Mining Large Graphs: Algorithms, Inference, and Discoveries

U Kang [1], Duen Horng Chau [2], Christos Faloutsos [3]

School of Computer Science, Carnegie Mellon University
5000 Forbes Ave, Pittsburgh PA 15213, United States
[1]ukang@cs.cmu.edu
[2]dchau@cs.cmu.edu
[3]christos@cs.cmu.edu

***Abstract*—How do we find patterns and anomalies, on graphs with billions of nodes and edges, which do not fit in memory? How to use parallelism for such terabyte-scale graphs? In this work, we focus on *inference*, which often corresponds, intuitively, to "guilt by association" scenarios. For example, if a person is a drug-abuser, probably its friends are so, too; if a node in a social network is of male gender, his dates are probably females. We show how to do inference on such huge graphs through our proposed HADOOP LINE GRAPH FIXED POINT (HA-LFP), an efficient parallel algorithm for sparse billion-scale graphs, using the HADOOP platform.**

Our contributions include (a) the design of HA-LFP, observing that it corresponds to a fixed point on a *line graph* induced from the original graph; (b) scalability analysis, showing that our algorithm scales up well with the number of edges, as well as with the number of machines; and (c) experimental results on two private, as well as two of the largest publicly available graphs — the Web Graphs from Yahoo! (6.6 billion edges and *0.24 Tera bytes*), and the Twitter graph (3.7 billion edges and *0.13 Tera bytes*). We evaluated our algorithm using M45, one of the top 50 fastest supercomputers in the world, and we report patterns and anomalies discovered by our algorithm, which would be invisible otherwise.

***Index Terms*—HA-LFP, Belief Propagation, Hadoop, Graph Mining**

I. INTRODUCTION

Given a large graph, with millions or billions of nodes, how can we find patterns and anomalies? One method to do that is through "guilt by association": if we know that nodes of type "A" (say, males) tend to interact/date nodes of type "B" (females), we can infer the unknown gender of a node, by checking the gender of the majority of its contacts. Similarly, if a node is a telemarketer, most of its contacts will be normal phone users (and *not* telemarketers, or 800 numbers).

We show that the "guilt by association" approach can find useful patterns and anomalies, in large, real graphs. The typical way to handle this is through the so-called *Belief Propagation (BP)* [1], [2]. BP has been successfully used for social network analysis, fraud detection, computer vision, error-correcting codes [3], [4], [5], and many other domains. In this work, we address the research challenge of *scalability* — we show how to run BP on a very large graph with billions of nodes and edges. Our contributions are the following:

1) We observe that the Belief Propagation algorithm is essentially a recursive equation on the *line graph* induced from the original graph. Based on this observation, we formulate the BP problem as finding a fixed point on the line graph. We propose the LINE GRAPH FIXED POINT (LFP) algorithm and show that it is a generalized form of a linear algebra equation.
2) We formulate and devise an efficient algorithm for the LFP that runs on the HADOOP platform, called HADOOP LINE GRAPH FIXED POINT (HA-LFP).
3) We run experiments on a HADOOP cluster and analyze the running time. We analyze the large real-world graphs including YahooWeb and Twitter with HA-LFP, and show patterns and anomalies.

The rest of the paper is organized as follows. Section II discusses the related works on the Belief Propagation and HADOOP. Section III describes our formulation of the Belief Propagation in terms of LINE GRAPH FIXED POINT (LFP), and Section IV provides a fast algorithm in HADOOP. Section V shows the scalability results, and Section VI gives the results of analyzing the large, real-world graphs. We conclude in Section VII.

To enhance readability of this paper, we have listed the symbols frequently used in this paper in Table I. The reader may want to return to this table throughout this paper for a quick reference of their meanings.

Symbol	**Definition**
V	Set of nodes in a graph
E	Set of edges in a graph
n	Number of nodes in a graph
l	Number of edges in a graph
S	Set of states
$\phi_i(s)$	Prior of node i being in state s
$\psi_{ij}(s', s)$	Edge potential when nodes i and j being in states s' and s, respectively
$m_{ij}(s)$	Message that node i sends to node j expressing node i's belief of node j's being in state s
$b_i(s)$	Belief of node i being in state s

TABLE I
TABLE OF SYMBOLS

II. Background

The related work forms two groups, Belief Propagation(BP) and large graph mining with MapReduce/Hadoop.

A. Belief Propagation(BP)

Belief Propagation(BP) [1] is an efficient inference algorithm for probabilistic graphical models. Since its proposal, it has been widely, and successfully, used in a myriad of domains to solve many important problems (some are seemingly unrelated at the first glance). For example, BP is used in some of the best error-correcting codes, such as the *Turbo code* and *low-density parity-check* code, that approach channel capacity. In computer vision, BP is among the top contenders for stereo shape estimation and image restoration (e.g., denoising) [3]. BP has also been used for fraud detection, such as for unearthing fraudsters and their accomplices lurking in online auctions [4], and pinpointing misstated accounts in general ledger data for the financial domain [5].

BP is typically used for computing the *marginal distribution* for the unobserved nodes in a graph, conditional on the observed ones; we will only discuss this version in this paper, though with slight and trivial modifications to our implementation, the *most probable distribution* of node states can also be computed.

BP was first proposed for *trees* [1] and it could compute the exact marginal distributions; it was later applied on general graphs [6] as an approximate algorithm. When the graph contains cycles or loops, the BP algorithm applied on it is called *loopy BP*, which is also the focus of this work.

BP is generally applied on graphs whose nodes have finite number of states (treating each node as a *discrete* random variable). Gaussian BP is a variant of BP where its underlying distributions are Gaussian [7]. Generalized BP [2] allows messages to be passed between subgraphs, which can improve accuracy in the computed beliefs and promote convergence.

BP is computationally-efficient; its running time scales linearly with the number of edges in the graph. However, for graphs with *billions* of nodes and edges — a focus of our work — this cost becomes significant. There are several recent works that investigated parallel BP on multicore shared memory [8] and MPI [9], [10]. However, all of them assume the graphs would fit in the main memory (of a single computer, or a computer cluster). Our work specifically tackles the important, and increasingly prevalent, situation where the graphs would not fit in main memory.

B. Large Graph Mining with MapReduce and Hadoop

Large scale graph mining poses challenges in dealing with massive amount of data. One might consider using a sampling approach to decrease the amount of data. However, sampling from a large graph can lead to multiple nontrivial problems that do not have satisfactory solutions [11]. For example, which sampling methods should we use? Should we get a random sample of the edges, or the nodes? Both options have their own share of problems: the former gives poor estimation of the graph diameter, while the latter may miss high-degree nodes.

A promising alternative for large graph mining is MapReduce, a parallel programming framework [12] for processing web-scale data. MapReduce has two advantages: (a) The data distribution, replication, fault-tolerance, load balancing is handled automatically; and furthermore (b) it uses the familiar concept of functional programming. The programmer needs to define only two functions, a *map* and a *reduce*. The general framework is as follows [13]: (a) the *map* stage reads the input file and emits (key, value) pairs; (b) the *shuffling* stage sorts the output and distributes them to reducers; (c) the *reduce* stage processes the values with the same key and emits another (key, value) pairs which become the final result.

Hadoop [14] is the open source version of MapReduce. Hadoop uses its own distributed file system HDFS, and provides a high-level language called PIG [15]. Due to its excellent scalability and ease of use, Hadoop is widely used for large scale data mining(see [16] [17] [18] [19]). Other variants which provide advanced MapReduce-like systems include SCOPE [20], Sphere [21], and Sawzall [22].

III. Proposed Method

In this section, we describe Line graph Fixed Point (Lfp), our proposed parallel formulation of the BP on Hadoop. We first describe the standard BP algorithm, and then explains our method in detail.

A. Belief Propagation

We provide a quick overview of the Belief Propagation(BP) algorithm, which briefly explains the key steps in the algorithm and their formulation; this information will help our readers better understand how our implementation nontrivially captures and optimizes the algorithm in latter sections. For detailed information regarding BP, we refer our readers to the excellent article by Yedidia et al. [2].

