DIPARTIMENTO DI INFORMATICA
E SISTEMISTICA ANTONIO RUBERTI

SAPIENZA
UNIVERSITÀ DI ROMA

Large-Scale Graph Biconnectivity in MapReduce

Giorgio Ausiello
Donatella Firmani
Luigi Laura
Emanuele Paracone

Technical Report n. 4, 2012

Large-Scale Graph Biconnectivity in MapReduce

Giorgio Ausiello, Donatella Firmani and Luigi Laura
Dep. of Computing and System Sciences
"Sapienza" Univ. of Rome
{ausiello,firmani,laura}@dis.uniroma1.it

Emanuele Paracone
Dep. of Computer Science, Systems and Production
"Tor Vergata" Univ. of Rome
emanuele.paracone@gmail.com

Abstract

The MapReduce framework, originally proposed by Google [8], and its open source implementation, Hadoop [27], are nowadays considered the standard frameworks, both in industry and academia, to deal with petabyte scale datasets. In this paper we describe a two-rounds MapReduce approach to *biconnectivity* in undirected graphs, that is the computation of the set of articulation points, the set of bridges and the set of biconnected components of a graph G. We recall that an articulation point (resp. a bridge) is a vertex (resp. an edge) whose removal increases the number of connected components. A biconnected component is a maximal biconnected subgraph, i.e., it does not include neither articulation points nor bridges.

In order to minimize the communication cost, the algorithm is based on a *summary* of the input data set, that is a compact data structure from which queries on biconnectivity properties can be answered. This summary, called *navigational sketch* [3], was originally designed in the data streams framework [20] and was implicitly proved to be incrementally maintainable. Here we define it in a different framework in order to prove that it is *mergeable* [1]. Mergeability is the key property of summaries in distributed or parallel computation: in particular, it provides a way to split the computation of the summary across separate subsets of the original data set, and thus to exploit the parallelism of the MapReduce framework.

We finally discuss a scenario in which it is assumed that the machines have limited memory, showing tradeoffs between the memory available and the number of rounds of the algorithm. We conclude the paper with an experimental analysis that, on the basis of different executions of an Hadoop implementation of the algorithm against large-scale real world graphs, confirms the effectiveness of our approach.

Keywords: Articulation Points, Bridges, Biconnected Components, Map Reduce, Data Streaming.

1 Introduction

In recent years, we have witnessed an incredible growth of the available data to be processed, and in this scenario the MapReduce framework proposed by Google [8], together with its open source implementation, Hadoop [27], emerged as the *de facto* standard framework, both in industry and academia, to deal with petabyte scale datasets [17]. As an example, according to recent estimates, Facebook and Yahoo! process daily, respectively, 80 terabytes [21] and 120 terabytes [15] in their Hadoop clusters, whilst Google reaches the astonishing rate of 20 petabytes per day in its MapReduce clusters [9]. Just few month ago Google reported that they succeded in sorting 10 petabytes of data in 33 minutes, using 8000 machines [7].

The diffusion of this framework naturally has led to the development of algorithmic tools able to efficiently solve problems in this new MapReduce paradigm of computation, that, roughly speaking, alternates a (massively) parallel phase (the mappers) with a sequential one (the reducers); in order to properly evaluate these new algorithms, distinct theoretical models have been presented in the literature [16, 11, 13]. However, so far in the (MapReduce) literature relatively few works appeared dealing with graph problems, despite the fact that also the size of available graph data increased consistently in recent years. Indeed, samples of the Webgraph, i.e. the graph whose nodes are webpages and whose (directed) edges are the links between them, nowadays can easily have billions of nodes, and there are several other graph data types available including call graphs, citation graphs, and social networks: for example, only few month ago the media gave a huge coverage to the *four degrees of separation* in Facebook observed by Backstrom et al. [4]. The authors analyzed the entire Facebook network, consisting in approximately 721 millions users and 69 billions links.

In this work we add to this list of graph algorithms in MapReduce a very basic tool for the structural analysis of a graph: the computation of all the biconnectivity properties of a graph, i.e. biconnected components, bridges, and articulation points.

1.1 Contributions

In this paper we present a two-rounds MapReduce algorithm, based on the navigational sketch of a graph G, i.e. $\texttt{ns}(G)$, originally introduced in [3]. In Section 3, we prove that the navigational sketch is a fully mergeable [1] summary, and this will be the key property in the design of our algorithm. Furthermore, we extend the notion of full mergeability, defining a property, called shuffleable mergeability, that allows the summary to be easily computed in a multi-rounds MapReduce schema. Roughly speaking, as the name suggest, in a MapReduce context, the shuffleable mergeability allows the summary, computed in a single machine, to be broken in smaller pieces, in such a way that the pieces can more easily arranged to fit in the available memory. This theoretical results build up on the *cactus representation* of a navigational sketch, defined in Section 2.

In Section 4 we describe a two-rounds MapReduce algorithm, analyzed in the

formal model proposed by Karloff et al. [16], and in case of limited memory scenario, we propose a multi-rounds algorithm. We conclude with a brief experimental evaluation, that confirms the effectiveness of our approach.

1.2 Related Work

If we consider the traditional model of computation it is possible to compute articulation points, bridges, and biconnected components of an undirected graph with a simple Depth First Search (DFS) based algorithm, as shown, for example, in the classical textbook by Cormen et al. [6]. Tarjan and Vishkin [24] addressed the biconnected components computation in the CRCW PRAM model. In an online setting the problem has been studied by Westbrook and Tarjan [26], while more recently Ausiello et al. proposed an algorithm in the (semi-)streaming model of computation [3].

