HADI: Mining Radii of Large Graphs

U KANG
Carnegie Mellon University
CHARALAMPOS E. TSOURAKAKIS
Carnegie Mellon University
ANA PAULA APPEL
USP at São Carlos
CHRISTOS FALOUTSOS
Carnegie Mellon University
and
JURE LESKOVEC
Stanford University

Given large, multi-million node graphs (e.g., Facebook, web-crawls, etc.), how do they evolve over time? How are they connected? What are the central nodes and the outliers? In this paper we define the Radius plot of a graph and show how it can answer these questions. However, computing the Radius plot is prohibitively expensive for graphs reaching the planetary scale.

There are two major contributions in this paper: (a) We propose HADI (HAdoop DIameter and radii estimator), a carefully designed and fine-tuned algorithm to compute the radii and the diameter of massive graphs, that runs on the top of the HADOOP/MAPREDUCE system, with excellent scale-up on the number of available machines (b) We run HADI on several real world datasets including YahooWeb (*6B edges, 1/8 of a Terabyte*), one of the largest public graphs ever analyzed.

Thanks to HADI, we report fascinating patterns on large networks, like the surprisingly small effective diameter, the multi-modal/bi-modal shape of the Radius plot, and its palindrome motion over time.

Categories and Subject Descriptors: H.2.8 [**Database Management**]: Database Applications – Data Mining

General Terms: Algorithms, Experimentation

Additional Key Words and Phrases: radius plot, small web, HADI, hadoop, graph mining

1. INTRODUCTION

How do real, Terabyte-scale graphs look like? Is it true that the nodes with the highest degree are the most central ones, i.e., have the smallest radius? How do we

Author's address: U Kang, Computer Science Department, School of Computer Science, Carnegie Mellon University, 5000 Forbes Avenue, 15213 Pittsburgh PA USA, `ukang@cs.cmu.edu`.

compute the diameter and node radii in graphs of such size?

Graphs appear in numerous settings, such as social networks (Facebook, LinkedIn), computer network intrusion logs, who-calls-whom phone networks, search engine clickstreams (term-URL bipartite graphs), and many more. The contributions of this paper are the following:

(1) *Design:* We propose HADI, a scalable algorithm to compute the radii and diameter of network. As shown in Figure 1 (c), our method is *7.6*× faster than the naive version.
(2) *Optimization and Experimentation:* We carefully fine-tune our algorithm, and we test it on one of the largest public web graph ever analyzed, with several *billions* of nodes and edges, spanning 1/8 of a Terabyte.
(3) *Observations:* Thanks to HADI, we find interesting patterns and observations, like the "Multi-modal and Bi-modal" pattern, and the surprisingly small effective diameter of the Web. For example, see the Multi-modal pattern in the radius plot of Figure 1, which also shows the effective diameter and the center node of the Web('google.com').

The HADI algorithm (implemented in Hadoop) and several datasets are available at `http://www.cs.cmu.edu/∼ukang/HADI`. The rest of the paper is organized as follows: Section 2 defines related terms and a sequential algorithm for the Radius plot. Section 3 describes large scale algorithms for the Radius plot, and Section 4 analyzes the complexity of the algorithms and provides a possible extension. In Section 5 we present timing results, and in Section 6 we observe interesting patterns. After describing backgrounds in Section 7, we conclude in Section 8.

2. PRELIMINARIES; SEQUENTIAL RADII CALCULATION

2.1 Definitions

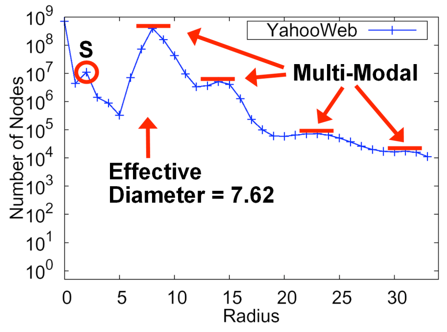
In this section, we define several terms related to the radius and the diameter. Recall that, for a node v in a graph G, the *radius* $r(v)$ is the distance between v and a reachable node farthest away from v. The *diameter* $d(G)$ of a graph G is the maximum radius of nodes $v \in G$. That is, $d(G) = \max_v r(v)$ [Lewis. 2009].

Since the radius and the diameter are susceptible to outliers (e.g., long chains), we follow the literature [Leskovec et al. 2005] and define the *effective* radius and diameter as follows.

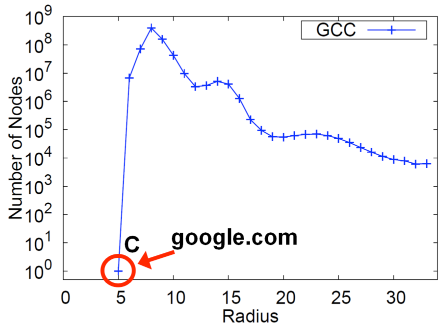
Definition 1 Effective Radius. *For a node v in a graph G, the effective radius $r_{eff}(v)$ of v is the 90th-percentile of all the shortest distances from v.*

Definition 2 Effective Diameter. *The effective diameter $d_{eff}(G)$ of a graph G is the minimum number of hops in which 90% of all connected pairs of nodes can reach each other.*

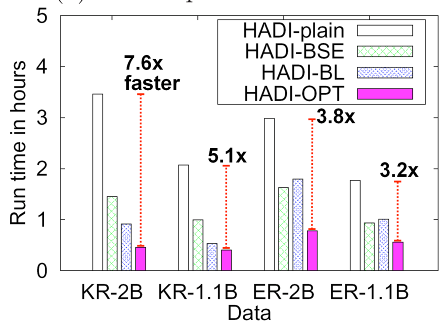
The effective radius is very related to *closeness centrality* that is widely used in network sciences to measure the importance of nodes [Newman. 2005]. Closeness centrality of a node v is the mean shortest-path distance between v and all other nodes reachable from it. On the other hand, the effective radius of v is 90% quantile of the shortest-path distances. Although their definitions are slightly different, they



(a) Radius plot of YahooWeb



(b) Radius plot of GCC of YahooWeb



(c) Running time of HADI

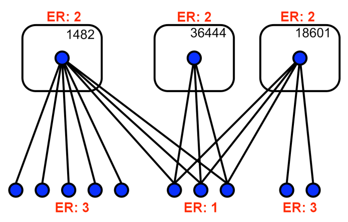
Fig. 1. **(a)** Radius plot(Count versus Radius) of the YahooWeb graph. Notice the effective diameter is surprisingly small. Also notice the peak(marked 'S') at radius 2, due to star-structured disconnected components.
(b) Radius plot of GCC(Giant Connected Component) of YahooWeb graph. The *only* node with radius 5 (marked 'C') is `google.com`.
(c) Running time of HADI with/without optimizations for Kronecker and Erdős-Rényi graphs with billions edges. Run on the M45 HADOOP cluster, using 90 machines for 3 iterations. HADI-OPT is up to **7.6**× faster than HADI-plain.

share the same spirit and can both be used as a measure of the 'centrality', or the time to spread information from a node to all other nodes.

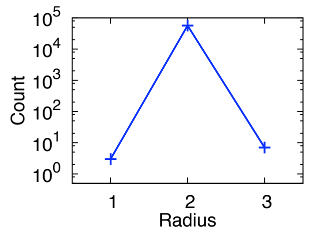
We will use the following three Radius-based Plots:

(1) **Static Radius Plot** (or just "Radius plot") of graph G shows the distribution (count) of the effective radius of nodes at a specific time. See Figure 1 and 2 for the example of real-world and synthetic graphs.
(2) **Temporal Radius Plot** shows the distributions of effective radius of nodes at several times(see Figure 14 for an example).
(3) **Radius-Degree Plot** shows the scatter-plot of the effective radius $r_{eff}(v)$ versus the degree d_v for each node v, as shown in Figure 12.

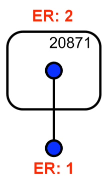
Table I lists the symbols used in this paper.



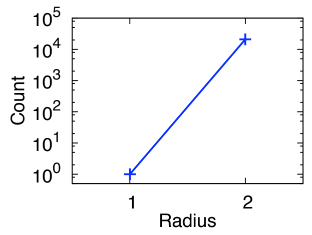
(a) Near-Bipartite-Core



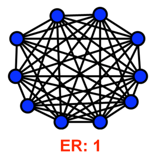
(b) Radius plot of the Near-Bipartite-Core graph



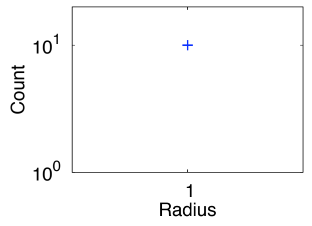
(c) Star



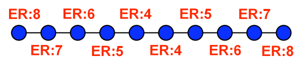
(d) Radius plot of the Star graph



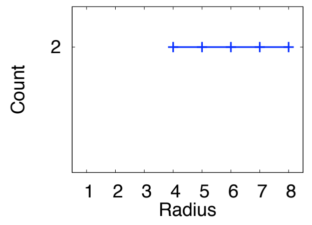
(e) Clique



(f) Radius plot of the Clique graph



(g) Chain



(h) Radius plot of the Chain graph

Fig. 2. Radius plots of real-world and synthetic graphs. (a) and (c) are from the anomalous disconnected components of YahooWeb graph. (e) and (g) are synthetic examples to show the radius plots. ER means the Effective Radius of nodes. A node inside a rounded box represents N nodes specified in the top area of the box. For example, (c) is a compact representation of a star graph where the core node at the bottom is connected to 20,871 neighbors. Notice that Radius plots provide concise summary of the structure of graphs.

