Advance Access publication on 6 January 2016 doi:10.1093/comjnl/bxv118

# Frequent Subgraph Mining Based on Pregel

Xiang Zhao[1,2]*, Yifan Chen[1], Chuan Xiao[3], Yoshiharu Ishikawa[3] and Jiuyang Tang[1,2]

[1]*College of Information System and Management, National University of Defense Technology, Changsha, Hunan, China*
[2]*Collaborative Innovation Center of Geospatial Technology, Wuhan, Hubei, China*
[3]*Graduate School of Information Science, Nagoya University, Furo-cho, Chikusa-ku, Nagoya, Japan*
**Corresponding author: xiangzhao@nudt.edu.cn*

**Graph is an increasingly popular way to model complex data, and the size of single graphs is growing toward massive. Nonetheless, executing graph algorithms efficiently and at scale is surprisingly challenging. As a consequence, distributed programming frameworks have emerged to empower large graph processing.** Pregel**, as a popular computational model for processing billion-vertex graphs, has been employed to improve the scalability of many algorithms. In this paper, we investigate frequent subgraph mining on single large graphs using** Pregel**. We present the first distributed algorithm based on** Pregel **for single massive graphs. In addition, two optimizations are proposed to enhance the algorithm, reducing communication cost and distribution overhead. Extensive experiments conducted on real-life data confirm the effectiveness and efficiency of the proposed algorithm and techniques.**

*Keywords: frequent subgraph mining; single massive graphs; Pregel*

*Received 20 May 2015; revised 26 October 2015*
Handling editor: Daniel Paulusma

## 1. INTRODUCTION

Graph data model is an increasingly popular way to represent data in various application fields, including social networks, bioinformatics, web graphs, etc. Recent decades have witnessed a rapid proliferation of sizes and volumes of graph-structured data. A variety of fundamental problems have been investigated on graphs, including subgraph search and matching [1, 2], structure similarity queries [3, 4], distance and reachability queries [5, 6], etc.

Frequent pattern mining has been a focused theme in data mining for over a decade. Abundant literature was dedicated to this area, making tremendous progress, including frequent itemset mining, sequential pattern mining and so forth. Frequent subgraphs are subgraphs found from a *collection of graphs* or *single large graph* with support no less than a user-specified threshold. Frequent subgraphs are useful at characterizing graph datasets, classifying and clustering graphs, and building structural indices [7]. We differentiate the two aforementioned scenarios—multi-graph and single-graph, and this paper focuses on frequent subgraph mining (FSM) on the *single-graph* setting.

Nowadays, graphs (networks) are growing toward massive. Take Facebook as an example. The number of active users of Facebook increased to one billion in late 2012, which was merely less than 9 years after its founding. By modeling users as vertices, friendships as edges, we have an overwhelmingly large graph of billion vertices; other typical massive networks are observed in the forms of phone call networks, protein interaction networks, world wide web, etc. Given the *rapid growth* of applications of FSM in various disciplines, as well as the *sheer size* of real-life graphs, an efficient method for distributed FSM at scale is of high demand.

Distributed FSM from single massive graphs is challenging, due to not only the *special constraints* of FSM algorithm design, but also the *deficient support* from existing distributed programming frameworks. First, an FSM algorithm computes the support of a candidate subgraph over the entire input graph. In a distributed platform, if the input graph is partitioned over various worker nodes, the local support of the subgraph is not much useful for deciding whether subgraph is globally frequent. Also, the support computation cannot be delayed arbitrarily, since candidate frequent subgraphs can be generated only

from frequent subgraphs as per Apriori principle. Additionally, although there are several existing models, including MapReduce [8], the de facto big data processing framework, they also do not accommodate graph algorithms [9]. Among the distributed graph processing frameworks, Pregel [10] is recognized for its scalability, flexibility, fault tolerance and a number of other attractive features. It is a vertex-centric programming model such that developers usually only need to submit processing scripts on vertices to the framework, which will handle the remaining issues such as graph partition and synchronization. However, it is suggested that structure-related computation may not fit Pregel naturally [11]. Therefore, mapping a structure mining algorithm onto Pregel requires non-trivial efforts, since Pregel does not specify implementation details for self-defined functions.

In this paper, we focus on efficient implementation of FSM over single massive graphs on a Pregel-like extensible computing platform. To the best of our knowledge, this is among the first attempts to address the problem at scale under a modern distributed programming framework.

In summary, we make the following contribution:

(i) We propose a systematic solution for FSM in single massive graphs using Pregel-like distributed programming paradigm.
(ii) We devise two optimization techniques to enhance the baseline algorithm, reducing communication cost and distribution overhead, respectively.
(iii) We evaluate the resulting algorithm pegi with extensive experiments on public real-life data. The experiment results confirm the efficiency and scalability of the proposed methods.

*Organization.* Sections 2 and 3 discuss related work and preliminaries, respectively. Section 4 presents the baseline algorithm, followed by optimizations in Section 5. Section 6 describes our experiments, and we conclude the paper in Section 7.

## 2. RELATED WORK

While FSM was extensively studied, how to overcome the rapid growth of graph data remains open. Following discusses related work in three directions— FSM on multi-graph setting, FSM on single-graph setting and distributed graph processing.

*Mining Graph Collections.* Apriori was utilized to mine frequent subgraphs on transaction settings by AGM [12]. AGM generates candidate graphs by adding a vertex at a time; as an improvement, FSG [13] puts forward edge-growth mining. Both methods adopt the breadth-first search, i.e. first compute size-$k$ frequent subgraphs, based on which size-$(k+1)$ frequent subgraphs are computed thereafter.

Distinctively, recent approaches follow depth-first search, with gSpan [14] as a representative. gSpan relies on a novel canonical graph labeling to assist with search space pruning. Later, another graph representation was employed to reduce the overhead of subgraph isomorphism tests [15]. Recently, GASTON [16] proposed to categorize graphs into paths, trees and cyclic graphs, and developed accelerative techniques, respectively. A follow-up work [17] provides more insight into the categorization of graphs for speedup.

Compared with the aforementioned in-memory algorithms, ADI-Mine [18] is a disk-based algorithm leveraging a three-level ADI-index. To enhance with parallel computing, SUBDUE [19] describes a shared-memory parallel approach by partitioning tasks. Recently, MapReduce was employed, in which pattern size grows in each round of a MapReduce job. The state-of-the-art MapReduce-based solution is attributed to a two-step filter-and-refinement method [20], which incorporates techniques to predict candidate patterns and reduce inter-machine communication cost. Analogous work that adapts single-machine sequential algorithms into the MapReduce platform include [21, 22]. Note that their object differs from ours in that they focus on a large collection of graphs, rather than from a single large network. Therefore, in a similar flavor, we contend that recompiling a single-machine FSM algorithm on single graphs into a distributed fashion also worth dedicated effort.

*Mining Single Graphs.* As an equally important problem, this line of research focuses on mining single large graphs but with less work, to which our work belongs. Efforts were first dedicated to defining appropriate support measures, e.g. MI, HO and MIS. SIGRAM uses MIS [23], and follows a *grow-and-store* approach—it needs to store the intermediate results for support evaluation. A parallel version over a multi-core machine was also implemented to enhance its efficiency [24]. To avoid the computational complexity of MIS, HO [25] and MI [26] were proposed.

The most recent work is attributed to GraMi [27], which formulates the FSM problem as a constraint satisfaction problem. It finds only the minimal set of instances to satisfy the support threshold, and hence, improves performance. While we are not aware of any published distributed solution for the identical problem, this paper describes the *first* triumph, which outperforms GraMi in terms of both efficiency and scalability.

Among others, we are also aware of several *approximate* FSM algorithms, e.g. Grew [28] and gApprox [29].

*Distributed Graph Processing.* Distributed graph processing framework is inevitable in handling massive graphs. MapReduce [8] is a universal tool for processing large volume of data, including graphs [30]. Hence, it has been used to compute personalized PageRank [31], connected components [32], etc. Lately, a high-level query language GLog [33] was introduced on MapReduce for graph analysis.

Since MapReduce is considered not a perfect candidate for processing graphs, other programming frameworks came in response, including Pregel [10] and its variants [34], GraphLab [9], Trinity [35], Mizan [36], etc. Based on Pregel,

algorithmic optimizations are investigated [37]. Based on Trinity, a number of applications were studied for online performance, including distance queries and subgraph matching [38, 39], etc.

There is work looking into advanced partition management [40, 41], structural querying [11, 42] and handling dynamic graphs [43] in distributed frameworks.

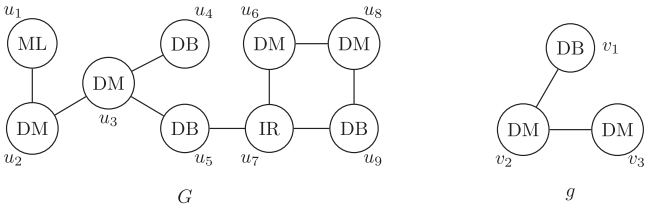


**FIGURE 1.** Example graph and subgraph.

## 3. PRELIMINARIES

Section 3.1 introduces the concepts related to FSM on single graphs, and Section 3.2 is a brief Pregel primer.

### 3.1. Frequent subgraph mining

For ease of exposition, we focus on *simple* graphs, i.e. undirected graphs with neither self-loops nor multiple edges. A labeled graph $G$ is represented in a triple $(V_G, E_G, l_G)$, where $V_G$ is a set of vertices, $E_G \subseteq V_G \times V_G$ is a set of edges, and $l_G : V_G \cup E_G \to \mathcal{L}$ is a labeling function that assigns labels to vertices and edges. $|V_G|$ and $|E_G|$ are the number of vertices and edges in $G$, respectively. $l_G(v)$ denotes the label of vertex $v \in V_G$. $l_G(u, v)$ denotes the label of edge $e \in E_G$, where $e = (u, v)$.

