CHARACTERIZING ACCURACY AND PERFORMANCE TRADEOFFS IN
GRAPH SAMPLING FOR GRAPH PROPERTY COMPUTATIONS

BY

TEJ CHAJED

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Bachelor of Science in Electrical and Computer Engineering
in the College of Engineering of the
University of Illinois at Urbana-Champaign, 2014

Urbana, Illinois

Adviser:

Prof. Indranil Gupta
ABSTRACT

In this thesis, we present a systematic way to characterize the tradeoffs between accuracy and cost in graph sampling. This characterization is heavily dependent on graph structure. Here we focus on vector graph properties, which consist of a value per node in the graph (e.g., PageRank, degree).

We present a new technique for assessing the accuracy of a property based on the algorithm used to compute it. Next, we describe how to interpret several features of accuracy-performance tradeoff curves. Finally, we present our analysis of actual accuracy-cost curves for both real-world and synthetic graphs. Conclusions from the analysis include that the structure of a graph is more important than its scale for the purposes of sampling, and that different structures require different sampling approaches.

Keywords: graph analytics; graph sampling; measurement
To my parents and family, for their unconditional love and never-ending support.
# TABLE OF CONTENTS

CHAPTER 1  INTRODUCTION ...........................................  1

CHAPTER 2  BACKGROUND ............................................  4
  2.1 Graph processing frameworks ..................................  4
  2.2 Computation models ...........................................  6
  2.3 Graph approximation ...........................................  7
  2.4 Previous graph sampling studies ...............................  9

CHAPTER 3  GRAPH SAMPLING ....................................  10
  3.1 Algorithms ..................................................  10
  3.2 Evaluation metrics .........................................  11

CHAPTER 4  EVALUATION ..........................................  14
  4.1 Evaluation methodology .....................................  15
  4.2 Real-world graphs .........................................  20
  4.3 Random graph models .....................................  25
  4.4 Characteristics of Tradeoff Curves ..........................  31

CHAPTER 5  CONCLUSION .........................................  34

REFERENCES .........................................................  36
CHAPTER 1

INTRODUCTION

One emerging area in analytics is graph processing. With today’s computational power, organizations and researchers are increasingly capable and interested in studying structured data that can be viewed as a network or graph consisting of vertices and edges connecting them. Examples of networks that people are interested in studying include the Internet, viewed as a set of routers and machines connected by physical links; the electrical power grid, as a set of power plants connected by transmission and distribution lines; and the friendship graph of social networks consisting of users linked by friendships. Graph analytics aims to answer questions about the structural characteristics of these graphs, such as the resilience of the network to failures of edges in the case of the Internet and power grid [1, 2] and predicting the rest of a user’s connections on a social network such as Facebook, LinkedIn and Twitter [3, 4]. Techniques of network analysis have been applied to areas as diverse as analyzing networks of drug misuse, detecting communities in networks, studying the size of the Internet graph, text summarization in natural language processing, and understanding the brain based on neural connectivity [5, 6, 7, 8, 9].

Graph processing increasingly tackles large-scale graphs that are too large to fit on a single machine. Several systems have been built to address graph processing, especially distributed frameworks, such as GraphLab [10] and Apache Giraph. However, sometimes the scale of a graph processing task exceeds the capacity of even a cluster, such as when the graph of interest is the nodes that make up the neurons of the brain or the pages of the World Wide Web connected by hyperlinks. Furthermore, distributed graph processing frameworks incur high communication overhead and some frameworks also have a memory cost associated with more nodes, making running on fewer nodes attractive for performance. In other cases query latency is very important and the user is willing to trade some accuracy in the answer in return
for substantial speed increases.

When faced with a large quantity of data, one approach is to sample the data to retain its essential properties while approximating the details of the results. This creates a tradeoff between accuracy and cost. This tradeoff in the context of graph sampling is the focus of this thesis. Sampling is quite natural for data that takes the form of records. If the analytics goal is aggregation queries, then the mean and variance of the data can assist in providing robust estimates when there is enough data. Statistical results such as the law of large numbers and the central limit theorem give results for conditions under which these estimates are provably accurate (in the limit as the size of the data approaches infinity). There are systems built on exactly this idea for approximating aggregation queries for record data [11, 12].

For graphs there are a number of challenges to sampling. First, analytic approaches to sampling are difficult. Coming up with unbiased estimators for graph properties (that is, transforming a measurement on a sampled graph into one for the whole graph so that it is correct on average) is a difficult task that must be carried out analytically for each desired property [13]. Second, some of the little analytical work that has been carried out has found that naïve graph sampling can bias the result heavily; this thesis is particularly targeted at networks such as the Internet that are so large that our only data on them is a sample [14, 15].

Broadly, the goal of sampling is to select a smaller graph that is somehow representative of the larger graph while reducing cost. Ideally analytics on the smaller graph can be transformed into results about the larger graph. Whereas this is natural for data records, for graphs the problems are non-trivial. First arises the question of how to select a sample of the vertices and edges together; we answer this question by describing some existing algorithms in Section 3.1 on page 10. Next, what constitutes a representative sample? We posit that the answer might depend on both the desired metric and the graph structure itself. Finally, we admit the possibility that a graph may not be reasonably sampled in a computationally efficient manner. All of these questions concern the tradeoff presented by sampling between accuracy of the final result and the computational cost. From experimental data we find that some synthetic and real-world graphs exhibit favorable sampling properties, while some other real-world graphs are difficult to sample. One observation from our data is that size is less of a determining factor than
structure. In studying structure, we find that several common random graph models result in graphs that are easy to sample.

The main contributions of this thesis are as follows:

• We describe a new method of assessing the accuracy of a sample based on the algorithm used to compute it (Chapter 3).

• We describe how to interpret several features of accuracy-cost tradeoff curves, using numerical measures of accuracy and cost (Chapter 4).