Symbol	Definition
G	a graph
n	number of nodes in a graph
m	number of edges in a graph
d	diameter of a graph
h	number of hops
$N(h)$	number of node-pairs reachable in $\leq h$ hops (neighborhood function)
$N(h,i)$	number of neighbors of node i reachable in $\leq h$ hops
$b(h,i)$	Flajolet-Martin bitstring for node i at h hops.
$\hat{b}(h,i)$	Partial Flajolet-Martin bitstring for node i at h hops

Table I. Table of symbols

2.2 Computing Radius and Diameter

To generate the Radius plot, we need to calculate the effective radius of every node. In addition, the effective diameter is useful for tracking the evolution of networks. Therefore, we describe our algorithm for computing the effective radius and the effective diameter of a graph. As described in Section 7, existing algorithms do not scale well. To handle graphs with billions of nodes and edges, we use the following two main ideas:

(1) We use an approximation rather than an exact algorithm.
(2) We design a parallel algorithm for HADOOP/MAPREDUCE (the algorithm can also run in a parallel RDBMS).

We first describe why exact computation is infeasible. Assume we have a 'set' data structure that supports two functions: *add()* for adding an item, and *size()* for returning the count of distinct items. With the set, radii of nodes can be computed as follows:

(1) For each node i, make a set S_i and initialize by adding i to it.
(2) For each node i, update S_i by adding one-step neighbors of i to S_i.
(3) For each node i, continue updating S_i by adding 2,3,...-step neighbors of i to S_i. If the size of S_i before and after the addition does not change, then the node i reached its radius. Iterate until all nodes reach their radii.

Although simple and clear, the above algorithm requires too much space, $O(n^2)$, since there are n nodes and each node requires n space in the end. Since exact implementation is hopeless, we turn to an approximation algorithm for the effective radius and the diameter computation. For the purpose, we use the Flajolet-Martin algorithm [Flajolet and Martin 1985; Palmer et al. 2002] for counting the number of distinct elements in a multiset. While many other applicable algorithms exist (e.g., [Beyer et al. 2007; Charikar et al. 2000; Garofalakis and Gibbon 2001]), we choose the Flajolet-Martin algorithm because it gives an unbiased estimate, as well as a tight $O(\log n)$ bound for the space complexity [Alon et al. 1996].

The main idea of Flajolet-Martin algorithm is as follows. We maintain a bitstring $BITMAP[0 \ldots L-1]$ of length L which encodes the set. For each item to add, we do the following:

(1) Pick an $index \in [0 \ldots L-1]$ with probability $1/2^{index+1}$.

(2) Set $BITMAP[index]$ to 1.

Let R denote the index of the leftmost '0' bit in $BITMAP$. The main result of Flajolet-Martin is that the unbiased estimate of the size of the set is given by

$$\frac{1}{\varphi}2^R \tag{1}$$

where $\varphi = 0.77351\cdots$. The more concentrated estimate can be get by using multiple bitstrings and averaging the R. If we use K bitstrings R_1 to R_K, the size of the set can be estimated by

$$\frac{1}{\varphi}2^{\frac{1}{K}\sum_{l=1}^{K} R_l} \tag{2}$$

Picking an index from an item depend on a value computed from a hash function with the item as an input. Thus, merging the two set A and B is simply bitwise-OR'ing the bitstrings of A and B without worrying about the redundant elements. The application of Flajolet-Martin algorithm to radius and diameter estimation is straightforward. We maintain K Flajolet-Martin (FM) bitstrings $b(h,i)$ for each node i and current hop number h. $b(h,i)$ encodes the number of nodes reachable from node i within h hops, and can be used to estimate radii and diameter as shown below. The bitstrings $b(h,i)$ are iteratively updated until the bitstrings of all nodes stabilize. At the h-th iteration, each node receives the bitstrings of its neighboring nodes, and updates its own bitstrings $b(h-1,i)$ handed over from the previous iteration:

$$b(h,i) = b(h-1,i) \text{ BIT-OR } \{b(h-1,j) | (i,j) \in E\} \tag{3}$$

where "BIT-OR" denotes bitwise-OR function. After h iterations, a node i has K bitstrings that encode the *neighborhood function* $N(h,i)$, that is, the number of nodes within h hops from the node i. $N(h,i)$ is estimated from the K bitstrings by

$$N(h,i) = \frac{1}{0.77351}2^{\frac{1}{K}\sum_{l=1}^{K} b_l(i)} \tag{4}$$

where $b_l(i)$ is the position of leftmost '0' bit of the l^{th} bitstring of node i. The iterations continue until the bitstrings of all nodes stabilize, which is a necessary condition that the current iteration number h exceeds the diameter $d(G)$. After the iterations finish at h_{max}, we calculate the effective radius for every node and the diameter of the graph, as follows:

—$r_{eff}(i)$ is the smallest h such that $N(h,i) \geq 0.9 \cdot N(h_{max}, i)$.

—$d_{eff}(G)$ is the smallest h such that $N(h) = \sum_i N(h,i) = 0.9 \cdot N(h_{max})$. If $N(h) > 0.9 \cdot N(h_{max}) > N(h-1)$, then $d_{eff}(G)$ is linearly interpolated from $N(h)$ and $N(h-1)$. That is, $d_{eff}(G) = (h-1) + \frac{0.9 \cdot N(h_{max}) - N(h-1)}{N(h) - N(h-1)}$.

Algorithm 1 shows the summary of the algorithm described above.

The parameter K is typically set to 32[Flajolet and Martin 1985], and $MaxIter$ is set to 256 since real graphs have relatively small effective diameter. The NewFM-Bitstring() function in line 2 generates K FM bitstrings [Flajolet and Martin 1985].

Algorithm 1: Computing Radii and Diameter

```
   Input: Input graph G and integers $MaxIter$ and $K$
   Output: $r_{eff}(i)$ of every node $i$, and $d_{eff}(G)$
1  for $i = 1$ to $n$ do
2      $b(0, i) \leftarrow$ NewFMBitstring($n$);
3  end
4  for $h = 1$ to $MaxIter$ do
5      $Changed \leftarrow 0$;
6      for $i = 1$ to $n$ do
7          for $l = 1$ to $K$ do
8              $b_l(h, i) \leftarrow b_l(h-1, i)$BIT-OR$\{b_l(h-1, j) | \forall j$ adjacent from $i\}$;
9          end
10         if $\exists l$ s.t. $b_l(h, i) \neq b_l(h-1, i)$ then
11             increase $Changed$ by 1;
12         end
13     end
14     $N(h) \leftarrow \sum_i N(h, i)$;
15     if $Changed$ equals to $0$ then
16         $h_{max} \leftarrow h$, and break for loop;
17     end
18 end
19 for $i = 1$ to $n$ do
20     // estimate eff. radii
21     $r_{eff}(i) \leftarrow$ smallest $h'$ where $N(h', i) \geq 0.9 \cdot N(h_{max}, i)$;
22 end
23 $d_{eff}(G) \leftarrow$ smallest $h'$ where $N(h') = 0.9 \cdot N(h_{max})$;
```

The effective radius $r_{eff}(i)$ is determined at line 21, and the effective diameter $d_{eff}(G)$ is determined at line 23.

Algorithm 1 runs in $O(dm)$ time, since the algorithm iterates at most d times with each iteration running in $O(m)$ time. By using approximation, Algorithm 1 runs faster than previous approaches (see Section 7 for discussion). However, Algorithm 1 is a sequential algorithm and requires $O(n \log n)$ space and thus can not handle extremely large graphs (more than billions of nodes and edges) which can not be fit into a single machine. In the next sections we present efficient parallel algorithms.

3. PROPOSED METHOD

In the next two sections we describe HADI, a parallel radius and diameter estimation algorithm. As mentioned in Section 2, HADI can run on the top of both a MapReduce system and a parallel SQL DBMS. In the following, we first describe the general idea behind HADI and show the algorithm for MapReduce. The algorithm for parallel SQL DBMS is sketched in Section 4.

3.1 HADI Overview

HADI follows the flow of Algorithm 1; that is, it uses the FM bitstrings and iteratively updates them using the bitstrings of its neighbors. The most expensive operation in Algorithm 1 is line 8 where bitstrings of each node are updated. Therefore, HADI focuses on the efficient implementation of the operation using MapReduce framework.

It is important to notice that HADI is a disk-based algorithm; indeed, memory-based algorithm is not possible for Tera- and Peta-byte scale data. HADI saves two kinds of information to a distributed file system (such as HDFS (Hadoop Distributed File System) in the case of Hadoop):

—**Edge** has a format of (*srcid*, *dstid*).
—**Bitstrings** has a format of (*nodeid*, $bitstring_1$, ..., $bitstring_K$).