The MapReduce framework has been presented in the work of Dean and Ghemawat [8]. So far, few graph problems have been addressed in this framework: Lin and Schatz [19] showed how to implement message passing style algorithms in MapReduce, thus obtaining algorithms that run in $O(d)$ rounds, where d is the diameter of the graph. Suri and Vassilvitskii [23] designed an algorithm to count triangles in massive graphs; Karloff et al. [16] show how to compute the minimum spanning tree (MST) and undirected $s - t$ connectivity; Lattanzi et al. [17] proposed a general technique for graph problems, called *filtering*, and showed how to use this technique to compute the MST, maximal matchings, approximate weighted matchings, approximate vertex and edge covers and minimum cuts.

As previously mentioned, so far three models of computation for MapReduce have been presented, respectively, by Feldman et al. [11], Karloff et al. [16], and Goodrich [13]; in these works the authors show, in different ways, that a large class of PRAM algorithms can be efficiently simulated via MapReduce. Even if the relationship between $\mathcal{NC}$ class and the class of problems efficiently solvable in MapReduce is still a partially open problems, those results provide significative upper bounds on the number of rounds for a large set of problems in the MapReduce framework: Circuit Padding, Triangle Counting, Minimum Spanning Tree of an undirected graph, and Connected Components, just to name a few. Moreover, some of them have been studied in MapReduce, achieving better performances than the PRAMs simulation under certain assumptions: as an example the simulations provide a $\Omega(\log n)$ upperbound for the Minimum Spanning Tree problem (and thus Connected Components), that has been improved for dense graphs with constant-rounds algorithms [16, 17], while for sparse graphs the problem is still open, as pointed our recently by Vassilvitskii [25].

An overview of MapReduce. We briefly recall the feature of the MapReduce computing paradigm (see [8, 9] for details). The computation proceeds in rounds; in each round there are three distinct phases: map, shuffle, and reduce. The shuffle phase is operated by the MapReduce system, and therefore, in order to write a

MapReduce program, a programmer needs only to define a *mapper* and a *reduce* function [16]:

(1) **Mapper**: A mapper is a function that takes as input an ordered $\langle key; value \rangle$ pair of binary strings. As output the mapper produces a finite multiset of new $\langle key; value \rangle$ pairs.

(2) **Reducer**: A reducer is a function that takes as input a binary string k which is the key, and a sequence of values $v_1, v_2, \ldots$ which are also binary strings. As output, the reducer produces a multiset of pairs of binary strings $\langle k; v_{k,1} \rangle$, $\langle k; v_{k,2} \rangle \ldots$ having the same key k.

Therefore, a MapReduce algorithm is defined as follows. FOR $r = 1, 2, ..., R$ DO:

1. EXECUTE MAP. Feed each pair $\langle k; v \rangle$ in U_r to mapper μ_r. The mapper will generate a sequence of tuples $\langle k_1; v_{k,1} \rangle$, $\langle k_2; v_{k,2} \rangle \ldots$. Let U'_r be the multiset of $\langle key; value \rangle$ pairs output by μ_r.

2. SHUFFLE. For each k, let $V_{k,r}$ be the multiset of values v_i associated to the same key k in U'_r. The underlying MapReduce implementation constructs the multisets $V_{k,r}$.

3. EXECUTE REDUCE. For each k, feed k and some arbitrary permutation of $V_{k,r}$ to a separate instance of reducer ρ_r, and run it. The reducer will generate a sequence of tuples $\langle k; v_1 \rangle$, $\langle k; v_2 \rangle \ldots$. Let U_r be the multiset of $\langle key; value \rangle$ pair output by all instances of ρ_r

2 Cactus representation

The *navigational sketch* of a graph has been introduced, in a streaming context, in the work of Ausiello et al. [3]. We define it formally in the next section, where we show that it is a fully mergeable summary [1]. Roughly speaking, looking also at Figure 1, we can derive a navigational sketch of a connected graph in the following way: given the graph G, for each biconnected components bcc in G, replace all the edges between vertices in bcc with a tree made of uniquely colored edges: one (any) vertex in bcc is the root of the tree, and all the other vertices are the leaves. Therefore, we represent each biconnected components with a uniquely colored tree, as shown in Figure 1. All the bridges of G are replaced by *solid* edges, i.e. edges with no color.

We recall that a *cactus graph* is a connected graph in which any two simple cycles have at most one vertex in common. Let us now introduce the cactus representation of a navigational sketch, also shown in Figure 1, that is defined as follows.

Definition 2.1 (Cactus Representation) *The cactus representation of a navigational sketch $N = (V, E)$, denoted as* `cactus(`N`)`*, is a cactus graph $C = (V, E')$,*

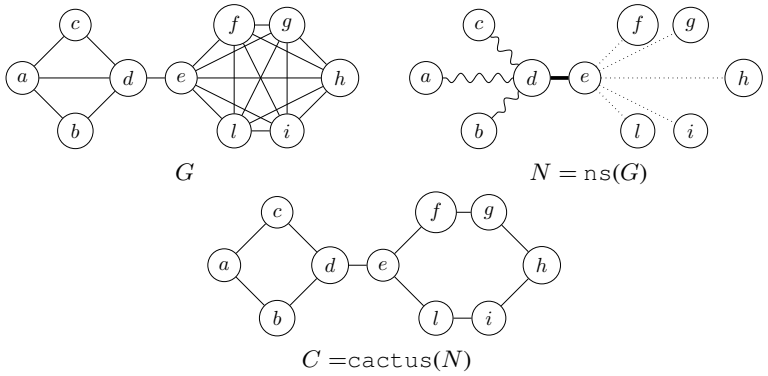

Figure 1: An example of a graph G, its navigational sketch $N = \texttt{ns}(G)$, and the corresponding cactus representation $C = \texttt{cactus}(N)$. In the navigational sketch, we represent the solid edges with the line style ▬, and the colored edges with line styles ····· and ∿.

where there is a simple cycle containing all and only the vertices connected by edges of the same color, for each color of the edges, and there is an edge for each solid edge of N.

