Readings in Algorithms - CS167

Filtering: A Method for Solving Graph Problems in MapReduce

A paper by Lattanzi, Moseley, Suri and Vassilvitskii

Nikolaos A. Platias

Stanford University, Spring 2014

1 Introduction

The paper by Lattanzi et al. develops a novel technique for solving graph problems in MapReduce [8]. MapReduce is an extensively used framework for parallel computation and large scale data analysis, both in industry and academia, since its formulation in 2004 by Dean et al. at Google [5]. We will review its key ideas and capabilities, as they will prove useful in the development of the algorithms that we will consider. We will also review MRC, a model for reasoning about the efficiency of MapReduce algorithms, which was introduced in a paper by Karloff et al. [4] (the paper has two authors in common with the paper at hand). One of the essential elements of MRC is the restriction of the number of machines available to an algorithm, as well as the amount of memory per machine, to be sublinear in the size of the input. This restriction alone allows for total memory across machines that is superlinear in the size of the input. All algorithms that we will discuss belong to a more restricted subset of MRC algorithms, in that the total memory available across machines is linear in the size of the input. This restriction improves the applicability of the algorithms, as very large inputs may make MRC algorithms that do not obey this impractical.

An important feature of the discussed algorithms, both theoretically and practically, is that they only require a constant number of MapReduce rounds, provided that each machine has memory $O(n^{1+\epsilon})$ where n is the number of nodes in the input graph. This is an important milestone for MapReduce algorithms, as we will discuss in Section 2. The authors also show how to implement many of the algorithms when each machine has memory only linear in n (in more rounds), thus demonstrating a recurring tradeoff between the amount of memory per machine and the number of MapReduce rounds.

The main motivation for the algorithms that we'll see is the need to process massive graphs that don't fit in the memory of a single machine. Examples of such graphs are citation graphs, phone call graphs, social networks and the web graph. Solving interesting graph problems on such graphs requires novel techniques, as the graph must be split across multiple machines. One of the key limitations of traditional graph algorithms is that they require random access to the edge set. We will need to overcome this limitation in the algorithms we develop, since no single machine will have access to the full edge set.

The technique that the paper introduces for designing graph algorithms in this setting is called *filtering*. Filtering algorithms have two stages. First, they selectively drop, or filter, parts of the input in parallel, harnessing the parallelization provided by MapReduce, until the input is small enough to fit into a single machine. Then, they complete the computation on the reduced input. The key challenge in filtering algorithms is to sufficiently reduce the input so that it fits into a single machine without losing grasp of an optimal or approximately optimal solution. To address this challenge, some of the filtering algorithms that we will see will need to harness both the parallel and the sequential power of MapReduce. Filtering has several incarnations, and we will explore a few of those in the algorithms that follow.

The exposition is organized as follows: in Section 2 we review some preliminaries, including MapReduce and MRC. In Section 3 we explore two algorithms for finding MSTs in MapReduce: we present a previously existing algorithm as well as a filtering algorithm that addresses some of its limitations. In Section 4 we explore filtering algorithms for finding a maximal matching.

2 Preliminaries

2.1 MapReduce

MapReduce is a programming paradigm that formalizes the repeated process of partitioning data across machines, doing some computation on that data, then aggregating relevant parts together for further computation. MapReduce is designed for large computing clusters consisting of commodity processors connected by Ethernet cables, a standard setting in modern parallel computing [2]. Part of its power comes from the abstraction it provides to the programmer: it shields him from the low-level details of the parallelization, all of which are handled by the run-time system, and allows him to focus on the crux of the computation, summarized by two functions, Map (or mapper) and Reduce (or reducer).

A MapReduce computation consists of several rounds, each of which has a Map stage, a Shuffle stage and a Reduce stage. The basic unit of information is a $\langle key; value \rangle$ pair. The Map stage is executed by a number of Map tasks, each run on a different machine. A Map task executes the mapper, which takes as input a single $\langle key; value \rangle$ pair and produces as output a multiset of new $\langle key; value \rangle$ pairs. A Map task may execute the mapper on many $\langle key; value \rangle$ pairs sequentially, but operates on a *single* $\langle key; value \rangle$ pair at a time. In the Shuffle stage, the $\langle key; value \rangle$ pairs produced by the mappers are aggregated and distributed across a set of Reduce tasks, such that all *values* with the same *key* are grouped together and sent to the same Reduce task. Finally, the Reduce stage is executed by a number of Reduce tasks, each run on a different machine. A Reduce task executes the reducer, which takes as input all of the values associated with a single key k and produces a multiset of $\langle key; value\rangle$ pairs with the same key k. Similarly to Map tasks, Reduce tasks may execute the reducer on many *keys* and their associated values sequentially, but operate on a *single* key and its associated values at a time. After each MapReduce round, the output of the Reduce stage serves as input to the Map stage of the next round. Note that a Map or Reduce task is assigned to a machine by the system, and that the algorithm designer does not have to worry about the distribution of mappers and reducers to tasks. His sole concern is the mapper and reducer.