Combining the bitstrings of each node with those of its neighbors is very expensive operation which needs several optimizations to scale up near-linearly. In the following sections we will describe three HADI algorithms in a progressive way. That is we first describe HADI-naive, to give the big picture and explain why it such a naive implementation should not be used in practice, then the HADI-plain, and finally HADI-optimized, the proposed method that should be used in practice. We use Hadoop to describe the MapReduce version of HADI.

3.2 HADI-naive in MapReduce

HADI-naive is inefficient, but we present it for ease of explanation.

Data The edge file is saved as a sparse adjacency matrix in HDFS. Each line of the file contains a nonzero element of the adjacency matrix of the graph, in the format of (*srcid*, *dstid*). Also, the bitstrings of each node are saved in a file in the format of (*nodeid*, $flag$, $bitstring_1$, ..., $bitstring_K$). The $flag$ records information about the status of the nodes(e.g., 'Changed' flag to check whether one of the bitstrings changed or not). Notice that we *don't know* the physical distribution of the data in HDFS.

Main Program Flow The main idea of HADI-naive is to use the bitstrings file as a logical "cache" to machines which contain edge files. The bitstring update operation in Equation (3) of Section 2 requires that the machine which updates the bitstrings of node i should have access to (a) all edges adjacent from i, and (b) all bitstrings of the adjacent nodes. To meet the requirement (a), it is needed to reorganize the edge file so that edges with a same source id are grouped together. That can be done by using an Identity mapper which outputs the given input edges in (*srcid*, *dstid*) format. The most simple yet naive way to meet the requirement (b) is sending the bitstrings to every reducer which receives the reorganized edge file.

Thus, HADI-naive iterates over two-stages of MapReduce. The first stage updates the bitstrings of each node and sets the 'Changed' flag if at least one of the bitstrings of the node is different from the previous bitstring. The second stage counts the number of changed nodes and stops iterations when the bitstrings stabilized, as illustrated in the swim-lane diagram of Figure 3.

Although conceptually simple and clear, HADI-naive is unnecessarily expensive, because it ships all the bitstrings to all reducers. Thus, we propose HADI-plain

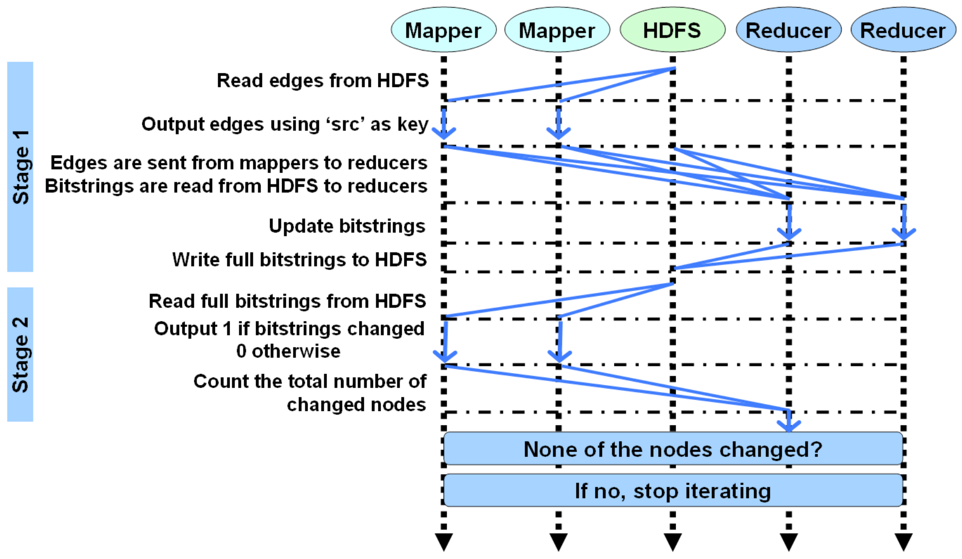

Mapper
Mapper
HDFS
Reducer
Reducer
Stage 1
Read edges from HDFS
Output edges using 'src' as key
Edges are sent from mappers to reducers
Bitstrings are read from HDFS to reducers
Update bitstrings
Write full bitstrings to HDFS
Stage 2
Read full bitstrings from HDFS
Output 1 if bitstrings changed
0 otherwise
Count the total number of changed nodes
None of the nodes changed?
If no, stop iterating

Fig. 3. One iteration of HADI-naive. *Stage 1.* Bitstrings of all nodes are sent to every reducer. *Stage 2.* Sums up the count of changed nodes.

and additional optimizations, which we explain next.

3.3 HADI-plain in MapReduce

HADI-plain improves HADI-naive by *copying only the necessary bitstrings to each reducer*. The details are next:

Data As in HADI-naive, the edges are saved in the format of ($srcid$, $dstid$), and bitstrings are saved in the format of ($nodeid$, $flag$, $bitstring_1$, ..., $bitstring_K$) in files over HDFS. The initial bitstrings generation, which corresponds to line 1-3 of Algorithm 1, can be performed in completely parallel way. The $flag$ of each node records the following information:

—**Effective Radii** and Hop Numbers to calculate the effective radius.
—**Changed** flag to indicate whether at least a bitstring has been changed or not.

Main Program Flow As mentioned in the beginning, HADI-plain copies only the necessary bitstrings to each reducer. The main idea is to replicate bitstrings of node j exactly x times where x is the in-degree of node j. The replicated bitstrings of node j is called the *partial bitstring* and represented by $\hat{b}(h,j)$. The replicated $\hat{b}(h,j)$'s are used to update $b(h,i)$, the bitstring of node i where (i,j) is an edge in the graph. HADI-plain iteratively runs three-stage MapReduce jobs until all bitstrings of all nodes stop changing. Algorithm 2, 3, and 4 shows HADI-plain, and Figure 4 shows the swim-lane. We use h for denoting the current iteration number which starts from h=1. Output(a,b) means to output a pair of data with the key a and the value b.

Stage 1 We generate (key, value) pairs, where the key is the node id i and the value is the partial bitstrings $\hat{b}(h,j)$'s where j ranges over all the neighbors adjacent

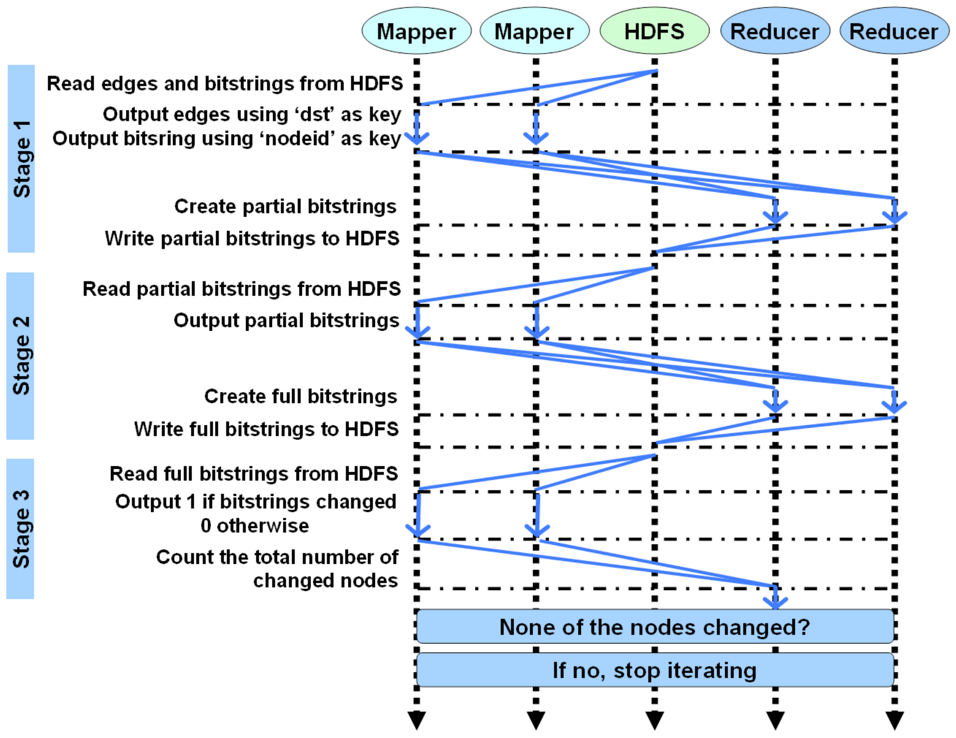

Mapper
Mapper
HDFS
Reducer
Reducer
Stage 1
Read edges and bitstrings from HDFS
Output edges using 'dst' as key
Output bitsring using 'nodeid' as key
Create partial bitstrings
Write partial bitstrings to HDFS
Stage 2
Read partial bitstrings from HDFS
Output partial bitstrings
Create full bitstrings
Write full bitstrings to HDFS
Stage 3
Read full bitstrings from HDFS
Output 1 if bitstrings changed
0 otherwise
Count the total number of
changed nodes
None of the nodes changed?
If no, stop iterating

Fig. 4. One iteration of HADI-plain. *Stage 1.* Edges and bitstrings are matched to create partial bitstrings. *Stage 2.* Partial bitstrings are merged to create full bitstrings. *Stage 3.* Sums up the count of changed nodes, and compute N(h), the neighborhood function. Computing N(h) is not drawn in the figure for clarity.

from node i. To generate such pairs, the bitstrings of node j are grouped together with edges whose *dstid* is j. Notice that at the very first iteration, bitstrings of nodes do not exist; they have to be generated on the fly, and we use the *Bitstring Creation Command* for that. Notice also that line 22 of Algorithm 2 is used to propagate the bitstrings of one's own node. These bitstrings are compared to the newly updated bitstrings at Stage 2 to check convergence.

