Streaming Graph Partitioning for Large Distributed Graphs

Isabelle Stanton*
University of California, Berkeley
Berkeley, CA
isabelle@eecs.berkeley.edu

Gabriel Kliot
Microsoft Research
Redmond, WA
gkliot@microsoft.com

Abstract

Extracting knowledge by performing computations on graphs is becoming increasingly challenging as graphs grow in size. A standard approach distributes the graph over a cluster of nodes, but performing computations on a distributed graph is expensive if large amount of data have to be moved. Without partitioning the graph, communication quickly becomes a limiting factor in scaling the system up. Existing graph partitioning heuristics incur high computation and communication cost on large graphs, sometimes as high as the future computation itself. Observing that the graph has to be loaded into the cluster, we ask if the partitioning can be done at the same time with a lightweight streaming algorithm.

We propose natural, simple heuristics and compare their performance to hashing and METIS, a fast, offline heuristic. We show on a large collection of graph datasets that our heuristics are a significant improvement, with the best obtaining an average gain of 76%. The heuristics are scalable in the size of the graphs and the number of partitions. Using our streaming partitioning methods, we are able to speed up PageRank computations on Spark [36], a distributed computation system, by 18% to 39% for large social networks.

A newer version of this manuscript has appeared at 18th ACM SIGKDD Conference on Knowledge Discovery and Data Mining (KDD 2012), Beijing, China.

*The author was supported by NSF and NDSEG graduate research fellowships and NSF grant CCF-0830797. The work was partially completed while the author was an intern at Microsoft Research.

1 Introduction

Modern graph datasets are huge. The most obvious example is the World Wide Web, where crawls by large search engines currently consist of over one trillion links and are expected to exceed ten trillion within the next year. Individual websites also contain enormous graph data. At the time of writing, Facebook consisted of over 800 million active users, with over one hundred billion friend links [1]. There are over 900 million additional objects (communities, pages, events, etc.) that interact with the user nodes. In July 2009, Twitter had over 41.7 million users with over 1.47 billion social relations [22]. Since then, it has been estimated that Twitter has grown to over 200 million users, of which well over 100 million are active. Examples of large graph datasets are not limited to the Internet and social networks - biological networks like those representing protein interactions are of a similar size. Despite the size of these graphs, it is still necessary to perform computations over the data, such as calculating PageRank, broadcasting Twitter updates, using Steiner tree solutions to identify protein associations [13], as well as many other applications.

The above graphs can consist of terabytes of compressed data when stored on disks, and are all far too large for a single commodity type machine to efficiently perform computations. A standard solution is to split the data across a large cluster of commodity machines and then use parallel, distributed algorithms for the computation. This approach introduces a host of systems engineering problems, of which we focus only on the problem of data layout. For graph data, this is called *balanced graph partitioning*. The goal is to minimize the number of cross partition edges, while keeping the number of nodes in every partition balanced.

Good graph partitioning algorithms are very useful for many reasons. First, graphs that we encounter and care about in practice are not random. The edges display a great deal of locality, whether due to the vertices being geographically close in social networks, or related by topic or domain on the web. This locality gives us hope that good partitions, or at least partitions that are significantly better than random cuts, exist in real graphs. Next, inter-machine communication, even on the same local network, is substantially more expensive than inter-processor communication. Network latency is measured in microseconds while inter-process communication is measured in nanoseconds. This disparity substantially slows down processing when the network must be used. For large graphs, the data to be moved may border on gigabytes, causing network links to become saturated.

The primary problem with partitioning complicated graph data is that it is difficult to create a linear ordering of the data that maintains locality of the edges. By this, we mean that if it is possible to embed the vertices of a graph into a line such that none of the edges are 'too long', then a good balanced cut exists in the graph. Note that such an ordering may not even exist at all. It is well-known that there is a strong connection between graph partitioning and the eigenvectors and eigenvalues of the corresponding Laplacian matrix of the graph via the Cheeger bound [30]. This connection has inspired many spectral solutions to the problem, including the Arora-Rao-Vazirani algorithm [11] and the many works that followed.

However, spectral methods do not scale well to big data, in part because of the running time, and in part because current formulations require full information about the graph. This has inspired local spectral partitioning methods, like EvoCut [9], but local methods still need access to large portions of the graph, rely on complex distributed coordination, and also still require a large amount of computation after the data has been loaded. Thus, we look for a new type of solution. In this paper, we concern ourselves with the problem of a *graph loader*. A *graph loader* is a program that reads serial graph data from a disk onto a cluster. It must make a decision about the location of each node as it is loaded. The goal is to find a close to optimal balanced partitioning with as little computational overhead as possible. This problem is also known as *streaming graph partitioning*.

For some types of graphs, the partitioning problem can be entirely bypassed by using the rich meta data associated with the vertices. For instance, clustering web page by URL produces a quite good partitioning for the web. Another example is that people tend to be friends with people who are geographically nearby. When such data for a social network is available, this produces an improved cut over a node ID hashing approach. Unfortunately, this data might not always be available in a graph, and even if it is, it is not always clear which features are useful for partitioning. Our goal in this work is to find a general streaming

algorithm that relies only on the graph structure and works regardless of the graph data.

1.1 Applications

Our motivating example for studying this problem is a large distributed graph computation system. This is far from an abstract problem. All distributed computation frameworks, like MapReduce, Hadoop, Orleans [16] and Spark [37] have methods for handling the distribution of data across the cluster. Unfortunately, for graphs, these methods are not tuned to minimize communication complexity, and saturating the network becomes a significant barrier to scaling the system up. Recently, the popularity of building distributed systems for graph computation has exploded, especially due to interest from the database community. Examples of these systems include Pregel [26], GraphLab [25], InfiniteGraph, HyperGraphDB, Ne04j, and Microsoft's Trinity [4] and Horton [3], to name but a few. Even for these graph specific systems, the graphs are laid out using a hash of the node ID to select a partition. If a good pseudorandom hash function is chosen, this is equivalent to using a random cut as the graph partitioning and will result in approximately balanced partitions. However, computations on the graph run more slowly when a hash partitioning is used instead of a better partitioning, due to the high communication cost. Fortunately, these systems tend to support custom partitioning, so it is relatively easy to substitute a more sophisticated method, provided it scales to the size of the graph. As our experiments show, even using our simple streaming partitioning techniques can allow systems of this type to complete computations at least 20% faster.

1.2 Theoretical Difficulties

At first glance, theoretically, a good streaming graph partitioning algorithm is impossible. It is extremely easy to create example graphs and orderings for any algorithm that will cause it to perform poorly. As a simple example, consider a graph consisting of a cycle. The optimal balanced 2-partition cuts only 2 edges. On the other hand, if the vertices are given in an order of 'all even nodes then all odd nodes', we won't observe any edges until the odd nodes arrive. Without any information, the best an algorithm could do would be to try and balance the number of vertices it has seen across the 2 partitions. This leads to an expected cut of $\frac{n}{4}$ edges. The worst algorithm might put all even nodes in one partition leading to all edges being cut!

