

Filtering: A Method for Solving Graph Problems in MapReduce

Silvio Lattanzi*
Google, Inc.
New York, NY, USA
silviolat@gmail.com

Benjamin Moseley†
University of Illinois
Urbana, IL, USA
bmosele2@illinois.edu

Siddharth Suri
Yahoo! Research
New York, NY, USA
ssuri@yahoo-inc.com

Sergei Vassilvitskii
Yahoo! Research
New York, NY, USA
sergei@yahoo-inc.com

ABSTRACT

The MapReduce framework is currently the *de facto* standard used throughout both industry and academia for petabyte scale data analysis. As the input to a typical MapReduce computation is large, one of the key requirements of the framework is that the input cannot be stored on a single machine and must be processed in parallel. In this paper we describe a general algorithmic design technique in the MapReduce framework called *filtering*. The main idea behind filtering is to reduce the size of the input in a distributed fashion so that the resulting, much smaller, problem instance can be solved on a single machine. Using this approach we give new algorithms in the MapReduce framework for a variety of fundamental graph problems for sufficiently dense graphs. Specifically, we present algorithms for minimum spanning trees, maximal matchings, approximate weighted matchings, approximate vertex and edge covers and minimum cuts. In all of these cases, we parameterize our algorithms by the amount of memory available on the machines allowing us to show tradeoffs between the memory available and the number of MapReduce rounds. For each setting we will show that even if the machines are only given substantially sublinear memory, our algorithms run in a constant number of MapReduce rounds. To demonstrate the practical viability of our algorithms we implement the maximal matching algorithm that lies at the core of our analysis and show that it achieves a significant speedup over the sequential version.

Categories and Subject Descriptors

F.2.2 [Analysis of Algorithms and Problem Complexity]: Non-numerical Algorithms and Problems

*Work done while visiting Yahoo! Labs.

†Work done while visiting Yahoo! Labs. Partially supported by NSF grants CCF-0728782 and CCF-1016684.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

SPAA'11, June 4–6, 2011, San Jose, California, USA.

Copyright 2011 ACM 978-1-4503-0743-7/11/06 ...\$10.00.

General Terms

Algorithms, Theory

Keywords

MapReduce, Graph Algorithms, Matchings

1. INTRODUCTION

The amount of data available and requiring analysis has grown at an astonishing rate in recent years. For example, Yahoo! processes over 100 billion events, amounting to over 120 terabytes, daily [7]. Similarly, Facebook processes over 80 terabytes of data per day [17]. Although the amount of memory in commercially available servers has also grown at a remarkable pace in the past decade, and now exceeds a once unthinkable amount of 100 GB, it remains woefully inadequate to process such huge amounts of data. To cope with this deluge of information people have (again) turned to parallel algorithms for data processing. In recent years MapReduce [3], and its open source implementation, Hadoop [20], have emerged as the standard platform for large scale distributed computation. About 5 years ago, Google reported that it processes over 3 petabytes of data using MapReduce in one month [3]. Yahoo! and Facebook use Hadoop as their primary method for analyzing massive data sets [7, 17]. Moreover, over 100 companies and 10 universities are using Hadoop [6, 21] for large scale data analysis.

Many different types of data have contributed to this growth. One particularly rich datatype that has captured the interest of both industry and academia is massive graphs. Graphs such as the World Wide Web can easily consist of billions of nodes and trillions of edges [16]. Citation graphs, affiliation graphs, instant messenger graphs, and phone call graphs have recently been studied as part of social network analysis. Although it was previously thought that graphs of this nature are sparse, the work of Leskovec, Kleinberg and Faloutsos [14] dispelled this notion. The authors analyzed the growth over time of 9 different massive graphs from 4 different domains and showed that graphs become denser. Specifically, if $n(t)$ and $e(t)$ denote the number of nodes and edges at time t , respectively, they show that $e(t) \propto n(t)^{1+c}$, where $1 \geq c > 0$. They lowest value of c they find is 0.08, but they observe three graphs with $c > 0.5$. The algorithms we present are efficient for such dense graphs, as well as their sparser counterparts.

Previous approaches to graph algorithms on MapReduce attempt to shoehorn message passing style algorithms into the framework

[9, 15]. These algorithms often require $O(d)$ rounds, where d is the diameter of the input graph, even for such simple tasks as computing connected components, minimum spanning trees, etc. A round in a MapReduce computation can be very expensive time-wise, because it often requires a massive amount of data (on the order of terabytes) to be transmitted from one set of machines to another. This is usually the dominant cost in a MapReduce computation. Therefore minimizing the number of rounds is essential for efficient MapReduce computations. In this work we show how many fundamental graph algorithms can be computed in a constant number of rounds. We use the previously defined model of computation for MapReduce [13] to perform our analysis.

1.1 Contributions

All of our algorithms take the same general approach, which we call *filtering*. They proceed in two stages. First, the algorithms use the parallelization of MapReduce to selectively drop, or *filter*, parts of the input with the goal of reducing the problem size so that the result is small enough to fit into a single machine’s memory. In the second stage the algorithms compute the final answer on this reduced input. The technical challenge is to choose enough edges to drop but still be able to compute either an optimal or provably near optimal solution. The filtering step differs in complexity depending on the problem and takes a few slightly different forms. We exhibit the flexibility of this approach by showing how it can be used to solve a variety of graph problems.

In Section 2.4 we apply the filtering technique to computing the connected components and minimum spanning trees of dense graphs. The algorithm, which is much simpler, and more efficient algorithm than the one that appeared in [13], partitions the original input and solves a subproblem on each partition. The algorithm recurses until the data set is small enough to fit into the memory of a single machine.

In Section 3, we turn to the problem of matchings, and show how to compute a maximal matching in three MapReduce rounds in the model of [13]. The algorithm begins by solving a subproblem on a small sample of the original input. We then use this interim solution to prune out the vast majority of edges of the original input, thus dramatically reducing the size of the remaining problem, and recursing if it is not small enough to fit onto a single machine. The algorithm allows for a tradeoff between the number of rounds and the available memory. Specifically, for graphs with at most n^{1+c} edges and machines with memory at least $n^{1+\epsilon}$ our algorithm will require $O(c/\epsilon)$ rounds. If the machines have memory $O(n)$ then our algorithm requires $O(\log n)$ rounds.

We then use this algorithm as a building block, and show algorithms for computing an 8-approximation for maximum weighted matching, a 2-approximation to Vertex Cover and a $3/2$ -approximation to Edge Cover. For all of these algorithms, the number of machines used will be at most $O(N/\eta)$ where N is the size of the input and η is the memory available on each machine. That is, these algorithms require just enough machines to fit the entire input on all of the machines. Finally, in Section 4 we adapt the seminal work of Karger [10] to the MapReduce setting. Here the filtering succeeds with a limited probability; however, we argue that we can replicate the algorithm enough times in parallel so that one of the runs succeeds without destroying the minimum cut.

1.2 Related Work

The authors of [13] give a formal model of computation of MapReduce called *MRC* which we will briefly summarize in the next section. There are two models of computation that are similar to *MRC*. We describe these models and their relationship to

MRC in turn. We also discuss how known algorithms in those models relate to the algorithms presented in this work.

The algorithms presented in this paper run in a constant number of rounds when the memory per machine is superlinear in the number of vertices ($n^{1+\epsilon}$ for some $\epsilon > 0$). Although this requirement is reminiscent of the semi-streaming model [4], the similarities end there, as the two models are very different. One problem that is hard to solve in semi-streaming but can be solved in *MRC* is graph connectivity. As shown in [4], in the semi-streaming model, without a superlinear amount of memory it is impossible to answer connectivity queries. In *MRC*, however, previous work [13] shows how to answer connectivity queries when the memory per machine is limited to $n^{1-\epsilon}$, albeit at the cost of a logarithmic number of rounds. Conversely, a problem that is trivial in the semi-streaming model but more complicated in *MRC* is finding a maximal matching. In the semi-streaming model one simply streams through the edges, and adds the edge to the current matching if it is feasible. As we show in Section 3, finding a maximal matching in *MRC* is a computable, but non-trivial endeavor. The technical challenge for this algorithm stems from the fact that no single machine can see all of the edges of the input graph, rather the model *requires* the algorithm designer to parallelize the processing¹.