Stage 2 Bitstrings of node i are updated by combining partial bitstrings of itself and nodes adjacent from i. For the purpose, the mapper is the Identity mapper (output the input without any modification). The reducer combines them, generates new bitstrings, and sets $flag$ by recording (a) whether at least a bitstring changed or not, and (b) the current iteration number h and the neighborhood value $N(h,i)$ (line 11). This h and $N(h,i)$ are used to calculate the effective radius of nodes after all bitstrings converge, i.e., don't change. Notice that only the last neighborhood $N(h_{last},i)$ and other neighborhoods $N(h',i)$ that satisfy $N(h',i) \geq 0.9 \cdot N(h_{last},i)$ need to be saved to calculate the effective radius. The output of Stage 2 is fed into the input of Stage 1 at the next iteration.

Stage 3 We calculate the number of changed nodes and sum up the neighborhood value of all nodes to calculate $N(h)$. We use only two unique keys(key_for_changed

Algorithm 2: HADI Stage 1

```
   Input: Edge data $E = \{(i, j)\}$,
   Current bitstring $B = \{(i, b(h-1, i))\}$ or
   Bitstring Creation Command $BC = \{(i, cmd)\}$
   Output: Partial bitstring $B' = \{(i, b(h-1, j))\}$
1  Stage1-Map(key $k$, value $v$)
2  begin
3      if $(k, v)$ is of type $B$ or $BC$ then
4          Output($k, v$);
5      else if $(k, v)$ is of type $E$ then
6          Output($v, k$);
7  end
8  Stage1-Reduce(key $k$, values $V[]$)
9  begin
10     SRC ← [];
11     for $v \in V$ do
12         if $(k, v)$ is of type $BC$ then
13             $\hat{b}(h-1, k)$ ←NewFMBitstring();
14         else if $(k, v)$ is of type $B$ then
15             $\hat{b}(h-1, k) \leftarrow v$;
16         else if $(k, v)$ is of type $E$ then
17             Add $v$ to $SRC$;
18     end
19     for $src \in SRC$ do
20         Output($src, \hat{b}(h-1, k)$);
21     end
22     Output($k, \hat{b}(h-1, k)$);
23 end
```

and key_for_neighborhood), which correspond to the two calculated values. The analysis of line 3 can be done by checking the $flag$ field and using Equation (4) in Section 2. The variable *changed* is set to 1 or 0, based on whether the bitmask of node k changed or not.

When all bitstrings of all nodes converged, a MapReduce job to finalize the effective radius and diameter is performed and the program finishes. Compared to HADI-naive, the advantage of HADI-plain is clear: bitstrings and edges are evenly distributed over machines so that the algorithm can handle as much data as possible, given sufficiently many machines.

3.4 HADI-optimized in MapReduce

HADI-optimized further improves HADI-plain. It uses two orthogonal ideas: "block operation" and "bit shuffle encoding". Both try to address some subtle performance issues. Specifically, Hadoop has the following two major bottlenecks:

Algorithm 3: HADI Stage 2

```
Input: Partial bitstring $B = \{(i, \hat{b}(h-1, j)\}$
Output: Full bitstring $B = \{(i, b(h, i)\}$
1  Stage2-Map(key $k$, value $v$) // Identity Mapper
2  begin
3      Output($k, v$);
4  end
5  Stage2-Reduce(key $k$, values $V[]$)
6  begin
7      $b(h, k) \leftarrow 0$;
8      for $v \in V$ do
9          $b(h, k) \leftarrow b(h, k)$ BIT-OR $v$;
10     end
11     Update $flag$ of $b(h, k)$;
12     Output($k, b(h, k)$);
13 end
```

Algorithm 4: HADI Stage 3

```
Input: Full bitstring $B = \{(i, b(h, i))\}$
Output: Number of changed nodes, Neighborhood $N(h)$
1  Stage3-Map(key $k$, value $v$)
2  begin
3      Analyze $v$ to get ($changed$, $N(h, i)$);
4      Output($key\_for\_changed$,$changed$);
5      Output($key\_for\_neighborhood$, $N(h, i)$);
6  end
7  Stage3-Reduce(key $k$, values $V[]$)
8  begin
9      $Changed \leftarrow 0$;
10     $N(h) \leftarrow 0$;
11     for $v \in V$ do
12         if $k$ is $key\_for\_changed$ then
13             $Changed \leftarrow Changed + v$;
14         else if $k$ is $key\_for\_neighborhood$ then
15             $N(h) \leftarrow N(h) + v$;
16     end
17     Output($key\_for\_changed$,$Changed$);
18     Output($key\_for\_neighborhood$, $N(h)$);
19 end
```

—Materialization: at the end of each map/reduce stage, the output is written to the disk, and it is also read at the beginning of next reduce/map stage.

—Sorting: at the *Shuffle* stage, data is sent to each reducer and sorted before they are handed over to the *Reduce* stage.

HADI-optimized addresses these two issues.

Block Operation Our first optimization is the block encoding of the edges and the bitstrings. The main idea is to group w by w sub-matrix into a super-element in the adjacency matrix E, and group w bitstrings into a super-bitstring. Now, HADI-plain is performed on these super-elements and super-bitstrings, instead of the original edges and bitstrings. Of course, appropriate decoding and encoding is necessary at each stage. Figure 5 shows an example of converting data into block-format.

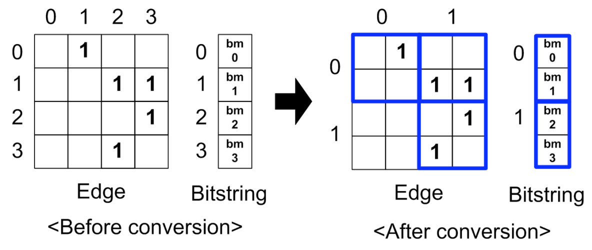


Fig. 5. Converting the original edge and bitstring to blocks. The 4-by-4 edge and length-4 bitstring are converted to 2-by-2 super-elements and length-2 super-bitstrings. Notice the lower-left super-element of the edge is not produced since there is no nonzero element inside it.

By this block operation, the performance of HADI-plain changes as follows:

—*Input size* decreases in general, since we can use fewer bits to index elements inside a block.
—*Sorting time* decreases, since the number of elements to sort decreases.
—*Network traffic* decreases since the result of matching a super-element and a super-bitstring is a bitstring which can be at maximum *block_width* times smaller than that of HADI-plain.
—*Map and Reduce functions* takes more time, since the block must be decoded to be processed, and be encoded back to block format.

For reasonable-size blocks, the performance gains (smaller input size, faster sorting time, less network traffic) outweigh the delays (more time to perform the map and reduce function). Also notice that the number of edge blocks depends on the community structure of the graph: if the adjacency matrix is nicely clustered, we will have fewer blocks. See Section 5, where we show results from block-structured graphs (‘Kronecker graphs’ [Leskovec et al. 2005]) and from random graphs (‘Erdős-Rényi graphs’ [Erdős and Rényi 1959]).

Bit Shuffle Encoding In our effort to decrease the input size, we propose an encoding scheme that can compress the bitstrings. Recall that in HADI-plain, we use K (e.g., 32, 64) bitstrings for each node, to increase the accuracy of our estimator. Since HADI requires $O(K(m+n)\log n)$ space, the amount of data increases when K is large. For example, the YahooWeb graph in Section 6 spans

120 GBytes (with 1.4 billion nodes, 6.6 billion edges). However the required disk space for just the bitstrings is $32 \cdot (1.4B + 6.6B) \cdot 8$ byte = 2 Tera bytes (assuming 8 byte for each bitstring), which is more than 16 times larger than the input graph.

The main idea of Bit Shuffle Encoding is to carefully reorder the bits of the bitstrings of each node, and then use Run Length Encoding. By construction, the leftmost part of each bitstring is almost full of one's, and the rest is almost full of zeros. Specifically, we make the reordered bit strings to contain long sequences of 1's and 0's: we get all the first bits from all K bitstrings, then get the second bits, and so on. As a result we get a single bit-sequence of length $K \cdot |bitstring|$, where most of the first bits are '1's, and most of the last bits are '0's. Then we encode only the length of each bit sequence, achieving good space savings (and, eventually, time savings, through fewer I/Os).

4. ANALYSIS AND DISCUSSION

In this section, we analyze the time/space complexity of HADI and its possible implementation at RDBMS.