We can partially bypass this problem by picking the input ordering ourselves. The three popular orderings considered in the literature are adversarial, random, and stochastic. The above cycle example is an adversarial order and demonstrates that the streaming graph partitioning problem may have arbitrarily bad solutions under that input model. Given that, and the fact that adversarial input is unrealistic in our setting - we have control over the data - we focus on input that results from either a random ordering, or the output of a graph search algorithm. The second option, a graph search ordering, is a simplification of the ordering returned by a graph crawler.

We leave open the theoretical problem of analyzing the performance of our heuristics. While there are current approaches to analyzing streaming algorithms, there are two primary issues with applying them to streaming graph partitioning. The first is that previous analytic techniques usually assume that the data is either in a worst-case or randomized ordering. We don't expect any streaming graph partitioning algorithm to perform well in a worst-case ordering as it is too easy to 'hide' the edges from the algorithm. Additionally, our experiments show that both breadth-first and depth-first orderings of a graph have superior performance for many heuristics, yet this makes the ordering difficult to analyze. The second complication is that graph partitioning is an NP-hard problem, even when given full information. While there has been some work in online NP-hard problems, like bin packing, one usually compares the performance to an offline optimal solution. For graph partitioning, this optimal solution is hard to compute, yet comparing with the solution given by an offline approximation algorithm may be even more complicated.

1.3 The Streaming Model

In this paper, we consider a simple streaming graph model. We have a cluster of k machines, each with memory capacity C, such that the total capacity, kC, is large enough to hold the whole graph. The graph is $G = (V, E)$ where V is the set of vertices, and E is the set of edges. The graph may be either directed or undirected. The vertices arrive in a stream in some order with the set of edges where it is a member. We consider three orders: random, breadth-first search and depth-first search. As vertices arrive, a partitioning algorithm must decide to place the vertex on one of the k machines. A vertex is never moved after it has been placed. In order to give the heuristics maximal flexibility, we allow the partitioning algorithm access to the entire subgraph defined by all vertices previously seen. This is a rather strong assumption, but the heuristics studied in this paper use only local (depth 1) information about this subgraph. We extend the model by allowing a buffer of size $b \leq C$ so that the partitioning algorithm may decide to place any node in the buffer, rather than the one at the front of the stream. In this work we consider only $b = 1$ or $b = C$.

Our model assumes serial input and a single loader. This is somewhat unrealistic for a real system where there may be many graph loaders working in parallel on independent portions of the stream. While we will not explore this option, the heuristics we investigate can be easily adapted to a parallel setting where each loads its portion of the graph independently from the others, sharing information only through a distributed lookup table of vertices to partition IDs.

1.4 Contributions

We provide a rigorous, empirical study of a set of natural heuristics for streaming balanced graph partitioning. To the best of our knowledge, this is a first attempt to study this problem. We evaluate these heuristics on a large collection of graph datasets, from various domains, such as the World Wide Web, social networks, finite-element meshes and synthetic datasets from some popular generative models - preferential-attachment [14], RMAT [24] and Watts-Strogatz [33]. We compare the results of our streaming heuristics to both the standard approach of hash based partitioning, and METIS [20], a well-regarded, fast, offline partitioning heuristic.

Our results show that some of the heuristics are good and some are surprisingly bad. Our best performing heuristic is a weighted variant of the greedy algorithm. It has a significant improvement over the hashing approach without significantly increasing the computational overhead and obtains an average gain of 76% of the possible improvement in the number of edges cut. On some graphs, with some orderings, a variety of heuristics obtain results which are very close to the offline METIS result. By using the synthetic datasets, we are also able to show that our heuristics scale with the size of the graph and the number of partitions. We demonstrate the value of the best heuristic by using it to partition both the LiveJournal and the Twitter graph for PageRank computation using the Spark cluster system [36]. These are large crawls of real social networks, and we are able to improve the running time of the PageRank algorithm by 18% to 39% by changing the data layout alone. Our experimental results motivate us to recommend that this is an interesting problem worthy of future research and is a viable preprocessing step for graph computation systems.

Our streaming partitioning is not intended to substitute for a full information graph partitioning. Certain systems or applications that need as good a partitioning as possible will still want to repartition the graph after it has been fully loaded onto the cluster. These systems can still greatly benefit from our optimization as a distributed offline partitioning algorithm started from an already reasonably partitioned graph will require less communication and may need to move fewer vertices, causing it to run faster. Our streaming partitioning algorithms can be viewed as a preprocessing optimization step that cannot hurt in exchange for a very small additional computation cost for every loaded vertex.

2 Related Work

Graph partitioning has a rich and varied history. It encompasses many different problems, and has many proposed solutions, from the very simple to the very sophisticated. We cannot hope to give proper coverage to the whole field and will only focus on the most relevant formulation - balanced k-partitioning. The

goal is, given a graph G as input and a number k, to cut G into k balanced pieces while minimizing the number of edges cut. This problem is known to be NP-Hard, even if one relaxes the balanced constraint to 'approximately' balanced [10]. There are many approximation algorithms for this problem. Andreev and Racke give an LP-based solution that obtains a $O(\log n)$ approximation [10]. Even *et al.* [17] provide a different LP formulation based on spreading metrics that also obtains an $O(\log n)$ approximation ratio. Both require full information about the graph and are offline solutions. There are many offline heuristics that also solve this problem with an unknown performance guarantee, like METIS [20], PMRSB [15], Chaco [18], and many more. In practice, these heuristics are quite effective, but many are intended for scientific computing purposes and focus on finite element meshes. One can also use any balanced 2-partitioning algorithm to obtain an approximation to a balanced k-partitioning when k is a power of 2, losing at most a $\log n$ factor [11].

While we are unaware of any previous work on the exact problem statement that we study - one pass balanced k partitioning - there has been much work on many related streaming problems, primarily graph sparsification in the semi-streaming model, cut projections in the streaming model as well as online algorithms in general, like online bipartite matching. This work includes both algorithms and lower bounds on space requirements.

The first area is streaming graph problems where multiple passes on the stream are allowed. This work includes estimating PageRank [32] and cut projections [31]. While PageRank has been used for local partitioning [8], the approach in [8] uses personalized PageRank vectors which does not easily generalize the approach in [32]. Additionally, cut projections do not maintain our balanced criterion, and the algorithm uses multiple passes.

Closer to our work is that of Bahmani *et al.* [12] where, assuming the nodes are fixed but the edges arrive in adversarial order, they are able to maintain an accurate estimate of the PageRank of each node with one pass. This is an example of the *semi-streaming* model. In the semi-streaming model, further results are known with regards to finding minimum cuts, i.e. no balance requirement. In particular, Jin Ahn and Guha [6] give a one pass $\tilde{O}(n/\epsilon^2)$ space algorithm that sparsifies a graph such that each cut is approximated to within a $(1+\epsilon)$ factor. Kelner and Levin [21] similarly produce a spectral sparsifier with $O(n \log n/\epsilon^2)$ edges in $\tilde{O}(m)$ time. While both of these results are related to partitioning, we do not have a prespecified computation task, so we cannot be sure that a sparsified graph will give accurate answers. Finally, lower bounds are known with regards to the space complexity of both the problem of finding a minimum and maximum cut. Zelke [38] has shown that this cannot be computed in one pass with $o(n^2)$ space. By contrast, our methods require at most $O(n)$ processing space, but do not have any theoretical guarantees.