A relevant property of any cactus graph is a linear bound on the number of edges with respect to the number of vertices, as stated in the following theorem.

Theorem 2.2 *Given a cactus graph G with n vertices and m edges, $m \leq \gamma(n-1)$, with $\gamma = \frac{3}{2}$.*

Proof. Let G be the cactus graph. Now, let us remove each edge $e = (v_1, v_2)$ that does not belong to any cycle by contracting v_1 and v_2 into a single vertex. Let us denote by h the number of edges removed, and by G' the obtained graph; it holds that $m' = m - h$ and $n' = n - h$.

Now consider the graph H, whose vertices are the biconnected components of G', and there is an edge between two vertices if the corresponding biconnected components share a vertex. Note that, by construction, H is a chordal graph. Let us consider a simplicial vertex v in H; we recall that Dirac proved that every chordal graph has a simplicial vertex [10]. In G' there is a simple cycle C, corresponding to v, that has only one vertex in common with edges outside C. Let us contract all the vertices in C into a single one, let G'' be the resulting graph, and note that it holds $m'' = m' - |C|$ and $n'' = n' - |C| + 1$, and there is a chordal graph $H' = H \backslash v$ that corresponds to G''. We can repeat the process until there is only one cycle[1]; in this last step we remove $|C|$ vertices and $|C|$ edges. Let us denote with k the number of simple cycles in the original graph, and by $\overline{C}$ the average number of vertices in each cycle, i.e. $\overline{C} = \sum_1^k C_i/k$. It holds $m = h + k \cdot \overline{C}$, and $n = h + k \cdot \overline{C} - (k-1)$.

[1]Stated in other words, we follow a *perfect elimination order* [12] to remove the vertices from the chordal graph H.

Let us note that, for any cycle C, it holds $|C| \geq 3$, thus implying $\overline{C} \geq 3$. It is easy to see that, with $h \geq 0$ and $\overline{C} \geq 3$, it holds that $m \leq 3/2(n-1)$. □

3 Navigational Sketch

In this section we describe the *navigational sketch* data structure, that represents the biconnectivity properties of an undirected graph, with space sub-linear in the size of the graph, i.e., the number of edges. We require, without loss of generality, that G is connected, and we define the navigational sketch as follows.

Definition 3.1 *Given a connected graph $G = (V, E)$, its navigational sketch is a tree $N = (V_N, E_N)$, where the set of nodes is the same of G, i.e., $V_N = V$, and the set of edges E_N is distinguished in two types:* solid *and* colored *edges. The following properties hold:*

1. BR. *the* bridges *of G are the* solid *edges of N;*
2. BCC. *the* biconnected components *of G are represented with a subtree, inside a tree in N, with one father and $b-1$ children (where b is the cardinality of the biconnected component); all the edges in the subtree are of the same color, and this color is unique inside N.*

All navigational sketches $N_1, \ldots, N_h$ of a graph $G = (V, E)$, with $|V| = n$, have also these properties:

1. SIZE. The size of each navigational sketch N_i is $O(n)$, since N_i is a tree.
2. AP. Two edges $\langle u_1, v\rangle, \langle u_2, v\rangle$ in N_i have different colors or at least one of them is solid, if and only if v is an articulation point of G.
3. UNIQUENESS OF CACTUS REPRESENTATION. Any pair of navigational sketches N_1, N_2 of G has the same cactus representation C, i.e., $\texttt{cactus}(N_1) = \texttt{cactus}(N_2)$.
4. REFLEXIVITY. Any navigational sketch N_i of G is a navigational sketch of its cactus representation C.

Furthermore, the definition of navigational sketch can be also generalized to graphs containing h connected component $G = \{G_1, G_2, \ldots, G_h\}$, with $h \geq 2$, and it is a forest containing h navigational sketches, each of them representing a connected component of G, and all the above properties still hold. In particular SIZE is verified since the size of the navigational sketch, in this case, is $O(n-h) = O(n)$.

Computation of a navigational sketch. The function $\texttt{ns}(G)$, that finds a navigational sketch of a graph G, can be computed using the streaming [20] algorithm described in [3]. The algorithm finds a navigational sketch N using $O(n)$ memory, if implemented by a RAM with $O(\log n)$-length words, and $O(m\alpha(m,n))$ processing time if $m \geq n \log n$, where α is the the functional inverse of Ackermann's function. Westbrook and Tarjan proved in [26] that a $O(m\alpha(m,n))$ processing time algorithm is optimal in the online model of computation, therefore this also holds in the streaming model of computation.

As claimed in [1], a streaming algorithm with small space, as the proposed one for the computation of $\texttt{ns}(G)$, implies an *incrementally maintainable summary*. A summary σ on a data set S is any compact data structure from which certain queries on S can be answered accurately, while requiring much lower resources with respect to answering the same queries directly on S. When S is accessible in its entirety, the summary σ can be constructed off-line; more generally, the summary is supposed to be maintained in the presence of updates to S, especially when S is observed as a stream. As known so far from the work in [3], the navigational sketch is a summary for biconnectivity properties of a graph, and indeed represents an incrementally maintainable summary, but as discussed in [1], there could be need for stronger requirements on summaries where there is a process of repeatedly merging together two summaries of (separate) data sets to obtain a summary of their union. In the following we prove the *mergeability* of the navigational sketch and discuss how to speed-up the computation of $\texttt{ns}(G)$ switching to a parallel framework.