4.1 Time and Space Analysis

We analyze the algorithm complexity of HADI with M machines for a graph G with n nodes and m edges with diameter d. We are interested in the time complexity, as well as the space complexity.

Lemma 1 Time Complexity of HADI. *HADI takes* $O(\frac{d(m+n)}{M} \log \frac{m+n}{M})$ *time.*

Proof. (Sketch) The Shuffle steps after Stage1 takes $O(\frac{m+n}{M} \log \frac{m+n}{M})$ time which dominates the time complexity. □

Notice that the time complexity of HADI is less than previous approaches in Section 7($O(n^2 + mn)$, at best). Similarly, for space we have:

Lemma 2 Space Complexity of HADI. *HADI requires* $O((m+n) \log n)$ *space.*

Proof. (Sketch) The maximum space $k \cdot ((m + n) \log n)$ is required at the output of `Stage1`-Reduce. Since k is a constant, the space complexity is $O((m + n) \log n)$. □

4.2 HADI in parallel DBMSs

Using relational database management systems (RDBMS) for graph mining is a promising research direction, especially given the findings of [Pavlo et al. 2009]. We mention that HADI can be implemented on the top of an Object-Relational DBMS (parallel or serial): it needs repeated joins of the edge table with the appropriate table of bit-strings, and a user-defined function for bit-OR-ing. We sketch a potential implementation of HADI in a RDBMS.

Data In parallel RDBMS implementations, data is saved in tables. The edges are saved in the table E with attributes *src*(source node id) and *dst*(destination node id). Similarly, the bitstrings are saved in the table B with

Main Program Flow The main flow comprises iterative execution of SQL statements with appropriate UDF(user defined function)s. The most important and expensive operation is updating the bitstrings of nodes. Observe that the operation can be concisely expressed as a SQL statement:

```
SELECT INTO B_NEW E.src, BIT-OR(B.b)
  FROM E, B
  WHERE E.dst=B.id
  GROUP BY E.src
```

The SQL statement requires BIT-OR(), a UDF function that implements the bit-OR-ing of the Flajolet-Martin bitstrings. The RDBMS implementation iteratively runs the SQL until B_NEW is same as B. B_NEW created at an iteration is used as B at the next iteration.

5. SCALABILITY OF HADI

In this section, we perform experiments to answer the following questions:

—Q1: How fast is HADI?
—Q2: How does it scale up with the graph size and the number of machines?
—Q3: How do the optimizations help performance?

5.1 Experimental Setup

We use both real and synthetic graphs in Table II for our experiments and analysis in Section 5 and 6, with the following details.

—YahooWeb: web pages and their hypertext links indexed by Yahoo! Altavista search engine in 2002.
—Patents: U.S. patents, citing each other (from 1975 to 1999).
—LinkedIn: people connected to other people (from 2003 to 2006).
—Kronecker: Synthetic Kronecker graphs [Leskovec et al. 2005] using a chain of length two as the seed graph.

Graph	Nodes	Edges	File	Description
YahooWeb	1.4 B	6.6 B	116G	page-page
LinkedIn	7.5 M	58 M	1G	person-person
Patents	6 M	16 M	264M	patent-patent
Kronecker	177 K	1,977 M	25G	synthetic
	120 K	1,145M	13.9G	
	59 K	282 M	3.3G	
Erdős-Rényi	177 K	1,977 M	25G	random $G_{n,p}$
	120 K	1,145 M	13.9G	
	59 K	282 M	3.3G	

Table II. Datasets. B: Billion, M: Million, K: Thousand, G: Gigabytes

For the performance experiments, we use synthetic Kronecker and Erdős-Rényi graphs. The reason of this choice is that we can generate any size of these two types of graphs, and Kronecker graph mirror several real-world graph characteristics, including small and constant diameters, power-law degree distributions, etc. The number of nodes and edges of Erdős-Rényi graphs have been set to the same as those of the corresponding Kronecker graphs. The main difference of Kronecker compared to Erdős-Rényi graphs is the emergence of a block-wise structure of the adjacency matrix, from its construction [Leskovec et al. 2005]. We will see how this characteristic affects in the running time of our block-optimization in the next sections.

HADI runs on *M45*, one of the fifty most powerful supercomputers in the world. M45 has 480 hosts (each with 2 quad-core Intel Xeon 1.86 GHz, running RHEL5), with 3Tb aggregate RAM, and over 1.5 Peta-byte disk size.

Finally, we use the following notations to indicate different optimizations of HADI:

—HADI-BSE: HADI-plain with bit shuffle encoding.
—HADI-BL: HADI-plain with block operation.
—HADI-OPT: HADI-plain with both bit shuffle encoding and block operation.

5.2 Running Time and Scale-up

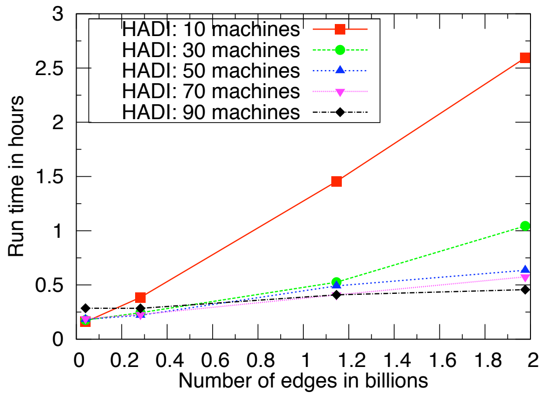


Fig. 6. Running time versus number of edges with HADI-OPT on Kronecker graphs for three iterations. Notice the excellent scalability: linear on the graph size (number of edges).

Figure 6 gives the wall-clock time of HADI-OPT versus the number of edges in the graph. Each curve corresponds to a different number of machines used (from 10 to 90). HADI has excellent scalability, with its running time being linear on the number of edges. The rest of the HADI versions (HADI-plain, HADI-BL, and HADI-BSE), were slower, but had a similar, linear trend, and they are omitted to avoid clutter.

Figure 7 gives the throughput $1/T_M$ of HADI-OPT. We also tried HADI with one machine; however it didn't complete, since the machine would take so long that it would often fail in the meanwhile. For this reason, we do not report the typical scale-up score $s = T_1/T_M$ (ratio of time with 1 machine, over time with M machine),

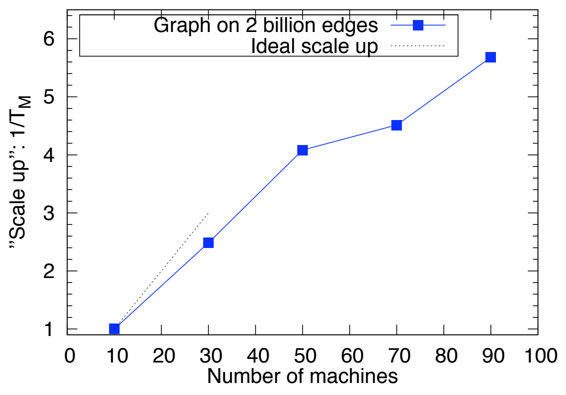


Fig. 7. "Scale-up" (throughput $1/T_M$) versus number of machines M, for the Kronecker graph (2B edges). Notice the near-linear growth in the beginning, close to the ideal(dotted line).

and instead we report just the inverse of T_M. HADI scales up near-linearly with the number of machines M, close to the ideal scale-up.

5.3 Effect of Optimizations

Among the optimizations that we mentioned earlier, which one helps the most, and by how much? Figure 1 (c) plots the running time of different graphs versus different HADI optimizations. For the Kronecker graphs, we see that block operation is more efficient than bit shuffle encoding. Here, HADI-OPT achieves **7.6**× better performance than HADI-plain. For the Erdős-Rényi graphs, however, we see that block operations do not help more than bit shuffle encoding, because the adjacency matrix has no block structure, while Kronecker graphs do. Also notice that HADI-BLK and HADI-OPT run faster on Kronecker graphs than on Erdős-Rényi graphs of the same size. Again, the reason is that Kronecker graphs have fewer nonzero blocks (i.e., "communities") by their construction, and the "block" operation yields more savings.

6. HADI AT WORK

HADI reveals new patterns in massive graphs which we present in this section.

6.1 Static Patterns

6.1.1 *Diameter.* What is the diameter of the Web? Albert et al. [Albert et al. 1999] computed the diameter on a directed Web graph with ≈ 0.3 million nodes, and conjectured that it is around 19 for the 1.4 billion-node Web as shown in the upper line of Figure 8. Broder et al. [Broder et al. 2000] used sampling from ≈ 200 million-nodes Web and reported 16.15 and 6.83 as the diameter for the directed and the undirected cases, respectively. What should be the effective diameter, for a significantly larger crawl of the Web, with billions of nodes? Figure 1 gives the surprising answer:

Observation 1 Small Web. *The effective diameter of the YahooWeb graph (year: 2002) is surprisingly small ($\approx 7 \sim 8$).*

The previous results from Albert et al. and Broder et al. are based on the average diameter. For the reason, we also computed the average diameter and show the comparison of diameters of different graphs in Figure 8. We first observe that the average diameters of all graphs are relatively small (< 20) for both the directed and the undirected cases. We also observe that the Albert et al.'s conjecture of the diameter of the directed graph is over-pessimistic: both the sampling and HADI reported smaller values for the diameter of the directed graph. For the diameter of the undirected graph, we observe the constant or shrinking diameter pattern [Leskovec et al. 2007].

