significantly change the isolated vertex count because it intentionally follows the degree distribution of the original Kroncker model, in which many vertices have zero degree in expectation. The appropriate correction for isolated vertices is to always report the true vertex count and, if necessary, to invert Equation 6.2 and/or bisect the model's parameters to reach the desired count.



(a) Sixteen 2×2 seeds.



(b) Eight 2×2 and five 3×3 seeds.

Figure 6.13: K-core CCDFs of an original and smoothed Kronecker graph.

The combing effect in Kronecker models is not limited to degree frequency, but is also present in other parameters that broadly measure "centrality," such as k-cores. Figure 6.13 depicts the k-core CCDF of an original Kronecker graph and an equivalent smoothed Kronecker graph.

6.6 Conclusions

In this Chapter we present a "fix" to the Kronecker model that strongly dampens its unnatural combing behavior and thereby makes the model a more appropriate source of synthetic benchmark data. Our smoothing fix significantly improves over a previous fix proposed by Seshadhri et. al, and in particular, addresses the statistical requirements of their fix that inhibit benchmarking.

To be absolutely clear, the intent of our work is different from the authors of the original R-MAT and Kronecker models. As originally proposed, R-MAT and Kronecker are data mining models whose "parameters" are really metrics given by a best-fit of the model to a given graph. In this context, combing, isolated vertices, and even the model's optimistic default isomorphism are not really flaws, but rather features of the model that one must account for in the fitting process. Leskovec's original KronFit algorithm specifically explores the isomorphism space[75], and a later algorithm by Kim and Leskovec does a better job of accounting for isolated vertices[65]. Kim and Leskovec's later MAG model mixes different seed matrices that, to some extent, dampen combing[66], although it is still visible in their degree-frequency plots. To the best of our knowledge, our resampling method and our method of permuting the Kronecker operand order are new.

The use of the Kronecker model as a synthetic benchmark model is a *de facto* practice that has gradually become standardized and is now widespread, as shown in the previous chapter. In this context the model is not merely a mining abstraction, but instead takes on a life of its own; real systems are evaluated and presented on the basis of Kronecker graphs. The Kronecker model enjoys this use not simply because of its "realism," but also because of practical features such as its trivial operations, trivial parallelism, small data structures and small parameters that are easy to publish and therefore easy to reproduce and compare. Our "fix" is intended for this ecosystem, and in particular, it maintains the trivial parallelism and backwards compatibility with published parameters via resampling.

In principle, future work could expand on our model and take advantage of its new parameters, e.g., arbitrary mixes of different seed distributions targeting graphs with nearly arbitrary scale (rather than log scale). However, we think it is time that the graph benchmarking community recognize that they have distinct needs from the data mining community. "Scale" is a great example of a concept whose meaning is significantly different between benchmarking and mining contexts. If we want to claim that our systems improve scalability, then we need to be pushing for a benchmarking practice where we can plot with scale and have well-defined expectations. Similarly, smoothing the degree distribution brings the model more in line with our expectations. Our expectations are important because they determine our hypotheses and therefore what conclusions we can draw from our experiments. Though our fix is pragmatic, it directly addresses what we see as an urgent epistemic problem in the graph systems field and in "big data" systems in general.

7 Conclusion

This dissertation presents several projects that logically follow from one another within the common topic of high-performance graph data processing. In Chapter 3 I introduced SHEEP, a distributed graph partitioner and vertex sorter that reduces communication volumes and runs extremely fast on large graphs (Chapter 3). SHEEP is so fast that it is ultimately bound by data bandwidth at scale, which is arguably the best case for a partitioner without *a priori* partitioning assumptions. SHEEP abandons the streaming and label propagation idioms that are common in contemporary high-performance partitioners, in favor of traditional idioms such as elimination trees and distributed union-find data structures. In this regard SHEEP is one of several papers published in 2015 that bridged contemporary graph systems and traditional HPC linear algebra systems, such as GraphMat[116].

I developed SHEEP because I strongly believe that graph data *shapes* have a large performance impact but are criminally neglected by contemporary research. Partitioning, vertex sorting, and elimination trees are simply my preferred way of thinking about graph data shapes. While evaluating SHEEP I was often advised to benchmark end-to-end on "10 iterations of PageRank on Twitter," and "the Graph500 benchmark," which were *de facto* community standards at that time. I was skeptical of these practices precisely because my familiarity with public graph data suggested many potential errors and unanswered questions. What is a meaningful baseline for an arbitrary 20 iterations of a naive algorithm implementation on a highly specific graph? This question serves as a proxy for a more general investigation, but on some level it strikes me as simply absurd as it sounds. Around this time I co-authored several papers with Peter Macko, in which we followed many of these best/worst practices, such as scalability plots with Kronecker scale on the x-axis[83]. We even co-authored a benchmark suite[84], thereby contributing to the plurality of standards.

I presented a metastudy of graph systems publications (Chapter 5), because I wanted to commit these thoughts to a formal framework and support them with data. I collected papers from what I regarded as the most important conferences within a critical time period following the publication of Giraph. This was far too many papers for me to meaningfully engage with, so I committed to a principled method to reduce the paper count to an essential core. This method intentionally featured myself as an agent in the process, because search cannot currently match the nuanced judgement of domain-specific expertise. I presented data on 65 graph systems evaluations from top-tier conferences in the period from 2011 to 2015. I confirmed a skew long-tailed distribution among benchmark algorithms and benchmark datasets. For algorithms I feel that this result was expected, but I was surprised by the concentration among popular datasets, in particular, because I recognized and documented flaws in many of those datasets.