DEFINITION 3.1 (Subgraph isomorphism). *A graph $g$ is subgraph isomorphic to another graph $G$, denoted by $g \sqsubseteq G$, if there exists an injection $f : V_g \to V_G$ such that* (1) $\forall v \in V_g$, $f(v) \in V_G \wedge l_g(v) = l_G(f(v))$ *and* (2) $\forall (u, v) \in E_g$, $(f(u), f(v)) \in E_G \wedge l_g(u, v) = l_G(f(u), f(v))$. *$g$ is also called a subgraph of $G$, and $f(g)$ is an embedding of $g$ in $G$.*

Consider two graphs $g \sqsubseteq G$, and a *minimum support threshold* $\tau$; assume there is a function $\phi$ to measure the *support* of $g$ in $G$. If $\phi(g) \geq \tau$, $g$ is a *frequent subgraph* of data graph $G$. There are several ways to measure the support of a subgraph $g$ in a single graph $G$, and the most intuitive way is to count the isomorphisms of $g$ in $G$.

EXAMPLE 1. Consider the collaboration network $G$ in Fig. 1, with authors represented by vertices and co-authorship by edges, and the label in the vertex indicates the community that the author belongs to. Given a subgraph $g$, there are three subgraph isomorphisms from $g$ to $G$, i.e. $v_1 - v_2 - v_3$ to $u_4 - u_3 - u_2$, $u_5 - u_3 - u_2$ and $u_9 - u_8 - u_6$, respectively. Further, consider a minimum support threshold $\tau = 3$, and $\phi$ is defined as Definition 3.1. $\phi(g) = 3 \geq \tau$, and thus, $g$ is a frequent subgraph of $G$.

Unfortunately, the aforementioned metric is not *anti-monotonic* [23, 25, 26], since a subgraph may appear less times than its extensions. For instance, consider in Fig. 1 vertex `IR` and its extension `IR-DB` on $G$. It is easy to verify that the support of the former is 1 while the support of its extension is 2. Anti-monotonicity is crucial to developing algorithms that can effectively prune the search space, without which they have to carry out exhaustive search. As a consequence, existing literature presents several anti-monotonic support metrics based on (1) minimum image (MI) [26], (2) harmful overlap (HO) [25] and (3) maximum independent sets (MIS) [23]. These measures are all established on subgraph isomorphisms, but differ in the extent of *compatible* overlap among them, and hence, the computational complexity. In particular, MI is the only metric that can be computed efficiently, while HO and MIS involve solving NP-complete problems; the result set of MI is always a *superset* of those of HO and MIS, and therefore, the desired results can be further derived from the results of MI with additional computation. Therefore, we adopt MI as the support measure in the sequel, whereas the algorithms are readily to be extended to others with minor effort.

DEFINITION 3.2 (Minimum image-based support). *Consider a set of distinct subgraph isomorphisms $F = \{f_i\}$ from $g$ to $G$, where $i \in [1, |F|]$. Let $F(v)$ denote the set of distinct vertices $u \in V_G$ such that there exists an isomorphism $f_i$ mapping $v \in V_g$ to $u$. The minimum image-based support of $g$ in $G$ is defined as $\phi(g) = \min\{|F(v)|\}$, $v \in V_g$.*

Further to the definition, $F(v)$ are the *images* of $v \in V_g$ in $G$ with respect to $g$, and hence, the *conditional support* of $v$ with respect to $g$, denoted by $\phi_g(v) = |F(v)|$, which may be less than $|F|$. We shorten 'minimum image-based support' to 'support' onwards; to distinguish, 'images' always refers to single vertices, while 'embeddings' can be vertices if $g$ is a vertex, or subgraphs if $g$ has at least two vertices.

EXAMPLE 2. Consider the graphs in Fig. 1, and threshold $\tau = 3$. $\phi(v_1) = 3 \geq \tau$, and hence, `DB` is a frequent single vertex. Recall the three subgraph isomorphisms $F$ from $g$ to $G$ in Example 1. The images of $v_1$ with respect to $g$, i.e. $F(v_1)$ are $\{u_4, u_5, u_9\}$ and $F(v_2) = \{u_3, u_8\}$, $F(v_3) = \{u_2, u_6\}$. Thus, the conditional support of $v_1$, $v_2$ and $v_3$ with respect to $g$ are 3, 2 and 2, respectively. In other terms, $\phi_g(v1) = 3$, $\phi_g(v_2) = 2$, $\phi_g(v_3) = 2$, and hence, $\phi(g) = \min\{3, 2, 2\} = 2 < \tau$. Therefore, $g$ is not a frequent subgraph in $G$.

Formally, the problem of *FSM in a single graph* considers a data graph $G$ and a minimum support threshold $\tau$, and finds

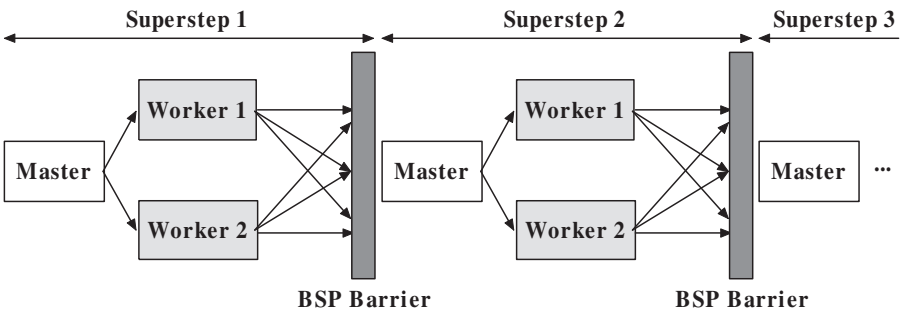


**FIGURE 2.** Master/workers computation model.

all subgraphs $g$ in $G$ such that $\phi(g) \geq \tau$. This paper concerns solving the problem exactly and at scale, and the algorithm works for both connected and disconnected $G$. Additionally, several existing work proposes to mine maximal patterns, we argue that these desired subgraphs can be derived from our answers with further computation. Afterwards, we will focus on producing all frequent subgraphs.

### 3.2. Pregel overview

The iterative graph processing architecture Pregel [10] is based on the *bulk synchronous parallel* model of distributed computation. Pregel uses a master/workers model—one instance acts as the *master*, while the others become *workers*. The basic computation model of Pregel is shown in Fig. 2, illustrated with three supersteps on a master and two workers.

Given a single large graph, the vertices, identified by an ID, are firstly distributed by a *partitioner* across workers running on different computing nodes. The default partitioner is a hash function on vertex IDs. Computation is achieved by iterations, namely *supersteps*. The master performs serial computation and coordination between supersteps, and all workers conduct parallel computation and synchronize at the end of supersteps.

All algorithms in Pregel are implemented in a vertex-centric fashion. Specifically, every vertex has a vertex value, a set of edges and a set of messages sent to it in the previous superstep. In other terms, in superstep $i$, the vertex can receive the messages sent by other vertices in superstep $i-1$, query and update the information of the current vertex and its edges, initiate topology mutation, communicate with global aggregation, and send messages to other vertices for superstep $i+1$. After all vertices finish their computation, a global synchronization allows global data to be aggregated, and messages to be delivered.

Giraph[1] originated as the open-source counterpart to Pregel. To implement a graph algorithm, users instantiate methods master.compute() and vertex.compute() of the master and vertex classes, respectively. To enable master/vertex to perform multiple functions, compute() is executed based on a switch of multiple cases per superstep, corresponding to different functions of master/vertex. Hence, the interleave of these cases between the master and vertices working together accomplishes a task.

*Aggregators* are a mechanism for global communication, and data transmission. Each vertex can provide a value to an aggregator in superstep $i$, the system combines those values and the result is available to all vertices in superstep $i+1$. It is possible to define a *sticky* aggregator for input values from all supersteps, which may hold a value, an array or even a map. We rely on these functions for global coordination. In particular, two methods are to be instantiated,

(i) Aggregate($\alpha, x$), which enables the master/current vertex to send a value $x$ to the aggregator named $\alpha$ and
(ii) GetAggregatedValue($\beta$), which enables the master/current vertex to retrieve the data stored in the aggregator named $\beta$.

## 4. THE MINING ALGORITHM

This section introduces the algorithm pegi for FSM in Pregel. We first present a single-machine sequential algorithm for FSM on single graphs, and then map it onto the Pregel model resulting pegi. Its compute methods on the master and vertex are detailed thereafter.

### 4.1. Baseline algorithm

An FSM algorithm has to traverse all possible subgraphs of the data graph. By carefully organizing the subgraphs into a *candidate generation tree*, a commonly adopted approach is to conduct a depth-first search on the tree. A tree node represents a subgraph, or *pattern*, and the parent–child relation depicts the growth of a pattern. In particular, a subgraph is extended to one of its children by attaching every time a new edge. The extended subgraph is included as an answer if it is frequent and has not been discovered previously. This generation process ensures that the unambiguously defined candidate generation tree comprises all patterns. Additionally, anti-monotone

[1] https://giraph.apache.org/

pruning is utilized to shrink the search space, i.e. any extension of an infrequent graph cannot be frequent.

