Distributed Graph Summarization

Xingjie Liu[*] Yuanyuan Tian[†] Qi He[◇][*] Wang-Chien Lee[§] John McPherson[†]

[*] Square Inc jliu@squareup.com

[†] IBM Almaden Research Center {ytian, jmcphers}@us.ibm.com

[◇] LinkedIn Inc qhe@linkedin.com

[§] The Pennsylvania State University wlee@cse.psu.edu

ABSTRACT

Graph has been a ubiquitous and essential data representation to model real world objects and their relationships. Today, large amounts of graph data have been generated by various applications. Graph summarization techniques are crucial in uncovering useful insights about the patterns hidden in the underlying data. However, all existing works in graph summarization are single-process solutions, and as a result cannot scale to large graphs. In this paper, we introduce three distributed graph summarization algorithms to address this problem. Experimental results show that the proposed algorithms can produce good quality summaries and scale well with increasing data sizes. To the best of our knowledge, this is the first work to study distributed graph summarization methods.

1. INTRODUCTION

Graph has been a ubiquitous and essential data representation to model real world objects and their relationships. Today, large amounts of graph data have been generated by various applications, including social networks, biological networks, WWW, etc. With the overwhelming wealth of information encoded in these graphs, there is a crucial need for tools to summarize large graphs into concise forms that can be easily understood.

Graph summarization has attracted a lot of research interests recently. Various graph summarization techniques [12, 9, 13] have been proposed to help users extract and understand the information encoded in large graphs. The goal of graph summarization methods is to produce a compact and informative *summary graph* that uncovers the underlying topology characteristics of the original graph. For example, Figure 1(b) and Figure 1(c) show two summaries of the original graph G in Figure 1(a). The summaries themselves are also graphs. Every node in a summary, called a *super-node*, contains a set of nodes from the original graph. Every edge in a summary, called a *super-edge*, represents restrictively an all-to-all relationship between the nodes in the corresponding super-nodes. For example, in Figure 1(b), the super-edge between $\mathcal{V}_1$ and $\mathcal{V}_2$ means that every node in $\mathcal{V}_1$ (v_1) is connected to every node in $\mathcal{V}_2$ $(v_2$ and $v_3)$. Notice that not all summaries are lossless. The summary graph in Figure 1(b) exactly recreates the original graph, but the summary graph in Figure 1(c) reconstructs into a graph different from the original. In particular, the edge $\langle v_2, v_4\rangle$ is present in the original graph but is *missing* in the summary, whereas the edge $\langle v_2, v_3\rangle$ is *spurious* in the summary as it does not exist in the original graph. The total number of missing and spurious edges is the *error* associated a summary graph. Intuitively, a smaller summary, which is easier to visualize and understand, also tends to introduce more errors. Therefore, the challenge for graph summarization is to produce relatively small summaries while minimizing the errors.

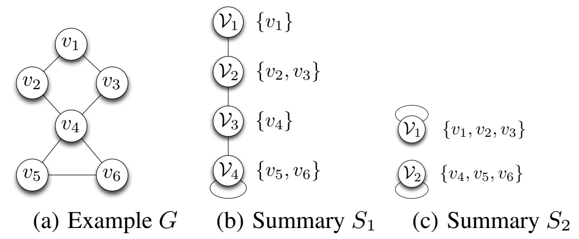

v_1
v_2
v_3
v_4
v_5
v_6
$\mathcal{V}_1$ $\{v_1\}$
$\mathcal{V}_2$ $\{v_2, v_3\}$
$\mathcal{V}_3$ $\{v_4\}$
$\mathcal{V}_4$ $\{v_5, v_6\}$
$\mathcal{V}_1$ $\{v_1, v_2, v_3\}$
$\mathcal{V}_2$ $\{v_4, v_5, v_6\}$
(a) Example G
(b) Summary S_1
(c) Summary S_2

Figure 1: Graph Summarization Example

With the skyrocketing expansion of graph data size in recent years, mining and visualizing large graphs become more and more difficult. Graph summarization can be used to help compress large graphs into more manageable size to visualize and study. Since the summaries are also graphs themselves, investigation of various graph properties can be conducted on the summary graphs instead. Although there are several existing graph summarization methods that are quite effective and efficient in producing summaries [9, 12], all of them are single-process in-memory solutions. Processing millions to billions of nodes and edges (with intermediate data structures) easily requires more memory than provided in a single machine and parallelization of computation for efficiency. This paper exactly addresses this need. To the best of our knowledge, this is the first work to study distributed graph summarization methods.

There are a number of challenges for implementing distributed graph summarization algorithms. First of all, as nodes and edges are distributed in different machines, a seemingly simple operation in the centralized graph summarization algorithm requires message passing and careful coordination across multiple node in the distributed environment. Secondly, the centralized algorithm doesn't need to worry about how computation is distributed, but a good distributed graph summarization method should fully distribute computation across different machines for efficient parallelization. Last but not the least, as computation and communication costs will be the dominating factors in the distributed graph summarization al-

[*]This work was done while this author was in IBM Almaden Research Center.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

CIKM'14, November 3–7, 2014, Shanghai, China.
Copyright 2014 ACM 978-1-4503-2598-1/14/11 ...$15.00.
http://dx.doi.org/10.1145/2661829.2661862.

gorithm, smart techniques are needed to avoid unnecessary communication and computation as much as possible.

In this paper, we proposed three distributed algorithms for large scale graph summarization, implemented on top of Apache Giraph (`giraph.apache.org`), an open source distributed graph processing platform. The first algorithm, DistGreedy, is a non-trivial adaptation of a centralized greedy algorithm. However, the greediness of this algorithm requires examining all pairs of nodes with 2-hop distance, thus causes a large amount of computation and communication cost. The second algorithm, called Dist-Random, reduces the number of examined node pairs using random selection. But randomness negatively affects the effectiveness of the algorithm. Our last algorithm, Dist-LSH, addresses the limitations of the previous two algorithms. It employs a novel technique called, Striped-MinHash, to directly pinpoint the good candidates for examination, thus completely eliminate the computation and the network cost associated with unnecessary examinations. Through experiments we demonstrate the effectiveness and efficiency of the proposed algorithms. Although the proposed algorithms are evaluated under Apache Giraph, they are generic for other parallel processing frameworks because data and computations are partitioned into each graph summary nodes.

2. PRELIMINARIES

Graph summarization was first proposed independently by [12] and [9] in 2008. In both works, a *summary graph* is defined as a compact graph representing the original graph topology (see Figure 1), and *graph summarization* is the process to construct a summary graph from a given graph. The two works differ mainly in whether the nodes in the original graph are of the same type in the problem definition. In this work, we adopt the same setting as in [9], i.e., all nodes are of the same type. For ease of presentation, we consider only undirected graphs in this paper. Adapting methods discussed in this paper for directed graphs is fairly straightforward, hence omitted due to space limit.

Given a graph $G = (V, E)$, a *summary graph* for G is denoted as $S(G) = (V_S, E_S)$. The summary $S(G)$ is an aggregated graph, in which $V_S = \{\mathcal{V}_1, \mathcal{V}_2, \cdots, \mathcal{V}_k\}$ is a partition of the nodes in V ($\bigcup_{i=1}^{k} \mathcal{V}_i = V$ and $\forall i \neq j$, $\mathcal{V}_i \bigcap \mathcal{V}_j = \emptyset$). We call each $\mathcal{V}_i$ a *super-node*, representing an aggregation of a subset of the original nodes. For simplicity, we use $\mathcal{V}(v)$ to denote the super-node that an original node v belongs to. In addition, Each $\langle \mathcal{V}_i, \mathcal{V}_j \rangle \in E_S$ is called a *super-edge*, representing all-to-all connections between nodes in $\mathcal{V}_i$ and nodes in $\mathcal{V}_j$. In other words, $\forall v_m \in \mathcal{V}_i, v_n \in \mathcal{V}_j$, $\langle v_m, v_n \rangle \in \langle \mathcal{V}_i, \mathcal{V}_j \rangle$. Due to the all-to-all connection representation of super-edges, a graph summarization process may introduce information loss. Let $\Pi_{i,j}$ denote the all-to-all connections between the two corresponding node sets $\mathcal{V}_i$ and $\mathcal{V}_j$, and $A_{i,j}$ represent the set of actual edges between them in the original graph. If the super-edge $\langle \mathcal{V}_i, \mathcal{V}_j \rangle$ exists in the summary graph, then $|\Pi_{i,j}| - |A_{i,j}|$ *spurious* edges are introduced. Otherwise, $|A_{i,j}|$ edges are *missing* from the summary graph. More formally, we define the error associated with a pair of super-nodes $\mathcal{V}_i$ and $\mathcal{V}_j$ in a summary graph as follows:

$$e_{i,j} = \begin{cases} |\Pi_{i,j}| - |A_{i,j}|, & \text{if } \langle \mathcal{V}_i, \mathcal{V}_j \rangle \in E_S \\ |A_{i,j}|, & \text{if } \langle \mathcal{V}_i, \mathcal{V}_j \rangle \notin E_S. \end{cases} \quad (1)$$

Accordingly, the total error for a summary graph $S(G)$ can be defined as $E(S(G)) = \sum_{i=1}^{|V_S|} \sum_{j=1}^{|V_S|} e_{i,j}$.

