Under consideration for publication in Knowledge and Information Systems

PEGASUS: Mining Peta-Scale Graphs

U Kang, Charalampos E. Tsourakakis, and Christos Faloutsos
School of Computer Science, Carnegie Mellon University, Pittsburgh PA, USA

Abstract. In this paper, we describe PEGASUS, an open source Peta Graph Mining library which performs typical graph mining tasks such as computing the diameter of the graph, computing the radius of each node and finding the connected components. As the size of graphs reaches several Giga-, Tera- or Peta-bytes, the necessity for such a library grows too. To the best of our knowledge, PEGASUS is the first such library, implemented on the top of the HADOOP platform, the open source version of MAPREDUCE.

Many graph mining operations (PageRank, spectral clustering, diameter estimation, connected components etc.) are essentially a repeated matrix-vector multiplication. In this paper we describe a very important primitive for PEGASUS, called `GIM-V` (Generalized Iterated Matrix-Vector multiplication). `GIM-V` is highly optimized, achieving (a) good scale-up on the number of available machines, (b) linear running time on the number of edges, and (c) more than *5 times* faster performance over the non-optimized version of `GIM-V`.

Our experiments ran on M45, one of the top 50 supercomputers in the world. We report our findings on several real graphs, including one of the largest publicly available Web Graphs, thanks to Yahoo!, with $\approx$ 6,7 billion edges.

Keywords: PEGASUS; graph mining; GIM-V; Generalized Iterative Matrix-Vector Multiplication; Hadoop

1. Introduction

Graphs are ubiquitous: computer networks, social networks, mobile call networks, the World Wide Web (Broder et al, 2000), protein regulation networks to name a few.

The large volume of available data, the low cost of storage and the stunning

Received xxx
Revised xxx
Accepted xxx

success of online social networks and web2.0 applications all lead to graphs of unprecedented size. Typical graph mining algorithms silently assume that the graph fits in the memory of a typical workstation, or at least on a single disk; the above graphs violate these assumptions, spanning multiple Giga-bytes, and heading to Tera- and Peta-bytes of data.

A promising tool is parallelism, and specifically MapReduce (Dean et al, 2004) and its open source version, Hadoop. Based on Hadoop, here we describe PeGaSus, a graph mining package for handling graphs with *billions* of nodes and edges. The PeGaSus code and several dataset are at **http://www.cs.cmu.edu/∼pegasus**. The contributions are the following:

1. Unification of seemingly different graph mining tasks, via a generalization of matrix-vector multiplication (`GIM-V`).
2. The careful implementation of `GIM-V`, with several optimizations, and several graph mining operations (PageRank, Random Walk with Restart(RWR), diameter estimation, and connected components). Moreover, the method is linear on the number of edges, and scales up well with the number of available machines.
3. Performance analysis, pinpointing the most successful combination of optimizations, which lead to up to *5 times* better speed than naive implementation.
4. Analysis of large, real graphs, including one of the largest publicly available graph that was ever analyzed, Yahoo's web graph.

The rest of the paper is organized as follows. Section 2 presents the related work. Section 3 describes our framework and explains several graph mining algorithms. Section 4 discusses optimizations that allow us to achieve significantly faster performance in practice. In Section 5 we present timing results and Section 6 our findings in real world, large scale graphs. We conclude in Section 7.

2. Background and Related Work

The related work forms two groups, graph mining, and Hadoop.

Large-Scale Graph Mining. There are a huge number of graph mining algorithms, computing communities (eg., (Chen et al, 2009), DENGRAPH (Falkowski et al, 2007), METIS (Karypis et al, 1999), (Narasimhamurthy et al, 2010)), subgraph discovery(e.g., GraphSig (Ranu et al, 2009), (Ke et al, 2009), (Hintsanen et al, 2008), (Cheng et al, 2008), gPrune (Zhu et al, 2007), gApprox (Chen et al, 2007), gSpan (Yan et al, 2002), Subdue (Ketkar et al, 2005), HSIGRAM / VSIGRAM (Kuramochi et al, 2004), ADI (Wang et al, 2004), CSV (Wang et al, 2008), (Lahiri et al, 2010)), finding important nodes (e.g., PageRank (Brin et al, 1998) and HITS (Kleinberg, 1998)), computing the number of triangles (Tsourakakis et al, KDD, 2009; Tsourakakis et al, Arxiv, 2009; Tsourakakis, 2010), computing the diameter (Kang et al, 2010), topic detection (Qian et al, 2009), attack detection (Shrivastava et al, 2008), clustering (Peng et al, 2010) (Long et al, 2010), with too-many-to-list alternatives for each of the above tasks. Most of the previous algorithms do not scale, at least directly, to several millions and billions of nodes and edges.

For connected components, there are several algorithms, using Breadth-First

Search, Depth-First-Search, "propagation" (Shiloach et al, 1982; Awerbuch et al, 1983; Hirschberg et al, 1979), or "contraction" (Greiner, 1994) . These works rely on a shared memory model which limits their ability to handle large, disk-resident graphs.

MapReduce and Hadoop. MapReduce is a programming framework (Dean et al, 2004) (Aggarwal et al, 2004) for processing huge amounts of unstructured data in a massively parallel way. MapReduce has two major advantages: (a) the programmer is oblivious of the details of the data distribution, replication, load balancing etc. and furthermore (b) the programming concept is familiar, i.e., the concept of functional programming. Briefly, the programmer needs to provide only two functions, a *map* and a *reduce*. The typical framework is as follows (Ralf, 2008): (a) the *map* stage sequentially passes over the input file and outputs (key, value) pairs; (b) the *shuffling* stage groups of all values by key, (c) the *reduce* stage processes the values with the same key and outputs the final result.

Hadoop is the open source implementation of MapReduce. Hadoop provides the Distributed File System (HDFS) and PIG, a high level language for data analysis (Olston et al, 2008). Due to its power, simplicity and the fact that building a small cluster is relatively cheap, Hadoop is a very promising tool for large scale graph mining applications, something already reflected in academia, see (Papadimitriou et al, 2008; Kang et al, 2009). In addition to PIG, there are several high-level language and environments for advanced MapReduce-like systems, including SCOPE (Chaiken et al, 2008), Sawzall (Pike et al, 2005), and Sphere (Grossman et al, 2008).

3. Proposed Method

How can we quickly find connected components, diameter, PageRank, node proximities of very large graphs? We show that, even if they seem unrelated, eventually we can unify them using the `GIM-V` primitive, standing for Generalized Iterative Matrix-Vector multiplication, which we describe in the next.

3.1. Main Idea

`GIM-V`, or 'Generalized Iterative Matrix-Vector multiplication' is a generalization of normal matrix-vector multiplication. Suppose we have a n by n matrix M and a vector v of size n. Let $m_{i,j}$ denote the (i, j)-th element of M. Then the usual matrix-vector multiplication is

$M \times v = v'$ where $v'_i = \sum_{j=1}^{n} m_{i,j} v_j$.

There are three operations in the previous formula, which, if customized separately, will give a surprising number of useful graph mining algorithms:

1. `combine2`: multiply $m_{i,j}$ and v_j.
2. `combineAll`: sum n multiplication results for node i.
3. `assign`: overwrite previous value of v_i with new result to make v'_i.

In `GIM-V`, let's define the operator $\times_G$, where the three operations can be defined arbitrarily. Formally, we have:

```
SELECT E.sid, combineAll_{E.sid}(combine2(E.val,V.val))
  FROM E, V
  WHERE E.did=V.id
  GROUP BY E.sid
```

$v' = M \times_G v$
where $v'_i = \texttt{assign}(v_i, \texttt{combineAll}_i(\{x_j \mid j = 1..n, \text{and } x_j = \texttt{combine2}(m_{i,j}, v_j)\}))$.

The functions `combine2()`, `combineAll()`, and `assign()` have the following signatures (generalizing the product, sum and assignment, respectively, that the traditional matrix-vector multiplication requires):

1. $\texttt{combine2}(m_{i,j}, v_j)$: combine $m_{i,j}$ and v_j.
2. $\texttt{combineAll}_i(x_1, ..., x_n)$: combine all the results from `combine2()` for node i.
3. $\texttt{assign}(v_i, v_{new})$: decide how to update v_i with v_{new}.