We differentiate two types of edges that can be used to extend a subgraph $p$: (1) *forward edge*, if it introduces a new vertex, namely *target vertex*, to $p$ and (2) *backward edge*, if it is added between two vertices of $p$, which does not exist before. While they both extend $p$, backward edges do not affect the vertex set of $p$.

**Algorithm 1:** Baseline($G, \tau$)

```
   Input  : G is a graph; τ is a support threshold.
   Output: P is a set of frequent subgraphs, initialized to ∅.
1  P ← P^1 ← find frequent single edges in G;
2  foreach edge e ∈ P^1 do DFSMine (G, e) ;

   Function DFSMine(G, p)
3      enumerate 1-edge extension of p and embeddings;
4      foreach enumerated edge e for p do
5          p' ← p ∪ {e};
6          if e is a forward edge with target vertex v
           then
7              if φ_{p'}(v) < τ then continue;
8          if φ(p') < τ ∨ p' ∈ P then continue;
9          P ← P ∪ {p'} ;        /* find an answer */
10         DFSMine (G, p');
```

The pseudo-code in Algorithm 1 implements our baseline FSM algorithm on single graphs. Algorithm 1 takes as input a graph $G$ and a support threshold $\tau$, and outputs the complete set of frequent subgraphs. It first collects the set of frequent single edges in $G$ (Line 1), which are essentially the size-1 frequent subgraphs. Then, for each frequent subgraph found, it carries out an iterative DFS mining via function DFSMine (Line 2). Specifically, DFSMine first enumerates the 1-edge extension of the current subgraph $p$ (Line 4), which are $p$'s children in the candidate generation tree, as well as their occurrences. For each enumerated edge $e$, we construct a extended subgraph $p'$ (Line 5). If $e$ is a forward edge with target vertex $v$, we first compute the conditional support of $v$ with respect to $p'$. If it is below the support threshold, it will never contribute a new frequent subgraph, and we continue to examine other edges (Lines 6–7). Then, we evaluate the support of $p'$ on the vertices of original subgraph $p$. If $\phi(p')$ is not less than the threshold by anti-monotone pruning, we find an answer, as long as $p'$ is not seen in $P$ (Lines 8–9). Afterwards, it puts $p'$ into another round of DFSMine (Line 10). The algorithm terminates when no more candidate subgraphs can be generated.

**Algorithm 2:** master. compute()

```
1  switch phase do
2      case VERTEX:
3          V_f ← GetAggregatedValue(frq_v);
4          V_f ← get frequent vertices with images;
5          V_t ← retrieve images of first vertex in V_f;
6          Aggregate(nxt_v,V_t);
7      case GROW:
8          e ← GrowPattern(E_c, Φ);
9          Aggregate(nxt_e,e);
10     case UPDATE:
11         V_t ← GetAggregatedValue(nxt_v);
12         update global embedding tree by adding edges
           incident on V_t;
```

**Algorithm 3:** vertex. compute()

```
1  switch phase do
2      case VERTEX: Aggregate(frq_v, this.l_v) ;
3      case EXTEND: ExploreEdge(V_t) ;
4      case SUPPORT:
5          foreach distinct message m do
6              Aggregate(m.e, 1);
7      case TARGET:
8          if this is backtrack then update local
           embedding tree by removing last updated
           edges;
9          e ← GetAggregatedValue(nxt_e);
10         Aggregate(nxt_v, V_t(e));
```

### 4.2. Distributed paradigm of pegi

An important observation from the baseline algorithm is that it tests whether an edge can be used to extend the current frequent subgraph, and then proceeds if the edge meets the support threshold. On the distributed setting, as the data graph is distributed to the workers, the local support of an edge lower than the threshold does not necessary lead to the failure globally. Another observation is the 1-edge extension is enumerated on the basis of the occurrences of current subgraph; that is, when the algorithm generates the candidates for the subsequent round, it requires the embeddings of current subgraph. Thus, the most intuitive way to implement this is storing the embeddings and tracking the changes. As a consequence, it is non-trivial to adapt the algorithm developed for single machine to run under Pregel, which involves complex design of computation and interaction between the master and workers. We address the challenges by proposing pegi (Pregel-based frequent subgraph mining).

Given a massive graph, pegi first distributes the graph partitions according to available workers. We do not leverage advanced graph partitioner in this work, and apply the default random partitioner. Then, it iteratively conducts two types of jobs, i.e. *pattern growth* and *embedding discovery*, on the master and workers, respectively. In other terms, the step-control of pattern space traversal is carefully handled by the master node, and for each step of pattern growth it requires updates of newly discovered embeddings, which is carried out on the distributed workers.

To achieve the aforementioned functions, we conceive three cases for the master, and four cases for the vertices, as abstracted in Algorithms 2 and 3, respectively. We list the functional cases on the master and vertex in Tables 1 and 2, respectively; Table 3 summarizes the aggregators involved in the algorithms. Through interaction among the cases above, the baseline pegi executes and flows as depicted in Fig. 3.

**TABLE 1.** Cases of master.compute().

| Case | Function |
|---|---|
| VERTEX | Compute frequent single vertices |
| EXTEND | Determine next edge to grow current subgraph |
| UPDATE | Update global embeddings |

**TABLE 2.** Cases of vertex.compute().

| Case | Function |
|---|---|
| VERTEX | Send vertex label for aggregation |
| EXTEND | Explore to find candidate edges |
| SUPPORT | Compute conditional support of target vertices |
| TARGET | Send target vertices for updating embeddings |

**TABLE 3.** Aggregators.

| Name | Content |
|---|---|
| `frq_v` | Frequent single vertices and images |
| `nxt_v` | Target images of newly extended vertex |
| `sup_v` | Conditional support of target vertices |
| `nxt_e` | Next growing edge |
| `cnd_e` | Candidate edges |

(i) In the first superstep, the master skips its compute method, and each vertex runs into case VERTEX to send its vertex label for aggregation. Particularly, label along with the vertex ID is sent to aggregator `frq_v` for statistics (Line 2 of Algorithm 3), in order to produce the set of frequent single vertices.

(ii) In the second superstep, the master runs into case VERTEX, where it first derives frequent single vertices by accumulating the images of every vertex label in aggregator `frq_v`. The results are refereed by $V_f$, which is a `map` acting as a posting list, with frequent single vertices as entries and images as postings. Then, one frequent vertex is chosen as target vertex, i.e. the vertex to be extended to (Lines 3–5 of Algorithm 2). Its embeddings, namely *target images*, are distributed via aggregator `nxt_v`.

Afterwards, the vertices run into case EXTEND and execute ExploreEdge (to be detailed in Algorithm 5). It first updates the local embedding information by attaching the target images. Then, starting from these newly extended vertices, it explores its neighborhood to find candidate edges for subsequent supersteps from neighboring vertices. These edges are put in aggregator `cnd_e`. Additionally, in order to evaluate the support of target vertices for candidate forward edges, it sends a message to the neighboring vertices, i.e. potential target images. The message contains the edge that it follows, which is to be accumulated on those vertices shortly.

(iii) In the third superstep, the master idles, while the vertices run into case SUPPORT. Specifically, it reads the incoming messages, and for every distinct edge $e$, we increment its counter at the vertex. Recall that each candidate forward edge is associated with a target vertex to be extended to. Thus, the counter records in essence the local conditional support of the target vertex for a candidate forward edge. The values are then transmitted via a designated aggregator `sup_v` (Lines 5–6 of Algorithm 3), which will be used to determine the next growing edge.

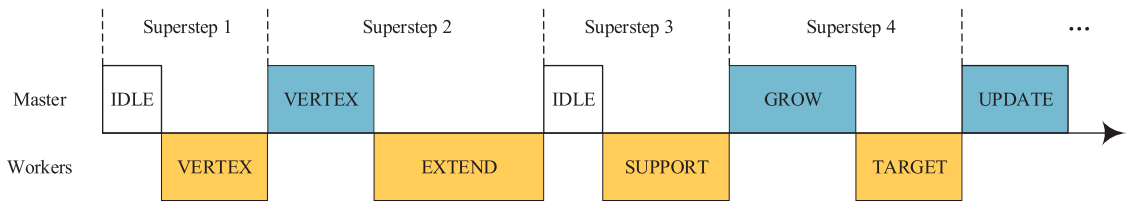


**FIGURE 3.** Execution flow of pegi.

(iv) In the fourth superstep, the master initiates case GROW executing GrowPattern (to be detailed Algorithm 4), where DFS-based pattern growth is conducted. Instead of proceeding iteratively as in Algorithm 1, we break from the procedure when an edge is chosen as the next growing edge, and set in aggregator `nxt_e`.

After that, the vertices start case TARGET, where they collect the target images following the chosen edge, and transmit them via aggregator `nxt_v` (Lines 9–10 of Algorithm 3). If backtrack is just executed in GrowPattern on the master, workers also synchronize with the master here by removing the last updated edges.

(v) In the firth superstep, the master runs into case UPDATE, where it obtains the target images from aggregator `nxt_v`, which are used to update the embedding trees for the newly extended vertices on the master. This completes the growth of the first edge, finishing one round of pattern growth. Next, the vertices start again case EXTEND, initiating another round of embedding discovery.

The mining process proceeds iteratively, and the distributed *grow-and-backtrack* terminates till no more edge extension is allowed.

**Remark.** One may note that Algorithm 1 starts from frequent edges; in contrast, we propose to find the set of frequent vertices and grow patterns from fixed vertices. This consideration is due to the excessively large amount of candidate edges that could be aggregated in the first step. They may not be accommodated by the master, and hence, easily become a bottleneck bringing down the performance. Starting from a fixed frequent vertex effectively reduces the number of first-round candidate edges, which is bounded by $O(\psi d_G)$, where $\psi$ is the average support of a frequent vertex, and $d_G$ is the average vertex degree of $G$.

