MINING LARGE GRAPHS

David F. Gleich, Michael W. Mahoney

1 INTRODUCTION

Graphs provide a general representation or data model for many types of data where pair-wise relationships are known or thought to be particularly important.[1] Thus, it should not be surprising that interest in graph mining has grown with the recent interest in "big data." Much of the big data generated and analyzed involves pair-wise relationships among a set of entities. For example, in e-commerce applications such as with Amazon's product database, customers are related to products through their purchasing activities; on the web, web pages are related through hypertext linking relationships; on social networks such as Facebook, individuals are related through their friendships; and so on. Similarly, in scientific applications, research articles are related through citations; proteins are related via metabolic pathways, co-expression, and regulatory network effects within a cell; materials are related through models of their crystalline structure; and so on.

While many graphs are small, many large graphs are now extremely LARGE. For example, in early 2008, Google announced that it had indexed over 1 trillion URLs on the internet, corresponding to a graph with over 1 trillion nodes [Alpert and Hajaj, 2008]; in 2012, the Facebook friendship network spanned 721 million individuals and had 137 billion links [Backstrom et al., 2012]; phone companies process a few trillion calls a year [Strohm and Homan, 2013]; the human brain has around 100 billion neurons and 100 trillion neuronal connections [Zimmer, 2011]; one of the largest reported graph experiments involved 4.4 trillion nodes and around 70 trillion edges in a synthetic experiment that required one petabyte of storage [Burkhardt and Waring, 2013]; and one of the largest reported experiments with a real-world graph involved over 1.5 trillion edges [Fleury et al., 2015].

Given the ubiquity, size, and importance of graphs in many application areas, it should come as no surprise that large graph mining serves numerous roles within the large-scale data analysis ecosystem. For example, it can help us learn new things about the world, including both the chemical and biological sciences [Martin et al., 2012; Stelzl et al., 2005] as well as results in the social and economic sciences such as the Facebook study that showed that any two people in the(ir) world can be connected through approximately four intermediate individuals [Backstrom et al., 2012]. Alternatively, large graph mining produces similarity information for recommendation, suggestion, and prediction from messy data [Boldi et al., 2008; Epasto et al., 2014]; it can also tell us how to optimize a data infrastructure to improve response time [Ugander and Backstrom, 2013]; and it can tell us when and how our data are anomalous [Akoglu et al., 2010].

David F. Gleich, Purdue University, dgleich@purdue.edu
Michael W. Mahoney, University of California Berkeley, mmahoney@stat.berkeley.edu
This material was published as Gleich and Mahoney, Mining Large Graphs. In *Handbook of Big Data*, Peter Bühlmann, Petros Drineas, Michael Kane, Mark van der Laan, editors, pages 191-220. CRC Press, 2016.

[1] In the simplest case, a graph $G = (V, E)$ consists of a set V of vertices or nodes and a set E of edges, where each edge consists of an undirected pairs of nodes. Of course, in many applications one is interested in graphs that have weighted or directed edges, that are time-varying, that have additional meta-information associated with the nodes or edges, and so on.

SCOPE AND OVERVIEW

In this chapter, we will provide an overview of several topics in the general area of mining large graphs. This is a large and complicated area. Thus, rather than attempting to be comprehensive, we will instead focus on what seems to us to be particularly interesting or underappreciated algorithmic developments that will in upcoming years provide the basis for an improved understanding of the properties of moderately large to very large informatics graphs. There are many reviews and overviews for the interested reader to learn more about graph mining; see, e.g., [Chakrabarti and Faloutsos, 2006; Bornholdt and Schuster, 2003]. An important theme in our chapter is that large graphs are often very different than small graphs and thus intuitions from small graphs often simply do not hold for large graphs. A second important theme is that, depending on the size of the graph, different classes of algorithms may be more or less appropriate. Thus, we will concern ourselves primarily with *what* is (and is not) even possible in large graph mining; we'll describe *why* one might (or might not) be interested in performing particular graph mining tasks that are possible; and we will provide brief comments on *how* to make a large graph mining task work on a large distributed system such as MapReduce cluster or a Spark cluster. Throughout, we'll highlight some of the common challenges, we'll discuss the heuristics and procedures used to overcome these challenges, and we'll describe how some of these procedures are useful outside the domain for which they were originally developed.[2] At several points, we will also highlight the relationships between seemingly distinct methods and unexpected, often implicit, properties of large-scale graph algorithms.

2 PRELIMINARIES

When data is[3] represented as a graph, the objects underlying the relationships are called *nodes* or *vertices*, and the relationships are called *edges*, *links*, or *arcs*. For instance, if we are considering a data set representing web pages and the links from one page to another, then the vertices represent the web pages and the edges represent those links between pages. The result is a directed graph because edges between pages need not be reciprocal. Thus, the idea with representing the data as a graph is that we can abstract the details of a particular domain away into the formation of a graph. Then we can take a domain specific question, such as "How do I understand phone calling patterns?", and we can rephrase that as a question about the vertices and edges in the graph that is used to model the data.

Let us note that there are often many ways to turn a set of data into a graph. There could be multiple types of possible edges corresponding to different types of relationships among the objects. This is common in what is known as semantic graph analysis and semantic graph mining. Determining what edges to use from such a graph is a fascinating problem that can often have a dramatic

[2]We have attempted in this preliminary work to strike a balance between providing accurate intuition about our perspective on large graph mining and precise formal statements. In the main text, we skew towards accurate intuition, and in some cases we provide additional technical caveats for the experts in the footnotes.

[3]Or "are"—aside from the linguistic issue, one of the challenges in developing graph algorithms is that graphs can be used to represent a single data point as well as many data points. For example, there is $N = 1$ web graph out there; but graphs are also used to represent correlations and similarities between many different data points, each of which is represented by a feature vector. Different research areas think about these issues in very different ways.

effect on the result and/or the scalability of an algorithm. In order to keep our discussion contained, however, we will assume that the underlying graph has been constructed in such a way that the graph mining tasks we discuss make sense on the final graph. Having a non-superficial understanding of what graph mining algorithms actually do and why they might or might not be useful often provides excellent guidance on choosing nodes/edges to include or exclude in the graph construction process.

2.1 GRAPH REPRESENTATIONS

The canonical graph we analyze is $G = (V, E)$ where V is the set of vertices and E is the set of edges. We will use n to denote the number of vertices. We assume that the number of edges is $O(n)$ as well, and we will use this for complexity results. If we wish to be specific, the number of edges will be $|E|$. Graphs can be either directed or undirected, although some algorithms may not make sense on both types.

While for tiny graphs, e.g., graphs containing fewer than several thousand nodes, one can take a "bird's eye" view and think about the entire graph since, e.g., it can be stored in processor cache, for larger graphs it is important to worry about how the data are structured to determine how algorithms run. There are two important representations of graphs that we need to discuss that have a large impact on what is and is not possible with large graph mining.

Edge list. The edge list is simply a list of pairs of vertices in the graph, one pair for each edge. Edges can appear in any order. There is no index, so checking whether or not an edge exists requires a linear scan over all edges. This might also be distributed among many machines. Edge lists are common in graphs created based on translating data from sources such as log files into relationships.

Adjacency list. Given a vertex, its adjacency list is the set of neighbors for that vertex. An adjacency list representation allows us to query for the set of neighbors in a time that we will consider constant.[4] Adjacency lists are common when graphs are an explicit component of the original data model.

The adjacency list is the most flexible format because an adjacency list can always serve as an edge list through a simple in-place transformation. In comparison, although building the adjacency list representation from an edge list is a linear-time operation, it may involve an expensive amount of data movement within a distributed environment.[5]

2.2 GRAPH MINING TASKS

We'll use the following representative problems/algorithms to help frame our discussion below.

RANDOM WALK STEPS. A random walk in a graph moves from vertex to vertex by randomly choosing a neighbor. For most adjacency list representations one step of a random walk is a constant time operation.[6] Running millions

[4]Various implementations and systems we consider may not truly guarantee *constant time* access to the neighborhood set of vertices, e.g., it may be $O(\log n)$, or be constant in some loosely amortized sense, but this is still a useful approximation for the purposes of distinguishing access patterns for algorithms in large graph mining.

[5]This data movement is a great fit for Google's MapReduce system.

[6]This is not always guaranteed because selecting a random neighbor may be an $O(d)$ operation, where d is the degree of the node, depending on the implementation details.

of random walk steps given an adjacency list representation of a large graph is easy,[7] and these steps can be used to extract a small region of a massive graph nearby that seed [Page et al., 1999; Pan et al., 2004].

CONNECTED COMPONENTS. Determining the connected components of a graph is a fundamental step in most large graph mining pipelines. On an adjacency list this can be done using breadth-first search in $O(n)$ time and memory. On an edge list this can also be done in $O(n)$ time and memory, assuming that the diameter of the graph does not grow with the size of the graph, by using semi-ring iterations [Kepner and Gilbert, 2011] that we will discuss in Section 6.

PAGERANK. PageRank [Page et al., 1999] is one of a host of graph centrality measures [Koschützki et al., 2005] that give information to address the question "What are the most important nodes in my graph?" See Ref. [Gleich, 2014] for a long list of examples of where it has been successfully used to analyze large graphs. Just as with connected components, it takes $O(n)$ time and memory to compute PageRank, in either the adjacency list representation or edge list representation. Computing PageRank is one of the most common primitives used to test large graph analysis frameworks (e.g., [Malewicz et al., 2010; Gonzalez et al., 2012; Shun and Blelloch, 2013]).

EFFECTIVE DIAMETER. The effective diameter of a graph is the length of the longest path necessary to connect 90% of the possible node pairs.[8] Understanding this value guides our intuition about short paths between nodes. Generating an accurate estimate of the effective diameter is possible in $O(n)$ time and memory using a simple algorithm with a sophisticated analysis [Palmer et al., 2002; Boldi et al., 2011b].

EXTREMAL EIGENVALUES. There are a variety of matrix structures associated with a graph. One of the most common is the adjacency matrix, denoted $\boldsymbol{A}$, where $A_{i,j} = 1$ if there is an edge between vertices i and j and $A_{i,j} = 0$ otherwise.[9] Another common matrix is the normalized Laplacian, denoted $\mathcal{L}$, where $\mathcal{L}_{i,i} = 1$, and $\mathcal{L}_{i,j} = 1/\sqrt{\text{degree}(i) \cdot \text{degree}(j)}$ if there is an edge between i and j, and $\mathcal{L}_{i,j} = 0$ otherwise. The largest and smallest eigenvalues and eigenvectors of the adjacency or normalized Laplacian matrix of a graph reveal a host of graph properties, from a network centrality score known as "eigenvector centrality" to the Fiedler vector that indicates good ways of splitting a graph into pieces [Mihail, 1989; Fiedler, 1973]. The best algorithms for these problems use the ARPACK software [Lehoucq et al., 1997], which includes sophisticated techniques to lock eigenvalues and vectors after they have converged. This method

[7] Current research efforts are devoted to running random walks with restarts for millions of seeds concurrently on edge list representations of massive graphs [Bahmani et al., 2011].

[8] The diameter of a graph is the length of the longest shortest path to connect all pairs of nodes that have a valid path between them. This measure is not reliable/robust, as many graphs contain a small number of outlier pieces that increase the diameter a lot. Clearly, the exact percentile is entirely arbitrary, but choosing 90% is common. A parameter-less alternative is the average distance in the graph.

[9] Formally, all matrices associated with a graph require a mapping of the vertices to the indicates 1 to n; however many implementations of algorithms with matrices on graphs need not create this mapping explicitly. Instead, the algorithm can use the natural vertices identifiers with the implicit understanding that the algorithm is equivalent to some ordering of the vertices.

TABLE 1. Several common graph primitives and their time and memory complexity.

random walk with restart	$O(1)$ time and $O(1)$ memory
connected components	$O(n)$ time and memory
PageRank	$O(n)$ time and memory
extremal eigenvalues	$O(n \log n)$ time and memory
triangle counting	$O(n\sqrt{n})$ time and memory
all-pairs shortest paths	$O(n^3)$ time and $O(n^2)$ memory

would require something like $O(nk \log n)$ time and memory to compute reasonable estimates of the extremal k eigenvalues and eigenvectors.[10]

TRIANGLE COUNTING. Triangles, or triples of vertices (i, j, k) where all are connected, have a variety of uses in large graph mining. For instance, counting the triangles incident to a node helps indicate the tendency of the graph to have interesting groups, and thus feature in many link prediction, recommendation systems, and anomaly detections schemes. Given a sparse graph (such that there are order n edges) computing the triangles takes $O(n\sqrt{n})$ work and memory.