The 'Iterative' in the name of `GIM-V` denotes that we apply the $\times_G$ operation until an algorithm-specific convergence criterion is met. As we will see in a moment, by customizing these operations, we can obtain different, useful algorithms including PageRank, Random Walk with Restart, connected components, and diameter estimation. But first we want to highlight the strong connection of `GIM-V` with SQL: When $\texttt{combineAll}_i()$ and `assign()` can be implemented by user defined functions, the operator $\times_G$ can be expressed concisely in terms of SQL. This viewpoint is important when we implement `GIM-V` in large scale parallel processing platforms, including HADOOP, if they can be customized to support several SQL primitives including JOIN and GROUP BY. Suppose we have an `edge` table `E(sid, did, val)` and a `vector` table `V(id, val)`, corresponding to a matrix and a vector, respectively. Then, $\times_G$ corresponds to the following SQL statement - we assume that we have (built-in or user-defined) functions $\texttt{combineAll}_i()$ and `combine2()`) and we also assume that the resulting table/vector will be fed into the `assign()` function (omitted, for clarity):

In the following sections we show how we can customize `GIM-V`, to handle important graph mining operations including PageRank, Random Walk with Restart, diameter estimation, and connected components.

3.2. `GIM-V` and PageRank

Our first application of `GIM-V` is PageRank, a famous algorithm that was used by Google to calculate relative importance of web pages (Brin et al, 1998). The PageRank vector p of n web pages satisfies the following eigenvector equation:

$$p = (cE^T + (1-c)U)p$$

where c is a damping factor (usually set to 0.85), E is the row-normalized adjacency matrix (source, destination), and U is a matrix with all elements set to $1/n$.

To calculate the eigenvector p we can use the power method, which multiplies an initial vector with the matrix, several times. We initialize the current PageRank vector p^{cur} and set all its elements to $1/n$. Then the next PageRank p^{next} is calculated by $p^{next} = (cE^T + (1-c)U)p^{cur}$. We continue to do the multiplication until p converges.

PageRank is a direct application of GIM-V. In this view, we first construct a matrix M by column-normalize E^T such that every column of M sum to 1. Then the next PageRank is calculated by $p^{next} = M \times_G p^{cur}$ where the three operations are defined as follows:

1. $\texttt{combine2}(m_{i,j}, v_j) = c \times m_{i,j} \times v_j$
2. $\texttt{combineAll}_i(x_1, ..., x_n) = \frac{(1-c)}{n} + \sum_{j=1}^{n} x_j$
3. $\texttt{assign}(v_i, v_{new}) = v_{new}$

3.3. GIM-V and Random Walk with Restart

Random Walk with Restart(RWR) is an algorithm to measure the proximity of nodes in graph (Pan et al, 2004). In RWR, the proximity vector r_k from node k satisfies the equation:

$r_k = cMr_k + (1-c)e_k$

where e_k is a n-vector whose k^{th} element is 1, and every other elements are 0. c is a restart probability parameter which is typically set to 0.85 (Pan et al, 2004). M is a column-normalized and transposed adjacency matrix, as in Section 3.2. In GIM-V, RWR is formulated by $r_k^{next} = M \times_G r_k^{cur}$ where the three operations are defined as follows (δ_{ik} is the *Kronecker delta*, equal to 1 if $i = k$ and 0 otherwise):

1. $\texttt{combine2}(m_{i,j}, v_j) = c \times m_{i,j} \times v_j$
2. $\texttt{combineAll}_i(x_1, ..., x_n) = (1-c)\delta_{ik} + \sum_{j=1}^{n} x_j$
3. $\texttt{assign}(v_i, v_{new}) = v_{new}$

3.4. GIM-V and Diameter Estimation

HADI (Kang et al, 2010) is an algorithm to estimate the diameter and radius of large graphs. The diameter of a graph is the maximum of the length of the shortest path between every pair of nodes. The radius of a node v_i is the number of hops that we need to reach the farthest-away node from v_i. The main idea of HADI is as follows. For each node v_i in the graph, we maintain the number of neighbors reachable from v_i within h hops. As h increases, the number of neighbors increases until h reaches it maximum value. The diameter is h where the number of neighbors within $h+1$ does not increase for every node. For further details and optimizations, see (Kang et al, 2010).

The main operation of HADI is updating the number of neighbors as h increases. Specifically, the number of neighbors within hop h reachable from node v_i is encoded in a probabilistic bitstring b_i^h which is updated as follows:

$b_i^{h+1} = b_i^h$ BITWISE-OR $\{b_k^h \mid (i,k) \in E\}$

In GIM-V, the bitstring update of HADI is represented by

$b^{h+1} = M \times_G b^h$

where M is an adjacency matrix, b^{h+1} is a vector of length n which is updated by

b_i^{h+1} =assign(b_i^h,combineAll$_i$($\{x_j \mid j = 1..n$, and x_j =combine2$(m_{i,j}, b_j^h)\}$)),
and the three operations are defined as follows:

1. combine2$(m_{i,j}, v_j) = m_{i,j} \times v_j$.
2. combineAll$_i(x_1, ..., x_n)$ = BITWISE-OR$\{x_j \mid j = 1..n\}$
3. assign(v_i, v_{new}) = BITWISE-OR(v_i, v_{new}).

The $\times_G$ operation is run iteratively until the bitstring for all the nodes do not change.

3.5. GIM-V and Connected Components

We propose HCC, a new algorithm for finding connected components in large graphs. Like HADI, HCC is an application of GIM-V with custom functions. The main idea is as follows. For every node v_i in the graph, we maintain a component id c_i^h which is the minimum node id within h hops from v_i. Initially, c_i^h of v_i is set to its own node id: that is, $c_i^0 = i$. For each iteration, each node sends its current c_i^h to its neighbors. Then c_i^{h+1}, component id of v_i at the next step, is set to the minimum value among its current component id and the received component ids from its neighbors. The crucial observation is that this communication between neighbors can be formulated in GIM-V as follows:

$c^{h+1} = M \times_G c^h$

where M is an adjacency matrix, c^{h+1} is a vector of length n which is updated by
c_i^{h+1} =assign(c_i^h,combineAll$_i$($\{x_j \mid j = 1..n$, and x_j =combine2$(m_{i,j}, c_j^h)\}$)),
and the three operations are defined as follows:

1. combine2$(m_{i,j}, v_j) = m_{i,j} \times v_j$.
2. combineAll$_i(x_1, ..., x_n)$ = MIN$\{x_j \mid j = 1..n\}$.
3. assign(v_i, v_{new}) = MIN(v_i, v_{new}).

By repeating this process, component ids of nodes in a component are set to the minimum node id of the component. We iteratively do the multiplication until component ids converge. The upper bound of the number of iterations in HCC are determined by the following theorem.

Theorem 1 (Upper bound of iterations in Hcc). HCC requires maximum d iterations where d is the diameter of the graph.

Proof. The minimum node id is propagated to its neighbors at most d times. □

Since the diameter of real graphs are relatively small, HCC completes after small number of iterations.

4. Fast Algorithms for GIM-V

How can we parallelize the algorithm presented in the previous section? In this section, we first describe naive HADOOP algorithms for GIM-V. After that we propose several faster methods for GIM-V.

Algorithm 1 `GIM-V` BASE Stage 1.