The parallel power of MapReduce comes from the fact that the Map tasks as well as the Reduce tasks in each round are executed in parallel. Its sequential nature comes from the important observation that the Map stage needs to completely finish before the Reduce stage can begin. The details in the above exposition are presented mostly for completeness, as we will not specify precise mappers and reducers for each algorithm that we consider. However, it will become clear that filtering can be naturally expressed in MapReduce. The main idea is that the filtering process is executed by the reducers, with the mappers serving the purpose of partitioning the (possibly reduced) input across machines for the next filtering step. The filtering thus proceeds in sequential stages, each corresponding to one or more MapReduce rounds. In each stage, the filtering takes place in parallel, since the mappers and reducers in each round execute in parallel. This combination of sequential and parallel computation will be essential in some of the filtering algorithms that we will encounter.

2.2 *MRC*

MRC is a model for reasoning about the efficiency of MapReduce algorithms, conceived by Karloff et al [4]. It places restrictions on the number of machines available to the algorithm, the amount of memory per machine, the runtime of mappers/reducers and the number of rounds in the computation. There are two key ideas that motivate the formulation of *MRC*. First, the full power of MapReduce is realized when the input size is massive. Hence, if a MapReduce algorithm is to have any practical bearing, the number of machines available to it as well as the amount of memory per machine should be substantially sublinear in the size of the input. Second, the performance bottleneck in most MapReduce computations is the time required for machines to transfer data amongst each other across the network. This idea is most aptly captured by the Communication Cost Model [2, 3] for measuring the performance of MapReduce algorithms, introduced by Afrati et al., according to which the communication cost of an algorithm is the sum of the input sizes to all tasks implementing the algorithm. As such, a MapReduce round can be very costly time-wise, so it is essential to minimize the number of rounds.

Formally, *MRC* is parametrized by an integer i. Let N be the size of the input and fix $\epsilon > 0$.

An algorithm in MRC^i consists of a sequence $\langle \mu_1, \rho_1, ..., \mu_R, \rho_R \rangle$ of mappers and reducers which output the correct answer with probability at least 3/4, such that:

- Each μ_r/ρ_r is a randomized mapper/reducer implemented by a RAM with $O(logN)$-length words that uses $O(N^{1-\epsilon})$ space and time polynomial in N.
- The total number of machines available is $O(N^{1-\epsilon})$.
- The total size of all $\langle key; value \rangle$ pairs output by μ_r is $O(N^{2-2\epsilon})$.
- The number of rounds $R = O(log^i N)$.

Observe that the total amount of memory available across all machines is $O(N^{2-2\epsilon})$. This is where the third requirement comes from: all $\langle key; value \rangle$ pairs output by μ_r have to be stored temporarily before the Reduce stage begins (since the Map stage needs to complete first), hence their total size cannot exceed the total amount of available memory. It should also be noted that we strive for algorithms that only require a constant number of rounds (i.e. in MRC^0).

Note that for $\epsilon < 1/2$ the overall available memory is superlinear in N. In the paper at hand, the authors consider a more restricted subset of MRC algorithms where the total memory available across machines is $O(N)$. In particular, it is assumed that each machine has memory $\eta = O(N^{1-\epsilon})$ and that the number of machines available is $O(N/\eta)$, just enough (asymptotically) to hold the input. Those stricter assumptions improve the applicability of filtering algorithms.

Now we discuss a powerful lemma [4] which provides a framework for designing algorithms in MRC and guarantees that the MRC constraints will be satisfied. First, we introduce the concept of an MRC-parallelizable function, which will be used as a building block for our lemma.

Definition 2.1. *Let S be a set. Call a function f on S MRC-parallelizable if there exist functions g, h such that:*

- *For any partition $T = \{T_1, ..., T_k\}$ of S, where $\cup_i T_i = S$ and $T_i \cap T_j = \emptyset$ for $i \neq j$, f can be expressed as $f(S) = h(g(T_1), ..., g(T_k))$.*
- *g and h can be computed in time polynomial in $|S|$.*

In essence, an MRC-parallelizable function can be computed on a set S by first partitioning S into parts, evaluating g on each part and then combining the results with h. The following lemma harnesses the power of MRC-parallelizable functions:

Lemma 2.2. *Consider a universe $\mathcal{U}$ of size N and a collection $S = \{S_1, ..., S_k\}$ of subsets of $\mathcal{U}$ such that $\sum_{i=1}^{k} |S_i| \leq N^{2-2\epsilon}$ and $k \leq N^{2-3\epsilon}$. Let $F = \{f_1, ..., f_k\}$ be a collection of MRC-parallelizable functions. Then the output $f_1(S_1), ..., f_k(S_k)$ can be computed using $O(N^{1-\epsilon})$ reducers with $O(N^{1-\epsilon})$ memory per reducer in 2 rounds.*

The lemma guarantees that MRC-parallelizable functions can be computed in MRC on a set of subsets of a universe $\mathcal{U}$, provided that the number of subsets is not too large and that their total size does not exceed the overall amount of memory available to the algorithm. This lemma will prove useful when we try to implement our Maximal Matching algorithm in MapReduce.

2.3 Dense Graphs

A graph $G = (V, E)$ with $|V| = n$, $|E| = m$ is said to be c-dense if $m = n^{1+c}$ where $0 < c \leq 1$. The algorithms that we will see are designed for c-dense graphs. A result by Leskovec et al. [6] that justifies the prevalence of dense graphs states that social graphs grow dense over time. In particular, if $n(t)$, $e(t)$ is the number of nodes and edges of G at time t, $e(t) \propto n(t)^{1+c}$ for some $0 < c \leq 1$.

2.4 Chernoff Bounds

Let us recall a formulation of the Chernoff bounds to which we will refer in our further analysis. Let X be a sum of n independent random variables $X_i \in \{0, 1\}$ where $E[X] = \mu$.
Then for any $\delta \geq 0$ we have $Pr[X > (1 + \delta)\mu] \leq e^{\frac{-\delta^2 \mu}{2+\delta}}$.

3 Finding MSTs in MapReduce

In this Section we will explore two algorithms for finding MSTs in MapReduce. The first algorithm (Vertex Partition) [4] was presented in the paper introducing MRC. The second algorithm (Edge Partition), our first example of a filtering algorithm, will be shown to address some of its limitations.

3.1 Vertex Partition Algorithm for MST

Given a graph $G = (V, E)$ where $|V| = n$, $|E| = m$ and $m = n^{1+c}$, we want to compute its MST. We can assume without loss of generality that edge weights are unique (for example, by appending an index to each weight to break ties). The main idea of the algorithm is simple: we randomly partition V into k equally sized disjoint subsets $V = V_1 \cup V_2 \cup ... \cup V_k$, and for each $i < j$ we define the induced edge sets $E_{i,j} = \{(u, v) \in E | u, v \in V_i \cup V_j\}$, as well as the induced subgraphs $G_{i,j} = (V_i \cup V_j, E_{i,j})$. The algorithm computes the Minimum Spanning Forest $M_{i,j}$ of each $G_{i,j}$, then returns the MST of the graph $H = (V, \cup_{i,j} M_{i,j})$. The reason this algorithm works is that any edges that are part of the MST of a subgraph of G can be safely discarded, as they will not be part of the MST of G either. We formalize this idea in the following theorem.

Algorithm 1 Vertex Partition Algorithm for MST

```
Input: $G = (V, E)$
Output: The MST of $G$
1 Randomly partition $V$ into $k$ equally sized disjoint subsets $V = V_1 \cup V_2 \cup ... \cup V_k$
2 for $i < j \in \{1, ..., k\}$ compute in parallel
3       $E_{i,j} = \{(u, v) \in E | u, v \in V_i \cup V_j\}$
4       $G_{i,j} = (V_i \cup V_j, E_{i,j})$
5       $M_{i,j}$, the Minimum Spanning Forest of $G_{i,j}$
6 end for
7 return the MST of $H = (V, \cup_{i,j} M_{i,j})$
```

It is important to note that lines 3-5 can be computed in parallel, which is where the power of this algorithm lies. We will see that with high probability, the sets $E_{i,j}$ will fit in a single machine. We will also show that H fits in a single machine. In a MapReduce implementation, mappers would thus distribute edges to create the sets $E_{i,j}$ and reducers would compute $M_{i,j}$ on each $G_{i,j}$. Finally, a single reducer would compute the MST of H.

3.1.1 Analysis

Theorem 3.1. *The algorithm finds the MST of G.*

Proof. We observe that any edge e discarded by the algorithm when computing $M_{i,j}$ on $G_{i,j}$ must have been the heaviest edge on a cycle in $E_{i,j}$ by the cycle property of the MST. Since $E_{i,j} \subset E$, e is also the heaviest edge on a cycle in G, so it will not be part of the MST of G. The same applies to edges discarded in the computation of the MST of H, so our algorithm returns the MST of G. $\square$

We state without proof (see [4]) a theorem that establishes an upper bound on the size of every $E_{i,j}$ when we choose $k = n^{c/2}$, allowing us to show that the algorithm is in MRC.