I supported these criticisms with an evaluation of someone else's system, Galois[68 96], which I regard as high quality research. My evaluation originally consisted of a potpourri of systems, benchmarks and datasets totalling something like 1500 cases. Frankly, this was a poor method to produce knowledge, because it lacked direction and included many absurd cases that were not best practices for each scenario. By limiting my focus to a mature high-performance system, I was able to show that graph data shapes have a real performance impact on results that could realistically be published and compared. In particular I showed that 20 iterations of PageRank are in fact as problematic as my intuition had long suggested.

At the time I conducted my metastudy the graph systems field was also facing criticism from a well-known article by Frank McSherry et al.[89] However, a key distinction between my metastudy and McSherry's article is that his method is a comparative evaluation *reductio ad absurdum*. McSherry's demonstration of a preposterous result is striking and effective, but it does not expose the complex history by which the preposterous became publishable. My conclusion from my metastudy was that many evaluation errors were a natural consequence of citation and comparison processes applied to a limited corpus of public graph datasets. The citation of and comparison to prior work are best practices for good reasons, but the limitations of public graph datasets are considerably more thorny. If I may be opinionated: the foundation myth of the "big data" movement is that data has value, but when that value is regarded as capital goods, then "big data" is not compatible with open data science. It is extremely difficult to show our work has general impact when we do not possess diverse, substantial datasets.

Finally, in Chapter 6 I presented a pragmatic contribution to the this dataset problem in the form of a synthetic graph generator. This generator reuses the existing parameters of the popular R-MAT/Kronecker generator, but applies a novel method based on isomorphisms of the Kronecker product to smooth problematic structures out of the synthetic graph. From my prior evaluations, I was familiar with the Graph500 Kronecker generator, and my metastudy confirmed that it was a *de facto* standard even for evaluations outside the scope of the Graph500 specification. In graph data mining the Kronecker model is somewhat outdated, but as a synthetic benchmark it remains popular due to distinct benchmark requirements, such as easily shared parameters and extreme scalability. I think it's obvious that the "Kronecker" model was adapted to benchmarks only because of its complete backwards compatibility with the previously popular R-MAT model. Thus, backwards compatibility from my own model to the Kronecker model was an important requirement.

7.1 Future Work and Thoughts

The performance effects of graph data shapes are one instance of much more general data path effects in systems. In all my research I have regularly encountered results that are best explained by data paths, yet I find that our language for discussing and designing around data paths is primitive compared to execution. For example, I believe that data paths are the critical bottleneck in most "big" graph data applications, yet the majority of graph analysis systems in the past five years research sophisticated execution idioms such as scheduling, task distribution, etc. Only a handful of systems bother with simple data path idioms, such as integer list compression[21] or space-filling curves[88], even though these idioms have proven performance impact. I think there are several reasons for this, one of which is that execution is clearly important, and I don't mean to imply otherwise. But I also think that the sophisticated language with which we discuss and design execution paths is a major source of inspiration for future research: one can bring up "amorphous parallelism" at a conference dinner and quickly solicit a lot of interesting ideas. And, to some extent, I think that we romanticize computational parallelism in a way that we do not romanticize the memory bus.

I would like to contribute to a language of data path idioms in system design. My experience is with graph systems, and fortunately graphs are sufficiently general to describe large parts of computer science. Abstract data models and metrics, such as partitions and communication volumes, need not apply only to graphs as input data; they are potentially methods to reason about the data paths embodied in the system itself. Systems consist of components that communicate through paths with varying latency and throughput, and this creates bottlenecks and opportunities that are precisely analogous to input data shaping. At scale, these paths are computer networks, and graphical analysis methods such as partitioning have produced important results on networked systems. I see no reason why these results couldn't extend to data paths at every scale, such as cache communications. In such a future, our historic obsession with trivially serial data access patterns will not suffice, so I have enjoyed recent research papers on non-serial data prefetching and caching[4].

However, computer systems are ultimately empirical in that they are judged by quantitative measurements. It is tempting to think that systems are exempt from the abstruse issues of the philosophy of science, if only because end-to-end runtime feels like such an object quantity. Of course we all acknowledge "experimental error," but in my metastudy I try to show that this error is best understood in terms of contextual and qualitative impacts, instead of error bars. In different fields and even conferences I've encountered strikingly different conventions for what constitutes a well-supported, and therefore publishable, result. I think these conventions are largely emergent and, in general, are in need of testing.

I don't think another benchmark suite or metric is necessarily a good direction for future work. Graph analysis systems have at this point measured every conceivable quantity of benchmark abstractions such as PageRank or triangle counting. Rather, it's time for mature graph systems to apply themselves to concrete problems in need of solutions. I think it's noteworthy that the Netflix Prize challenge is fondly remembered and regularly used as a benchmark despite its legal and ethical issues[94], and the fact that the winning algorithm was never implemented in practice because the gains "did not seem to justify the engineering effort."[1]. Similarly, the most influential graph benchmark of the past decade is arguably Kwak et. al's Twitter dataset[71], which was almost certainly not intended as a systems benchmark. So if you want to make a positive contribution to graph systems benchmarks, I think you should *create new datasets* coupled with interesting challenges and then publish them with high quality meta-data.