The BP algorithm is an efficient method to solve inference problems for probabilistic graphical models, such as Bayesian networks and pairwise Markov random fields (MRF). In this work, we focus on pairwise MRF, which has seen empirical success in many domains (e.g., Gallager codes, image restoration) and is also simpler to explain; the BP algorithms for other types of graphical models are mathematically equivalent [2].

When we view an undirected simple graph $G = (V, E)$ as a pairwise MRF, each node i in the graph becomes a random variable X_i, which can be in a discrete number of states S. The goal of the inference is to find the marginal distribution $P(x_i)$ for all node i, which is an NP-complete problem.

Fortunately, BP may be used to solve this problem approximately (for MRF; exactly for trees). At a high level, BP infers the "true" (or so-called "hidden") distribution of a node from some prior (or "observed") knowledge about the node, and from the node's neighbors. This is accomplished through iterative message passing between all pairs of nodes v_i and v_j. We use $m_{ij}(x_j)$ to denote the message sent from i to j, which intuitively represents i's opinion about j's likelihood of being in state x_j. The prior knowledge about a node i, or the prior probabilities of the node being in each possible state are expressed through the *node potential function* $\phi(x_i)$. This prior probability may simply be called a *prior*. The message-passing procedure stops if the messages no longer change much from one iteration to the another — or equivalently when the nodes' marginal probabilities are no longer changing much. The estimated marginal probability is called *belief*, or symbolically $b_i(x_i)$ ($\approx P(x_i)$).

In detail, messages are obtained as follows. Each edge e_{ij} is associated with messages $m_{ij}(x_j)$ and $m_{ji}(x_i)$ for each possible state. Provided that all messages are passed in every iteration, the order of passing can be arbitrary. Each message vector m_{ij} is normalized to sum to one. Normalization also prevents numerical underflow (or zeroing-out values). Each outgoing message from a node i to a neighbor j is generated based on the incoming messages from the node's other neighbors. Mathematically, the message-update equation is:

$$m_{ij}(x_j) = \sum_{x_i} \phi_i(x_i)\psi_{ij}(x_i, x_j)\frac{\prod_{k\in N(i)} m_{ki}(x_i)}{m_{ji}(x_i)} \quad (1)$$

where $N(i)$ is the set of nodes neighboring node i, and $\psi_{ij}(x_i, x_j)$ is called the *edge potential*; intuitively, it is a function that *transforms* a node's incoming messages collected into the node's outgoing ones. Formally, $\psi_{ij}(x_i, x_j)$ equals the probability of a node i being in state x_i and that its neighbor j is in state x_j.

The algorithm stops when the beliefs converge (within some threshold, e.g., 10^{-5}), or a maximum number of iterations has finished. Although convergence is not guaranteed theoretically for general graphs, except for those that are trees, the algorithm often converges in practice, where convergence is quick and the beliefs are reasonably accurate. When the algorithm ends, the node beliefs are determined as follows:

$$b_i(x_i) = c\phi_i(x_i)\prod_{k\in N(i)} m_{ki}(x_i) \quad (2)$$

where c is a normalizing constant.

B. Recursive Equation

As seen in the last section, BP is computed by iteratively running equations (1) and (2), as described in Algorithm 1.

In a shared-memory system in which random access to memory is allowed, the implementation of Algorithm 1 might be straightforward. However, large scale algorithm for MAPREDUCE requires careful thinking since the random access is not allowed and the data are read sequentially within mappers and reducers. A good news is that the two equations (1) and (2) involve only local communications between neighboring nodes, and thus it seems hopeful to develop a parallel algorithm for HADOOP. Naturally, one might think of an iterative algorithm in which nodes exchange messages to update its beliefs using an extended form of matrix-vector multiplication [17]. In such formulation, a current belief vector and the message matrix is combined to compute the next belief vector. Thus, we want a recursive equation to update the belief

Algorithm 1: Belief Propagation

```
Input  : Edge $E$,
         node prior $\phi^{n\times 1}$, and
         propagation matrix $\psi^{S\times S}$
Output: Belief matrix $b^{n\times S}$

1 begin
2     while $m$ does not converge do
3         for $(i,j) \in E$ do
4             for $s \in S$ do
5                 $m_{ij}(s) \leftarrow$
                  $\sum_{s'} \phi_i(s')\psi_{ij}(s',s)\prod_{k\in N(i)\backslash j} m_{ki}(s')$;
6     for $i \in V$ do
7         for $s \in S$ do
8             $b_i(s) \leftarrow c\phi_i(s)\prod_{k\in N(i)} m_{ki}(s)$;
9 end
```

vector. However, such an equation cannot be derived due to the denominator $m_{ji}(x_i)$ in Equation (1). If it were not for the denominator, we could get the following modified equation where the superscript t and $t-1$ mean the iteration number:

$$\begin{aligned} m_{ij}(x_j)^{(t)} &= \sum_{x_i} \phi_i(x_i)\psi_{ij}(x_i, x_j)\prod_{k\in N(i)} m_{ki}(x_i)^{(t-1)} \\ &= \sum_{x_i} \psi_{ij}(x_i, x_j)\frac{b_i(x_i)^{(t-1)}}{c} \end{aligned}$$

and thus

$$\begin{aligned} b_i(x_i)^{(t)} &= c\phi_i(x_i)\prod_{k\in N(i)} m_{ki}(x_i)^{(t-1)} \\ &= \phi_i(x_i)\prod_{k\in N(i)}\sum_{x_k}\psi_{ki}(x_k, x_i)b_k(x_k)^{(t-2)} \quad (3) \end{aligned}$$

Notice that the recursive equation (3) is a fake, imaginary equation derived from the assumption that equation (1) has no denominator. Although the recursive equation for the belief vector cannot be acquired by this way, there is a more direct and intuitive way to get a recursive equation. We will describe how to get it in the next section.

C. Main Idea: Line graph Fixed Point(LFP)

How can we get the recursive equation for the BP? What we need is a tractable recursive equation well-suited for large scale MAPREDUCE framework. In this section, we describe LINE GRAPH FIXED POINT (LFP), our formulation of BP in terms of finding the fixed point of an induced graph from the original graph. As seen in the last section, a recursive equation to update the beliefs cannot be acquired due to the denominator in the message update equation. Our main idea to solve the problem is to flip the notion of the nodes and edges in the original graph and thus use the equation (1), without modification, as the recursive equation for updating the 'nodes' in the new formulation. The 'flipping' means we consider an induced graph, called the *line graph*, whose nodes correspond to edges in the original graph, and the two nodes in the induced graph are connected if the corresponding edges in the original graph are incident. Notice that for each edge (i, j) in the original graph, two messages need to be defined since m_{ij} and m_{ji} are different. Thus, the line graph should be directed, although the original graph is undirected. Formally, we define the 'directed line graph' as follows.

Definition 1 (Directed Line Graph): Given a directed graph G, its directed line graph L(G) is a graph such that each node of L(G) represents an edge of G, and there is an edge from v_i to v_j of L(G) if the corresponding edges e_i and e_j form a length-two directed path from e_i to e_j in G.

For example, see Figure 1 for a graph and its directed line graph. To convert a undirected line graph G to a directed line graph $L(G)$, we first convert G to a directed graph by converting each undirected edge to two directed edges. Then, a directed edge from v_i to v_j in $L(G)$ is created if their corresponding edges e_i and e_j form a directed path e_i to e_j in G.

Now, we derive the exact recursive equation on the line graph. Let G be the original undirected graph with n nodes and l edges, and $L(G)$ be the directed line graph of G with $2l$ nodes as defined by Definition 1. The (i, j)th element $L(G)_{i,j}$ is defined to be 1 if the edge exist, or 0 otherwise. Let m be a $2l$-vector whose element corresponding to the edge (i, j) in G contains the reverse directional message m_{ji}. The reason of this reverse directional message will be described soon. Let ϕ be a n-vector containing priors of each node. We build a $2l$-vector φ as follows: if the kth element φ_k of φ corresponds to an edge (i, j) in G, then set φ_k to $\phi(i)$. A standard matrix-vector multiplication with vector addition operation on $L(G)$, m, φ is

$m' = L(G) \times m + \varphi$
where
$m'_i = \sum_{j=1}^{n} L(G)_{i,j} \times m_j + \varphi_i$.