Finally, also relevant are online algorithms. Again, this a broad area so we can mention only the closest influence on our work is that of Mehta *et al.* [27] on an algorithm for AdWords. Here, the problem is a generalized online bipartite matching - one side of the graph is fixed with a certain budget, and the other side appears online with all edges. They show that a modified greedy algorithm where the selected match maximizes the product of the greedy choice with a penalty based on the fraction of the budget left is optimal. This paper inspired our weighted heuristics.

3 Heuristics and Stream Orders

In this paper, we will examine multiple heuristics and stream orders. We now formally define each one.

3.1 Heuristics

The notation P^t refers to the set of partitions at time t. Each individual partition is referred to by its index $P^t(i)$ so $\cup_{i=1}^{k} P^t(i)$ is equal to all of the vertices placed so far. Let v denote the vertex that arrives at time t in the stream, $\Gamma(v)$ refers to the set of vertices that v neighbors and $|S|$ refers to the number of elements in a set S. C is the capacity constraint on each partition, i.e. at all times t, for each partition i, $|P^t(i)| \leq C$. Each of the heuristics gives an algorithm for selecting the index *ind* of the partition where v is assigned. The first seven heuristics do not use a buffer, while the last three do.

1. Balanced - Assign v to a partition of minimal size, breaking ties randomly.

$$ind = \arg\min_{i \in [k]} \{|P^t(i)|\}$$

2. Chunking - Divide the stream into contiguous chunks of size C and fill the partitions completely, one by one.

$$ind = \lceil t/C \rceil$$

3. Hashing - Given a hash function $H : V \to \{1 \cdots k\}$, assign v to

$$ind = H(v)$$

We use $H(v) = (v \mod k) + 1$ in the experiments.

4. (Weighted) Deterministic Greedy - Assign v to the partition where it has the most edges. Weight this by a penalty function based on the capacity of the partition, penalizing larger partitions. Break ties using Balanced.

$$ind = \arg\max_{i \in [k]} \{|P^t(i) \cap \Gamma(v)| w(t,i)\}$$

where $w(t,i)$ is a weighted penalty function:

- $w(t,i) = 1$ for unweighted greedy
- $w(t,i) = 1 - \frac{|P^t(i)|}{C}$ for linear weighted
- $w(t,i) = 1 - \exp\{|P^t(i)| - C\}$ for exponentially weighted

5. (Weighted) Randomized Greedy - Assign v according to the distribution defined by

$$Pr(i) = \frac{|P^t(i) \cap \Gamma(v)| w(t,i)}{Z}$$

where Z is the normalizing constant and $w(t,i)$ is the above 3 penalty functions.

6. (Weighted) Triangles - Assign v according to

$$\arg\max_{i \in [k]} \{\frac{|E(P^t(i) \cap \Gamma(v), P^t(i) \cap \Gamma(v))|}{\binom{|P^t(i) \cap \Gamma(v)|}{2}} w(t,i)\}$$

where $w(t,i)$ is the above 3 penalty functions.

7. Balance Big - Assume we have some way of differentiating high and low degree nodes. If v is of high-degree, use Balanced. If it is of low-degree, use Deterministic Greedy.

The following heuristics all use a buffer.

8. Prefer Big - Maintain a buffer of size C. Assign all high degree nodes with Balanced, and then stream in more nodes. If the buffer is entirely low degree nodes, then use Deterministic Greedy to clear the buffer.

9. Avoid Big - Maintain a buffer of size C and a threshold on large nodes. Greedily assign all small nodes in the buffer. When the buffer is entirely large nodes, use Deterministic Greedy to clear the buffer.

10. Greedy EvoCut - Use EvoCut [9] on the buffer to find small Nibbles with good conductance. Select a partition for each Nibble using Deterministic Greedy.

Each of these heuristics has a different motivation with some arguably more natural than others. Balanced and Chunking are simple ways of load balancing while ignoring the graph structure.

Hashing is a popular method currently used by many real systems [26]. The benefit of Hashing is that every vertex can be quickly found, from any machine in the cluster, without the need to maintain a distributed mapping table. If the IDs of the nodes are consecutive, the hash function $H(v) = (v \mod k) + 1$ makes Balanced and Hashing equivalent. More generally, when the distribution of node IDs is unknown or not uniform, a pseudorandom hash function should be used, which makes Hashing equivalent to a random cut.

The greedy approach is standard in optimization algorithms, although the weighted penalty is inspired by analysis of other online algorithms [27]. The randomized versions of these algorithms were explored because adding randomness can often be shown to theoretically improve the worst-case performance.

The (Weighted) Triangles heuristic is inspired by work showing that social networks have high clustering coefficients. It exploits this by finding triangles completed triangles among the vertices neighbors in a partition and overweighting their importance.

Heuristics Balance Big, Prefer Big, and Avoid Big assume we have a way to differentiate high and low degree nodes. This assumption is based on the fact that most of the graphs we care about have power law degree distribution and, therefore, have high and low degree nodes. These three heuristics propose different treatments for the small number of high degree nodes and the large number of low degree nodes.

Balance Big uses the high degree nodes as seeds for the partitions to 'attract' the low degree nodes. The buffered version, Prefer Big, allows the algorithm more choice in finding these seeds. Avoid Big explores the idea that the high degree nodes form the expander portion of the graph, so perhaps the low degrees nodes can be partitioned after the high degree nodes have been removed.

The final heuristic, Greedy EvoCut, uses EvoCut [9], a local partitioning algorithm, on the buffer. This algorithm has very good theoretical guarantees with regards to the found cuts, and the amount of work spent to find them, but the guarantees do not apply to the way we use it.

3.2 Stream Orders

In a sense, the stream ordering is the key to having a heuristic perform well. A simple example is Chunking, where, if we had an optimal partitioning, and then created an ordering consisting of 'all nodes in partition 1, then all nodes in partition 2' and so on, Chunking would also return an optimal partition. For each heuristic, we can define optimal orderings, but, unfortunately, actually generating them reduces to solving balanced graph partitioning so we must settle for orderings that are easy to compute.

We consider the following three stream orderings:

- *Random* - This is a standard ordering in streaming literature and assumes that the vertices arrive in an order given by a random permutation of the vertices.

- *BFS* - This ordering is generated by selecting a starting node from each connected component of the graph uniformly at random. The ordering is then given by the results of a breadth-first search that starts at the given node. If there are multiple connected components, the component ordering is done at random.

- *DFS* - This ordering is identical to the *BFS* ordering except that depth-first search is used instead of breadth-first search.

