Handbook of Big Data

Bühlmann Peter, Drineas Petros, Kane Michael, van der Laan Mark

Mining Large Graphs

Publication details
David F. Gleich, Michael W. Mahoney
Published online on: 18 Feb 2016

How to cite: - David F. Gleich, Michael W. Mahoney. 18 Feb 2016, Mining Large Graphs from: Handbook of Big Data Routledge.
Accessed on: 10 Oct 2018
12

Mining Large Graphs

David F. Gleich and Michael W. Mahoney

CONTENTS

12.1 Introduction ........................................... 192
12.1.1 Scope and Overview ............................ 192
12.2 Preliminaries ........................................... 193
12.2.1 Graph Representations .......................... 193
12.2.2 Graph Mining Tasks .............................. 194
12.2.3 Classification of Large Graphs ................. 195
12.2.4 Large Graph Mining Systems .................. 197
12.2.5 Sources for Data .................................. 197
12.3 Canonical Types of Large-Scale Graph Mining Methods .... 198
12.3.1 Geodesic Neighborhood-Based Graph Mining .. 198
12.3.2 Diffusion Neighborhood-Based Graph Mining .. 199
12.3.3 Generalized Matrix-Vector Products Graph Mining .... 199
12.4 Mining with Geodesic Neighborhoods .............. 199
12.4.1 Single Neighborhoods ............................ 199
12.4.2 All Neighborhoods ................................ 200
12.4.3 Complexities with Power-Law Graphs .......... 200
12.5 Mining with Diffusion Neighborhoods ............. 201
12.5.1 ACL Push Methods ............................... 203
12.5.2 Random Walks with Restart ..................... 203
12.6 Mining with Generalized Matrix-Vector Products ...... 204
12.6.1 Algorithms with Standard Matrix-Vector Products .... 205
12.6.2 Algorithms with Semi-Ring Matrix-Vector Products .... 205
12.6.3 General Updates .................................. 207
12.6.4 Edge-Oriented Updates ............................ 207
12.6.5 Complexities with Power-Law Graphs .......... 208
12.6.6 Implementations in SQL ......................... 208
12.7 Limitations and Tradeoffs in Mining Large Graph ...... 208
12.7.1 Graph Drawing ...................................... 209
12.7.2 Viral Propagation .................................. 210
12.7.3 Communities and Modularity .................... 211
12.7.3.1 Preliminaries .................................. 211
12.7.3.2 Modularity Definition ......................... 212
12.7.3.3 Modularity as a Cut Measure ................. 212
12.8 Large Graph Mining in the Future .................. 214
12.8.1 Mining with Streaming Procedures .............. 214
12.8.2 Mining with Generalized Matrix-Matrix Products .... 214
12.1 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. 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 through 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 one trillion URLs on the internet, corresponding to a graph with over one trillion nodes [1]; in 2012, the Facebook friendship network spanned 721 million individuals and had 137 billion links [2]; phone companies process a few trillion calls a year [3]; the human brain has around 100 billion neurons and 100 trillion neuronal connections [4]; 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 space [5]; and one of the largest reported experiments with a real-world graph involved over 1.5 trillion edges [6].

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 [7,8] 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 [2]. Alternatively, large graph mining produces similar information for recommendation, suggestion, and prediction from messy data [9,10]; it can also tell us how to optimize a data infrastructure to improve response time [11]; and it can tell us when and how our data are anomalous [12].

12.1.1 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, for example, [13,14]. An important theme in our chapter is that large graphs are often very different than small graphs and thus intuitions from small graphs

*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 pair of nodes. Of course, in many applications one is interested in graphs that have weighted or directed edges, are time-varying, have additional meta-information associated with the nodes or edges, and so on.
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. At several points, we will also highlight the relationships between seemingly distinct methods and unexpected, often implicit, properties of large-scale graph algorithms.

12.2 Preliminaries

When data is 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 dataset 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 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.

12.2.1 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.

*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 toward accurate intuition, and in some cases, we provide additional technical caveats for the experts in the footnotes.

†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.
While for tiny graphs, for example, graphs containing fewer than several thousand nodes, one can take a *bird’s eye* view and think about the entire graph since, for example, 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. We need to discuss two important representations of graphs 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 this set of neighbors in a time that we will consider constant.* Adjacency lists are common when graphs are an explicit component of the original data model.