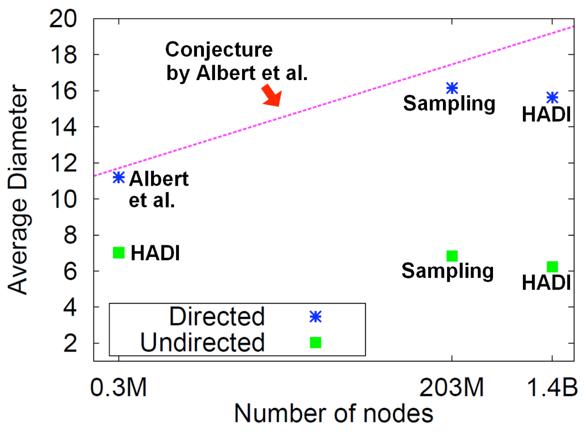
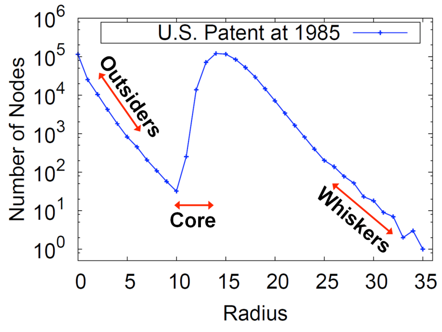
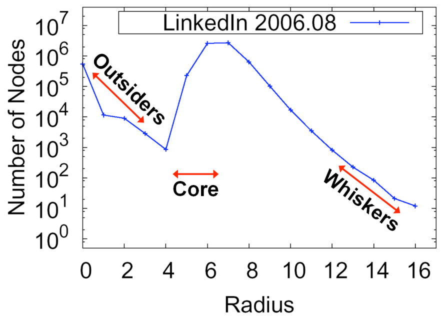


Fig. 8. Average diameter vs. number of nodes in lin-log scale for the three different Web graphs, where M and B represent millions and billions, respectively. (0.3M): web pages inside nd.edu at 1999, from Albert et al.'s work. (203M): web pages crawled by Altavista at 1999, from Broder et al.'s work (1.4B): web pages crawled by Yahoo at 2002 (YahooWeb in Table II). The annotations(Albert et al., Sampling, HADI) near the points represent the algorithms for computing the diameter. The Albert et al.'s algorithm seems to be an exact breadth first search, although not clearly specified in their paper. Notice the relatively small diameters for both the directed and the undirected cases. Also notice that the diameters of the undirected Web graphs remain near-constant.

6.1.2 *Shape of Distribution.* The next question is, how are the radii distributed in real networks? Is it Poisson? Lognormal? Figure 1 gives the surprising answer: multimodal! In other relatively small networks, however, we observed bi-modal structures. As shown in the Radius plot of U.S. Patent and LinkedIn network in Figure 9, they have a peak at zero, a dip at a small radius value (9, and 4, respectively) and another peak very close to the dip. Given the prevalence of the bi-modal shape, our conjecture is that the multi-modal shape of YahooWeb is possibly due to a mixture of relatively smaller sub-graphs, which got loosely connected recently.



(a) U.S. Patent



(b) LinkedIn

Fig. 9. Static Radius Plot(Count versus Radius) of U.S. Patent and LinkedIn graphs. Notice the bi-modal structure with 'outsiders'(nodes in the DCs), 'core'(central nodes in the GCC), and 'whiskers'(nodes connected to the GCC with long paths).

Observation 2 Multi-modal and Bi-modal. *The Radius distribution of the Web graph has a multi-modal structure. Many smaller networks have the bi-modal structure.*

About the bi-modal structure, a natural question to ask is what are the common properties of the nodes that belong to the first peak; similarly, for the nodes in the first dip, and the same for the nodes of the second peak. After investigation, the former are nodes that belong to the disconnected components (DCs); nodes in the dip are usually core nodes in the giant connected component (GCC), and the nodes at the second peak are the vast majority of well connected nodes in the GCC. Figure 10 exactly shows the radii distribution for the nodes of the GCC (in blue), and the nodes of the few largest remaining components.

In Figure 10, we clearly see that the second peak of the bimodal structure came from the giant connected component. But, where does the first peak around radius 0 come from? We can get the answer from the distribution of connected component of the same graph in Figure 11. Since the ranges of radius are limited by the size of connected components, we see the first peak of Radius plot came from the disconnected components whose size follows a power law.

Now we can explain the three important areas of Figure 9: '*outsiders*' are the nodes in the disconnected components, and responsible for the first peak and the negative slope to the dip. '*Core*' are the central nodes with the smallest radii from the giant connected component. '*Whiskers*' [Leskovec et al. 2008] are the nodes connected to the GCC with long paths(resembling a whisker), and are the reasons of the second negative slope.

6.1.3 *Radius plot of GCC.* Figure 1(b) shows a striking pattern: all nodes of the GCC of the YahooWeb graph have radius 6 or more, except for 1 (only!). Inspection shows that this is `google.com`. We were surprised, because we would expect a few more popular nodes to be in the same situation (eg., Yahoo, eBay, Amazon).

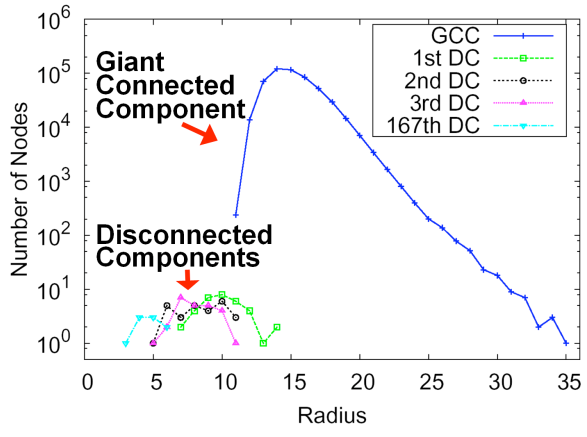


Fig. 10. Radius plot (Count versus radius) for several connected components of the U.S. Patent data in 1985. In blue: the distribution for the GCC (Giant Connected Component); rest colors: several DC (Disconnected Component)s.

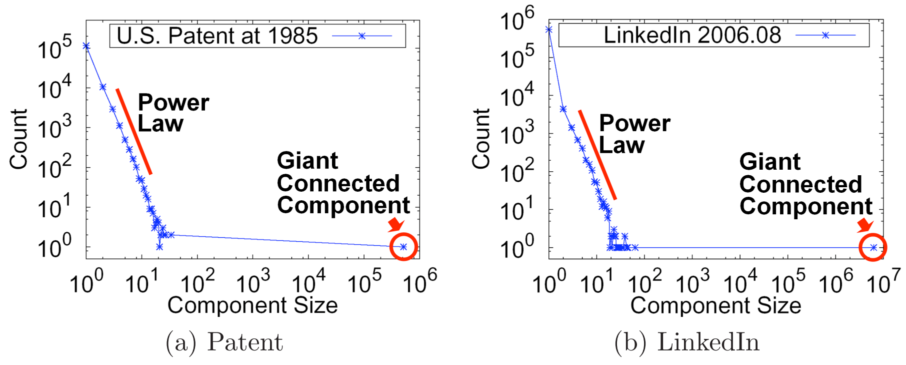
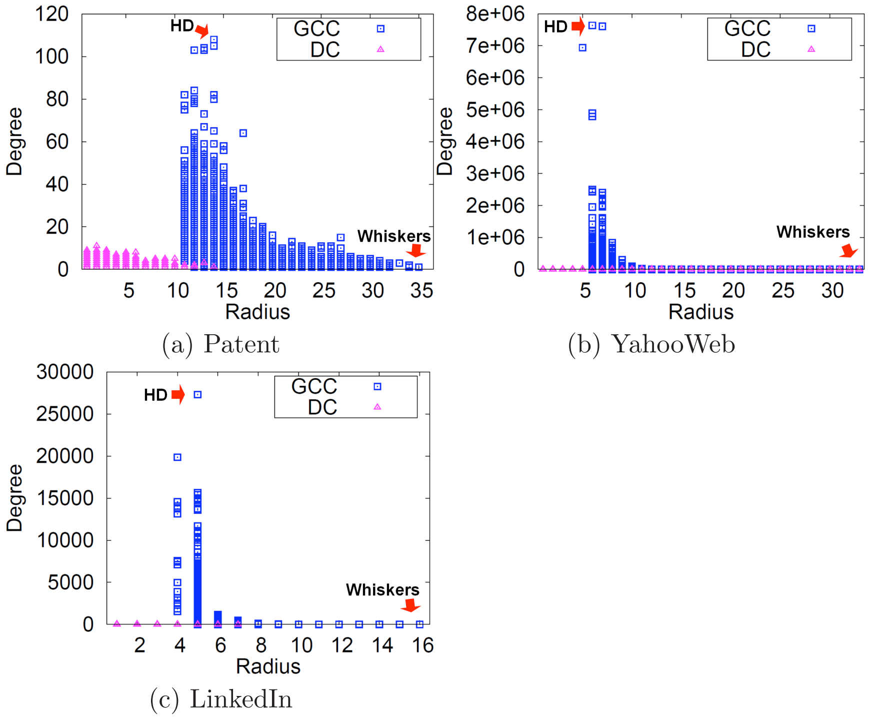


Fig. 11. Size distribution of connected components. Notice the size of the disconnected components(DCs) follows a power-law which explains the first peak around radius 0 of the radius plots in Figure 9.