• We present our analysis of actual accuracy-cost tradeoff curves on both real-world and synthetic graphs (Chapter 4).
In this chapter, we give an overview of the prior work related to graph sampling. We begin with the frameworks that aim to make graph analysis possible and efficient. Next, we discuss more general approaches to approximation in graph analytics that have attempted to trade off accuracy for cost (in some cases to make analysis possible). Finally, we mention some specific work studying graph sampling itself.

2.1 Graph processing frameworks

While the techniques of this thesis are generally independent of the graph processing framework under consideration, some aspects of graph processing frameworks are relevant and we did implement our system in GraphLab in particular. A common model for graph processing frameworks, especially earlier systems inspired by Pregel [16], is the Bulk Synchronous Parallel (BSP) model of computation [17]. Pregel has users write vertex programs, which perform some small computation on a vertex (which might have an associated value, such as a distance for single-source shortest path or a PageRank). In each iteration, a vertex program can read its neighbors’ values, update its own value, and send messages of arbitrary structure to its neighbors. The framework implements a barrier between iterations so that communication only occurs over iterations.

There are many graph processing frameworks that make various trade-offs between ease of programming, expressibility, efficiency and scalability. Pregel is well known as one of the most popular abstractions for graph processing, but details of the implementation are scarce and proprietary. One open-source implementation from Apache is Giraph [18]. Giraph makes the core Pregel abstraction available to the rest of the world while adding some
additional features that make it easier to perform necessary data processing
tasks that aren’t related to the core graph algorithm. For example, master
computation allows users to run small segments of code in a non-distributed
fashion between iterations. Our system is actually one such data processing
task that is orthogonal to the algorithm itself, and takes advantage of a
similar feature provided in GraphLab. Note that this feature can easily be
implemented by alternating between running one iteration of the computa-
tion and a one-iteration implementation of the master computation, though
possibly with some performance overhead.

Another graph processing system, GPS [19], similarly implements Pregel
with some additional features. It also contributes some work on intelligent
dynamic load balancing to make graph processing more efficient for natural
to the algorithm itself, and takes advantage of a
similar feature provided in GraphLab. Note that this feature can easily be
implemented by alternating between running one iteration of the computa-
tion and a one-iteration implementation of the master computation, though
possibly with some performance overhead.

Another graph processing system, GPS [19], similarly implements Pregel
with some additional features. It also contributes some work on intelligent
dynamic load balancing to make graph processing more efficient for natural
to the algorithm itself, and takes advantage of a
similar feature provided in GraphLab. Note that this feature can easily be
implemented by alternating between running one iteration of the computa-
tion and a one-iteration implementation of the master computation, though
possibly with some performance overhead.

Another graph processing system, GPS [19], similarly implements Pregel
with some additional features. It also contributes some work on intelligent
dynamic load balancing to make graph processing more efficient for natural
to the algorithm itself, and takes advantage of a
similar feature provided in GraphLab. Note that this feature can easily be
implemented by alternating between running one iteration of the computa-
tion and a one-iteration implementation of the master computation, though
possibly with some performance overhead.

Another graph processing system, GPS [19], similarly implements Pregel
with some additional features. It also contributes some work on intelligent
dynamic load balancing to make graph processing more efficient for natural
to the algorithm itself, and takes advantage of a
similar feature provided in GraphLab. Note that this feature can easily be
implemented by alternating between running one iteration of the computa-
tion and a one-iteration implementation of the master computation, though
possibly with some performance overhead.

Another graph processing system, GPS [19], similarly implements Pregel
with some additional features. It also contributes some work on intelligent
dynamic load balancing to make graph processing more efficient for natural
to the algorithm itself, and takes advantage of a
similar feature provided in GraphLab. Note that this feature can easily be
implemented by alternating between running one iteration of the computa-
tion and a one-iteration implementation of the master computation, though
possibly with some performance overhead.

Another graph processing system, GPS [19], similarly implements Pregel
with some additional features. It also contributes some work on intelligent
dynamic load balancing to make graph processing more efficient for natural
to the algorithm itself, and takes advantage of a
similar feature provided in GraphLab. Note that this feature can easily be
implemented by alternating between running one iteration of the computa-
tion and a one-iteration implementation of the master computation, though
possibly with some performance overhead.

Another graph processing system, GPS [19], similarly implements Pregel
with some additional features. It also contributes some work on intelligent
dynamic load balancing to make graph processing more efficient for natural
to the algorithm itself, and takes advantage of a
similar feature provided in GraphLab. Note that this feature can easily be
implemented by alternating between running one iteration of the computa-
tion and a one-iteration implementation of the master computation, though
possibly with some performance overhead.
mentation in a library is unlikely to pose a problem to users. Work thus far has improved efficiency mainly by focusing on communication. One unique example of this is found in some recent (not yet published at the time of this writing) work that explores various optimizations for graph algorithms [24]: one general approach they find to speed up many computations is to perform the last few steps of a computation serially. This avoids all communication costs for a portion of the computation where the active vertices have become small enough. It is clear that this is the only avenue for improvement — if the computation is exactly preserved. Lifting the restriction of exact results opens up the possibility of simply computing less. This is particularly fruitful since cost is a highly nonlinear function of size. If a graph is too large to fit in a cluster’s memory, distributed out-of-core techniques are needed, which can be much slower (and more complex). If a graph becomes small enough to fit on fewer nodes, communication overhead is reduced for any partitioning strategy. Finally, if a distributed computation becomes a single-node one, communication cost becomes negligible and the speed up is potentially multiple orders of magnitude.

2.2 Computation models

While there are many specialized systems for graph processing, as described in Section 2.1 on page 4, some analytics can be performed even in traditional dataflow systems. For example, Naiad, a distributed system for data parallel operations that supports both batch and streaming applications efficiently, provides the PowerGraph gather-apply-scatter programming model as a library [25]. Apache Giraph, which provides a Pregel-like abstraction, is built on top of Hadoop’s MapReduce paradigm [18]. A paper from Oracle Labs researchers even argued for using a relational database combined with the in-memory graph processing of the Green-Marl compiler, which they argued from experimental evidence outperforms graph databases such as Neo4j that offer customized storage for graphs [26].