Thus far, we have not explained the management of embeddings, procedures of GrowPattern on the master and ExploreEdge on the vertex. Following discusses our consideration and details the implementations.

#### 4.2.1. *On embeddings*

To facilitate edge support evaluation, we adopt the *grow-and-store* approach [23], and thus, embeddings of the current subgraph are carefully materialized in a tree structure, namely *embedding tree*. In particular, we employ DFS encoding scheme [14] to assist the candidate generation, such that each subgraph in the candidate generation tree is expressed by a corresponding DFS code. Thus, we can linearize the vertices of an embedding according to their order in the DFS code. Thus, the linearized embedding of a pattern is of the same length (or depth) as its DFS code. Using *null* as the tree root, we gradually merge two embeddings from the first vertex to the last, forming a tree structure, as long as they share the same data vertex at the identical depth.

Note that we maintain all the embeddings of the current subgraph on the master, namely the *global* embedding tree; for every worker, only the embeddings starting from vertices on the worker are kept, referred as *local* embedding trees. It is noted that there is a choice of whether or not to store embeddings of the current subgraph [23, 27]. Existing single machine solution [27] contends that storing all the embeddings may hinder the algorithm from processing large graphs for memory constraint. While we are not against it, it is believed that the case is different in a distributed system. In the latter, distributing the embeddings to its owner worker enables the system to function as a 'memory cloud', and hence, alleviates the space overhead of maintaining embeddings. Seeing the advantage of fast embedding discovery, therefore, we choose to store the embeddings of the current subgraph as intermediate results. In implantation, embedding tree is realized by using the context function of Pregel, such that all the vertices on the same worker is able to access it.

#### 4.2.2. *On master*

Among various functions performed on the master by master.compute(), as shown in Algorithm 2, case GROW does the crucial work on the master—pattern growth. We outline the major steps of case GROW in Algorithm 4, Particularly, it takes as input the set of candidate edges and the conditional support of their target vertices in aggregators `cnd_e` and `sup_v`, respectively, and produces the edge chosen to grow in aggregator `nxt_e`. Note the candidate edges $E_c$ includes both forward and backward edges, while $\Phi$ is the conditional support of target vertices for candidate forward edges. In other words, for every candidate forward edge $e$, we have a corresponding value in $\Phi$ equal the conditional support of the target vertex for $e$. It will be looked up shortly in DFSGrow to determine the next growing edge. The chosen frequent edge is distributed via aggregator `nxt_e` such that embedding discovery can be carried out on the distributed workers thereafter (Line 3).

We then proceed to explain DFSGrow in Algorithm 4, whose implementation is similar to Algorithm 1. To simulate the DFS process in a distributed fashion, we employ a stack $S$ to reserve the iteration states for backtracking. $S$ is a globally defined stack of edge sets, initialized to $\emptyset$. Algorithm 4 takes as input a set of candidate edges $E_c$, conditional support of target vertices $\Phi$ and the current subgraph $p$, and computes frequent subgraphs based on $p$ iteratively as output. Specifically, for each candidate edge $e$, we first append it to $p$, and remove it from $E_c$ to ensure the search space rooted at this subgraph will not be explored multiple times (Line 5). Then, we conduct edge support evaluation to test whether $p'$ is frequent. Specifically, if $e$ is a forward edge with target vertex $v$, we seek the conditional support of $v$ in $\Phi$. It proceeds only if $\phi_{p'}(v)$ passes the support threshold (Lines 6–8); otherwise, it continues to examine other edges. Then, we (further) evaluate the support

**Algorithm 4:** GrowPattern($E_c$, $\Phi$)

**Input** : $E_c$ in aggregator `cnd_e` is a set of edges; $\Phi$ in aggregator `sup_v` is a set of supports.
**Output**: $e$ in aggregator `nxt_e` is a chosen edge.

1 $p \leftarrow$ get current subgraph;
2 $e \leftarrow$ DFSGrow($E_c, \Phi, p$);
3 Aggregate(`nxt_e`, $e$);

**Function** DFSGrow($E_c$, $\Phi$, $p$)
4 **foreach** edge $e \in E_c$ **do**
5 $p' \leftarrow p \cup \{e\}$, $E_c \leftarrow E_c \setminus \{e\}$;
6 **if** $e$ is forward edge with target vertex $v$ **then**
7 $\phi_{p'}(v) \leftarrow$ find support of $v$ in $\Phi$;
8 **if** $\phi_{p'}(v) < \tau$ **then** continue;
9 $\phi(p') \leftarrow$ evaluate support on vertices of $p$;
10 **if** $\phi(p') < \tau \wedge p' \in P$ **then** continue;
11 $P \leftarrow P \cup \{p'\}$, push $E_c$ into stack $S$;
12 **return** $e$
13 **if** $S$ is empty **then return** *null* ;
14 **else**
15 update global embeddings tree by removing embeddings of $e$;
16 $E_c \leftarrow$ pop the top set of edges from stack $S$;
17 **return** DFSGrow($E_c$, $\Phi$, $p$)

of $p'$ on the vertices of $p$ leveraging the global embedding tree. If $\phi(p')$ exceeds the support threshold, and $p'$ is not discovered previously, we find an answer, and then push the remaining edges in $E_c$ as a set into stack $S$ for backtrack (Lines 9–12). At last, we break from the procedure by returning the next growing edge.

After screening all edges in $E_c$ such that no more edge is chosen as the next growing edge, we start backtracking. We first check if stack $S$ is empty to decide whether to halt (Line 13), as empty $S$ implies the finish of mining under one frequent vertex. If not, we go back to the precedent node in the candidate generation tree (Lines 15–17). Specifically, we remove $e$ from the current subgraph, discard the last updated edges from the embedding tree, and pop the top edge set out of $S$. The mining procedure is then called again. Iteratively in this way, we conduct a complete traverse of the candidate generation tree, examining all possible subgraphs.

#### *4.2.3. On vertex*

Embedding discovery is carried out on the workers in a distributed vertex-centric fashion, and the major step is to explore for candidate edges. Hence, the core function ExploreEdge of case EXTEND in vertex.compute() is presented in Algorithm 5, which finds local candidate edges starting from the newly extended vertex.

Algorithm 5 takes as input the newly extended vertices $V_t$ in aggregator `nxt_v`, and outputs candidate edges $E_c$ for the

**Algorithm 5:** ExploreEdge($V_t$)

**Input** : $V_t$ in aggregator `nxt_v` is a set of vertices.
**Output**: $E_c$ in aggregator `cnd_e` is a set of edges.

1 update local embedding tree by adding edges incident on $V_t$;
2 $E_c \leftarrow$ explore edges starting from this vertex;
3 **foreach** candidate forward edge $e \in E_c$ **do**
4 $V_t \leftarrow$ recall target images via $e$;
5 **foreach** vertex $v \in V_t$ **do** SendMessage($e$, $v$) ;
6 Aggregate(`cnd_e`, $E_c$);

subsequent rounds in aggregator `cnd_e`. In particular, as a new edge is attached to the current subgraph, we instantiate it on the local embeddings incident on $V_t$ (Line 1). Next, we collect the candidate edges starting from this vertex in $E_c$ (Line 2), which is to be sent to the master via aggregator `cnd_e` (Line 6). For each candidate forward edge $e$, the corresponding target images are retrieved in $V_t$; additionally, we send the candidate forward edge as a message to each of the target images (Lines 3–5), which will be later accumulated for the conditional support of target vertices.

### 4.3. Illustration and analysis

So far, we have presented the complete algorithm of pegi. Putting them together, we illustrate one round of pattern growth and embedding discovery in Example 3.

EXAMPLE 3. Consider in Fig. 4 data graph $G$ and current subgraph $p$ (black) with candidate edges (red in color version, or gray in black and white version), and assume $\tau = 3$. Five example embeddings of $p$ in $G$ are listed below, and hence, it is easy to verify that $p$ is a frequent subgraph. To grow patterns based on $p$, we first find candidate edges in case EXTEND of vertex compute method. In particular, we detail the computation on $u_4$. Every neighboring edge of $u_4$ is examined with respect to the embeddings of $p$. For instance, by checking edge $(u_4, u_5)$, four candidate forward edges can be discovered, which may be used to grow $p$. Subsequently, each candidate forward edge is sent to the corresponding potential target images. In the next superstep, target images receive the messages containing the candidate edges; for each received *distinct* candidate edge, a target image contributes 1 to the local conditional support of the target vertex. Then, the master obtains the candidate edges in case GROW. For instance, consider $(u_4, u_5)$ w.r.t. $f_1$, and the new subgraph is denoted by $p'$. $u_0$,$u_2$ and $u_5$ receive messages containing candidate forward edge (`DM`, `ML`), $\phi_{p'}(v_5)$ is evaluated as 3.