In the above equation, four operations are used to get the result vector:

1) `combine2`$(L(G)_{i,j}, m_j)$: multiply $L(G)_{i,j}$ and m_j.
2) `combineAll`$_i(y_1, ..., y_n)$: sum n multiplication results for node i.
3) `sumVector`(φ_i, v_{aggr}): add φ_i to the result v_{aggr} of `combineAll`.
4) `assign`$(m_i, oldval_i, newval_i)$: overwrite the previous value $oldval_i$ of m_i with the new value $newval_i$ to make m'_i.

Now, we generalize the operators $\times$ and $+$ to $\times_G$ and $+_G$, respectively, so that the four operations can be any functions of their arguments. In this generalized setting, the matrix-vector multiplication with vector addition operation becomes

$m' = L(G) \times_G m +_G \varphi$
where
m'_i = `assign`$(m_i, oldval_i,$ `sumVector`$(\varphi_i,$`combineAll`$_i(\{y_j \mid j = 1..n,$ and $y_j =$`combine2`$(L(G)_{i,j}, m_j)\})))$.

An important observation is that the BP equation (1) can be represented by this generalized form of the matrix-vector multiplication with vector addition. For simplifying the explanation, we omit the edge potential ψ_{ij} since it is a tiny information(e.g. 2 by 2 or 3 by 3 table), and the summation over x_i, both of which can be accommodated easily. Then, the BP equation (1) is expressed by

$$m' = L(G)^T \times_G m +_G \varphi \quad (4)$$
$$m'' = ChangeMessageDirection(m') \quad (5)$$

where
m'_i = `sumVector`$(\varphi_i,$`combineAll`$_i(\{y_j \mid j = 1..n,$ and $y_j =$`combine2`$(L(G)^T_{i,j}, m_j)\}))$
, the four operations are defined by

1) `combine2`$(L(G)_{i,j}, m_j) = L(G)_{i,j} \times m_j$
2) `combineAll`$_i(y_1, ..., y_n) = \prod_{j=1}^{n} y_j$
3) `sumVector`$(\varphi_i, v_{aggr}) = \varphi_i \times v_{aggr}$
4) `assign`$(m_i, oldval_i, newval_i) = newval_i/val_i$

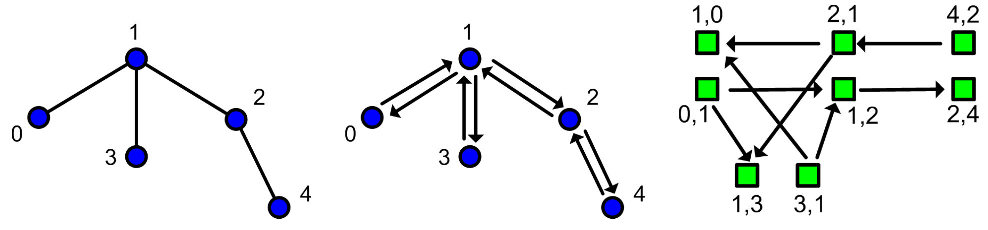
, and the ChangeMessageDirection function is defined by Algorithm 2. The computed m'' of equation (5) is the updated message which can be used as m in the next iteration. Thus, our LINE GRAPH FIXED POINT (LFP) comprises running the equation (4) and (5) iteratively until a fixed point, where the message vector converges, is found.

Two details should be addressed for the complete description of our method. First, notice that $L(G)^T$, instead of $L(G)$, is used in the equation (4). The reason is that a message should aggregate other messages pointing *to* itself, which is the reverse direction of the line graph construction. Second, what is the use of ChangeMessageDirection function? We mentioned earlier that the bp equation (1) contained a denominator m_{ji} which is the reverse directional message. Thus, the input message vector m of equation (4) contains the reverse directional message. However, the result message vector m' of equation (4) contains the forward directional message. For the m' to be used in the next iteration, it needs to change the direction of the messages, and that is what ChangeMessageDirection does.

Algorithm 2: ChangeMessageDirection

```
Input: message vector $m$ of length $2l$
Output: new message vector $m'$ of length $2l$
1: for $k \in 1..2l$ do
2:     $(i, j) \leftarrow$ edge in $G$ corresponding to $m_k$;
3:     $k' \leftarrow$ element index of $m$ corresponding to the edge $(j, i)$ in $G$
4:     $m'_{k'} \leftarrow m_k$
5: end for
```

In sum, a generalized matrix-vector multiplication with addition is the recursive message update equation which is run



(a) Original graph (b) Directed graph (c) Directed line graph

Fig. 1. Converting a undirected graph to a directed line graph. (a to b): replace a undirected edge with two directed edges. (b to c): for an edge (i, j) in (b), make a node (i, j) in (c). Make a directed edge from (i, j) to (k, l) in (c) if $j = k$ and $i \neq l$. The rectangular nodes in (c) corresponds to edges in (b).

until convergence. The resulting algorithm LFP is summarized in Algorithm 3.

Algorithm 3: LINE GRAPH FIXED POINT (LFP)

```
Input  : Edge $E$ of a undirected graph $G = (V, E)$,
         node prior $\phi^{n \times 1}$, and
         propagation matrix $\psi^{S \times S}$
Output: Belief matrix $b^{n \times S}$
1  begin
2      $L(G) \leftarrow$ directed line graph from $E$;
3      $\varphi \leftarrow$ line prior vector from $\phi$;
4      while m does not converge do
5          for $s \in S$ do
6              $m(s)^{next} = L(G) \times_G m^{cur} +_G \varphi$;
7      for $i \in V$ do
8          for $s \in S$ do
9              $b_i(s) \leftarrow c\phi_i(s) \prod_{j \in N(i)} m_{ji}(s)$;
10 end
```

IV. FAST ALGORITHM FOR HADOOP

In this section, we first describe the naive algorithm for LFP and propose an efficient algorithm.

A. Naive Algorithm

The formulation of BP in terms of the fixed point in the line graph provides an intuitive way to understand the computation. However, a naive algorithm without careful design is not efficient for the following reason. In a naive algorithm, we first build the matrix for the line graph $L(G)$ and the message vector, and apply the recursive equation on them. The problem is that a node in G with degree d will generate $d(d-1)$ edges in $L(G)$. Since there exists many nodes with a very large degree in real-world graphs due to the well-known power-law degree distribution, the number of nonzero elements will grow too large. For example, the YahooWeb graph in Section V has several nodes with the several-million degree. As a result, the number of nonzero elements in the corresponding line graph is more than 1 trillion. Thus, we need an efficient algorithm for dealing with the problem.

B. Lazy Multiplication

The main idea to solve the problem in the previous section is not to build the line graph explicitly: instead, we do the same computation on the original graph, or perform a 'lazy' multiplication. The crucial observation is that the edges in the original graph G contain all the edge information in $L(G)$: each edge $e \in E$ of G is a node in $L(G)$, and $e_1, e_2 \in G$ are adjacent in $L(G)$ if and only if they share the node in G. For each edge (i, j) in G, we associate the reverse message m_{ji}. Then, grouping edges by source node id i enables us to get all the messages pointing *to* the source node. Thus, for each node j of i's neighbors, the updated message m_{ij} is computed by calculating $\frac{\prod_{k \in N(i)} m_{ki}(x_i)}{m_{ji}(x_i)}$ from the messages in the grouped edges (incorporating priors and the propagation matrix is described soon). Since we associate the reverse message for each edge, the output triple (src, dst, reverse message) is (j, i, m_{ij}).