6.1.4 *"Core" and "Whisker" nodes.* The next question is, what can we say about the connectivity of the *core* nodes, and the *whisker* nodes? For example, is it true that the highest degree nodes are the most central ones (i.e. minimum radius)? The answer is given by the "Radius-Degree" plot in Figure 12: This is a scatter-plot, with one dot for every node, plotting the degree of the node versus its radius. We also color-coded the nodes of the GCC (in blue), while the rest are in magenta.

Observation 3 High degree nodes. *The highest degree node (a) belongs to the core nodes inside the GCC but (b) are* not *necessarily the ones with the smallest radius.*



(a) Patent (b) YahooWeb (c) LinkedIn

Fig. 12. Radius-Degree plots of real-world graphs. HD represents the node with the highest degree. Notice that HD belongs to core nodes inside the GCC, and whiskers have small degree.

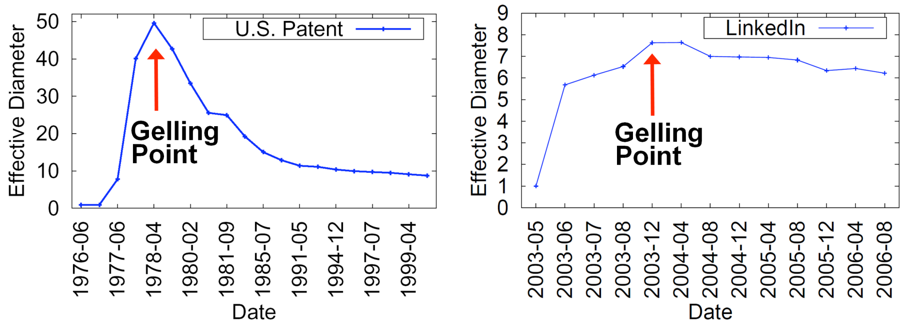
The next observation is that *whisker* nodes have small degree, that is, they belong to chains (as opposed to more complicated shapes)

6.1.5 *Radius plots of anomalous DCs.* The radius plots of some of the largest disconnected components of YahooWeb graph show anomalous radius distributions as opposed to the bi-modal distribution. The graph in Figure 2 (a) is the largest disconnected component which has a near-bipartite-core structure. The component is very likely to be a link farm since almost all the nodes are connected to the three nodes at the bottom center with the effective radius 1. Similarly, we observed many star-shaped disconnected components as shown in Figure 2 (c). This is also a strong candidate for a link farm, where a node with the effective radius 1 is connected to all the other nodes with the effective radius 2.

6.2 Temporal Patterns

Here we study how the radius distribution changes over time. We know that the diameter of a graph typically grows with time, spikes at the 'gelling point', and then shrinks [Mcglohon et al. 2008; Leskovec et al. 2007]. Indeed, this holds for our datasets as shown in Figure 13.

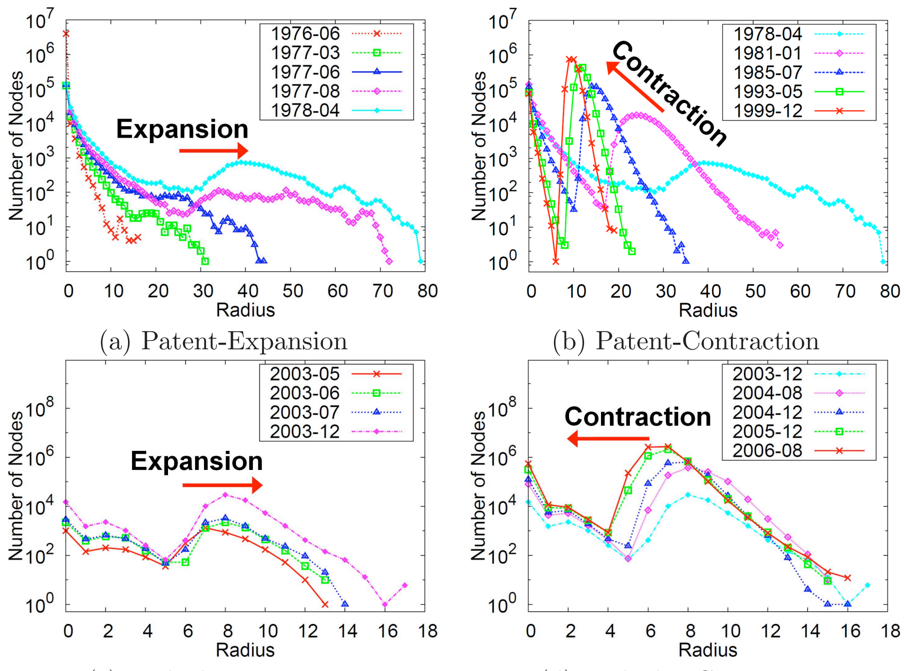
The question is, how does the radius distribution change over time? Does it still



(a) Patent (b) LinkedIn

Fig. 13. Evolution of the effective diameter of real graphs. The diameter increases until a 'gelling' point, and starts to decrease after the point.

have the bi-modal pattern? Do the peaks and slopes change over time? We show the answer in Figure 14 and Observation 4.



(a) Patent-Expansion (b) Patent-Contraction

(c) LinkedIn-Expansion (d) LinkedIn-Contraction

Fig. 14. Radius distribution over time. "Expansion": the radius distribution moves to the right until the gelling point. "Contraction": the radius distribution moves to the left after the gelling point.

OBSERVATION 4 EXPANSION-CONTRACTION. *The radius distribution expands to the right until it reaches the gelling point. Then, it contracts to the left.*

Another striking observation is that the decreasing segments seem to be well fit by a line, in log-lin axis, thus indicating an exponential decay.

OBSERVATION 5 EXPONENTIAL DECAYS. *The decreasing segments of several, real radius plots seem to decay exponentially, that is*

$$count(r) \propto \exp(-cr) \tag{5}$$

for every time tick after *the gelling point.* $count(r)$ *is the number of nodes with radius* r*, and* c *is a constant.*

For the Patent and LinkedIn graphs, the correlation coefficient was excellent, (typically, -0.98 or better).

7. BACKGROUND

We briefly present related works on algorithms for radius and diameter computation, as well as on large graph mining.

Computing Radius and Diameter The typical algorithms to compute the radius and the diameter of a graph include Breadth First Search (BFS) and Floyd's algorithm [Cormen et al. 1990]. Both approaches are prohibitively slow for large graphs, requiring $O(n^2 + nm)$ and $O(n^3)$ time, where n and m are the number of nodes and edges, respectively. For the same reason, related BFS or all-pair shortest-path based algorithms like [Ferrez et al. 1998; Bader and Madduri 2008; Ma and Ma 1993; Sinha et al. 1986] can not handle large graphs.

A sampling approach starts BFS from a subset of nodes, typically chosen at random as in [Broder et al. 2000]. Despite its practicality, this approach has no obvious solution for choosing the representative sample for BFS.

Large Graph Mining There are numerous papers on large graph mining and indexing, mining subgraphs [Ke et al. 2009; You et al. 2009], ADI[Wang et al. 2004], gSpan[Yan and Han 2002], graph clustering [Satuluri and Parthasarathy 2009], Graclus [Dhillon et al. 2007], METIS [Karypis and Kumar 1999], partitioning [Daruru et al. 2009; Chakrabarti et al. 2004; Dhillon et al. 2003], tensors [Kolda and Sun 2008; Tsourakakis 2009], triangle counting [Becchetti et al. 2008; Tsourakakis 2008; Tsourakakis et al. 2009; Tsourakakis et al. 2009] , minimum cut [Aggarwal et al. 2009], to name a few. However, none of the above computes the diameter of the graph or radii of the nodes.

Large scale data processing using scalable and parallel algorithms has attracted increasing attentions due to the needs to process web-scale data. Due to the volume of the data, platforms for this type of processing often choose "shared-nothing" architecture. Two promising platforms for such large scale data analysis are (a) MAPREDUCE and (b) parallel RDBMS.

The MAPREDUCE programming framework processes huge amounts of data in a massively parallel way, using thousands or millions commodity machines. It has advantages of (a) fault-tolerance, (b) familiar concepts from functional programming, and (c) low cost of building the cluster. HADOOP, the open source version of MAPREDUCE, is a very promising tool for massive parallel graph mining applications(e.g., see [Papadimitriou and Sun 2008; Kang et al. 2009; Kang et al. 2010;

Kang et al. 2010; Kang et al. 2010]). Other advanced MapReduce-like systems include [Grossman and Gu 2008; Chaiken et al. 2008; Pike et al. 2005].