Input: Matrix $M = \{(id_{src}, (id_{dst}, mval))\}$, Vector $V = \{(id, vval)\}$
Output: Partial vector $V' = \{(id_{src}, \texttt{combine2}(mval, vval)\}$

```
1: Stage1-Map(Key k, Value v):
2: if (k, v) is of type V then
3:    Output(k, v);                                   // (k: id, v: vval)
4: else if (k, v) is of type M then
5:    (id_dst, mval) ← v;
6:    Output(id_dst, (k, mval));                      // (k: id_src)
7: end if
8:
9: Stage1-Reduce(Key k, Value v[1..m]):
10: saved_kv ←[ ];
11: saved_v ←[ ];
12: for v ∈ v[1..m] do
13:    if (k, v) is of type V then
14:       saved_v ← v;
15:       Output(k, ("self", saved_v));
16:    else if (k, v) is of type M then
17:       Add v to saved_kv;                           // (v: (id_src, mval))
18:    end if
19: end for
20: for (id'_src, mval') ∈ saved_kv do
21:    Output(id'_src, ("others",combine2(mval', saved_v)));
22: end for
```

4.1. `GIM-V` BASE: Naive Multiplication

`GIM-V` BASE is a two-stage algorithm whose pseudo code is in Algorithm 1 and 2. The inputs are an edge file and a vector file. Each line of the edge file contains one $(id_{src}, id_{dst}, mval)$ which corresponds to a non-zero cell in the adjacency matrix M. Similarly, each line of the vector file contains one $(id, vval)$ which corresponds to an element in the vector V. `Stage1` performs `combine2` operation by combining columns of matrix(id_{dst} of M) with rows of vector(id of V). The output of `Stage1` are (key, value) pairs where key is the source node id of the matrix(id_{src} of M) and the value is the partially combined result(`combine2`$(mval, vval)$). This output of `Stage1` becomes the input of `Stage2`. `Stage2` combines all partial results from `Stage1` and assigns the new vector to the old vector. The `combineAll`$_i()$ and `assign`() operations are done in line 15 of `Stage2`, where the "self" and "others" tags in line 15 and line 21 of `Stage1` are used to make v_i and v_{new} of `GIM-V`, respectively.

This two-stage algorithm is run iteratively until application-specific convergence criterion is met. In Algorithm 1 and 2, Output(k, v) means to output data with the key k and the value v.

4.2. `GIM-V` BL: Block Multiplication

`GIM-V` BL is a fast algorithm for `GIM-V` which is based on block multiplication. The main idea is to group elements of the input matrix into blocks or submatrices

Algorithm 2 `GIM-V` BASE Stage 2.

Input: Partial vector $V' = \{(id_{src}, vval')\}$
Output: Result Vector $V = \{(id_{src}, vval)\}$

```
1: Stage2-Map(Key k, Value v):
2: Output(k, v);
3:
4: Stage2-Reduce(Key k, Value v[1..m]):
5: others_v ←[ ];
6: self_v ←[ ];
7: for v ∈ v[1..m] do
8:    (tag, v') ← v;
9:    if tag = "same" then
10:      self_v ← v';
11:   else if tag = "others" then
12:      Add v' to others_v;
13:   end if
14: end for
15: Output(k,assign(self_v,combineAll_k(others_v)));
```

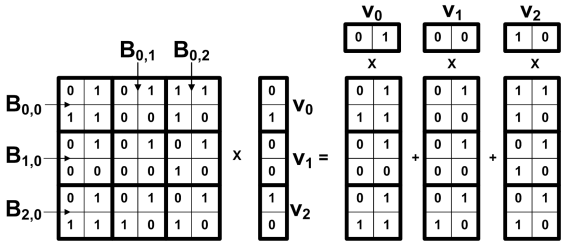


Fig. 1. `GIM-V` BL using 2 x 2 blocks. $B_{i,j}$ represents a matrix block, and v_i represents a vector block. The matrix and vector are joined block-wise, not element-wise.

of size b by b. Also we group elements of input vectors into blocks of length b. Here the grouping means we put all the elements in a group into one line of input file. Each block contains only non-zero elements of the matrix or vector. The format of a matrix block with k nonzero elements is $(row_{block}, col_{block}, row_{elem_1}, col_{elem_1}, mval_{elem_1}, ..., row_{elem_k}, col_{elem_k}, mval_{elem_k})$. Similarly, the format of a vector block with k nonzero elements is $(id_{block}, id_{elem_1}, vval_{elem_1}, ..., id_{elem_k}, vval_{elem_k})$. Only blocks with at least one nonzero elements are saved to disk. This block encoding forces nearby edges in the adjacency matrix to be closely located; it is different from HADOOP's default behavior which do not guarantee co-locating them. After grouping, `GIM-V` is performed on blocks, not on individual elements. `GIM-V` BL is illustrated in Figure 1.

In our experiment at Section 5, `GIM-V` BL is more than 5 times faster than `GIM-V` BASE. There are two main reasons for this speed-up.

– **Sorting Time** Block encoding decrease the number of items to sort in the shuffling stage of HADOOP. We observed that one of the main bottleneck of programs in HADOOP is its shuffling stage where network transfer, sorting, and disk I/O happens. By encoding to blocks of width b, the number of lines in

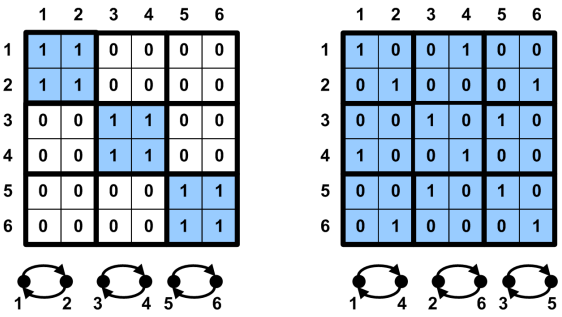


Fig. 2. Clustered vs. non-clustered adjacency matrices for two isomorphic graphs. The edges are grouped into 2 by 2 blocks. The left graph uses only 3 blocks while the right graph uses 9 blocks.

the matrix and the vector file decreases to $1/b^2$ and $1/b$ times of their original size, respectively for full matrices and vectors.

– **Compression** The size of the data decreases significantly by converting edges and vectors to block format. The reason is that in `GIM-V` BASE we need 4×2 bytes to save each (srcid, dstid) pair since we need 4 bytes to save a node id using Integer. However in `GIM-V` BL we can specify each *block* using a block row id and a block column id with two 4-byte Integers, and refer to elements inside the block using $2 \times logb$ bits. This is possible because we can use log$b$ bits to refer to a row or column inside a block. By this block method we decreased the edge file size(e.g., more than 50% for YahooWeb graph in Section 5).

4.3. `GIM-V` CL: Clustered Edges

When we use block multiplication, another advantage is that we can benefit from clustered edges. As can be seen from Figure 2, we can use smaller number of blocks if input edge files are clustered. Clustered edges can be built if we can use heuristics in data preprocessing stage so that edges are clustered, or by co-clustering (e.g., see (Papadimitriou et al, 2008)). The preprocessing for edge clustering need to be done only once; however, they can be used by every iteration of various application of `GIM-V`. So we have two variants of `GIM-V`: `GIM-V` CL, which is `GIM-V` BASE with clustered edges, and `GIM-V` BL-CL, which is `GIM-V` BL with clustered edges. Be aware that clustered edges is only useful when combined with block encoding. If every element is treated separately, then clustered edges don't help anything for the performance of `GIM-V`.

4.4. `GIM-V` DI: Diagonal Block Iteration

As mentioned in Section 4.2, the main bottleneck of `GIM-V` is its shuffling and disk I/O steps. Since `GIM-V` iteratively runs Algorithm 1 and 2, and each Stage requires disk IO and shuffling, we could decrease running time if we decrease the number of iterations.

In Hcc, it is possible to decrease the number of iterations. The main idea is to multiply diagonal matrix blocks and corresponding vector blocks as much as possible in one iteration. Remember that multiplying a matrix and a vector

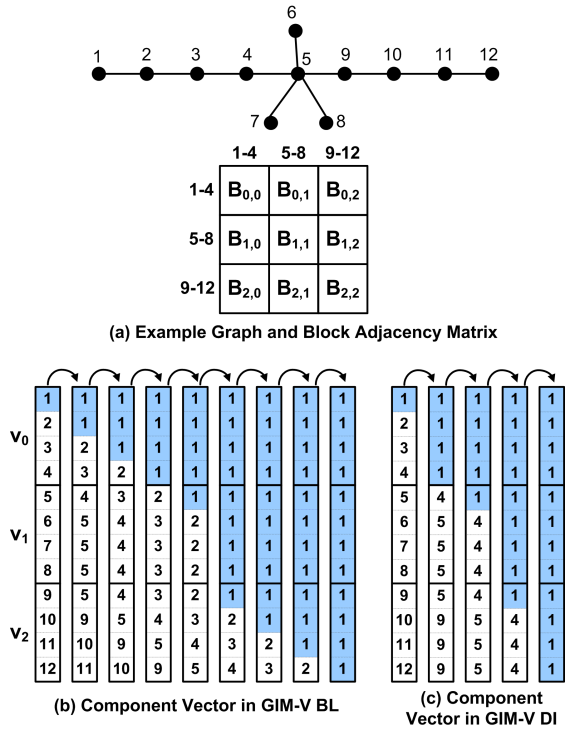


Fig. 3. Propagation of component id(=1) when block width is 4. Each element in the adjacency matrix of (a) represents a 4 by 4 block; each column in (b) and (c) represents the vector after each iteration. `GIM-V` DL finishes in 4 iterations while `GIM-V` BL requires 8 iterations.

corresponds to passing node ids to one step neighbors in HCC. By multiplying diagonal blocks and vectors until the contents of the vectors do not change in one iteration, we can pass node ids to neighbors located more than one step away. This is illustrated in Figure 3.