Although parallel algorithms are gaining a resurgence, this is an area that was widely studied previously under different models of parallel computation. The most popular model is the PRAM model, which allows for a polynomial number of processors with shared memory. There are hundreds of papers for solving problems in this model and previous work [5, 13] shows how to simulate certain types of PRAM algorithms in *MRC*. Most of these results yield *MRC* algorithms that require $\Omega(\log n)$ rounds, whereas in this work we focus on algorithms that use $O(1)$ rounds. Nonetheless, to compare with previous work, we next describe PRAM algorithms that either can be simulated in *MRC*, or could be directly implemented in *MRC*. Israel and Itai [8] give an $O(\log n)$ round algorithm for computing maximal matchings on a PRAM. It could be implemented in *MRC*, but would require $O(\log n)$ rounds. Similarly, [19] gives a distributed algorithm which yields constant factor approximation to the weighted matching problem. This algorithm, which could also be implemented in *MRC*, takes $O(\log^2 n)$ rounds. Finally, Karger’s algorithm is in *RNC* but also requires $O(\log^2 n)$ rounds. We show how to implement it in MapReduce in a constant number of rounds in Section 4.

2. PRELIMINARIES

2.1 MapReduce Overview

We remind the reader about the salient features of the MapReduce computing paradigm (see [13] for more details). The input, and all intermediate data, is stored in $\langle \text{key}; \text{value} \rangle$ pairs and the computation proceeds in rounds. Each round is split into three consecutive phases: map, shuffle and reduce. In the map phase the input is processed one tuple at a time. All $\langle \text{key}; \text{value} \rangle$ pairs emitted by the map phase which have the same *key* are then aggregated by the MapReduce system during the shuffle phase and sent to the same machine. Finally each key, along with all the values associated with it, are processed together during the reduce phase.

Since all the values with the same key end up on the same machine, one can view the map phase as a kind of routing step that determines which values end up together. The key acts as a (logi-

¹In practice this requirement stems from the fact that even streaming through a terabyte of data requires a non-trivial amount of time as the machine remains IO bound.

cal) address of the machine, and the system makes sure all of the $\langle \text{key}; \text{value} \rangle$ pairs with the same key are collected on the same machine. To simplify our reasoning about the model, we can combine the reduce and the subsequent map phases. Looking at the computation through this lens, every round each machine performs some computation on the set of $\langle \text{key}; \text{value} \rangle$ pairs assigned to it (reduce phase), and then designates which machine each output value should be sent to in the next round (map phase). The shuffle ensures that the data is moved to the right machine, after which the next round of computation can begin. In this simpler model, we shall only use the term machines as opposed to mappers and reducers.

More formally, let ρ_j denote the reduce function for round j , and let μ_{j+1} denote the map function for the following round of an \mathcal{MRC} algorithm [13] where $j \geq 1$. Now let $\phi_j(x) = \mu_{j+1} \odot \rho_j(x)$. Here ρ_j takes as input some set of $\langle \text{key}; \text{value} \rangle$ pairs denoted by x and outputs another set of $\langle \text{key}; \text{value} \rangle$ pairs. We define the \odot operator to feed the output of $\rho_j(x)$ to μ_{j+1} one $\langle \text{key}; \text{value} \rangle$ pair at a time. Thus ϕ_j denotes the operation of first executing the reducer function, ρ_j , on the set of values in x and then executing the map function, μ_{j+1} , on each $\langle \text{key}; \text{value} \rangle$ pair output by $\rho_j(x)$ individually. This syntactic change allows the algorithm designer to avoid defining mappers and reducers and instead define what each machine does during each round of computation and specify which machine each output $\langle \text{key}; \text{value} \rangle$ pair should go to.

We can now translate the restrictions on ρ_j and μ_j from the \mathcal{MRC} model of [13] to restrictions on ϕ_j . Since we are joining the reduce and the subsequent map phase, we combine the restrictions imposed on both of these computations. There are three sets of restrictions: those on the number of machines, the memory available on each machine and the total number of rounds taken by the computation. For an input of size N , and a sufficiently small $\epsilon > 0$, there are $N^{1-\epsilon}$ machines, each with $N^{1-\epsilon}$ memory available for computation. As a result, the total amount of memory available to the entire system is $O(N^{2-2\epsilon})$. See [13] for a discussion and justification. An algorithm in \mathcal{MRC} belongs to \mathcal{MRC}^i if it runs in worst case $O(\log^i N)$ rounds. Thus, when designing a \mathcal{MRC}^0 algorithm there are three properties that need to be checked:

- **Machine Memory:** In each round the total memory used by a single machine is at most $O(N^{1-\epsilon})$ bits.
- **Total Memory:** The total amount of data shuffled in any round is $O(N^{2-2\epsilon})$ bits².
- **Rounds:** The number of rounds is a constant.

2.2 Total Work and Work Efficiency

Next we define the amount of work done by an \mathcal{MRC} algorithm by taking the standard definition of work efficiency from the PRAM setting and adapting it to the MapReduce setting. Let $w(N)$ denote the amount of work done by an r -round, \mathcal{MRC} algorithm on an input of size N . This is simply the sum of the amount of work done during each round of computation. The amount of work done during round i of a computation is the product of the number of machines used in that round, denoted $p_i(N)$, and the worst case running time of each machine, denoted $t_i(N)$. More specifically,

$$w(N) = \sum_{i=1}^r w_i(N) = \sum_{i=1}^r p_i(N) t_i(N). \quad (1)$$

²In other words, the total amount of data shuffled in any round must be less than the total amount of memory in the system.

Algorithm: MST(V,E)

```

1: if  $|E| < \eta$  then
2:   Compute  $T^* = MST(E)$ 
3:   return  $T^*$ 
4: end if
5:  $\ell \leftarrow \Theta(|E|/\eta)$ 
6: Partition  $E$  into  $E_1, E_2, \dots, E_\ell$  where  $|E_i| < \eta$  using a
   universal hash function  $h: E \rightarrow \{1, 2, \dots, \ell\}$ .
7: In parallel: Compute  $T_i$ , the minimum spanning tree on
    $G(V, E_i)$ .
8: return  $MST(V, \cup_i T_i)$ 

```

Figure 1: Minimum spanning tree algorithm

If the amount of work done by an \mathcal{MRC} algorithm matches the running time of the best known sequential algorithm, we say the \mathcal{MRC} algorithm is *work efficient*.

2.3 Notation

Let $G = (V, E)$ be an undirected graph, and denote by $n = |V|$ and $m = |E|$. We will call G , c -dense, if $m = n^{1+c}$ where $0 < c \leq 1$. In what follows we assume that the machines have some limited memory η . We will assume that the number of available machines is $O(m/\eta)$. Notice that the number of machines is just the number required to fit the input on all of the machines simultaneously. All of our algorithms will consider the case where $\eta = n^{1+\epsilon}$ for some $\epsilon > 0$. For a constant ϵ , the algorithms we define will take a constant number of rounds and lie in \mathcal{MRC}^0 [13], beating the $\Omega(\log n)$ running time provided by the PRAM simulation constructions (see Theorem 7.1 in [13]). However, even when $\eta = O(n)$ our algorithms will run in $O(\log n)$ rounds. This exposes the memory vs. rounds tradeoff since most of the algorithms presented take fewer rounds as the memory per machine increases. We now proceed to describe the individual algorithms, in order of progressively more complex filtering techniques.