Another approach to graph analytics considers a graph in terms of its adjacency matrix. For a graph $G$ with $n$ nodes, the adjacency matrix $A \in \mathbb{R}^{n \times n}$ has $A_{ij} = 1$ if there is an edge between nodes $i$ and $j$. Note that $A$ is symmetric for an undirected graph. When a graph is sparse and only a small
fraction of the possible \( n^2 \) edges are actually present, the adjacency matrix is also sparse. Many graph algorithms can be expressed as matrix operations on the adjacency matrix [27]. Systems can take advantage of this representation and use existing software and techniques for sparse linear algebra, originally developed for scientific computation applications such as solving PDEs, to approach large-scale graph algorithms. Presto is one system that provides an R interface to graph processing, where graphs and algorithms are represented in terms of matrices and matrix operations [28]. On the theoretical side, viewing graphs as matrices lets researchers approach graph algorithms differently in terms of linear algebra [29]. The entire field of spectral graph theory is built on understanding a graph in terms of the eigenvalues and eigenvectors of its adjacency matrix.

2.3 Graph approximation

While there are not (yet) generic techniques for approximating graph analytics queries, researchers have invented techniques for specific computations of interest. Many of these approaches utilize sketch techniques borrowed from streaming algorithms in order to achieve their speed. For example, [30] uses a sketch algorithm to compute approximate distances, which it then uses to construct approximate shortest paths. HADI [31] uses the Flajolet-Martin algorithm for counting distinct elements to estimate graph radii (the maximum shortest distance from a given vertex \( v \)). HyperANF was used to compute the diameter and other interesting features of Facebook, demonstrating that Facebook users have an average distance between them of just 4.74 (this means that two users can be connected with 3.74 intermediate friends, on average) [32]. Followup work on HyperBall [33] used, among other optimizations, HyperLogLog counters, a more efficient approximate counter than the Flajolet-Martin algorithm, to compute interesting properties of the 8 billion edge ClueWeb09 web graph\(^1\).

While these approximations were able to achieve impressive results, making computation on enormous graphs possible with limited computing resources, they were created for specific algorithms that had to be re-designed with

\(^1\text{A dataset created under the NSF Cluster Exploratory program. Available at http://lemurproject.org/clueweb09/}\)
approximation in mind. Furthermore, some of the advantages gained by these sketches were due to characteristics of the graph itself (in particular, small-world or scale-free graphs are considered). Here we focus on more generic techniques that could apply to large graphs with less ideal structure, or to smaller graphs but new and experimental algorithms that are not easy to approximate.

The approach taken in the shortest path algorithm, HADI, HyperANF, and HyperBall all use approximate data structures as part of a standard algorithm and then prove properties about the graph approximation in terms of the guarantees provided by the data structures. Some more theoretical approaches instead use modern algorithmic tricks to create approximation algorithms for specific classes of problems. One common class of problems for this approach are partition problems: these are a class of problems that divide a graph into two or more partitions by assigning labels to each vertex. Examples include min-cut and $k$-coloring. These approaches also focus on particular graph models. While this makes theoretical bounds harder, worst-case bounds are often unrepresentative of what very large real-world graphs look like, and the asymptotic runtime can often be proven to be much lower than the worst-case for random graph models [34, 35]. An application of partitioning is actually to divide a graph onto multiple machines, where the goal is to minimize edges going between partitions since these edges incur communication [36]. This kind of careful partitioning can be valuable if a sequence of analytics will be performed on the same graph and the cost of partitioning is amortized over multiple jobs.

One interesting approach that could be used to approximate a graph is factorization. This is an approach used in latent variable models when studying probabilistic graphical models. In that context, random variables are nodes of a graph and edges encode the independence relationships among the variables (the exact interpretation of the edges depends on the particular model). Latent variable models study these graphs in terms of unobserved, “latent” variables that give the graph its structure [37]. Any graph can be represented in terms of latent variables, also called factors, potentially approximating its structure. This is equivalent to representing the adjacency matrix $Y$ of a graph as $Y \approx ZZ^T$, where $Z$ is tall and skinny; more precisely, if $Z \in \mathbb{R}^{n \times r}$, then $r \ll n$. $r$ is the desired number of factors [38].

Another unique approach, while not technically an approximation, is to
focus on compressing the graph. The WebGraph [39] framework and project focuses on efficient encoding of web graphs, with the idea of exploiting the structure of the web graph and streaming algorithms on top of that data to represent the graph very efficiently. This provides benefits similar to sampling: fitting the data in less RAM means fewer machines, which reduces costs and can remove the need for any communication by fitting the data in one machine with lots of memory. Their approach takes advantage of characteristics of the web graph specifically (for example, the easy-to-find redundancy of links produced by a common header page for a site), but their integer compression scheme could be used more generally for power-law graphs.

2.4 Previous graph sampling studies

Several graph sampling methods have been described and evaluated in earlier literature. An earlier paper [40] evaluated several sampling approaches for 5 (real-world) graphs for a few different computations. Part of the focus of that work was on a “back-in-time” sampling goal, which uses sampling to obtain an approximation of what a graph looked like when it was a smaller size. As a consequence the graphs evaluated were of only modest size due to the difficulty of finding time-annotated graph datasets.

A more theoretical approach to sampling is taken in a paper that presents frontier sampling [41], a computationally efficient approach that turns out to be equivalent to a multidimensional random walk. The evaluation of the approach focused on social network graphs and only considered a small number of analytics metrics, focusing on degree distribution and some global parameters. Later work from the authors considered directed graphs, whereas we focus on undirected graphs. Their findings showed with a combination of a theoretical error lower bound and an empirical study of the Flickr social network that at least for the Flickr graph, no estimator could estimate the indegree distribution effectively for a sample of the graph. This has no impact on the outdegree distribution, so when the graph is undirected the degree distribution is still easily estimated. However, their work provides strong evidence that some problems are fundamentally difficult or impossible to speed up with sampling approaches [42].
CHAPTER 3

GRAPH SAMPLING

In this chapter, we cover the details of the new sampling evaluation approach we present in this thesis. There are some necessary preliminaries on graph sampling and the specific algorithm we use here. Our new sampling evaluation approach considers several properties of the algorithm execution; we describe those properties in this section. Finally, we describe how they are synthesized into a measure of algorithm execution similarity.