We see that in Figure 3 (c) we multiply $B_{i,i}$ with v_i several times until v_i do not change in one iteration. For example in the first iteration v_0 changed from {1,2,3,4} to {1,1,1,1} since it is multiplied to $B_{0,0}$ four times. `GIM-V` DI is especially useful in graphs with long chains.

The upper bound of the number of iterations in HCC DI with chain graphs are determined by the following theorem.

Theorem 2 (Upper bound of iterations in Hcc DI). In a chain graph with length m, it takes maximum $2 * \lceil m/b \rceil - 1$ iterations in HCC DI with block size b.

Proof. The worst case happens when the minimum node id is in the beginning of the chain. It requires 2 iterations(one for propagating the minimum node id inside the block, another for passing it to the next block) for the minimum node id to move to an adjacent block. Since the farthest block is $\lceil m/b \rceil - 1$ steps away, we need $2 * (\lceil m/b \rceil - 1)$ iterations. When the minimum node id reached the farthest away block, `GIM-V` DI requires one more iteration to propagate the minimum

Algorithm 3 Renumbering the minimum node

Input: Edge $E = \{(id_{src}, id_{dst})\}$,
current minimum node id $minid_{cur}$,
new minimum node id $minid_{new}$
Output: Renumbered Edge $V = \{(id'_{src}, id'_{dst})\}$

```
1: Renumber–Map(key k, value v):
2: src ← k;
3: dst ← v;
4: if src = minid_cur then
5:    src ← minid_new;
6: else if src = minid_new then
7:    src ← minid_cur;
8: end if
9: if dst = minid_cur then
10:   dst ← minid_new;
11: else if dst = minid_new then
12:   dst ← minid_cur;
13: end if
14: Output(src, dst);
```

node id inside the last block. Therefore, we need $2*(\lceil m/b \rceil - 1)+1 = 2*\lceil m/b \rceil - 1$ iterations. □

4.5. GIM-V NR: Node Renumbering

In HCC, the minimum node id is propagated to the other parts of the network within at most d steps, where d is the diameter of the network. If the node with the minimum id(which we call 'minimum node') is located at the center of the network, then the number of iterations is small, close to $d/2$. However, if it is located at the boundary of the network, then the number of iteration can be close to d. Therefore, if we preprocess the edges so that the minimum node id is swapped to the center node id, the number of iterations and the total running time of HCC would decrease.

Finding the center node with the minimum radius could be done with the HADI (Kang et al, 2010) algorithm. However, the algorithm is expensive for the pre-processing step of HCC. Therefore, we instead propose the following heuristic for finding the center node: we choose the center node by sampling from the highest-degree nodes. This heuristic is based on the fact that nodes with large degree have small radii (Kang et al, 2010). Moreover, computing the degree of very large graphs is trivial in MapReduce and could be performed quickly with one job.

After finding a center node, we need to renumber the edge file to swap the current minimum node id with the center node id. The MapReduce algorithm for this renumbering is shown in Algorithm 3. Since the renumbering requires only filtering, it can be done with a Map-only job.

4.6. Analysis

We analyze the time and space complexity of GIM-V. In the theorems below, M is the number of machines.

Theorem 3 (Time Complexity of GIM-V). One iteration of GIM-V takes $O(\frac{V+E}{M} log \frac{V+E}{M})$ time.

Proof. Assuming uniformity, mappers and reducers of Stage1 and Stage2 receives $O(\frac{V+E}{M})$ records per machine. The running time is dominated by the sorting time for $\frac{V+E}{M}$ records, which is $O(\frac{V+E}{M} log \frac{V+E}{M})$. □

Theorem 4 (Space Complexity of GIM-V). GIM-V requires $O(V + E)$ space.

Proof. We assume the value of the elements of the input vector v is constant. Then the theorem is proved by noticing that the maximum storage is required at the output of Stage1 mappers which requires $O(V + E)$ space up to a constant. □

5. Performance and Scalability

We do experiments to answer following questions:

Q1 How does GIM-V scale up?
Q2 Which of the proposed optimizations(block multiplication, clustered edges, and diagonal block iteration, node renumbering) gives the highest performance gains?

The graphs we used in our experiments at Section 5 and 6 are described in Table 1 [1] .

We run PeGaSus in M45 Hadoop cluster by Yahoo! and our own cluster composed of 9 machines. M45 is one of the top 50 supercomputers in the world with 1.5 Pb total storage and 3.5 Tb memory. For the performance and scalability experiments, we used synthetic Kronecker graphs (Leskovec et al, 2005) since we can generate them with any size, and they are one of the most realistic graphs among synthetic graphs.

5.1. Results

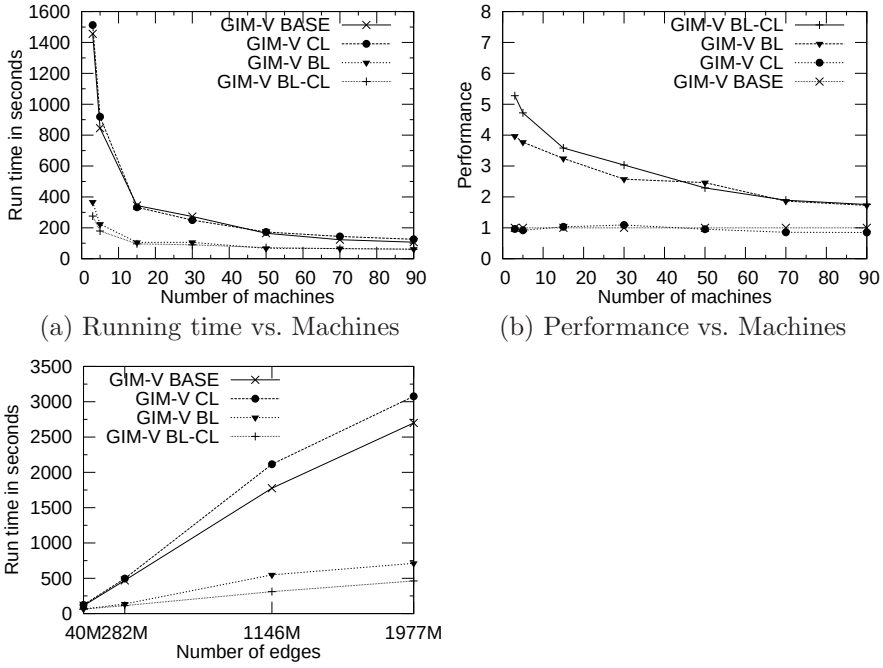
We first show how the performance of our method changes as we add more machines. Figure 4 shows the running time and performance of GIM-V for PageRank with Kronecker graph of 282 million edges, and size 32 blocks if necessary.

In Figure 4 (a), for all of the methods the running time decreases as we add more machines. Note that clustered edges(GIM-V CL) didn't help performance unless it is combined with block encoding. When it is combined, however, it showed the best performance (GIM-V BL-CL).

[1] Wikipedia: http://www.cise.ufl.edu/research/sparse/matrices/
Kronecker, DBLP: http://www.cs.cmu.edu/∼pegasus
YahooWeb, LinkedIn: released under NDA.
flickr, Epinions, patent: not public data.