Once a user selects a summary resolution, i.e. the number of super-nodes, naturally our goal is to generate a summary graph that minimizes the total error.

Graph Summarization Problem: Given a graph G and a desired number of super-nodes k, compute a summary graph $S(G)$ with k super-nodes, such that the summary error is minimized.

It has been proved that graph summarization is NP-hard [12]. The difficult part is determining the super-nodes V_S, Once the super-nodes are decided, constructing the super-edges with minimum summary error can be achieved in polynomial time.

Recall that the error between a pair of super-nodes $\mathcal{V}_i$ and $\mathcal{V}_j$ comes from either $|\Pi_{i,j}| - |A_{i,j}|$ spurious edges, or $|A_{i,j}|$ missing edges. Accordingly, the optimal edge assignment strategy is simply adding an super-edge $\langle \mathcal{V}_i, \mathcal{V}_j \rangle$ when $|A_{i,j}| < \frac{1}{2}|\Pi_{i,j}|$, or leaving $\mathcal{V}_i$ and $\mathcal{V}_j$ unconnected otherwise.

Under this super-edge assignment strategy, the connection error among each pair of super-nodes $\mathcal{V}_i$ and $\mathcal{V}_j$ is:

$$e^*_{i,j} = \min\{|\Pi_{i,j}| - |A_{i,j}|, |A_{i,j}|\}. \quad (2)$$

Thus, the graph summarization problem is essentially the problem of determining the k super-nodes.

Centralized Algorithms. In [9], two centralized heuristic-based algorithms for graph summarization, Greedy and Random, are proposed. Although our graph summarization definition is slightly different, these two algorithms are still applicable after minor changes. Below, we will briefly describe the two centralized algorithms, as they serve as the baseline for our distributed algorithms.

Both the greedy and the randomized algorithms start with a summary graph initialized as the original graph, and iteratively merge a pair of super-nodes into one super-node to form a summary graph with a lower resolution, until k super-nodes remain in the final summary graph. In each iteration, both algorithms choose a super-node pair that introduces low error increase. The error increase of merging two super-nodes $\mathcal{V}_i$ and $\mathcal{V}_j$ to form one super-node $\mathcal{V}_m$ is:

$$\Delta_{i,j} = e^*_{m,\cdot} - (e^*_{i,\cdot} + e^*_{j,\cdot} - e^*_{i,j}) \quad (3)$$

Here, $e^*_{i,\cdot} = \sum_{\forall \mathcal{V}_j \in V_S} e^*_{i,j}$ means the total error associated with super-node $\mathcal{V}_i$. In the above equation, $e^*_{m,\cdot}$ denotes the total connection error associated with the merged super-node $\mathcal{V}_m$, and $e^*_{i,\cdot} + e^*_{j,\cdot} - e^*_{i,j}$ is the total errors associated with $\mathcal{V}_i$ and $\mathcal{V}_j$ before the merge. As $e^*_{i,j}$ is counted twice in $e^*_{i,\cdot} + e^*_{j,\cdot}$, we need to subtract $e^*_{i,j}$ from $e^*_{i,\cdot} + e^*_{j,\cdot}$.

The two algorithms differ in the policy of choosing which pair of nodes to merge in each iteration. The Greedy algorithm always examines all node pairs within 2-hop aways and chooses the pair with the minimum error (smallest $\Delta_{i,j}$), while the Random algorithm first randomly picks a node and merges it with the best node in its 2-hop neighborhood.

3. GIRAPH OVERVIEW

Giraph is an open source implementation of Pregel [8] proposed by Google. It supports both iterative algorithms and vertex-to-vertex communication in a distributed graph, thus is a natural fit for us to implement distributed graph summarization algorithms. Note that although this paper only demonstrates how distributed graph summarization algorithms are implemented in Giraph, the same algorithms can be easily implemented in other similar graph processing systems, such as GraphLab [7] and Trinity [11], with minor adaption.

A typical Giraph program consists of an input step, where the graph is initialized (e.g., loading and distributing vertices to worker machines), followed by a sequence of iterations, called *supersteps*, separated by global synchronization barriers, and finally an output step to write down the results.

Giraph employs a vertex-centric model. Each vertex is considered an independent computing unit that inherits from the predefined **Vertex** class. Each vertex has a unique id, a set of outgoing edges and application-dependent attributes of the vertex and its edges which are necessary for the computation. A vertex instance carries two states (*active* and *inactive*). In superstep i, each active vertex can receive messages sent to it by other vertices in superstep $i-1$, query and update the information of itself and its edges, initiate graph topology mutation, communicate with global aggregation variables, and send messages to other vertices for the next superstep $i+1$. All these computation logics are executed in a user-overridable function named COMPUTE(). After all active vertices finish their local COMPUTE() functions in a superstep, a global synchronization phase allows global data to be aggregated from each vertex's submitted values, and messages created by each vertex to be delivered to their destinations. At the beginning, all vertices are *active*. A vertex can voluntarily deactivate itself by calling VOTETOHALT() or be passively activated by some incoming messages from other vertices. The overall program terminates when every vertex *votes to halt* and there is no message to any vertex.

In Giraph, messaging is the major mechanism for communication. By calling SENDMESSAGE$(dest, msg)$, a vertex can send a message to the destination vertex $dest$ with the message body msg. Note that a vertex can send message(s) to any vertex (not only to its neighbors). Aggregator is a mechanism for global communication and synchronization. By calling AGGREGATE$(aggr, val)$, the vertex provides a value val for the global aggregator variable $aggr$. After each superstep, Giraph aggregates the provided values from all vertices for each aggregator variable. The aggregator variables will be available to all vertices in the next superstep.

The Giraph computing framework consists of one master and a number of workers. The master is responsible for coordinating and controlling the computation process, and each worker works on a subset of vertices and executes their computations. At start, a customizable PREAPPLICATION() call is executed in the master to initialize application-specific global data structure (e.g. global aggregators). Then, each superstep starts with calling the overridable PRESUPERSTEP() function and ends with calling another overridable POSTSUPERSTEP() function in the master for updating global data structures in each superstep. When the whole job finishes, a customizable POSTAPPLICATION() function is executed in the master for finalizing the global data structure.

4. DISTRIBUTED GRAPH SUMMARIZATION

For distributed graph summarization, we follow the same iterative merging mechanism in the centralized algorithm in Section 2: starting from the original graph as the summary (each node is a super-node) and iteratively merging super-nodes until k super-nodes left. However, there are two major challenges to achieve this iterative merging mechanism in the Giraph distributed environment:

1. It is very easy to decide which pairs of super-nodes are good candidates for merge and perform these merge operations in a centralized algorithm, since everything runs in a single process with shared memory. But in the Giraph distributed environment, all the decisions and operations have to be done in a distributed way through message passing and synchronization.

2. The centralized algorithm only merges the optimal pair of super-nodes in each iteration. And it requires $N-k$ iterations to produce a summary of size k, where N is the number of nodes in the original graph. However, to fully utilize the parallelization in a distributed environment, we need to find multiple pairs of nodes to merge, and simultaneously merge them in each iteration.

The above two challenges define two crucial tasks that a distributed graph summarization algorithm needs to perform in each iteration: Candidates-Find task and Merge task. The Candidates-Find task decides on the pairs of super-nodes to be merged, whereas the Merge task executes these merges.

In this paper, we propose three distributed graph summarization algorithms: DistGreedy, DistRandom and DistLSH. The three algorithms share the same operations in the Merge task, but differ in how merge candidates are selected. DistGreedy and DistRandom are modeled after the centralized Greedy and Random algorithms introduced in Section 2, respectively. Both suffer from significant drawbacks. To address their limitation, we propose a novel distributed graph summarization algorithm, called DistLSH.

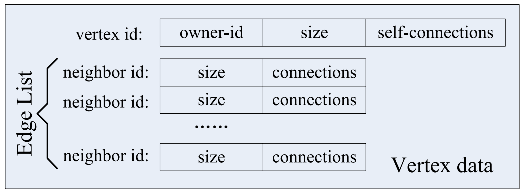


Figure 2: Giraph vertex's data structure

For all three algorithms, we can naturally map each *super-node* in the summary graph as a Giraph vertex and the neighbors of the super-node as the adjacent vertices for the Giraph vertex. The data structure for the Giraph vertex is shown in Figure 2. Each Giraph vertex has three attributes associated with vertices and two attributes associated with edges:

- *vertex.owner-id* points to which other super-node this super-node has been merged to; if *owner-id* equals to this super-node's id, then it indicates that this super-node is legitimate (hasn't been merged into any other super-node yet) in the current summary.
- *vertex.size* records the number of nodes in the original graph contained in this super-node.
- *vertex.selfconn* represents the number of edges in connecting the nodes inside this super-node.
- *edge.size* caches the number of nodes in the other adjacent super-node of the edge to avoid an additional round of query for this value.
- *edge.conn* is the number of edges in the original graph between this super-node and the neighbor.