Each of these stream orderings has a different justification. The random ordering is a standard assumption when theoretically analyzing streaming algorithms. While we generate these orderings by selecting a random permutation of the vertices, one could view this as a special case of a generic ordering that does not respect connectivity of the graph. The benefit of a random ordering is that it avoids adversarially bad orderings. The downside is that it does not preserve any locality in the edges so we expect it to do poorly for statistical reasons like the Birthday paradox. Via the Birthday paradox, we can argue that for sparse graphs, we expect to go through $O(\sqrt{n})$ of the vertices before we find a first edge.

Both BFS and DFS are natural ways of linearizing graphs and are highly simplified models of a web crawler. In practice, web crawlers are a combination of local search approaches - they follow links, but fully explore domains and sub-domains before moving on. This is breadth-first search between domains, and depth-first search within. The main benefit of both orderings is that they guarantee that the partitioner sees edges in the stream immediately. Additionally, they maintain some locality. Each has their drawbacks, but it should be noted that BFS is a subroutine that is often used in partitioning algorithms to find a good cut, particularly for rounding fractional solutions to LPs [17].

4 Evaluation Setup

We conducted extensive experimental evaluation to discover the performance and trends of stream partitioning heuristics on a variety of graphs. The first question we ask is which of these heuristics are reasonable. Our results highlight a few, but different types of graphs and different stream orderings affect the results. The next question is whether we can recommend a best heuristic, restricted to a particular graph type. The final is whether or not the heuristics scale to larger graphs. Our intent is to use this style of solution for computation systems that include trillions of edges, yet in our initial experiments our largest graph has 1.4 million edges. We address this question by using synthetic datasets to show that the heuristics scale. Additionally, in Section 6, we use our heuristics on two much larger social networks successfully.

4.1 Datasets

We used several sources to collect multiple datasets for our experiments. The primary sources are SNAP [23] and the Graph Partitioning Archive [5]. A few other datasets were also procured and are cited on a by-paper basis. The SNAP graphs used are: soc-Slashdot0811, wiki-Vote and web-NotreDame. The Partitioning Archive graphs used are: 3elt, 4elt, and vibrobox. We also used: Astrophysics collaborations (astro-ph) [29], C. Elegans Neural Network (celegans) [33, 34], and the Marvel Comics social network [7]. We two large social networks (LiveJournal [28] and Twitter [22]) to evaluate our heuristics in a real system in Section 6. We created synthetic datasets using popular generative models, preferential attachment (BA) [14], Watts-Strogatz [33], the RMAT generator [24], and a power-law graph generator with clustering [19]. Three of the synthetic datasets, BA, WS, and PL were created with the NetworkX python package. For each model, we created a degree distribution with average degree $O(\log n)$ (average degree of 10 edges for 1,000 nodes, 13 for 10,000, and 25 for 50,000). This fully specifies the BA model. For WS and PL we used .1 as the rewiring probability. The final datasets were created with the Python Web Graph Generator, a variant of the RMAT generator [2]. The RMAT or Kronecker parameters used by this implementation are [0.45,0.15;0.15,0.25].

These datasets were chosen to balance both size and variety. All are small enough so that we can find offline solutions with METIS, while still big enough to be representative of the asymptotic behavior of these graph types. The collection captures a variety of real graphs, primarily focusing on finite-element meshes and power-law degree distribution graphs. Finite-element meshes (FEM) are used for scientific computing purposes to model simulations like the flow over a wing, while power-law (and other heavy-tailed) distributions capture nearly all 'natural' graphs, like the World Wide Web, social networks, and protein networks. In general, it is known that finite-element meshes have good partitions because their edges are highly local, while natural graphs are more difficult to partition because they have high expansion and low diameter. We give basic statistics about each graph, as well as its type and source in Table 1.

4.2 Methodology

We examined all the combinations of datasets, heuristics and steam orders and ran each experiment 5 times on each combination. The *Random* ordering is a random permutation of the vertices, while *BFS* and *DFS* were created by sampling a random vertex to be the root of the BFS or DFS algorithm. Each of the heuristics was run on the same ordering. We ran each experiment on 2, 4, 8, and 16 partitions and fixed the imbalance

Name	$\|V\|$	$\|E\|$	Type	Source
3elt	4720	13,722	FEM	[5]
4elt	15606	45,878	FEM	[5]
vibrobox	12,328	165,250	FEM	[5]
celegans	297	2,148	Protein	[33, 34]
astro-ph	18,772	396,160	Citation	[29]
Slashdot0811	77,360	504,230	Social	[23]
wiki-Vote	7,115	99,291	Social	[23]
Marvel	6,486	427,018	Social	[7]
web-ND	325,729	1,497,134	Web	[23]
BA	1,000	9,900	Synth.	[14]
BA	10,000	129,831	Synth.	[14]
BA	50,000	1,249,375	Synth.	[14]
RMAT	1,000	9,175	Synth.	[24]
RMAT	10,000	129,015	Synth.	[24]
RMAT	50,000	1,231,907	Synth.	[24]
WS	1,000	5,000	Synth.	[33]
WS	10,000	120,000	Synth.	[33]
WS	50,000	3,400,000	Synth.	[33]
PL	1,000	9,878	Synth.	[19]
PL	10,000	129,763	Synth.	[19]
PL	50,000	1,249,044	Synth.	[19]
LiveJournal	$4.6*10^6$	$77.4*10^6$	Social	[28]
Twitter	$41.7*10^6$	$1.468*10^9$	Social	[22]

Table 1: Graph datasets summary

such that no partition held more than 5% more vertices than its share. The imbalance was chosen as a reasonable setting of this parameter in practice.

5 Evaluation Results

In all of the following figures, the y-axis has been scaled to zoom in on the data. The ordering of the heuristics in the figures is the one given in Table 2.

5.1 Upper and lower bounds

In order to evaluate the quality of our heuristics, we must establish good upper and lower bounds for the performance. A natural upper bound is the approach currently used in practice - hashing the node ID and mapping it to a partition. This approach completely ignores the edges so its expected performance is that it cuts a $\frac{k-1}{k}$ fraction of edges for k partitions. This bound is marked by the upper black line in our figures. We expect Balanced and Hashing to always perform at this level, as well as Chunking on a random order.

The lower bound can be picked in many more ways. For instance, one might solve the relaxed k-partitioning Linear Program for each graph in order to bound the number of edges that must be cut. However, we are focused on realistic approaches so we compare against a practical and fast approach, the partition produced by METIS. While METIS has no theoretical guarantees, it is widely respected and produces quite good cuts in practice, and is thus a good offline comparison for our empirical work. We used METIS v4.0.3 on each of the graphs. This value is marked as the lower black line in our figures. Note that METIS is given significantly more information than the streaming heuristics, so we would not expect them to produce partitioning of the same quality. Any heuristic with partitioning quality between these two lines

is an improvement.

5.2 Performance on three graph types

We have included figures of the results for three of the graphs, a synthetic graph, a social network graph and a finite-element mesh with the goal of covering all three major types of graphs.