2.4 Warm Up: Connected Components and Minimum Spanning Trees

We present the formal algorithm for computing minimum spanning trees (the connected components algorithm is identical). The algorithm works by partitioning the edges of the input graph into subsets of size η and sending each subgraph to its own machine. Then, each machine throws out any edge that is guaranteed not to be a part of any MST because it is the heaviest edge on some cycle in that machine's subgraph. If the resulting graph fits into memory of a single machine, the algorithm terminates. Otherwise, the algorithm recurses on the smaller instance. We give the pseudocode in Figure 1.

We assume the algorithm is given a c -dense graph; each machine has memory $\eta = O(n^{1+\epsilon})$, and that the number of machines $\ell = \Theta(n^{c-\epsilon})$. Thus the algorithm only uses enough memory, across the entire system, to store the input. We show that every iteration reduces the input size by $n^{c/\epsilon}$, and thus after $\lceil c/\epsilon \rceil$ iterations the algorithm terminates.

LEMMA 2.1. *Algorithm MST(V,E) terminates after $\lceil c/\epsilon \rceil$ iterations and returns the Minimum Spanning Tree.*

PROOF. To show correctness, note that any edge that is not part of the MST on a subgraph of G is also not part of the MST of G by the cycle property of minimum spanning trees.

It remains to show that (1) the memory constraints of each machine are never violated and (2) the total number of rounds is lim-

ited. Since the partition is done randomly, an easy Chernoff argument shows that no machine gets assigned more than η edges with high probability. Finally, note that $|\bigcup_i T_i| \leq \ell(n-1) = O(n^{1+c-\epsilon})$. Therefore after $\lceil c/\epsilon \rceil - 1$ iterations the input is small enough to fit onto a single machine, and the overall algorithm terminates after $\lceil c/\epsilon \rceil$ rounds. \square

LEMMA 2.2. *The $MST(V, E)$ algorithm does $O(\frac{cm}{\epsilon}\alpha(m, n))$ total work.*

PROOF. During a specific iteration, randomly partitioning E into E_1, E_2, \dots, E_ℓ requires a linear scan over the edges which is $O(m)$ work. Computing the minimum spanning tree M_i of each part of the partition using the algorithm of [2] takes $O(\ell \frac{m}{\epsilon} \alpha(m, n))$ work. Computing the MST of G_{sparse} on one machine using the same algorithm requires $\ell(n-1)\alpha(m, n) = O(m\alpha(m, n))$ work. \square

For constant ϵ the MRC algorithm uses $O(m\alpha(m, n))$ work. Since the best known sequential algorithm [11] runs in time $O(m)$ in expectation, the MRC algorithm is work efficient up to a factor of $\alpha(m, n)$.

3. MATCHINGS AND COVERS

The maximum matching problem and its variants play a central role in theoretical computer science, so it is natural to determine if is possible to efficiently compute a maximum matching, or, more simply, a maximal matching, in the MapReduce framework. The question is not trivial. Indeed, due to the constraints of the model, it is not possible to store (or even stream through) all of the edges of a graph on a single machine. Furthermore, it is easy to come up with examples where the partitioning technique similar to that used for MSTs (Section 2.4) yields an arbitrarily bad matching. Simply sampling the edges uniformly, or even using one of the sparsification approaches [18] appears unfruitful because good sparsifiers do not necessarily preserve maximal matchings.

Despite these difficulties, we are able to show that by combining a simple sampling technique and a post-processing strategy it is possible to compute an unweighted maximal matching and thus a 2-approximation to the unweighted maximum matching problem using only machines with memory of size $O(n)$ and $O(\log n)$ rounds. More generally, we show that we can find a maximal matching on c -dense graphs in $O(c/\epsilon)$ rounds using machines with $\Omega(n^{1+\epsilon})$ memory; only three rounds are necessary if $\epsilon = 2c/3$. We extend this technique to obtain an 8-approximation algorithm for maximum weighted matching and use similar approaches to approximate the vertex and edge cover problems. This section is organized as follows: first we present the algorithm to solve the unweighted maximal matching, and then we explain how to use this algorithm to solve the weighted maximum matching problem. Finally, we show how the techniques can be adapted to solve the minimum vertex and the minimum edge cover problems.

3.1 Unweighted Maximal Matchings

The algorithm works by first sampling $O(\eta)$ edges and finding a maximal matching M_1 on the resulting subgraph. Given this matching, we can now safely remove edges that are in conflict (i.e. those incident on nodes in M_1) from the original graph G . If the resulting filtered graph, H is small enough to fit onto a single machine, the algorithm augments M_1 with a matching found on H . Otherwise, we augment M_1 with the matching found by recursing on H . Note that since the size of the graph reduces from round to round, the effective sampling probability increases, resulting in a larger sample of the remaining graph.

Formally, let $G(V, E)$ be a simple graph where $n = |V|$ and $|E| \leq n^{1+c}$ for some $c > 0$. We begin by assuming that each of the machines has at least η memory. We fix the exact value of η later, but require that $\eta \geq 40n$. We give the pseudocode for the algorithm below:

1. Set $M = \emptyset$ and $S = E$.
2. Sample every edge $(u, v) \in S$ uniformly at random with probability $p = \frac{\eta}{10|S|}$. Let E' be the set of sampled edges.
3. If $|E'| > \eta$ the algorithm fails. Otherwise give the graph $G(V, E')$ as input to a single machine and compute a maximal matching M' on it. Set $M = M \cup M'$.
4. Let I be the set of unmatched vertices in G . Compute the subgraph of G induced by I , $G[I]$, and let $E[I]$ be the set of edges in $G[I]$. If $|E_i| > \eta$, set $S = E[I]$ and return to step 2. Otherwise continue to step 5.
5. Compute a maximal matching M'' on $G[I]$ and output $M = M \cup M''$.

To proceed we need the following technical lemma, which shows that with high probability every induced subgraph with sufficiently many edges, has at least one edge in the sample.

LEMMA 3.1. *Let $E' \subseteq E$ be a set of edges chosen independently with probability p . Then with probability at least $1 - e^{-n}$, for all $I \subseteq V$ either $|E[I]| < 2n/p$ or $E[I] \cap E' \neq \emptyset$.*

PROOF. Fix one such subgraph, $G[I] = (I, E[I])$ with $|E[I]| \geq 2n/p$. The probability that none of the edges in $E[I]$ were chosen to be in E' is $(1-p)^{|E[I]|} \leq (1-p)^{2n/p} \leq e^{-2n}$. Since there are at most 2^n total possible induced subgraphs $G[I]$, the probability that there exists one that does not have an edge in E' is at most $2^n e^{-2n} \leq e^{-n}$. \square

Next we bound the number of iterations the algorithm takes. Note that, the term iteration refers to the number of times the algorithm is repeated. This does not refer to a MapReduce round.

LEMMA 3.2. *If $\eta \geq 40n$ then the algorithm runs for at most $O(\log n)$ iterations with high probability. Furthermore, if $\eta = n^{1+\epsilon}$, where $0 < \epsilon < c$ is a fixed constant, then the algorithm runs in at most $\lceil c/\epsilon \rceil$ iterations with high probability.*

PROOF. Fix an iteration i of the algorithm and let p be the sampling probability for this iteration. Let E_i be the set of edges at the beginning of this iteration, and denote by I be the set of unmatched vertices after this iteration. From Lemma 3.1, if $|E[I]| \geq 2n/p$ then an edge of $E[I]$ will be sampled with high probability. Note that no edge in $E[I]$ is incident on any edge in M' . Thus, if an edge from $E[I]$ is sampled then our algorithm would have chosen this edge to be in the matching. This contradicts the fact that no vertex in I is matched. Hence, $|E[I]| \leq 2n/p \leq \frac{20n|E_i|}{\eta}$ with high probability.