Since the introduced data structure is associated to edges, the space complexity is still linear to the number of edges $O(|E|)$. Based on this data structure, we rewrite Eq. 2 as

$$e^*_{i,j} = \min\{size_i \times size_j - conn_{i,j}, conn_{i,j}\}. \quad (4)$$

Algorithm 1 shows an overview of the three distributed graph summarization algorithms. Each iteration of the distributed algorithms is processed in multiple supersteps. The first few supersteps perform the Candidates-Find task (details will be provided in Section 5), and the remaining supersteps perform the Merge task (details will be provided in Section 6). We use an aggregator, called *ExecutionPhase*, as the global coordinator to indicate which phase of an iteration the COMPUTE() function is currently executing in. Based on the previous value of *ExecutionPhase*, we can set the right value to this aggregator in the PRESUPERSTEP function before each superstep starts. Another aggregator, called *ActiveNodes*, is used to keep track of the number of super-nodes in the current summary. When the summary size is less or equal to the required size k, the value of the ExecutionPhase will be set to DONE. In

Algorithm 1: Distributed Graph Summarization Overview

```
// Executed by each vertex in each superstep.
COMPUTE(MessageIterator msgs)
    phase ←GETAGGREGATORVALUE("ExecutionPhase");
    if phase=DONE then
        VOTETOHALT();
    else
        if phase is part of Candidates-Find task then
            FINDCANDIDATES(msgs)       // see Section 5
        else
            MERGE(msgs)                // see Section 6

// Executed before each superstep starts.
PRESUPERSTEP()
    prevPhase ←GETAGGREGATORVALUE("ExecutionPhase");
    active-nodes ←GETAGGREGATORVALUE("ActiveNodes");
    if active-nodes $\le k$ then
        current-phase ← DONE
    else
        current-phase ←NEXTPHASE()
    SETAGGREGATORVALUE("ExecutionPhase",
    current-phase);
```

this case, in the COMPUTE() function, every vertex will vote to halt. Then the whole program will finish.

At the end of the program, the resulting graph will be written down. Using this graph, we can derive the summary graph. For each vertex, the *owner-id* will tell us whether it is a super-node in the resulting summary, using the adjacent vertices with their $size$ and $conn$ information, we can infer whether a super-edge should exist or not (see Section 2). If the actual nodes in each super-node are also desired, again the *owner-id* can help us derive the membership information. Essentially, the id of each vertex and its *owner-id* form an edge in a *ownership forest*. The leaf nodes are the nodes from the original graph and the roots are the super-nodes in the summary. A simple connected component algorithm on this graph will compute all the memberships. The label propagation based connected component algorithm [5] can be implemented straight forwardly in Giraph. If the resulting summary has size less than k (this could happen because multiple merges happen in each iteration), we randomly choose a number of roots in the forest and reverse the merges to get the summary of size k. However, in most practical cases, users are content enough with a summary of a size roughly equal to k. Since these post processing steps are relatively trivial, we omit the details in the interest of space.

5. FINDING MERGE CANDIDATES

We now explain in detail how to find pairs of super-nodes as candidates to merge in DistGreedy, DistRandom and DistLSH.

5.1 DistGreedy

As DistGreedy is based on the centralized Greedy algorithm, it looks at super-nodes that are 2-hops away to each other and thrives to find the pairs with minimum error increase. To control the number of super-node pairs to be merged in each iteration, we use a threshold called *ErrorThreshold* as the cutoff for which pairs qualify as merge candidates. More precisely, every pairs with error increase less than *ErrorThreshold* will become merge candidates. Initially, *ErrorThreshold* is 0, which means we start with merging pairs with no error increase at all. Whenever the number of merge candidates fall below 5% of the current summary size, the algorithm increases *ErrorThreshold* by a controllable parameter, called *ThresholdIncrease*, for the subsequent iterations.

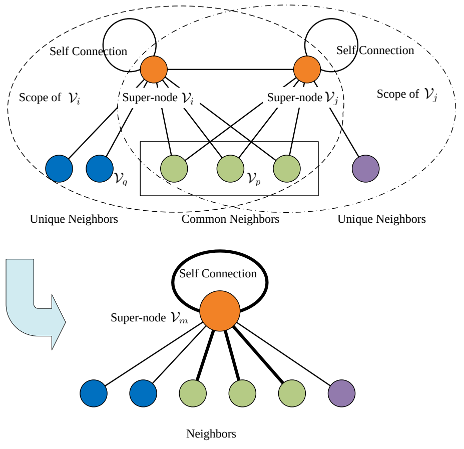


Figure 3: Node merge: from before-merge to after-merge.

The major task in the DistGreedy algorithm is to compute the actual error increase for each pair of 2-hop-away super-nodes. This is fairly simple in the centralized Greedy algorithm, but becomes much more complex in the distributed environment, as the information to compute the error increase is distributed in different places. As shown in Figure 3, the error increase for merging a pair of super-nodes $\mathcal{V}_i$ and $\mathcal{V}_j$ can be decomposed into 3 parts:

- Common Neighbor Error Increase ($\Delta^{\text{com}}_{i,j}$): This part of error increase is associated with the connections to the common neighbors of the two super-nodes.
- Unique Neighbor Error Increase ($\Delta^{\text{uni}}_{i,j}$): $\Delta^{\text{uni}}_{i,j}$ captures the error increase brought by the connections to the unique neighbors of the two super-nodes.
- Self Error Increase ($\Delta^{\text{self}}_{i,j}$): This last part of error increase comes from the self connections of the two super-nodes as well as the connection between the two super-nodes if there is any.

In Figure 3, we use *scope* to define all the information each Giraph vertex knows: this super-node's own *size*, *selfconn*, *conn* to all its neighbors and the neighbors' *sizes*. It is clear that the bits and pieces used to compute the total error increase for merging two super-nodes are distributed in the scopes of different Giraph vertex. In the following, we will describe in detail how to compute the different parts of the error increase using $\mathcal{V}_i$ and $\mathcal{V}_j$ in Figure 3 as an example.

Computing $\Delta^{\text{com}}_{i,j}$ requires the error increase associated with the connections of $\mathcal{V}_i$ and $\mathcal{V}_j$ to all their common neighbors. For a common neighbor, say $\mathcal{V}_p$, the error before the merge is $e^*_{i,p} + e^*_{j,p}$ (ref. Eq. 4). The error after the merge $e^*_{m,p} = \min\{(size_i + size_j) \times size_p - (conn_{i,p} + conn_{j,p}), conn_{i,p} + conn_{j,p}\}$.

Thus, the error increase of merging $\mathcal{V}_i$ and $\mathcal{V}_j$ w.r.t. common neighbor $\mathcal{V}_p$ is $\Delta^p_{i,j} = e^*_{m,p} - e^*_{i,p} - e^*_{j,p}$. Luckily, all the information needed to compute $\Delta^p_{i,j}$ can be found in the scope of the common neighbor $\mathcal{V}_p$. As a result, the total $\Delta^{\text{com}}_{i,j}$ can be collectively computed by all the common neighbors, $\Delta^{\text{com}}_{i,j} = \sum_{\mathcal{V}_p} \Delta^p_{i,j}$.