One might regard synthetic graph generators as mere substitutes for real data, but I don't think this is accurate. Real data is most interesting when coupled with a real problem of interest; PageRank on Twitter is a synthetic problem that's interest is to quantify aspects of the system. Real data is full of eccentric features, such as optimistic default orders, that are interesting in reality but can inhibit our understanding of the system, because their effects are complex or uncontrolled. In principle synthetic generators with the right parameters grant us fine-grained experimental controls that should deepen our understanding of our systems. However, synthetic systems benchmarks have distinct needs that are not shared by realistic graph models from the data mining community.

A
Proofs

In Section 3.4.1 we made the following claim about the elimination game. Recall G is an undirected graph, P is a total order, and T is a tree produced by the elimination game.

Theorem 1. *Let $G[V <_P z]$ be the subgraph induced on G by vertices less than z. Then, z is the parent in T of exactly the P-maximum vertices in the disjoint components of $G[V <_P z]$ that z joins together in $G[V \leq_P z]$.*

Proof. By lemmas:

Lemma 1. *For all $(x, y) \in T_E$, $x <_T y$ in the partial order defined by T. Because $(x, y) \in T_E$ iff $x <_P y$, it follows that T defines a suborder of P. Therefore, for all $x \in T_V$, x must be the P-maximum vertex in $subt(x)$.*

Lemma 2. *Let $G[V <_P z]$ be the subgraph induced on G by vertices less than z. By Corollary 1.1, subt(x) and subt(y) are disconnected in $G[V <_P z]$. But, there must exist an edge (x', z) in $G[V \leq_P z]$ such that x' in subt(x):*

If x is a child of z, then in iteration x of the elimination game, $(x, z) \in H_E$. If $(x, z) \in H_E$, either $(x, z) \in G_E$ or there exists a prior iteration x' where $(x', z) \in H_E$. Again, either $(x', z) \in G_E$ or there exists a prior iteration. Because initially $H = G$ this must terminate in an edge in G_E. There must also exist (y', z) such that y' in subt(y).

Lemma 3. *If we apply Lemma 2 to $G[V <_P x]$ it follows that for each child of x there must be an edge in $G[V \leq_P x]$ that connects x to that child's subtree. The same is recursively true of the children of x's children, etc. It follows inductively that subt(x) must be a connected component in $G[V <_P z]$. The same is true of subt(y). By Lemma 2, subt(x) and subt(y) are disjoint components in $G[V <_P z]$, but z connects these components in $G[V \leq_P z]$.*

Therefore, by Lemmas 1 and 3 vertex z is the parent of exactly the P-maximum vertices in the disjoint components of $G[V <_P z]$ that z joins in $G[V \leq_P z]$. □

In Section 3.4.2 we made the following claim:

Theorem 2. *Let G_1 and G_2 be two subgraphs of G such that $G_1 \cup G_2 = G$. Let $t(G, P)$ be the elimination tree produced by union-find on G in order P. Then,*

$$t(t(G_1, P) \cup t(G_2, P), P) = t(G, P)$$

Proof. Let G_1 and G_2 be subgraphs of G such that $G_1 \cup G_2 = G$. For clarity, let $t(G) = t(G, P)$ for constant P. Let $G' = t(G_1) \cup t(G_2)$. We must show that $t(G') = t(G)$.

By induction we will show that in iteration z of the union-find $t(G')$ gains exactly the same children of z as would $t(G)$. Clearly this is true in the first iteration where no children are gained: assume this is true of every iteration before z.

In iteration z let the *predecessors of z in $t(G)$* be all $(x, z) \in G_E, x <_p z$. The predecessors of z in $t(G')$ are all $(x, z) \in G'_E, x <_p z$. Since $G' = t(G_1) \cup t(G_2)$, and $t(G_1)$ and $t(G_2)$ both define suborders of P, the predecessors of z in $t(G')$ are exactly the children of z in $t(G_1)$ and $t(G_2)$.

The predecessors of z in $t(G_1)$ are all $(x, z) \in G_E, x <_p z$, and similarly G_2. But since G_{1E} and G_{2E} are subsets of G_E and $G_{1E} \cup G_{2E} = G_E$, every predecessor of z in $t(G)$ must be a predecessor of z in $t(G_1)$ or $t(G_2)$. So, every predecessor in G is seen by the algorithm in either $t(G_1)$ or $t(G_2)$. By the definition of the algorithm, for every such predecessor the maximum vertex y in its component in $G_1[V <_P z]$ or $G_2[V <_P z]$ is a child of Z in $t(G_1)$ or $t(G_2)$. In either case, y is then a predecessor of z in $t(G')$.

Therefore, for any predecessor (x, z) of z in $t(G)$, there exists a corresponding predecessor (y, z) in $t(G')$ such that y is a vertex in a component that contains x in either $G_1[V <_P z]$ or $G_2[V <_P z]$. Since both are subgraphs of $G[V <_P z]$, it must be the case that y is contained in the same component as x in $G[V <_P z]$. Therefore, by inductive assumption x and y are contained in the same set of the union-find U at iteration z of $t(G')$, and therefore $U.find(x) = U.find(y)$. Therefore, $t(G')$ finds the same children as $t(G)$, *though the predecessors by which it finds them may differ.* $\square$

References

[1] (2012). Netflix recommendations: Beyond the 5 stars (part 1). http://techblog.netflix.com/2012/04/netflix-recommendations-beyond-5-stars.html.