An issue in computing $\frac{\prod_{k \in N(i)} m_{ki}(x_i)}{m_{ji}(x_i)}$ is that a straightforward implementation requires $N(i)(N(i) - 1)$ multiplication which is prohibitively large. However, we decrease the number of multiplication to $2N(i)$ by first computing $t = \prod_{k \in N(i)} m_{ki}(s')$, and for each $j \in N(i)$ computing $t/m_{ji}(s')$.

The only remaining pieces of the computation is to incorporate the prior ϕ and the propagation matrix ψ. The propagation matrix ψ is a tiny bit of information, so it can be sent to every reducer by a variable passing functionality of HADOOP. The prior vector ϕ can be large, since the length of the vector can be the number of nodes in the graph. In the HADOOP algorithm, we also group the ϕ by the node id: each node prior is grouped together with the edges(messages) whose source id is the node id. Algorithm 4 shows the high-level algorithm of HADOOP LINE GRAPH FIXED POINT (HA-LFP). Algorithm 5 shows the BP message initialization algorithm which requires only a Map function. Algorithm 6 shows the HADOOP algorithm for the message update which implements the algorithm described above. After the messages converge, the final belief is computed by Algorithm 7.

C. Analysis

We analyze the time and the space complexity of HA-LFP. The main result is that one iteration of the message update on the line graph has the same complexity as one matrix-vector

Algorithm 4: HADOOP LINE GRAPH FIXED POINT (HA-LFP)

```
Input  : Edge $E$ of a undirected graph $G = (V, E)$,
         node prior $\phi^{n\times 1}$, and
         propagation matrix $\psi^{S\times S}$
Output: Belief matrix $b^{n\times S}$
begin
    Initialization(); // Algorithm 5
    while $m$ does not converge do
        MessageUpdate(); // Algorithm 6
    BeliefComputation(); // Algorithm 7
end
```

Algorithm 5: HA-LFP Initialization

```
Input  : Edge $E = \{(id_{src}, id_{dst})\}$,
         Set of states $S = \{s_1, ..., s_p\}$
Output: Message Matrix $M =$
         $\{(id_{src}, id_{dst}, m_{dst,src}(s_1), ..., m_{dst,src}(s_p))\}$
Initialization-Map(Key k, Value v);
begin
    Output(($k, v$), ($\frac{1}{|S|}$, ..., $\frac{1}{|S|}$));    // (k: $id_{src}$, v: $id_{dst}$)
end
```

Algorithm 6: HA-LFP Message Update

```
Input  : Set of states $S = \{s_1, ..., s_p\}$,
         Current Message Matrix $M^{cur} =$
         $\{(sid, did, m_{did,sid}(s_1), ..., m_{did,sid}(s_p))\}$,
         Prior Matrix $\Phi = \{(id, \phi_{id}(s_1), ..., \phi_{id}(s_p))\}$,
         Propagation Matrix $\psi$
Output: Updated Message Matrix $M^{next} =$
         $\{(id_{src}, id_{dst}, m_{dst,src}(s_1), ..., m_{dst,src}(s_p))\}$
MessageUpdate-Map(Key k, Value v);
begin
    if $(k, v)$ is of type $M$ then
        Output($k, v$);    // (k: $sid$, v: $did, m_{did,sid}(s_1), ..., m_{did,sid}(s_p)$)

    else if $(k, v)$ is of type $\Phi$ then
        Output($k, v$);    // (k: $id$, v: $\phi_{id}(s_1), ..., \phi_{id}(s_p)$)

end
MessageUpdate-Reduce(Key k, Value v[1..r]);
begin
    $temps[1..p] \leftarrow [1..1]$;
    $saved\_prior \leftarrow [\ ]$;
    HashTable<int, double[1..p]> $h$;
    foreach $v \in v[1..r]$ do
        if $(k, v)$ is of type $\Phi$ then
            $saved\_prior[1..p] \leftarrow v$;

        else if $(k, v)$ is of type $M$ then
            $(did, m_{did,sid}(s_1), ..., m_{did,sid}(s_p)) \leftarrow v$;
            $h.add(did, (m_{did,sid}(s_1), ..., m_{did,sid}(s_p)))$;
            foreach $i \in 1..p$ do
                $temps[i] = temps[i] \times m_{did,sid}(s_i)$;

    foreach $(did, (m_{did,sid}(s_1), ..., m_{did,sid}(s_p))) \in h$ do
        $outm[1..p] \leftarrow 0$;
        foreach $u \in 1..p$ do
            foreach $v \in 1..p$ do
                $outm[u] = outm[u] + saved\_prior[v]\psi(v, u)temps[v]/m_{did,sid}(s_v)$;
        Output($did, (sid, outm[1], ..., outm[p])$);
end
```

multiplication on the original graph. In the lemmas below, M is the number of machines.

*Lemma 1 (**Time Complexity of** **HA-LFP**):* One iteration of HA-LFP takes $O(\frac{V+E}{M}log\frac{V+E}{M})$ time. It could take $O(\frac{V+E}{M})$ time if HADOOP uses only hashing, not sorting, on its shuffling stage.

Proof: Notice that the number of states is usually very small(2 or 3), thus can be considered as a constant. Assuming uniform distribution of data to machines, the time complexity is dominated by the MessageUpdate job. Thanks to the 'lazy multiplication' described in the previous section, both Map and Reduce takes linear time to the input. Thus, the time complexity is $O(\frac{V+E}{M}log\frac{V+E}{M})$, which is the sorting time for $\frac{V+E}{M}$ records. It could be $O(\frac{V+E}{M})$, if HADOOP performs only hashing without sorting on its shuffling stage. ■

A similar results holds for space complexity.

*Lemma 2 (**Space Complexity of** **HA-LFP**):* HA-LFP requires $O(V + E)$ space.

Proof: The prior vector requires $O(V)$ space, and the message matrix requires $O(2E)$ space. Since the number of edges is greater than the number of nodes, HA-LFP requires $O(V + E)$ space, in total. ■

V. EXPERIMENTS

In this section, we present experimental results to answer the following questions:

Q1 How fast is HA-LFP, compared to a single-machine disk-based Belief Propagation algorithm?

Q2 How does HA-LFP scale up on the number of machines?

Q3 How does HA-LFP scale up on the number of edges?

We performed experiments in the M45 HADOOP cluster by Yahoo!. The cluster has total 480 machines with 1.5 Petabyte total storage and 3.5 Terabyte memory. The single-machine experiment was done in a machine with 3 Terabyte of disk and 48 GB memory. The single-machine BP algorithm is a scaled-up version of a memory-based BP which reads all the nodes, not the edges, into a memory. That is, the single-machine BP

Algorithm 7: HA-LFP Belief Computation

```
Input  : Set of states $S = \{s_1, ..., s_p\}$,
         Current Message Matrix $M^{cur} = \{(sid, did, m_{did,sid}(s_1), ..., m_{did,sid}(s_p))\}$,
         Prior Matrix $\Phi = \{(id, \phi_{id}(s_1), ..., \phi_{id}(s_p))\}$
Output: Belief Vector $b = \{(id, b_{id}(s_1), ..., b_{id}(s_p))\}$
BeliefComputation-Map(Key k, Value v);
begin
    if $(k, v)$ is of type $M$ then
        Output($k, v$);   // (k: $sid$, v: $did, m_{did,sid}(s_1), ..., m_{did,sid}(s_p)$)

    else if $(k, v)$ is of type $\Phi$ then
        Output($k, v$);   // (k: $id$, v: $\phi_{id}(s_1), ..., \phi_{id}(s_p)$)

end
BeliefComputation-Reduce(Key k, Value v[1..r]);
begin
    $b[1..p] \leftarrow [1..1]$;
    foreach $v \in v[1..r]$ do
        if $(k, v)$ is of type $\Phi$ then
            $prior[1..p] \leftarrow v$;
            foreach $i \in 1..p$ do
                $b[i] = b[i] \times prior[i]$;

        else if $(k, v)$ is of type $M$ then
            $(did, m_{did,sid}(s_1), ..., m_{did,sid}(s_p)) \leftarrow v$;
            foreach $i \in 1..p$ do
                $b[i] = b[i] \times m_{did,sid}(s_i)$;


    Output($k, (b[1], ..., b[p])$);
end
```

loads only the node information into a memory, but it reads the edges sequentially from the disk for every message update, instead of loading all the edges into a memory once for all.