Figure 1 depicts the performance on the PowerLaw Clustered graph [19] of size 10,000 with 4 partitions. This is one of our synthetic graphs where the model is intended to capture power law graphs observed in nature. The lower bound provided by METIS is 58.9% of the edges cut, while the upper bound for 4 partitions is 75%. The first heuristic, Avoid Big, is worse than a random cut. Linear Deterministic Greedy and Balance Big both perform very well for all 3 stream orderings. These each had a best average performance of 61.7% and 63.2% of the edges cut respectively, corresponding to 82% and 73% of the possible gain in performance. This gain was calculated as the fraction of edges cut by the heuristic minus the fraction cut by METIS, divided by the fraction cut by a random cut minus the fraction cut by METIS ($\frac{\text{heuristic}-\text{METIS}}{\text{random}-\text{METIS}}$).

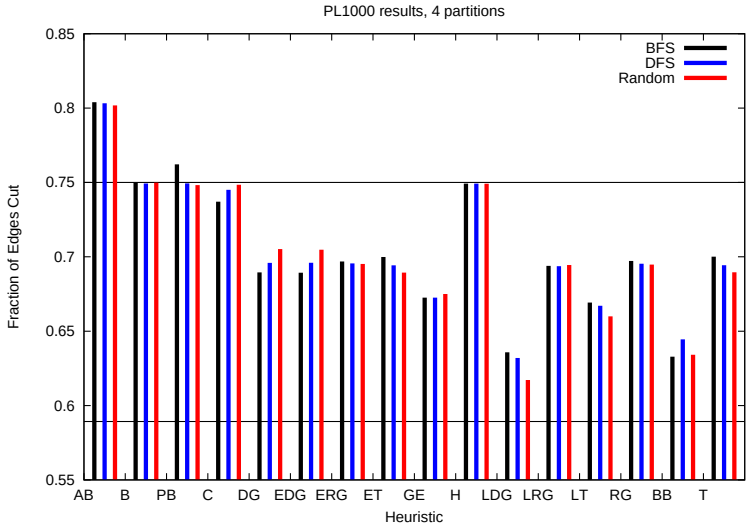

Figure 1: PL1000 results. The top line is the cost of a random cut and the bottom line is METIS. The best heuristic is Linear Deterministic Greedy. The figures are best viewed in color.

Figure 2 is our results for a social network, the Marvel Comics network [7], with 8 partitions. The Marvel network is synthetic, as it is the result of characters interactions in books, but studies have shown it is quite similar to real social networks. This network has a significantly better lower bound from METIS with only 32.2% of edges being cut. The upper bound is $7/8 = 87.5\%$ cut. The two heuristics that perform at the upper bound level are Balanced and Hashing, with Chunking on the random order also performing poorly, as expected. Again, the best heuristic is Linear Deterministic Greedy, with 48%, 48.7% and 50.8% edges cut for the *BFS*, *DFS* and *Random* orderings respectively. This constitutes a gain of 71.3%, 70% and 66% respectively from the random cut towards our lower bound.

Figure 3 contains the results for a finite-element mesh, 4elt [5], with 4 partitions. The change in graph structure gives us quite different results, not the least of which is that the lower bound from METIS now cuts only 0.7% of edges. The random cut upper bound remains at 75%, providing a huge range for improvement. Surprisingly, Chunking performs extremely well here for the *BFS* and *DFS* orders, at 4.7% and 5.7% cut respectively. Translating these numbers into gain provides 94.7% and 93.3% of the optimal improvement. Chunking performs poorly on the *Random* order as expected. The other heuristic that performs well is Greedy EvoCut, obtaining 5.1% and 5% cuts for *BFS* and *DFS* respectively. Linear Deterministic Greedy obtains 9.4%, 20.3%, and 30.6% cuts for *BFS*, *DFS* and *Random* respectively. In fact, all of the heuristics

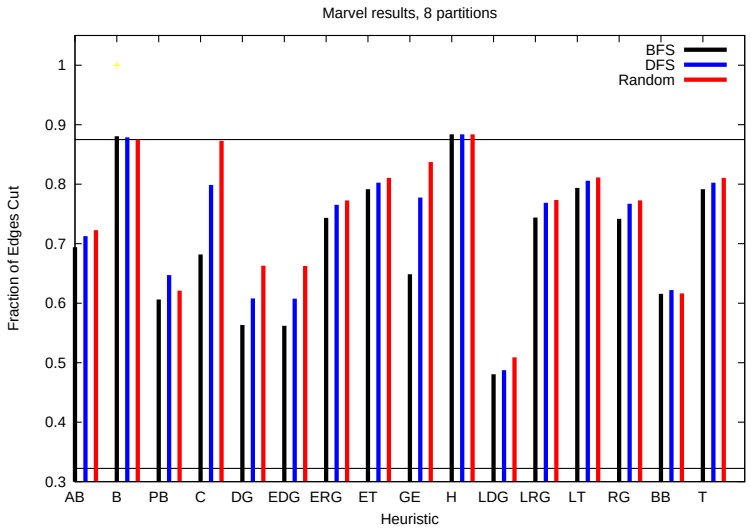

Figure 2: Marvel results. The top line is the cost of a random cut and the bottom line is METIS. The best heuristic is Linear Deterministic Greedy.

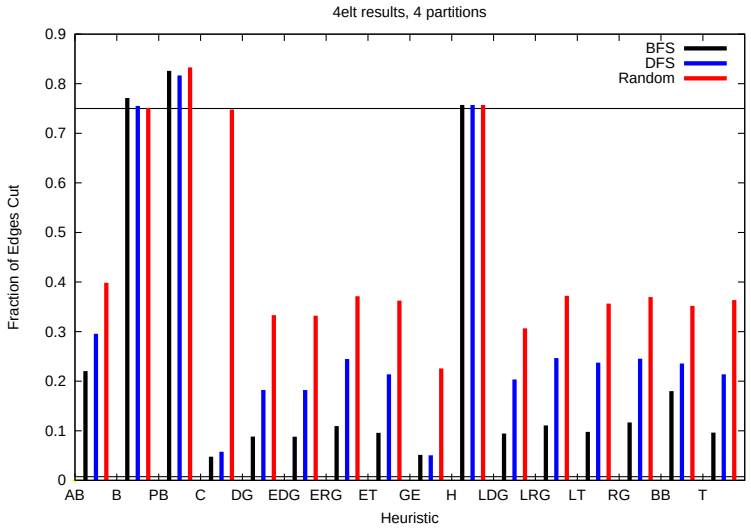

Figure 3: 4elt results. The top line is the cost of a random cut and the bottom line is METIS (0.7% edges cut).

beyond Balanced and Hashing are vast improvements over the current approaches. We also see that the *BFS* ordering is a strict improvement for all approaches over the *DFS* and *Random* orderings.