[2] (2017). altavista-2002. http://law.di.unimi.it/webdata/altavista-2002/.

[3] (2017). twitter-2010. http://law.di.unimi.it/webdata/twitter-2010/.

[4] Ainsworth, S. & Jones, T. M. (2016). Graph prefetching using data structure knowledge. In *Proceedings of the 2016 International Conference on Supercomputing* (pp.39).: ACM.

[5] Albert, R., Jeong, H., & Barabási, A.-L. (2000). Error and attack tolerance of complex networks. *Nature*, 406(6794), 378–382.

[6] Alexanderson, G. (2006). About the cover: Euler and königsberg's bridges: A historical view. *Bulletin of the american mathematical society*, 43(4), 567–573.

[7] Andreev, K. & Räcke, H. (2004). Balanced graph partitioning. In *Proceedings of the sixteenth annual ACM symposium on Parallelism in algorithms and architectures* (pp. 120–124).: ACM.

[8] Angles, R. & Gutierrez, C. (2005). Querying rdf data from a graph database perspective. In *European Semantic Web Conference* (pp. 346–360).: Springer.

[9] Arifuzzaman, S., Khan, M., & Marathe, M. (2013). Patric: A parallel algorithm for counting triangles in massive networks. In *ACM International Conference on Information and Knowledge Management*.

[10] Ashcraft, C. & Liu, J. W. (1998). Robust ordering of sparse matrices using multisection. *SIAM Journal on Matrix Analysis and Applications*, 19(3), 816–832.

[11] Avery, C. (2011). Giraph: Large-scale graph processing infrastructure on hadoop. *Hadoop Summit*.

[12] Bader, D. A. & Madduri, K. (2005). Design and implementation of the hpcs graph analysis benchmark on symmetric multiprocessors. In *International Conference on High-Performance Computing* (pp. 465–476).: Springer.

[13] Barabási, A.-L. & Albert, R. (1999). Emergence of scaling in random networks. *science*, 286(5439), 509–512.

[14] Bertele, U. & Brioschi, F. (1972). *Nonserial dynamic programming*. Academic Press.

[15] Bharat, K., Broder, A., Henzinger, M., Kumar, P., & Venkatasubramanian, S. (1998). The connectivity server: Fast access to linkage information on the web. *Comput. Netw. ISDN Syst.*, 30(1-7), 469–477.

[16] Bodlaender, H. L., Fomin, F. V., Koster, A. M., Kratsch, D., & Thilikos, D. M. (2012). A note on exact algorithms for vertex ordering problems on graphs. *Theory of Computing Systems*, 50(3), 420–432.

[17] Bodlaender, H. L., Gilbert, J. R., Hafsteinsson, H., & Kloks, T. (1995). Approximating treewidth, pathwidth, frontsize, and shortest elimination tree. *Journal of Algorithms*, 18(2), 238–255.

[18] Boguñá, M., Papadopoulos, F., & Krioukov, D. V. (2010). Sustaining the internet with hyperbolic mapping. *CoRR*, abs/1009.0267.

[19] Boldi, P., Rosa, M., Santini, M., & Vigna, S. (2011). Layered label propagation: A multiresolution coordinate-free ordering for compressing social networks. In *Proceedings of the 20th international conference on World wide web* (pp. 587–596).: ACM.

[20] Boldi, P., Santini, M., & Vigna, S. (2008). A large time-aware graph. *SIGIR Forum*, 42(2), 33–38.

[21] Boldi, P. & Vigna, S. (2004). The webgraph framework i: compression techniques. In *Proceedings of the 13th international conference on World Wide Web* (pp. 595–602).: ACM.

[22] Bourse, F., Lelarge, M., & Vojnovic, M. (2014). Balanced graph edge partition. In *20th ACM International Conference on Knowledge Discovery and Data mining* (pp. 1456–1465).: ACM.

[23] Broadbent, S. R. & Hammersley, J. M. (1957). Percolation processes. In *Mathematical Proceedings of the Cambridge Philosophical Society*, volume 53 (pp. 629–641).: Cambridge Univ Press.

[24] Bronson, N., Amsden, Z., Cabrera, G., Chakka, P., Dimov, P., Ding, H., Ferris, J., Giardullo, A., Kulkarni, S., Li, H., et al. (2013). Tao: Facebook's distributed data store for the social graph. In *Presented as part of the 2013 USENIX Annual Technical Conference (USENIX ATC 13)* (pp. 49–60).

[25] Bu, Y., Borkar, V., Jia, J., Carey, M. J., & Condie, T. (2014). Pregelix: Big (ger) graph analytics on a dataflow engine. *Proceedings of the VLDB Endowment*, 8(2), 161–172.

[26] Buluç, A. & Gilbert, J. R. (2011). The combinatorial blas: Design, implementation, and applications. *International Journal of High Performance Computing Applications*, (pp. 1094342011403516).

[27] Buluç, A. & Madduri, K. (2011). Parallel breadth-first search on distributed memory systems. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis* (pp.65).: ACM.