ALL-PAIRS PROBLEMS. Explicit all-pairs computations (shortest paths, commute times, graph kernels [Kondor and Lafferty, 2002]) on graphs are generally infeasible for large graphs. Sometimes there are algorithms that enable fast (near constant-time) queries of any given distance pair or the closest k-nodes query; and there is a class of algorithms that generate so-called Nyström approximations of these distances that yields near-constant time queries. Finding exact scalable methods for these problems is one of the open challenges in large graph mining.

There are of course many other things that could be computed, e.g., the δ-hyperbolicity properties of a graph with an $\Theta(n^4)$ algorithm [Adcock et al., 2013]; but these are many of the most representative problems/algorithms in which graph miners are interested. See Table 1 for a brief summary.

2.3 A CLASSIFICATION OF LARGE GRAPHS

We now classify large graphs based on their size. As always with a classification, this is only a rough guide that aids our intuition about some natural boundaries in how properties of graph mining change with size. We will use the previous list of tasks from Section 2.2 to provide context for what is and is not possible as graphs get larger. Again, let n be the number of vertices in the graph. Also, recall that realistic graphs are typically *extremely* sparse, e.g., roughly tens to at most hundreds of edges per node on average; thus, the number of nodes and the number of edges are both $O(n)$.

Small graphs (under 10k vertices). For the purposes of this chapter, a small graph has fewer than 10,000 vertices. At this size, standard algorithms run easily. For instance, computing all-pairs, shortest paths takes $O(n^3)$ time and $O(n^2)$ memory. This is not a problem for any modern computer.[11]

[10] This bound is not at all precise, but a fully precise bound is not a useful guide to practice; and this statement represents a working intuition for how long it takes compared to other ideas.

[11] That being said, this does not mean that the naïve cubic algorithm is best, e.g., faster

A large Small graph (10k-1M vertices). Moving beyond small graphs reveals a regime of what we will call "large small" graphs. These are graphs where $O(n^2)$ time algorithms are possible, but $O(n^2)$ memory algorithms become prohibitive or impossible.[12] We consider these graphs more strongly associated with small graphs though, because there are many tasks, such as diameter computations, that can be done exactly on these graphs, with some additional time. Two differences are worth noting: (i) many of the most important properties of graphs in this regime (and larger) are very different than the properties of small graphs [Leskovec et al., 2009]; and (ii) even if quadratic time computations are possible, they can be challenging, and they can become prohibitive if they are used in an exploratory data analysis mode or as part of a large cross validation computation. Thus, some of the algorithms we will discuss for larger graphs can be used fruitfully in these situations.

Small Large graphs (1M-100M vertices). This chapter is about large graph mining, and in many ways the transition between small and large graphs occurs around one million vertices. For instance, with a graph of 5 million vertices, algorithms that do $O(n^2)$ computation are generally infeasible without specialized computing resources. That said, with appropriate considerations being given to computational issues, graphs with between 1M and 100M vertices are reasonably easy to mine with fairly sophisticated techniques given modest computing resources. The basic reason for this is the extreme sparsity of real world networks. Real-world graphs in this size regime typically have an average degree between 5 and 100. Thus, even a large real-world graph would have at most a few billion edges. This would consume a few gigabytes of memory and could easily be tackled on a modern laptop or desktop computer with 32GB of memory. For instance, computing a PageRank vector on a graph with 60M vertices takes a few minutes on a modern laptop. We view this regime attractively as it elicits many of the algorithmic and statistical challenges of mining much larger graphs, without the programming and databases and systems overhead issues of working with even larger problems.

Large graphs (100M-10B vertices). With a few hundred million or even a few billion vertices, the complexity of running even simple graph algorithms increases. However, in this regime, even the largest public networks will fit into main memory on large shared memory systems with around 1TB of main memory.[13] This was the motivation of the Ligra project [Shun and Blelloch, 2013]. Also, the effective diameter computations on the Facebook networks with 70B edges were done on a single shared memory machine [Backstrom et al., 2012].

LARGE graphs (over 10B vertices). With over 10 billion vertices, even shared memory systems are unable to cope with the scale of the networks. The particular number defining this threshold will no doubt become outdated at some point, but there are and will continue to be sufficiently massive networks where shared memory machines no longer work and specialized distributed techniques

algorithms that have been developed for much larger graphs can implicitly regularize against noise in the graph [Andersen et al., 2006; Gleich and Mahoney, 2014]. Thus, they might be better even for rather small graphs, even when more expensive computations are possible.

[12] On large distributed high performance computers, algorithms with $O(n^2)$ memory and $O(n^3)$ computation are possible on such graphs; however, these systems are not commonly used to mine graphs in this size range.

[13] It may be faster to work with them on distributed systems, but our point is that shared memory implementations are possible and far easier than distributed implementations.

are required. This is the case, for instance, with the entire web graph. Note that while global mining tasks may require a distributed memory computer, it is often possible to extract far smaller subsets of these LARGE graphs that are only Large and can be easily handled on a shared memory computer.[14] The types of problems that we can expect to solve on extremely LARGE graphs are only very simple things like triangles, connected components, PageRank, and label propagation [Burkhardt and Waring, 2013; Fleury et al., 2015].

2.4 LARGE GRAPH MINING SYSTEMS

Mining large graphs can be done with custom software developed for each task. However, there are now a number of graph mining systems (and there continue to be more that are being developed) that hope to make the process easier. These systems abstract standard details away and provide a higher-level interface to manipulate algorithms running on a graph. Three relevant properties of such systems are the following.

BATCH OR ONLINE. A batch system must process the entire graph for any task, whereas an online system provides access to arbitrary regions of the graph more quickly.

ADJACENCY OR EDGE LIST. A system that allows adjacency access enables us to get *all* neighbors of a given node. A system that allows edge list access only gives us a set of edges.

DISTRIBUTED OR CENTRALIZED. If the graph mining system is distributed, then systems can only access local regions of the graph that are stored on a given machine, and the data that are needed to understand the remainder of the graph may be remote and difficult to access; a centralized system has a more holistic view of the graph.

For instance, a MapReduce graph processing system is a batch, distributed system that provides either edge list [Cohen, 2009; Kang et al., 2009] or adjacency access [Lin and Dyer, 2010]; GraphLab [Gonzalez et al., 2012] is a distributed, online, adjacency system; and Ligra [Shun and Blelloch, 2013] is an online, edge list, centralized system.

2.5 SOURCES FOR DATA

One of the vexing questions that often arises is: "Where do I get data to test my graph algorithm?" Here, we highlight a few sources.

STANFORD NETWORK ANALYSIS PROJECT (SNAP)

`https://snap.stanford.edu/data/index.html`

This website has a variety of social network data up to a few billion edges.[15] There is also a host of metadata associated with the networks there, including some ground-truth data for real-world communities in large networks.

[14] This is likely the best strategy for most of those who are interested in non-trivial analytics on LARGE graphs.

[15] Detailed empirical results for these and many other informatics graphs have been reported previously [Leskovec et al., 2009].

LABORATORY FOR WEB ALGORITHMICS (LAW)

http://law.di.unimi.it/datasets.php

The LAW group at the University of Milano maintains data sets for use with their graph compression library. They have a variety of web graphs and social networks up to a few billion edges.

WEB GRAPHS: CLUEWEB AND COMMON CRAWL

http://www.lemurproject.org/clueweb12/webgraph.php/

http://http://webdatacommons.org/hyperlinkgraph/

Both the ClueWeb group and Common Crawl groups maintains web graphs from their web crawling and web search engine projects. The most recent of these has 3.5 billion vertices and 128 billion edges.[16] The link graph is freely available while access to the entire crawl information including the page text requires purchasing access (ClueWeb) or may be access via Amazon's public data sets (Common Crawl).

THE UNIVERSITY OF FLORIDA SPARSE MATRIX COLLECTION

http://www.cise.ufl.edu/research/sparse/matrices/

There is a close relationship between sparse matrices and graph theory through the adjacency (or Laplacian) matrix. The Florida sparse matrix repository contains many adjacency matrices for many real-world graphs. These range in size from a few thousand vertices up to hundreds of millions of edges. For instance, the data sets from the recent DIMACS challenge on graph partitioning [Bader et al., 2013] are all contained in this repository. Many of these data sets come from much more structured scientific computing applications.

3 CANONICAL TYPES OF LARGE-SCALE GRAPH MINING METHODS

There are three canonical types of large-scale graph mining methods that cover the vast majority of use cases and algorithms. At root, these describe data access patterns; and depending on the implementation details (some of which we will discuss in the next few sections) they can be used to implement a wide range of graph algorithms for a wide range of graph problems.

3.1 GEODESIC NEIGHBORHOOD-BASED GRAPH MINING

Geodesic neighborhood-based computations involve a vertex, its neighboring vertices, and the edges among them. They are among the easiest to scale to large graphs, and they support a surprising variety of different applications, e.g., anomaly detection [Akoglu et al., 2010]. These methods are typically very "easy" to scale to large graphs when working simultaneously with all of the vertices; and to determine whether or not an algorithm that uses this primitive will scale to even larger graphs the main issue is the size of the highest degree node. Two examples of tasks that can be accomplished with geodesic neighborhood-based graph mining are the triangle counting and computing extremal eigenvalues of all neighborhoods.

[16] Web graphs require special treatment in terms of number of nodes due to the presence of a crawling frontier.

3.2 DIFFUSION NEIGHBORHOOD-BASED GRAPH MINING

Diffusion neighborhood-based computations can be thought of as a "softer" or "fuzzy" version of geodesic neighborhood-based computations.[17] They can be used to answer questions such as "What does the region of this graph look like around this specific object?" These methods are also "easy" to scale to massive graphs because they are methods that do *not* need to explore the entire graph. For example, random walks with restart are an instance of this idea.[18] One should think of running these diffusion neighborhoods on only only $O(\log n)$ or $O(\sqrt{n})$ of the nodes instead of on all nodes.

3.3 GENERALIZED MATRIX-VECTOR PRODUCTS GRAPH MINING

The bulk of our chapter will be focused on what is possible with large graph mining that must use the entire graph. Because such graphs have billions or trillions of edges, nearly-linear time algorithms are the only algorithms that can run on such massive graphs. Despite this limitation, there are a tremendous number of useful mining tasks that can be done in near linear time. For instance, we can compute an accurate estimate of the effective diameter of a graph in near linear time, which is what Facebook used to determine that there are roughly 4 degrees of separation between individuals [Backstrom et al., 2012]. Thus, effective diameter, extremal eigenvalues, PageRank, connected components, and host of other ideas [Mahoney et al., 2012] are all instances of generalized matrix-vector product graph mining. The importance of this primitive is frequently suggested in the literature for scalable graph algorithms [Kepner and Gilbert, 2011; Kang et al., 2009]. There is a problem with high-degree nodes with large neighborhoods (think of Barak Obama in Twitter, or a molecule like water in a cell that interacts with a huge number of other molecules) for straightforward use of this type of mining, but there are many ideas about how to address this challenge [Kang and Faloutsos, 2011; Gonzalez et al., 2012].

4 MINING WITH GEODESIC NEIGHBORHOODS

A geodesic neighborhood of a vertex is the induced subgraph of a vertex v and all of its neighbors within r-steps. A surprisingly wide and useful set of graph mining tasks are possible by analyzing these geodesic neighborhoods.

4.1 SINGLE NEIGHBORHOODS

Perhaps the simplest large graph mining task involves the 1-step geodesic neighborhood of a single node called the target. This is useful for visualizing a small piece of a large network to build intuition about that target node. In the context of social networks, this one step neighborhood is also called the *egonet*. We can then perform a variety of analyses on that neighborhood to understand its role. Common examples of target nodes are suspicious individuals in a social network and curious proteins and metabolites in biological networks. A single neighborhood can also reveal structural holes predicted by social theory [Burt, 1995].

[17] In a slightly more precise sense, these are spectral-based, or diffusion-based, relaxations of vertex neighborhood methods.