Theorem 3.2. *If $k = n^{c/2}$ then with high probability the size of every $E_{i,j}$ is $\tilde{O}(n^{1+c/2})$*

The above theorem guarantees that with high probability the input to all reducers in the first round (computing the MST of each $G_{i,j}$) is sublinear in the size of the graph $N = O(n^{1+c})$. Note that in the first round we execute $\binom{k}{2} = O(n^c)$ reducers, each of which produces a Minimum Spanning Forest of size at most $2n/k - 1 = O(n^{1-c/2})$, since the number of vertices in $G_{i,j}$ is $2n/k$. Hence the size of H, which is the input to the only reducer in the second round, is $O(n^{1+c/2})$ which is again sublinear in N. Also note that the total size of $\langle key; value \rangle$ pairs output by mappers in both rounds is sublinear in $O(N^2)$, and that the number of rounds is constant. We conclude that the algorithm is in MRC^0.

3.1.2 Limitations

First, let us try to evaluate the algorithm in terms of Communication Cost. Call an edge internal if both of its endpoints lie in the same vertex partition V_i. An important observation is that in the first round we replicate all internal edges $k-1$ times: if an edge e has both of its endpoints in V_i, it will be part of $E_{i,1}, ..., E_{i,i-1}, E_{i,i+1}, ..., E_{i,k}$, each of which is created in parallel to be sent to the appropriate reducer. Thus, if we end up with a lot of internal edges the communication cost can be very high. It is possible to have $O(m)$ internal edges (albeit with low probability), in which case the communication cost of the algorithm will be $O(n^{1+3c/2})$, which can be very high.

Another limitation may be the amount of memory required per machine. It is possible that some reducers will require $\Omega(n^{1+c/2})$ memory, which may be a lot for large n and c.

4 Edge Partition Algorithm for MST

As above, our input graph $G = (V, E)$ satisfies $|V| = n$, $|E| = m$ and $m = n^{1+c}$. This algorithm will address the limitations described above and will make the filtering process concrete. It will also make significantly weaker assumptions about the amount of available memory (as mentioned earlier): we assume that each machine has memory $\eta = O(n^{1+\epsilon})$ for some $0 < \epsilon < c$ and that the number of machines available is $O(N/\eta)$. This assumption is clearly weaker in terms of overall memory across machines, which is asymptotically just enough to hold the graph. It is also weaker in terms of the memory per machine: ϵ is a knob that we can adjust depending on n and on the amount of memory our machines possess. This addresses the second limitation that we discussed above.

The algorithm is very similar to the previous one. The key difference is that it partitions the edge set rather than the vertex set. The filtering process works as follows: we partition the edge set across the available machines, compute the MST of each induced subgraph (thus *filtering* out edges that fall off of the MST) and iterate until the filtered graph is small enough to fit in memory. We then complete the MST computation in a single machine. More formally:

Algorithm 2 Edge Partition Algorithm for MST

```
**Input:** $G = (V, E)$
**Output:** The MST of $G$
1 while $|E| > \eta$
2       Let $\ell = \Theta(|E|/\eta)$ the number of machines in this round
3       Partition $E$ into $E_1, .., E_\ell$ by assigning the edges uniformly at random
4       In Parallel: Compute the MST $T_i$ of $G = (V, E_i)$ for each $i$
5       Let $E = \cup_i T_i$
6 end while
7 return the MST of $G = (V, E)$
```

Implementing this algorithm in MapReduce is very natural. First note that each iteration of the while loop corresponds to a MapRedcue round. E is partitioned in the Map stage across ℓ reducers, each of which computes the MST of the subgraph it is given. In the last round, a single reducer computes the MST of the filtered graph. The filtering thus takes place in rounds, and is done in parallel within each round.

Theorem 4.1. *The algorithm finds the MST of G.*

Proof. The proof is identical to the corresponding proof of the previous algorithm: all discarded edges do not belong to the MST of G. □

Theorem 4.2. *The algorithm terminates after at most $\lceil c/\epsilon \rceil$ rounds.*

Proof. The key insight is that we can bound the number of "surviving" edges after each round by $\ell \cdot (n-1)$, since we collect a Minimum Spanning Forest from each partition. Let E be the edge set at the start of the current round. Note that $\ell = \Theta(|E|/n^{1+\epsilon})$, so the number of "surviving edges" is at most $|E| \cdot n^{-\epsilon}$. Since $|E| = O(n^{1+c})$ before the first round, in at most $\lceil c/\epsilon \rceil - 1$ rounds E fill fit in a single machine, at which point there is a single round left. □

Assuming c, ϵ are constants the number of rounds required is constant.