Recall in Section 3.1 that we employ MI as the support metric. Hence, we further check the global embedding tree, and verify the conditional support of the other vertices also meets the threshold $\tau = 3$ (cf. Section 3.1 and Example 2). That is,

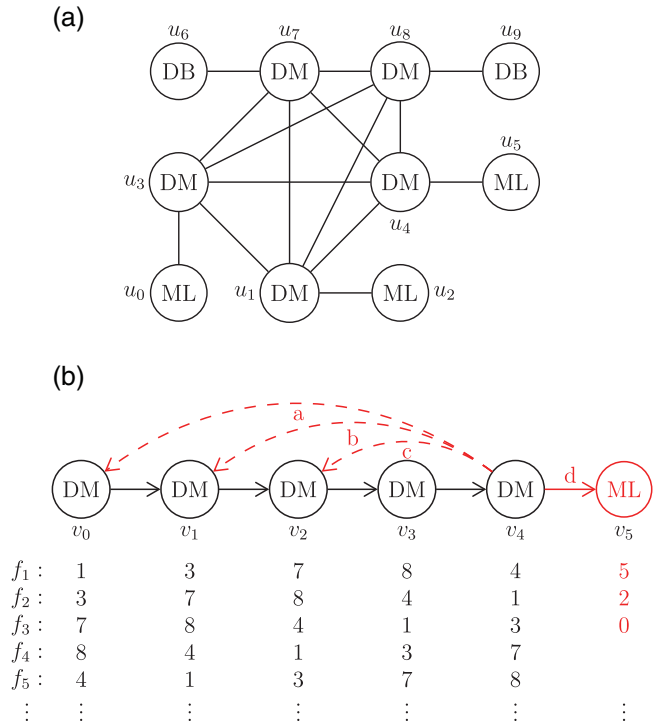


**FIGURE 4.** Example of pattern growth. (a) Data Graph $G$, (b) Subgraph $p$.

$\forall i \in \{0,1,2,3,4,5\}, \phi_{p'}(v_i) \geq 3$; thus, $\phi(p') \geq 3$, and $p'$ is determined to be frequent based on MI. Afterwards, if a candidate forward edge is selected, one more superstep is required to aggregate the target images, e.g. $u_0$,$u_2$ and $u_5$ for $p'$, for updating the local and global embedding trees.

*Analysis.* The correctness of the algorithm is guaranteed by the completeness of the search procedure and the correctness of the support evaluation. It is easy to verify that the proposed algorithm computes the set of frequent subgraphs in a given massive graph, without missing or redundant results.

Space cost, communication cost and number of supersteps are major concerns for investigating a Pregel-based algorithm [34]. We first study memory consumption. The major memory usage is from the storage of embeddings. Consider a pattern $p$, and assume the extreme case that all the vertices and edges of the data graph possess an identical vertex label. The total number of embeddings is $O(d_p^{(|V_p|-1)}|V_G|)$, where $d_G$ is the average vertex degree of $G$. Hence, the maximum memory required on each worker is in the same order of magnitude.

As a consequence, the memory consumption is heavily related to two factors, namely label distribution and density of graphs. The aforementioned worst case only occurs when the assumption is realized. We will see in Section 6 that real-life graphs usually incur much less memory footprint with general label distributions.

Then, we analyze the communication cost. In particular, we are interested in the total number of messages passing in the system, since this requires costly communication among the workers over the network. Recall in Algorithm 5 that for each candidate forward edge, the newly extended vertices send messages to neighboring vertices. Therefore, in one round of pattern growth, the total number of messages is $O(d_G|V_G||E_G|)$.

Next, we investigate the number of supersteps required for discovering a pattern. Intuitively, the more supersteps, the more synchronization barriers, the larger distribution overhead. As explained in the algorithm, every time one edge is grown on the pattern, and exactly two superstep is required. Recall that all patterns are discovered through a tree-structured search space. Moreover, for each leaf node in the search tree, one more superstep is required to confirm that the candidate edge set is empty and the current pattern is on a leaf. As the number of leaves equals $O(|P|)$, the total number of supersteps required is $2|P| + O(|P|) = O(|P|)$, where $P$ is the set of frequent subgraphs.

In light of the analysis above, we may improve algorithmic efficiency, if aggregated vertices and required supersteps can be reduced. In the sequel, we will devise two optimization techniques working orthogonally to achieve better overall performance.

## 5. OPTIMIZATIONS

This section introduces optimizations on top of the baseline algorithm, to reduce communication cost (Section 5.1) and synchronization overhead (Section 5.2), respectively.

### 5.1. Filtering for less message passing

Large amount of data aggregation incurs great communication cost due to network delay. Algorithm performance can be enhanced if we can reduce such cost. We address the issue below, and start with a motivating example.

EXAMPLE 4. Consider in Fig. 4 graph $G$ and current pattern $p$, and threshold $\tau = 3$. Recall in Algorithm 3 that messages are sent to $\{u_6, u_9\}$ for candidate edge (DM, DB), in order to obtain the conditional support of the target vertex. However, the candidate edge is verified shortly to be infrequent, and hence, the two messages were sent in vain.

Motivated by the example, we contend that if we can filter out these non-promising target vertices, we are able to decrease the message passing in the system, and thus, reduce communication cost. To this end, we propose to first obtain an upperbound of the support that never underestimate the real value, and use this to compare with the threshold. If the upperbound is less than the

threshold, the target vertex cannot be extended to, and the candidate forward edge needs not to be tested.

To obtain the estimation, for each candidate edge, we simply aggregate the number of neighboring vertices that match the candidate edge, and the aggregated number is an upperbound of the conditional support of the target vertex.

LEMMA 5.1. *Consider current subgraph p and target vertex u, and let $v \in V_t$ is a newly extended vertex, and $M(v)$ is v's neighboring vertices that match u.*

$$\phi_p(u) \leq \sum_v |M(v)|.$$

*Proof.* (sketch) $\bigcup_v M(v)$ includes all the target images with respect to $u$. Based on Definition 3.2, the vertices that contribute to the conditional support of $u$ are always composes a subset of $\bigcup_v M(v)$, since there may be duplicate images for $u$ among all the embeddings of $p$. Therefore, the correctness of the inequation follows. □

To implement this idea, we need to augment the compute methods of the master and vertex. In particular, an extra superstep is used for executing filtering. In Algorithm 5, after getting the candidate edges, for each candidate forward edge $e$ with target vertex $u$, we aggregate the number of target images via aggregator `est_v`. In the subsequent superstep, the master obtains the upperbound of $\phi_p(u)$ from the aggregator, and then, compares with the threshold. If the upperbound is less than the threshold, we discard the target vertex $u$; otherwise, we put it into aggregator `cnd_v`. After testing all the target vertices, the surviving ones are distributed to the workers. Then, the algorithm flows back to sending messages to target images, in order to compute the real support. Note this time messages are only sent to target images of the vertices in aggregator `cnd_v`. We omit the pseudo-code in the interest of space.

We remark that this optimization technique reduces not only the message passing, but also the data aggregation in the system, since less target vertices and images are transmitted to compute the exact conditional support. Nevertheless, it is admitted that the aforementioned benefits come at the cost of one extra superstep, in comparison with the basic pegi. Thus, the technique is particularly useful in reducing communication cost when there are more *infrequent* candidate forward edges.

### 5.2. Coupling growth of multiple edges

As there is a synchronization barrier between supersteps, a large number of supersteps increase distribution overhead. To achieve elegant responsiveness of the algorithm, the less supersteps the better. In the following, we investigate whether supersteps can be reduced. Let us first look at an example.

EXAMPLE 5. Consider graphs in Fig. 4, and four candidate edges $a, b, c, d$ (red in color version, or gray in black and white version). The search space subtree rooted at $p$ is depicted in Fig. 5. Each node represents a pattern; for example, node $+abcd$ denotes the pattern with all four edges attached to $p$. All the subgraphs in Fig. 5 are frequent, and duplicate nodes are removed. Specifically, 15 patterns are discovered in 15 steps of pattern growth in basic pegi. We observe that if backward edge $a$ can grow on $p$, $b$ and $c$ can continue to grow without breaking from DFSGrow, as backward edges do not increase but only prune the existing embeddings. Based on the pruned embeddings, the support of $b$ and $c$ can be calculated, and hence, DFSGrow can proceed. In comparison, this approach requires only eight mining steps. It ceases until we reach $d$, as $d$ is a forward edge, which may bring embeddings on extra vertices.

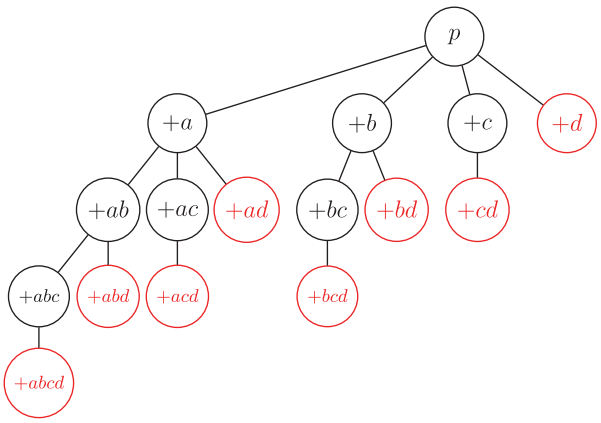


**FIGURE 5.** Candidate generation tree rooted at $p$.

Motivated by the example, we formalize the idea into a recursive version of Algorithm 4. The first difference of the recursive algorithm lies in that we check whether the current subgraph $p$ is visited, rather than remove $e$ from $E_c$ (Line 5 of Algorithm 4). Afterwards, when $e$ is verified to be frequent and $p$ is canonical, we do not break; instead, we check whether $e$ is a backward edge. If yes, it continues with a recursive calculation until the next edge is a forward edge. Upon encountering a forward edge, which is the exit of the recursion, we synchronize it to the workers, and push remaining $E_c$ into the stack for backtrack. This completes the batch processing of backward edges of the current round, and then, we carry on the iterative pattern growth.

## 6. EXPERIMENTS

This section presents our experiment results and analysis.