[18] There is a way to reduce this task to the previous geodesic primitive, but the examples in subsequent sections show that the two scenarios have a different "flavor," and they can lead to very different results in practice.

Implementation. Any mining task that depends on a single neighborhood is well-suited to systems that permit fast neighbor access through the adjacency list. Note that getting a single neighborhood would require two passes over an edge-list or adjacency-list representation stored in a file or in a batch system. Thus, single neighborhood queries are inefficient in such systems.

4.2 ALL NEIGHBORHOODS

A richer class of graph mining tasks involve using *all* of the 1-step geodesic neighborhoods, i.e., the 1-step geodesic neighborhood of all of the nodes. For example, consider the task of counting the number of triangles of a massive network. Each triangle in the network is an edge in a 1-step neighborhood that does not involve the target node. Thus, by measuring properties of all neighborhoods, we can compute the number of triangles each vertex is associated with as well as the clustering coefficients. More complex all-neighborhoods analysis also enables various types of graph summaries and motif detection.

Implementation. A computation involving all neighborhoods is easy to parallelize by recognizing that each individual neighborhood computation is independent. Forming some of the local neighborhood may be expensive, although this work can be balanced in a variety of standard ways.

Approximations. Forming all of the 1-step neighborhoods takes work that scales as $O(n\sqrt{n})$ for sparse graphs – see the discussion in Section 2. As graphs become LARGE, even this level of computation isn't feasible. Streaming computations are an active research area that provides an alternative (see Section 8.1).

Example: Oddball anomaly detection One example of how neighborhood mining works in practice is given by the so-called "Oddball anomaly detection" method [Akoglu et al., 2010]. The goal of this graph mining task is to find anomalous nodes in a network. To do this, one can compute the following statistics for each local neighborhood graph:

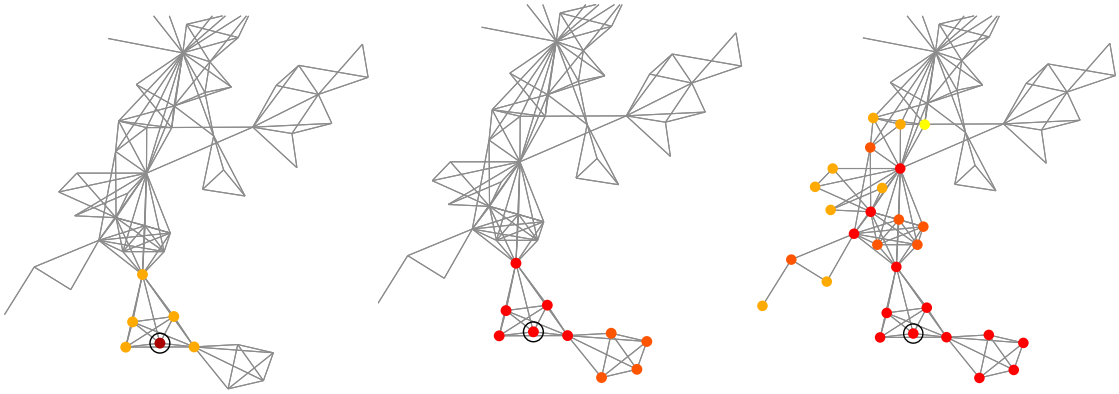
- The number of vertices of the neighborhood
- The number of edges of the neighborhood
- The total weight of the neighborhood (for weighted graphs)
- The largest eigenvalue of the adjacency matrix for the neighborhood

The result of each analysis is a single real-valued number that is a feature of the node having to do with the vertex neighborhood. Thus, the overall result is a set of 4 features associated with each vertex.[19] Oddball then applies an outlier detection method to this four-dimensional data set, and it is able to distinguish a variety of types of anomalous nodes in the Twitter network [Akoglu et al., 2010].

4.3 COMPLEXITIES WITH POWER-LAW GRAPHS

One of the challenges that arises when doing all-neighborhoods analysis of real-world networks is the highly skewed distribution of vertex degrees. These networks possess a few vertices of extremely high degree. Constructing and manipulating these vertex neighborhoods then becomes challenging. For instance, on the Twitter network there are nodes with millions of neighbors such as President Obama around the time of his reelection in 2012. Performing local analysis of this neighborhood

[19] That is, this method involves associating a feature vector with each node, where the labels of the feature vector provide information about the node and its place in the graph.



(a) Geodesic neighborhood (b) Small diffusion neighborhood (c) Larger diffusion neighborhood

FIGURE 1. An illustration (in (a)) of a geodesic neighborhood in Newman's netscience graph [Newman, 2006] around the circled node, with neighbors colored yellow. Diffusion neighborhoods (in (b) and (c)) of the circled node for comparison; nodes are colored based on their diffusion value, red is large and orange and yellow are smaller. Note that the diffision neighborhood does not extend beyond the natural borders in the graph.

itself becomes a large graph mining task. This same problem manifests itself in a variety of different ways. For batch computations in MapReduce, it is called the curse of the last reducer [Suri and Vassilvitskii, 2011]. There are no entirely satisfying, general solutions to these skewed degree problems, and it is a fundamental challenge for large-scale machine learning and data analysis more generally. Strategies to handle them include using additional computational resources for these high-degree nodes such as large shared memory systems [Shun and Blelloch, 2013] or vertex and edge splitting frameworks [Gonzalez et al., 2012].

5 MINING WITH DIFFUSION NEIGHBORHOODS

The use of graph diffusions in graph mining is typically a formalization of the following idea:

> *Importance flows from a source node along edges of the graph to target nodes.*

The mechanics of how the diffusion behaves on an edge of the graph determines the particular type of diffusion. Well known examples are:

- the PageRank diffusion [Page et al., 1999];
- the Katz diffusion [Katz, 1953];
- the heat kernel diffusion [Kondor and Lafferty, 2002; Chung, 2007];
- the truncated random walk diffusion [Spielman and Teng, 2008];

These diffusions, and many many minor variations, are frequently invented and reinvented under a host of different names [Lin and Cohen, 2010]: in biology networks, for instance, a PageRank diffusion may also be called an information diffusion [Lisewski and Lichtarge, 2010]; spectral methods and local spectral methods implement variants of this idea [Andersen et al., 2006]; diffusion-based information is also known as *guilt by association* [Koutra et al., 2011]; and so on.

A diffusion neighborhood can be thought of as a soft or fuzzy version of a geodesic distance-based neighborhood. It is a neighborhood that, intuitively,

follows the "shape" of the graph instead of following the geodesic distance.[20] We illustrate the difference on a simple example in Figure 1. These diffusion neighborhoods are commonly used for large graph mining because they can be computed extremely quickly. For instance, it's possible to compute the diffusion neighborhood of a random node in the Clueweb12 dataset (with 60B edges) on a modest server that costs less than $7,500 within a second or two. Importantly, just as with a geodesic distance neighborhood, finding these diffusion neighborhoods can be done without exploring the entire graph.

Although there is a formal mathematical setting for diffusions, here we maintain an informal discussion.[21] Suppose that source of a diffusion is only a single node u in the graph. Then the PageRank diffusion models where dye (or heat or mass or ...) is injected at u and flows under the following dynamics.

1. At a node, the dye is evenly divided among all neighbors of u; and
2. along each edge, only β of the dye survives transmission.

This type of diffusion is often thought to be appropriate for modeling how a scarce resource such as attention, importance, influence, or association could flow in a graph, where each edge is partially uncertain. A diffusion neighborhood is then the region of the graph where the values of the diffusion, or some normalized value of the diffusion, exceed a threshold. Using a small value for the threshold will select a large region of the graph, whereas using a large value will select a small region of the graph. In Figure 1, we illustrate two different diffusion neighborhoods on a small network by varying the threshold used to create them.

Diffusion neighborhoods are one of the most scalable primitives in large graph mining, and they can be used to support a large variety of tasks. Similarly to a geodesic neighborhood of a vertex being easy to compute, so too, given adjacency access, the diffusion neighborhood is easy to compute in the same setting. One can use the following two related strategies.

1. The Andersen-Chung-Lang push procedure (Section 5.1).
2. The random walks with restart method (Section 5.2).

Before describing these methods, we first review a small number of the many applications of diffusion neighborhood ideas.

Example: Guilt-by-association mining In guilt-by-association mining, the graph describes relationships in some type of connected system where one believes there to be a functional connection between nodes. This type of mining is commonly used with biological networks where connections are putative relationships between biological objects (species, genes, drugs, etc.) or with social networks where the edges are potential influence links. In biology, diffusion neighborhoods answer the question: "What might I be missing if I'm interested in a particular node?" The result is a set of predictions about what should be predicted based on a diffusion from a particular node [Lisewski and Lichtarge, 2010; Koutra et al., 2011; Morrison et al., 2005].

[20] In some cases, one can make a precise connection with notions of diffusion distance or resistance distance, and in other cases the connection is only informal.

[21] The majority of this section applies to all types of diffusion neighborhoods using adaptations of the ideas to the nature of those diffusions [Bonchi et al., 2012; Ghosh et al., 2014; Vigna, 2009; Baeza-Yates et al., 2006].

Example: Semi-supervised learning Semi-supervised learning is closely related to the guilt-by-association mining. The general idea is the same, but the setup changes slightly. As a canonical example, consider a large graph with a few labeled nodes. One often believes that the labels should remain relatively smoothly-varying over the edges, and so the semi-supervised learning problem is to propagate the small set of known labels through the rest of the graph [Zhou et al., 2003]. A diffusion neighborhood mining scheme produces what should be a high precision set where the label applies. Note that we must use the scalable strategies listed below to propagate the diffusion through the network; straightforward techniques and naïve implementations often do not scale.

Example: Local community detection Communities in large graphs are sets of vertices that are in some sense internally cohesive and/or separate from other vertices.[22] Diffusion neighborhoods are an important component of many community detection methods [Andersen et al., 2006; Leskovec et al., 2009].[23] These methods identify a community around a seed node by propagating a diffusion and then truncating it to a high quality set of nodes through a procedure called a sweep cut. Repeating this process for multiple nodes can yield high quality overlapping communities [Whang et al., 2013] on small large (as well as small and large) graphs.

5.1 ANDERSEN-CHUNG-LANG PUSH METHODS

The Andersen-Chung-Lang (ACL) push method is a scalable method to propagate, or evaluate, a diffusion, given a seed node or set of seed nodes [Andersen et al., 2006].[24] It maintains a list of vertices where the diffusion propagation needs to be updated and a set of diffusion values. At each step, it picks a node and acquires the update, then "pushes" the influence of the update to the node's neighbors. (Hence the name.) The algorithm ends once all remaining updates are below a threshold. The result is a set of diffusion values on a small set of nodes. When push is used for community detection, one can use these values to generate a set that satisfies a worst-case approximation bound guaranteeing the set returned by the algorithm is not too far away from the best possible.[25]

Implementation. Given adjacency access, it is possible to scale this method to arbitrary sized graphs as the method needs to access the adjacency information for a constant number of nodes. Also, a variety of graph mining systems support updating a diffusion only on the *needs-to-be-updated* set [Shun and Blelloch, 2013; Gonzalez et al., 2012; Nguyen et al., 2013].

5.2 RANDOM WALKS WITH RESTART

One alternative to using the ACL push procedure is to employ a Monte Carlo approach. Diffusions are associated with random-walk processes and we can simply simulate the random walk to propagate the diffusion [Avrachenkov et al., 2007; Borgs et al., 2013; Pan et al., 2004]. Tracking where the random walk

[22] It would require another chapter to discuss communities in large networks in appropriate depth [Schaeffer, 2007; Leskovec et al., 2009; Xie et al., 2013].

[23] In particular, spectral algorithms that are commonly used to detect communities have strong connections with diffusion-based neighborhood methods [Jeub et al., 2015].

[24] Variants and extensions exist for a host of other diffusions [Bonchi et al., 2012; Ghosh et al., 2014; Kloster and Gleich, 2014].

[25] This is done with a sweep cut procedure and uses a localized instance of a Cheeger inequality.

moves in the graph provides most of the information on the diffusion, and a few thousand random walks provide acceptable accuracy in many cases [Avrachenkov et al., 2007; Borgs et al., 2013].