Lemma 4.1. *The memory constraints of all machines are satisfied with high probability.*

Proof. Fix an iteration of the algorithm and a machine i, and note that the number of edges E_i assigned to it is a sum of independent indicator random variables. Furthermore, by choosing $\ell = 2|E|/\eta$ we have $E[|E_i|] = \eta/2$. By applying the Chernoff bound with $\delta = 2$ we obtain $Pr[|E_i| > \eta] \leq e^{-\eta} = e^{-n^{1+\epsilon}}$. By applying the union bound over all machines and all iterations, the overall probability of violating memory constraints is at most $\lceil c/\epsilon \rceil \cdot \ell \cdot e^{-n^{1+\epsilon}} = O(n^{c-\epsilon} \cdot e^{-n^{1+\epsilon}})$ □

Furthermore, it is clear that the total $\langle key; value \rangle$ pair size constraint is satisfied. The algorithm is thus in MRC^0.

Note the importance of our assumption that G is c-dense: if that was not the case we would have $\eta = O(n^{1-\epsilon})$ and it is unclear how the filtering would work (we would not be able to argue that the size of the edge set is reduced by $n^{-\epsilon}$ per round).

Comparing the Edge Partition algorithm to the Vertex Partition algorithm, an advantage beyond the more flexible assumptions is that we can guarantee an $O(m)$ communication cost. This follows directly from the observation that there is a constant number of rounds, and that in each round the communication cost is $O(m)$: $O(m)$ for the edges since there is no replication, and $O(n\ell) = O(n^{1+c-\epsilon}) = O(m)$ for the vertices.

5 Finding a Maximal Matching in MapReduce

In this Section we explore approaches to finding a maximal matching in MapReduce. We will see that we will need to refine the filtering approach taken in the Edge Partition algorithm to make it work with maximal matchings.

Recall: a **matching** M is a subset of E such that no two edges in M share an endpoint. A **maximal matching** is a matching that cannot be augmented without first deleting edges from it.

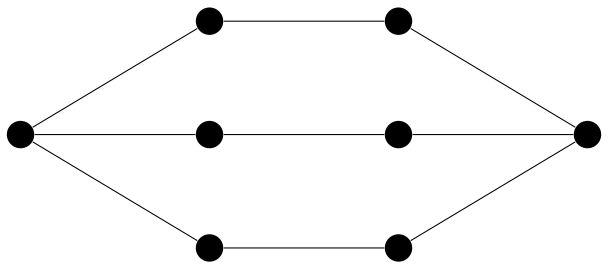
5.1 A first Filtering attempt

We formulate an initial filtering algorithm for finding a maximal matching by emulating our Edge Partition filtering algorithm: we partition the edge set across the available machines, compute a maximal matching on each induced subgraph (thus filtering out edges that fall off of the matching) and iterate until the filtered graph is small enough to fit in memory. We then return the maximal matching found in the filtered graph. More formally:

Algorithm 3 A Filtering attempt for Maximal Matching

```
Input: $G = (V, E)$
Output: A maximal matching on $G$
1 while $|E| > \eta$
2     Let $\ell = \Theta(|E|/\eta)$ the number of machines in this round
3     Partition $E$ into $E_1, .., E_\ell$ by assigning the edges uniformly at random
4     In Parallel: Compute a maximal matching $M_i$ on $G = (V, E_i)$ for each $i$
5     Let $E = \cup_i M_i$
6 end while
7 return a maximal matching on $G = (V, E)$
```

Unfortunately, our faithful reproduction of the previous filtering algorithm does not work. The problem is that the matchings that we compute on each partition do not coordinate and may therefore cancel each other out. To make this problem concrete, consider a family of graphs G_n with two "central" vertices, connected by n 3-edge strands. The graph G_3 is shown below. First observe that any maximal matching on G_n has size at least n: the matching must contain at least one edge per strand, otherwise the middle edge of an "unoccupied" strand can be added without touching any other edges of the matching.



We now demonstrate a simple scenario that has non-negligible probability where the algorithm will make a mistake on G_n. Consider the case where in the initial edge partition there exist 3 strands that are undivided, i.e. all three edges on the strand are sent to the same machine. It is possible that each machine picks the left and right edge from each of those strands as a maximal matching. If this happens, eventually at most two edges will "survive" among those 3 strands. This is because only one of the three left edges can survive, since all three are incident on the left "central" node. The same applies to the right edges. The resulting matching will have size at most $n - 1$ and thus cannot be maximal, and the probability that this occurs is $1/O(1)$. We can also construct scenarios (with low probability) where the produced matching is arbitrarily bad, i.e. it is arbitrarily far from being maximal.