The adjacency list is the most flexible format because it 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.†

### 12.2.2 Graph Mining Tasks

We will 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.‡ Running millions of random walk steps given in an adjacency list representation of a large graph is easy,§ and these steps can be used to extract a small region of a massive graph nearby that seed [16,17].

**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 [18] that we will discuss in Section 12.6.

**PageRank.** PageRank [16] is one of a host of graph centrality measures [19] that give information to address the question “What are the most important nodes in my graph?” See [20] 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., [21–23]).

* Various implementations and systems we consider may not truly guarantee constant time access to the neighborhood set of vertices, for example, 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.

† This data movement is a great fit for Google’s MapReduce system.

‡ 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.

§ Current research efforts are devoted to running random walks with restarts for millions of seeds concurrently on edge list representations of massive graphs [15].
Effective diameter. The effective diameter of a graph is the length of the longest path necessary to connect 90% of the possible node pairs. 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 [24,25].

Extremal eigenvalues. There are a variety of matrix structures associated with a graph. One of the most common is the adjacency matrix, denoted \(A\), where \(A_{i,j} = 1\) if there is an edge between vertices \(i\) and \(j\) and \(A_{i,j} = 0\) otherwise. Another common matrix is the normalized Laplacian, denoted \(L\), where \(L_{i,i} = 1\) and \(L_{i,j} = 1/\sqrt{\text{degree}(i) \cdot \text{degree}(j)}\) if there is an edge between \(i\) and \(j\), and \(L_{i,j} = 0\) 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 [26,27]. The best algorithms for these problems use the ARPACK software [28], which includes sophisticated techniques to lock eigenvalues and vectors after they have converged. This method would require something like \(O(nk \log n)\) time and memory to compute reasonable estimates of the extremal \(k\) eigenvalues and eigenvectors.

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, and graph kernels [29]) 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, for example, the \(\delta\)-hyperbolicity properties of a graph with an \(\Theta(n^4)\) algorithm [30]; however, these are many of the most representative problems/algorithms in which graph miners are interested. See Table 12.1 for a brief summary.

### 12.2.3 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 12.2.2

---

*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.*

†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.

‡This bound is not at all precise, but a fully precise bound is not a useful guide to practice; this statement represents a working intuition for how long it takes compared to other ideas.
TABLE 12.1
Several common graph primitives and their time and memory complexity.

<table>
<thead>
<tr>
<th>Primitive</th>
<th>Time Complexity</th>
<th>Memory Complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td>Random walk with restart</td>
<td>$O(1)$ time and $O(1)$ memory</td>
<td></td>
</tr>
<tr>
<td>Connected components</td>
<td>$O(n)$ time and memory</td>
<td></td>
</tr>
<tr>
<td>PageRank</td>
<td>$O(n)$ time and memory</td>
<td></td>
</tr>
<tr>
<td>Extremal eigenvalues</td>
<td>$O(n \log n)$ time and memory</td>
<td></td>
</tr>
<tr>
<td>Triangle counting</td>
<td>$O(n\sqrt{n})$ time and memory</td>
<td></td>
</tr>
<tr>
<td>All-pairs shortest paths</td>
<td>$O(n^3)$ time and $O(n^2)$ memory</td>
<td></td>
</tr>
</tbody>
</table>

...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, for example, 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.

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.\(^\dagger\) 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: (1) the most important properties of graphs in this regime (and larger) are very different than the properties of small graphs [33], and (2) 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 five million vertices, algorithms that do $O(n^2)$ computations 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 32 Gb 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.

\(^*\)That being said, this does not mean that the na"ive cubic algorithm is best, for example, faster algorithms that have been developed for much larger graphs can implicitly regularize against noise in the graph [31,32]. Thus, they might be better even for rather small graphs, even when more expensive computations are possible.

\(^\dagger\)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.
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.* This was the motivation of the Ligra project [23]. Also, the effective diameter computations on the Facebook networks with 70B edges were done on a single shared memory machine [2].

**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 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.† The types of problems that we can expect to solve on extremely LARGE graphs are only very simple things such as triangles, connected components, PageRank, and label propagation [5,6].

### 12.2.4 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 as follows:

**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 [34,35] or adjacency access [36]; GraphLab [22] is a distributed, online, adjacency system; and Ligra [23] is an online, adjacency list, centralized system.

### 12.2.5 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**

[https://snap.stanford.edu/data/index.html](https://snap.stanford.edu/data/index.html)

---

*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.