5.3 Performance on all graphs: summary and discussion

We present the gain in the performance of each heuristic in Table 2, averaged over all datasets from Table 1 (except for LiveJournal and Twitter) and all runs, for each ordering. The best heuristic is Linear Deterministic Greedy for all orderings, following by Balance Big. Greedy EvoCut is also quite successful on the *BFS* and *DFS* orderings, but is computationally much more expensive than the other two approaches as it involves running an iterative evolving set process. Note that Balance Big is a combination of the Greedy and Balanced

Heuristic		*BFS*	*DFS*	*Random*
Avoid Big	AB	-27.3	-38.6	-46.4
Balanced	B	-1.5	-1.3	-0.2
Prefer Big	PB	-9.5	-18.6	-23.1
Chunking	C	37.6	35.7	0.7
Deterministic Greedy	DG	57.7	54.7	45.4
Exp. Det. Greedy	EDG	59.4	56.2	47.5
Exp. Rand. Greedy	ERG	45.6	45.6	38.8
Exp. Triangles	ET	50.7	49.3	41.6
Greedy EvoCut	GE	60.3	58.6	43.1
Hashing	H	-1.9	-2.1	-1.7
Linear Det. Greedy	LDG	76	73	75.3
Linear Rand. Greedy	LRG	46.4	44.9	39.1
Linear Triangles	LT	55.4	54.6	49.3
Randomized Greedy	RG	45.5	44.9	38.7
Balance Big	BB	67.8	68.5	63.3
Triangles	T	49.7	48.4	40.2

Table 2: The average gain of each heuristic over all of our datasets and partitions sizes.

strategies, assigned based on node degree. There are some universally bad heuristics, namely Prefer Big and Avoid Big. Both of these are significantly worse than doing nothing.

We further examine this data by restricting the results by type of graphs. As stated earlier, finite-element meshes have good balanced edge cuts. For these types of graphs, no heuristic performed worse than the Hashing approach, and most did significantly better. For the *BFS* ordering, Linear Deterministic Greedy had an average 86.6% gain, with Deterministic Greedy closely behind at 84.2%. For the *DFS* ordering, the Greedy EvoCut approach performed best at 78.8%, with all 3 deterministic greedy approaches closely behind at 74.9% (exp), 74.8% (unweighted) and 75.8% (linear). Finally, as always, the *Random* ordering was the hardest, but Linear Deterministic Greedy was also the best with 63% improvement. No other method achieved more than 56%. The surprising result for finite-element meshes is how well the Chunking heuristic performed. For the *BFS* order, it averaged an 80% improvement and 72% for the *DFS*. This is a huge improvement for such a simple heuristic, although it is entirely due to the topology of the networks, and the fact that *BFS* is often used in partitioning algorithms to find good cuts. When given a *Random* ordering, Chunking had only a 0.2% average improvement, as expected.

The results for the social networks were much more varied. Here, the Prefer Big and Avoid Big both have large negative improvements, meaning both should never be used for power law degree networks with high expansion. For all three orderings, Linear Deterministic Greedy was clearly the superior approach. For *BFS*, it had an average gain of 71%, while the second best performance was from both Exponential Deterministic Greedy and Deterministic Greedy at 60.5%. For *DFS*, Linear Deterministic Greedy had a 70% improvement, while the other two deterministic greedy approaches had only 52.9%. Finally, for a *Random* ordering, Linear Deterministic Greedy achieved a 64% improvement, with the other greedy approaches at only 42%.

Given that the Linear Deterministic Greedy algorithm performed so well, even compared with the other variants, one may ask why. At a high level, the penalty functions form a continuum. The unweighted version has a very strict cutoff - the penalty only applies when the partition is full and gives no indication that this restriction is approaching. The exponential penalty function has similar performance to the unweighted version. Presumably, this is because while the exponential function does give some indication that the partition is nearly full, it does not do so until it is very close to being full. The linear weighting might be performing better because it provides information about the capacity constraint earlier. Finally, since $1 - x \approx e^{-x}$ when $0 < x < 1$, the linear weighting can be seen as a normalized exponential weighting. This normalization term allows the penalty to take effect much earlier in the process and smooths the information

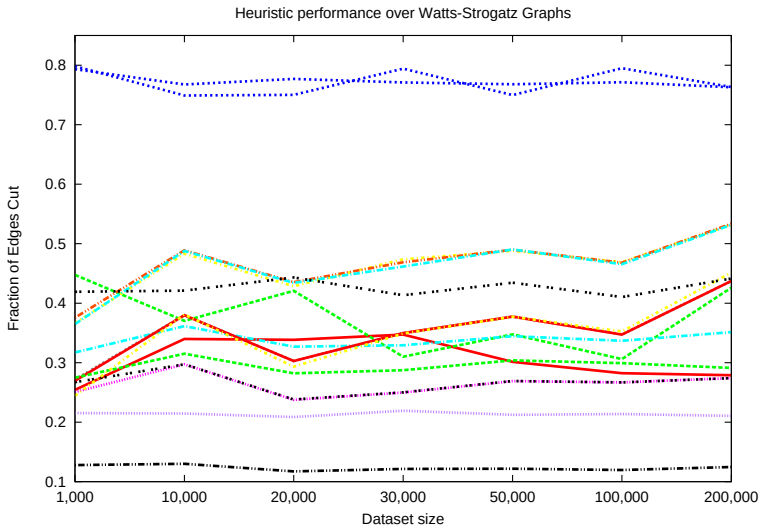

Figure 4: The ordering is fixed to *BFS* and number of partitions to 4. Each line represents a heuristic's performance over 7 sizes of the Watts-Strogatz graphs. The bottom line is METIS. The next bottom purple line is Linear Deterministic Greedy. Figure best viewed in color.

by preventing the size of the partition from affecting the prediction. As this is a continuum, this parameter could be further fine-tuned for different types of graphs.

5.4 Scalability in the graph size

All of our datasets discussed so far are tiny when compared with some of the graphs used in practice. While the above results are promising, it is important to understand whether the heuristics scale with the size of the graph. We used the synthetic datasets in order to control for the variance in different graphs. The key assumption is that using the same generative model with similar parameter settings will guarantee similar graph statistics while allowing the number of edges and nodes to vary. We began by looking at the results for the four generative models, BA, RMAT, WS, and PL. For each of these we had 3 data points: 1,000 vertices, 10,000 vertices, and 50,000 vertices. In order to get a better picture, we created additional graphs with 20,000, 30,000, 100,000 and 200,000 vertices. We will present only the results for the Watts-Strogatz graphs, but all other graphs exhibit quite similar results.

The labels in Figure 4 have been elided for clarity of the image. The bottom black line is METIS. This shows that our idea that the fraction of edges cut should scale with the size of the graph holds - it is approximately 12% for each graph. Next, there is clearly a best heuristic for this type of graph, the purple line. It corresponds to the Linear Deterministic Greedy heuristic. It has an average edge cut of 21% over all sizes of the graphs. Finally, all of the lines are approximately constant. The noise in the performance of each algorithm is due to the random nature of the orderings, and would decrease with further trials.

5.5 Scalability in the number of partitions

The other question is how the partitioning quality scales with the number of partitions. First, note that the fraction of edges cut must necessarily increase as we increase the number of partitions, and also that we are not trying to find an optimal number of partitions for the graph. As before, we only present data on one graph in Figure 5, the 50,000 node PowerLaw Clustered graph, but all graphs have similar characteristics. The performance of the heuristics closely tracks that of METIS.