For $\Delta^{\text{uni}}_{i,j}$, the computation requires only unique neighbors of each super-node. As a result, $\mathcal{V}_i$ and $\mathcal{V}_j$ can independently compute this part of error increase. As an example, for the unique neighbor $\mathcal{V}_q$ in Figure 3, $\Delta^q_{i,j} = e^*_{m,q} - e^*_{i,q}$. The error increase associated with $\mathcal{V}_i$'s unique neighbors thus is $\Delta^{\text{uni-}i}_{i,j} = \sum_{\mathcal{V}_q} \Delta^q_{i,j}$. And $\mathcal{V}_j$ can similarly compute $\Delta^{\text{uni-}j}_{i,j}$. The total $\Delta^{\text{uni}}_{i,j}$ is a simple sum of the two: $\Delta^{\text{uni}}_{i,j} = \Delta^{\text{uni-}i}_{i,j} + \Delta^{\text{uni-}j}_{i,j}$. However, a tricky issue is that each super-node has to know who are the unique neighbors for each candidate

Algorithm 2: FINDCANDIDATES() for DistGreedy

```
1  FINDCANDIDATES(MessageIterator msgs)
       // Show execution flow of merge candidate $\mathcal{V}_i$ and $\mathcal{V}_j$.
2      phase ← GETAGGREGATORVALUE("ExecutionPhase");
3      switch phase do
4          case Common-Neighbor-Error-Increase Phase
           // Executed in vertex $\mathcal{V}_p$.
5              for $\forall \mathcal{V}_i, \mathcal{V}_j \in \mathcal{N}_p$ do
6                  Compute $\Delta^p_{i,j}$;
                   // Register common neighbor and send $\Delta^p_{i,j}$ to merge host.
7                  SENDMESSAGE($\mathcal{V}_i, \langle \mathcal{V}_j, \mathcal{V}_p, \Delta^p_{i,j} \rangle$);
8                  SENDMESSAGE($\mathcal{V}_j, \langle \mathcal{V}_i, \mathcal{V}_p \rangle$);
9          case Unique-Neighbor-Error-Increase Phase
           // Executed in vertex $\mathcal{V}_j$.
10             for $\langle \mathcal{V}_j, \mathcal{V}_p, \Delta^p_{i,j} \rangle \in$ messages do
11                 $\Delta^{\text{com}}_{ij} \leftarrow \Delta^{\text{com}}_{i,j} + \Delta^p_{ij}$;
12                 Common Neighbors $\text{CN}_{i,j} \leftarrow \text{CN}_{i,j} \cup \{\mathcal{V}_p\}$
13             for $\mathcal{V}_q \in \mathcal{N}_i \wedge \mathcal{V}_q \notin CN_{ij}$ do
14                 Compute $\Delta^q_{i,j}$;
15                 $\Delta^{\text{uni-}j}_{i,j} \leftarrow \Delta^{\text{uni-}j}_{i,j} + \Delta^q_{i,j}$
16             if $i < j$ then
                   // Send $\Delta^{\text{uni-}j}_{i,j}$ as well as own vertex info to the other merge candidate.
17                 SENDMESSAGE($\mathcal{V}_i, \langle \mathcal{V}_j, \text{size}_j, \text{conn}_j, \Delta^{\text{uni-}j}_{i,j} \rangle$);
18             else
19                 Store $\Delta^{\text{uni-}j}_{i,j}$ ;
20         case Self-Error-Increase Phase  // Executed in vertex $\mathcal{V}_i$.
21             for $\langle \mathcal{V}_j, size_j, conn_j, \Delta^{uni\text{-}j}_{i,j} \rangle \in$messages do
22                 Compute $\Delta^{\text{self}}_{i,j}$;
23                 $\Delta_{i,j} \leftarrow \Delta^{\text{self}}_{i,j} + \Delta^{\text{com}}_{i,j} + \Delta^{\text{uni-}i}_{i,j} + \Delta^{\text{uni-}j}_{i,j}$;
24                 if $\Delta_{i,j} \leq$ ErrorThreshold then
                       // Notify the other merge candidate $\mathcal{V}_j$ to merge to me.
25                     SENDMESSAGE($\mathcal{V}_j, \langle \mathcal{V}_j \rightarrow \mathcal{V}_i \rangle$);
```

merge partner . This requires the common neighbors to register with the two super-nodes before hand.

Computing $\Delta^{\text{self}}_{i,j}$ requires collaboration between $\mathcal{V}_i$ and $\mathcal{V}_j$. Between the two super-nodes, the one with a larger id, say $\mathcal{V}_j$, sends its *selfconn* to $\mathcal{V}_i$. Then at $\mathcal{V}_i$, $\Delta^{\text{self}}_{i,j}$ can be computed as $\Delta^{\text{self}}_{i,j} = e^*_{m,m} - e^*_{i,i} - e^*_{j,j} - e^*_{i,j}$, where a self-loop error $e^*_{x,x}$ $(x = \{m, i, j\})$ is $\min\{\frac{size_x(size_x-1)}{2} - selfconn_x, selfconn_x\}$, and $size_m = size_i + size_j$.

Lastly, all the three parts of error increase will be aggregated at the super-node with the smaller id, $\mathcal{V}_i$ in our example. This requires messages from common neighbors for $\Delta^{\text{com}}_{i,j}$ and messages from $\mathcal{V}_j$ for $\Delta^{\text{uni-}j}_{i,j}$ and for $\mathcal{V}_j$*.selfconn*. Then $\mathcal{V}_i$ can simply test whether the total error increase is below *ErrorThreshold* or not to decide on whether the two super-nodes should be merged.

Algorithm 2 shows the pseudo code for DistGreedy's FindCandidates function. There are three phases for this function. These three phases correspond to the computation of the three different parts of error increase. In different phases, the Giraph vertex plays different roles in the computation. Again, here aggregator *ExecutionPhase* is to indicate which phase the current superstep is in. In the first phase, the Giraph vertex plays the role of a common neighbor, $\mathcal{V}_p$, to a potential merge candidate $\mathcal{V}_i$ and $\mathcal{V}_j$. Since the neighbors of $\mathcal{V}_p$ are all two hops away from each other, all neighbor pairs are potential candidates to merge. As a result, $\mathcal{V}_p$ will compute $\Delta^p_{i,j}$ for all pairs of neighbors $\mathcal{V}_i$ and $\mathcal{V}_j$ where $i \neq j$, and send $\Delta^p_{i,j}$ to the super-node in the pair with the smaller id, $\mathcal{V}_i$. It also sends a message to $\mathcal{V}_i$ and $\mathcal{V}_j$ to register itself as a common neighbor. In the second phase, the current Giraph vertex plays the role as one super-node in a potential merge candidate $(\mathcal{V}_i, \mathcal{V}_j)$. If it is the super-node with the smaller id, $\mathcal{V}_i$, then it will receive a message $\Delta^p_{i,j}$ from each common neighbor $\mathcal{V}_p$ of $\mathcal{V}_i$ and $\mathcal{V}_j$. It will aggregate these values to compute $\Delta^{\text{com}}_{i,j}$. At the same time, it also registers $\mathcal{V}_p$ as a common neighbor of $\mathcal{V}_i$ and $\mathcal{V}_j$. Since now it knows all the common neighbors, it can infer who are the unique neighbors and compute $\Delta^{\text{uni-}i}_{i,j}$. On the other hand, if the current Giraph vertex is the super-node with the larger id, $\mathcal{V}_j$, then it will receive a message from each common neighbor $\mathcal{V}_p$ of $\mathcal{V}_i$ and $\mathcal{V}_j$, indicating that $\mathcal{V}_p$ is a common neighbor of $\mathcal{V}_i$ and $\mathcal{V}_j$. As a result, the unique neighbors are known, and $\Delta^{\text{uni-}j}_{i,j}$ can be computed and sent to $\mathcal{V}_i$. In addition, it also send its *selfconn*$_j$ to $\mathcal{V}_i$. In the third phase, the current Giraph vertex plays the role of the super-node with the smaller id, $\mathcal{V}_i$, in a potential merge candidate $(\mathcal{V}_i, \mathcal{V}_j)$. Now, $\mathcal{V}_i$ has already computed $\Delta^{\text{com}}_{i,j}$ and $\Delta^{\text{uni-}i}_{i,j}$ in the second phase, it will also receive $\Delta^{\text{uni-}j}_{i,j}$ and *selfconn*$_j$. Using *selfconn*$_j$ and its own information, $\mathcal{V}_i$ can compute $\Delta^{\text{self}}_{i,j}$. At the end, the total error increase $\Delta_{i,j}$ can be computed and a decision on whether to merge $\mathcal{V}_i$ and $\mathcal{V}_j$ will be made based on *ErrorThreshold*.

To analyze the time complexity of this algorithm, we use d to denote the average number of neighbors of a vertex. Therefore, the average number of 2-hop away neighbors for a vertex is d^2. In DistGreedy, the computation of all the different $\Delta^{\text{com}}_{i,j}$ for each vertex $\mathcal{V}_i$ is essentially a loop over all its 2-hop away neighbors. So, its time complexity is $O(d^2 \cdot N)$, where N is the total number of vertices. Same complexity analysis applies to the $\Delta^{\text{self}}_{i,j}$ computation phase. The $\Delta^{\text{uni}}_{i,j}$ computation phase iterates through each 1-hop neighbor $\mathcal{V}_q$ to compute $\Delta^q_{i,j}$ for every 2-hop neighbor $\mathcal{V}_j$, and thus has a time complexity of $O(d^3 \cdot N)$. Overall, DistGreedy has a time complexity of $O(d^3 \cdot N)$ for each Candidates-Find step.

5.2 DistRandom

The DistGreedy algorithm described in the previous section blindly examines all super-node pairs of 2 hops away to each other to see whether they should be merged. This process incurs a large amount of computation and network messages. In order to reduce the number of super-node pairs to be examined, DistRandom randomly selects some super-node pairs to examine. In the selection process, we want every super-node to have a chance to be merged with another super-node. Similar to DistGreedy, DistRandom also has the following three supersteps.

1. Every super-node randomly selects one neighbor and sends a message to this neighbor, including its *size*, *selfconn*, all neighbors' *size* and *conn*.