The graphs we used in our experiments at Section V and VI are summarized in Table II [1], with the following details.

- YahooWeb: web pages and their links, crawled by Yahoo! at year 2002.
- Twitter: social network(who follows whom) extracted from Twitter, at June 2010 and Nov 2009.
- Kronecker: synthetic Kronecker graph [23] with similar properties as real-world graphs.
- VoiceCall: phone call records(who calls whom) during Dec. 2007 to Jan. 2008 from an anonymous phone service provider.
- SMS: short message service records(who sends to whom) during Dec. 2007 to Jan. 2008 from an anonymous phone service provider.

[1]YahooWeb: released under NDA.
Twitter: http://www.twitter.com
Kronecker: [23]
VoiceCall, SMS: not public data.

Graph	Nodes	Edges	File	Desc.
YahooWeb	1,413 M	6,636 M	0.24 TB	page-page
Twitter'10	104 M	3,730 M	0.13 TB	person-person
Twitter'09	63 M	1,838 M	56 GB	person-person
Kronecker	177 K	1,977 M	25 GB	synthetic
	120 K	1,145M	13.9 GB	
	59 K	282 M	3.3 GB	
VoiceCall	30 M	260 M	8.4 GB	who calls whom
SMS	7 M	38 M	629 MB	who sends to whom

TABLE II
ORDER AND SIZE OF NETWORKS. M: MILLON.

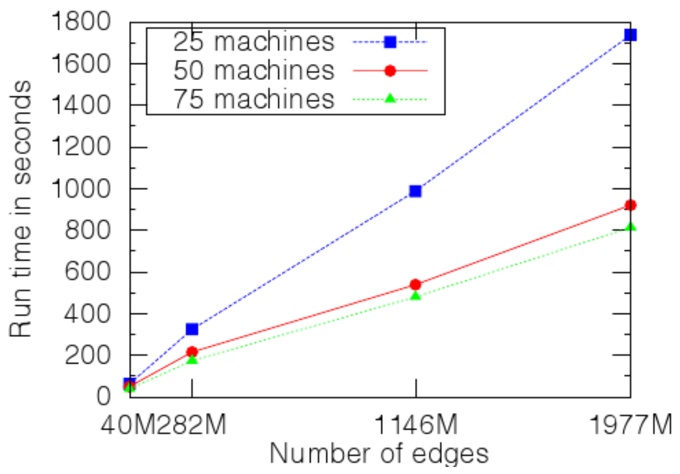
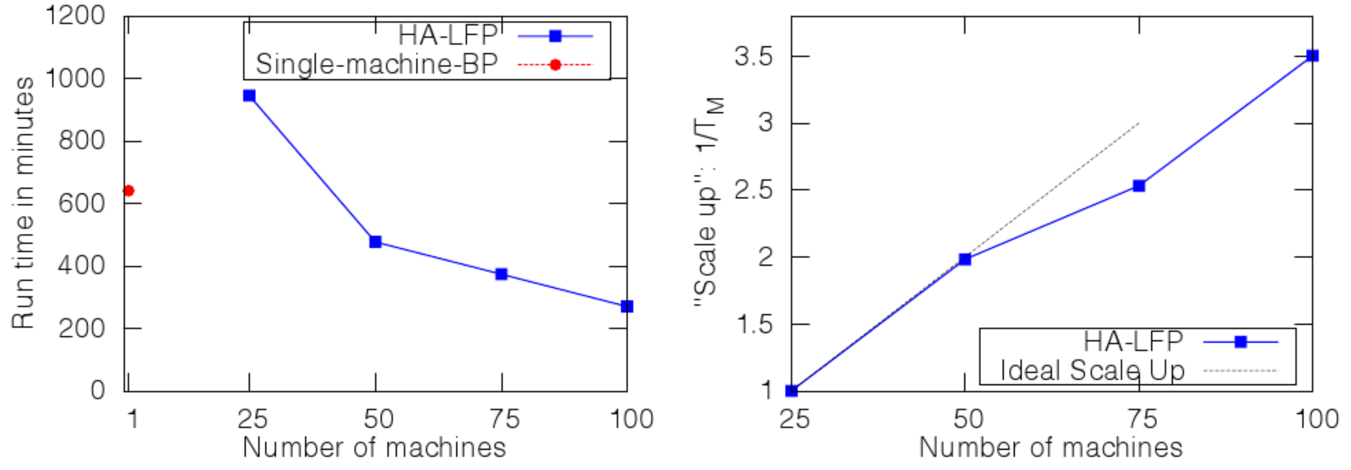


Fig. 3. Running time of 1 iterations of message update in HA-LFP on Kronecker graphs. Notice that the running time scales-up linear to the number of edges.

A. Results

Between HA-LFP and the single-machine BP, which one runs faster? At which point does the HA-LFP outperform the single-machine BP? Figure 2 (a) shows the comparison of running time of the HA-LFP and the single-machine BP. Notice that HA-LFP outperforms the single-machine BP when the number of machines exceeds 40. The HA-LFP requires more machines to beat the single-machine BP due to the fixed costs for writing and reading the intermediate results to and from the disk. However, for larger graphs whose nodes do not fit into a memory, HA-LFP is the only solution to the best of our knowledge.

The next question is, how does our HA-LFP scale up on the number of machines and edges? Figure 2 (b) shows the scalability of HA-LFP on the number of machines. We see that our HA-LFP scales up linearly close to the ideal scale-up. Figure 3 shows the linear scalability of HA-LFP on the number of edges.



(a) Running Time (b) Scale-Up with Machines

Fig. 2. Running time of HA-LFP with 10 iterations on the YahooWeb graph with 1.4 billion nodes and 6.7 billion edges. **(a)** Comparison of the running times of HA-LFP and the single-machine BP. Notice that HA-LFP outperforms the single-machine BP when the number of machines exceed ≈40. **(b)** "Scale-up" (throughput $1/T_M$) versus number of machines M, for the YahooWeb graph. Notice the near-linear scale-up close to the ideal(dotted line).

B. Discussion

Based on the experimental results, what are the advantages of HA-LFP? In what situations should it be used? For a small graph whose nodes and edges fit in the memory, the single-machine BP is recommended since it runs faster. For a medium-to-large graph whose nodes fit in the memory but the edges do not fit in the memory, HA-LFP gives the reasonable solution since it runs faster than the single-machine BP. For a very large graph whose nodes do not fit in the memory, HA-LFP is the only solution. We summarize the advantages of the HA-LFP here:

- **Scalability**: HA-LFP is *the only* solution when the nodes information can not fit in memory. Moreover, HA-LFP scales up near-linearly.
- **Running Time**: Even for a graph whose node information fits into a memory, HA-LFP ran 2.4 times faster.
- **Fault Tolerance**: HA-LFP enjoys the fault tolerance that HADOOP provides: data are replicated, and the failed programs due to machine errors are restarted in working machines.

VI. ANALYSIS OF REAL GRAPHS

In this section, we analyze real-world graphs using HA-LFP and show important findings.

A. HA-LFP on YahooWeb

Given a web graph, how can we separate the educational('good') web pages from the adult('bad') web pages? Manually investigating billions of web pages would take so much time and efforts. In this section, we show how to do it using HA-LFP. We use a simple heuristic to set priors: the web pages which contain 'edu' have high goodness prior(0.95), and the web pages which contain either 'sex', 'adult', or 'porno' have low goodness prior(0.05). Among 11.8 million web pages containing sexually explicit keywords, we keep 10% of the pages as a validation set (goodness prior 0.5), and use the rest 90% as a training set by setting the goodness prior 0.05. Also, among 41.7 million web pages containing 'edu', we randomly sample 11.8 million web pages, so that the number equals with that of adult pages given prior, and use 10% as a validation set(goodness prior 0.5), and use the rest 90% as a training set(goodness prior 0.95). The edge potential function is given by Table III. It is given by our observation that good pages tend to point to other good pages, while bad pages might point to good pages, as well as bad pages, to boost their ranking in web search engines.