Parallel RDBMS systems, including Vertica and Aster Data, are based on traditional database systems and provide high performance using distributed processing and query optimization. They have strength in processing structured data. For detailed comparison of these two systems, see [Pavlo et al. 2009]. Again, none of the above articles shows how to use such platforms to efficiently compute the diameter or radii.

8. CONCLUSIONS

Our main goal is to develop an open-source package to mine Giga-byte, Tera-byte and eventually Peta-byte networks. We designed HADI, an algorithm for computing radii and diameter of Tera-byte scale graphs, and analyzed large networks to observe important patterns. The contributions of this paper are the following:

—*Design:* We developed HADI, a scalable MapReduce algorithm for diameter and radius estimation, on massive graphs.
—*Optimization:* Careful fine-tunings on HADI, leading to up to *7.6*× faster computation, linear scalability on the size of the graph (number of edges) and near-linear speed-up on the number of machines. The experiments ran on the M45 Hadoop cluster of Yahoo, one of the 50 largest supercomputers in the world.
—*Observations:* Thanks to HADI, we could study the diameter and radii distribution of one of the largest public web graphs ever analyzed (over 6 *billion* edges); we also observed the "Small Web" phenomenon, multi-modal/bi-modal radius distributions, and palindrome motions of radius distributions over time in real networks.

Future work includes algorithms for additional graph mining tasks like computing eigenvalues, and outlier detection, for graphs that span Tera- and Peta-bytes.

ACKNOWLEDGMENTS

This work was partially funded by the National Science Foundation under Grants No. IIS-0705359, IIS-0808661, CAPES (PDEE project number 3960-07-2), CNPq, Fapesp and under the auspices of the U.S. Dept. of Energy by Lawrence Livermore National Laboratory under contract DE-AC52-07NA27344. We would like to thank YAHOO! for the web graph and access to the M45, and Adriano A. Paterlini for feedback. The opinions expressed are those of the authors and do not necessarily reflect the views of the funding agencies.

REFERENCES

Aggarwal, C. C., Xie, Y., and Yu, P. S. 2009. Gconnect: A connectivity index for massive disk-resident graphs. *PVLDB*.

Albert, R., Jeong, H., and Barabasi, A.-L. 1999. Diameter of the world wide web. *Nature* 401, 130–131.

Alon, N., Matias, Y., and Szegedy, M. 1996. The space complexity of approximating the frequency moments.

Bader, D. A. and Madduri, K. 2008. A graph-theoretic analysis of the human protein-interaction network using multicore parallel algorithms. *Parallel Comput.*.

Becchetti, L., Boldi, P., Castillo, C., and Gionis, A. 2008. Efficient semi-streaming algorithms for local triangle counting in massive graphs. In *KDD*.

Beyer, K., Haas, P. J., Reinwald, B., Sismanis, Y., and Gemulla, R. 2007. On synopses for distinct-value estimation under multiset operations. SIGMOD.

Broder, A., Kumar, R., Maghoul, F., Raghavan, P., Rajagopalan, S., Stata, R., Tomkins, A., and Wiener, J. 2000. Graph structure in the web. *Computer Networks 33*.

Chaiken, R., Jenkins, B., Larson, P.-A., Ramsey, B., Shakib, D., Weaver, S., and Zhou, J. 2008. Scope: easy and efficient parallel processing of massive data sets. *VLDB*.

Chakrabarti, D., Papadimitriou, S., Modha, D. S., and Faloutsos, C. 2004. Fully automatic cross-associations. In *KDD*.

Charikar, M., Chaudhuri, S., Motwani, R., and Narasayya, V. 2000. Towards estimation error guarantees for distinct values. PODS.

Cormen, T., Leiserson, C., and Rivest, R. 1990. *Introduction to Algorithms*. The MIT Press.

Daruru, S., Marin, N. M., Walker, M., and Ghosh, J. 2009. Pervasive parallelism in data mining: dataflow solution to co-clustering large and sparse netflix data. In *KDD*.

Dhillon, I. S., Guan, Y., and Kulis, B. 2007. Weighted graph cuts without eigenvectors a multilevel approach. *IEEE TPAMT*.

Dhillon, I. S., Mallela, S., and Modha, D. S. 2003. Information-theoretic co-clustering. In *KDD*.

Erdős, P. and Rényi, A. 1959. On random graphs. *Publicationes Mathematicae*.

Ferrez, J.-A., Fukuda, K., and Liebling, T. 1998. Parallel computation of the diameter of a graph. In *HPCSA*.

Flajolet, P. and Martin, G. N. 1985. Probabilistic counting algorithms for data base applications. *Journal of Computer and System Sciences*.

Garofalakis, M. N. and Gibbon, P. B. 2001. Approximate query processing: Taming the terabytes. *VLDB*.

Grossman, R. L. and Gu, Y. 2008. Data mining using high performance data clouds: experimental studies using sector and sphere. *KDD*.

Kang, U., Chau, D., and Faloutsos, C. 2010. Inference of beliefs on billion-scale graphs. *The 2nd Workshop on Large-scale Data Mining: Theory and Applications*.

Kang, U., Tsourakakis, C., Appel, A. P., Faloutsos, C., and Leskovec., J. 2010. Radius plots for mining tera-byte scale graphs: Algorithms, patterns, and observations. *SIAM International Conference on Data Mining*.

Kang, U., Tsourakakis, C., and Faloutsos, C. 2010. Pegasus: Mining peta-scale graphs. *Knowledge and Information Systems*.

Kang, U., Tsourakakis, C. E., and Faloutsos, C. 2009. Pegasus: A peta-scale graph mining system - implementation and observations. *ICDM*.

Karypis, G. and Kumar, V. 1999. Parallel multilevel k-way partitioning for irregular graphs. *SIAM Review 41,* 2, 278–300.

Ke, Y., Cheng, J., and Yu, J. X. 2009. Top-k correlative graph mining. *SDM*.

Kolda, T. G. and Sun, J. 2008. Scalable tensor decompositions for multi-aspect data mining. In *ICDM*.

Leskovec, J., Chakrabarti, D., Kleinberg, J. M., and Faloutsos, C. 2005. Realistic, mathematically tractable graph generation and evolution, using kronecker multiplication. *PKDD*, 133–145.

Leskovec, J., Kleinberg, J., and Faloutsos, C. 2007. Graph evolution: Densification and shrinking diameters. *ACM Trans. Knowl. Discov. Data 1,* 1, 2.

Leskovec, J., Lang, K. J., Dasgupta, A., and Mahoney, M. W. 2008. Statistical properties of community structure in large social and information networks. In *WWW '08*.

Lewis., T. G. 2009. Network science: Theory and applications. *Wiley*.

Ma, J. and Ma, S. 1993. Efficient parallel algorithms for some graph theory problems. *JCST*.

Mcglohon, M., Akoglu, L., and Faloutsos, C. 2008. Weighted graphs and disconnected components: patterns and a generator. *KDD*.

Newman., M. 2005. A measure of betweenness centrality based on random walks. *Social Networks*.

Palmer, C. R., Gibbons, P. B., and Faloutsos, C. 2002. Anf: a fast and scalable tool for data mining in massive graphs. *KDD*, 81–90.

Papadimitriou, S. and Sun, J. 2008. Disco: Distributed co-clustering with map-reduce. *ICDM*.

Pavlo, A., Paulson, E., Rasin, A., Abadi, D. J., Dewitt, D. J., Madden, S., and Stonebraker, M. 2009. A comparison of approaches to large-scale data analysis. SIGMOD.

Pike, R., Dorward, S., Griesemer, R., and Quinlan, S. 2005. Interpreting the data: Parallel analysis with sawzall. *Scientific Programming Journal*.

Satuluri, V. and Parthasarathy, S. 2009. Scalable graph clustering using stochastic flows: applications to community discovery. *KDD*.

Sinha, B. P., Bhattacharya, B. B., Ghose, S., and Srimani, P. K. 1986. A parallel algorithm to compute the shortest paths and diameter of a graph and its vlsi implementation. *IEEE Trans. Comput.*.

Tsourakakis, C. E. 2008. Fast counting of triangles in large real networks without counting: Algorithms and laws. In *ICDM*. 608–617.

Tsourakakis, C. E. 2009. Mach: Fast randomized tensor decompositions. *CoRR abs/0909.4969*.

Tsourakakis, C. E., Kang, U., Miller, G. L., and Faloutsos, C. 2009. Doulion: Counting triangles in massive graphs with a coin. *KDD*.

Tsourakakis, C. E., Kolountzakis, M. N., and Miller, G. L. 2009. Approximate triangle counting. *CoRR abs/0904.3761*.

Wang, C., Wang, W., Pei, J., Zhu, Y., and Shi, B. 2004. Scalable mining of large disk-based graph databases. *KDD*.

Yan, X. and Han, J. 2002. gspan: Graph-based substructure pattern mining. *ICDM*.

You, C. H., Holder, L. B., and Cook, D. J. 2009. Learning patterns in the dynamics of biological networks. In *KDD*.