Now consider the first iteration of the algorithm, let $G_1(V_1, E_1)$ be the induced graph on the unmatched nodes after the first step of the algorithm. The above argument implies that $|E_1| \leq \frac{20n|E_0|}{\eta} \leq \frac{20n|E|}{\eta} \leq \frac{|E|}{2}$. Similarly $|E_2| \leq \frac{20n|E_1|}{\eta} \leq \frac{(20n)^2|E_0|}{\eta^2} \leq \frac{|E|}{2^2}$. So after i iterations $|E_i| \leq \frac{|E|}{2^i}$. The first part of the claim follows.

To conclude the proof note that if $\eta = n^{1+\epsilon}$, we have that $|E_i| \leq \frac{|E|}{n^{\epsilon i}}$, and thus the algorithm terminates after $\lceil c/\epsilon \rceil$ iterations. \square

We continue by showing the correctness of the algorithm.

THEOREM 3.1. *The algorithm finds a maximal matching of $G = (V, E)$ with high probability.*

PROOF. First consider the case that the algorithm does not fail. Assume, for the sake of contradiction, that there exists an edge $(u, v) \in E$ such that neither u nor v are matched in the final matching M that is output. Consider the last iteration of the algorithm. Since $(u, v) \in E$ and u and v are not matched, $(u, v) \in E[I]$. Since this is the last run of the algorithm, a maximal matching M'' of $G[I]$ is computed on one machine. Since M'' is maximal, either u or v or both must be matched in it. All of the edges of M'' get added to M in the last step, which gives our contradiction.

Next, consider the case that the algorithm failed. This occurs due to the set of edges E' having size larger than η in some iteration of the algorithm. Note that $\mathbf{E}[|E'|] = |\mathcal{S}| \cdot p = \eta/10$ in a given iteration. By the Chernoff Bound it follows that $|E'| \geq \eta$ with probability smaller than $2^{-\eta} \leq 2^{-40n}$ (since $\eta \geq 40n$). By Lemma 3.2 the algorithm completes in at most $O(\log n)$ rounds, thus the total failure probability is bounded by $O(\log n 2^{-40n})$ using the union bound. \square

Finally we show how to implement this algorithm in MapReduce.

COROLLARY 3.1. *The Maximal Matching algorithm can be implemented in three MapReduce rounds when $\eta = n^{1+2c/3}$. Furthermore, when $\eta = n^{1+\epsilon}$ then the algorithm runs for $3\lceil c/\epsilon \rceil$ rounds and $O(\log n)$ rounds when $\eta \geq 40n$.*

PROOF. By Lemma 3.2 the algorithm runs for one iteration with high probability when $\eta = n^{1+2c/3}$, $\lceil c/\epsilon \rceil$ iterations when $\eta = n^{1+\epsilon}$. Therefore it only remains to describe how to compute the graph $G[I]$. For this we appeal to Lemma 6.1 in [13], where the set S_i are the edges incident on node i , and the function f_i drops the edge i if it is matched and keeps it otherwise. Hence, each iteration of the algorithm requires 3 MapReduce rounds. \square

LEMMA 3.3. *The maximal matching algorithm presented above is work efficient when $\eta = n^{1+\epsilon}$ where $0 < \epsilon < c$ is a fixed constant.*

PROOF. By Lemma 3.2 when $\eta = n^{1+\epsilon}$ there are at most a constant number of iterations of the algorithm. Thus it suffices to show that $O(m)$ work is done in a single iteration. Sampling each edge with probability p requires a linear scan over the edges, which is $O(m)$ work. Computing a maximal matching on one machine can be done using a straightforward, greedy semi-streaming algorithm requiring $|E'| \leq \eta \leq m$ work. Computing $G[I]$ can be done as follows. Load M' onto m^ϵ machines where $0 < \epsilon < c$ and partition E among those machines. Then, if an edge in E is incident on an edge in M' the machines drop that edge, otherwise that edge is in $G[I]$. This results in $O(m)$ work to load all of the data onto the machines and $O(m)$ work to compute $G[I]$. Since $G[I]$ has at most m edges, computing M'' on one machine using the best known greedy semi-streaming algorithm also requires $O(m)$ work. \square

Since the vertices in a maximal matching provide a 2-approximation to the vertex cover problem, we get the following corollary.

COROLLARY 3.2. *A 2-approximation to the optimal vertex cover can be computed in three MapReduce rounds when $\eta = n^{1+2c/3}$. Further, when $\eta = n^{1+\epsilon}$ then the algorithm runs for $3\lceil c/\epsilon \rceil$ rounds and $O(\log n)$ rounds when $\eta \geq 40n$. This algorithm does $O(m)$ total work when $\eta = n^{1+\epsilon}$ for a constant $\epsilon > 0$.*

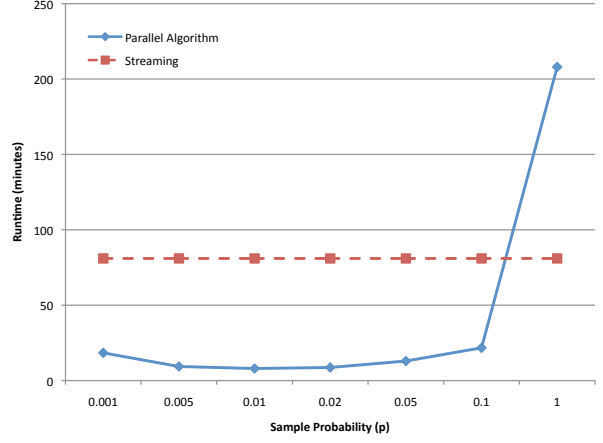


Figure 2: The running time of the MapReduce matching algorithm for different values of p , as well as the baseline provided by the streaming implementation.

3.1.1 Experimental Validation

In this Section we experimentally validate the above algorithm and demonstrate that it leads to significant runtime improvements in practice. Our data set consists of a sample of a graph of the twitter follower network, previously used in [1]. The graph has 50,767,223 nodes, 2,669,502,959 edges, and takes about 44GB when stored on disk. We implemented the greedy streaming algorithm for maximum matching as well as the three phase MapReduce algorithm described above. The streaming algorithm remains I/O bounded and completes in 81 minutes. The total running times for the MapReduce algorithm with different values for the sampling probability p are given in Figure 2.

The MapReduce algorithm achieves a significant speedup (over 10x) over a large number of values for p . The speed up is the result of the fact that a single machine never scans the whole input. Both the sampling in stage 1 and the filtering in stage 2 are performed in parallel. Note that the parallelization does not come for free, and the MapReduce system has non-trivial overhead over the straightforward streaming implementation. For example, when $p = 1$, the MapReduce algorithm essentially implements the streaming algorithm (since all of the edges are mapped onto a single machine), however the running time is almost 2.5 times slower. Overall these results show that the algorithms proposed are not only interesting from a theoretical viewpoint, but are viable and useful in practice.

3.2 Maximum Weighted Matching

We present an algorithm that computes an approximation to the maximum weighted matching problem using a constant number of MapReduce rounds. Our algorithm takes advantage of both the sequential and parallel power of MapReduce. Indeed, it will compute several matchings in parallel and then combine them on a single machine to compute the final result. We assume that the maximum weight on an edge is polynomial in $|E|$ and we prove an 8-approximation algorithm. Our analysis is motivated by the work of Feigenbaum et al. [4], but is technically different since no single machines sees all of the edges.