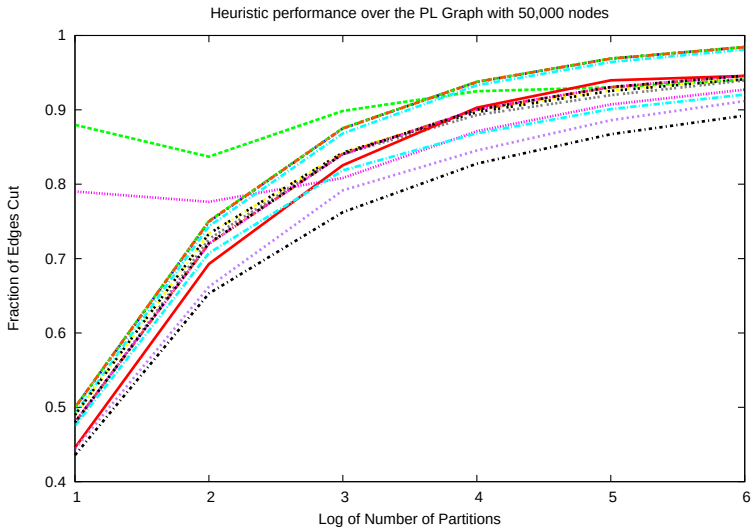

Figure 5: The ordering is fixed to *BFS* and the number of partitions ranges from 2 through 64. Each line represents a heuristic, and connects that heuristics performance over the 6 partition sizes. The bottom line is METIS. The next bottom purple line is Linear Deterministic Greedy. Figure best viewed in color.

6 Results on a Real System

After evaluating the performance of the partitioning algorithm, we naturally ask whether the improvement in the partitioning makes any measurable difference for real computation systems. To evaluate our partitioning scheme in a real cluster application, we used an implementation of PageRank [36] in Spark [37], a cluster computing framework for iterative applications developed at UC Berkeley. Spark provides the ability to keep the working set of the application (i.e., the graph topology and PageRank values) in memory across iterations, so that the algorithm is primarily limited by the cost of communication between nodes. Other recently proposed frameworks for large-scale graph processing, such as Pregel [26] and GraphLab [25], also keep data in memory and are expected to exhibit similar performance characteristics.

There are many graph algorithms implemented for the Spark system, but we chose PageRank for two reasons. One is the popularity of this specific algorithm, and the other is its generality. PageRank is a specialized matrix multiplication, and many graph algorithms can be expressed similarly. Additionally, Spark has two implementations of PageRank: a naïve version that sends a message on the network for each edge, and a more sophisticated combiner version that aggregates all messages between each partition.

We used the Linear Deterministic Greedy heuristic, since it worked best on all of our datasets. For PageRank, the quantity that should be balanced is the number of edges in each partition, not the number of nodes. This is because the number of edges controls the amount of computation performed in sparse matrix multiplication and we want this to be equal for all partitions. We slightly modified Linear Deterministic Greedy to use this quantity for the weight penalty. We used two datasets, one is a crawl of LiveJournal [28] with 4.6 million nodes and 77.4 million edges, and the other is the Twitter graph [22] previously mentioned with 41.7 million nodes and 1.468 billion edges. While neither are Internet scale, they are both large enough to show the effects of reduced communication on a distributed computation.

For LiveJournal, we partitioned the data into 100 pieces, with imbalance of at most 2%. We used the stream order provided by the authors of the dataset which is an unknown ordering. Our Linear Deterministic Greedy streaming partitioning reduced the number of edges cut to 47,361,254 edges compared with 76,234,872 for the Hashing partitioning. We then ran 5 iterations of both versions of PageRank, and repeated this experiment 5 times. With the improved partitioning, all iterations of the naïve PageRank [36] completed in an average of 181.5 seconds, instead of 296.2 seconds with the hashed partitioning. This timing information

	LJ Hash	LJ Streamed
Naïve PR Mean	296.2s	181.5s
Naïve PR STD	5.5 s	2.2 s
Combiner PR Mean	155.1 s	110.4 s
Combiner PR STD	1.5 s	0.8 s
	Twitter Hash	Twitter Streamed
Naïve PR Mean	1199.4 s	969.3 s
Naïve PR STD	81.2 s	16.9 s
Combiner PR Mean	599.4 s	486.8 s
Combiner PR STD	14.4 s	5.9 s

Table 3: Timing data for 5 iterations of PageRank computation on Spark for LiveJournal and Twitter graphs, Hashing vs. Linear Deterministic Greedy streamed partitioning.

used 10 machines and 20 cores on Amazon's EC2. This is a 38.7% reduction in the computation time, just by streaming the data onto the system in a slightly smarter way. When the combiner version was used instead, our streaming partitioning took 110.4 seconds to complete while the hashed version took 155.1 seconds. This is a 28.8% reduction in computation time. These results, along with standard deviations, are summarized in Table 3.

We repeated the same experiment for the Twitter [22] dataset. Twitter is much bigger so we partitioned the graph into 400 pieces with a maximum imbalance of 2%. The Linear Deterministic Greedy streamed partitioning cut 1.341 billion edges, while the Hashing partitioning cut 1.464 billion. We used 50 machines with 100 cores. The total computation time is much longer due to the increase in size. The naïve PageRank took 969.3 seconds on average to complete 5 iterations, while the hashed partitioning took 1199.4 seconds, a 19.1% speedup. The combiner PageRank took 486.8 seconds for our version, and 599.4 seconds for the hashed partitioning, a 18.8% speedup. For both graphs, there was additional time associated with loading the graph, about 200 seconds for Twitter and 80 seconds for LiveJournal, but this was not affected by the partitioning method.

These results show that with very little engineering effort, a simple preprocessing step that considers the graph partitioning can yield a large improvement in the running time. The best heuristic is the greedy approach with linear weighting, which can be computed for each arriving node in time that is linear in the number of edges, given an access to the distributed lookup table for the cluster and knowledge of the current loads of the machines. The improvement in running time is entirely due to the reduced network communication.

7 Conclusions and Future Work

We have demonstrated that simple, one-pass streaming graph partitioning heuristics can drastically improve the number of edges cut in distributed graph data. In particular, our best performing heuristic is the linear weighted variant of the greedy algorithm. This is a simple and effective preprocessing step for large graph computation systems, as the data must be loaded onto the cluster at some point any way. One might still be required to perform a full graph partitioning once the graph has been fully loaded. However, as it will be re-partitioning an already reasonably partitioned graph, there will be less communication cost and it potentially may need to move fewer vertices, and will therefore be faster. Using our streaming approach as preprocessing step can only benefit any future computation while incurring only small cost.

There are several future directions for our work. First, there is the theoretical work. A framework should be developed for proving the performance of these heuristics. The main complications are the addition of the *BFS* and *DFS* stream orderings, and the fact that the offline optimal solution is NP-hard to compute. However, the best heuristic also performed well on the *Random* ordering, so it may be possible to prove a bound on its performance with additional assumptions like the graph is generated by a specific model.

The second direction for future work is to exploit the fact that different heuristics perform well on different graphs. In order to use this in practice, one might build a system based on *empirical hardness models*, e.g. SATzilla [35], and use machine learning techniques to find optimal heuristics based on the features of input.