2. The neighbor receives the message and forwards it to a random chosen neighbor with an id smaller than the sender.

3. The 2-hop away neighbor receives this message and use it to compute the error increase. If the error increase is above *ErrorThreshold*, then a merge decision is made.

On average each super-node will receive a message from one of its 2-hop neighbors. Calculating the error increase takes $O(d)$ time. So, DistRandom has a time complexity of $O(d \cdot N)$ for one Candidates-Find step.

5.3 DistLSH

For a large network with millions of nodes, evaluating all possible super-node pairs in 2-hop distance can be prohibitively expensive. As many such super-node pairs do not qualify the merge criteria, a lot of work is wasted. Although DistRandom can reduce the number of super-node pairs examined, by random sampling it also misses some really good merge candidates. In this section, we introduce another distributed graph summarization algorithm, called DistLSH. DistLSH leverages a technique called Locality Sensitive Hashing (LSH) [4] to quickly find out pairs of super-nodes which are likely to be good merges. However, as we will show later in this section, LSH cannot directly applied to our problem. Instead, we invent a novel approach, called Striped-MinHash.

LSH Background: We first provide some brief background information on LSH. LSH is a method to probabilistically reduce dimensions of high-dimension data. Using LSH, similar data items in the high-dimension space are hashed into the same buckets with high probabilities. In this study, we adopt one type of LSH, called MinHash [2]. MinHash is used to quickly estimate the similarity between two sets. For a set A and a hash function $h(\cdot)$ that maps each element a_i in A to an integer number, the MinHash of A is defined as the element a_i of A with the minimum hash value of $h(\cdot)$. More precisely, $h_{\min}(A) = a_i$ s.t. $\forall a_j \in A, h(a_i) \leq h(a_j)$. A very nice property of MinHash is that the probability of hash collision of two sets A and B is exactly the Jaccard similarity [2] of the two sets: $\Pr(h_{\min}(A) = h_{\min}(B)) = |A \cap B|/|A \cup B|$.

Before understanding the intuition behind our Striped-LSH approach, we first introduce a concept, called *connection weight*. Formally, the *connection weight* between two super-nodes $\mathcal{V}_i$ and $\mathcal{V}_j$ is defined as $w_{i,j} = |A_{i,j}|/|\Pi_{i,j}|$. With $w_{i,j}$, we can rewrite the definition of $e^*_{i,j}$ (ref. Eq. 2) as

$$e^*_{i,j} = \begin{cases} |A_{i,j}| & \text{if } w_{i,j} < 0.5 \\ |\Pi_{i,j}| - |A_{i,j}| & \text{otherwise} \end{cases} \tag{5}$$

The DistLSH algorithm only examines super-node pairs that are highly likely to be good merges that bring in minimal error increases. We observe that a good merge should satisfy the following two criteria: 1) the two super-nodes share a lot of common neighbors, 2) they have similar weights to each neighbor. The first criterion is easy to understand, in the following we explain why the second is important. Let the neighbor sets of $\mathcal{V}_i$ and $\mathcal{V}_j$ be N_i and N_j, respectively. For every neighbor $\mathcal{V}_p \in N_i \cup N_j$, we look at $w_{i,p}$ and $w_{j,p}$. In the case when $\mathcal{V}_p$ is only a unique neighbor to one of the super-nodes, say $\mathcal{V}_i$, then $w_{j,p} = 0$. When the weights of the two connections are very similar, $w_{i,p} \approx w_{j,p}$, then after merging $\mathcal{V}_i$ and $\mathcal{V}_j$ into $\mathcal{V}_m$, $w_{m,p} = \frac{|A_{i,p}|+|A_{j,p}|}{|\Pi_{i,p}|+|\Pi_{j,p}|} \approx w_{i,p}$. When $w_{i,p}$ and $w_{j,p}$ are both less than 0.5, $e^*_{i,p} = |A_{i,p}|$, $e^*_{j,p} = |A_{j,p}|$ and $e^*_{m,p} = |A_{i,p}| + |A_{j,p}|$. As a result, the error increase w.r.t neighbor $\mathcal{V}_p$ is 0, because $\Delta^p_{i,j} = e^*_{m,p} - e^*_{i,p} - e^*_{j,p} = 0$. Similarly, when $w_{i,p}$ and $w_{j,p}$ are both greater than 0.5, the error increase is also 0. On the other hand, if $w_{i,p}$ and $w_{j,p}$ are very different, e.g. $w_{i,p} \ll 0.5$ and $w_{j,p} \gg 0.5$, then $w_{m,p}$ will be somewhere between $w_{i,p}$ and $w_{j,p}$. If $w_{m,p} < 0.5$, $\Delta^p_{i,j} = 2|A_{j,p}| - |\Pi_{j,p}|$, otherwise, $\Delta^p_{i,j} = |\Pi_{i,p}| - 2|A_{i,p}|$. In both cases, $\Delta^p_{i,j} > 0$. Therefore, if for every neighbor in $N_i \cup N_j$, the connection weights of $\mathcal{V}_i$ and $\mathcal{V}_j$ are similar, the total error increase for merging the two super-nodes will be minimal, making them a perfect merge candidate.

We have found out what good merges look like, the next question is how to efficiently pinpoint such good merges. If we apply the MinHash approach to the neighbor set of each super-node, we can quickly find out the super-node pairs with similar neighbor sets. But this is not enough, as the important connection weights are not considered at all in MinHash. In order to address this limitation of MinHash, we propose a Striped-MinHash approach.

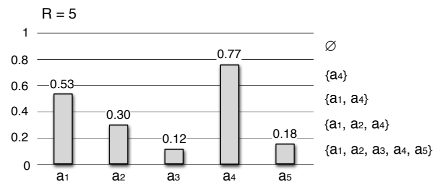
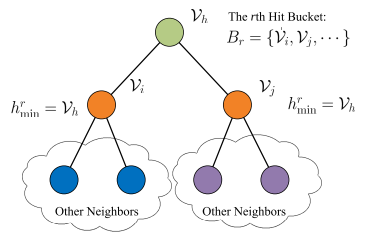


Figure 4: Striped-MinHash Example.

As shown in Figure 4, Striped-MinHash takes a weighted set $A = \{a_1 : w_1, ..., a_i : w_i, ...\}$ as input, divides the weight range $[0, 1]$ into R stripes, and applies a different MinHash function to each stripe. In stripe r, the r-th MinHash function $h^r_{\min}$ is applied to a derived set $A^r = \{a_i | w_i > \frac{r-1}{R}\}$. Given two weighted sets A and B, if at strip r, they are hashed to the same bucket, then we say that A and B are a hash hit at strip r, denoted as $hit^r(A, B) = 1$. Otherwise, $hit^r(A, B) = 0$. The total number of hits for A and B is defined as $hit(A, B) = \sum_{r=1}^{R} hit^r(A, B)$. It is clearly to see that the more similar these two weighted sets are, the more hits they are likely to get using Striped-MinHash.

We directly apply the Striped-MinHash approach in our distributed Dist-LSH algorithm to find out super-node pairs that have similar *weighted* neighbor sets (each neighbor is associated with the connection weight), a.k.a good merge candidates. We use a parameter *HitsTeshold* to control the merge candidates: only super-node pairs with number of hits more than *HitsTeshold* will be merged. Initially *HitsTeshold*$=$ R, whenever the number of merge candidates falls below 5% of the current summary size (so that more than 95% of the current graph can not be compressed), the algorithm decreases *HitsTeshold* by 1 for the subsequent iterations.



$\mathcal{V}_h$
The rth Hit Bucket:
$B_r = \{\mathcal{V}_i, \mathcal{V}_j, \cdots\}$
$\mathcal{V}_i$
$\mathcal{V}_j$
$h^r_{\min} = \mathcal{V}_h$
$h^r_{\min} = \mathcal{V}_h$
Other Neighbors
Other Neighbors

Figure 5: Not only is $\mathcal{V}_h$ the result of MinHash, but also it is the local coordinator of hash collisions in Giraph.

Applying Striped-MinHash and deciding on the merge candidates is carried out in a fully distributed fashion in three supersteps as shown in Algorithm 3. Again, we use the aggregator *ExecutionPhase* to indicate what operations each superstep should perform.

In the first phase (Hash Phase), each Giraph vertex performs Striped-MinHash on its super-node $\mathcal{V}_i$'s weighted neighbor set. For each strip r, the result of the MinHash $h^r_{\min}$ is a super-node $\mathcal{V}_h$ from the neighbor set. Note that MinHash is used here in a very smart way. As shown in Figure 5, MinHash doesn't only tell us the hash value, but also points us to a valid super-node in the current summary graph where the collection of hash collisions can be executed. $\mathcal{V}_i$ will then send a message to notify $\mathcal{V}_h$ that the r-th MinHash is hashed to $\mathcal{V}_h$.