The input of the algorithm is a simple graph $G(V, E)$ and a weight function $w : E \rightarrow \mathbb{R}$. We assume that $|V| = n$ and $|E| = n^{1+c}$ for a constant $c > 0$. Without loss of generality, assume that $\min\{w(e) : e \in E\} = 1$ and $W = \max\{w(e) : e \in E\}$. The algorithm works as follows:

1. Split the graph G into $G_1, G_2, \dots, G_{\lceil \log W \rceil}$, where G_i is the graph on the set V of vertices and contains edges with weights in $(2^{i-1}, 2^i]$.
2. For $1 \leq i \leq \lceil \log W \rceil$ run the maximal matching algorithm on G_i . Let M_i be the maximal matching for the graph G_i .
3. Set $M = \emptyset$. Consider the edge sets sequentially, in descending order, $M_{\lceil \log W \rceil}, \dots, M_2, M_1$. When considering an edge $e \in M_i$, we add it to the matching if and only if $M \cup \{e\}$ is a valid matching. After all edges are considered, output M .

LEMMA 3.4. *The above algorithm outputs an 8-approximation to the weighted maximum matching problem.*

PROOF. Let OPT be the maximum weighted matching in G and denote by $V(M_i)$ the set of vertices incident on the edges in M_i . Consider an edge $(u, v) = e \in \text{OPT}$, such that $e \in G_j$ for some j . Let i^* be the maximum i such that $\{u, v\} \cap V(M_{i^*}) \neq \emptyset$. Note that i^* must exist, and $i^* \geq j$ else we could have added e to M_j . Therefore, $w(e) \leq 2^{i^*}$.

Now for every such edge $(u, v) \in \text{OPT}$ we select one vertex from $\{u, v\} \cap V(M_{i^*})$. Without loss of generality, let that v be the selected vertex. We say that v is a *blocking vertex* for e . For each blocking vertex v , we associate its incident edge in M_{i^*} and call it the *blocking edge* for e . Let $V_b(i)$ be the set of blocking vertices in $V(M_i)$, we have that

$$\sum_{i=1}^{\lceil \log W \rceil} 2^i |V_b(i)| \geq \sum_{e \in \text{OPT}} w(e).$$

This follows from the fact that every vertex can “block” at most one $e \in \text{OPT}$ and that OPT is a valid matching. Note also that from the definition of *blocking vertex* if $(u, v) \in M \cap M_j$ then $u, v \notin \bigcup_{k < j} V_b(k)$.

Now suppose that an edge $(x, y) \in M_k$ is discarded by step 3 of the algorithm. This can happen if and only if there is an edge already present in the matching with a higher weight adjacent to x or y . Formally, there is a $(u, v) \in M$, $(u, v) \in M_j$ with $j > k$ and $\{u, v\} \cap \{x, y\} \neq \emptyset$. Without loss of generality assume that $\{u, v\} \cap \{x, y\} = x$ and consider such an edge (x, v) . We say that (x, v) killed the edge (x, y) and the vertex y . Notice that an edge $(u, v) \in M$ and $(u, v) \in M_j$ kills at most two edges for every M_k with $k < j$ and kills at most two nodes in $V_b(k)$. Finally we also define $(u, v)_b$ as the set of *blocking vertices* associated with the *blocking edge* (u, v) .

Now consider $V_b(k)$, each blocking vertex was either killed by one of the edges in the matching M , or is adjacent to one of the edges in M_k . Furthermore, the total weight of the edges in OPT with that were blocked by a *blocking vertex* killed by (u, v) is at most

$$\sum_{i=1}^{j-1} 2^i |\{V_b(i) \text{ killed by } (u, v)\}| \leq \sum_{i=1}^{j-1} 2^{i+1} \leq 2^{j+1} \leq 4w((u, v)). \quad (2)$$

To conclude, note that each edge in OPT that is not in M was either blocked directly by an edge in M , or was blocked by a vertex that was killed by an edge in M . To bound the former, consider an edge $(u, v) \in M_j \cap M$. Note that this edge can be incident on at most 2 edges in OPT , each of weight $2^j \leq 2w((u, v))$, and thus the weight in OPT incident on an edge (u, v) is $4w((u, v))$.

Putting this together with Equation 2 we conclude:

$$8 \sum_{(u, v) \in M} w((u, v)) \geq \sum_{e \in \text{OPT}} w(e).$$

□

Furthermore we can show that the analysis of our algorithm is essentially tight. Indeed there exists a family of graphs where our algorithm finds a solution with weight $\frac{w(\text{OPT})}{8-o(1)}$ with high probability. We prove the following lemma in Appendix A.

LEMMA 3.5. *There is a graph where our algorithm computes a solution that has value $\frac{w(\text{OPT})}{8-o(1)}$ with high probability.*

Finally, suppose that the weight function $w : E \rightarrow \mathbb{R}$ is such that $\forall e \in E, w(e) \in O(\text{poly}(|E|))$ and that each machine has memory at least $\eta \geq \max\{2n \log^2 n, |V| \lceil \log^2 W \rceil\}$. Then we can run the above algorithm in MapReduce using only one more round than the maximal matching algorithm. In the first round we split G into $G_1, \dots, G_{\lceil \log W \rceil}$; then we run the maximal matching algorithm of the previous subsection in parallel on $\lceil \log W \rceil$ machines. In the last round, we run the last step on a single machine. The last step is always possible because we have at most $|V| \lceil \log W \rceil$ edges each with weights of size $\log W$.

THEOREM 3.2. *There is an algorithm that finds a 8-approximation to the maximum weighted matching problem on a c dense graph using machines with memory $\eta = n^{1+\epsilon}$ in $3\lfloor c/\epsilon \rfloor + 1$ rounds with high probability.*

COROLLARY 3.3. *There is an algorithm that, with high probability, finds a 8-approximation to the maximum weighted matching problem that runs in four MapReduce rounds when $\eta = n^{1+2/3c}$.*

To conclude the analysis of the algorithm we now study the work amount of the maximum matching algorithm.

LEMMA 3.6. *The amount of work performed by the maximum matching algorithm presented above is $O(m)$ when $\eta = n^{1+\epsilon}$ where $0 < \epsilon < c$ is a fixed constant.*

PROOF. The first step of the algorithm requires $O(m)$ work as it can be done using a linear scan over the edges. In the second step, by Lemma 3.3 each machine performs work that is linear in the number of edges that are assigned to the machine. Since the edges are partitioned across the machines, the total work done in the second step is $O(m)$. Finally we can perform the third step by a semi-streaming algorithm that greedily adds edges in the order $M_{\lceil \log W \rceil}, \dots, M_2, M_1$, requiring $O(m)$ work. □

3.3 Minimum Edge Cover

Next we turn to the minimum edge cover problem. An edge cover of a graph $G(V, E)$ is a set of edges $E^* \subseteq E$ such that each vertex of V has at least one endpoint in E^* . The minimum edge cover is an edge cover E^* of minimum size.

Let $G(V, E)$ be a simple graph. The algorithm to compute a edge cover is as follows:

1. Find a maximal matching M of G using the procedure described in Section 3.1.
2. Let I be the set of uncovered vertices. For each uncovered vertex, take any edge incident on the vertex in I . Let this set of edges be U .
3. Output $E^* = M \cup U$.

Note that this procedure produces a feasible edge cover E^* . To bound the size of E^* let OPT denote the size of the minimum

edge cover for the graph G and let OPT_m denote the size of the maximum matching in G . It is known that the minimum edge cover of a graph is equal to $|V| - \text{OPT}_m$. We also know that $|U| = |V| - 2|M|$. Therefore, $|E^*| = |V| - |M| \leq |V| - \frac{1}{2}\text{OPT}_m$ since a maximal matching has size at least $\frac{1}{2}\text{OPT}_m$. Knowing that $\text{OPT}_m \leq |V|/2$ and using Corollary 3.1 to bound the number of rounds we have the following theorem.