[28] Chakrabarti, D., Zhan, Y., & Faloutsos, C. (2004). R-mat: A recursive model for graph mining. In *SDM*, volume 4 (pp. 442–446).: SIAM.

[29] Chen, R., Shi, J., Chen, Y., & Chen, H. (2015). Powerlyra: Differentiated graph computation and partitioning on skewed graphs. In *10th ACM SIGOPS European Conference on Computer Systems*: ACM.

[30] Chiba, N. & Nishizeki, T. (1985). Arboricity and subgraph listing algorithms. *SIAM Journal on Computing*, 14(1), 210–223.

[31] Chung, F. & Lu, L. (2002). The average distances in random graphs with given expected degrees. *Proceedings of the National Academy of Sciences*, 99(25), 15879–15882.

[32] Dean, J. & Ghemawat, S. (2008). Mapreduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1), 107–113.

[33] DeWitt, D. & Stonebraker, M. (2008). Mapreduce: A major step backwards. *The Database Column*, 1, 23.

[34] Dominguez-Sal, D., Urbón-Bayes, P., Giménez-Vanó, A., Gómez-Villamor, S., Martínez-Bazan, N., & Larriba-Pey, J.-L. (2010). Survey of graph database performance on the hpc scalable graph analysis benchmark. In *International Conference on Web-Age Information Management* (pp. 37–48).: Springer.

[35] Dorogovtsev, S. N., Goltsev, A. V., & Mendes, J. F. F. (2006). K-core organization of complex networks. *Physical review letters*, 96(4), 040601.

[36] Ekanadham, K., Horn, W., Kumar, M., Jann, J., Moreira, J., Pattnaik, P., Serrano, M., Tanase, G., & Yu, H. (2016). Graph programming interface (gpi): a linear algebra programming model for large scale graph computations. In *Proceedings of the ACM International Conference on Computing Frontiers* (pp. 72–81).: ACM.

[37] Erdös, P. & Rényi, A. (1959). On random graphs, i. *Publicationes Mathematicae (Debrecen)*, 6, 290–297.

[38] Erdös, P. & Rényi, A. (1960). On the evolution of random graphs. *Publ. Math. Inst. Hung. Acad. Sci*, 5(17-61), 43.

[39] Feng, W.-c. & Cameron, K. (2007). The green500 list: Encouraging sustainable supercomputing. *Computer*, 40(12).

[40] Fredman, M. & Saks, M. (1989). The cell probe complexity of dynamic data structures. In *21st ACM Symposium on Theory of Computing* (pp. 345–354).: ACM.

[41] Gadepally, V., Bolewski, J., Hook, D., Hutchison, D., Miller, B., & Kepner, J. (2015). Graphulo: Linear algebra graph kernels for nosql databases. In *Parallel and Distributed Processing Symposium Workshop (IPDPSW), 2015 IEEE International* (pp. 822–830).: IEEE.

[42] George, A. (1973). Nested dissection of a regular finite element mesh. *SIAM Journal on Numerical Analysis*, 10(2), 345–363.

[43] George, A. & Liu, J. W. (1989). The evolution of the minimum degree ordering algorithm. *SIAM Review*, 31(1), 1–19.

[44] Goldberg, A. V. & Harrelson, C. (2005). Computing the shortest path: A search meets graph theory. In *Proceedings of the sixteenth annual ACM-SIAM symposium on Discrete algorithms* (pp. 156–165).: Society for Industrial and Applied Mathematics.

[45] Gonzalez, J. E., Low, Y., Gu, H., Bickson, D., & Guestrin, C. (2012). Powergraph: Distributed graph-parallel computation on natural graphs. In *Operating Systems Design and Implementation*, volume 12 (pp.2).

[46] Gonzalez, J. E., Xin, R. S., Dave, A., Crankshaw, D., Franklin, M. J., & Stoica, I. (2014). Graphx: Graph processing in a distributed dataflow framework. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)* (pp. 599–613).

[47] Groër, C., Sullivan, B. D., & Poole, S. (2011). A mathematical analysis of the r-mat random graph generator. *Networks*, 58(3), 159–170.

[48] Guo, Y., Pan, Z., & Heflin, J. (2005). Lubm: A benchmark for owl knowledge base systems. *Web Semantics: Science, Services and Agents on the World Wide Web*, 3(2), 158–182.

[49] Halin, R. (1976). S-functions for graphs. *Journal of Geometry*, 8(1-2), 171–186.

[50] Han, M. & Daudjee, K. (2015). Giraph unchained: barrierless asynchronous parallel execution in pregel-like graph processing systems. *Proceedings of the VLDB Endowment*, 8(9), 950–961.

[51] Hart, P. E., Nilsson, N. J., & Raphael, B. (1968). A formal basis for the heuristic determination of minimum cost paths. *IEEE transactions on Systems Science and Cybernetics*, 4(2), 100–107.

[52] Heggernes, P. (2006). Minimal triangulations of graphs: A survey. *Discrete Mathematics*, 306(3), 297–317.

[53] Henzinger, M. R., Klein, P., Rao, S., & Subramanian, S. (1997). Faster shortest-path algorithms for planar graphs. *journal of computer and system sciences*, 55(1), 3–23.