In the second phase (Collision Phase), a Giraph vertex $\mathcal{V}_h$ will receive a number of messages indicating who is hashed to $\mathcal{V}_h$ in which strip. For each strip r, $\mathcal{V}_h$ collects all the super-nodes hashed to itself. Note that every pair of super-nodes in this collection is

Algorithm 3: FINDCANDIDATES() for Dist-LSH

```
FINDCANDIDATES(MessageIterator msgs)
    phase ← GETAGGREGATORVALUE("ExecutionPhase");
    switch phase do
        case Hash Phase          // Executed in vertex $\mathcal{V}_i$.
            for $r \leftarrow 1, 2, \cdots, R$ do
                Construct hash function $h_r(\cdot)$;
                Candidate neighbor set
                $C_u \leftarrow \{\mathcal{V}_h | \mathcal{V}_h \in \mathcal{N}_i \wedge w_{p,i} > (r-1)/R\}$;
                $\mathcal{V}_h \leftarrow \arg\min(h_r(\mathcal{V}_j))$;
                // Notify $\mathcal{V}_h$ that it is the $r$th striped
                   minhash result of $\mathcal{V}_i$.
                SENDMESSAGE($\mathcal{V}_h, \langle r, \mathcal{V}_i \rangle$);
        case Collision Phase   // Executed in vertex $\mathcal{V}_h$.
            for $\langle r, \mathcal{V}_i \rangle \in$ msgs do  // Hash result message
            with hash function id and source node.
                // Put the node $\mathcal{V}_i$ into $r$th
                   collision bucket.
                The $r$th collision bucket $B_r \leftarrow B_r \cup \{\mathcal{V}_i\}$;
            for $r \leftarrow 1, 2, \cdots, R$ do
                for $\forall \mathcal{V}_i \in B_r \wedge \mathcal{V}_j \in B_r \wedge x < y$ do
                    // Collect the partial hit
                       count among $\mathcal{V}_i$ and $\mathcal{V}_j$
                    Partial Hit count $C_{i,j} \leftarrow C_{i,j} + 1$
            for $\forall \langle \mathcal{V}_i, \mathcal{V}_j \rangle$ s.t. $C_{i,j} > 0$ do
                // Notify $\mathcal{V}_i$ that it hits $C_{i,j}$ times
                   with $\mathcal{V}_j$
                SENDMESSAGE($\mathcal{V}_i, \langle \mathcal{V}_j, C_{i,j} \rangle$);
        case Aggregate Phase   // Executed in vertex $\mathcal{V}_i$.
            for $\langle \mathcal{V}_j, C_i, j \rangle \in$ msgs do          // Collision
            Message contain the node that I
            collided in Collision phase.
                Collision counter $C_j \leftarrow C_j + C_{i,j}$;
            for $\mathcal{V}_j$ s.t. $C_j > MergeThreshold$ do
                // Notify the other merge candidate $\mathcal{V}_j$
                   to merge to me.
                SENDMESSAGE($\mathcal{V}_j, \langle \mathcal{V}_j \rightarrow \mathcal{V}_i \rangle$);
```

actually a hash collision. Suppose, in both strip r and r', $\mathcal{V}_i$ and $\mathcal{V}_j$ are hashed to $\mathcal{V}_h$. Then, $\mathcal{V}_h$ can perform a local aggregation to count the partial number of hits between $\mathcal{V}_i$ and $\mathcal{V}_j$. After that, $\mathcal{V}_h$ sends this partial count to the super-node with a smaller id among the pair, $\mathcal{V}_i$.

In the third phase (Aggregation Phase), super-node $\mathcal{V}_i$ receives partial counts of hits for itself with other super-nodes, and compute the total numbers of hits. Then for each potential partner, it tests whether the number of hits is above the $HitsThreshold$.

We now analyze the complexity of the Candidates-Find step of DistLSH. In Hash Phase, there are R striped hashes and each requires iterating through the neighbor set, resulting in $O(R \cdot d)$ time. In Collision Phase, since each super-node sends R messages in previous phase, each super-node also receives R messages on average and puts them into R buckets. In the following, we estimate that the expected number of collision pairs is $(R-1)N/2$.

THEOREM 5.1. *The expected pairs of merge candidates in each FindCandidates step of DistLSH is* $(R-1)N/2$.

PROOF. We assume that super-nodes fall into R buckets with uniform distribution. Therefore, the number of nodes in one bucket (denoted as t) follows the binomial distribution $\Pr(t) \sim \mathcal{B}(n = R, p = 1/R)$. A bucket having t nodes will generate $t \cdot (t-1)/2$ pairs of merge candidates, the expected number of total pairs for each bucket is $\sum_{t=0}^{R} \Pr(t) \cdot t \cdot (t-1)/2 = \mathbf{E}(t(t-1)/2)$. As each $\mathbf{E}(t(t-1)/2) = 1/2(\mathbf{E}((t-1)^2) + \mathbf{E}(t) - 1) = 1/2(Var(t) + \mathbf{E}(t) - 1) < (1-1/R)/2 = (R-1)/2R$ and we have R buckets in total, so there are $(R-1)/2$ total pairs of merge candidate. For all vertices, there are $(R-1)N/2$ pairs of merge candidates. □

Based on Theorem 5.1, the time complexity of Collision Phase is $O(R/2 \cdot N)$. Finally, in the Aggregate Phase each node will receive $R/2$ hit messages on average. The overall time complexity of DistLSH is $O((dR + R) \cdot N)$ in each Candidates-Find step.

6. MERGING SUPER-NODES

After the Candidates-Find task, we obtain a set of super-node pairs to be merged. In this section, we provide details on how to merge these super-nodes distributedly. For every vertex merge, instead of creating a new merged super-node, we always reuse the super-node with the smaller id as the merged super-node. Specifically, the super-node with larger id shall set its *owner-id* to the merged super-node, and call VOTETOHALT() to turn itself to inactive.

As discussed in the previous section, we know that the decision on whether to merge super-nodes $\mathcal{V}_i$ and $\mathcal{V}_j$ is always made at the super-node with the smaller id, $\mathcal{V}_i$. Therefore, at the beginning of the Merge task, $\mathcal{V}_i$ already knows which other super-node(s) will be merged into it. However, the tricky issue is that there could be another merge decision that requires $\mathcal{V}_i$ merged into $\mathcal{V}_g$. In this case, $\mathcal{V}_j$ should be eventually merged into $\mathcal{V}_g$.To efficiently merge multiple super-node pairs distributedly, we introduce a repeatable merge decision propagation phase to ensure all the super-nodes know whom they eventually should be merge into. This design decision is essential to save overall supersteps and messages, since vertex id is much cheaper to propagate than real vertex data.

In *Decision Propagation Phase*, we add a *Logical And* aggregator called *PropagationDone* to detect whether there are un-propagated merge decisions. Before each superstep, this aggregator is initialized to TRUE. During the execution of the superstep, if any vertex thinks propagation should be continued, it will aggregate FALSE to *PropagationDone*, which will result in FALSE at the end of the superstep. If *PropagationDone* is FALSE, the next superstep will continue the merge decision propagation until all the super-nodes know their eventual merge destination.

Taking the above merge case as an example. In the first superstep, the super-node with the smaller id in a merge candidate will send the merge decision to the other super-node. In our case, $\mathcal{V}_i$ will notify $\mathcal{V}_j$ and $\mathcal{V}_g$ will notify $\mathcal{V}_i$. In the second superstep, $\mathcal{V}_j$ receives a message from $\mathcal{V}_i$ and changes its *owner-id* to $\mathcal{V}_i$. Similarly, super-node $\mathcal{V}_i$ receives a message from $\mathcal{V}_g$ and changes its *owner-id* to $\mathcal{V}_g$. However, $\mathcal{V}_i$ remembers that it has sent a merge decision to $\mathcal{V}_j$, so it needs to propagate the new owner $\mathcal{V}_g$ to $\mathcal{V}_j$. Because of this, $\mathcal{V}_i$ aggregates FALSE to *PropagationDone* and the propagation continues in the next superstep. In the third superstep, $\mathcal{V}_j$ finally receives the new merge destination $\mathcal{V}_g$ and updates its *owner-id* to $\mathcal{V}_g$. At the end of this superstep, every super-node agrees on the fact that all the merge decisions are propagated and we are ready to actually merge the super-nodes.

Recall that in the vertex data structure, each super-node keeps its neighbor's basic information (*nbr.size*, *nbr.conn*). In *Connection Switch Phase*, each super-nodes to be merged shall notify its neighbors to update this neighbor information. For example, $\mathcal{V}_i$ sends out a connection switch message to every neighbor to notify them to alter connection destination from $\mathcal{V}_i$ to $\mathcal{V}_g$. This step ensures that when $\mathcal{V}_i$ turns itself to inactive, its neighbor's connections are not corrupted.