### 6.1. Experiment set-up

We used Giraph for experiments. All experiments were conducted using Java JRE v1.6.0 on Amazon EC2. By default,

**TABLE 4.** Data statistics.

| Dataset | $\|V\|$ | $\|E\|$ | $\|l_V\|$ | $\gamma$ | Size (GB) |
|---|---|---|---|---|---|
| TT | 11 316 811 | 85 331 846 | 100 | 7.54 | 1.73 |
| LJ | 4 847 571 | 68 993 773 | 30 | 14.23 | 2.26 |
| UP | 3 774 768 | 16 522 438 | 418 | 4.38 | 0.45 |

21 instances were used, one of which was designated as the NameNode. The standard configuration of the instance for NameNode was `m1.medium`, with one CPU and 3.75 GB memory. The remaining instances were `m3.xlarge`, with four CPU and 15 GB memory. To ensure adequate memory for every worker, the number of workers is set to 20, i.e. one instance for each worker.

We experimented on different workload settings over the following real-life datasets:

(i) ***Twitter* (TT)**[2]: This graph models the social news of Twitter, which consists of 11 316 811 vertices and 85 331 846 edges. A vertex represents a Twitter user, and an edge represents an interaction between the two users connected by the edge.
(ii) ***LiveJournal* (LJ)**[3]: LiveJournal is a free online community with almost 10 million members; a significant fraction of these members are highly active. LiveJournal allows members to maintain journals, individual and group blogs, and it allows people to declare which other members are their friends and to which communities they belong.
(iii) ***US Patents* (UP)**[4]: This graph represents the reference relations between US patents. This graph contains 3 774 768 nodes and 16 522 438 edges. We used the property class as the label collection, and hence, it has 418 labels in total.

Among the three datasets, UP possesses labels; for TT and LJ, we randomly added labels to the vertices and edges. That is, 100 distinct labels were used for vertices and six for edges on TT, and 30 for vertices and one for edges on LJ. To better mimic the label distribution on real-life networks, the randomization followed the Gaussian distribution. Table 4 lists the statistics of the datasets, where $\gamma = |E|/|V|$. Through Table 4, we can see that the three datasets are of different characteristics, i.e. LJ is much denser than the other two, TT is larger in terms of vertex number while the vertices of UP have the most distinct labels.

The following values were measured and reported: (1) peak memory consumption per machine; (2) total data transmission; (3) number of supersteps and (4) elapsed time.

[2] http://socialcomputing.asu.edu/datasets/Twitter
[3] http://snap.stanford.edu/data/soc-LiveJournal1.html
[4] http://vlado.fmf.uni-lj.si/pub/networks/data/patents/Patents.htm

### 6.2. Evaluating proposed techniques

We implemented the baseline algorithm under *distributed* paradigm to demonstrate the effectiveness of the framework, denoted by ‘ Baseline’ (BA), constituted of Algorithms 2 and 3. On top of Baseline, we further implemented the two optimization techniques, resulting

(i) +Filter, labeled by ‘FT’, which incorporates the filtering technique in Section 5.1 to reduce communication cost;
(ii) +Backward, labeled by ‘BE’, which employs the optimization technique leveraging backward edges in Section 5.2 for reducing distribution overhead and
(iii) +All, labeled by ‘AL’, which integrates all the proposed techniques.

Through Fig. 6a–l, we observe that Baseline, +Filter and +Backward work well on the three massive graphs, and can effectively obtain the frequent subgraphs from the graphs. In particular, we first evaluate the effect of +Filter and +Backward on memory consumption, and plot the results in Fig. 6a–c. The results reveal that more memory is required to carry on +Backward than Baseline. This is justifiable for that the compute method on the master has to go through a recursive function, bringing a slight increase of memory footprint. In comparison, the requirement for memory would be much smaller if +Filter were adopted, around 2000 MB spared in maximum. Rather than aggregate all the data in one superstep, which can be excessively memory-consuming, updating embedding information separately saves the memory but at the cost of more supersteps. However, as illustrated in Fig. 6j–l, incorporating +Filter actually results in a drop in total running time.

Next, we study the effect of these techniques on communication cost, with results plot in Fig. 6d–f. Communication cost is implied by the total data transmission in TB. As +Backward does not result in any change of data transmission, we omit it in this comparison. Specifically, communication cost is divided into two categories, namely data aggregation cost and message-passing cost. Data aggregation includes the data synchronized between the mater and workers through the aggregators, while message passing refers the messages sent among vertices. In general, message-passing cost accounts for the majority, which is reflected by the bar of ‘BA’ in Fig. 8a–c. As +Filter is designed to reduce message passing, the result demonstrates its effectiveness that the total communication cost drops dramatically, though it sees a slight increase in the data aggregation cost. Note that the effectiveness of +Filter is moderate on LJ. This could be justified by the shortage of target vertices in each round due to the shortage of distinct labels. In specific, the widest gap is 0.610 TB when the threshold $\tau = 3000$ on TT, 0.405 TB on LJ with $\tau = 500$ and 0.24 TB on UP with $\tau = 600$.

We also recorded the number of supersteps, and plot the results in Fig. 6g–i. It can be revealed that +Filter actually

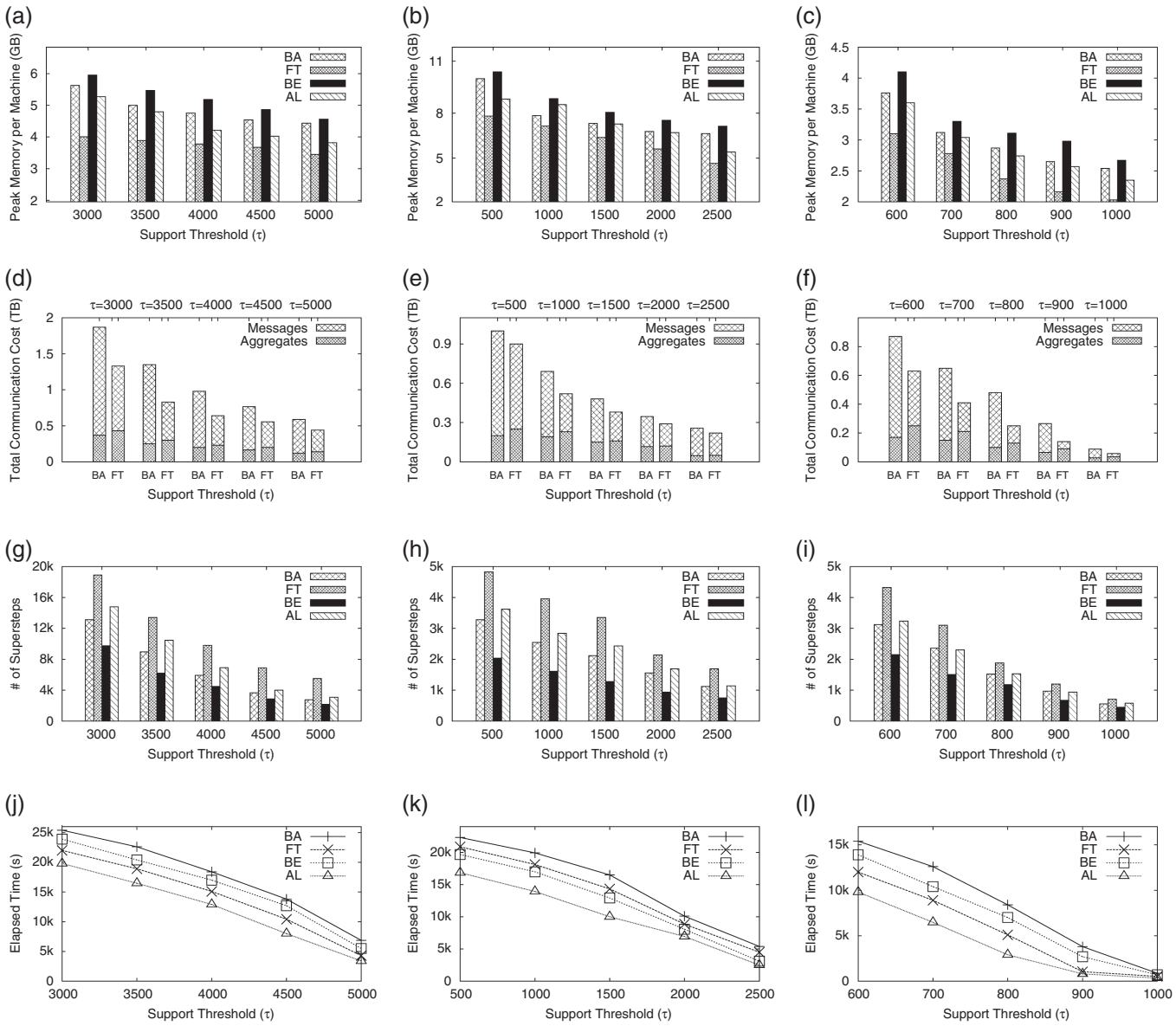


**FIGURE 6.** Experiment results—I. (a) TT: Memory consumption, (b) LJ: Memory Consumption, (c) UP: Memory Consumption, (d) TT: Communication Cost, (e) LJ: Communication Cost, (f) UP: Communication Cost, (g) TT: Supersteps, (h) LJ: Supersteps, (i) UP: Supersteps, (j) TT: Elapsed Time, (k) LJ: Elapsed Time and (l) UP: Elapsed Time.

increases the number of supersteps, while +Backward reduces it but performs differently. In nature, +Filter reduces the amount of messages at the expense of one extra superstep, and +Backward shrinks the number of supersteps when involving backward edges. Consequently, +All enlarges the number of supersteps in a moderate rate, and the gap between Baseline and +All narrows down when the threshold $\tau$ becomes large.