Another direction is to note that, when modeling communication, hypergraphs are often used instead of graphs. Hypergraph partitioning is somewhat harder than graph partitioning, and it is unclear if similarly simple heuristics exist that could perform well on a streaming hypergraph problem. Additionally, the right model of the stream for hypergraphs should be defined.

A fourth direction is to incorporate the network topology of the cluster. In our experiments on Spark, we generated 400 partitions but ignored the fact that sets of 4 partitions would be assigned to the same machine - one for each core. Similarly, the interconnect network between a set of machines on the same rack is faster than between racks. A simple approach would be to use a recursive approach where the head machine partitions between racks, then a head machine on the rack partitions between the in-rack machines, and finally the machine itself partitions between cores.

A somewhat different problem is what to do when the data changes slightly. It is untenable to reload the whole graph, so one would prefer to be able to incrementally update the system. One approach might be to use spare capacity in the system to run a heuristic like Kernigan-Lin, but another might be to simulate the streaming partitioning based on a stream ordering from the current partition, i.e. all nodes from partition 1, then all nodes from partition 2, etc. Additionally, an incremental partitioning solution would allow the system to update and improve the partitioning in response to the system usage and data access patterns and not just the data changes.

Acknowledgments

The authors would like to acknowledge the many conversations that inspired this work in particular those with Nikhil Devanur, Sameh Elkinety, Sreenivas Gollapudi, Yuxiong He, Nina Mishra, Rina Panigrahy, Yuval Peres, Satish Rao, Burton Smith and David B. Wecker. Technical support was provided by Matei Zaharia, who gave us access to Spark.

References

[1] `http://facebook.com/press/info.php?statistics`, Nov 2011.

[2] `http://pywebgraph.sourceforge.net`.

[3] `http://research.microsoft.com/ldg`, Nov 2011.

[4] `http://research.microsoft.com/trinity`, Nov 2011.

[5] `http://staffweb.cms.gre.ac.uk/~wc06/partition`.

[6] K. Jin Ahn and S. Guha. Graph sparsification in the semi-streaming model. *ICALP*, 2009.

[7] R. Alberich, J. Miro-Julia, and F. Rossello. Marvel universe looks almost like a real social network. *arXiv:cond-mat/0202174*, 2002.

[8] R. Andersen, F. Chung, and K. Lang. Local graph partitioning using pagerank vectors. In *FOCS*, pages 475–486, 2006.

[9] R. Andersen and Y. Peres. Finding sparse cuts locally using evolving sets. In *STOC*, pages 235–244, 2009.

[10] K. Andreev and H. Racke. Balanced graph partitions. *Theory of Computing Systems*, 39:929–939, 2006.

[11] S. Arora, S. Rao, and U. Vazirani. Expander flows, geometric embeddings and graph partitioning. *J. ACM*, 56(2), 2009.

[12] B. Bahmani, A. Chowdhury, and A. Goel. Fast incremental and personalized pagerank. *PVLDB*, 4(3):173–184, 2010.

[13] M. Bailly-Bechet, C. Borgs, A. Braunstein, J. Chayes, A. Dagkessamanskaia, J.-M. Francois, and R. Zecchina. Finding undetected protein associations in cell signaling by belief propagation. *arxiv*, 2011.

[14] A-L Barabasi and R Albert. Emergence of scaling in random networks. *Science*, 286(5439):509–512, 1999.

[15] S. Barnard. PMRSB: Parallel multilevel recursive spectral bisection. In *Supercomputing*, 1995.

[16] S. Bykov, A. Geller, G. Kliot, J. Larus, R. Pandya, and J. Thelin. Orleans: Cloud computing for everyone. In *ACM Symposium on Cloud Computing*, 2011.

[17] G. Even, J. Naor, S. Rao, and B. Schieber. Fast approximate graph partitioning algorithms. *SIAM J. Comput*, 28(6):2187–2214, 1999.

[18] B. Hendrickson and R. Leland. A multilevel algorithm for partitioning graphs. In *Supercomputing*, 1995.

[19] P. Holme and B. J. Kim. Growing scale-free networks with tunable clustering. Phys. Rev. E, 2002.

[20] G. Karypis and V. Kumar. Multilevel graph partitioning schemes. In *ICPP*, pages 113–122, 1995.

[21] J. Kelner and A. Levin. Spectral sparsification in the semi-streaming setting. *STACS*, pages 440–451, 2011.

[22] H. Kwak, C. Lee, H. Park, and S. Moon. What is Twitter, a social network or a news media? In *WWW*, pages 591–600, 2010.

[23] J. Leskovec. `http://snap.stanford.edu/snap`, 2011.

[24] J. Leskovec, D. Chakrabarti, J. Kleinberg, C. Faloutsos, and Z. Ghahramani. Kronecker graphs: An approach to modeling networks. *JMLR*, 2010.

[25] Y. Low, J. Gonzalez, A. Kyrola, D. Bickson, C. Guestrin, and J. Hellerstein. Graphlab: A new framework for parallel machine learning. In *UAI*, 2010.

[26] G. Malewicz, M. Austern, A. Bik, J. Dehnert, I. Horn, N. Leiser, and G. Czajkowski. Pregel: a system for large-scale graph processing. PODC, 2009.

[27] A. Mehta, A. Saberi, U. Vazirani, and V. Vazirani. Adwords and generalized online matching. *J. ACM*, 54(5), 2007.

[28] A. Mislove, M. Marcon, K. Gummadi, P. Druschel, and B. Bhattacharjee. Measurement and analysis of online social networks. In *ACM/USENIX Internet Measurement Conference*, pages 29 – 42, 2007.

[29] M. E. J. Newman. The structure of scientific collaboration networks. *Natl Acad Sci*, 98:404–9, 2001.

[30] S. Rao and U. Vazirani. Lecture notes for cs270 at uc berkeley. 2011.

[31] A. Das Sarma, S. Gollapudi, and R. Panigrahy. Sparse cut projections in graph streams. In *ESA*, pages 480–491, 2009.

[32] A. Das Sarma, S. Gollapudi, and R. Panigrahy. Estimating pagerank on graph streams. *J. ACM*, 58(3):13, 2011.

[33] D. J. Watts and S. H. Strogatz. Collective dynamics of 'small-world' networks. *Nature*, 393:440–442, 1998.

[34] J. G. White, E. Southgate, J. N. Thompson, and S. Brenner. The structure of the nervous system of the nematode caenorhabditis elegans. *Phil. Trans. R. Soc. London*, 314:1–340, 1986.

[35] L. Xu, F. Hutter, H. H. Hoos, and K. Leyton-Brown. Satzilla2009: an automatic algorithm portfolio for sat. 2009.

[36] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. Franklin, S. Shenker, and I. Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. *Technical Report UCB/EECS-2011-82*, July 2011.

[37] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica. Spark: Cluster Computing with Working Sets. In *HotCloud*, 2010.

[38] M. Zelke. Intractability of min- and max-cut in streaming graphs. *Information Processing Letters*, 111(3):145 – 150, 2011.