The reason our attempt failed is that the matchings computed in different machines can cancel each other out due to shared vertices. We didn't have this problem when dealing with MSTs because of the cycle property, which guaranteed that the edges we discarded would not be part of the overall MST. Unfortunately we don't have a similar property here. To overcome this, we will have to enforce coordination between the matchings by filtering out edges that are incompatible with all locally computed maximal matchings.

5.2 A Greedy Sequential Algorithm for finding a Maximal Matching

To shed light on this idea, let us briefly consider a greedy sequential algorithm for finding a maximal matching [7]. The main idea is to iteratively pick an edge, add it to the matching and then discard all of the edges incident on its endpoints, until there are no edges left. More formally:

Algorithm 4 A Greedy Sequential Algorithm for finding a Maximal Matching

```
Input: $G = (V, E)$
Output: A maximal matching on $G$
1 Let $M = \emptyset$
2 while $E \neq \emptyset$
3     Pick $(u, v) \in E$
4     Add $(u, v)$ to $M$
5     Remove from $E$ all edges incident on $u$ and $v$
6 end while
7 return $M$
```

The reason this algorithm produces a matching is that no edges in the matching can share endpoints, by construction. The reason it is maximal is that we iterate until we have exhausted E.

5.3 A Filtering Algorithm for finding a Maximal Matching

How can we adopt the strategy we just saw to a filtering algorithm? The key idea is that whenever we add an edge (or set of edges) to the matching, we remove the edges that conflict. This sequential aspect of the algorithm is unavoidable, so we will have to harness the sequential power of MapReduce to achieve it.

Again we assume that our input graph G is c-dense ($m = n^{1+c}$). We will consider two cases in our analysis for the amount of memory per machine η: $\eta \geq 40n$ and $\eta = n^{1+\epsilon}$ for some constant $0 < \epsilon < c$. The gist of the algorithm is as follows: start with an empty matching M. Sample a set of edges that fits in a single machine, compute a maximal matching M' on them and add it to M, then in parallel discard all edges that touch the matching. Repeat this process until the edges fit in a single machine. Finally compute a maximal matching M'' on the filtered graph and add it to M. The filtering thus occurs in stages, where in each stage we filter out the edges that conflict with the newly computed matching.

A convenient way of specifying the "surviving" edges from each filtering step is in terms of unmatched vertices. A vertex is unmatched if no edge in M is incident on it. If I is the set of unmatched vertices after a filtering step, the "surviving" edges will be precisely the edges with both endpoints in I, i.e. the edges in the set $E[I]$ (the projection of E onto the set of vertices I). The precise algorithm is the following:

Algorithm 5 A Filtering Algorithm for finding a Maximal Matching

```
Input: $G = (V, E)$
Output: A maximal matching on $G$
1 Let $M = \emptyset$ and $S = E$
2 while $|S| > \eta$
3       Sample every edge $(u, v) \in S$ independently with probability $p = \frac{\eta}{10|S|}$
4       Let $E'$ be the set of sampled edges. If $|E'| > \eta$ the algorithm fails.
5       In a single machine, compute a maximal matching $M'$ on $G' = (V, E')$
6       Set $M = M \cup M'$
7       In Parallel: Compute the set of unmatched vertices $I$ as well as $E[I]$
8       Set $S = E[I]$
9 end while
10 Compute a maximal matching $M''$ on $G = (V, S)$
11 return $M \cup M''$
```

We are first concerned with bounding the number of iterations the algorithm requires. Our strategy will be the following: we will try to obtain an upper bound on the number of "surviving" edges $E[I]$ after each filtering step, which will immediately imply a bound on the number of iterations.

We begin by proving a lemma that will be instrumental in bounding the number of "surviving" edges. The lemma says that with high probability, any sufficiently large induced subgraph of G (obtained by projecting G onto a subset of the vertices I) will have an edge in our sample set E'. But first, we prove another lemma that will be necessary for an upper bound in the following proof.