Speed-up: parallel computation of a navigational sketch. The streaming approach proposed in [3] takes indeed, as any streaming algorithm over a stream S of s items, $\Omega(s)$ time, that is, for a graph algorithm, linear time in the number of the edges of the graph G. We prove in the following, that the function $\texttt{ns}(G)$ can be parallelized and that it is possible to compute a navigational sketch of a graph $G = (V, E)$, with $|V| = n$ and $|E| = m$, in $o(m)$ time without increasing the memory requirement. In [1] the authors define two variants of *mergeability*, full and one-way mergeabiliy, the first of which is reported below. We denote as $\sigma(S, \epsilon)$ a summary on a data set S with approximation error ϵ.

Definition 3.2 *A summary $\sigma(S, \epsilon)$ is fully mergeable if the size $\sigma(S, \epsilon) \leq k(\frac{1}{\epsilon}, |S|)$ is bounded by an absolute function $k()$, and there exists an algorithm $\mathcal{A}$ that produces the summary on $\sigma(S_1 \uplus S_2, \epsilon)$[2] from any two input summaries $\sigma(S_1, \epsilon)$ and $\sigma(S_2, \epsilon)$.*

Theorem 3.3 *The navigational sketch is a fully mergeable summary $\sigma(S, \epsilon)$, where S is the set of edges of an undirected input graph G and $\epsilon = 0$. The size is sublinear in $|S|$, if $|S| = n^{1+d}$, where n is the number of vertices of G. The algorithm $\mathcal{A}$ that executes the merge step, can be any algorithm that, given two input navigational sketches N_1 and N_2, computes $\texttt{ns}(\texttt{cactus}(N_1) \cup \texttt{cactus}(N_2))$.*

[2] $\uplus$ denotes multiset addition

Proof. We assume without loss of generality, that the algorithm that computes the function $\texttt{ns}(G)$ incrementally maintains the navigational sketch, as the streaming algorithm in [3], and we denote with $\texttt{ns}(N, G')$ the navigational sketch N updated incrementally with G'. We denote with $u \leftrightarrow_G v$ that the vertices u, v lie in the same biconnected component of the graph G, and with $N \rightarrow u, v$ the fact that a navigational sketch N answers positively to a query "are u, v in the same biconnected component?".

Given a graph $G = G_1 \cup G_2$ and two input navigational sketches $N_1 = \texttt{ns}(G_1)$ and $N_2 = \texttt{ns}(G_2)$, by Def 2.1 and property BCC, $\texttt{cactus}(N_2)$ contains a cycle for all and only the biconnected components of G_2, therefore, for any pair of vertices $u, v \in G$, it holds that $N_1 \rightarrow u, v$ iff $u \leftrightarrow_{G_1} v$ and $u \leftrightarrow_{\texttt{cactus}(N_2)} v$ iff $u \leftrightarrow_{G_2} v$. Furthermore, if neither $u \leftrightarrow_{G_1} v$ or $u \leftrightarrow_{G_2} v$ are verified, it holds that if $u \leftrightarrow_G v$ therefore $\texttt{ns}(N_1, \texttt{cactus}(N_2)) \rightarrow u, v$ since N_1 is incrementally maintainable. Therefore, if $N = \texttt{ns}(N_1, \texttt{cactus}(N_2))$, N is a navigational sketch of G.

By property SIZE, $k(\frac{1}{\epsilon}, |S|) = n$, where n is the number of vertices of the input graph, and since the approximation error $\epsilon = 0$, the thesis follows. □

By Theorem 3.3 the navigational sketch can be merged in an arbitrary fashion for an indefinite number of steps, i.e., with any arbitrary computation trees [16, 11], and the size does not depend linearly on number of merges.

The mergeability is used, ad discussed in the following, as a basic block to speed-up the computation of a navigational sketch, but is not sufficient anymore in the limited memory MapReduce scenario described in Sec. 4. We will use a more general property, that we call *shuffleable mergeability*, that can be proved for the navigational sketch.

Lemma 3.4 *By Theorem 3.3, for any partition of the edges of a graph $G = \{G_1, \ldots, G_h\}$, where $\bigcup_i G_i = G$ and $G_i \cap G_j = \emptyset$ for any $i \neq j$, the function $\texttt{ns}(G) = \texttt{ns}(C_h)$, where $C_h = \bigcup_{i \in [h]} \texttt{cactus}(\texttt{ns}(G_i))$*[3].

Computing $\texttt{ns}(C_h)$ is never more time consuming than computing $\texttt{ns}(G)$, since C_h is smaller than G (by The. 2.2, $\texttt{cactus}(\texttt{ns}(G_i))$ is smaller than G_i up to a factor $\gamma = \frac{3}{2}$) as formalized in the lemma below.

Lemma 3.5 *For any edge partition of an undirected graph $G = \{G_1, \ldots, G_h\}$ with n vertices, where $\bigcup_i G_i = G$ and $G_i \cap G_j = \emptyset$ for any $i \neq j$, the size of $\bigcup_{i \in [h]} \texttt{cactus}(\texttt{ns}(G_i))$ is $O(hn)$, that is $O(n)$ for a constant value of h.*

Lemmas 3.4 and 3.5 describe an algorithm (see Alg. 1) to compute $\texttt{ns}(G)$ efficiently exploiting parallelism on a partition of G: $\texttt{ns}(G_1), \ldots, \texttt{ns}(G_h)$ can be computed in parallel and $\texttt{ns}(C_h)$ still requires $O(n)$ memory, while only $O(n \log n)$ processing time. If Alg. 1 is analyzed in the CRCW PRAM model, the **for all** section is executed in $O(m' \alpha(m', n))$ time, where $m' = \max_{i \in [h]} |E_i|$. If it holds $m' = m^{1-\epsilon} \geq n \log n$, for any $\epsilon > 0$, Alg. 1 takes sub-linear processing time in the number of the edges of the graph G, i.e., $o(m)$.