Algorithm 4: MERGE()

```
nextPhase()
    phase ← GetAggregatorValue("MergePhases");
    if phase = Decision Propagation Phase then
        if GetAggregatorValue("PropagationDone") then
            currentPhase ← nextPhase;
        else
            currentPhase ← Decision Propagation Phase;
// Show execution flow of merge candidate $\mathcal{V}_i$ and $\mathcal{V}_j$.
Merge(MessageIterator msgs)
    phase ← GetAggregatorValue("MergePhases");
    switch phase do
        case Decision Propagation Phase // Executed in super-node $\mathcal{V}_i$.
            propogation-done ← TRUE;
            for $\langle \mathcal{V}_j \to \mathcal{V}_g \rangle \in$ msgs do    // Merge Decision Message
                // Message: from_id → to_id.
                if $g <$ owner-id then
                    owner-id ← $g$;
                    propogation-done ← FALSE;
            Aggregate("PropagationDone", propogation-done);
            if NOT propogation-done then
                for $\forall \mathcal{V}_j \in PS_i$ do    // PS$_i$ contains all nodes that will merge to me, see Algorithm 2 & 3.
                    SendMessage($\mathcal{V}_j, \langle \mathcal{V}_j \to \mathcal{V}_i \rangle$);
        case Connection Switch Phase
            if owner-id ≠ self-id then
                // Asking neighbor to update its edge list.
                SendMessageToAllNeighbors(⟨self-id → owner-id⟩);
        case Connection Merge Phase
            for $\langle \mathcal{V}_p \to \mathcal{V}_q \rangle \in$ msgs do    // Edge update messages
                Change edge id from $V_p$ to $V_q$;
            // Send all my info to owner node and stop.
            if owner-id ≠ self-id then
                Pack node values and all edges to info-msg;
                SendMessage(ower-id, info-msg);
                VoteToHalt();
        case State Update Phase
            for info-msg ∈ msgs do
                Merge node values and edges to myself.
            if self-size is modified then
                // Notify all my neighbors to update my new size.
                SendMessageToAllNeighbors (⟨self-id, self-size⟩);
```

In *Connection Merge Phase*, receivers of the connection switch messages shall update their neighbor list with the new neighbor ids. In addition, each super-node that needs to be merged will also send its self-data to merge owner, including *self.size*, *self.conn*, all neighbor's *nbr.size*s and *nbr.conn*s.

Finally, in the *State Update Phase*, each merge owner receives the information from all the super-nodes that needs to be merged into it, and performs the actual merge by updating all the attributes in the local data structure. At the end of this phase, it also sends out a message to every neighbor with its new *nbr.size* information.

Dataset Name	# Nodes	# Edges
Enron Email Network	$36,692$	$367,662$
Gowalla Social Network	$565,642$	$2,431,625$
Skitter Internet Trace Graph	$1,696,415$	$11,095,298$

Table 1: Dataset Description

Optimization for Hub Nodes. We found out that many real graph data sets exist a small number hub nodes. These hub nodes often connect to a large number of low degree nodes. In many cases, these nodes only connect to the hub nodes and have degree of one. These hub nodes create computation imbalance problems for both DistGreedy and DistLSH algorithms. In DistGreedy, a hub node results in a huge number of neighbor pairs to be examined for merge candidacy; whereas in DistLSH, the large number of one-degree neighbors of a hub node will create a large amount of hash collisions gathered at the hub node. To solve the problem, it is fairly easy to see that all the 1-degree nodes of the hub node can be safely merged into one super-node without introducing any error increase. In fact, for all of our distributed algorithms, we first introduce 3 additional supersteps to merge all the 1-degree neighbors of hub nodes, then execute the corresponding algorithms. This optimization for hub nodes significantly reduces the computation and network overhead. In all of our experiments in Section 7, the algorithms tested include this special optimization.

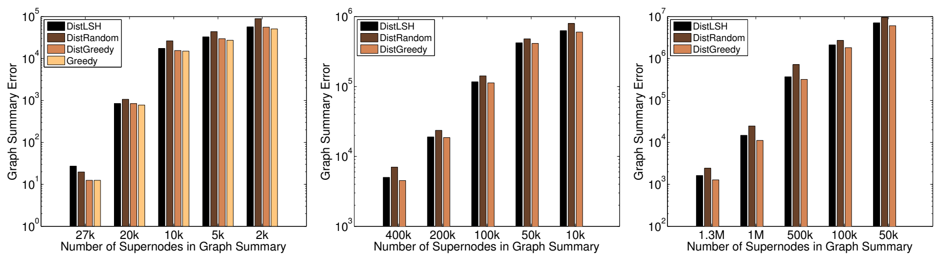
7. EXPERIMENTAL EVALUATION

We now conduct experiments with various distributed graph summarization implementations on a cluster of 16-node IBM SystemX iDataPlex dx340. Each server consisted of two quad-core Intel Xeon E5540 64-bit 2.8GHz processors, 32GB RAM, and interconnected using 1GB Ethernet. Each server ran Ubuntu Linux (kernel version 2.6.32-24), Java 1.6, and Giraph trunk version downloaded in June 2012. Each server was configured to run up to 6 workers concurrently. The following configuration parameters were set in order to boost performance: Giraph check-pointing is turned off, -Dgiraph.useNetty=true is turned on in Giraph to use Netty for message passing, and a maximum of 3GB JVM heap space was used per worker. All experiments were repeated 3 or more times. We report the average of those measurements.

We designed the experiments with the following objectives: First, we want to find out whether our distributed algorithms strike a good balance between effectiveness and efficiency. Second, we want to study the effect of various parameter settings in our algorithms. Third, we want to evaluate the scalability of our algorithms, with increasing graph sizes, decreasing summary sizes, and different number of workers.

We used both real datasets and synthetic datasets in our experiments. The three real datasets used are: the Enron email network, the As-Skitter Internet trace graph (both from `snap.stanford.edu/data`), and the Gowalla location based social network [6]. Table 1 summarizes their basic statistics. We also use the R-MAT model [3] in the GTgraph suites [1] to generate synthetic graphs with power-law degree distributions and small-world characteristics. We set the average node degree in each synthetic graph to 5, and used the default values for the other parameters in the generator. With synthetic graphs, we can freely examine the scalability of our distributed techniques across controlled graph sizes.

In all our experiments, we measured the quality of a summary graph using the errors introduced by summarization (see Section 2), and the efficiency of a graph summarization method using execution time. The execution time includes the time for special handling



(a) Enron Dataset (b) Gowalla Dataset (c) As-skitter Dataset

Figure 6: Log-scaled graph summary error histograms across different graph summary sizes for three real datasets.

of hub nodes, but excludes the time for the post processing step described in Section 4.

7.1 Balance between Effectiveness and Efficiency

We first compare the summary qualities produced by the three distributed algorithms as well as their performance. As there is no prior art on distributed graph summarization, we use the summary qualities of the centralized Greedy algorithm as a standard to evaluate the effectiveness of the distributed algorithms. However, as the centralize algorithm cannot handle large scaled graphs, we are only able to apply it to the smallest Enron dataset.

Figure 6 and 7 collectively demonstrate the graph summary qualities and running time of the three distributed algorithm on the three real datasets, with different summary sizes. For DistLSH, the number of stripes (R) in Striped-MinHash is set to be 50.

First of all, as shown in Figure 6(a), DistGreedy consistently produces summaries with very comparable qualities (less than 9% difference) to the centralized greedy algorithm. As a result, for large datasets, we can use DistGreedy as the standard for quality comparison. In addition, DistLSH only introduces less than 20% more errors than Greedy (ref. Figure 6(a)). Compared to the centralized Greedy algorithm, the additional errors incurred by DistGreedy are due to multiple merges in each iteration, and the extra errors in DistLSH are caused by both the multiple merges as well as the candidates finding heuristic.

As Figure 6 shows, with the largest summary sizes ($70\% \sim 80\%$ of original graph size), all graphs can be summarized with small number of errors as it is easy to find many vertices with similar adjacent neighbors at the beginning. As summaries become more compact, qualities of summaries degrade. One may wonder why DistLSH performs slightly worse than other algorithms on the Enron dataset when the summary size is $27K$, around 73% of the original graph. This is because the graph is quite small. The error generated by the centralized Greedy algorithm at this stage is only 10. Therefore, in this extreme case, several sub-optimal node merges in DistLSH algorithm may deteriorate summary quality. But when further summarizing the graph into more compact summaries, the DistLSH's accidental sub-optimal merges won't affect the overall summary quality much.

Figure 6 indicates that DistLSH consistently produces only slightly larger (no more than 30%) summary errors than DistGreedy, and Figure 7 shows that DistLSH is $2 \sim 11$ times faster than DistGreedy under all settings. DistLSH strikes a good balance between effectiveness and efficiency in graph summarization: 1) it consistently produces summaries with comparable qualities to DistGreedy or even centralized Greedy; 2) DistLSH is much faster than the other distributed algorithms.