[54] Hong, S., Chafi, H., Sedlar, E., & Olukotun, K. (2012). Green-marl: a dsl for easy and efficient graph analysis. In *ACM SIGARCH Computer Architecture News*, volume 40 (pp. 349–362).: ACM.

[55] Hong, S., Van Der Lugt, J., Welc, A., Raman, R., & Chafi, H. (2013). Early experiences in using a domain-specific language for large-scale graph analysis. In *First International Workshop on Graph Data Management Experiences and Systems* (pp.5).: ACM.

[56] Idreos, S., Kersten, M. L., & Manegold, S. (2007). Database cracking. In *Conference on Innovative Data systems Research*, volume 3 (pp. 1–8).

[57] Iyer, S., Killingback, T., Sundaram, B., & Wang, Z. (2013). Attack robustness and centrality of complex networks. *PloS ONE*, 8(4), e59613.

[58] Kang, U., Tong, H., Sun, J., Lin, C.-Y., & Faloutsos, C. (2011). Gbase: a scalable and general graph management system. In *Proceedings of the 17th ACM SIGKDD international conference on Knowledge discovery and data mining* (pp. 1091–1099).: ACM.

[59] Kang, U., Tsourakakis, C. E., & Faloutsos, C. (2009). Pegasus: A peta-scale graph mining system implementation and observations. In *2009 Ninth IEEE International Conference on Data Mining* (pp. 229–238).: IEEE.

[60] Karypis, G. (2013). A software package for partitioning unstructured graphs, partitioning meshes, and computing fill-reducing orderings of sparse matrices. *University of Minnesota, Department of Computer Science and Engineering, Minneapolis, MN*.

[61] Karypis, G. & Kumar, V. (1998a). A fast and high quality multilevel scheme for partitioning irregular graphs. *SIAM Journal on Scientific Computing*, 20(1), 359–392.

[62] Karypis, G. & Kumar, V. (1998b). A parallel algorithm for multilevel graph partitioning and sparse matrix ordering. *Journal of Parallel and Distributed Computing*, 48(1), 71–95.

[63] Kawarabayashi, K.-i. & Mohar, B. (2007). Some recent progress and applications in graph minor theory. *Graphs and combinatorics*, 23(1), 1–46.

[64] Kepner, J., Arcand, W., Bergeron, W., Bliss, N., Bond, R., Byun, C., Condon, G., Gregson, K., Hubbell, M., Kurz, J., et al. (2012). Dynamic distributed dimensional data model (d4m) database and computation system. In *2012 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)* (pp. 5349–5352).: IEEE.

[65] Kim, M. & Leskovec, J. (2011). The network completion problem: Inferring missing nodes and edges in networks. In *Proceedings of the 2011 SIAM International Conference on Data Mining* (pp. 47–58).: SIAM.

[66] Kim, M. & Leskovec, J. (2012). Multiplicative attribute graph model of real-world networks. *Internet Mathematics*, 8(1-2), 113–160.

[67] Kulkarni, M., Burtscher, M., Casçaval, C., & Pingali, K. (2009). Lonestar: A suite of parallel irregular programs. In *Performance Analysis of Systems and Software, 2009. ISPASS 2009. IEEE International Symposium on* (pp. 65–76).: IEEE.

[68] Kulkarni, M., Pingali, K., Walter, B., Ramanarayanan, G., Bala, K., & Chew, L. P. (2007). Optimistic parallelism requires abstractions. *ACM SIGPLAN Notices*, 42(6), 211–222.

[69] Kundu, S. & Misra, J. (1977). A linear tree partitioning algorithm. *SIAM Journal on Computing*, 6(1), 151–154.

[70] Kwak, H. (2017). What is twitter, a social network or a news media? `http://an.kaist.ac.kr/traces/WWW2010.html`.

[71] Kwak, H., Lee, C., Park, H., & Moon, S. (2010). What is Twitter, a social network or a news media? In *WWW '10: Proceedings of the 19th international conference on World wide web* (pp. 591–600). New York, NY, USA: ACM.

[72] Kyrola, A., Blelloch, G. E., & Guestrin, C. (2012). Graphchi: Large-scale graph computation on just a pc. In *Operating Systems Design and Implementation*, volume 12 (pp. 31–46).

[73] Lawson, C. L., Hanson, R. J., Kincaid, D. R., & Krogh, F. T. (1979). Basic linear algebra subprograms for fortran usage. *ACM Transactions on Mathematical Software (TOMS)*, 5(3), 308–323.

[74] Leskovec, J., Chakrabarti, D., Kleinberg, J., & Faloutsos, C. (2005). Realistic, mathematically tractable graph generation and evolution, using kronecker multiplication. In *European Conference on Principles of Data Mining and Knowledge Discovery* (pp. 133–145).: Springer.

[75] Leskovec, J., Chakrabarti, D., Kleinberg, J., Faloutsos, C., & Ghahramani, Z. (2010). Kronecker graphs: An approach to modeling networks. *Journal of Machine Learning Research*, 11(Feb), 985–1042.

[76] Leskovec, J. & Faloutsos, C. (2007). Scalable modeling of real graphs using kronecker multiplication. In *Proceedings of the 24th international conference on Machine learning* (pp. 497–504).: ACM.

[77] Leskovec, J. & Krevl, A. (2014). SNAP Datasets: Stanford large network dataset collection. `http://snap.stanford.edu/data`.