The overall performance is compared in Fig. 6j–l. We see that the elapsed time for all the proposed algorithms drops with the increase of $\tau$, where the proposed optimization techniques witness a remarkable improvement. It is justifiable that by introducing +Backward, the running time can be saved for less supersteps and less synchronization barriers. However, it is not immediately clear that +Filter can reserve the time due to the increases in the number of supersteps. In comparison with Baseline, +Filter filters out considerable number of non-promising candidate edges, and only small amount of edges are sent as messages, largely reducing the communication cost. As the message passing requires network communication, it can be rather expensive. Despite the increase of supersteps, the contribution for reducing communication cost is more significant. Finally, by incorporating both +Backward and

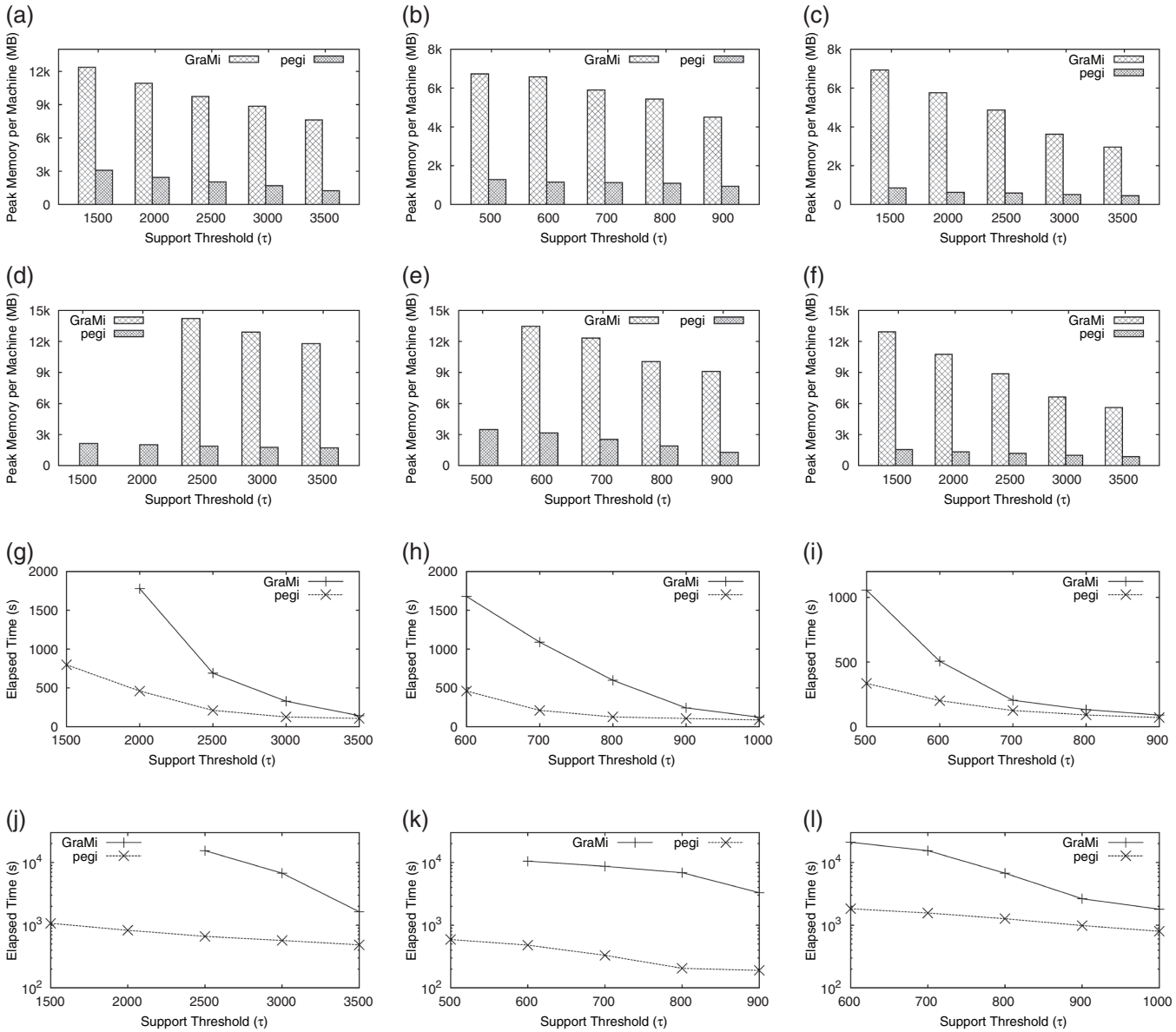


**FIGURE 7.** Experiment results—II. (a) TT(20%): Memory Consumption, (b) LJ(20%): Memory Consumption, (c) UP(20%): Memory Consumption, (d) TT(40%): Memory Consumption, (e) LJ(40%): Memory Consumption, (f) UP(40%): Memory Consumption, (g) TT(20%): Elapsed Time, (h) LJ (20%): Elapsed Time, (i) UP (20%): Elapsed Time, (j) TT (40%): Elapsed Time, (k) LJ (40%): Elapsed Time and (l) UP (40%): Elapsed Time.

+Filter, the mining algorithm is getting about 5000 s down in maximum, compared with Baseline.

Note that the decrease of running time is more noteworthy by +Filter than by +Backward on TT and UP, while the opposite is seen on LJ. It could be attributed to the density difference of the two datasets. That is, LJ is denser, which results in more backward edges in the mining process; TT and UP have more vertices, so that the effectiveness of +Filter is more significant. This is inspiring, as no matter the dataset is dense or sparse, +All works well due to the mutual complement effects of +Filter and +Backward.

In the sequel, we equipped all the proposed techniques to the baseline algorithm, i.e. pegi (+All), and further evaluated it through comparisons with existing solutions.

### 6.3. Comparing with existing algorithms

Since we were not aware of any existing distributed solution for our problem, a comparison with the state-of-the-art algorithm GraMi [27] was carried out, which is the most recently published single-machine solution. Thus, we compared them in terms of memory consumption and elapsed time. GraMi was

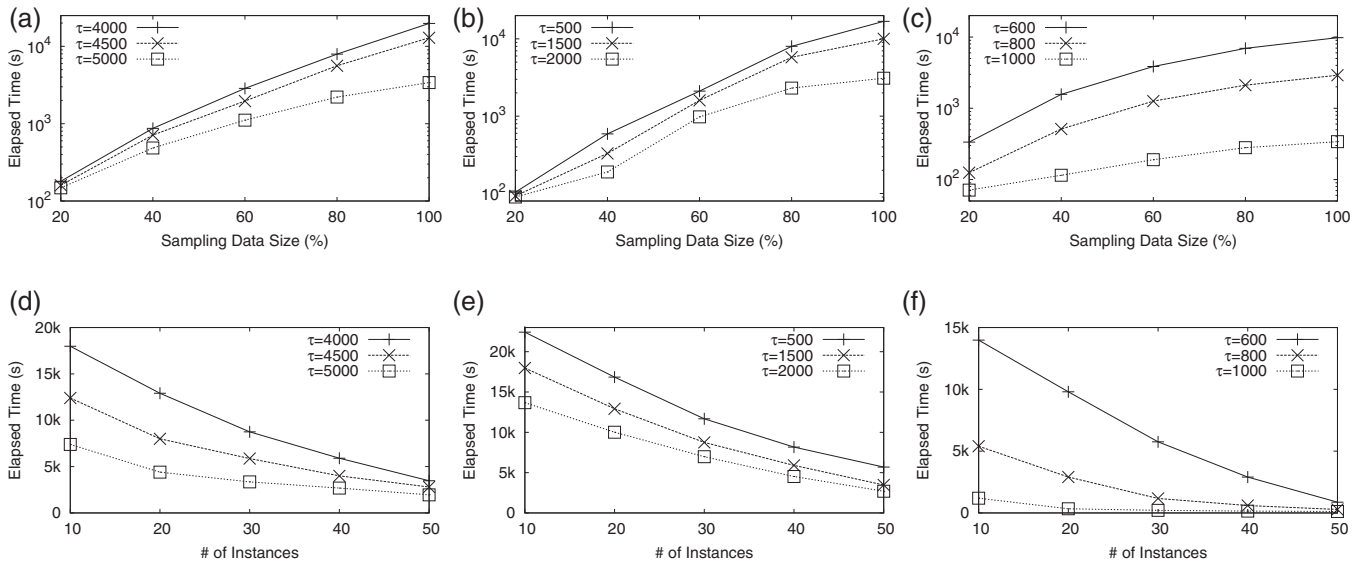


**FIGURE 8.** Experiment results—III. (a) TT: Data Scalability, (b) LJ: Data Scalability, (c) UP: Data Scalability, (d) TT: Instance Scalability, (e) LJ: Instance Scalability and (f) UP: Instance Scalability.

run on a single instance `m3.xlarge`, and experiments were carried on part of the three datasets (20% and 40% samples, respectively).

*Forest Fire* [44] was used to sample the data, which can well maintain the properties of original graphs. Note that we shrank the support threshold to ensure adequate number of frequent subgraphs in the sample graphs. We plot the results in Fig. 7a–k.