Implementation. Implementation with adjacency access is trivial, as it simply involves a sequence of neighborhood queries that constitute a random walk. The situation is different for implementations without neighborhood access. If the graph is stored in a manner that does not permit efficient neighborhood access, then any of the techniques in the next section on generalized matrix-vector products will work. However, these are often inefficient. It may take 20-100 "passes" over the graph in order to evaluate the diffusion from a single seed. Diffusion neighborhoods are usually computed for multiple seeds (usually between 10 and 10000), and recent research gives a few strategies to compute these random walks simultaneously [Bahmani et al., 2011].

6 MINING WITH GENERALIZED MATRIX-VECTOR PRODUCTS

Generalized matrix-vector products are one of the most flexible and scalable ways to mine large graphs [Kang et al., 2009; Kepner and Gilbert, 2011]. As the name suggests, these methods emerge from a generalized notion of a matrix-vector product on generalizations of the adjacency matrix of a graph. Scalable methods for matrix-vector products date back to the dawn of computing [Troyer, 1968]; and the use with generalized matrix-vector products was recognized early [Carré, 1971]. All of these methods and algorithms apply to batch systems rather than online systems.[26]

A matrix-vector product $\mathbf{y} = \boldsymbol{A}\mathbf{x}$ with the adjacency matrix of a graph expresses the computational primitive:

$$y_v = \sum_{u \in N(v)} x_u, \tag{1}$$

where $N(v)$ is the neighbor set of node v. If $\boldsymbol{A}$ represents a directed graph, then matrix-vector products with $\boldsymbol{A}$ sum over the out-neighbors of v, whereas matrix-vector products with $\boldsymbol{A}^T$ sum over the in-neighbors of v. In a more general sense, a matrix-vector product can be seen as a special case of the following computational primitive.

Update vertex v's data based on a function f of its neighbors data. (2)

The standard matrix vector product uses summation as the function. Iterative sequences of these operations, with different functions, compute connected components, single-source shortest paths, label propagation, effective diameters, and distance histograms, as we'll see shortly. Operations such as minimum spanning trees [Kepner and Gilbert, 2011], maximum weight matchings [Bayati et al., 2008], and message passing methods [Zhang and Moore, 2014] fit into the same framework as well.

[26] A standard use case is to use the result of a batch generalized matrix-vector product algorithm to enable or accelerate an online operation. For instance, compute recommendations using generalized matrix-vector products and store some results for online queries.

Graph system support The generalized matrix vector product idea is incorporated in a few different software libraries under a different guise. Pregel expresses this concept through the idea of a vertex and edge programming interface [Malewicz et al., 2010]. Both GraphLab and Ligra adopt this same type of vertex and edge programming [Gonzalez et al., 2012; Shun and Blelloch, 2013]. These vertex and edge programs specify the aggregation operation on v as well as the information transmitted from u to v. Pegasus makes the matrix-vector interpretation explicit [Kang et al., 2009], as does Combinatorial BLAS [Buluç and Gilbert, 2011].

Two types of functions We will consider two classes of functions f. The first class is a *reduction operation*, which is generalization of the summation operation. Reduction operations are associative functions of their data. That is, we can apply f to a subset of the neighbor information and then later integrate that with the rest of the information. The second class is just a *general function* f that can do anything with the neighbor information. One simple example related to what we'll see below is computing the *median* value of all neighbors. This is not associative and depends on the entire set of elements. This distinction is important, as various optimizations employed by graph systems, such as the vertex-splitting in GraphLab [Gonzalez et al., 2012], only work for reduction functions.

6.1 ALGORITHMS WITH STANDARD MATRIX-VECTOR PRODUCTS

Even the standard matrix-vector product is key to many large graph mining methods. We give two examples below.

Implementation. A parallel matrix-vector product is easy to implement on a centralized graph system as each vertex runs its own update equation independently. (This assumes that all updates are independent, as they are in all of the following examples.) Distributed implementations require a means to *move* data along the edges of the graph. These are often precomputed at the start of a procedure given the current data distribution, or maintained with some type of distributed hash-table. Matrix-vector products can easily work with adjacency or edge-list information, which makes them a highly flexible graph mining primitive.

Example: PageRank The global PageRank vector is the result of a diffusion—with seeds *everywhere* in the graph. This yields information about the important nodes from *all* vertices. It is usually computed on a directed graph using the iteration:

$$\textbf{Initialize: } x_v^{(\text{start})} = 1, \qquad \textbf{Iterate: } x_v^{(\text{next})} = \alpha \sum_{u \in N^{\text{in}}(v)} x_u^{(\text{cur})}/d_v + 1,$$

where $N^{\text{in}}(v)$ is the set of in-neighbors. This is just a small adjustment to the standard matrix-vector product above. Usually α is taken to be 0.85,[27] and 20 or 30 iterations suffice for most purposes for this value of α.

Example: Extremal eigenvalues and approx. triangles The extremal eigenvalues and eigenvectors of the adjacency matrix and normalized Laplacian

[27]Note, though, that α really is just a regularization parameter, and so its value should be chosen according to a model selection rule. See [Gleich, 2014] for a discussion on values of α.

matrix can be provided by the ARPACK software [Lehoucq et al., 1997], and its parallel PARPACK variant [Maschhoff and Sorensen, 1996], or through simple subspace iterative methods. The key to all of these ideas is to perform a sequence of matrix-vector products with the adjacency or Laplacian matrix. Extremal eigenvalues of the adjacency matrix provide an accurate estimate of the total number of triangles in a graph at the cost of a few matrix-vector products [Tsourakakis, 2008]. The extremal eigenvectors of the normalized Laplacian matrix indicate good ways to split the graph into pieces [Mihail, 1989; Fiedler, 1973].

6.2 ALGORITHMS WITH SEMI-RING MATRIX-VECTOR PRODUCT

Our first generalization of matrix-vector products involves changing what *addition* and *multiplication* by using a semi-ring.[28] [29] We use $\oplus$ and $\otimes$ to denote the "changed" addition and multiplication operations to distinguish them from the usual operations. In which case, a classic example is the *min-plus* semi-ring, where we set $a \oplus b = \min(a, b)$ and $a \otimes b = a + b$. Each of these new operations has their own set of *identity elements*, just like adding 0 and multiplying by 1 do not change the answer. The identity elements in *min-plus* are: $\textcircled{0} = \infty$ and $\textcircled{1} = 0$. Note that using the *min-plus* semi-ring means that we continue to work with numbers, but just change the way these numbers are manipulated by these operations.

A wide variety of classic graph algorithms can be expressed as generalized matrix-vector products using a semi-ring. This idea is more fully explored in the edited volume: *Graph Algorithms in the Language of Linear Algebra* [Kepner and Gilbert, 2011]. Note that for a general matrix and vector $\boldsymbol{A}$ and $\mathbf{x}$ the matrix-vector $\mathbf{y} = \boldsymbol{A}\mathbf{x}$ produces the element-wise computation:

$$y_i = A_{i,1} \times x_1 + A_{i,2} \times x_2 + \cdots + A_{i,n} \times x_n.$$

The idea with a semi-ring generalized matrix vector product is that we replace all of these algebraic operations with their semi-ring counterparts:

$$y_i = A_{i,1} \otimes x_1 \oplus A_{i,2} \otimes x_2 \oplus \cdots \oplus A_{i,n} \otimes x_n.$$

Implementation. Implementations of these semi-ring iterative algorithms work just like the implementations of the standard matrix-vector products described above. The only difference is that the actual operations involved change. Note that the semi-ring methods are all *reduction functions* applied to the neighbor data because semi-rings are guaranteed to be associative.

Example: Single-source shortest paths In fact, using the *min-plus* algebra we can encode the solution of a single-source shortest path computation.[30] Recall

[28] More formally, a semi-ring is a set that is closed under two binary operations: $\otimes$ and $\oplus$ along with their respective identity elements: $\textcircled{1}$, the multiplicative identity element and $\textcircled{0}$, the additive identity element. These operations must be associative and distributive.

[29] This generalization may seem peculiar to readers who have not seen it before. It is similar to the usual matrix-vector product in that it can be formally written in the same way. Relatedly, if communication is a more precious resource than computation, then algorithms that communicate in similar ways—which is what writing algorithms in terms of primitives such as matrix-vector multiplication is essentially doing—can potentially provide more sophisticated computation (than the usual "multiply, then sum" that the usual matrix-vector product performs) at little or no additional time cost. This is the case, and considering algorithms that can be expressed in this way, i.e., as matrix-vector products with non-standard semi-ring matrix-vector multiplication, i.e., perform different computations once the bits have been communicated, is *much* more powerful than considering just the usual matrix-vector product.

[30] This can also be used to compute a breadth-first search.

that this operation involves computing the shortest path distance from a source vertex s to all other vertices v. Let $A_{v,u}$ be the distance between vertex v and u, $A_{v,u} = ⓪$ if they are not connected, and $A_{v,v} = ①$ otherwise. Consider the iteration:

$$\textbf{Initialize:} \quad x_v^{(\text{start})} = \begin{cases} ① & v = s \\ ⓪ & v \neq s, \end{cases}$$

$$\textbf{Iterate:} \quad x_v^{(\text{next})} = A_{v,1} \otimes x_1^{(\text{cur})} \oplus A_{v,1} \otimes x_2^{(\text{cur})} \oplus \cdots \oplus A_{v,1} \otimes x_n^{(\text{cur})}$$
$$x_v^{(\text{next})} = \min_{u \in N(v) \cup \{v\}} [A_{v,u} + x_u^{(\text{cur})}].$$

At each iteration, we find the shortest path to all vertices that are one link further than each previous step. This iteration is closely related to Dijkstra's algorithm without a priority queue.

Example: Connected components There is also a *min-times* semi-ring, where $a \oplus b = \min(a, b)$ and $a \otimes b = a \times b$ (the regular multiplication operation). Here, $① = 1$ and $⓪ = \infty$. Let $A_{v,u} = ①$ if v and v have an edge, $A_{v,u} = ⓪$ otherwise, and let $A_{v,v} = ①$. Using this semi-ring, we can compute the connected components of a graph:

$$\textbf{Initialize:} \quad x_v^{(\text{start})} = \text{unique id for } v$$

$$\textbf{Iterate:} \quad x_v^{(\text{next})} = A_{v,1} \otimes x_1^{(\text{cur})} \oplus A_{v,1} \otimes x_2^{(\text{cur})} \oplus \cdots \oplus A_{v,1} \otimes x_n^{(\text{cur})}$$
$$x_v^{(\text{next})} = \min_{u \in N(v) \cup \{v\}} [x_u^{(\text{cur})}].$$

Once the values do not change in an iteration, all vertices in the same connected component will have the same value on their vertex. If the vertices are labeled 1 to n, then using those labels suffice for the unique ids.

6.3 GENERAL UPDATES

The most general of the generalized matrix-vector product operations apply arbitrary functions to the neighbor data. For instance, in the example we'll see with label propagation clustering, each neighbor sends labels to a vertex v and the vertex takes the *most frequent* incoming label. This operation is not a reduction as it depends on all of the neighboring data, which eliminates some opportunities to optimize intermediate data transfer.[31] Each of the three examples we will see use different functions, but the unifying theme of these operations is that each step is an instance of Eqn. (2) for some function f. Consequently, all of the parallelization, distribution, and system support is identical between all of these operations.

Implementation. These operations are easy to implement both for adjacency and edge list access in centralized or distributed settings. Getting the information between neighbors, i.e., the communication, is the difficult step. In distributed settings, there may be a great deal of data movement required and optimizing this is an active area of research.

[31] In MapReduce environments, one example of this optimization is the use of local combiners to reduce the number of key-value pairs sent to the reducers.

Example: Label propagation for clusters and communities Label propagation is a method to divide a graph into small clusters or communities [Raghavan et al., 2007] that has been used to optimize distributing graph vertices to processors [Ugander and Backstrom, 2013] and to optimize the ordering of vertices in adjacency matrices [Boldi et al., 2011a]. It works by giving each vertex a unique id (like in the connected components algorithm) and then having vertices iteratively assume the id of the most frequently seen label in their neighborhood (where ties are broken arbitrarily). As we already explained, this is an instance of a generalized matrix-vector product. A few iterations of this procedure suffice for most graphs. The output depends strongly on how ties break and a host of other implementation details.