3.1 Algorithms

There are three broad categories of approaches to graph sampling: methods based on random node selection, random edge selection, and random walk-based techniques. These approaches are well described in [40], where the authors find that random walk and forest-fire approaches work best. We chose to implement the forest-fire technique and it was used in all of the experiments described. For completeness, we describe the algorithm here. Future work will address other sampling techniques, as the best approach varied greatly across metrics in [40].

Consider a graph $G = (V, E)$ and a sampling target $n < |V|$ for the number of vertices in the sample graph. In forest-fire sampling, we begin with the entire graph selected. At each iteration we start “fires” at a small number of nodes, deleting it from the sample. When a node is burned, it has a chance to recursively ignite its neighbors, with a $p = 0.15$ chance of dying out. When the target number of nodes is reached, this process is stopped. It is worth noting that in our current implementation edges are selected if both incident nodes are selected and nodes without incident edges are deleted. This does not currently affect the analytics as the dropped vertices have PageRank 0, but may become significant as we expand to other analytics tasks where
isolated vertices can affect the result.

3.2 Evaluation metrics

We propose a novel way of evaluating whether a sampled graph behaves similarly to the full graph with respect to a particular algorithm for computing a graph property. Rather than focus on the result of the computation, we observe characteristics of the computation itself, which we call features. A feature for our purposes is a scalar or vector quantity computed at an iteration of the algorithm that characterizes the algorithm’s execution. Examples include the difference between successive values or the sum of values across the entire graph at each iteration. This concept is similar to the features used as inputs in machine learning. We also prefer generic features that are independent of any specific computation to avoid generating and evaluating features for different algorithms.

Generically, many algorithms can be viewed as computations on a conceptual vector \( v \in \mathbb{R}^{|V|} \), with a component for each vertex.\(^1\) For example, in PageRank each vertex has a partial value for its PageRank and while computing single-source shortest path each vertex has a tentative shortest distance from the source vertex. These algorithms fit into the pseudo code given in Algorithm 1. A listing of the features we consider that are generic to all iterative algorithms is given in Table 3.1. Each feature is normalized by \( \sqrt{|V|} \). This accounts for graphs of different sizes, resulting in comparably sized features. For example, a vector \( v \) of all 1’s has normalized size 1 regardless of number of nodes in the graph under this normalization.

### Algorithm 1

The generic form of an iterative algorithm.

\[
i \leftarrow 1
\]

**repeat**

\[
v_{i+1} \leftarrow f(G, i, v_i)
\]

\[
i \leftarrow i + 1
\]

**until** \( \|v_i - v_{i-1}\| < tol \)

We include one generic error measurement as a feature: the root mean square error (rmse). In order to compute this feature, we need \( v_\infty \), the final

\(^1\)In fact, the values at each node can be part of any vector space \( E \), so that \( v \in E^{|V|} \) would be a more accurate description.
Table 3.1: The generic features used to summarize the execution of an algorithm.

<table>
<thead>
<tr>
<th>name</th>
<th>value at iteration i</th>
</tr>
</thead>
<tbody>
<tr>
<td>diff</td>
<td>$\frac{1}{\sqrt{</td>
</tr>
<tr>
<td>2nd diff</td>
<td>$\frac{1}{\sqrt{</td>
</tr>
<tr>
<td>size</td>
<td>$\frac{1}{\sqrt{</td>
</tr>
<tr>
<td>rmse</td>
<td>$\frac{1}{\sqrt{</td>
</tr>
</tbody>
</table>

vector after convergence. Thus we first run the algorithm and save $v_\infty$ and then re-run to collect the rmse at each iteration. Applications can also have custom error measures: in the case of PageRank, often only the ranking of the vectors with the highest PageRanks is important. We then use the rank error, defined as the sum of the differences between the ranking of the top-$k$ vertices with the true PageRanks $v_\infty$ and the current PageRanks $v_i$. In our experiments we fix $k = 100$. Note that computing these error features requires having the final pageranks; in practice, we must run the algorithm once to get the final result and then run it again to gather features. The overhead of running the algorithm twice is only a cost for this assessment; for a real analytics task, there would be no need to gather these features and the algorithm would be run only once.

In GraphLab, we implement our feature calculations quite naturally in terms of aggregation operators. These are scheduled tasks which can run after every iteration before the next iteration starts. The features are calculated at each vertex and then aggregated to calculate the norm and normalize, which can then be saved to disk.

These execution features give us a unique measure of similarity between the sampled and original graph that can be defined in terms of the specific algorithm under consideration. We apply some machine learning and statistical techniques to quantify the similarity. Consider the following problem: given the execution trace in the form of a sequence of features, what is the estimated error at each iteration? We build a model of the error in terms of the features from the sampled graph; when this model predicts error well in the full graph, then the two graphs have a similar pattern of execution. This approach goes beyond simply visually comparing the features and determining when the series are “similar,” as well as quantifying the similarity.
via the error in this prediction.

For this learning problem, we have a small amount of data, consisting of our three features for each iteration. PageRank typically converges in somewhere between 10 and 40 iterations in practice across a variety of graph structures and independent of scale. In any case graph algorithms are limited in how many iterations they can take due to the expensive communication cost associated with iterations (unlike numeric algorithms that might take 100s or 1000s of steps). The features and error values are often highly nonlinear in practice. These characteristics of the estimation problem make a K-nearest neighbors (KNN) regression an effective regression technique, in which for a given iteration, the set of features computed in the sampled graph most similar to the features in the full graph are assumed to belong to the iteration with the correct error estimate. We use an implementation of KNN regression provided by scikit-learn [43].