†This is likely the best strategy for most of those who are interested in non trivial analytics on LARGE graphs.
This website has a variety of social network data up to a few billion edges.\footnote{Detailed empirical results for these and many other informatics graphs have been reported previously [33].} There is also a host of metadata associated with the networks there, including some ground-truth data for real-world communities in large networks.

**Laboratory for Web Algorithmics**
http://law.di.unimi.it/datasets.php
The Laboratory for Web Algorithmics group at the University of Milano maintains datasets 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://webdatacommons.org/hyperlinkgraph/
Both the ClueWeb group and Common Crawl groups maintain 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.\footnote{Web graphs require special treatment in terms of number of nodes due to the presence of a crawling frontier.} 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 datasets (Common Crawl).

**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 datasets from the recent DIMACS challenge on graph partitioning [37] are all contained in this repository. Many of these datasets come from much more structured scientific computing applications.

## 12.3 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 methods 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.

### 12.3.1 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, for example, anomaly detection [12]. These methods are typically very easy to scale to large graphs when working simultaneously with all of the vertices. 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.

12.3.2 Diffusion Neighborhood-Based Graph Mining

Diffusion neighborhood-based computations can be thought of as a softer or fuzzy version of geodesic neighborhood-based computations. 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 easy to scale to massive graphs because they do not need to explore the entire graph. For example, random walks with restart are an instance of this idea.† One should think of running these diffusion neighborhoods on only \( O(\log n) \) or \( O(\sqrt{n}) \) of the nodes instead of on all nodes.

12.3.3 Generalized Matrix-Vector Products Graph Mining

The bulk of our chapter will focus 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 four degrees of separation between individuals [2]. Thus, effective diameter, extremal eigenvalues, PageRank, connected components, and host of other ideas [38] 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 [18,35]. There is a problem with high-degree nodes with large neighborhoods (think of Barack Obama on 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 [22,39].

12.4 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.

12.4.1 Single Neighborhoods

Perhaps the simplest large graph mining task involves the one-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

---

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

†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.
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 [40].

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.

12.4.2 All Neighborhoods

A richer class of graph mining tasks involve using all of the one-step geodesic neighborhoods, that is, the one-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 one-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 one-step neighborhoods takes work that scales as $O(n\sqrt{n})$ for sparse graphs—see the discussion in Section 12.2. As graphs become LARGE, even this level of computation is not feasible. Streaming computations are an active research area that provides an alternative (see Section 12.8.1).

Example: Oddball anomaly detection. One example of how neighborhood mining works in practice is given by the so-called Oddball anomaly detection method [12]. 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 four features associated with each vertex. Oddball then applies an outlier detection method to this four-dimensional dataset, and it is able to distinguish a variety of types of anomalous nodes on the Twitter network [12].

12.4.3 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

*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 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 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 [41]. 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 [23] or vertex and edge splitting frameworks [22].

12.5 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 [16]
- The Katz diffusion [42]
- The heat kernel diffusion [29, 43]
- The truncated random walk diffusion [44]

These diffusions, and many minor variations, are frequently invented and reinvented under a host of different names [45]: in biology networks, for instance, a PageRank diffusion may also be called an information diffusion [46]; spectral methods and local spectral methods implement variants of this idea [31]; diffusion-based information is also known as guilt by association [47]; 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. We illustrate the difference on a simple example in Figure 12.1. These diffusion neighborhoods are commonly used for large graph mining because they can be computed extremely quickly. For instance, it is 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 $7500 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.† Suppose that source of a diffusion is only a single node $u$ in the graph.

---

*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.
†The majority of this section applies to all types of diffusion neighborhoods using adaptations of the ideas to the nature of those diffusions [49–52].
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 $u$.
2. Along each edge, only $\beta$ 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 12.1, 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 (ACL) push procedure (Section 12.5.1).
2. The random walks with restart method (Section 12.5.2).

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 [46,47,53].

**FIGURE 12.1**

An illustration (a) of a geodesic neighborhood in Newman’s netscience graph [48] around the circled node, with neighbors colored yellow. Diffusion neighborhoods: (b) small diffusion neighborhood and (c) larger diffusion neighborhood 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 diffusion neighborhood does not extend beyond the natural borders in the graph.
**Example:** Semi-supervised learning. Semi-supervised learning is closely related to 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 [54]. 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. Diffusion neighborhoods are an important component of many community detection methods [31,33]. 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 sweep cut. Repeating this process for multiple nodes can yield high-quality overlapping communities [58] on small large (as well as small and large) graphs.