THEOREM 3.3. *There is an algorithm that, with high probability, finds a $\frac{3}{2}$ -approximation to the minimum edge cover in MapReduce. If each machine has memory $\eta \geq 40n$ then the algorithm runs in $O(\log n)$ rounds. Further, if $\eta = n^{1+\epsilon}$, where $0 < \epsilon < c$ is a fixed constant, then the algorithm runs in $3\lceil c/\epsilon \rceil + 1$ rounds.*

COROLLARY 3.4. *There is an algorithm that, with high probability, finds a $\frac{3}{2}$ -approximation to the minimum edge cover in four MapReduce rounds when $\eta = n^{1+2/3c}$.*

Now we prove that the amount of work performed by the edge cover algorithm is $O(m)$.

LEMMA 3.7. *The amount of work performed by the edge cover algorithm presented above is $O(m)$ when $\eta = n^{1+\epsilon}$ where $0 < \epsilon < c$ is a fixed constant.*

PROOF. By Lemma 3.3 when $\eta = n^{1+\epsilon}$ the first step of the algorithm can be done performing $O(m)$ operations. The second step can be performed by a semi streaming algorithm that requires $O(m)$ work. Thus the claim follows. \square

4. MINIMUM CUT

Whereas in the previous algorithms the filtering was done by dropping certain edges, this algorithm filters by contracting edges. Contracting an edge, will obviously reduce the number of edges and may either keep the number of vertices the same (in the case we contracted a self loop), or reduce it by one. To compute the minimum cut of a graph we appeal to the contraction algorithm introduced by Karger [10]. The algorithm has a well known property that the random choices made in the early rounds succeed with high probability, whereas those made in the later rounds have a much lower probability of success. We exploit this property by showing how to filter the input in the first phase (by contracting edges) so that the remaining graph is guaranteed to be small enough to fit onto a single machine, yet large enough to ensure that the failure probability remains bounded. Once the filtering phase is complete, and the problem instance is small enough to fit onto a single machine, we can employ any one of the well known methods to find the minimum cut in the filtered graph. We then decrease the failure probability by running several executions of the algorithm in parallel, thus ensuring that in one of the copies the minimum cut survives this filtering phase.

The complicating factor in the scheme above is contracting the right number of edges so that the properties above hold. We proceed by labeling each edge with a random number between 0 and 1 and then searching for a threshold t so that contracting all of the edges with label less than t results in the desired number of vertices. Typically such a search would take logarithmic time, however, by doing the search in parallel across a large number of machines, we can reduce the depth of the recursion tree to be constant. Moreover, to compute the number of vertices remaining after the first t edges are contracted, we refer to the connected components algorithm in Section 2.4. Since the connected components algorithm uses a small number of machines, we can show that even with many parallel invocations we will not violate the machine budget. We present the algorithm and its analysis below. Also, the algorithm uses two subroutines, Find_t and Contract which are defined in turn.

Algorithm 1 $\text{MinCut}(E)$

```

1: for  $i = 1$  to  $n^{\delta_1}$  (in parallel) do
2:   tag  $e \in E$  with a number  $r_e$  chosen uniformly at random from  $[0, 1]$ 
3:    $t \leftarrow \text{Find}_t(E, 0, 1)$ 
4:    $E_i \leftarrow \text{Contract}(E, t)$ 
5:    $C_i \leftarrow \text{min cut of } E_i$ 
6: end for
7: return minimum cut over all  $C_i$ 

```

4.1 Find Algorithm

The pseudocode for the algorithm to find the correct threshold is given below. The algorithm performs a parallel search on the value t so that contracting all edges with weight at most t results in a graph with n^{δ_3} vertices. The algorithm invokes n^{δ_2} copies of the connected components algorithm, each of which uses at most $n^{c-\epsilon}$ machines, with $n^{1+\epsilon}$ memory.

Algorithm 2 $\text{Find}_t(E, \min, \max)$

```

1: {Uses  $n^{\delta_2+c/\epsilon}$  machines.}
2:  $\gamma \leftarrow \frac{\max - \min}{n^{\delta_2}}$ 
3: for  $j = 1$  to  $n^{\delta_2}$  (in parallel) do
4:    $\tau_j \leftarrow \min + j\gamma$ 
5:    $E_j \leftarrow \{e \in E \mid r_e \leq \tau_j\}$ 
6:    $cc_j \leftarrow \text{number of connected components in } G = (V, E_j)$ 
7: end for
8: if there exists a  $j$  such that  $cc_j = n^{\delta_3}$  then
9:   return  $j$ 
10: else
11:   return  $\text{Find}_t(E, \tau_j, \tau_{j+1})$  where  $j$  is the smallest value s.t.  $cc_j < n^{\delta_3}, cc_{j+1} > n^{\delta_3}$ 
12: end if

```

4.2 Contraction Algorithm

We state the contraction algorithm and prove bounds on its performance.

Algorithm 3 $\text{Contract}(E, t)$

```

1:  $CC \leftarrow \text{connected components in } \{e \in E \mid r_e \leq t\}$ 
2: let  $h : [n] \rightarrow [n^{\delta_4}]$  be a universal hash function
3: map each edge  $(u, v)$  to machine  $h(u)$  and  $h(v)$ 
4: map the assignment of node  $u$  to its connected component  $CC(u)$ , to machine  $h(u)$ 
5: on each reducer rename all instances of  $u$  to  $CC(u)$ 
6: map each edge  $(u, v)$  to machine  $h(u) + h(v)$ 
7: Drop self loops (edges in same connected component)
8: Aggregate parallel edges

```

LEMMA 4.1. *The Contract algorithm uses n^{δ_4} machines with $O(\frac{m}{n^{\delta_4}})$ space with high probability.*

PROOF. Partition V into parts $P_j = \{v \in V \mid 2^{j-1} < \deg(v) \leq 2^j\}$. Since the degree of each node is bounded by n , there are at most $\log n$ parts in the partition. Define the volume of part j as $V_j = |P_j| \cdot 2^j$. Parts having volume less than $m^{1-\epsilon}$ could all be mapped to one reducer without violating its space restriction. We now focus on parts with $V_j > m^{1-\epsilon}$, and so let P_j be such a part. Thus P_j contains between $\frac{m^{1-\epsilon}}{2^j}$ and $\frac{2m^{1-\epsilon}}{2^j}$ vertices. Let ρ

be an arbitrary reducer. Since h is universal, the probability that any vertex $v \in P_j$ maps to ρ is exactly $n^{-\delta_4}$. Therefore, in expectation, the number of vertices of P_j mapping to ρ is at most $\frac{2m^{1-\epsilon}}{2^j n^{\delta_4}}$. Since each of these vertices has degree at most 2^j , in expectation the number of edges that map to ρ is at most $\frac{2m^{1-\epsilon}}{n^{\delta_4}}$. Let the random variable X_j denote the number of vertices from P_j that map to ρ . Say that a bad event happens if more than $\frac{4m^{1-\epsilon}}{2^j}$ vertices of V_j map to ρ . Chernoff bounds tell us that the probability of such an event happening is $O(1/n^{2\delta_4})$,

$$\Pr \left[X_j > \frac{10m^{1-\epsilon}}{n^{\delta_4}} \right] < 2^{-\left(\frac{10m^{1-\epsilon}}{n^{\delta_4}}\right)} < \frac{1}{n^{2\delta_4}}. \quad (3)$$

Taking a union bound over all n^{δ_4} reducers and $\log n$ parts, we can conclude that the probability of any reducer being overloaded is bounded below by $1 - o(1)$. \square

4.3 Analysis of the MinCut Algorithm

We proceed to bound the total number of machines, maximum amount of memory, and the total number of rounds used by the MinCut algorithm.