Among the distributed algorithms, DistRandom is neither effective nor efficient in most experimental settings. DistRandom only appears to be more efficient than the other two in the case of large summary sizes ($70\% \sim 80\%$ of original graph size). This is because at the beginning of summarization, there exist a large number of good potential merges. Even with random selection, the chance of getting a good merge is still high. However, as the summary becomes more and more compact, DistRandom results in a lot of bad choices not satisfying the merge criteria. As a result, DistRandom usually requires a lot more iterations to produce a summary with desired size.

7.2 Number of Stripes on DistLSH

In DistLSH, the number of stripes R in Striped-MinHash is a parameter directly affecting the effectiveness and efficiency. When R is 1, DistLSH degrades to an algorithm that finds the merge candidates simply based on MinHash, at the price of losing all connection weight information. However, although a larger R can capture the weight information better, it requires more computation. Figure 8 illustrates how a larger R reduces the graph summarization errors. It seems that 50 is a good knee point that can reduce the error significantly without much sacrifice in performance. In the interest of space, we do not display the execution time of DistLSH with different R, as the running time follows a pattern of linear increase with R.

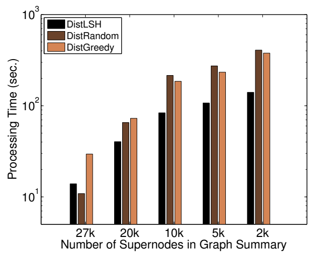
7.3 Scalability of Our Distributed Techniques

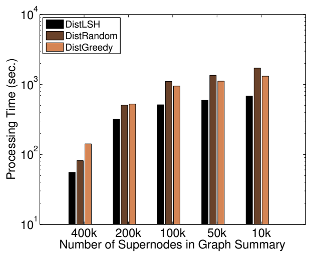
We test the scalability of our distributed algorithms on a number of synthetic graphs. Figure 9 shows the execution time of distributed algorithms with increasing graph sizes from 1 million to 100 million nodes, using a fixed number of 80 workers. The target graph summary size is set as 0.1% of the original graph. Here the increase from around 600 seconds to roughly 6,000 seconds in DistLSH demonstrates its sub-linear runtime increase with graph sizes. Unfortunately, the largest graph sizes that DistGreedy and DistRandom can process in a reasonable amount of time (less than 2 hours) are 4 millions and 1 millions, respectively. With these relatively small graph datasets, we can still see that DistLSH is much faster than DistGreedy and DistRandom.

To show how our distributed algorithms scale with the number of workers, Figure 10 presents the runtime for a synthetic graph with 1 million vertices, when the number of Giraph workers varies from 20 to 80. Here the drop from $2,174$ seconds to 636 seconds using 4 times as many as workers represents a speedup of about 3.4. Due to the synchronization cost between supersteps, the ideal speedup 4 is impossible to achieve. 3.4 is already a very decent speedup ratio.

8. RELATED WORK

Graph summarization is a relatively new concept in graph analysis. It was proposed independently by [12] and [9] in 2008. Despite the differences in these two works, both employ a same fundamental summary model, in which a summary itself is a graph, more compact yet informative. The nodes in the summary graph repre-


(a) Enron Dataset


(b) Gowalla Dataset

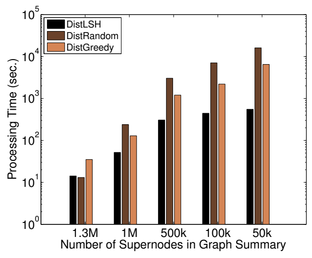
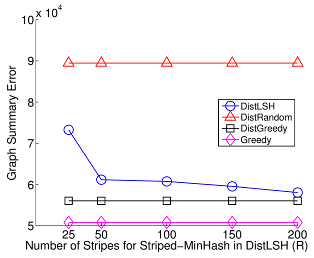
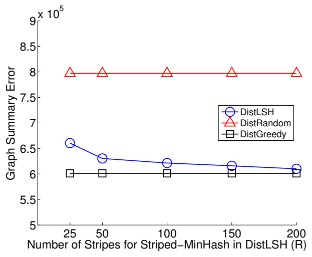

(c) As-skitter Dataset

Figure 7: Log-scaled running time histograms across different graph summary sizes for three real datasets.


(a) Enron Dataset


(b) Gowalla Dataset

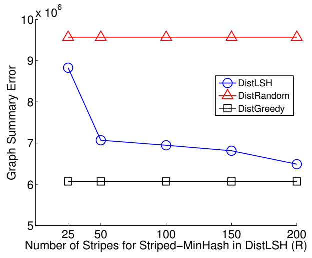

(c) As-skitter Dataset

Figure 8: The effect of minhash layer number on DistLSH. The number of super-nodes for Enron, As-skitter and Gowalla are 2K, 10K and 50K respectively.

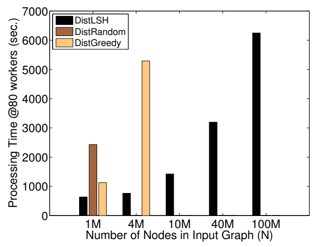


Figure 9: Scale with input size.

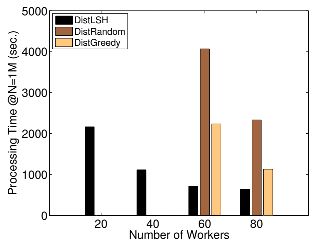


Figure 10: Scale with cluster size.

sent groups of nodes from the original graph, and edges in the summary describe the relationships between groups of nodes. In [9], besides the summary graph, a set of edge corrections is also maintained, so that the original graph can be reconstructed from summary graph by applying the edge corrections. Based on Rissanen's Minimum Description Length (MDL) principle [10], the authors in [9] formulated the graph compression problem into an optimization problem, which minimizes the sum of the size of the summary graph and the size of the edge correction set. The graph summarization model in [12] incorporates attributes associated with nodes and different types of edges besides the normal graph structure. As a result, the graph summarization method in [12] is able to let users select node attributes and edge types of interests and produce summaries with desired resolutions. A subsequent work [13] addressed two limitations of [12], namely the issues of a large number of attributes being selected and attributes with large value ranges. However, none of the existing works addresses the issue of large scale graph summarization in a distributed environment.

9. CONCLUSION

This paper introduces three distributed algorithms for large-scale graph summarization on the Giraph distributed computing framework. Among them, the DistLSH algorithm applies the novel Striped-MinHash technique to linearly pinpoint the set of possible merge candidates, and achieves a nice balance between effectiveness and efficiency. Scalability testing on large synthetic graphs indicates that DistLSH scales nicely with graph sizes and cluster sizes. As DistLSH has been shown a practical algorithm for large-scale graph summarization on real email/Internet/social networks data, in the future work, we plan to mine the graph patterns from the summaries produced by DistLSH, and study the correlation between graph patterns and summary sizes.

10. REFERENCES

[1] D. A. Bader and K. Madduri. Gtgraph: A suite of synthetic graph generators. www.cse.psu.edu/ madduri/software/GTgraph.

[2] A. Z. Broder, M. Charikar, A. M. Frieze, and M. Mitzenmacher. Min-wise independent permutations (extended abstract). In *STOC '98*, pages 327–336.

[3] D. Chakrabarti, Y. Zhan, and C. Faloutsos. R-mat: A recursive model for graph mining. In *SDM'04*.

[4] P. Indyk and R. Motwani. Approximate nearest neighbors: towards removing the curse of dimensionality. In *STOC'98*, pages 604–613.

[5] J. Lin and C. Dyer. *Data-Intensive Text Processing with MapReduce*, volume 3. 2010.

[6] X. Liu, Q. He, Y. Tian, W. Lee, J. McPherson, and J. Han. Event-based social networks: Linking the online and offline social worlds. In *SIGKDD'12*.

[7] Y. Low, D. Bickson, J. Gonzalez, C. Guestrin, A. Kyrola, and J. M. Hellerstein. Distributed GraphLab: a framework for machine learning and data mining in the cloud. *PVLDB*, 5(8):716–727, 2012.

[8] G. Malewicz, M. H. Austern, A. J. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski. Pregel: a system for large-scale graph processing. In *SIGMOD'10*, pages 135–146.

[9] S. Navlakha, R. Rastogi, and N. Shrivastava. Graph summarization with bounded error. In *SIGMOD'08*, pages 567–580.

[10] J. Rissanen. Modeling by shortest data description. *Automatica*, 14:465–471, 1978.

[11] B. Shao, H. Wang, and Y. Li. Trinity: A Distributed Graph Engine on a Memory Cloud,. In *SIGMOD'13*, 2013.

[12] Y. Tian, R. Hankins, and J. M. Patel. Efficient aggregation for graph summarization. In *SIGMOD'08*, pages 419–432.

[13] N. Zhang, Y. Tian, and J. M. Patel. Discovery-driven graph summarization. In *ICDE'10*, pages 880–891.