[78] Liu, J. W. (1990). The role of elimination trees in sparse factorization. *SIAM Journal on Matrix Analysis and Applications*, 11(1), 134–172.

[79] Lothian, J., Powers, S., Sullivan, B. D., Baker, M., Schrock, J., & Poole, S. W. (2013). Synthetic graph generation for data-intensive hpc benchmarking: Background and framework.

[80] Lovász, L. (2006). Graph minor theory. *Bulletin of the American Mathematical Society*, 43(1), 75–86.

[81] Low, Y., Bickson, D., Gonzalez, J., Guestrin, C., Kyrola, A., & Hellerstein, J. M. (2012). Distributed graphlab: a framework for machine learning and data mining in the cloud. *Proceedings of the VLDB Endowment*, 5(8), 716–727.

[82] Low, Y., Gonzalez, J., Kyrola, A., & GraphLab, D. (2010). A new framework for parallel machine learning. In *Proceedings of the 26th Conference on Uncertainty in Artificial Intelligence (UAI 2010)* (pp. 340–349).

[83] Macko, P., Marathe, V. J., Margo, D. W., & Seltzer, M. I. (2015). Llama: Efficient graph analytics using large multiversioned arrays. In *International Conference on Data Engineering*: IEEE.

[84] Macko, P., Margo, D., & Seltzer, M. (2013). Performance introspection of graph databases. In *Proceedings of the 6th International Systems and Storage Conference* (pp.18).: ACM.

[85] Malewicz, G., Austern, M. H., Bik, A. J., Dehnert, J. C., Horn, I., Leiser, N., & Czajkowski, G. (2010). Pregel: a system for large-scale graph processing. In *2010 ACM SIGMOD International Conference on Management of Data* (pp. 135–146).: ACM.

[86] Margo, D. & Seltzer, M. (2015). A scalable distributed graph partitioner. *Proceedings of the VLDB Endowment*, 8(12), 1478–1489.

[87] Mattson, T., Bader, D., Berry, J., Buluc, A., Dongarra, J., Faloutsos, C., Feo, J., Gilbert, J., Gonzalez, J., Hendrickson, B., Kepner, J., Leiserson, C., Lumsdaine, A., Padua, D., Poole, S., Reinhardt, S., Stonebraker, M., Wallach, S., & Yoo, A. (2013). Standards for graph algorithm primitives. In *2013 IEEE High Performance Extreme Computing Conference (HPEC)* (pp. 1–2).

[88] McSherry, F. (2013). Graph analysis and hilbert space-filling curves.

[89] McSherry, F., Isard, M., & Murray, D. G. (2015). Scalability! but at what cost? In *HotOS*: Citeseer.

[90] Meyer, U. & Sanders, P. (2003). δ-stepping: a parallelizable shortest path algorithm. *Journal of Algorithms*, 49(1), 114–152.

[91] Miao, H., Liu, X., Huang, B., & Getoor, L. (2013). A hypergraph-partitioned vertex programming approach for large-scale consensus optimization. In *International Conference on Big Data* (pp. 563–568).: IEEE.

[92] Murphy, R. C., Wheeler, K. B., Barrett, B. W., & Ang, J. A. (2010). Introducing the graph 500. *Cray User's Group (CUG).*

[93] Murray, D. G., McSherry, F., Isaacs, R., Isard, M., Barham, P., & Abadi, M. (2013). Naiad: a timely dataflow system. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles* (pp. 439–455).: ACM.

[94] Narayanan, A. & Shmatikov, V. (2008). Robust de-anonymization of large sparse datasets. In *Security and Privacy, 2008. SP 2008. IEEE Symposium on* (pp. 111–125).: IEEE.

[95] Newman, M. E. (2001). The structure of scientific collaboration networks. *Proceedings of the National Academy of Sciences*, 98(2), 404–409.

[96] Nguyen, D., Lenharth, A., & Pingali, K. (2013). A lightweight infrastructure for graph analytics. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles* (pp. 456–471).: ACM.

[97] Page, L., Brin, S., Motwani, R., & Winograd, T. (1999). The pagerank citation ranking: bringing order to the web.

[98] Parter, S. (1961). The use of linear graphs in gauss elimination. *SIAM Review*, 3(2), 119–130.

[99] Pothen, A. & Toledo, S. (2004). Elimination structures in scientific computing. *Handbook on Data Structures and Applications*, (pp. 59–1).

[100] Prabhakaran, V., Wu, M., Weng, X., McSherry, F., Zhou, L., & Haradasan, M. (2012). Managing large graphs on multi-cores with graph awareness. In *USENIX Annual Technical Conference* (pp. 41–52).

[101] Price, D. d. S. (1976). A general theory of bibliometric and other cumulative advantage processes. *Journal of the American society for Information science*, 27(5), 292–306.

[102] Raghavan, U. N., Albert, R., & Kumara, S. (2007). Near linear time algorithm to detect community structures in large-scale networks. *Physical review E*, 76(3), 036106.

[103] Robertson, N. & Seymour, P. D. (1984). Graph minors. iii. planar tree-width. *Journal of Combinatorial Theory, Series B*, 36(1), 49–64.