### 12.5.1 ACL Push Methods

The ACL push method is a scalable method to propagate, or evaluate, a diffusion, given a seed node or set of seed nodes [31]. 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.

**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 [22,23,60].

### 12.5.2 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 [17,61,62]. Tracking where the random walk moves in the graph provides most of the information on the diffusion, and a few thousand random walks provide acceptable accuracy in many cases [61,62].

**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

---

*It would require another chapter to discuss communities in large networks in appropriate depth [33, 55,56].
†In particular, spectral algorithms that are commonly used to detect communities have strong connections with diffusion-based neighborhood methods [57].
‡Variants and extensions exist for a host of other diffusions [49,50,59].
§This is done with a sweep cut procedure and uses a localized instance of a Cheeger inequality.
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 10,000), and recent research gives a few strategies to compute these random walks simultaneously [15].

### 12.6 Mining with Generalized Matrix-Vector Products

Generalized matrix-vector products are one of the most flexible and scalable ways to mine large graphs [18,35]. 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 [63], and their use with generalized matrix-vector products was recognized early [64]. All of these methods and algorithms apply to batch systems rather than online systems.*

A matrix-vector product \( y = Ax \) with the adjacency matrix of a graph expresses the computational primitive:

\[
y_v = \sum_{u \in N(v)} x_u, \tag{12.1}
\]

where \( N(v) \) is the neighbor set of node \( v \). If \( A \) represents a directed graph, then matrix-vector products with \( A \) sum over the out-neighbors of \( v \), whereas matrix-vector products with \( A^T \) 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 \( (12.2) \)

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 will see shortly. Operations such as minimum spanning trees [18], maximum weight matchings [65], and message passing methods [66] fit into the same framework as well.

**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 [21]. Both GraphLab and Ligra adopt this same type of vertex and edge programming [22,23]. 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 [35], as does Combinatorial BLAS [67].

**Two types of functions.** We will consider two classes of function \( f \). The first class is a reduction operation, which is a 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 will 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 [22], only work for reduction functions.

---

* 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.
12.6.1 Algorithms with Standard Matrix-Vector Products

Even the standard matrix-vector product is a 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:

Initialize: \( x_v^{(\text{start})} = 1 \),
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 \( \alpha \) is taken to be 0.85,\(^*\) and 20 or 30 iterations suffice for most purposes for this value of \( \alpha \).

Example: Extremal eigenvalues and approximate number of triangles. The extremal eigenvalues and eigenvectors of the adjacency matrix and normalized Laplacian matrix can be provided by the ARPACK software \([28]\), and its parallel PARPACK variant \([68]\), 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 \([69]\). The extremal eigenvectors of the normalized Laplacian matrix indicate good ways to split the graph into pieces \([26,27]\).

12.6.2 Algorithms with Semi-Ring Matrix-Vector Products

Our first generalization of matrix-vector products involves changing what addition and multiplication by using a semi-ring.\(^\dagger\)\(^\ddagger\) We use \( \oplus \) and \( \otimes \) to denote the changed addition and multiplication operations to distinguish them from the usual operations. A classic example of this case 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 \( \oplus \) respectively.

\(^*\)Note, though, that \( \alpha \) really is just a regularization parameter, and so its value should be chosen according to a model selection rule. See \([20]\) for a discussion on values of \( \alpha \).

\(^\dagger\)More formally, a semi-ring is a set that is closed under two binary operations: \( \otimes \) and \( \oplus \) along with their respective identity elements: \( 1 \), the multiplicative identity element and \( 0 \), the additive identity element. These operations must be associative and distributive.

\(^\ddagger\)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, that is, as matrix-vector products with non-standard semi-ring matrix-vector multiplication, that is, is much more powerful than considering just the usual matrix-vector product.
multiplying by 1 do not change the answer. The identity elements in min-plus are $\square = \infty$ and $\circ = 0$. 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 [18]. Note that for a general matrix and vector $A$ and $x$ the matrix-vector $y = Ax$ 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:

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

**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. Recall 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 $u$, $A_{v,u} = \square$ if they are not connected and $A_{v,v} = \circ$ otherwise. Consider the iteration:

**Initialize:** $x_{v}^{(\text{start})} = \begin{cases} \circ & v = s \\ \square & v \neq s \end{cases}$

**Iterate:**

$$x_{v}^{(\text{next})} = A_{v,1} \odot x_{1}^{(\text{cur})} \oplus A_{v,2} \odot x_{2}^{(\text{cur})} \oplus \cdots \oplus A_{v,n} \odot x_{n}^{(\text{cur})}$$

$$x_{v}^{(\text{next})} = \min_{u \in N(v) \cup \{v\}} \left[ A_{v,u} + x_{u}^{(\text{cur})} \right].$$