Example: Distance histograms and average distance A distance histogram of a network shows the number of vertex pairs separated by k links. It is a key component to the effective diameter computation and also the average distance. Recall that the effective diameter is the smallest k such that (say) 90% of all pairs are connected k links. If these computations are done exactly, we need one shortest-path computation from each node in the graph, which has a terrible complexity. However, the distance histogram and the neighborhood function of a node can both be approximated, with high accuracy, using generalized matrix-vector products. The essential idea is the Flajolet-Martin count sketch [Flajolet and Martin, 1985] and the HyperLogLog counter [Flajolet et al., 2007] to *approximate* the number of vertices at distance exactly k from each vertex. Both of these approximations maintain a small amount of data associated with each vertex. This information is aggregated in a specific way at each vertex to update the approximation.[32] The aggregation is formally a reduction, and so this method can take advantage of those optimizations. These techniques have been demonstrated on Large graphs with almost 100 billion edges.

6.4 EDGE-ORIENTED UPDATES

We have described these generalized matrix-vector products as updating quantities at each vertex. There is no limitation to vertex-oriented updates only. The same ideas apply edge-oriented updates by viewing them as generalized matrix-vector products with the *line-graph* or *dual-graph* of the input graph. In the dual graph, we replace each edge with a vertex and connect each new vertex to the vertices that represent all adjacent edges.

6.5 COMPLEXITIES WITH POWER-LAW GRAPHS

Power-law, or highly skewed, degree distribution pose problems for efficient implementations of generalized matrix-vector products at scale. The PowerGraph extension of GraphLab [Gonzalez et al., 2012] contains a number of ideas to improve performance with formal reductions and power-law graphs based on vertex splitting ideas that create virtual "copies" of vertices with lower degree. These large degree vertices, however, tend not to prohibit implementations of these ideas on large graphs and only make them slower.

[32] The specifics of the algorithm are not our focus here, see [Boldi et al., 2011b] for a modern description.

6.6 IMPLEMENTATIONS IN SQL

Generalized matrix-vector products are possible to implement even in traditional database systems that implement SQL [Cohen et al., 2009]. Using a distributed SQL database such as Greenplum will evaluate these tasks with reasonable efficiency even for large graphs. As an example, here we illustrate how to perform the generalized matrix-vector product for connected components in SQL. The graph is stored as a table that provides edge-list access.[33] The columns head and tail indicate the start and end of each edge. The initial vector is stored as a table with a vertex id and the unique id associated with it (which could be the same).

```
edges :  id | head | tail
x :  id | comp
```

In the iteration, we create the vector $x^{(\text{next})}$ from x:

```
CREATE TABLE xnext AS (
  SELECT e.tail AS id, MIN(x.comp) AS comp
  FROM edges e INNER JOIN x ON e.head = x.id
  GROUP BY e.tail );
```

This query takes the graph structure, joins it with the vector such that each component of the table x is mapped to the head of each edge. Then we group-by the tail of each edge and take the *MIN* function over all components. This is exactly what the iteration in the connected components example did.

7 LIMITATIONS AND TRADEOFFS IN MINING LARGE GRAPH

In many cases, individuals who employ graph mining tools want to obtain some sort of qualitative understanding of or insight into their data. This soft and subtle goal differs in important ways from simply using the graph to obtain better prediction accuracy in some well-defined downstream machine learning task. In particular, it can differ from the use of the algorithms and techniques we've focused on in the previous sections, e.g., when those algorithms are used as black-box components in larger analytics frameworks such as various machine learning pipelines that are increasingly common. A common temptation in this setting is to use the intuition obtained from mining small graph, assuming or hoping that the intuition thereby obtained is relevant or useful for much larger graphs. *In general, it is not.* Using intuition obtained from small graphs can lead to qualitatively incorrect understanding of the behavior and properties of larger graphs; and it can lead to qualitatively incorrect understanding of the behavior of algorithms that run on larger graphs.

At root, the reason that our intuition fails in larger graphs is that—for typical informatic graphs—the "local" structure of a large graph is distinct from and qualitatively different than its "global" structure.[34] A small subgraph of size roughly 100 vertices *is* a global structure in a graph with only 1000 vertices; but it is a local structure in a graph with millions of vertices. As typical graphs get

[33] We assume that the graph structure is undirected, so both edges (u, v) and (v, u) are stored, and that each vertex has a self-loop.

[34] Informally, by local structure we mean, e.g., the properties of a single node and its nearest neighbors; while by global structure we mean the properties of the graph as a whole or that involve a constant fraction of the nodes of the graph.

larger and larger, much of the local structure does not change or grow [Leskovec et al., 2009; Jeub et al., 2015]; instead, one simply observes more and more varied pockets of local structure. Hence one aspect of our point: a fundamental difference between small graph mining and large graph mining is that for large graphs, the global structure of the graph (think of the "whole graph") and its local structure (think of "vertex neighborhoods") are very different; while, for small graphs, these two types of structures are much more similar.

Moreover, in a large realistic graph, these local structures connect up with each other in ways that are nearly random/quasirandom, or just slightly better than random/quasirandom. A good example of this to keep in mind is the case of community detection, as first described in [Leskovec et al., 2009] and as elaborated upon in [Jeub et al., 2015]. The result of those exhaustive empirical investigations was that large real-world graphs do *not* have *good* large communities. This is very different than working with graphs of a few thousand nodes, where good clusters and communities of size 5%-25% of the graph do exist. As the graphs get larger and larger, the good clusters/communities stay roughly the same size. Thus, if we insist on finding good communities then we may find hundreds or thousands of good small communities in graphs with millions or more of nodes, but we won't find good large communities. That being said, in a large realistic graph, there certainly are large groups of nodes (think 10% of the graph size) with *better than random* community structure (e.g., [Ugander and Backstrom, 2013; Whang et al., 2013]); and there certainly are large groups of nodes (again, think 10% of the graph size) with slightly better community quality score (for whatever score is implicitly or explicitly being optimized by the community detection algorithm that one decides to run) than the community quality score that an arbitrary 10% of the nodes of the graph would have; and there are many methods that find these latter structures.

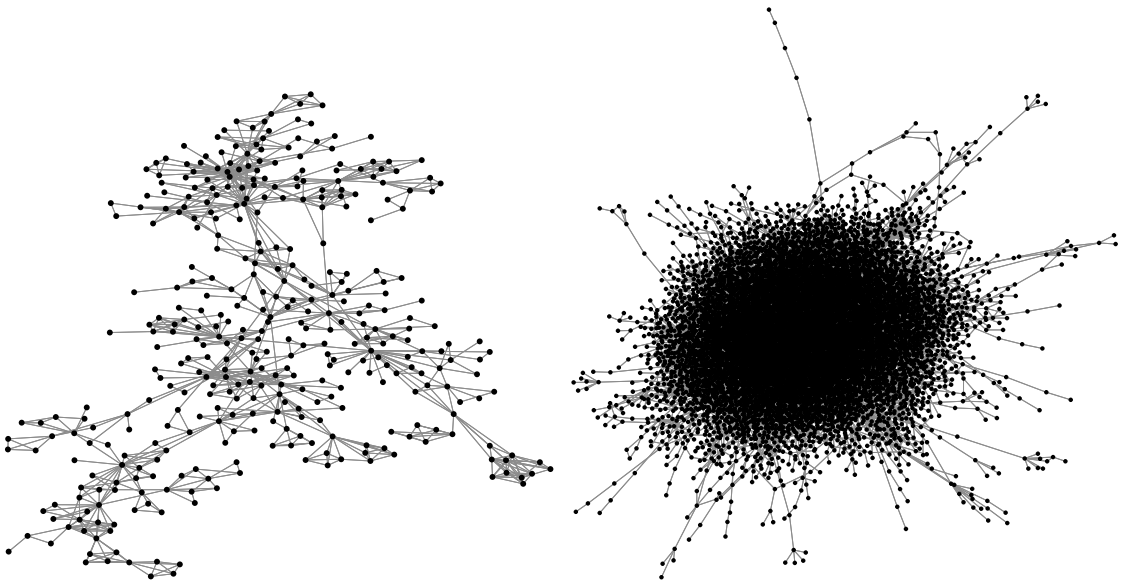
In the remainder of this section, we illustrate three examples in which the qualitative difference between large graphs and small graphs manifests and our natural small graph intuition fails: graph drawing, "viral" propagation, and modularity-based communities.

7.1 GRAPH DRAWING

Perhaps the pictorially most vivid illustration of the difference between small graphs and large graphs is with respect to visualization and graph drawing. There is no shortage of graph layout ideas that proclaim to visualize *large graphs* [Adai et al., 2004; Martin et al., 2011]. While graph layout algorithms are often able to find interesting and useful structures in graphs with around one thousand vertices, they almost universally fail at finding any useful or interesting structure in graphs with more than 10,000 vertices.[35] The reason for this is that graph drawing algorithms attempt to show both the local and global structure simultaneously by seeking an arrangement of vertices that respects the local edge structure for all edges in the graph. This is not possible for graphs with strong expander-like properties [Leskovec et al., 2009; Jeub et al., 2015]. Relatedly, as we will explain shortly in terms of communities, there is surprisingly little global structure to be found.

A better strategy for large graphs is to use summary features to reveal the

[35] These "failures" can be quite beautiful, though, from an artistic point of view.



(a) A 250 vertex graph

(b) A 10,000 vertex graph

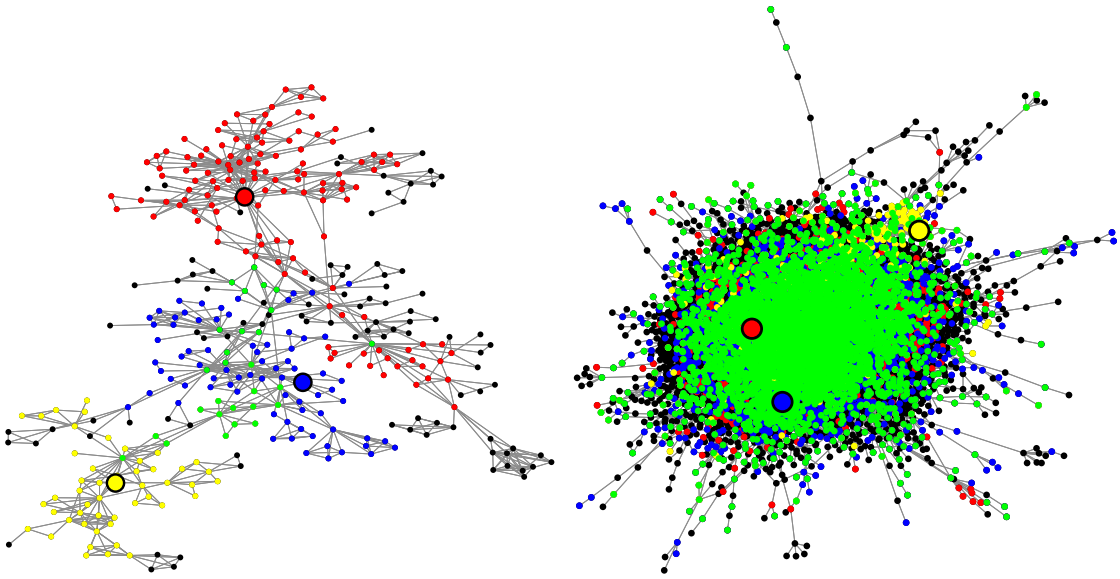
FIGURE 2. Here, we show the result of drawing two graphs: a small one, and a large one. The small graph drawing (a) shows considerable local structure and reveals some overall topology of the relationships (the graph is Newman's network science collaborators [Newman, 2006]). The large graph drawing (b) shows what is affectionately called a "hairball" and does not reveal any meaningful structure (the graph is a human protein-protein interaction network [Klau, 2009]). The failure of graph drawing to show any meaningful structure in large networks is an example of how we should not draw intuition from small graphs when mining large graphs.

graph structure. This is essentially how the oddball anomaly detection method works [Akoglu et al., 2010]. Each vertex is summarized with a few small local features. The result is a set of less-artistic-but-more-informative scatter plots that show multivariate relationships among the vertices. Anomalies are revealed because they are outliers in the space of local features. Hive plots of networks are an attempt to make these multivariate plots reveal some of the correlation structure among these attributes on the edges [Krzywinski et al., 2012].