We apply this measure of similarity, the \textit{rmse prediction error}, in the following chapter as a measure of accuracy in studying the tradeoff between accuracy and cost. In presenting actual accuracy-cost curves we derive a few characteristics and interpret them in the context of what they mean for the amenability of a graph to sampling.
CHAPTER 4
EVALUATION

We implemented rmse prediction error computation for evaluating sampling for graph properties in GraphLab. The relevant implementation includes random-walk forest fire based sampling, PageRank [44] (adapted from the GraphLab implementation), instrumentation to extract features from iterative algorithms, and computing the predicted rmse error from these features. The current evaluation only uses a handful of real graphs and randomly generated graphs combined with PageRank as an example algorithm, but it can easily be extended to other algorithms.

As noted above, we do not depend on GraphLab’s BSP programming model for our approaches. GraphLab has a newer “Warp engine,” which schedules vertex operations as tasks asynchronously with tunable consistency levels. In this framework we would no longer have a notion of iterations and would instead run feature calculations at some fixed time interval. The feature calculations would operate on a snapshot of the data, but if this staleness is not a problem for the algorithm itself then it should not be a concern for the algorithm’s execution features.

Distributed sampling over large graphs can be challenging to fit into the programming models of existing dedicated graph processing frameworks. While random walks are efficiently implemented as vertex-vertex communication, sampling random nodes or edges from the entire graph is less obviously parallelizable. Frameworks such as Spark offer more general programming models [45] that could make both types of computation efficient by treating graphs as tables of records (for each node or edge) or structured objects, depending on the task at hand [46].

While our techniques are can run in a distributed fashion (and indeed GraphLab automatically provides support for distributed execution via MPI), our evaluation is over graphs small enough to fit on one machine. We ran experiments on a single node in the Illinois Cloud Computing Testbed (CCT).
These machines have two quad-core Intel Xeon CPUs running at 2.66 GHz equipped with 16 GB of DRAM. Graphs were loaded from local disk storage.

Each graph studied was first sampled at several sampling percentages. These percentages are defined in terms of the target vertices to keep. The actual sampled graphs varied greatly in size, since the on-disk representation captured only edges and ignored vertices with no incident edges. The sampling process has not yet been optimized so performance of sampling is not reported. In particular, forest fire sampling can be much more aggressive in starting fires for very low target percentages, a feature that has not been implemented. Regardless, sampling time is dominated by graph loading and writing time for sampling percentages above 50%. We then compute PageRank for each sampled graph, as well as the original (the 100% sampling goal). We then run a second pass where we also compute the execution features described in Table 3.1 (including the error estimates, which rely on the true PageRanks $v_\infty$).

First, in the subsequent Section 4.1 we will give an overview of the data we collected for each graph and sampling goal. Next, we present results for real-world graphs in Section 4.2 on page 20 to get a sense of the performance of sampling in realistic scenarios. Finally, we evaluate the system on several random graph models to understand sampling for graphs with readily described structures and to study the effect of graph size. We performed our analysis in IPython [47] and produced the figures in this chapter with matplotlib [48].

4.1 Evaluation methodology

We define a sampling goal as a fraction less than one for the target number of vertices to keep from the original graph. Edges that are incident on deleted nodes are also deleted. All sampling percentages in this chapter represent sampling goals. When we reference a graph sampled to 100% or refer to the 100% sampling goal, this is the unchanged, full graph. For each graph, we first sampled it every 5% from 20 to 100%. We then additionally sampled every 1% from 90 to 100%. The lower bound of 20 was determined after some experiments showed that none of our real-world example graphs sampled well below 20%. The additional sampling near 100% was to better capture the
behavior of graphs that did not sample well, which often had a drastic drop in accuracy in the 90–100% range that we wanted to capture. The sampling method we used, forest-fire, works by deleting nodes and edges from the graph (rather than starting from nothing and selecting the sample), so sampling was less computationally efficient for lower percentages. This is not a fundamental limitation of the technique but requires more careful implementation, especially in a distributed setting with typical graph processing computation models.

In Figure 4.1 we explain how sampling affects performance for graphs with some illustrative examples. The sampling percentage defines the target number of nodes, but the number of resulting edges is a property of the graph structure. While the cit-patents graph has a linear relationship between the number of nodes and edges when sampled, wiki-talk is highly skewed and removing even a small number of nodes makes the graph an order of magnitude smaller in terms of edges. The number of edges in the sampled graph determines its “size” as far as the in-memory representation and runtime costs are concerned, as illustrated by the consistently linear runtime vs. size plots.

Next we synthesize the features of the sampled and full graphs by using
Figure 4.2: Synthesized sampled and full graph features using prediction.
the sampled graph to predict error using the features from the 100% run in Figure 4.2 on the preceding page. The $y$ curve in the upper group of plots is the true rmse in the 100% run; it is repeated in each graph for direct comparison. The $\hat{y}$ curve gradually becomes more accurate as more nodes are sampled. Note that the predicted rmse for skitter-100 is created by modeling error as a function of execution features and then applying the model to the same features used for creating the model. The error inherent to using K-Means as a model is evident in this plot, where the $\hat{y}$ curve is clearly not the same as the $y$ curve.

As a measure of similarity between the sets of features, we use the relative prediction error $\|y - \hat{y}\| / \|y\|$ when predicting the rmse of the PageRank; higher prediction error means less similarity. Each of the top six plots becomes a single one of the points in the summary plot at the bottom of Figure 4.2. Looking at prediction error over sampling percentage quantifies the closeness of PageRank execution for the sampled graphs against the full graph. The point at 100% sampling is the best rmse function K-nearest neighbors regression can model and represents a baseline for prediction error. If the prediction were more expressive or the function were easier to fit, the error at 100% sampling could be 0, but in practice it is higher. In any case it serves as a baseline for the other sampling percentages. Taking this into account, the samples from 60–100% have relatively low prediction error and thus are close samples for PageRank. The prediction error rising for 50–60% indicates that the prediction is not robust to different random samplings below the 60% sampling goal threshold.