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 $a \odot b = a \times b$ (the regular multiplication operation). Here, $\circ = 1$ and $\square = \infty$. Let $A_{v,u} = \circ$ if $v$ and $u$ have an edge, $A_{v,u} = \square$ if they are not connected, and $A_{v,v} = \circ$. Using this semi-ring, we can compute the connected components of a graph:

**Initialize:** $x_{v}^{(\text{start})} = \text{unique id for } v$

**Iterate:**

$$x_{v}^{(\text{next})} = A_{v,1} \odot x_{1}^{(\text{cur})} \oplus A_{v,2} \odot x_{2}^{(\text{cur})} \oplus \cdots \oplus A_{v,n} \odot x_{n}^{(\text{cur})}$$

$$x_{v}^{(\text{next})} = \min_{u \in N(v) \cup \{v\}} \left[ x_{u}^{(\text{cur})} \right].$$

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.

*This can also be used to compute a breadth-first search as well.*
12.6.3 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 will 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.* 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 Equation 12.2 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, that is, 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.

**Example: Label propagation for clusters and communities.** Label propagation is a method to divide a graph into small clusters or communities [70] that has been used to optimize distributions of graph vertices to processors [11] and to optimize the ordering of vertices in adjacency matrices [71]. 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 by $k$ 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 [72] and the HyperLogLog counter [73] 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.†

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.

12.6.4 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.

---

*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.
†The specifics of the algorithm are not our focus here, see [25] for a modern description.
12.6.5 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 [22] 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.

12.6.6 Implementations in SQL

Generalized matrix-vector products are possible to implement even in traditional database systems that implement SQL [74]. 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. 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).

\[
\text{edges: id | head | tail}
\]

\[
\text{x: id | comp}
\]

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

\[
\text{CREATE TABLE } x_{\text{next}} \text{ AS (}
\]

\[
\text{SELECT e.tail AS id, MIN(x.comp) AS comp}
\]

\[
\text{FROM edges e INNER JOIN x ON e.head = x.id}
\]

\[
\text{GROUP BY e.tail );}
\]

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 them by the tail of each edge and take the \( \text{MIN} \) function over all components. This is exactly what the iteration in the connected components example did.

12.7 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 have focused on in the previous sections, for example, 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; it can lead to qualitatively incorrect understanding of the behavior of algorithms that run on larger graphs.

\*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.
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.* A small subgraph of size roughly 100 vertices is a global structure in a graph with only 1000 vertices; however, it is a local structure in a graph with millions of vertices. As typical graphs get larger and larger, much of the local structure does not change or grow [33,57]; 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 [33] and as elaborated upon in [57]. 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 of 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 (for example, [11,58]); 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; 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.

### 12.7.1 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 [75,76]. While graph layout algorithms are often able to find interesting and useful structures in graphs with around 1000 vertices, they almost universally fail at finding any useful or interesting structure in graphs with more than 10,000 vertices.† 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 [33,57]. 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 graph structure. This is essentially how the oddball anomaly detection method works [12]. Each vertex is summarized with a few small local features. The result is a set of

---

*Informally, by local structure, we mean, for example, 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.

†These failures can be quite beautiful, though, from an artistic point of view.
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 [78].

12.7.2 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 12.2. 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.* The results of this simple experiment are in Figure 12.3, 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. 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 [57, Figures 12 and 13].

![FIGURE 12.2](image)

(a) a 379 node graph
(b) a 8887 node graph

Graph drawing for a small and large graph. 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 [48]). 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 [77]). 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.

*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.
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.

12.7.3 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 [33,55,57]. Another popular community quality metric is known as modularity [79]. 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.

12.7.3.1 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:

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

where $d_i$ is the degree of node $i$. We follow the convention $\text{vol}(G) = \text{vol}(V)$ to denote the total volume of the graph. The $\text{edges}$ function counts the number of edges between subsets of vertices and counts both edges:
edges(S, T) = \sum_{i \in S, j \in T} A_{i,j} \quad \text{and} \quad \text{edges}(S) = \text{edges}(S, S).