Name	Nodes	Edges	Description
YahooWeb	1,413 M	6,636 M	WWW pages in 2002
LinkedIn	7.5 M	58 M	person-person in 2006
	4.4 M	27 M	person-person in 2005
	1.6 M	6.8 M	person-person in 2004
	85 K	230 K	person-person in 2003
Wikipedia	3.5 M	42 M	doc-doc in 2007/02
	3 M	35 M	doc-doc in 2006/09
	1.6 M	18.5 M	doc-doc in 2005/11
Kronecker	177 K	1,977 M	synthetic
	120 K	1,145 M	synthetic
	59 K	282 M	synthetic
	19 K	40 M	synthetic
WWW-Barabasi	325 K	1,497 K	WWW pages in nd.edu
DBLP	471 K	112 K	document-document
flickr	404 K	2.1 M	person-person
Epinions	75 K	508 K	who trusts whom

Table 1. Order and size of networks.



(a) Running time vs. Machines

(b) Performance vs. Machines

(c) Running time vs. Edges

Fig. 4. Scalability and Performance of GIM-V. (a) Running time decreases quickly as more machines are added. (b) The performance(=1/*running time*) of 'BL-CL' wins more than 5x (for n=3 machines) over the 'BASE'. (c) Every version of `GIM-V` shows linear scalability.

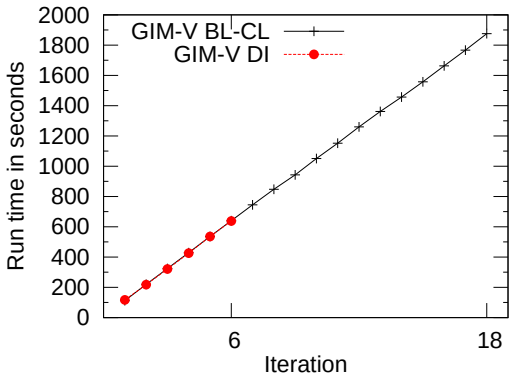


Fig. 5. Comparison of GIM-V DI and GIM-V BL-CL for HCC. GIM-V DI finishes in 6 iterations while GIM-V BL-CL finishes in 18 iterations due to long chains.

In Figure 4 (b), we see that the relative performance of each method compared to GIM-V BASE method decreases as number of machines increases. With 3 machines (minimum number of machines which HADOOP 'distributed mode' supports), the fastest method(GIM-V BL-CL) ran 5.27 times faster than GIM-V BASE. With 90 machines, GIM-V BL-CL ran 2.93 times faster than GIM-V BASE. This is expected since there are fixed component(JVM load time, disk I/O, network communication) which can not be optimized even if we add more machines.

Next we show how the performance of our methods changes as the input size grows. Figure 4 (c) shows the running time of GIM-V with different number of edges under 10 machines. As we can see, all of the methods scales linearly with the number of edges.

Next, we compare the performance of GIM-V DI and GIM-V BL-CL for HCC in graphs with long chains. For this experiment we made a new graph whose diameter is 17, by adding a length 15 chain to the 282 million Kronecker graph which has diameter 2. As we see in Figure 5, GIM-V DI finished in 6 iteration while GIM-V BL-CL finished in 18 iteration. The running time of both methods for the first 6 iterations are nearly same. Therefore, the diagonal block iteration method decrease the number of iterations while not affecting the running time of each iteration much.

Finally, we compare the number of iterations with/without renumbering. Figure 6 shows the degree distribution of LinkedIn. Without renumbering, the minimum node has degree 1, which is not surprising since about 46 % of the nodes have degree 1 due to the power-law behavior of the degree distribution. We show the number of iterations after changing the minimum node to each of the top 5 highest-degree nodes in Figure 7. We see that the renumbering decreased the number of iterations to 81 % of the original. Similar results are observed for the Wikipedia graph in Figure 8 and 9. The original minimum node has degree 1, and the number of iterations decreased to 83 % of the original after renumbering.

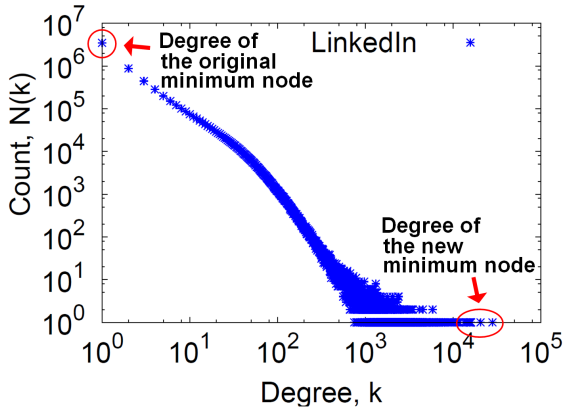


Fig. 6. Degree distribution of LinkedIn. Notice that the original minimum node has degree 1, which is highly probable given the power-law behavior of the degree distribution. After the renumbering, the minimum node is replaced with a highest-degree node.

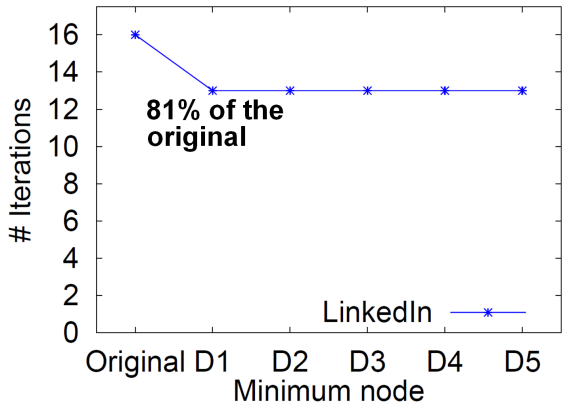


Fig. 7. Number of iterations vs. the minimum node of LinkedIn, for connected components. Di represents the node with i-th largest degree. Notice that the number of iterations decreased by 19 % after renumbering.

6. GIM-V At Work

In this section we use PEGASUS for mining very large graphs. We analyze connected components, diameter, and PageRank of large real world graphs. We show that PEGASUS can be useful for finding patterns, outliers, and interesting observations.

6.1. Connected Components of Real Networks

We used the LinkedIn social network and Wikipedia page-linking-to-page network, along with the YahooWeb graph for connected component analysis. Figure 10 show the evolution of connected components of LinkedIn and Wikipedia

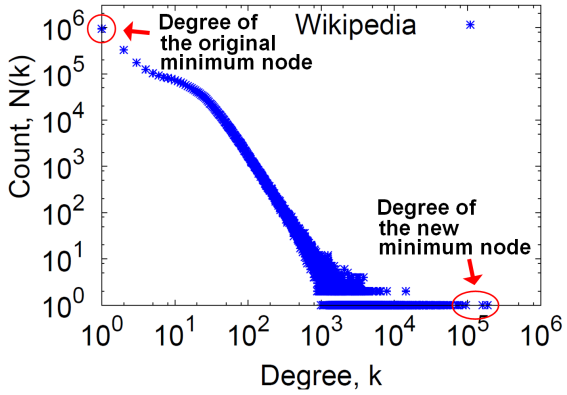


Fig. 8. Degree distribution of Wikipedia. Notice that the original minimum node has degree 1, as in LinkedIn. After the renumbering, the minimum node is replaced with a highest-degree node.