7.2 VIRAL PROPAGATION

Another qualitative goal in large graph mining is to understand the spread of information within a network. This is often called *viral propagation* due to its relationship with how a virus spreads through a population. This is also a property that is fundamentally different between large graphs and small graphs. Consider the two graphs from Figure 2. For each graph, we place three "seed" nodes in these graphs, and we look at how far information would spread from these seeds to the rest of the graph in three steps.[36] The results of this simple experiment are in Figure 3, and it too illustrates this difference between small and large graphs. In small graphs, each of the viral propagations from the source nodes find their own little region of the graph; each region can be meaningfully interpreted, and there is only a very little overlap, between different regions. In the large graph, the viral propagations quickly spread and intersect and overlap throughout the graph.

[36]Here, we are looking at the three-step geodesic neighborhoods of each of the three seeds, but we observe similar results with diffusion-based dynamics and other dynamics.



(a) Propagation in a 250 vertex graph

(b) Propagation in a 10,000 vertex graph

FIGURE 3. Here, we illustrate three steps of geodesic propagation in a small graph (a) and a larger graph (b). The propagations start from three nodes (the big red, yellow, and blue ones). Green nodes are overlaps among the yellow and blue propagations. These figures show that in large graphs, propagations and diffusions quickly spread everywhere, whereas in small graphs, propagations and diffusions stay somewhat isolated. (We did not draw all edges in (b) which causes some colored nodes to appear out of "nowhere" in the "arms" of the figure.) The qualitatively different connectivity properties between large and small graphs is an example of how we should not draw intuition from small graphs when mining large graphs.

This qualitative difference is of fundamental importance, and is not limited to our relatively-simple notion of information propagation; instead, it also holds much more generally for more complex diffusions [Jeub et al., 2015, Figures 12 and 13].

7.3 COMMUNITIES AND MODULARITY

Community detection is, for many, the holy grail of graph mining. Communities, or clusters, are thought to reveal or hint at deeper structures and deeper design principles that help to understand or explain a graph. Most people start off by saying that communities are sets of nodes that in some sense have more and/or better connections internally than with the remainder of the graph. Conductance is probably the combinatorial quantity that most-closely captures the intuition underlying this bicriteria [Schaeffer, 2007; Leskovec et al., 2009; Jeub et al., 2015]. Another popular community quality metric is known as modularity [Newman and Girvan, 2004]. Here, we discuss what the modularity objective is and what structures the modularity objective finds in small graphs and in large graphs. As we will see, the types of structures that the modularity objective finds in small versus large graphs are qualitatively different.

Preliminaries For a set of vertices $S \subseteq V$ we use $\bar{S}$ to denote its complement. The volume of a set is a very simple measure of how much vertex information is in that set:

$$\mathrm{vol}(S) = \sum_{i \in S} d_i,$$

where d_i is the degree of node i. We follow the convention $\mathrm{vol}(G) = \mathrm{vol}(V)$ to denote the total volume of the graph. The *edges* function counts the number of edges between subsets of vertices and counts *both* edges:

$$\mathrm{edges}(S,T) = \sum_{i \in S, j \in T} A_{i,j} \qquad \text{and} \qquad \mathrm{edges}(S) = \mathrm{edges}(S,S).$$

The cut function measures the size of the interface between S and $\bar{S}$:

$$\mathrm{cut}(S) = \mathrm{edges}(S, \bar{S}) = \mathrm{cut}(\bar{S}). \tag{3}$$

(The function $\mathrm{cut}(S)$ is often thought to be a trivial or uninteresting measure for the cluster quality of a set S since it often returns singletons, even for graphs that clearly have good clusters.) Note that we have the following relationships:

$$\mathrm{edges}(S) = \mathrm{vol}(S) - \mathrm{edges}(S, \bar{S}) = \mathrm{vol}(S) - \mathrm{cut}(S),$$

$$\mathrm{vol}(S) = \mathrm{vol}(G) - \mathrm{vol}(\bar{S}).$$

We use a partition to represent a set of communities. A partition $\mathcal{P}$ of the vertices consists of disjoint subsets of vertices:

$$\mathcal{P} = \{S_1, \ldots, S_k\} \qquad S_i \cap S_j = \emptyset : i \neq j \qquad \bigcup_j S_j = V.$$

Modularity definition The modularity score for a vertex partition of a graph quantifies how well each group in the partition reflects the structure of an idealized *module* or community of the graph. The analogy comes from an engineering standpoint: a good component or independent module of a complex system should have an internal structure that is non-random. The same analogy is thought to apply to a community: a good community should have more internal structure than purely random connections. The modularity score Q of a subset of vertices S codifies this intuition:

$$Q(S) = \frac{1}{\mathrm{vol}(G)} \left(\mathrm{edges}(S) - \frac{1}{\mathrm{vol}(G)} \mathrm{vol}(S)^2 \right). \tag{4}$$

The term $(1/\mathrm{vol}(G))\,\mathrm{vol}(S)^2$ is the *expected* number of edges among vertices in S, assuming that edges are randomly distributed with the probability of an arbitrary edge (i,j) proportional to $d_i d_j$. Thus, modularity should be *large* when we find a set of vertices that looks non-random. The modularity score of a partition of the graph is then defined to be the sum of modularity scores for its constituent pieces:

$$Q(\mathcal{P}) = \sum_{S \in \mathcal{P}} Q(S).$$

Modularity as a cut measure Here, we will reformulate the modularity functions $Q(S)$ and $Q(\mathcal{P})$ in terms of the cut function of Eqn. (3),[37] and we will describe the implications of this reformulation for finding good communities in small versus large graphs.

[37] The following material was originally derived in collaboration with Ali Pinar at Sandia National Laboratories. He graciously allowed us to include the material with only an acknowledgment.

Consider, first, two-way partitions. For convenience's sake, let $\nu = \frac{1}{\mathrm{vol}(G)}$. Note that then:

$$\begin{aligned} Q(S) &= \nu(\underbrace{\mathrm{vol}(S) - \mathrm{cut}(S)}_{=\mathrm{edges}(S)} - \nu\,\mathrm{vol}(S)\,\underbrace{(\mathrm{vol}(G) - \mathrm{vol}(\bar{S}))}_{=\mathrm{vol}(S)}) \\ &= \nu(\nu\,\mathrm{vol}(S)\,\mathrm{vol}(\bar{S}) - \mathrm{cut}(S)). \end{aligned}$$

From this, we have that $Q(S) = Q(\bar{S})$ because $\mathrm{cut}(S) = \mathrm{cut}(\bar{S})$. Consider the modularity of a two-way partition and observe:[38]

$$\begin{aligned} Q(\mathcal{P}_2) &= \frac{1}{2}[Q(S) + Q(\bar{S}) + Q(S) + Q(\bar{S})] \\ &= \frac{\nu}{2}(\mathrm{edges}(S) - \nu\,\mathrm{vol}(S)^2 + \mathrm{edges}(\bar{S}) - \nu\,\mathrm{vol}(\bar{S})^2 \\ &\quad + 2\nu\,\mathrm{vol}(S)\,\mathrm{vol}(\bar{S}) - 2\,\mathrm{cut}(S)) \\ &= \frac{\nu}{2}\left(\mathrm{vol}(S) + \mathrm{vol}(\bar{S}) - 4\,\mathrm{cut}(S) + \nu(\mathrm{vol}(S) - \mathrm{vol}(\bar{S}))^2\right). \end{aligned}$$

Hence,

$$Q(S) = \frac{1}{4} - \frac{\nu}{4}\left(4\,\mathrm{cut}(S) + \nu(\mathrm{vol}(S) - \mathrm{vol}(\bar{S}))^2\right).$$

From this formulation of the objective we conclude:

THEOREM 1 *The best two way modularity partition corresponds to finding a subset* S *that* minimizes $\mathrm{cut}(S) + \nu/4(\mathrm{vol}(S) - \mathrm{vol}(\bar{S}))^2$.

In words, a two-way modularity partition is a minimum cut problem with a size constraint in terms of total volume. The constraint or bias toward having $\mathrm{vol}(S) = \mathrm{vol}(\bar{S})$ is extremely strong, however, and thus there is a very strong bias toward finding very well-balanced clusters, whether or not those clusters satisfy the intuitive bicriteria that communities should be sets of nodes that have more and/or better connections internally than with the remainder of the graph.

Consider, next, multi-way partitions; and we see that the generalization of modularity to multi-way partitions is equally illuminating:

$$Q(\mathcal{P}) = \sum_{S\in\mathcal{P}} Q(S) = \frac{|\mathcal{P}|}{4} - \frac{\nu}{4}\sum_{S\in\mathcal{P}}\left[4\,\mathrm{cut}(S) + \nu(\mathrm{vol}(S) - \mathrm{vol}(\bar{S}))^2\right],$$

where $|\mathcal{P}|$ is the number of partitions. An equivalent formulation helps to make the magnitude of the terms more clear:

$$\mathrm{vol}(G)Q(\mathcal{P}) = |\mathcal{P}|\frac{\mathrm{vol}(G)}{4} - \sum_{S\in\mathcal{P}}\left[\mathrm{cut}(S) + (\nu/4)(\mathrm{vol}(S) - \mathrm{vol}(\bar{S}))^2\right],$$

In words, when considering a multi-way partitioning problem with the modularity objective, adding a new community yields a bonus of $\mathrm{vol}(G)/4$, whereas the cost of this addition is proportional to the cut and the difference in volume. More concretely, optimal modularity partitions for the multi-way partitioning problem will provide a strong bias toward finding many clusters of roughly equal size, whether or not those clusters satisfy the intuitive bicriteria of being community-like, unless there are extremely good cuts in the network.

The point here is the following. In small graphs, such as those on the left of Figures 2 and 3 that lead to a nice visualization, the two terms of Eqn. (4)

[38] This result is derived in a slightly convoluted way, but we have yet to devise a more concise proof.

empirically capture the bicriteria that communities are sets of nodes that have more and/or better connections internally than with the remainder of the graph. In large graphs, however, this is typically not the case, and it is not the case for reasons that are fundamental to the structure of the modularity objective. There are applications in which it is of interest to find large clusters that, while not being particularly good, are slightly better than other clusters that are worse, and in those cases using modularity or another objective that provides an extremely strong bias toward finding well-balanced clusters might be appropriate. It is, however, very different than the intuition one obtains by applying the modularity objective on small graphs.

8 LARGE GRAPH MINING IN THE FUTURE

Large graph mining is a growing field, and there are many excellent ideas we cannot discussion in depth. We will conclude this chapter by highlighting two active research areas that are particularly exciting for their prospect to impact our mining capabilities in future years.

8.1 MINING WITH STREAMING PROCEDURES

Streaming procedures for graphs take as input a stream of edge insertions or edge deletions and must maintain an accurate or approximate representation of some aspect of the entire graph at any point in time. For instance, a simple measure might be the number of edges. These methods become complex because edges may be repeatedly inserted or deleted, but the count should not reflect these duplicate operations. Graph streams are highly related to batch methods for edge-list structures, and *variants of streaming algorithms may be the best algorithm to run even when the entire graph is available in memory.*[39] For instance, it is possible to compute accurate estimates of graph motif counts on graph streams [Jha et al., 2013]. We expect these procedures to be useful for rapid graph summarization methods. And there are a host of recent results on the opportunities of graph streaming procedures [McGregor, 2014]. As highlighted in that survey, a weakness in the graph streaming literature is that streaming algorithms tend to require undirected graphs. Most large graphs are directed.

8.2 MINING WITH GENERALIZED MATRIX-MATRIX PRODUCTS

Generalizations of matrix-matrix products are a challenging class of graph mining computations that apply to many *all-pairs* problems. All-pairs shortest paths and all-pairs commute time are two different methods to calculate "distances" between all pairs of vertices in a graph. Shortest paths operate on the graph structure exactly and use geodesic distance. Commute times are a distance based on the expected time for a random walk to visit a distant node and return. Graph kernels are a more general setting for commute times that enable a variety of notions of distance and affinity [Kondor and Lafferty, 2002]. All of these schemes are intractable to compute exactly for large graphs as the output information is $O(n^2)$. However, there are algorithms that enable fast (near constant-time) queries of

[39] The algorithmic-statistical issues underlying this observation are analogous to those underlying our empirical and theoretical results showing that the ACL push method is often the method of choice even for rather small graphs where more expensive diffusion-based procedures are certainly possible to perform.

any given distance pair or the closest k-nodes query. Moreover, there is a class of algorithms that generate so-called Nyström approximations of these distances that yields near-constant time queries. Finding scalable methods for these problems is one of the open challenges in large graph mining, and the research landscape of these methods is filled with approximations and estimations. For example, one of the best methods for link prediction in social networks are based on the *Katz matrix*, which is the result of a sequence of matrix-matrix products [Sui et al., 2013]. These products were approximated in order to make the computation efficient.