LEMMA 4.2. *The total number of machines used by the MinCut algorithm is $n^{\delta_1} (n^{\delta_2+c-\epsilon} + n^{\delta_4})$.*

PROOF. The algorithm begins by running n^{δ_1} parallel copies of a simpler algorithm which first invokes Find_t , to find a threshold t for each instance. This algorithm uses n^{δ_2} parallel copies of a connected component algorithm, which itself uses $n^{c-\epsilon}$ machines (see Section 2.4). After finding the threshold, we invoke the Contract algorithm, which uses n^{δ_4} machines per instance. Together this gives the desired number of machines. \square

LEMMA 4.3. *The memory used by each machine during the execution of MinCut is bounded by $\max\{n^{2\delta_3}, n^{1+c-\delta_4}, n^{1+\epsilon}\}$.*

PROOF. There are three distinct steps where we must bound the memory. The first is the searching phase of Find_t . Since this algorithm executes instances of the connected components algorithm in parallel, the results of Section 2.4 ensure that each instance uses at most $\eta = n^{1+\epsilon}$ memory. The second is the contraction algorithm. Lemma 4.1 assures us that the input to each machine is of size at most $O(\frac{m}{n^{\delta_4}})$. Finally, the last step of MinCut requires that we load an instance with n^{δ_3} vertices, and hence at most $n^{2\delta_3}$ edges onto a single machine. \square

LEMMA 4.4. *Suppose the amount of memory available per machine is $\eta = n^{1+\epsilon}$. MinCut runs in $O(\frac{1}{\epsilon\delta_2})$ number of rounds.*

PROOF. The only variable part of the running time is the number of rounds necessary to found a threshold τ so that the number of connected components in Find_t is exactly n^{δ_3} . Observe that after the k^{th} recursive call, the number of edges with threshold between min and max is $\frac{m}{n^{\delta_2 k}}$. Therefore the algorithm must terminate after at most $\frac{1+c}{\delta_2}$ rounds, which is constant for constant δ_2 . \square

We are now ready to prove the main theorem.

THEOREM 4.1. *Algorithm MinCut returns the minimum cut in G with probability at least $1 - o(1)$, uses at most $\eta = n^{1+\epsilon}$ memory per machine and completes in $O(\frac{1}{\epsilon^2})$ rounds.*

PROOF. We first show that the success probability is at least

$$1 - \left(1 - \frac{n^{2\delta_3}}{n^2}\right)^{n^{\delta_1}}.$$

The algorithm invokes n^{δ_1} parallel copies of the following approach: (1) simulate Karger's contraction algorithm [10] for the first $n - n^{\delta_3}$ steps resulting in a graph G_t and (2) Identify the minimum cut on G_t . By Corollary 2.2 of [12] step (1) succeeds with probability at least $p = \Omega(n^{2\delta_3-2})$. Since the second step can be made to fail with 0 probability, each of the parallel copies succeeds with probability at least p . By running n^{δ_1} independent copies of the algorithm, the probability that all of the copies fail in step (1) is at most $1 - (1 - p)^{n^{\delta_1}}$.

To prove the theorem, we must find a setting of the parameters $\delta_1, \delta_2, \delta_3$, and δ_4 so that the memory, machines, and correctness constraints are satisfied.

$$\begin{aligned} \max\{n^{2\delta_3}, n^{1+c-\delta_4}\} &\leq \eta = n^{1+\epsilon} && \text{Memory} \\ n^{\delta_1} (n^{\delta_2+c-\epsilon} + n^{\delta_4}) &= o(m) = o(n^{1+c}) && \text{Machines} \\ \left(1 - \frac{n^{2\delta_3}}{n^2}\right)^{n^{\delta_1}} &= o(1) && \text{Correctness} \end{aligned}$$

Setting $\delta_1 = 1 - \epsilon/2$, $\delta_2 = \epsilon$, $\delta_3 = \frac{1+\epsilon}{2}$, $\delta_4 = c$ satisfies all of them. \square

Acknowledgments

We would like to thank Ashish Goel, Jake Hofman, John Langford, Ravi Kumar, Serge Plotkin and Cong Yu for many helpful discussions.

5. REFERENCES

- [1] E. Bakshy, J. Hofman, W. Mason, and D. J. Watts. Everyone's an influencer: Quantifying influence on twitter. In *Proceedings of WSDM*, 2011.
- [2] Bernard Chazelle. A minimum spanning tree algorithm with inverse-Ackerman type complexity. *Journal of the ACM*, 47(6):1028–1047, November 2000.
- [3] Jeffrey Dean and Sanjay Ghemawat. MapReduce: Simplified data processing on large clusters. In *Proceedings of OSDI*, pages 137–150, 2004.
- [4] Joan Feigenbaum, Sampath Kannan, Andrew McGregor, Siddharth Suri, and Jian Zhang. On graph problems in a semi-streaming model. *Theoretical Computer Science*, 348(2–3):207–216, December 2005.
- [5] Michael T. Goodrich. Simulating parallel algorithms in the mapreduce framework with applications to parallel computational geometry. Second Workshop on Massive Data Algorithmics (MASSIVE 2010), June 2010.
- [6] Hadoop Wiki - Powered By. <http://wiki.apache.org/hadoop/PoweredBy>.
- [7] Blake Irving. Big data and the power of hadoop. Yahoo! Hadoop Summit, June 2010.
- [8] Amos Israel and A. Itai. A fast and simple randomized parallel algorithm for maximal matching. *Information Processing Letters*, 22(2):77–80, 1986.
- [9] U Kang, Charalampos Tsourakakis, Ana Paula Appel, Christos Faloutsos, and Jure Leskovec. HADI: Fast diameter estimation and mining in massive graphs with hadoop. Technical Report CMU-ML-08-117, CMU, December 2008.
- [10] David R. Karger. Global min-cuts in \mathcal{RNC} and other ramifications of a simple mincut algorithm. In *Proceedings of SODA*, pages 21–30, January 1993.

- [11] David R. Karger, Philip N. Klein, and Robert E. Tarjan. A randomized linear-time algorithm for finding minimum spanning trees. In *Proceedings of the twenty-sixth annual ACM symposium on Theory of computing*, Proceedings of STOC, pages 9–15, New York, NY, USA, 1994. ACM.
- [12] David R. Karger and Clifford Stein. An $\tilde{O}(n^2)$ algorithm for minimum cuts. In *Proceedings of STOC*, pages 757–765, May 1993.
- [13] Howard Karloff, Siddharth Suri, and Sergei Vassilvitskii. A model of computation for MapReduce. In *Proceedings of SODA*, pages 938–948, 2010.
- [14] Jure Leskovec, Jon Kleinberg, and Christos Faloutsos. Graphs over time: Densification laws, shrinking daimeters and possible explanations. In *Proc. 11th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, 2005.
- [15] Jimmy Lin and Chris Dyer. *Data-Intensive Text Processing with MapReduce*. Number 7 in Synthesis Lectures on Human Language Technologies. Morgan and Claypool, April 2010.
- [16] Grzegorz Malewicz, Matthew H. Austern, Aart J.C. Bik, James C. Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. Pregel: A system for large-scale graph processing. In *Proceedings of SIGMOD*, pages 135–145, Indianapolis, IN, USA, June 2010. ACM.
- [17] Mike Schroepfer. Inside large-scale analytics at facebook. Yahoo! Hadoop Summit, June 2010.
- [18] Daniel A. Spielman and Nikhil Srivastava. Graph sparsification by effective resistances. In *Proceedings of STOC*, pages 563–568, New York, NY, USA, 2008. ACM.
- [19] Mirjam Wattenhofer and Roger Wattenhofer. Distributed weighted matching. In *Proceedings of DISC*, pages 335–348. Springer, 2003.
- [20] Tom White. *Hadoop: The Definitive Guide*. O'Reilly Media, 2009.
- [21] Yahoo! Inc Press Release. Yahoo! partners with four top universities to advance cloud computing systems and applications research. <http://research.yahoo.com/news/2743>, April 2009.