	Good	Bad
Good	1-ϵ	ϵ
Bad	0.5	0.5

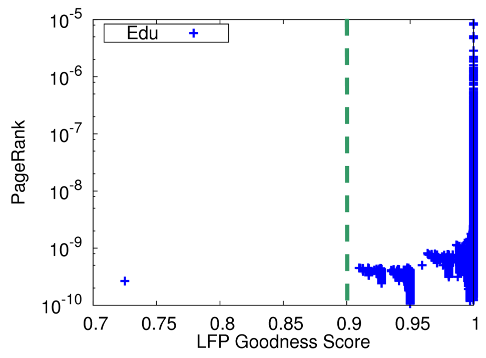
TABLE III
EDGE POTENTIAL FOR THE YAHOOWEB. ϵ IS SET TO 0.05 IN THE EXPERIMENTS. GOOD PAGES POINT TO OTHER GOOD PAGES WITH HIGH PROBABILITY. BAD PAGES POINT TO BAD PAGES, BUT ALSO GOOD PAGES WITH EQUAL CHANCES, TO BOOST THEIR RANK IN WEB SEARCH ENGINES.

Figure 4 shows the HA-LFP scores and the number of pages in the test set having such scores. Notice that almost all the pages with LFP score less than 0.9 in our test data contain adult web sites. Thus, the LFP score 0.9 can be used as a decision boundary for adult web pages.

Figure 5 shows the HA-LFP scores vs. PageRank scores of pages in our test set. We see that the PageRank cannot be used for differentiating between educational and adult web pages. However, HA-LFP can be used to spotting adult web pages, by using the threshold 0.9.

B. HA-LFP on Twitter and VoiceCall

We run HA-LFP on Twitter and VoiceCall data which are both social networks representing who follows whom or who



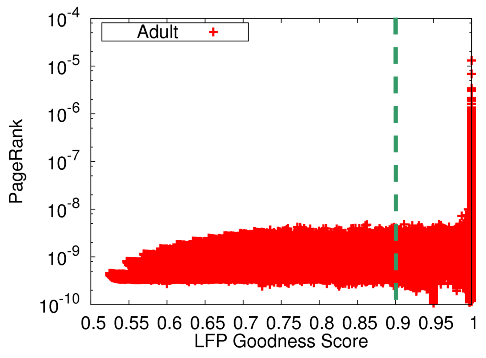


Fig. 5. HA-LFP scores vs. PageRank scores of pages in our test set. The vertical dashed line is the same decision boundary as in Figure 4. Note that in contrast to HA-LFP, PageRank scores cannot be used to differentiating the good from the bad pages.

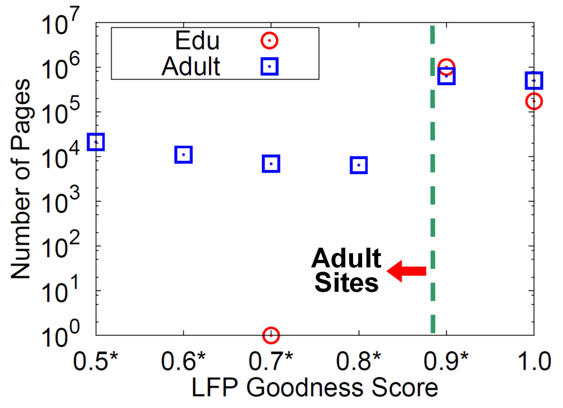


Fig. 4. HA-LFP scores and the number of pages in the test set having such scores. Note that pages whose goodness scores are less than 0.9(the left side of the vertical bar) are likely to be adult pages with very high chances.

calls whom. We define the three roles: 'celebrity', 'spammer', and normal people. We define a celebrity as a person with high in-degree ($>=$1000), and not-too-large out-degree($< 10 \times indegree$). We define a spammer as a person with high out-degree ($>=$1000), but low in-degree ($< 0.1 \times outdegree$). For celebrities, we set (0.8, 0.1, 0.1) for (celebrity, spammer, normal) prior probabilities. For spammers, we set (0.1, 0.8, 0.1) for (celebrity, spammer, normal) prior probabilities. The edge potential function is given by Table IV. It encodes our observation that celebrities tend to follow normal persons the most, spammers follow other spammers or normal persons, and normal persons follow other normal persons or celebrities.

	Celebrity	**Spammer**	**Normal**
Celebrity	0.1	0.05	0.85
Spammer	0.1	0.45	0.45
Normal	0.35	0.05	0.6

TABLE IV
EDGE POTENTIAL FOR THE TWITTER AND VOICECALL.

Figure 6 shows the HA-LFP scores of people in the Twitter and VoiceCall data. There are two clusters in both of the data. The large cluster starting from the 'Normal' vertex contains high degree nodes, and the small cluster below the large cluster contains low degree nodes.

C. *Finding Roles And Anomalies*

In the experiments of previous sections, we used several classes('bad' web sites, 'spammers', 'celebrities', etc.) of nodes. The question is, how can we find classes of a given graph? Finding out such classes is important for BP since it helps to set reasonable priors which could lead to quick convergence. In this section, we analyze real world graphs using the PEGASUS package [17] and give observations on the patterns and anomalies, which could potentially help determine the classes. We focus on the structural properties of graphs, including degree, connected component, and radius.

Using Degree Distributions. We first show the degree distributions of real world graphs in Figure 7. Notice that there are nodes with very high in or out degrees, which gives valuable information for setting priors.

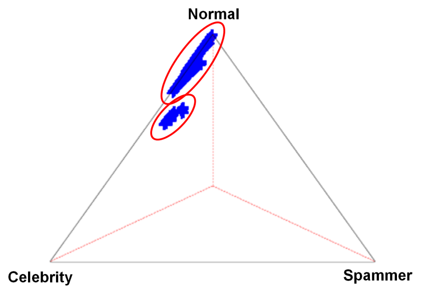
Observation 1 (High In or Out Degree Nodes): The nodes with high in-degree can have a high prior for 'celebrity', and the nodes with high out-degree but low in-degree can have a high prior for 'spammer'.

Most of the degree distributions in Figure 7 follow power law or log-normal. The VoiceCall in degree distribution(Figure 7 (e)) is different from other distributions since it contains mixture of distributions:

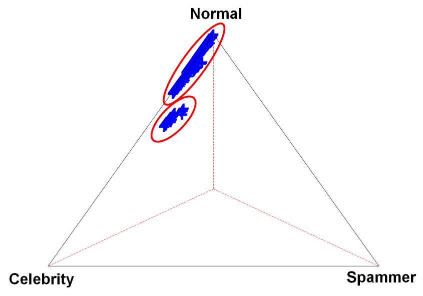
Observation 2 (Mixture of Lognormals in Degree Distribution): VoiceCall in degree distributions in Figure 7 seems to comprise two lognormal distributions shown in D1(red color) and D2(green color).

Another observation is that there are several anomalous spikes in the degree distributions in Figure 7 (b) and (d).

Observation 3 (Spikes in Degree Distribution): There is a huge spike at the out degree 1200 of YahooWeb data in Figure 7 (b). They came from online market pages from Germany, where the pages are linked to each other and forming link farms. Two outstanding spikes are also observed at the out degree 20 and 2001 of Twitter data in Figure 7 (d). The reason seems to be a hard limit in the maximum number of people to follow.