[3] $[h]$ denotes the set of integers ≥ 1 and $\leq h$

Alg. 1 A CRCW PRAM algorithm to compute $\texttt{ns}(G)$.

function streaming_$\texttt{ns}(G)$
(see [3])
1: $NS \leftarrow \emptyset$
2: **for** $\langle u, v\rangle \in G$ **do**
3: **if** $u, v \notin$ the same tree $T \subseteq NS$ **then**
4: $NS \leftarrow \langle u, v\rangle$ and $\ell(\langle u, v\rangle)$ n.d.
5: **else**
6: $P_{u,v} \leftarrow$ shortest $u - v$ path in NS
7: **if** $\forall \langle w, z\rangle \in P_{u,v}, \ell(\langle u, v\rangle) = l$ **then**
8: **break**
9: **else**
10: modify T as in Alg.2 of [3]
11: **end if**
12: **end if**
13: **end for**
14: **return** NS

function $\texttt{ns}(G)$
1: given a partition $G = \{G_1, \ldots, G_h\}$
2: $C_h \leftarrow \emptyset$
3: **for all** $i \in [h]$ **do**
4: $C_h \leftarrow C_h \cup$ $\cup\, \texttt{cactus}$(streaming_$\texttt{ns}(G_i)$)
5: **end for**
6: $NS \leftarrow$ streaming_$\texttt{ns}(C_h)$
7: **return** NS

4 Biconnectivity in the MapReduce framework

In this section we describe a two-rounds MapReduce algorithm to compute a navigational sketch of an undirected graph G, we analyze it in the $\mathcal{MRC}$ model [16], discuss a limited memory scenario, and conclude with a brief experimental evaluation using Hadoop [27]. As introduced in Sec. 1.2, it is possible to compute biconnected components in MapReduce by simulating the PRAM algorithms, however, this leads to an algorithm that runs in $O(\log n)$ rounds, as we show below.

PRAM Simulation via MapReduce. Tarjan and Vishkin [24] designed a parallel algorithm on CRCW PRAM for computing biconnected components in a connected undirected graph, and runs in $O(\log n)$ time, using $O(n + m)$ space and $O(n + m)$ processors. Theorem 5.1 in [13] provides a simulation via memory-bound MapReduce framework, that runs in $O(\log n \log_B(n + m))$ rounds with a constant space B and $O(\log n(n + m) \log_B(n + m))$ message complexity. The simulation leads to a $O(\log n)$ rounds upperbound for the biconnectivity problem in MapReduce.

4.1 A two-rounds MapReduce Algorithm

In this section we describe a two-rounds MapReduce algorithm. This algorithm, described in Alg. 2, is an implementation of Alg. 1 in the MapReduce framework: given a positive integer h and an undirected graph G with n vertices and m edges, represented by a sequence of pairs $\langle i, \text{edge}_i\rangle$ where $i = 1 \ldots m$, the first round of Alg. 2 computes $C_h = \bigcup_{i\in[h]} \texttt{cactus}(\texttt{ns}(G_i))$ using h reducer instances, respectively computing $\texttt{cactus}(\texttt{ns}(G_i))$ with any implementation of the $\texttt{ns}$ function,

while the map stage of the second round maps each edge of C_h to a single reducer instance that can therefore compute $\texttt{ns}(C_h) = \texttt{ns}(G)$.

Alg. 2 A MapReduce algorithm to compute $\texttt{ns}(G)$. G is represented by a sequence of pairs $\langle i, \text{edge}_i \rangle$.

ROUND 1: Find $C_h = \bigcup_{i \in [h]} \texttt{cactus}(\texttt{ns}(G_i))$

function MAP $\langle i, \langle u, v \rangle \rangle$
1: $j \leftarrow hash(i)$ (given $hash : [m] \to [h]$)
2: **return** $\langle j, \langle u, v \rangle \rangle$

function REDUCE $\langle i, G_i \rangle$
1: $G'_i \leftarrow \texttt{cactus}(\texttt{ns}(G_i))$
2: **return** $\langle i, G'_i \rangle$

ROUND 2: Compute $\texttt{ns}(C_h) = \texttt{ns}(G)$

function MAP $\langle i, \langle u, v \rangle \rangle$
1: \$ is a special symbol
2: **return** $\langle \$, \langle u, v \rangle \rangle$

function REDUCE $\langle \$, G_\$ \rangle$
1: $NS \leftarrow \texttt{ns}(G_\$)$
2: **return** $\langle \$, NS \rangle$

In a recent work [17] Lattanzi et al. describe a general algorithmic design technique in the MapReduce framework called *filtering*. While the main idea behind the algorithm described in Alg. 2 can be thought of as a filtering approach, we remark that the navigational sketch (and its cactus representation) is a summary of the input graph and while its size is smaller as in the filtering approach, the edge set is not a subset of the input graph.

4.2 Theoretical Analysis

We now analyze the algorithm in the $\mathcal{MRC}$ model of computation of MapReduce, summarized in Def. 4.1.

Definition 4.1 ([16]) *Given a finite sequence of pairs $\langle k_i; v_i \rangle$ where k_i and v_i are binary strings and $\sum_i |k_i| + |v_i| = s$, fix $\epsilon > 0$. An algorithm in $\mathcal{MRC}^i$ consists of a sequence of map and reduce operations which outputs the correct answer with probability at least $\frac{3}{4}$ where:*

1. *Each map or reduce operation is implemented by a RAM with $O(\log s)$-length words, that uses $O(s^{1-\epsilon})$ memory and time polynomial in s.*
2. *The total space used by the $\langle key; value \rangle$ pairs output by each map stage is $O(s^{2-2\epsilon})$.*
3. *The number of rounds is $O(\log^i s)$*

There exists Las Vegas and deterministic variants of $\mathcal{MRC}$, the latter of which are called $\mathcal{DMRC}$.