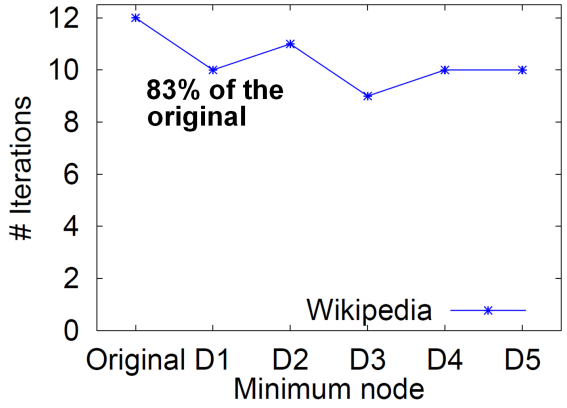


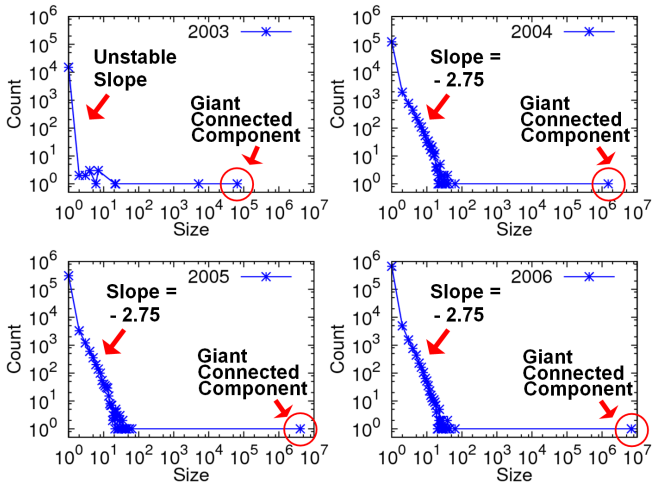
Fig. 9. Number of iterations vs. the minimum node of Wikipedia, for connected components. Di represents the node with i-th largest degree. Notice that the number of iterations decreased by 17 % after renumbering.

data. Figure 11 show the distribution of connected components in the YahooWeb graph. We have following observations.

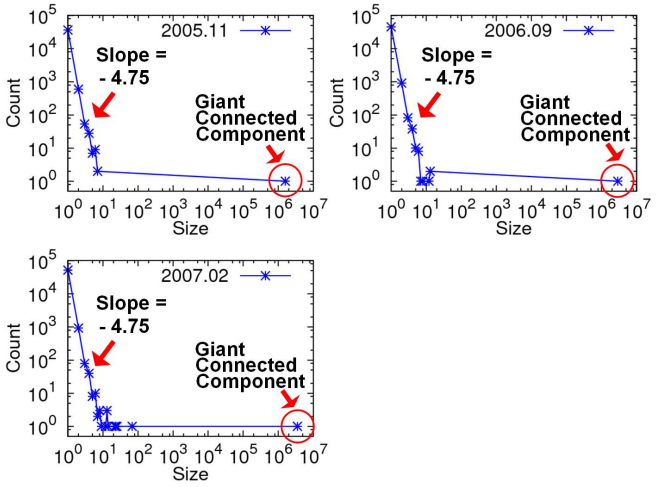
Power Law Tails in Connected Components Distributions We observed power law relation of count and size of small connected components in Figure 10(a),(b) and Figure 11. This reflects that the connected components in real networks are formed by processes similar to Chinese Restaurant Process and Yule distribution (Newman, 2005).

Stable Connected Components After Gelling Point In Figure 10(a), the distribution of connected components remain stable after a 'gelling' point(McGlohon et al, 2008) at year 2003.We can see that the slope of tail distribution do not change after year 2003. We observed the same phenomenon in Wikipedia graph in Figure 10 (b). The graph show stable tail slopes from the beginning, since the network were already mature in year 2005.

Absorbed Connected Components and Dunbar's number In Fig-



(a) Connected Components of LinkedIn



(b) Connected Components of Wikipedia

Fig. 10. The evolution of connected components. (a) The giant connected component grows for each year. However, the second largest connected component do not grow above Dunbar's number($\approx$ 150) and the slope of the tail remains constant after the gelling point at year 2003. (b) As in LinkedIn, notice the growth of giant connected component and the constant slope for tails.

ure 10(a), we find two large connected components in year 2003. However it became merged in year 2004. The giant connected component keeps growing, while the second and the third largest connected components do not grow beyond size 100 until they are absorbed to the giant connected component in Figure 10 (a) and (b). This agrees with the observation(McGlohon et al, 2008) that the size of the second/third connected components remains constant or oscillates. Lastly, the maximum connected component size except the giant connected component

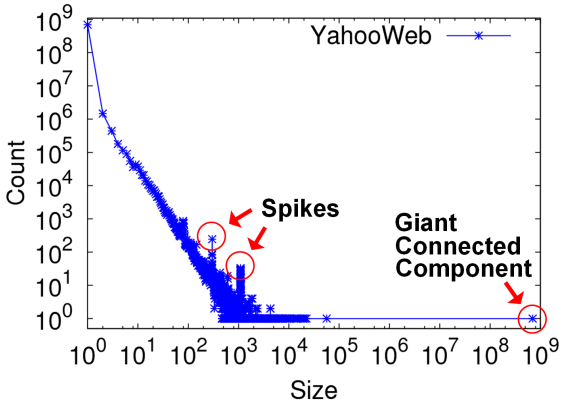


Fig. 11. Connected Components of YahooWeb. Notice the two anomalous spikes which are far from the constant-slope tail. They are domain selling or porn sites which are replicated from templates.

in the LinkedIn graph agrees well with Dunbar's number(Dunbar, 1998), which says that the maximum community size in social networks is roughly 150.

Anomalous Connected Components In Figure 11, we found two outstanding spikes. In the first spike at size 300, more than half of the components have exactly the same structure and they were made from a domain selling company where each component represents a domain to be sold. The spike happened because the company *replicated* sites using the same template, and injected the disconnected components into WWW network. In the second spike at size 1101, more than 80 % of the components are porn sites disconnected from the giant connected component. By looking at the distribution plot of connected components, we could find interesting communities with special purposes which are disconnected from the rest of the Internet.

6.2. PageRank scores of Real Networks

We analyzed the PageRank scores of the nodes of real graphs, using PEGASUS. Figure 12 and 13 show the distribution of the PageRank scores for the Web graphs, and Figure 14 shows the evolution of PageRank scores of the LinkedIn and Wikipedia graphs. We have the following observations.

Power Laws in PageRank Distributions In Figure 12, 13, and 14, we observe power-law relations between the PageRank score and the number of nodes with such PageRank. Pandurangan et. al.(Pandurangan et al, 2002) observed such a power-law relationship for a 1.69 million network. Our result is that the same observation holds true for about *1,000 times* larger network with 1.4 *billion* pages snapshot of the Internet. The top 3 highest PageRank sites for the year 2002 are `www.careerbank.com`, `access.adobe.com`, and `top100.rambler.ru`. As expected, they have huge in- degrees (from ≈70K to ≈70M).

PageRank and the Gelling Point In the LinkedIn network (see Figure 14 (a)), we see a discontinuity for the power-law exponent of the PageRank distribution, before and after the gelling point at year 2003. For the year 2003 (up to the gelling point), the exponent is 2.15; from 2004 (after the gelling point), the

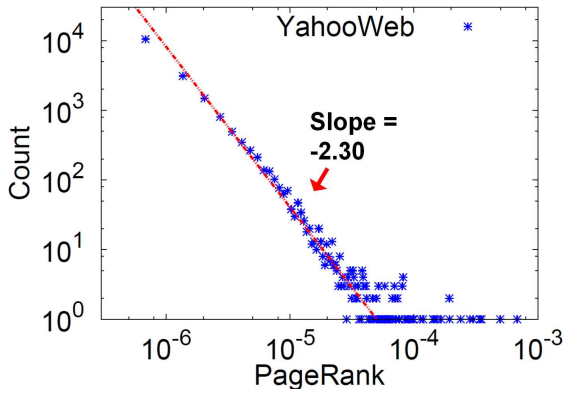


Fig. 12. PageRank distribution of YahooWeb. The distribution follows power law with exponent 2.30.