Lemma 5.1. $(1-p)^{-\frac{1}{p}} \geq e$ *for* $0 < p < 1$

Proof. First note that the function $f(p) = (1-p)^{-\frac{1}{p}}$ is increasing in the interval $(0, 1)$ (it is easy to check that its first derivative is positive). We will show that $\lim_{p \to 0} f(p) = e$, concluding the proof. Observe that $\lim_{p\to 0} f(p) = \lim_{p\to 0}(1 + \frac{1}{\frac{1}{p} - 1})^{\frac{1}{p}} = \lim_{x\to\infty}(1 + \frac{1}{x})^{x+1} = e$ where $x = \frac{1}{p} - 1$. □

Lemma 5.2. *Let E' be a set of edges sampled from E independently with probability p. Then with probability at least $1 - e^{-n}$ all $I \subset V$ satisfy either $|E[I]| < 2n/p$ or $E[I] \cap E' \neq \emptyset$.*

Proof. First we fix an induced subgraph $G[I] = (I, E[I])$ where $|E[I]| \geq 2n/p$. The probability that $E[I] \cap E' = \emptyset$, i.e. that no edge in $E[I]$ was sampled, is $(1-p)^{|E[I]|} \leq (1-p)^{2n/p} \leq e^{-2n}$, where the second inequality follows from the previous lemma. We now apply the union bound over all possible induced subgraphs (there are at most 2^n of them): the probability that there exists a subgraph with at least $2n/p$ edges, none of which was sampled, is at most $2^n \cdot e^{-2n} \leq e^{-n}$. The result follows directly. □

We will now use the above lemma to bound the number of iterations of the algorithm (note that this is not the number of MapReduce rounds).

Theorem 5.1. *If $\eta \geq 40n$ the algorithm terminates in $O(logn)$ iterations with high probability. Furthermore, if $\eta = n^{1+\epsilon}$ for some constant $0 < \epsilon < c$ the algorithm terminates in at most $\lfloor c/\epsilon \rfloor$ iterations with high probability.*

Proof. Consider the ith iteration of the algorithm, and denote by E_i the set of remaining edges at the beginning of the iteration, E' the set of sampled edges, M' the matching that is computed on E' and I the vertices that are unmatched at the end of the iteration. We want to obtain a bound on E_{i+1} relative to E_i. Note that E_{i+1} is the set of "surviving" edges $E[I]$. The key observation is that $E[I] \cap E' = \emptyset$, i.e. we did not sample any edges from $E[I]$ during this iteration. This is because edges in $E[I]$ do not share endpoints with edges in M', so if we had sampled an edge in $E[I]$ we would have added it in M', which is impossible since vertices in I are unmatched. We conclude, by the previous lemma, that $|E[I]| < 2n/p$ with high probability, hence $|E_{i+1}| < \frac{20n \cdot |E_i|}{\eta}$. We start with $|E| = n^{1+c}$.

- If $\eta \geq 40n$ this implies $|E_{i+1}| < |E_i|/2$, so the algorithm terminates in $log|E| = (1+c)logn = O(logn)$ iterations with high probability.
- If $\eta = n^{1+\epsilon}$ this implies $|E_{i+1}| < 20|E_i| \cdot n^{-\epsilon}$, so the algorithm terminates in $\lfloor c/\epsilon \rfloor$ iterations with high probability.

□

We are now ready to analyze the probability that our algorithm succeeds:

Theorem 5.2. *The algorithm finds a maximal matching of $G = (V, E)$ with high probability.*

Proof. We first show that if the algorithm does not fail (i.e. it terminates) then it produces a maximal matching. Assume the contrary; then there exists an edge $(u, v) \in E$ such that both of its endpoints are unmatched, so $u, v \in I$ in the last iteration of the algorithm. Then $(u, v) \in E[I]$ during the computation of M'' in the last step, so $(u, v) \in M'' \subset M$ which is a contradiction.

We now bound the probability that the algorithm fails, first by bounding the probability of failure in a fixed iteration. Failure occurs when $|E'| > \eta$. Observe that the random variable $|E'|$ is the sum of $|S|$ independent indicator random variables, each denoting whether or not an edge of S was sampled. We thus have $E[|E'|] = |S| \cdot p = \eta/10$. We apply the Chernoff bound with $\delta = 9$ to get $Pr[|E'| > \eta] \leq (e^{\frac{81}{110}})^{-\eta} \leq 2^{\eta} \leq 2^{-40n}$ since $\eta \geq 40n$. By our previous theorem the number of iterations is at most $O(logn)$, so by the union bound the probability of overall failure is at most $O(logn \cdot 2^{-40n})$. □

Finally, we discuss the implementation of the algorithm in MapReduce:

Corollary 5.3. *The Maximal Matching algorithm can be implemented in $O(logn)$ rounds when $\eta \geq 40n$ and in $3\lfloor c/\epsilon \rfloor$ rounds when $\eta = n^{1+\epsilon}$.*

Proof. We will show that each round of the algorithm can be implemented in 3 MapReduce rounds, satisfying all memory constraints with high probability. In the first round we sample the set E' (Map stage) and compute M' on it (Reduce stage). It remains to show how to compute $G[I] = (I, E[I])$. To do so, we apply Lemma 2.2. Our universe is $\mathcal{U} = E$, $k = n$ and the set S_i contains the edges incident on vertex i. The function f_i discards the edges in S_i if vertex i is matched. The functions f_i are trivially MRC-parallelizable, and the computation will require an additional 2 rounds according to the lemma. □