Given $\epsilon > 0$ and a graph $G = (V, E)$ where $|V| = n$ and $|E| = m$ represented by a sequence of pairs $\langle i, \text{edge}_i \rangle$ where $\sum_i |i| + |\text{edge}_i| = s$, in order to prove that the algorithm described in Alg. 2 is in $\mathcal{MRC}^0$, we set $h = m^\epsilon$ and we observe

that (i) by Lemma 3.5, the total space used by the $\langle key; value \rangle$ pairs output by each map stage is $O(m^\epsilon n) = O(s^{2-2\epsilon})$; (ii) the number of rounds is $O(1)$. Since for each map or reduce operation, implemented by a RAM, the processing time is $O(\text{poly}(m)) = O(\text{poly}(s))$, the following fact implies that Alg. 2 lies in $\mathcal{MRC}^0$.

Lemma 4.2 *Let $h = m^\epsilon$, with high probability the size of every $G_i = O(m^{1-\epsilon})$, therefore each map or reduce operation requires $O(s^{1-\epsilon})$ memory.*

Proof. If $hash : [m] \to [h]$ is a uniformly random function, in expectation the size of G_i, i.e., the number of edges mapped to a key i, is $\frac{m}{h}$. Since $h = m^\epsilon$, the size of every G_i is $O(m^{1-\epsilon})$ with high probability. Since $m \leq s$, the thesis follows. □

The classic sequential algorithm for computing biconnected components in a connected undirected graph due to Hopcroft and Tarjan [14] runs in linear time in the number of edges of the input graph, and is based on depth-first search. In Sec. 3 we showed that Alg. 1 leads to an effective speed-up with respect to the streaming and the classic approaches; here, in order to compare Alg. 2 with the classic approach, we use the definition of *total work* in Sec. 2.2 of [17].

We now prove that, if $O(m + m^\epsilon n) = O(m)$, for $\epsilon > 0$, the $\mathcal{MRC}$ algorithm in Alg. 2 is *work efficient*, that is, its total work matches the running time of the best known sequential algorithm.

Lemma 4.3 *Using the classic sequential algorithm as a basic block of the implementation of the navigational sketch function, the total work needed by the $\mathcal{MRC}^0$ algorithm is $O(m + m^\epsilon n)$.*

Proof. During the first round, partitioning the input graph G into $\{G_1, \ldots, G_h\}$ requires a linear scan over the edges which is $O(m)$ work. Computing the cactus representation of the navigational sketch of each part of the partition using the classic sequential algorithm as a basic block takes $O(m)$ work. Computing the navigational sketch on one machine using the same algorithm requires $O(m^\epsilon n)$ work. □

4.3 Limited memory scenario.

Given an input graph $G = (V, E)$, where $|V| = n$ and $|E| = m$, we now discuss the scenario in which the machines have some limited memory $k > n$, and that the number of available machines is $\frac{m}{k} \geq m^\epsilon$ in order to admit a solution in $\mathcal{MRC}$. If the available memory is $O(m^{1-\epsilon})$, we have seen in previous section an easy way to design a two-rounds MapReduce algorithm (Theorem 3.3). We now define a property, *shuffleable mergeability*, that provides a way to design a multi-rounds MapReduce algorithm working with memory smaller than k.

Definition 4.4 *A summary $\sigma(S, \epsilon)$ is f-shuffleble if the size $\sigma(S, \epsilon) \leq k(\frac{1}{\epsilon}, |S|)$ is bounded by an absolute function $k()$, and there exists an algorithm $\mathcal{A}$ that produces data set S' such that $\sigma(S, \epsilon) = \sigma(S', f(\epsilon))$ and the size $|S'|$ is bounded by $k()$.*

If $f(\epsilon) = \epsilon$, the summary is also fully mergeable: the algorithm that produces the summary on $\sigma(S_1 \uplus S_2, \epsilon)$ from any two input summaries $\sigma_1 = \sigma(S_1, \epsilon)$ and $\sigma_2 = \sigma(S_2, \epsilon)$, is given by $\sigma(\mathcal{A}(\sigma_1) \uplus \mathcal{A}(\sigma_2), \epsilon)$.

Theorem 4.5 *The navigational sketch is a ϵ-shuffleable summary $\sigma(S, \epsilon)$, where S is the set of edges of an undirected input graph G and $\epsilon = 0$. The size is sublinear in $|S|$, if $|S| = n^{1+d}$, where n is the number of vertices of G. The algorithm $\mathcal{A}$ can be any algorithm that, given a navigational sketch N, computes $\texttt{cactus}(N)$.*

Proof. The proof holds by the properties REFLEXIVITY of a navigational sketch. To verify the bound on the size of the data set produced by the cactus representation operator, see Lemma 2.2 and property SIZE of a navigational sketch. □

We now describe a multi-rounds MapReduce algorithm to compute a navigational sketch of an undirected graph G. The algorithm, described in Alg. 3, is a variation of Alg. 2. Let $G = \mathcal{G}_0$ be the input graph and $\mathcal{G}_r = (V, E_r)$, where $|E_r| = m_r$, be the input of r-th round of Alg. 3: the r-th round computes $\mathcal{G}_r = \bigcup_{i \in [\frac{m_r}{k}]} \texttt{cactus}(\texttt{ns}(G_i))$ (where G_i is the i-th subgraph of the partition of $\mathcal{G}_{r-1} = \{G_1 \dots G_h\}$) using any implementation of the navigational sketch function, and when the size of $\mathcal{G}_{r-1}$ is smaller than k, that is each edge of $\mathcal{G}_{r-1}$ has been mapped to a single reducer instance, computes $\texttt{ns}(\mathcal{G}_{r-1}) = \texttt{ns}(G)$.