ACKNOWLEDGMENTS

DFG would like to acknowledge the NSF through award CCF-1149756 for providing partial support. MWM would like to acknowledge the Army Research Office, the DARPA XDATA and GRAPHS programs, and the Department of Energy for providing partial support for this work.

REFERENCES

[Adai et al., 2004] A. T. Adai, S. V. Date, S. Wieland, and E. M. Marcotte. *LGL: creating a map of protein function with an algorithm for visualizing very large biological networks.* J Mol Biol, 340 (1), pp. 179–190, 2004. doi:10.1016/j.jmb.2004.04.047. Cited on page 20.

[Adcock et al., 2013] A. B. Adcock, B. D. Sullivan, O. R. Hernandez, and M. W. Mahoney. *Evaluating OpenMP tasking at scale for the computation of graph hyperbolicity.* In *Proc. of the 9th IWOMP*, pp. 71–83. 2013. Cited on page 5.

[Akoglu et al., 2010] L. Akoglu, M. McGlohon, and C. Faloutsos. *oddball: Spotting anomalies in weighted graphs*. In *Advances in Knowledge Discovery and Data Mining*, pp. 410–421. Springer Berlin / Heidelberg, 2010. doi:10.1007/978-3-642-13672-6_40. Cited on pages 1, 8, 10, and 21.

[Alpert and Hajaj, 2008] J. Alpert and N. Hajaj. *We knew the web was big ....* Official Google Blog. Available online http://googleblog.blogspot.com/2008/07/we-knew-web-was-big.html, 2008. Accessed on July 11, 2009. Cited on page 1.

[Andersen et al., 2006] R. Andersen, F. Chung, and K. Lang. *Local graph partitioning using PageRank vectors*. In *Proceedings of the 47th Annual IEEE Symposium on Foundations of Computer Science*. 2006. Cited on pages 6, 11, and 13.

[Avrachenkov et al., 2007] K. Avrachenkov, N. Litvak, D. Nemirovsky, and N. Osipova. *Monte carlo methods in pagerank computation: When one iteration is sufficient*. SIAM J. Numer. Anal., 45 (2), pp. 890–904, 2007. doi:10.1137/050643799. Cited on pages 13 and 14.

[Backstrom et al., 2012] L. Backstrom, P. Boldi, M. Rosa, J. Ugander, and S. Vigna. *Four degrees of separation*. In *Proceedings of the 4th Annual ACM Web Science Conference*, pp. 33–42. 2012. doi:10.1145/2380718.2380723. Cited on pages 1, 6, and 9.

[Bader et al., 2013] D. A. Bader, H. Meyerhenke, P. Sanders, and D. Wagner, editors. *Graph Partitioning and Graph Clustering. 10th DIMACS Implementation Challenge Workshop.*, American Mathematical Society, 2013. Cited on page 8.

[Baeza-Yates et al., 2006] R. Baeza-Yates, P. Boldi, and C. Castillo. *Generalizing PageRank: Damping functions for link-based ranking algorithms*. In *Proceedings of the 29th annual international ACM SIGIR conference on Research and development in information retrieval (SIGIR2006)*, pp. 308–315. 2006. doi:10.1145/1148170.1148225. Cited on page 12.

[Bahmani et al., 2011] B. Bahmani, K. Chakrabarti, and D. Xin. *Fast personalized pagerank on mapreduce*. In *Proceedings of the 2011 international conference on Management of data*, pp. 973–984. 2011. doi:10.1145/1989323.1989425. Cited on pages 4 and 14.

[Bayati et al., 2008] M. Bayati, D. Shah, and M. Sharma. *Max-product for maximum weight matching: Convergence, correctness, and lp duality*. Information Theory, IEEE Transactions on, 54 (3), pp. 1241–1251, 2008. doi:10.1109/TIT.2007.915695. Cited on page 14.

[Boldi et al., 2008] P. Boldi, F. Bonchi, C. Castillo, D. Donato, A. Gionis, and S. Vigna. *The query-flow graph: Model and applications*. In *Proceedings of the 17th ACM Conference on Information and Knowledge Management*, pp. 609–618. 2008. doi:10.1145/1458082.1458163. Cited on page 1.

[Boldi et al., 2011a] P. Boldi, M. Rosa, M. Santini, and S. Vigna. *Layered label propagation: A multiresolution coordinate-free ordering for compressing social networks*. In *Proceedings of the 20th WWW2011*, pp. 587–596. 2011a. doi:10.1145/1963405.1963488. Cited on page 18.

[Boldi et al., 2011b] P. Boldi, M. Rosa, and S. Vigna. *Hyperanf: Approximating the neighbourhood function of very large graphs on a budget*. In *Proceedings of the*

20th International Conference on World Wide Web, pp. 625–634. 2011b. doi:10.1145/1963405.1963493. Cited on pages 4 and 18.

[Bonchi et al., 2012] F. Bonchi, P. Esfandiar, D. F. Gleich, C. Greif, and L. V. Lakshmanan. *Fast matrix computations for pairwise and columnwise commute times and Katz scores*. Internet Mathematics, 8 (1-2), pp. 73–112, 2012. doi:10.1080/15427951.2012.625256. Cited on pages 12 and 13.

[Borgs et al., 2013] C. Borgs, M. Brautbar, J. Chayes, and S.-H. Teng. *Multi-scale matrix sampling and sublinear-time pagerank computation*. Internet Mathematics, Online, 2013. doi:10.1080/15427951.2013.802752. Cited on pages 13 and 14.

[Bornholdt and Schuster, 2003] S. Bornholdt and H. G. Schuster, editors. *Handbook of Graphs and Networks: From the Genome to the Internet*, John Wiley & Sons, 2003. Cited on page 2.

[Buluç and Gilbert, 2011] A. Buluç and J. R. Gilbert. *The Combinatorial BLAS: design, implementation, and applications*. International Journal of High Performance Computing Applications, 25 (4), pp. 496–509, 2011. arXiv:http://hpc.sagepub.com/content/25/4/496.full.pdf+html, doi:10.1177/1094342011403516. Cited on page 15.

[Burkhardt and Waring, 2013] P. Burkhardt and C. Waring. *An NSA big graph experiment*. Technical Report NSA-RD-2013-056002v1, National Security Agency, 2013. Cited on pages 1 and 7.

[Burt, 1995] R. Burt. *Structural Holes: The Social Structure of Competition*, Harvard University Press, 1995. Cited on page 9.

[Carré, 1971] B. A. Carré. *An algebra for network routing problems*. IMA Journal of Applied Mathematics, 7 (3), pp. 273–294, 1971. arXiv:http://imamat.oxfordjournals.org/content/7/3/273.full.pdf+html, doi:10.1093/imamat/7.3.273. Cited on page 14.

[Chakrabarti and Faloutsos, 2006] D. Chakrabarti and C. Faloutsos. *Graph mining: Laws, generators, and algorithms*. ACM Computing Surveys, 38 (1), p. 2, 2006. Cited on page 2.

[Chung, 2007] F. Chung. *The heat kernel as the PageRank of a graph*. Proceedings of the National Academy of Sciences, 104 (50), pp. 19735–19740, 2007. doi:10.1073/pnas.0708838104. Cited on page 11.

[Cohen, 2009] J. Cohen. *Graph twiddling in a MapReduce world*. Computing in Science and Engineering, 11 (4), pp. 29–41, 2009. doi:10.1109/MCSE.2009.120. Cited on page 7.

[Cohen et al., 2009] J. Cohen, B. Dolan, M. Dunlap, J. M. Hellerstein, and C. Welton. *MAD skills: New analysis practices for Big Data*. Proc. VLDB Endow., 2 (2), pp. 1481–1492, 2009. doi:10.14778/1687553.1687576. Cited on page 19.

[Epasto et al., 2014] A. Epasto, J. Feldman, S. Lattanzi, S. Leonardi, and V. Mirrokni. *Reduce and aggregate: Similarity ranking in multi-categorical bipartite graphs*. In *Proceedings of the 23rd International Conference on World Wide Web*, pp. 349–360. 2014. doi:10.1145/2566486.2568025. Cited on page 1.

[Fiedler, 1973] M. Fiedler. *Algebraic connectivity of graphs*. Czechoslovak Mathematical Journal, 23 (98), pp. 298–305, 1973. Cited on pages 4 and 16.

[Flajolet et al., 2007] P. Flajolet, Éric Fusy, O. Gandouet, and F. Meunier. *HyperLogLog: the analysis of a near-optimal cardinality estimation algorithm*. In *Conference on Analysis of Algorithms*, pp. 127–146. 2007. Cited on page 18.

[Flajolet and Martin, 1985] P. Flajolet and G. N. Martin. *Probabilistic counting algorithms for data base applications*. J. Comput. Syst. Sci., 31 (2), pp. 182–209, 1985. doi:10.1016/0022-0000(85)90041-8. Cited on page 18.

[Fleury et al., 2015] E. Fleury, S. Lattanzi, and V. Mirrokni. *ASYMP: Fault-tolerant graph mining via asynchronous message passing*. 2015. Under submission. Cited on pages 1 and 7.

[Ghosh et al., 2014] R. Ghosh, S.-h. Teng, K. Lerman, and X. Yan. *The interplay between dynamics and networks: Centrality, communities, and cheeger inequality*. In *Proceedings of the 20th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pp. 1406–1415. 2014. doi:10.1145/2623330.2623738. Cited on pages 12 and 13.

[Gleich, 2014] D. F. Gleich. *PageRank beyond the web*. arXiv, cs.SI, p. 1407.5107, 2014. Cited on pages 4 and 15.

[Gleich and Mahoney, 2014] D. F. Gleich and M. M. Mahoney. *Algorithmic anti-differentiation: A case study with min-cuts, spectral, and flow*. In *Proceedings of the International Conference on Machine Learning (ICML)*, pp. 1018–1025. 2014. Cited on page 6.

[Gonzalez et al., 2012] J. E. Gonzalez, Y. Low, H. Gu, D. Bickson, and C. Guestrin. *Powergraph: Distributed graph-parallel computation on natural graphs*. In *Presented as part of the 10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12)*, pp. 17–30. 2012. Cited on pages 4, 7, 9, 11, 13, 15, and 18.

[Jeub et al., 2015] L. G. S. Jeub, P. Balachandran, M. A. Porter, P. J. Mucha, and M. W. Mahoney. *Think locally, act locally: Detection of small, medium-sized, and large communities in large networks*. Phys. Rev. E, 91, p. 012821, 2015. doi:10.1103/PhysRevE.91.012821. Cited on pages 13, 20, and 22.

[Jha et al., 2013] M. Jha, C. Seshadhri, and A. Pinar. *A space efficient streaming algorithm for triangle counting using the birthday paradox*. In *Proceedings of the 19th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pp. 589–597. 2013. doi:10.1145/2487575.2487678. Cited on page 25.

[Kang and Faloutsos, 2011] U. Kang and C. Faloutsos. *Beyond 'caveman communities': Hubs and spokes for graph compression and mining*. In *Proceedings of the 2011 IEEE 11th International Conference on Data Mining*, pp. 300–309. 2011. doi:10.1109/ICDM.2011.26. Cited on page 9.

[Kang et al., 2009] U. Kang, C. Tsourakakis, and C. Faloutsos. *Pegasus: A peta-scale graph mining system implementation and observations*. In *Data Mining, 2009. ICDM '09. Ninth IEEE International Conference on*, pp. 229 –238. 2009. doi:10.1109/ICDM.2009.14. Cited on pages 7, 9, 14, and 15.

[Katz, 1953] L. Katz. *A new status index derived from sociometric analysis*. Psychometrika, 18 (1), pp. 39–43, 1953. doi:10.1007/BF02289026. Cited on page 11.

[Kepner and Gilbert, 2011] J. Kepner and J. Gilbert. *Graph Algorithms in the Language of Linear Algebra*, SIAM, Philadelphia, 2011. Cited on pages 4, 9, 14, and 16.