It is worth noting that our choice of the sampling probability p was critical in the success of the algorithm. On the one hand, we want p to be small enough so that the probability of failure is very small. This was achieved by guaranteeing that $E[|E'|] = \eta/10$, which is small and furthermore does not depend on S (making the analysis cleaner). On the other hand, we want p to be large enough so that the number of "surviving" edges shrinks fast: this was achieved by ensuring that $2n/p$, an upper bound (with high probability) on the number of "surviving" edges, was at least half as small as the current number of edges. Instrumental in both considerations was the fact that p is inversely proportional to the current number of edges $|S|$. It seems that this choice of p strikes just the right balance between low failure probability and small number of iterations.

Finally, an interesting side note is that the approach that we used to devise this algorithm (iteratively form partial matchings and discard the edges that conflict with them) was the main idea used by Israeli et al. [1] for a randomized algorithm that finds maximal matchings in the CRCW-PRAM model, in a paper which dates back to 1985. The CRCW-PRAM model allows multiple processors to share memory and perform Concurrent Reads and Writes. The way they form the matchings is different: at each stage, they find a sparse subgraph with maximum degree 2, they let each vertex in that subgraph randomly choose an incident edge, and then add to the matching those edges that were picked by both of their endpoints. After each partial matching is computed, they discard conflicting edges and iterate. They show that the expected number of phases of their algorithm is $O(log|E|)$.

6 Lessons Learned

The most important takeaway is that filtering is a powerful technique for solving graph problems, and that it is naturally expressible in MapReduce. In particular, we saw that filtering needs to harness both the sequential and parallel power of MapReduce. This was most crucial in the filtering algorithm for computing a maximal matching, where we overcame the hurdle of conflicting matchings by enforcing coordination in a sequential fashion. Another idea worth holding is the tradeoff between MapReduce rounds and amount of memory per machine. We saw this in several algorithms, and it teaches us that exploiting the available memory as much as possible can pay off a lot in terms of rounds. Finally, a worthy lesson is that even the simplest algorithms and algorithms that are old or come from different models can contain the main ideas for new of more complicated ones. We saw this when trying to devise a filtering algorithm for maximal matching: generalizing the greedy sequential algorithm led us to a good solution, which also appeared with a different flavor in the CRCW-PRAM model a long time ago.

7 References

[**1**] Amos Israel and A. Itai. 1986. A fast and simple randomized parallel algorithm for maximal matching. Inf. Process. Lett. 22, 2 (February 1986), 77-80.

[**2**] Anand Rajaraman and Jeffrey David Ullman. 2011. Mining of Massive Datasets. Cambridge University Press, New York, NY, USA.

[**3**] Foto N. Afrati and Jeffrey D. Ullman. 2010. Optimizing joins in a map-reduce environment. In Proceedings of the 13th International Conference on Extending Database Technology (EDBT '10), Ioana Manolescu, Stefano Spaccapietra, Jens Teubner, Masaru Kitsuregawa, Alain Leger, Felix Naumann, Anastasia Ailamaki, and Fatma Ozcan (Eds.). ACM, New York, NY, USA, 99-110.

[**4**] Howard Karloff, Siddharth Suri, and Sergei Vassilvitskii. 2010. A model of computation for MapReduce. In Proceedings of the twenty-first annual ACM-SIAM symposium on Discrete Algorithms (SODA '10). Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 938-948.

[**5**] Jeffrey Dean and Sanjay Ghemawat. 2004. MapReduce: simplified data processing on large clusters. In Proceedings of the 6th conference on Symposium on Opearting Systems Design and Implementation - Volume 6 (OSDI'04), Vol. 6. USENIX Association, Berkeley, CA, USA, 10-10.

[**6**] Jure Leskovec, Jon Kleinberg, and Christos Faloutsos. 2005. Graphs over time: densification laws, shrinking diameters and possible explanations. In Proceedings of the eleventh ACM SIGKDD international conference on Knowledge discovery in data mining (KDD '05). ACM, New York, NY, USA, 177-187.

[**7**] Serge Plotkin. Optimization Paradigms, Lecture Notes. 2014. Stanford University.

[**8**] Silvio Lattanzi, Benjamin Moseley, Siddharth Suri, and Sergei Vassilvitskii. 2011. Filtering: a method for solving graph problems in MapReduce. In Proceedings of the twenty-third annual ACM symposium on Parallelism in algorithms and architectures (SPAA '11). ACM, New York, NY, USA, 85-94.