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

\[
\text{cut}(S) = \text{edges}(S, \bar{S}) = \text{cut}(\bar{S}).
\] (12.3)

The function \( \text{cut}(S) \) 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:

\[
\begin{align*}
\text{edges}(S) &= \text{vol}(S) - \text{edges}(S, \bar{S}) = \text{vol}(S) - \text{cut}(S), \\
\text{vol}(S) &= \text{vol}(G) - \text{vol}(\bar{S}).
\end{align*}
\]

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

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

12.7.3.2 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 nonrandom. 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:

\[
Q(S) = \frac{1}{\text{vol}(G)} \left( \text{edges}(S) - \frac{1}{\text{vol}(G)} \text{vol}(S)^2 \right).
\] (12.4)

The term \( (1/\text{vol}(G)) \text{vol}(S)^2 \) 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:

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

12.7.3.3 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 Equation 12.3,\(^*\) and we will describe the implications of this reformulation for finding good communities in small versus large graphs.

\(^*\)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 \( \nu = 1/\text{vol}(G) \). Note that then:

\[
Q(S) = \nu (\text{vol}(S) - \text{cut}(S) - \nu \text{vol}(S) (\text{vol}(G) - \text{vol}(\bar{S}))) = \nu (\text{vol}(S) - \text{cut}(S)).
\]

From this, we have that \( Q(S) = Q(\bar{S}) \) because \( \text{cut}(S) = \text{cut}(\bar{S}) \). Consider the modularity of a two-way partition and observe\(^\ast\):

\[
Q(P_2) = \frac{1}{2} [Q(S) + Q(\bar{S}) + Q(S) + Q(\bar{S})] = \frac{\nu}{2} (\text{edges}(S) - \nu \text{vol}(S)^2 + \text{edges}(\bar{S}) - \nu \text{vol}(\bar{S})^2 + 2\nu \text{vol}(S) \text{vol}(\bar{S}) - 2 \text{cut}(S)) = \frac{\nu}{2} (\text{vol}(S) + \text{vol}(\bar{S}) - 4 \text{cut}(S) + \nu (\text{vol}(S) - \text{vol}(\bar{S}))^2).
\]

Hence,

\[
Q(S) = \frac{1}{4} - \frac{\nu}{4} (4 \text{cut}(S) + \nu (\text{vol}(S) - \text{vol}(\bar{S}))^2).
\]

From this formulation of the objective we conclude the following theorem.

**Theorem 12.1** The best two way modularity partition corresponds to finding a subset \( S \) that minimizes \( \text{cut}(S) + \nu/4 (\text{vol}(S) - \text{vol}(\bar{S}))^2 \).

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 \( \text{vol}(S) = \text{vol}(\bar{S}) \) is extremely strong, however, and thus there is a very strong bias toward finding very well-balanced clusters. This is true whether or not those clusters satisfy the intuitive bicriteria that communities should be set 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:

\[
Q(P) = \sum_{S \in P} Q(S) = |P|/4 - \frac{\nu}{4} \sum_{S \in P} [4 \text{cut}(S) + \nu (\text{vol}(S) - \text{vol}(\bar{S}))^2],
\]

where \(|P|\) is the number of partitions. An equivalent formulation helps to make the magnitude of the terms more clear:

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

In other words, when considering a multi-way partitioning problem with the modularity objective, adding a new community yields a bonus of \( \text{vol}(G)/4 \), 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.

\[^\ast\text{This result is derived in a slightly convoluted way, but we have yet to devise a more concise proof.}\]
Thus, in small graphs, such as those on the left of Figures 12.2 and 12.3 that lead to a nice visualization, Equation 12.4 empirically captures 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.

12.8 Large Graph Mining in the Future

Large graph mining is a growing field, and there are many excellent ideas we cannot discuss 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.

12.8.1 Mining with Streaming Procedures

Streaming procedures for graphs take input as 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 of 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.* For instance, it is possible to compute accurate estimates of graph motif counts on graph streams [80]. 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 [81]. 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.

12.8.2 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 distances 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 [29]. 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 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 yield near-constant time queries. Finding

---

*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.
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 \([82]\). These products were approximated in order to make the computation efficient.

---

**Acknowledgments**

DGF would like to acknowledge the NSF through award CCF-1149756 for providing partial support. MWM would like to acknowledge the Army Research Office, the Defense Advanced Research Projects Agency XDADA and GRAPHS programs, and the Department of Energy for providing partial support for this work.

---

**References**


10. Alessandro Epasto, Jon Feldman, Silvio Lattanzi, Stefano Leonardi, and Vahab Mirrokni. Reduce and aggregate: Similarity ranking in multi-categorical bipartite