(a) Twitter

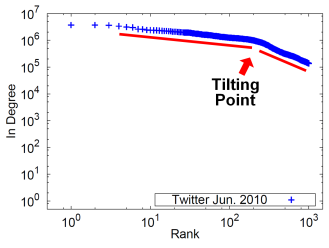


(b) VoiceCall

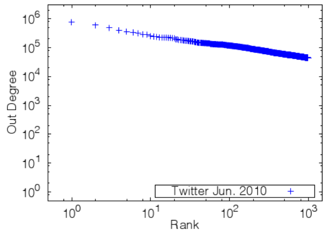
Fig. 6. Ha-Lfp scores of people in the Twitter and VoiceCall data. The points represent the scores of the final beliefs in each state, forming simplex in 3-dimensional space whose axes are the red lines that meet at the center(origin). Notice that people seem to form two groups, in *both* datasets, despite the fact that the two datasets are completely of different nature.

Finally, we study the highest degrees that are beyond the power-law or lognormal cutoff points using rank plot. Figure 8 shows the top 1000 highest in and out degrees and its rank(from 1 to 1000) which we summarize in the following observation.

Observation 4 (Tilt in Rank Plot): The out degree rank plot of Twitter data in Figure 8 (b) follows a power law with a single exponent. The in degree rank plot, however, comprises two fitting lines with a tilting point around rank 240. The tilting point divides the celebrities in two groups: super-celebrities (e.g., possibly, of international caliber) and plain celebrities (possibly, of national or regional caliber).



(a) In degree vs. Rank



(b) Out degree vs. Rank

Fig. 8. Degree vs. Rank. in Twitter Jun. 2010 data. Notice the change of slope around the tilting point in (a). The point can be used to distinguishing super-celebrities (e.g., of international caliber) versus plain celebrities (of national or regional caliber).

Using Connected Component Distributions. The distributions of the sizes of connected components in a graph informs us of the connectivity of the nodes (component size vs. number of components having that size). When these distributions are plotted over time, we may observe when certain nodes participate in various activities — patterns such as periodicity or anomalous deviations from such patterns can generate important insights.

Observation 5 (Periodic Dips and Surges): Figure 9 shows the temporal connected component distribution of the VoiceCall (who-calls-whom) data, where each data point was computed using one day's worth of data (i.e., a one-day snapshot). On every Sunday, we see a dip in the size of the giant connected component (largest component), and an accompanying surge in the number of connected components for the day. This periodicity highlights the typical and rather constant call volume during the work days, and lower volume outside them. Equipped with this information, we may infer that "business" phone numbers (nodes) are those that are regularly active during work days but not weekends; we may in turn characterize these "business" numbers as one class of nodes in our algorithm. The sizes of the second and third largest component oscillate about some small numbers (68 and 50 respectively), echoing previous research findings [24].

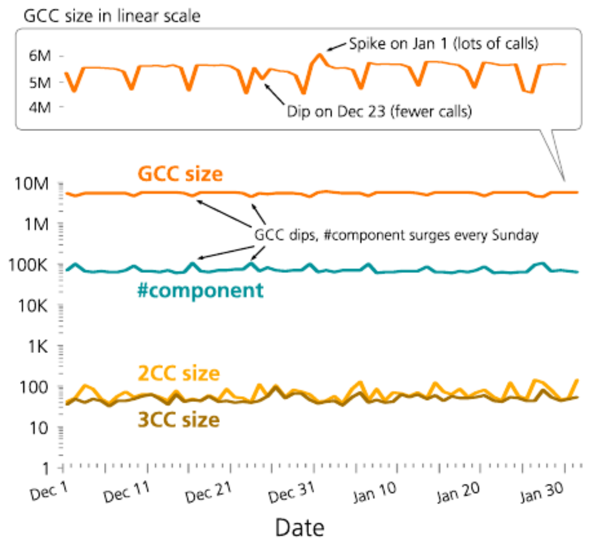
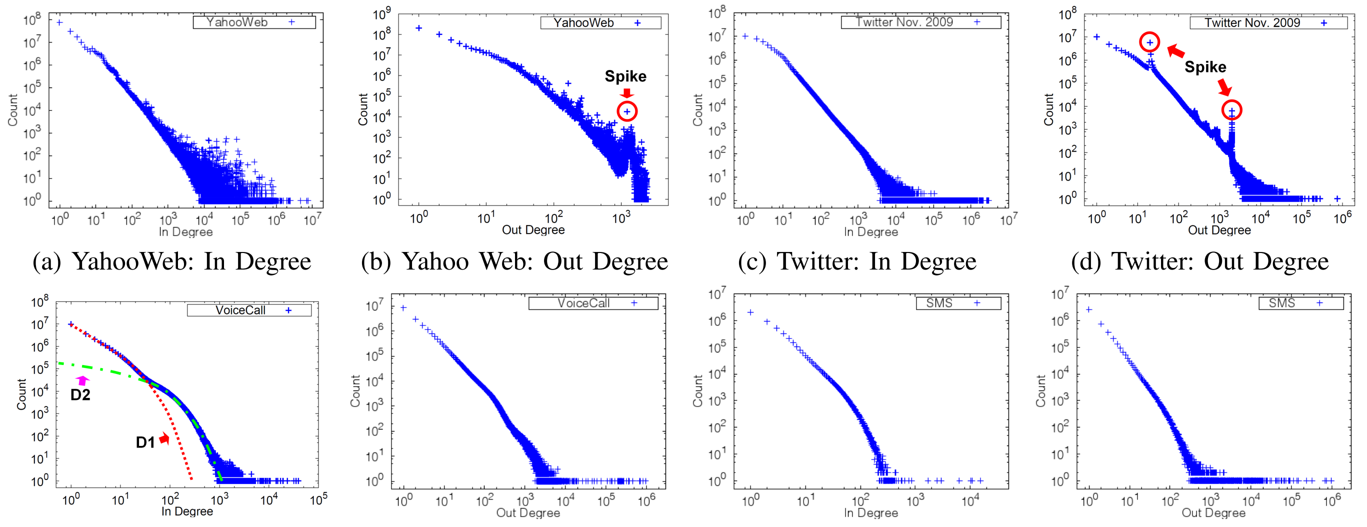


Fig. 9. **[Best Viewed In Color]** Temporal connected component distributions of the VoiceCall data, from Dec 1, 2007 to Jan 31, 2008, inclusively. Each data point computed using one day's worth of data (i.e., a one-day snapshot.) GCC, 2CC, and 3CC are the first (giant), second, and third largest components respectively. The turquoise line denotes the number of connected components. The temporal trend may be used to set priors for Ha-Lfp. See the text for details.

Using Radius Distributions. We next analyze the radius distributions of real graphs. Radius of a node is defined to be the 90%th percentile of all the distances to other nodes from it. Thus, nodes with low radii can reach other nodes in a small number of steps. Figure 10 shows the radius distributions of



(a) YahooWeb: In Degree (b) Yahoo Web: Out Degree (c) Twitter: In Degree (d) Twitter: Out Degree

(e) VoiceCall: In Degree (f) VoiceCall: Out Degree (g) SMS: In Degree (h) SMS: Out Degree

Fig. 7. **[(e): Best Viewed In Color]** Degree distributions of real world graphs. Notice many high in-degree or out-degree nodes which can be used to determine the classes for HA-LFP. Most distributions follow power-law or lognormal, except (e) which seems to be a mixture of two lognormal distributions. Also notice the several spikes which suggest anomalous nodes, suspicious activities, or software limits on the number of connections.

real graphs. In contrast to the VoiceCall and the SMS data, the Twitter data contains several anomalous nodes with long($>$10) radii.

Observation 6 (Suspicious Accounts Created By A User): The Twitter data contain several nodes with long radii. They form chains shown in Figure 11. Each chain seems to be created by one user, since the times in which accounts are created are regular.