[104] Robertson, N. & Seymour, P. D. (2004). Graph minors. xx. wagner's conjecture. *Journal of Combinatorial Theory, Series B*, 92(2), 325–357.

[105] Rodriguez, M. A. (2015). The gremlin graph traversal machine and language (invited talk). In *Proceedings of the 15th Symposium on Database Programming Languages* (pp. 1–10).: ACM.

[106] Roy, A., Mihailovic, I., & Zwaenepoel, W. (2013). X-stream: Edge-centric graph processing using streaming partitions. In *24th ACM Symposium on Operating Systems Principles* (pp. 472–488).: ACM.

[107] Salihoglu, S. & Widom, J. (2013). Gps: A graph processing system. In *25th International Conference on Scientific and Statistical Database Management*: ACM.

[108] Satish, N., Sundaram, N., Patwary, M. M. A., Seo, J., Park, J., Hassaan, M. A., Sengupta, S., Yin, Z., & Dubey, P. (2014). Navigating the maze of graph analytics frameworks using massive graph datasets. In *ACM SIGMOD International Conference on Management of Data* (pp. 979–990).: ACM.

[109] Schank, T. (2007). Algorithmic aspects of triangle-based network analysis.

[110] Seshadhri, C., Pinar, A., & Kolda, T. G. (2011). An in-depth study of stochastic kronecker graphs. In *2011 IEEE 11th International Conference on Data Mining* (pp. 587–596).: IEEE.

[111] Shi, J., Yao, Y., Chen, R., Chen, H., & Li, F. (2016). Fast and concurrent rdf queries with rdma-based distributed graph exploration. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)* (pp. 317–332).: USENIX Association.

[112] Shun, J. & Blelloch, G. E. (2013). Ligra: a lightweight graph processing framework for shared memory. In *ACM SIGPLAN Notices*, volume 48 (pp. 135–146).: ACM.

[113] Shun, J., Dhulipala, L., & Blelloch, G. E. (2015). Smaller and faster: Parallel processing of compressed graphs with ligra+. In *2015 Data Compression Conference* (pp. 403–412).: IEEE.

[114] Spielmat, D. A. & Teng, S.-H. (1996). Spectral partitioning works: Planar graphs and finite element meshes. In *Foundations of Computer Science, 1996. Proceedings., 37th Annual Symposium on* (pp. 96–105).: IEEE.

[115] Stanton, I. & Kliot, G. (2012). Streaming graph partitioning for large distributed graphs. In *18th ACM SIGKDD International Conference on Knowledge Discovery and Data mining* (pp. 1222–1230).: ACM.

[116] Sundaram, N., Satish, N., Patwary, M. M. A., Dulloor, S. R., Anderson, M. J., Vadlamudi, S. G., Das, D., & Dubey, P. (2015). Graphmat: High performance graph analytics made productive. *Proceedings of the VLDB Endowment*, 8(11), 1214–1225.

[117] Tamassia, R. (2013). *Handbook of graph drawing and visualization*. CRC press.

[118] Tauro, S. L., Palmer, C., Siganos, G., & Faloutsos, M. (2001). A simple conceptual model for the internet topology. In *Global Telecommunications Conference, 2001. GLOBECOM'01. IEEE*, volume 3 (pp. 1667–1671).: IEEE.

[119] Tian, Y., Balmin, A., Corsten, S. A., Tatikonda, S., & McPherson, J. (2013). From think like a vertex to think like a graph. *Proceedings of the VLDB Endowment*, 7(3), 193–204.

[120] Tsourakakis, C., Gkantsidis, C., Radunovic, B., & Vojnovic, M. (2014). Fennel: Streaming graph partitioning for massive scale graphs. In *7th ACM International Conference on Web Search and Data Mining* (pp. 333–342).: ACM.

[121] Tsourakakis, C. E. (2008). Fast counting of triangles in large real networks without counting: Algorithms and laws. In *Data Mining, 2008. ICDM'08. Eighth IEEE International Conference on* (pp. 608–617).: IEEE.

[122] Ullmann, J. R. (2010). Bit-vector algorithms for binary constraint satisfaction and subgraph isomorphism. *Journal of Experimental Algorithmics (JEA)*, 15, 1–6.

[123] Valiant, L. G. (1990). A bridging model for parallel computation. *Communications of the ACM*, 33(8), 103–111.

[124] Vigna, S. (2007). Stanford matrix considered harmful. *Web Information Retrieval and Linear Algebra Algorithms*, (07071).

[125] Wang, G., Xie, W., Demers, A. J., & Gehrke, J. (2013). Asynchronous large-scale graph processing made easy. In *CIDR*, volume 13 (pp. 3–6).

[126] Xin, R. S., Gonzalez, J. E., Franklin, M. J., & Stoica, I. (2013). Graphx: A resilient distributed graph system on spark. In *First International Workshop on Graph Data Management Experiences and Systems* (pp.2).: ACM.

[127] Yan, D., Cheng, J., Lu, Y., & Ng, W. (2014). Blogel: A block-centric framework for distributed computation on real-world graphs. *Proceedings of the VLDB Endowment*, 7(14), 1981–1992.

[128] Zhou, Y., Liu, L., Lee, K., & Zhang, Q. (2015). Graphtwist: fast iterative graph computation with two-tier optimizations. *Proceedings of the VLDB Endowment*, 8(11), 1262–1273.