Alg. 3 A multi-rounds version of Alg. 2 that works in a limited memory scenario.

ROUND i: Compute $\texttt{ns}(C_h) = \texttt{ns}(G)$
$C_h = \bigcup_{i \in [h]} \texttt{cactus}(\texttt{ns}(G_i))$

Given $hash : [m] \to [\lceil \frac{m}{k} \rceil]$
and $hash(i) = \lceil \frac{i}{k} \rceil$

function MAP $\langle i, \langle u, v \rangle \rangle$
1: $j \leftarrow hash(i)$
2: **return** $\langle j, \langle u, v \rangle \rangle$

function REDUCE $\langle i, G_i \rangle$
1: **if** $\lceil \frac{m}{k} \rceil = 1$ **then**
2: $NS \leftarrow \texttt{ns}(G_i)$
3: **return** $\langle i, NS \rangle$
(and end the computation)
4: **else**
5: $G'_i \leftarrow \texttt{cactus}(\texttt{ns}(G_i))$
6: **return** $\langle i, G'_i \rangle$
(and start another round)
7: **end if**

The function $hash : [m_r] \to [\lceil \frac{m_r}{k} \rceil]$ executed in the map stage, partitions the graph $\mathcal{G}_r$ in $h = \lceil \frac{m_r}{k} \rceil$ subgraphs, not uniformly random. In order to prove the bound on the numbers of rounds of the Alg. 2, this particular approach guarantees that each reduce instance will use exactly k memory, except for a single one that is uses instead $m_r \mod k$ memory when $m_r \mod k > 0$ (otherwise this instance does not exist).

Lemma 4.6 *Given a graph $G = (V, E)$ where $|V| = n$ and $|E| = m$, with high probability the number of rounds is $O(\log_{\frac{k}{n^\gamma}} m)$.*

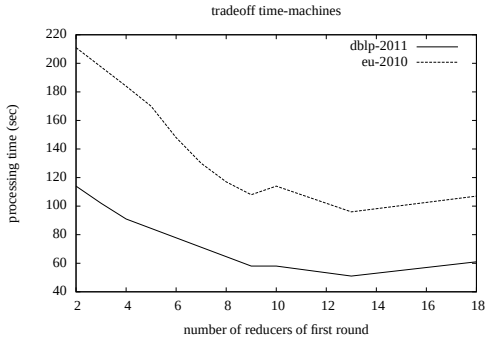

Figure 2: In this plot, the running time (in seconds) of the algorithm versus the number of machines (i.e. the reducers used in the first round).

Proof. Because of the above partition schema (i) no machine gets assigned more than k edges; (ii) at each round any reduce operation returns a cactus representation that is strictly smaller than k unless $k = n$ (but we know that $k > n$), i.e., $m_r < m_{r-1} < k$, in particular $m_r < \frac{m_{r-1}}{k}\gamma n$ where γ is the constant value defined in Sec. 2. Therefore, the size of the input graph decreases, each round, up to a factor $\frac{n}{k}\gamma$ with respect to the number of vertices n of G, and after $O(\log_{\frac{k}{n\gamma}} m)$ iterations the input graph $\mathcal{G}_r$ is small enough to fit onto a single machine, and the overall algorithm terminates. □

Lemma 4.7 *Given a graph $G = (V, E)$ where $|V| = n$ and $|E| = n^{1+d}$, if $k = n^{1+d'}$, with $d' \leq d$, with high probability the number of rounds is $O(\frac{d}{d'})$.*

4.4 Experimental Validation in Hadoop

In the following we briefly describe a preliminary experimental validation of the algorithm. A detailed experimental analysis goes behind the scope (and the size) of this paper; our goal here is to confirm the effectiveness of our approach.

Experimental settings. We used two graphs available online[4]: i) *dblp-2011* (with $n = 986k$, $m = 6.7M$): a co-citation graph, with data from the DBLP Computer Science Bibliography [18]; ii) *eu-2005* (with $n = 862k$, $m = 19.2M$): a 2005 crawl of the *.eu* domain [5]. We implemented our algorithms adapting the Java code provided with the textbook written by Sedgewick [22]. We rented (up to) 18 identical machines from the Amazon Elastic Computing Cloud (EC2) [2], *Large Standard Instance* type: 7.5 GB of memory, 4 EC2 Compute Units (2 virtual cores with 2 EC2 Compute Units each), 850 GB of local instance storage, 64-bit platform. Hadoop version is 0.20.203.0.

[4]The datasets are available at the url `http://law.dsi.unimi.it/`.

Overall processing time. In Figure 2 we can see how the overall processing time is affected by the number of reducers in the first round, i.e. the overall number of machine. It is clear from the plots that there is an optimal number of machines, i.e. both the plots have a minimum. This is interesting: after a certain threshold, adding more machines slows down the overall computation. This result can be easily explained: assume we have h machines; since the algorithm to compute the navigational sketches is linear in the input, we know that, in the first round each of the reducers has to process roughly m/h edges, producing a sketch whose size is $\approx n$. In the second round, the single reducer has to process $\approx h \cdot n$ edges; the overall processing time for a graph G is therefore $T(G) \approx m/h + h \cdot n$. We can derive the equation with respect to h, in order to compute the value h^* that minimizes $T(G)$, obtaining $dT/dh = n - m/h^2$, and solving for h, it holds $h^* = \sqrt{m/n}$. Therefore, the (theoretical) optimal number of machines is approximately the square root of the average degree of the graph. Our experiments confirmed this behavior but, in practice, we needed more machines than expected: the values computed analytically are, indeed, $h^* = 3$ for the *dblp-2011* graph, and $h^* = 5$ for the *eu-2005*.