VII. CONCLUSION

In this paper we proposed HADOOP LINE GRAPH FIXED POINT (HA-LFP), a HADOOP algorithm for the inferences of graphical models in billion-scale graphs. The main contributions are the followings:

- *Efficiency:* We show that the solution of inference problem in graphical models is a fixed point in line graph. We propose LINE GRAPH FIXED POINT (LFP), a formulation of BP on a line graph induced from the original graph, and show that it is a generalized version of a linear algebra operation. We propose HADOOP LINE GRAPH FIXED POINT (HA-LFP), an efficient algorithm carefully designed for LFP in HADOOP.
- *Scalability:* We do the experiments to compare the running time of the HA-LFP and a single-machine BP. We also gives the scalability results and show that HA-LFP has a near-linear scale up.
- *Effectiveness:* We show that our method can find interesting patterns and anomalies, on some of the largest publicly available graphs (Yahoo Web graph of 0.24 Tb, and twitter, of 0.13 Tb).

Future research directions include algorithms for mining based on graphical models, and tensor analysis on HADOOP ([25]).

ACKNOWLEDGMENT

The authors would like to thank YAHOO! for providing us with the web graph and access to the M45, and Brendan Meeder in CMU for providing Twitter data.

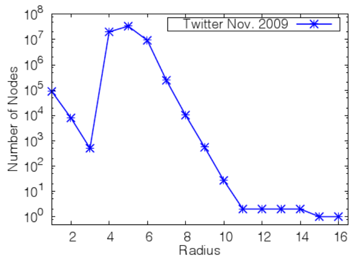
This material is based upon work supported by the National Science Foundation under Grants No. IIS-0705359, IIS0808661, and under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under contract No. DE-AC52-07NA27344.

Research was sponsored by the Army Research Laboratory and was accomplished under Cooperative Agreement Number W911NF-09-2-0053. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the Army Research Laboratory or the U.S. Government. The U.S. Government is authorized to reproduce and distribute reprints for Government purposes notwithstanding any copyright notation here on.

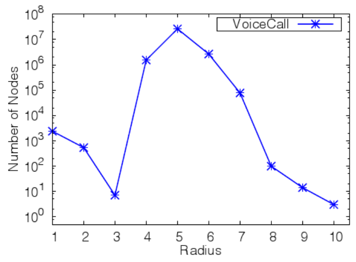
This work is also partially supported by an IBM Faculty Award. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation, or other funding parties.

REFERENCES

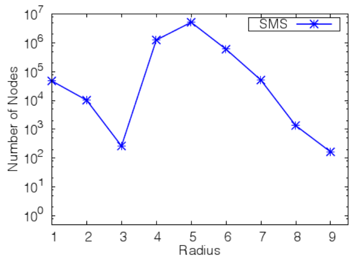
[1] J. Pearl, “Reverend Bayes on inference engines: A distributed hierarchical approach,” in *Proceedings of the AAAI National Conference on AI*, 1982, pp. 133–136.
[2] J. S. Yedidia, W. T. Freeman, and Y. Weiss, “Understanding belief propagation and its generalizations,” *Exploring Artificial Intelligence in the New Millenium*, 2003.
[3] P. Felzenszwalb and D. Huttenlocher, “Efficient belief propagation for early vision,” *International journal of computer vision*, vol. 70, no. 1, pp. 41–54, 2006.
[4] D. H. Chau, S. Pandit, and C. Faloutsos, “Detecting fraudulent personalities in networks of online auctioneers,” *PKDD*, 2006.



(a) Twitter



(b) VoiceCall



(c) SMS

Fig. 10. Radius distributions of real world graphs. Notice the nodes with long radius in the Twitter data. They are usually suspicious nodes as described in Figure 11.

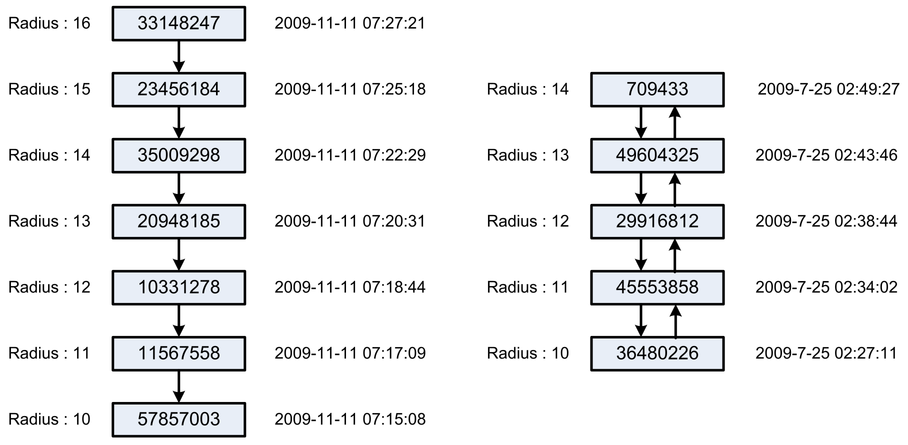


Fig. 11. Accounts with long radii in the Twitter Nov. 2009 data. Each box represents an account with the corresponding anonymized id. At the right of the boxes, the time that the account was created is shown. All the accounts are suspicious since they form chains with very low degree. They seem to be created from a user, based on the regular timestamps. Especially, all the 7 accounts in the left figure are from Mumbai, India.

[5] M. McGlohon, S. Bay, M. Anderle, D. Steier, and C. Faloutsos, “Snare: a link analytic system for graph labeling and risk detection,” in *Proceedings of the 15th ACM SIGKDD international conference on Knowledge discovery and data mining*. ACM, 2009, pp. 1265–1274.

[6] J. Pearl, *Probabilistic reasoning in intelligent systems: networks of plausible inference*. Morgan Kaufmann, 1988.

[7] Y. Weiss and W. Freeman, “Correctness of belief propagation in Gaussian graphical models of arbitrary topology,” *Neural Computation*, vol. 13, no. 10, pp. 2173–2200, 2001.

[8] J. E. Gonzalez, Y. Low, and C. Guestrin, “Residual splash for optimally parallelizing belief propagation,” *AISTAT*, 2009.

[9] J. Gonzalez, Y. Low, C. Guestrin, and D. O’Hallaron, “Distributed parallel inference on large factor graphs,” in *Conference on Uncertainty in Artificial Intelligence (UAI)*, Montreal, Canada, July 2009.

[10] A. Mendiburu, R. Santana, J. Lozano, and E. Bengoetxea, “A parallel framework for loopy belief propagation,” *GECCO*, 2007.

[11] J. Leskovec and C. Faloutsos, “Sampling from large graphs,” *KDD*, pp. 631–636, 2006.

[12] J. Dean and S. Ghemawat, “Mapreduce: Simplified data processing on large clusters,” *OSDI*, 2004.

[13] R. Lämmel, “Google’s mapreduce programming model – revisited,” *Science of Computer Programming*, vol. 70, pp. 1–30, 2008.

[14] “Hadoop information,” http://hadoop.apache.org/.

[15] C. Olston, B. Reed, U. Srivastava, R. Kumar, and A. Tomkins, “Pig latin: a not-so-foreign language for data processing,” in *SIGMOD ’08*, 2008, pp. 1099–1110.

[16] S. Papadimitriou and J. Sun, “Disco: Distributed co-clustering with map-reduce,” *ICDM*, 2008.

[17] U. Kang, C. Tsourakakis, and C. Faloutsos, “Pegasus: A peta-scale graph mining system - implementation and observations,” *IEEE International Conference on Data Mining*, 2009.

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

[19] U. Kang, M. McGlohon, L. Akoglu, and C. Faloutsos, “Patterns on the connected components of terabyte-scale graphs,” *IEEE International Conference on Data Mining*, 2010.

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

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

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

[23] J. Leskovec, D. Chakrabarti, J. M. Kleinberg, and C. Faloutsos, “Realistic, mathematically tractable graph generation and evolution, using kronecker multiplication,” *PKDD*, 2005.

[24] M. Mcglohon, L. Akoglu, and C. Faloutsos, “Weighted graphs and disconnected components: patterns and a generator,” *KDD*, pp. 524–532, 2008.

[25] T. G. Kolda and J. Sun, “Scalable tensor decompsitions for multi-aspect data mining,” *ICDM*, 2008.