APPENDIX

A. WEIGHTED MATCHING LOWER BOUND

LEMMA A.1. *There is a graph where our algorithm compute a solution that has value $\frac{w(OPT)}{8-o(1)}$ with high probability.*

PROOF. Let $G(V, E)$ a graph on n nodes and m vertices, and fix a $W = 2^{\lceil \log m \rceil}$. We say that a bipartite graph $G(V, E_1, E_2)$ is balanced if $|E_1| = |E_2|$. Consider the following graph: there is a central balanced bipartite clique, B_1 , on $\frac{n}{2 \log W}$ nodes and all the edges of the clique have weight $\frac{W}{2} + 1$. Every side of the central bipartite clique is also part of other $\log W - 1$ balanced bipartite cliques. We refer to those cliques as $B_2^{E_1}, B_3^{E_1}, \dots, B_W^{E_1}, B_2^{E_2}, B_3^{E_2}, \dots, B_W^{E_2}$. In both $B_i^{E_1}$ and $B_i^{E_2}$ we have that the weight of the edges in them have weight $\frac{W}{2^i} + 1$. Furthermore every node in B_1 is also connected with an additional node of degree one with an edge of weight W , and every node in $B_i^{E_1} \setminus B_1$ and $B_i^{E_2} \setminus B_1$ is connected to a node of degree one with an edge of weight $\frac{W}{2^{i-1}}$. Figure 3 shows the subgraph composed by B_1 and the two graphs $B_i^{E_1}$ and $B_i^{E_2}$.

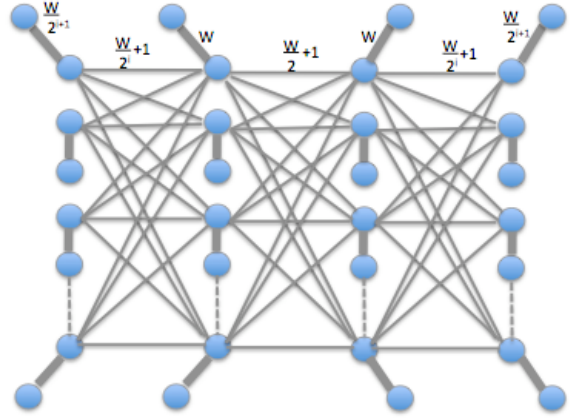


Figure 3: The subgraph (we have drawn only the edges of weight W , $\frac{W}{2} + 1$, $\frac{W}{2^i} + 1$ and $\frac{W}{2^{i+1}} + 1$) of the graph of G for which our algorithm finds a solution with value $\frac{w(OPT)}{8-o(1)}$ with high probability.

Note that the optimal weighted maximum matching for this graph is the one that is composed by all the edges incident to a node of degree one and its total value is

$$\begin{aligned}
 & 2W \cdot \frac{n}{2 \log W} + W \cdot \frac{n}{2 \log W} + \frac{W}{2} \cdot \frac{n}{2 \log W} + \dots + \frac{n}{2 \log W} \\
 &= 2W \frac{n}{2 \log W} \sum_{i=0}^W \frac{1}{2^i} \\
 &= 2W \frac{n}{2 \log W} \cdot \frac{1 - 2^{W+1}}{1 - 2^{-1}} \\
 &= (4 - o(1))W \frac{n}{2 \log W}.
 \end{aligned}$$

Now we will show an upper-bound on the performance of our algorithm that holds with high probability. Recall that in step one our algorithm splits the graph in G_1, \dots, G_W subgraph where the edges in G_i are in in $(2^{i-1}, 2^i]$ then it computes a maximal matching on G_i using the technique shown in the previous subsection. In particular the algorithm works as follows: it samples the edges in G_i with probability $\frac{1}{|E_i|^\epsilon}$ and then computes a maximal matching on it, finally it tries to match the unmatched nodes using the edges between the unmatched nodes in G_i .

To understand the value of the matching returned by the algorithm, consider the graph G_i note that this graph is composed only by the edges in $B_i^{E_1}, B_i^{E_2}$ and the edges connected to nodes of degree one with weight $\frac{W}{2^{i-1}}$. We refer to those last edges as the heavy edges in G_i . Note that the heavy edges are all connected to vertices in $B_i^{E_1} \setminus B_1$ and $B_i^{E_2} \setminus B_1$. Let s be the number of vertices in a side of $B_i^{E_1}$, note that G_i , for $i > 1$ has $6s$ nodes and $s^2 + 2s$ edges.

Recall that we sample an edges with probability $\frac{1}{C|V_i|^\epsilon} = \frac{1}{C(6s)^\epsilon}$, so we have that in expectation we sample $\Theta\left(2s \frac{1}{(2s)^\epsilon}\right) = \Theta\left((2s)^{1-\epsilon}\right)$ heavy edges, thus using the Chernoff bound we have that the probability that the number of sampled heavy edges is bigger or equal than $\Theta\left(3(6s)^{(1-\epsilon)}\right)$ is $e^{-3(6s)^{(1-\epsilon)}}$. Further notice that by lemma 3.1 for every set of node in $B_i^{E_1}$ or $B_i^{E_2}$ with $(6s)^{1+2\epsilon}$ edges we have at least an edge in it with probability $e^{-\sqrt{6s}(\frac{1}{C}(6s)^\epsilon - \log(6s))}$.

Thus the maximum number of nodes left unmatched in $B_i^{E_1}$ or $B_i^{E_2}$ after step 2 of the maximal matching algorithm is smaller than $(6s)^{\frac{1+2\epsilon}{2}} + \Theta\left(3(6s)^{(1-\epsilon)}\right)$ so even if we matched those nodes with the heavy edges in G_i we use at most $(6s)^{\frac{1+2\epsilon}{2}} + \Theta\left(3(6s)^{(1-\epsilon)}\right)$ of those edges. Thus we have that for every G_i , for every $i > 1$ the maximal matching algorithm uses at most $\Theta\left(6(6s)^{(1-\epsilon)} + (6s)^{\frac{1+2\epsilon}{2}}\right) = o\left(\frac{s}{\log^2 s}\right)$ heavy edges with probability $\left(1 - 2e^{-\sqrt{6s}\left(\frac{1}{C}(6s)^\epsilon - \log(6s)\right)}\right)$.

With the same reasoning, just with different constant, we notice that the same fact holds also for G_1 . So we have that for every G_i we use only $o\left(\frac{|V_i|}{\log^2 |V_i|}\right)$ heavy edges with probability $\left(1 - 2e^{-\Theta(\sqrt{|V_i|})}\right)$, further notice that every maximal matching that the algorithm computes it always matches the nodes in B_1 , because it is always possible to use the edges that connect those nodes to the nodes of degree 1.

Knowing that we notice that the final matching that our algorithm outputs is composed by the maximal matching of G_1 plus all the heavy edges in maximal matching of G_2, \dots, G_W . Thus we have that with probability $\prod_i \left(1 - 2e^{-\Theta(\sqrt{|V_i|})}\right) = 1 - o(1)^3$ the total weight of the computed solution is upper-bounded by $\left(\frac{W}{2} + 1\right) \frac{n}{2 \log W} + W \log W \cdot o\left(\frac{s}{\log^2 s}\right) = \frac{W}{2} \frac{n}{2 \log W} + o(W)$ and so the ratio between the optimum and the solution is

$$\frac{(4 - o(1))W \frac{n}{2 \log W}}{\frac{W}{2} \frac{n}{2 \log W} + o(W)} = \frac{1}{8 - o(1)}.$$

Thus the claim follows. \square

³Note that every $|V_i| \in \Theta\left(\frac{n}{\log W}\right)$