References

[1] Pankaj K. Agarwal, Graham Cormode, Zengfeng Huang, Jeff M. Phillips, Zhewei Wei, and Ke Yi. Mergeable summaries. In *3rd Workshop on Massive Data Algorithmics (MASSIVE)*, 2011.

[2] Amazon. Amazone elastic cloud computing. `http://aws.amazon.com/ec2/`.

[3] Giorgio Ausiello, Donatella Firmani, and Luigi Laura. Real-time monitoring of undirected networks: Articulation points, bridges, and connected and biconnected components. *Networks,* to appear, 2012. A preliminary version is available online at `http://www.dis.uniroma1.it/~laura/papers/pdf/AFL-Networks.pdf`.

[4] Lars Backstrom, Paolo Boldi, Marco Rosa, Johan Ugander, and Sebastiano Vigna. Four degrees of separation. *CoRR*, abs/1111.4570, 2011.

[5] P. Boldi and S. Vigna. The webgraph framework I: Compression techniques. In *Proceedings of the International Conference on World Wide Web (WWW)*, pages 595–602. ACM, 2004.

[6] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms, Third Edition*. The MIT Press, Cambridge, Massachusetts, United States, 2009.

[7] Grzegorz Czajkowski, Marin Dvorsk, Jerry Zhao, and Michael Conley. Sorting petabytes with mapreduce - the next episode.

http://googleresearch.blogspot.com/2011/09/sorting-petabytes-with-mapreduce-next.html.

[8] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: simplified data processing on large clusters. In *Proceedings of the 6th conference on Symposium on Opearting Systems Design & Implementation - Volume 6*, pages 10–10, Berkeley, CA, USA, 2004. USENIX Association.

[9] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: simplified data processing on large clusters. *Commun. ACM*, 51:107–113, January 2008.

[10] G. Dirac. On rigid circuit graphs. *Abhandlungen aus dem Mathematischen Seminar der Universitt Hamburg*, 25:71–76, 1961. 10.1007/BF02992776.

[11] Jon Feldman, S. Muthukrishnan, Anastasios Sidiropoulos, Cliff Stein, and Zoya Svitkina. On distributing symmetric streaming computations. In *Proceedings of the nineteenth annual ACM-SIAM symposium on Discrete algorithms*, SODA '08, pages 710–719, Philadelphia, PA, USA, 2008. Society for Industrial and Applied Mathematics.

[12] D.R. Fulkerson and O.A. Gross. Incidence matrices and interval graphs. *Pacific Journal of Math.*, 15:835–855, 1965.

[13] Michael T. Goodrich. Simulating parallel algorithms in the mapreduce framework with applications to parallel computational geometry. *CoRR*, abs/1004.4708, 2010.

[14] John Hopcroft and Robert Tarjan. Algorithm 447: efficient algorithms for graph manipulation. *Commun. ACM*, 16:372–378, June 1973.

[15] Blake Irving. Big data and the power of hadoop. Yahoo! Hadoop Summit, June 2010.

[16] Howard Karloff, Siddharth Suri, and Sergei Vassilvitskii. A model of computation for mapreduce. In *Proceedings of the Twenty-First Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '10, pages 938–948, Philadelphia, PA, USA, 2010. Society for Industrial and Applied Mathematics.

[17] Silvio Lattanzi, Benjamin Moseley, Siddharth Suri, and Sergei Vassilvitskii. Filtering: a method for solving graph problems in mapreduce. In *Proceedings of the 23rd ACM symposium on Parallelism in algorithms and architectures*, SPAA '11, pages 85–94, New York, NY, USA, 2011. ACM.

[18] Michael Ley. The dblp computer science bibliography. http://dblp.uni-trier.de/db/.

[19] Jimmy Lin and Michael Schatz. Design patterns for efficient graph algorithms in mapreduce. In *Proceedings of the Eighth Workshop on Mining and*

Learning with Graphs, MLG '10, pages 78–85, New York, NY, USA, 2010. ACM.

[20] S. Muthukrishnan. Data streams: Algorithms and applications. *Foundations and Trends in Theoretical Computer Science*, 1(2), 2005.

[21] Mike Schroepfer. Inside large-scale analytics at facebook. Yahoo! Hadoop Summit, June 2010.

[22] Robert Sedgewick. *Algorithms in Java - part 5: graph algorithms (3 .ed.)*. Addison-Wesley-Longman, 2003.

[23] Siddharth Suri and Sergei Vassilvitskii. Counting triangles and the curse of the last reducer. In *Proceedings of the 20th international conference on World wide web*, WWW '11, pages 607–614, New York, NY, USA, 2011. ACM.

[24] Robert Endre Tarjan and Uzi Vishkin. An efficient parallel biconnectivity algorithm. *SIAM J. Comput.*, 14(4):862–874, 1985.

[25] Sergei Vassilvitski. Mapreduce Algorithmics. In *First International Workshop on Large-Scale Distributed Computing. Shonan, Japan, 2012.* `http://www.nii.ac.jp/shonan/seminar011/`.

[26] Jeffery Westbrook and Robert Endre Tarjan. Maintaining bridge-connected and biconnected components on-line. *Algorithmica*, 7(1–6):433–464, 1992.

[27] Tom White. *Hadoop: The Definitive Guide*. O'Reilly, first edition, june 2009.