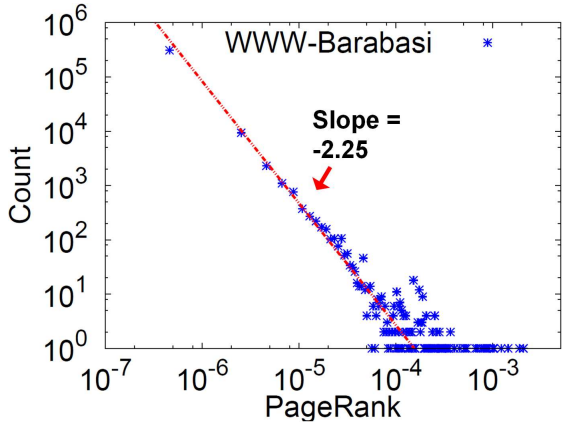
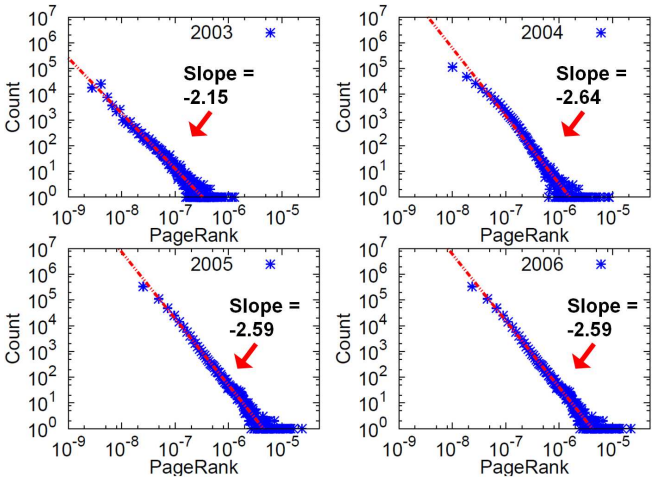
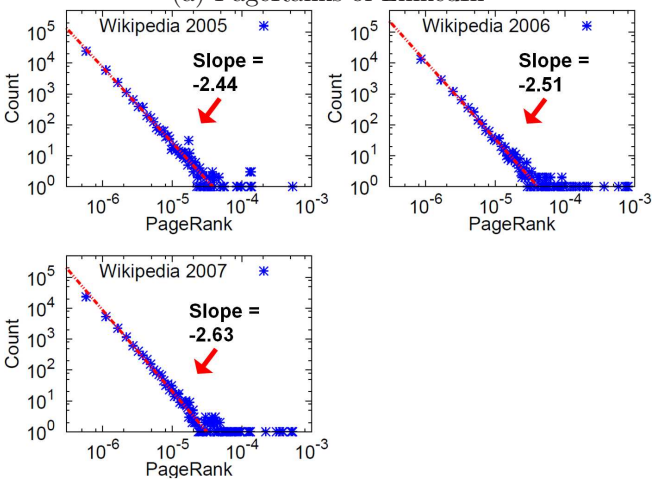


Fig. 13. PageRank distribution of WWW-Barabasi. The distribution follows power law with exponent 2.25.

exponent stabilizes around 2.59. Also, the maximum PageRank value at 2003 is around 10^{-6}, which is $\frac{1}{10}$ of the maximum PageRank from 2004. This behavior is explained by the emergence of the giant connected component at the gelling point: Before the gelling point, there are many small connected components where no outstanding node with large PageRank exists. After the gelling point, several nodes with high PageRank appear within the giant connected component. In the Wikipedia network (see Figure 14 (b)), we see the same behavior of the network after the gelling point. Since the gelling point is before year 2005, we see that the maximum PageRank-score and the slopes are similar for the three graphs from 2005.



(a) PageRanks of LinkedIn



(b) PageRanks of Wikipedia

Fig. 14. The evolution of PageRanks.(a) The distributions of PageRanks follows power-law. However, the exponent at year 2003, which is around the gelling point, is much different from year 2004, which are after the gelling point. The exponent increases after the gelling point and becomes stable. Also notice the maximum PageRank after the gelling point is about 10 times larger than that before the gelling point due to the emergence of the giant connected component. (b) Again, the distributions of PageRanks follows power-law. Since the gelling point is before year 2005, the three plots shows similar characteristics: the maximum PageRanks and the slopes are similar.

6.3. Diameter of Real Network

We analyzed the diameter and radius of real networks with PeGaSus. Figure 15 shows the radius plot of real networks. We have following observations:

Small Diameter For all the graphs in Figure 15, the average diameter was less than 6.09. This means that the real world graphs are well connected.

Constant Diameter over Time For LinkedIn graph, the average diameter

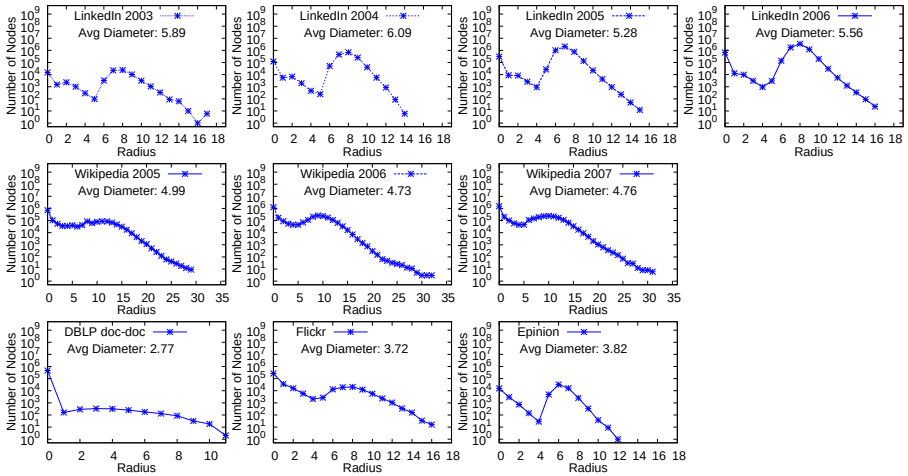


Fig. 15. Radius of real graphs. X axis: radius. Y axis: number of nodes. (Row 1) LinkedIn from 2003 to 2006. (Row 2) Wikipedia from 2005 to 2007. (Row 3) DBLP, flickr, Epinion. Notice that all the radius plots have the bimodal structure due to many smaller connected components(first mode) and the giant connected component(second mode).

was in the range of 5.28 and 6.09. For Wikipedia graph, the average diameter was in the range of 4.76 and 4.99. Note that the diameter do not monotonically increase as network grows: they remain constant or shrinks over time.

Bimodal Structure of Radius Plot For every plot, we observe bimodal shape which reflects the structure of these real graphs. The graphs have one giant connected component where majority of nodes belong to, and many smaller connected components whose size follows power law. Therefore, the first mode is at radius zero which comes from one-node components; second mode(e.g., at radius 6 in Epinion) comes from the giant connected component.

7. Conclusions

In this paper we proposed PEGASUS, a graph mining package for very large graphs using the HADOOP architecture. The main contributions are followings:

- We identified the common, underlying primitive of several graph mining operations, and we showed that it is a generalized form of a matrix-vector multiplication. We call this operation Generalized Iterative Matrix-Vector multiplication and showed that it includes the diameter estimation, the PageRank estimation, RWR calculation, and finding connected-components, as special cases.
- Given its importance, we proposed several optimizations (block-multiplication, diagonal block iteration, node renumbering etc) and reported the winning combination, which achieves more than *5 times* faster performance to the naive implementation.
- We implemented PEGASUS and ran it on M45, one of the 50 largest supercomputers in the world (3.5 Tb memory, 1.5Pb disk storage). Using PEGASUS and our optimized Generalized Iterative Matrix-Vector multiplication variants,

we analyzed real world graphs to reveal important patterns including power law tails, stability of connected components, and anomalous components. Our largest graph, "YahooWeb", spanned 120Gb, and is one of the largest publicly available graph that was ever studied.

Other open source libraries such as HAMA (Hadoop Matrix Algebra)can benefit significantly from PEGASUS. One major research direction is to add to PEGASUS an eigensolver, which will compute the top k eigenvectors and eigenvalues of a matrix. Another directions includes tensor analysis on HADOOP (Kolda et al, 2008), and inferences of graphical models in large scale.

Acknowledgements. The authors would like to thank YAHOO! for providing us with the web graph and access to the M45.

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 University of California Lawrence Livermore National Laboratory under contract DE-AC52-07NA27344 (LLNL-CONF-404625), subcontracts B579447, B580840.