![Figure 4.3: Combined runtime and prediction error for the skitter graph.](image)

To understand the poor sampling performance for 50-60% in the skitter graph observed in Figure 4.2, we look at some performance variables in Fig-
Figure 4.3 on the previous page: the size of the graph and the total runtime of PageRank. Generally the size increases with sampling percentage as expected, though not linearly since sampling percentage is defined in terms of nodes while size is measured in edges. Error generally decreases with greater runtime, but does roughly level off about 11 s. Both graphs have data points connected in order of sampling percentage. Because the size of the graph has high variance across samplings and each sampling percentage is independent, the noise manifests in a non-monotonic size over sampling percentages and makes the other graphs zig-zag.

The error-runtime graph captures the sampling variance in its zig-zag. It also directly shows the relationship between cost, in terms of runtime, and the resulting error estimation. Low error indicates that the algorithm, PageRank in this case, executes similarly in the sampled graph to the original graph, implying that the sample should perform well. In this example, we can see that even when the graph is sampled to reduce runtime by about a factor of 2, PageRank performs similarly.

The two deviations from the expected behavior — the decrease in size even as more nodes are sampled in the size graph and the zig-zag in the error-runtime graph — are related. The sampling process has high variance, so in this case the graphs with a 7 s runtime occasionally randomly be very close to the full graph and have low prediction error, but many times they may have much higher error. Setting a threshold for the sampling of this graph requires making a decision not only of the desired error, but also the probability of obtaining that error.

To get a sense of the overall structure of a graph, one simple tool is the degree distribution, illustrated for the skitter example in Figure 4.4. The
degree of a vertex is the number of edges incident on it; the distribution of degrees for all the vertices gives one view of the graph structure. The particular graph shown in the figure is the complementary cumulative degree distribution, giving \( F(d) \) where \( F(d) \) is the number of vertices with degree greater than \( d \). The distribution is unnormalized so that the sampled graphs are scaled by the sampling percentage. The skitter graph demonstrates a near power-law degree distribution (which would be a straight line on a log-log plot). As is typical for real-world, finite graphs, the tail is cut off, in this case fairly sharply.

We also see what happens to the graph structure as it is sampled: the graph nearly retains its structure. The sampling variance is difficult to see in this plot, but manifests itself in that the 50% distribution is actually out of order, falling just above the 20% line. This does not indicate a poor sample, merely one that is unexpectedly small; while sampling, this could be prevented if desired by stopping the forest fires when the number of edges drops to a specified threshold.

4.2 Real-world graphs

The real-world graphs studied are listed below with a brief description of the source material and graph characteristics.

- **amazon**, the co-purchasing graph of products on Amazon [49]. Products \( i \) and \( j \) have an undirected link between them if \( i \) and \( j \) are frequently purchased together. \(|V| \approx 3.3 \times 10^5, |E| \approx 9.3 \times 10^5\).

- **cit-patents**, a graph of citations among United States patents [50]. \(|V| \approx 3.8 \times 10^6, |E| \approx 1.7 \times 10^7\).

- **lj**, a graph of the LiveJournal social network friend relationships [49]. \(|V| \approx 4.0 \times 10^6, |E| \approx 3.5 \times 10^7\).

- **skitter**, an autonomous systems Internet topology graph [50]. \(|V| \approx 1.7 \times 10^6, |E| \approx 1.1 \times 10^7\).

- **youtube**, the social network component of YouTube users [49]. \(|V| \approx 1.1 \times 10^6, |E| \approx 3.0 \times 10^6\).
wiki-talk, a network of Wikipedia users connected if a user edited the talk page of the other user [3]. $|V| = 2.4 \times 10^6, |E| = 5 \times 10^6$.\footnote{This graph has many disconnected vertices; there are only $6 \times 10^5$ vertices with at least one incident edge.}

In Figures 4.5 to 4.6 on pages 22–23 we give an overview of the graphs with two important plots of results. The error-runtime tradeoff, as described in Figure 4.3 on page 18, gives the important tradeoff between runtime and feature similarity. The degree distributions for the graph and its samples are important for understanding the general structural differences between the graphs and the implications of structure for sampling. We will use degree distributions to make a connection between random graph models and these real-world graphs in Section 4.3 on page 25.

We make some observations on the above results to help interpret what they imply about the sampling of each of these graphs, ordering them roughly in order of best sampling performance to worst. The four graphs for which sampling is effective are presented in Figure 4.5 on the next page. The skitter graph has the best sampling performance, with nearly no degradation of prediction error for sampling that reduces the runtime to 40% of the original. However, it does suffers from some sampling variance at the low sampling percentages (recall that the points on the error-runtime graph are connected in order of sampling percentage, so the near monotonicity of this curve indicates consistent results). The youtube graph performs similarly well. The cit-patents graph samples fairly well, with the error increasing gradually as the sampling percentage is reduced. Notably, there is little sampling variance in this graph. The degree distribution of the samples is also similar in shape for all percentages and very similar for sampling percentages above 60%. The lj graph appears to sample well down to a runtime of about 75% of the original, but suffers from a large sampling variance below that level. The lj graph changes drastically with very few nodes deleted. Even so, the 98 and 99% sampling goals achieve the 75% reduction in runtime. This is particularly convenient for forest fire sampling, where the cost is proportional to the number of deleted nodes. In some ways the lj graph may actually sample the best, since the actual difference between the prediction error at its minimum and maximum over this range of sampling percentages is only from 0.50 to 0.58, in contrast with the other graphs, where at 20% sampling
Figure 4.5: Results for graphs where sampling is effective.
Figure 4.6: Results for graphs where sampling does not work well.
the error is nearly 1 (which indicates an error of the same magnitude as the function itself).