[Klau, 2009] G. Klau. *A new graph-based method for pairwise global network alignment*. BMC Bioinformatics, 10 (Suppl 1), p. S59, 2009. doi:10.1186/1471-2105-10-S1-S59. Cited on page 21.

[Kloster and Gleich, 2014] K. Kloster and D. F. Gleich. *Heat kernel based community detection*. In *Proceedings of the 20th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pp. 1386–1395. 2014. doi:10.1145/2623330.2623706. Cited on page 13.

[Kondor and Lafferty, 2002] R. I. Kondor and J. D. Lafferty. *Diffusion kernels on graphs and other discrete input spaces*. In *Proceedings of the Nineteenth International Conference on Machine Learning*, pp. 315–322. 2002. Cited on pages 5, 11, and 25.

[Koschützki et al., 2005] D. Koschützki, K. A. Lehmann, L. Peeters, S. Richter, D. Tenfelde-Podehl, , and O. Zlotowski. *Centrality indicies*. In *Network Analysis: Methodological Foundations*, chapter 3, pp. 16–61. Springer, 2005. doi:10.1007/b106453. Cited on page 4.

[Koutra et al., 2011] D. Koutra, T.-Y. Ke, U. Kang, D. H. Chau, H.-K. K. Pao, and C. Faloutsos. *Unifying guilt-by-association approaches: Theorems and fast algorithms*. In *ECML/PKDD*, pp. 245–260. 2011. doi:10.1007/978-3-642-23783-6_16. Cited on pages 11 and 12.

[Krzywinski et al., 2012] M. Krzywinski, I. Birol, S. J. Jones, and M. A. Marra. *Hive plots–rational approach to visualizing networks*. Briefings in Bioinformatics, 13 (5), pp. 627–644, 2012. arXiv:http://bib.oxfordjournals.org/content/13/5/627.full.pdf+html, doi:10.1093/bib/bbr069. Cited on page 21.

[Lehoucq et al., 1997] R. B. Lehoucq, D. C. Sorensen, and C. Yang. *ARPACK User's Guide: Solution of Large Scale Eigenvalue Problems by Implicitly Restarted Arnoldi Methods*, 1997. Cited on pages 4 and 16.

[Leskovec et al., 2009] J. Leskovec, K. Lang, A. Dasgupta, and M. Mahoney. *Community structure in large networks: Natural cluster sizes and the absence of large well-defined clusters*. Internet Mathematics, 6 (1), pp. 29–123, 2009. Cited on pages 6, 7, 13, 20, and 22.

[Lin and Cohen, 2010] F. Lin and W. Cohen. *Power iteration clustering*. In *Proceedings of the 27th International Conference on Machine Learning*. 2010. Cited on page 11.

[Lin and Dyer, 2010] J. Lin and C. Dyer. *Data-Intensive Text Processing with MapReduce*, Morgan and Claypool, 2010. Cited on page 7.

[Lisewski and Lichtarge, 2010] A. M. Lisewski and O. Lichtarge. *Untangling complex networks: Risk minimization in financial markets through accessible spin glass ground states*. Physica A: Statistical Mechanics and its Applications, 389 (16), pp. 3250–3253, 2010. doi:10.1016/j.physa.2010.04.005. Cited on pages 11 and 12.

[Mahoney et al., 2012] M. W. Mahoney, L. Orecchia, and N. K. Vishnoi. *A local spectral method for graphs: With applications to improving graph partitions and exploring data graphs locally*. Journal of Machine Learning Research, 13, pp. 2339–2365, 2012. Cited on page 9.

[Malewicz et al., 2010] G. Malewicz, M. H. Austern, A. J. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski. *Pregel: A system for large-scale graph processing*. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data*, pp. 135–146. 2010. doi:10.1145/1807167.1807184. Cited on pages 4 and 15.

[Martin et al., 2012] R. L. Martin, B. Smit, and M. Haranczyk. *Addressing challenges of identifying geometrically diverse sets of crystalline porous materials*. Journal of Chemical Information and Modeling, 52 (2), pp. 308–318, 2012. arXiv:http://dx.doi.org/10.1021/ci200386x, doi:10.1021/ci200386x. Cited on page 1.

[Martin et al., 2011] S. Martin, W. M. Brown, R. Klavans, and K. W. Boyack. *Openord: an open-source toolbox for large graph layout*. Proc. SPIE, 7868, pp. 786806–786806–11, 2011. doi:10.1117/12.871402. Cited on page 20.

[Maschhoff and Sorensen, 1996] K. J. Maschhoff and D. C. Sorensen. *P_arpack: An efficient portable large scale eigenvalue package for distributed memory parallel architectures*. In *PARA '96: Proceedings of the Third International Workshop on Applied Parallel Computing, Industrial Computation and Optimization*, pp. 478–486. 1996. doi:10.1007/3-540-62095-8_51. Cited on page 16.

[McGregor, 2014] A. McGregor. *Graph stream algorithms: A survey*. SIGMOD Rec., 43 (1), pp. 9–20, 2014. doi:10.1145/2627692.2627694. Cited on page 25.

[Mihail, 1989] M. Mihail. *Conductance and convergence of markov chains-a combinatorial treatment of expanders*. In *Foundations of Computer Science, 1989., 30th Annual Symposium on*, pp. 526 –531. 1989. doi:10.1109/SFCS.1989.63529. Cited on pages 4 and 16.

[Morrison et al., 2005] J. L. Morrison, R. Breitling, D. J. Higham, and D. R. Gilbert. *GeneRank: using search engine technology for the analysis of microarray experiments*. BMC Bioinformatics, 6 (1), p. 233, 2005. doi:10.1186/1471-2105-6-233. Cited on page 12.

[Newman, 2006] M. E. J. Newman. *Finding community structure in networks using the eigenvectors of matrices*. Phys. Rev. E, 74 (3), p. 036104, 2006. doi:10.1103/PhysRevE.74.036104. Cited on pages 11 and 21.

[Newman and Girvan, 2004] M. E. J. Newman and M. Girvan. *Finding and evaluating community structure in networks*. Phys. Rev. E, 69 (2), p. 026113, 2004. doi:10.1103/PhysRevE.69.026113. Cited on page 22.

[Nguyen et al., 2013] D. Nguyen, A. Lenharth, and K. Pingali. *A lightweight infrastructure for graph analytics*. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, pp. 456–471. 2013. doi:10.1145/2517349.2522739. Cited on page 13.

[Page et al., 1999] L. Page, S. Brin, R. Motwani, and T. Winograd. *The PageRank citation ranking: Bringing order to the web*. Technical Report 1999-66, Stanford University, 1999. Cited on pages 4 and 11.

[Palmer et al., 2002] C. R. Palmer, P. B. Gibbons, and C. Faloutsos. *Anf: a fast and scalable tool for data mining in massive graphs*. In *KDD '02: Proceedings of the eighth ACM SIGKDD international conference on Knowledge discovery and data mining*, pp. 81–90. 2002. doi:10.1145/775047.775059. Cited on page 4.

[Pan et al., 2004] J.-Y. Pan, H.-J. Yang, C. Faloutsos, and P. Duygulu. *Automatic multimedia cross-modal correlation discovery*. In *KDD '04: Proceedings of the tenth ACM SIGKDD international conference on Knowledge discovery and data mining*, pp. 653–658. 2004. doi:10.1145/1014052.1014135. Cited on pages 4 and 13.

[Raghavan et al., 2007] U. N. Raghavan, R. Albert, and S. Kumara. *Near linear time algorithm to detect community structures in large-scale networks*. Phys. Rev. E, 76, p. 036106, 2007. doi:10.1103/PhysRevE.76.036106. Cited on page 18.

[Schaeffer, 2007] S. E. Schaeffer. *Graph clustering*. Computer Science Review, 1 (1), pp. 27–64, 2007. doi:10.1016/j.cosrev.2007.05.001. Cited on pages 13 and 22.

[Shun and Blelloch, 2013] J. Shun and G. E. Blelloch. *Ligra: A lightweight graph processing framework for shared memory*. SIGPLAN Not., 48 (8), pp. 135–146, 2013. doi:10.1145/2517327.2442530. Cited on pages 4, 6, 7, 11, 13, and 15.

[Spielman and Teng, 2008] D. A. Spielman and S.-H. Teng. *A local clustering algorithm for massive graphs and its application to nearly-linear time graph partitioning*. arXiv, cs.DS, p. 0809.3232, 2008. Cited on page 11.

[Stelzl et al., 2005] U. Stelzl, U. Worm, M. Lalowski, C. Haenig, F. H. Brembeck, H. Goehler, M. Stroedicke, M. Zenkner, A. Schoenherr, S. Koeppen, J. Timm, S. Mintzlaff, C. Abraham, N. Bock, S. Kietzmann, A. Goedde, E. Toksöz, A. Droege, S. Krobitsch, B. Korn, W. Birchmeier, H. Lehrach, and E. E. Wanker. *A human protein-protein interaction network: A resource for annotating the proteome*. Cell, 122 (6), pp. 957 – 968, 2005. doi:10.1016/j.cell.2005.08.029. Cited on page 1.

[Strohm and Homan, 2013] C. Strohm and T. R. Homan. *NSA spying row in congress ushers in debate over Big Data*. Bloomberg, Online, pp. July–25, 2013. Cited on page 1.

[Sui et al., 2013] X. Sui, T.-H. Lee, J. Whang, B. Savas, S. Jain, K. Pingali, and I. Dhillon. *Parallel clustered low-rank approximation of graphs and its application to link prediction*. In *Languages and Compilers for Parallel Computing*, pp. 76–95. Springer Berlin Heidelberg, 2013. doi:10.1007/978-3-642-37658-0_6. Cited on page 26.

[Suri and Vassilvitskii, 2011] S. Suri and S. Vassilvitskii. *Counting triangles and the curse of the last reducer*. In *Proceedings of the 20th International Conference on World Wide Web*, pp. 607–614. 2011. doi:10.1145/1963405.1963491. Cited on page 11.

[Troyer, 1968] S. R. Troyer. *Sparse matrix multiplication*. Technical Report ILLIAC IV Document Number 191, University of Illinois, Urbana-Champagne, 1968. Cited on page 14.

[Tsourakakis, 2008] C. Tsourakakis. *Fast counting of triangles in large real networks without counting: Algorithms and laws*. In *Proceedings of the Eighth IEEE International Conference on Data Mining, 2008.*, pp. 608–617. 2008. doi:10.1109/ICDM.2008.72. Cited on page 16.

[Ugander and Backstrom, 2013] J. Ugander and L. Backstrom. *Balanced label propagation for partitioning massive graphs*. In *Proceedings of the 6th ACM international conference on Web search and data mining*, pp. 507–516. 2013. Cited on pages 1, 18, and 20.

[Vigna, 2009] S. Vigna. *Spectral ranking*. arXiv, cs.IR, p. 0912.0238, 2009. Cited on page 12.

[Whang et al., 2013] J. J. Whang, D. F. Gleich, and I. S. Dhillon. *Overlapping community detection using seed set expansion*. In *Proceedings of the 22nd ACM international conference on Conference on information and knowledge management*, pp. 2099–2108. 2013. doi:10.1145/2505515.2505535. Cited on pages 13 and 20.

[Xie et al., 2013] J. Xie, S. Kelley, and B. K. Szymanski. *Overlapping community detection in networks: The state-of-the-art and comparative study*. ACM Comput. Surv., 45 (4), pp. 43:1–43:35, 2013. doi:10.1145/2501654.2501657. Cited on page 13.

[Zhang and Moore, 2014] P. Zhang and C. Moore. *Scalable detection of statistically significant communities and hierarchies, using message passing for modularity*. Proceedings of the National Academy of Sciences, 111 (51), pp. 18144–18149, 2014. arXiv:http://www.pnas.org/content/111/51/18144.full.pdf+html, doi:10.1073/pnas.1409770111. Cited on page 14.

[Zhou et al., 2003] D. Zhou, O. Bousquet, T. N. Lal, J. Weston, and B. Schölkopf. *Learning with local and global consistency*. In *NIPS*. 2003. Cited on page 13.

[Zimmer, 2011] C. Zimmer. *100 trillion connections: New efforts probe and map the brain's detailed architecture*. Scientific American, 304, pp. 58–63, 2011. doi:10.1038/scientificamerican0111-58. Cited on page 1.