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

Aggarwal G, Data M, Rajagopalan S, Ruhl M (2004) On the Streaming Model Augmented with a Sorting Primitive. In Proceedings of FOCS, 2004

Awerbuch B, Shiloach A (1983) New Connectivity and MSF Algorithms for Ultracomputer and PRAM. In ICPP, 1983

Brin S, Page L (1998) The anatomy of a large-scale hypertextual (Web) search engine. In WWW, 1998

Broder A, Kumar R, Maghoul F, Prabhakar R, Rajagopalan S, Stata R, Tomkins A, Wiener J (2000) Graph structure in the Web. In Computer Networks 33, 2000

Chaiken R, Jenkins B, Larson P, Ramsey B, Shakib D, Weaver S, Zhou J (2008) SCOPE: easy and efficient parallel processing of massive data sets. In VLDB, 2008

Chen C, Yan X, Zhu F, Han J (2007) gApprox: Mining Frequent Approximate Patterns from a Massive Network. In IEEE International Conference on Data Mining, 2007

Chen J, Zaiane O, Goebel R (2009) Detecting Communities in Social Networks using Max-Min Modularity. In SIAM International Conference on Data Mining, 2009

Cheng J, Yu J, Ding B, Yu P, Wang H (2008) Fast Graph Pattern Matching. In ICDE, 2008

Dean J, Ghemawat S (2004) MapReduce: Simplified Data Processing on Large Clusters. In OSDI, 2004

Dunbar R (1998) Grooming, Gossip, and the Evolution of Language. In Harvard Univ Press, 1998

Falkowski T, Barth A, Spiliopoulou M (2007) DENGRAPH: A Density-based Community Detection Algorithm. In Web Intelligence, 2007

Greiner J (1994) A comparison of parallel algorithms for connected components. In Proceedings of the 6th ACM Symposium on Parallel Algorithms and Architectures, 1994

Grossman R, Gu Y (2008) Data mining using high performance data clouds: experimental studies using sector and sphere. In ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, 2008

Hintsanen P, Toivonen H (2008) Finding Reliable Subgraphs from Large Probabilistic Graphs. In PKDD, 2008

Hirschberg D, Chandra A, Sarwate D (1979) Computing Connected Components on Parallel Computers. In Communications of the ACM, 1979

Kang U, Tsourakakis C, Faloutsos C (2009) PEGASUS: A Peta-Scale Graph Mining System - Implementation and Observations. In IEEE International Conference on Data Mining, 2009

Kang U, Tsourakakis C, Appel A, Faloutsos C, Leskovec J (2010) Radius Plots for Mining Tera-byte Scale Graphs: Algorithms, Patterns, and Observations. In SIAM International Conference on Data Mining, 2010

Karypis G, Kumar V (1999) Parallel multilevel k-way partitioning for irregular graphs. In SIAM Review, 1999

Ke Y, Cheng J, Yu J (2009) Top-k Correlative Graph Mining. In SIAM International Conference on Data Mining, 2009

Ketkar N, Holder L, Cook D (2005) Subdue: Compression-Based Frequent Pattern Discovery in Graph Data In OSDM, 2005

Kleinberg J (1998) Authoritative sources in a hyperlinked environment. In Proc. 9th ACM-SIAM SODA, 1998

Kolda T, Sun J (2008) Scalable Tensor Decompsitions for Multi-aspect Data Mining In IEEE International Conference on Data Mining, 2008

Kuramochi M, Karypis G (2004) Finding Frequent Patterns in a Large Sparse Graph. In SIAM Data Mining Conference, 2004

Lahiri M, Berger-Wolf T (2010) Periodic subgraph mining in dynamic networks. In Knowledge and Information Systems (KAIS), DOI: 10.1007/s10115-009-0253-8, 2010

Leskovec J, Chakrabarti D, Kleinberg J, Faloutsos C (2005) Realistic, Mathematically Tractable Graph Generation and Evolution, Using Kronecker Multiplication. In Practice of Knowledge Discovery in Databases (PKDD), 2005

Long B, Zhang Z, Yu P (2010) A general framework for relation graph clustering. In Knowledge and Information Systems (KAIS), DOI: 10.1007/s10115-009-0255-6, 2010

McGlohon M, Akoglu L, Faloutsos C(2008) Weighted graphs and disconnected components: patterns and a generator. In ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, 2008

Narasimhamurthy A, Greene D, Hurley N, Cunningham P (2010) Partitioning large networks without breaking communities. In Knowledge and Information Systems (KAIS), DOI: 10.1007/s10115-009-0251-x, 2010

Newman M (2005) Power laws, Pareto distributions and Zipf's law. In ontemporary Physics, 2005

Olston C, Reed B, Srivastava U, Kumar R, Tomkins A (2008) Pig latin: a not-so-foreign language for data processing. In SIGMOD, 2008

Pan J, Yang H, Faloutsos C, Duygulu P (2004) Automatic Multimedia Cross-modal Correlation Discovery. In ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, 2004

Pandurangan G, Raghavan P, Upfal E (2002) Using PageRank to Characterize Web Structure. In COCOON, 2002

Papadimitriou S, Sun J (2008) DisCo: Distributed Co-clustering with Map-Reduce. In IEEE International Conference on Data Mining, 2008

Peng W, Li T (2010) Temporal relation co-clustering on directional social network and author-topic evolution. In Knowledge and Information Systems (KAIS), DOI: 10.1007/s10115-010-0289-92, 2010

Pike R, Dorward S, Griesemer R, Quinlan S (2005) Interpreting the data: Parallel analysis with Sawzall. In Scientific Programming Journal, 2005

Qian T, Srivastava J, Peng Z, Sheu P (2009) Simultaneouly Finding Fundamental Articles and New Topics Using a Community Tracking Method. In PAKDD, 2009

Ralf L (2008) Google's MapReduce programming model – Revisited. In Science of Computer Programming, 2008

Ranu S, Singh A (2009) GraphSig: A Scalable Approach to Mining Significant Subgraphs in Large Graph Databases. In ICDE, 2009

Shiloach Y, Vishkin U (1982) An O(logn) Parallel Connectivity Algorithm. In Journal of Algorithms, 1982

Shrivastava N, Majumder A, Rastogi R (2008) Mining (Social) Network Graphs to Detect Random Link Attacks. In ICDE, 2008

Tsourakakis C, Kang U, Miller GL, Faloutsos C (2009) DOULION: counting triangles in massive graphs with a coin. In Knowledge Discovery and Data Mining (KDD), 2009

Tsourakakis C, Kolountzakis M, Miller GL Approximate Triangle Counting. In Arxiv 0904.3761, 2009

Tsourakakis C (2010) Counting triangles in real-world networks using projections. In Knowledge and Information Systems (KAIS), DOI: 10.1007/s10115-010-0291-2, 2010

Wang C, Wang W, Pei J, Zhu Y, Shi B (2004) Scalable Mining of Large Disk-based Graph

Databases. In ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, 2004
Wang N, Parthasarathy S, Tan K, Tung A (2008) CSV: Visualizing and Mining Cohesive Subgraph. In SIGMOD, 2008
Yan X, Han J (2002) gSpan: Graph-Based Substructure Pattern Mining. In IEEE International Conference on Data Mining, 2002
Zhu F, Yan X, Han J, Yu P (2007) gPrune: A Constraint Pushing Framework for Graph Pattern Mining. In PAKDD, 2007

Author Biographies

U Kang is currently a Ph.D. student in the Computer Science Department, at Carnegie Mellon University, USA. He holds a Diploma in Computer Science and Engineering from the Seoul National University, Korea. His main research interests lie in the fields of large scale graph mining.

Charalampos Tsourakakis is currently a Ph.D. candidate in the Machine Learning Department, at Carnegie Mellon University, USA. He holds a Diploma in Electrical and Computer Engineering from the National Technical University of Athens. His main research interests lie in the fields of computational biology, machine learning and (multi)linear algebra.

Christos Faloutsos is a Professor at Carnegie Mellon University. He has received the Presidential Young Investigator Award by the National Science Foundation (1989), the Research Contributions Award in ICDM 2006, fifteen best paper awards, and several teaching awards. He has served as a member of the executive committee of SIGKDD; he has published over 200 refereed articles, 11 book chapters and one monograph. He holds five patents and he has given over 20 tutorials and 10 invited distinguished lectures. His research interests include data mining for streams and graphs, fractals, database performance, and indexing for multimedia and bio-informatics data.

Correspondence and offprint requests to: U Kang, Computer Science Department, Carnegie Mellon University, Pittsburgh, PA 15213, USA. Email: ukang@cs.cmu.edu