On the other hand, for two graphs sampling is very ineffective, which are presented in Figure 4.6 on the preceding page. The amazon graph does not sample well at all: the relative error goes to above 1 for any sampling and the degree distribution has a very different slope. The drastic change in behavior with even 99% sampling is similar to lj, but instead of just reducing runtime, here the much smaller graph samples poorly. Not only that, but the relative prediction error is greater than 1 at every sampling percentage. The wiki-talk graph has similarly poor sampling performance: the graph changes drastically with the slightest sampling. The large gap in runtimes between 100% and 99% indicates that a small fraction of nodes are responsible for most of the edges, and when they are not sampled PageRank converges quickly and very differently. The degree distribution corroborates this, with a large gap between 100% and 90% (recall the $y$-axis is in log scale).
4.3 Random graph models

We consider four random graph models: Erdős-Rényi, power-law generated via Barabási-Albert preferential attachment, random regular, and one application of the Chung-Lu model.

The Erdős-Rényi model is the simplest and oldest random graph model. There are minor variants in its definition, but we use the following: a \( G(n, p) \) graph is an undirected graph on \( n \) vertices where each pair of nodes is connected with probability \( p \), independent of all other pairs. One property of these graphs is that they rapidly become connected when \( p > \frac{\ln n}{n} \) as \( n \to \infty \).

A power-law graph is a graph whose degree distribution follows a power law; that is, for some \( \alpha > 1 \), \( F(d) \propto d^{-\alpha} \) where \( F(d) \) gives the number of nodes in the graph with degree greater than \( d \) (the complementary or inverse CDF of the degree distribution). Such graphs can be generated by the Barabási-Albert preferential attachment model, in which edges are randomly added to a graph with probability proportional to the degree of the incident nodes (edges are added in a “rich get richer” fashion). This algorithm can be shown to result in power-law graphs as \( n \to \infty \).

A \( d \)-regular graph is a graph where every node has degree exactly \( d \). Such graphs are essentially never seen in the real-world but are interesting from a theoretical perspective. We evaluate our system on a random regular graph, generated according to an algorithm described in [51].

Finally, we wanted to study a graph for which sampling was ineffective. Based on the real-world graphs, we decided to generate a graph with degree distribution similar in skew to the wiki-talk graph. The Chung-Lu model [52] studies graphs \( G(\vec{w}) \), where \( \vec{w} \in \mathbb{R}^n \) is a vector of weights specifying the expected degree distribution. Specifically, nodes \( i \) and \( j \) are connected randomly with probability proportional to \( w_i w_j \) independent of the other edges. NetworkX implements an algorithm described in [53], which we use to generate a graph with an expected degree distribution similar to the wiki-talk distribution. Because the graph is finite and our \( \vec{w} \) is not exactly the wiki-talk degree distribution, the generated graph does not have exactly the same degree distribution, but its behavior still resembles that of wiki-talk.

The short names for the specific graphs we present results for are described in Table 4.1 on the next page. All of the graphs were created with NetworkX [54]. These random graph models afford us an opportunity to study sampling
medium-er

![Graph](image)

big-er

![Graph](image)

Figure 4.7: Results for Erdős-Rényi random graphs.

in a context where the structure of the graph is much more readily described and understood. We also have the ability in this context to vary the size while preserving some essential quality of the structure, as these graph models have a size parameter. In Figures 4.7 to 4.10 on pages 26–29 we present a similar set of results as in Section 4.2 on page 20.

Table 4.1: Random graphs studied

| short name    | model          | $|V|$       | $|E|$       |
|---------------|----------------|-----------|-----------|
| big-er        | Erdős-Rényi    | $1 \times 10^6$ | $1 \times 10^7$ |
| medium-er     | Erdős-Rényi    | $5 \times 10^4$ | $1 \times 10^6$ |
| big-power     | Barabási-Albert| $1 \times 10^6$ | $1 \times 10^7$ |
| medium-power  | Barabási-Albert| $5 \times 10^4$ | $1 \times 10^6$ |
| big-regular   | regular        | $1 \times 10^6$ | $1 \times 10^7$ |
| medium-regular| regular        | $5 \times 10^4$ | $1 \times 10^6$ |
| gen-wiki      | Chung-Lu       | $1.3 \times 10^5$ | $6.2 \times 10^5$ |

One general result we can draw from these models is that they generally perform much better than the real-world graphs (gen-wiki is an exception, but was generated with the goal of being difficult to sample). The power-
medium-regular

big-regular

Figure 4.8: Results for random $d$-regular graphs.
medium-power

big-power

Figure 4.9: Results for power-law graphs.
Figure 4.10: Results for the gen-wiki Chung-Lu graph, along with the wiki-talk graph for comparison.
law and regular graphs in particular perform very well. Another general result we find is that a similar degree distribution is not a requirement for good sampling performance: the regular graphs slowly morph into Erdős-Rényi-like graphs as they are sampled, with the curvature different for every sampling percentage, but PageRank behaves very similarly down to about half the runtime anyway. While power-law graphs perform very well and maintain their characteristic distribution with the same slope, they have much higher variance than the regular and Erdős-Rényi graphs. While none of our real-world graphs appears to resemble an Erdős-Rényi graph, we do note that the skitter and youtube graphs, which have very good sampling performance, look similar to the random power-law graphs.

The behavior of power-law graphs under sampling is particularly noteworthy. The random Erdős-Rényi graphs show fairly consistent sampling behavior, with little variance and linear error-runtime graphs. Even the random regular graphs show little sampling variance and a monotonic, though nonlinear, error-runtime tradeoff. The power-law graphs, by contrast, have very high sampling variance, making them difficult to sample simply because of the possibility of getting an unlucky graph dissimilar from the original. Power-law graphs are known to be resilient to random node failures, in the sense of retaining a high level of connectivity compared to graphs with more uniform degree distributions (which includes Erdős-Rényi graphs of sufficient size) [55]. However, they are also vulnerable to targeted attacks on the few nodes that are important to network connectivity. Our sampling strategy is somewhere in between uniformly random and targeted: fires that delete nodes are started randomly, but spread along edges. Thus the central nodes that connect the graph are more likely to be hit by a fire. For an algorithm like PageRank, which identifies central nodes, the behavior is likely to be very different on a poorly connected graph and a well-connected one.

This same reasoning explains the even worse sampling of the gen-wiki graph, which was explicitly constructed to have a skewed degree distribution. However, whereas the power-law graphs see higher variance, gen-wiki is so extreme that almost any sampling deletes the handful of high degree nodes and drastically changes the graph. We make the observation from these cases that power-law may be inherently difficult to sample, or at least require a sampling technique that preserves their connectivity, such as uniform random node deletion.
It is also worth noting that this behavior is quite distinct in real-world graphs that appear to follow a power law, namely the skitter and youtube graphs. These graphs do have high sampling variance, but still have a flat portion of the error-tradeoff curve where they can be sampled with little change. The differences between real world and true power-law graphs are subtle, but the important difference for this case appears to be fewer of the very high degree nodes in real world graphs compared to randomly generated power-law graphs, manifested as a drop-off at the far end of the inverse degree distribution. This fits the explanation well: real-world graphs have slightly more nodes bearing the burden of connecting the graph.

In comparing the “medium” and “big” random graphs for the first three random graph models, we find that size does not affect the character of the results. Some slight differences are to be expected since the random models and generation algorithms are imperfect: for example, medium-power is not as close to an actual power law as big-power.

4.4 Characteristics of Tradeoff Curves

We have presented above the specific curves we use to study the tradeoff between accuracy and cost for several graphs. In particular, we use the rmse prediction error, the new metric described in Section 3.2, as the accuracy side of the tradeoff. For the cost side, which is intended to capture the resources consumed by the algorithm, we use the algorithm runtime. An alternative to runtime is the graph size. We prefer runtime because it is more easily understood and because in a cluster environment the cost of a job is in some sense the opportunity cost of running another job, and reducing the runtime of a job directly lowers its impact on cluster utilization, unlike, for example, the memory savings of a smaller graph.

Here we highlight characteristics of tradeoff curves and interpretations of these patterns in terms of the amenability of the graph to sampling for the chosen graph property, summarizing the important takeaways from the above graphs. The most obvious characteristic in these tradeoff curves is the minimal sampling percentage where the results are still similar. For the rmse prediction error, the threshold for similarity must be set in terms of the error at 100%, which varies from graph to graph. When the tradeoff curves
is almost linear this can pose a problem for interpretation, but this rarely happens; often there is an obvious sharp cutoff and its location is informative.

The shape of the tradeoff curve is useful to note. A sharp drop in error for increased cost, followed be a level segment till 100% sampling, is the perfect situation for sampling, where any sampling goal in the level segment has good accuracy while achieving reduced cost. On the other hand, there are graphs where any sampling quickly introduces error. These graphs are unlikely to sample well. Linear tradeoff curves are somewhere in between; for these graphs, it may be necessary to define a more precise, domain-specific accuracy measure where the minimum sampling goal can be determined by an accuracy threshold that is chosen for the situation.

We present tradeoff curves with points at each sampling goal, giving an additional dimension to the curve. One aspect that is apparent from this visualization is the rate at which the graph moves through the tradeoff space as it is sampled. A linear tradeoff curve can be a result of a gradual increase of error as the graph is sampled to lower sampling goals, or it can have a sharp increase; see the cit-patents tradeoff curve in Figure 4.5 on page 22 for an example of the former and the wiki-talk tradeoff curve in Figure 4.6 on page 23 for a stark example of the latter. This sharp increase means practically speaking, there are graphs where some of the tradeoff space is hard to target.

Another characteristic that these points afford is that they are connected in order of sampling percentage. The ideal curves are properly random random variables since the samples are random, where the mean of the curve should be monotonic, decreasing when measuring error vs cost. In addition, lower sampling percentages should always result in lower or equal cost on average. The curves do not always demonstrate this since the sampled points are random and independent (e.g., the 80% graph is not sampled from the 90% graph, but instead an independent sample of the full graph). These mis-ordered points indicate that the graph might be sensitive to randomness in the sampling process. This variance is undesirable because accuracy of the mean is only useful when the variance does not make the result unreliable in the sense of attributable to chance. We note that when sampling record data for aggregate queries, it is possible to compute the variance due to the sampling percentage when the data sets are large enough from the theory behind the central limit theorem. No analogous result exists for graph sampling —
and such a straightforward characterization as the central limit theorem for all graphs seems unlikely given current research.
CHAPTER 5

CONCLUSION

We have presented a systematic characterization and study of the accuracy vs cost tradeoff in graph sampling. We include a new approach to evaluating a sample, comparing the sample and the full graph based on the execution of a particular algorithm. We also describe how to interpret various characteristics of accuracy-cost tradeoff curves. While we have presented our study on PageRank with a handful of real-world and random graphs, the approach presented can be applied to more diverse scenarios. A result of our own study is that the scale of a graph has little effect on the performance of sampling, while structure is very significant. Certain common graph models, including random (Erdős-Rényi) and power-law, demonstrate good sampling performance, at least in the context of PageRank. We also find that power-law graphs do not sample well with the forest fire sampling algorithm that was successful for random graphs and this class of graph requires considering a different approach.

As future work we believe it is possible to predict whether a graph is easy to sample and to what percentage it can be safely sampled for a variety of algorithms based only on graph structure. An extension of this idea is to explore sampling algorithms in conjunction with graph structure, to understand which sampling algorithm to apply in a given situation. This approach differs from existing work that aims to provide good results on average for several graph properties; with a systematic study, we could provide better advice on which algorithm to use as a function of both graph structure and the intended computation. Finally, we note that while in this thesis we have focused on vector graph properties, the approach can be applied to scalar graph properties as well (e.g., diameter, number of connected components) and used for the same purposes on those algorithms.

Once we have sampled graphs, many applications become feasible to explore. For some analytics, it might be possible to learn something about the
full graph from the results for the smaller graph; however, this might require developing unbiased estimators for each measure of interest as in [15]. For some algorithms, such as PageRank, the results for the sample graph could initialize the algorithm for the large graph, accelerating convergence. Regardless of the use case, we have provided the foundations to understand from an empirical perspective questions about the applicability of sampling to graph analytics.
REFERENCES