As to memory required for running the algorithms (pegi measured by the peak memory usage of single machines among all workers, and GraMi measured by the peak value of the single instance), we read in Fig. 7a and d that the privilege of pegi over GraMi is evident on the 20% sample, nearly one order of magnitude smaller. That is, $10^3$ MB for pegi, and $10^4$ MB for GraMi. It is even more significant on the 40% sample, as GraMi ran out of memory when $\tau$ is small, e.g. $\tau = 1500$ and 2000; nonetheless, pegi levels off at the $10^3$ magnitude. This superiority is achieved by storing the data in multiple instances and applying the optimization technique for reducing data aggregation. The results on LJ and UP saw similarity in Fig. 7b, e, c and f. Note that although GraMi does not maintain the embeddings, it still needs to materialize many, if not all, embeddings for support computation, as long as the embeddings overlap with identical images. Additionally, by doing this, it misses the opportunity of computation sharing, and hence, has to grow embeddings every time from scratch. Shortly, will we see how this affects the time efficiency.

As to the overall performance, we plot the results in Fig. 7g–l. Figure 7g–i show that pegi outperforms GraMi, particularly when $\tau$ is small. As GraMi did not finish in 10 h, the actual time was not recorded. The superiority is more notable on 40% sample in Fig. 7j–l, shown in logarithmic scale. The running time for pegi is almost one magnitude superior over GraMi. Either, we could not record the time for GraMi at small $\tau$'s, due to out of memory (more than 15 GB space).

In short, the comparison experiment verifies that pegi performs better in general than GraMi, especially on large data sizes and small thresholds.

### 6.4. Evaluating scalability

Lastly, we demonstrate the scalability of the proposed methods. In this set of experiments, we randomly sampled fractions of the original graphs to run the algorithms. In particular, we sampled the original graph, resulting five sampled graph with $\{20\%, 40\%, 60\%, 80\%, 100\%\}$ vertices selected. While the number of vertices grow linearly, the number of frequent subgraphs and their embeddings increase sharply. This is due to the fact that edges of the sampled graph mounting exponentially. The results of scalability against dataset size are provided in Fig. 8a–c. Figure 8a showcases a significant growth of running time against the dataset size. Although the growing trend is noteworthy, it is slower than the exponential growth, especially when $\tau = 5000$, which does not increase linearly in the logarithmic scale in the figure. Figure 8b presents a similar trend; although the lines fluctuate slightly, it does not impact the general conclusion. Figure 8c shows even better performance in scalability. As $\tau$ is fixed, the number of patterns and embeddings should rise exponentially against data size. It is reasonable to see an exponential increase in the running time. However, we witness a slowdown in growth rate on the three datasets, demonstrating that pegi scales well on the real-life data.

To further evaluate its ability to handle massive graphs, we also conducted experiments on synthetic graphs with an order of magnitude more edge than UP. We used a synthetic graph generator,[5] which naturally measures graph size in terms of $|E|$. On a synthetic graph of 100M edges with $\gamma = 5$ and the number of distinct vertex and edge labels equal 100 and 5, respectively, for instance, when the support threshold was 5000, it took 6832 s to respond.

Besides data scalability, it is of importance to see how the algorithm performs against the number of computing nodes. We conducted the experiment by increasing 10 worker instances each time, from 10 to 50. Intuitively, the runtime performance improves along with the growth of computing nodes. According to the results in Fig. 8d–f, we confirm this prediction. In general, the three lines representing different $\tau$'s drop gradually with the increase of instance number, though the time saved is less significant. Particularly, on TT, the maximum speedup is 1.55 when the number of instances increases from 10 to 20 and $\tau = 4500$; the minimum speedup is as large as 1.17 when the number of instances increases from 40 to 50 and $\tau = 5000$. On LJ when $\tau = 1500$, the speedups are 1.49, 1.28, 1.12 and 1.08, respectively, every time we added 10 more instances to the system. For UP, the maximum speedup is achieved when increasing the number of instances from 10 to 20 and $\tau = 600$, and average value is as large as 1.64 for that support threshold. In the ideal scenario, which is hard to meet in practice, the number of instances and the elapsed time are negatively linear correlated. Lines in the figure shows good scalability, though the decreasing trend gradually slows down, due to the fact that the increase in the number of instances may pose burden on the synchronization within the cluster. It is also worth noting that it performs better in scalability when the threshold is getting larger. This could be justified, as larger threshold results in shorter length/size of discovered patterns. As the possible growing paths from the current embedding are larger, and more likely to be evenly distributed in the graph data when the current subgraph is shorter, the query tasks are more evenly dispatched over all the workers, leading to better scalability.

## 7. CONCLUSION

In this paper, we have studied the problem of FSM in a distributed environment. In particular, we base our solution on a popular big graph processing framework Pregel, and present the first solution of its kind. The compute methods on the master and vertex are carefully designed, working together to achieve a collective goal. Moreover, in order to enhance the mining performance, we propose two optimization techniques to reduce message passing and number of supersteps, respectively. Comprehensive experiments on real-life data confirm the efficiency and scalability of the proposal.

[5] http://www.cse.ust.hk/graphgen/

In the future, we plan to apply the proposed algorithm on various real-life applications, e.g. anti-spam emails and network security monitoring, to obtain potential interesting patterns. Additionally, it is of interest to investigate whether Pregel can be leveraged to solve the problem of mining constrained subgraph patterns and significant subgraph patterns on massive graphs.

## FUNDING

X.Z., Y.C. and J.T. were supported by NSFC No. 61402494 and 61402498, NSF of Hunan No. 2015JJ4009. C.X. and Y.I. were supported by Kakenhi 25280039 and 26540043. Funding to pay the Open Access publication charges for this article was provided by National Natural Science Foundation of China.

## REFERENCES

[1] Zhao, P. and Han, J. (2010) On graph query optimization in large networks. *PVLDB*, **3**, 340–351.

[2] Han, W.-S., Lee, J. and Lee, J.-H. (2013) Turbo$_{\text{iso}}$: Towards Ultrafast and Robust Subgraph Isomorphism Search in Large Graph Databases. *Proc. SIGMOD 13*, New York, NY, USA, June 22–27, pp. 337–348. ACM, New York, NY, USA.

[3] Zhao, X., Xiao, C., Lin, X., Liu, Q. and Zhang, W. (2013) A partition-based approach to structure similarity search. *PVLDB*, **7**, 169–180.

[4] Zhao, X., Xiao, C., Lin, X., Wang, W. and Ishikawa, Y. (2013) Efficient processing of graph similarity queries with edit distance constraints. *VLDB J.*, **22**, 1–26.

[5] Jin, R. and Wang, G. (2013) Simple, fast, and scalable reachability oracle. *PVLDB*, **6**, 1978–1989.

[6] Zhu, A.D., Xiao, X., Wang, S. and Lin, W. (2013) Efficient Single-Source Shortest Path and Distance Queries on Large Graphs. *Proc. KDD 13*, Chicago, IL, August 11–14, pp. 998–1006. ACM, New York, NY, USA.

[7] Aggarwal, C.C. and Wang, H. (2010) *Managing and Mining Graph Data* (1st edn). Springer, New York, NY.

[8] Dean, J. and Ghemawat, S. (2004) MapReduce: Simplified Data Processing on Large Clusters. *Proc. OSDI 04*, San Francisco, CA, December 6–8, pp. 137–150. USENIX Association, Berkeley, CA, USA.

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

[10] Malewicz, G., Austern, M.H., Bik, A.J.C., Dehnert, J.C., Horn, I., Leiser, N. and Czajkowski, G. (2010) Pregel: A System for Large-Scale Graph Processing. *Proc. SIGMOD 10*, Indianapolis, IN, June 6–11, pp. 135–146. ACM, New York, NY, USA.

[11] Gao, J., Zhou, C., Zhou, J. and Yu, J.X. (2014) Continuous Pattern Detection Over Billion-Edge Graph Using Distributed Framework. *Proc. ICDE 14*, Chicago, IL, March 31–April 4, pp. 556–567. IEEE, CS Washington, DC, USA.

[12] Inokuchi, A., Washio, T. and Motoda, H. (2000) An Apriori-based Algorithm for Mining Frequent Substructures from Graph

Data. *Proc. PKDD 00*, Lyon, France, September 13–16, pp. 13–23. Springer, Berlin.

[13] Kuramochi, M. and Karypis, G. (2001) Frequent Subgraph Discovery. *Proc. ICDM 01*, San Jose, CA, November 29–December 2, pp. 313–320. IEEE, CS Washington, DC, USA.

[14] Yan, X. and Han, J. (2002) gSpan: Graph-based Substructure Pattern Mining. *Proc. ICDM 02*, Maebashi City, Japan, December 9–12, pp. 721–724. IEEE, CS Washington, DC, USA.

[15] Huan, J., Wang, W. and Prins, J. (2003) Efficient Mining of Frequent Subgraphs in the Presence of Isomorphism. *Proc. ICDM 03*, Melbourne, FL, November 19–22, pp. 549–552. IEEE, CS Washington, DC, USA.

[16] Nijssen, S. and Kok, J.N. (2004) A Quickstart in Frequent Structure Mining Can Make a Difference. *Proc. KDD 04*, Seattle, WA, August 22–25, pp. 647–652. ACM, New York, NY, USA.

[17] Maunz, A., Helma, C. and Kramer, S. (2009) Large-Scale Graph Mining using Backbone Refinement Classes. *Proc. KDD 09*, Paris, France, June 28–July 1, pp. 617–626. ACM, New York, NY, USA.

[18] Wang, C., Wang, W., Pei, J., Zhu, Y. and Shi, B. (2004) Scalable Mining of Large Disk-Based Graph Databases. *Proc. KDD 04*, Seattle, WA, August 22–25, pp. 316–325. ACM, New York, NY, USA.

[19] Cook, D.J. and Holder, L.B. (1994) Substructure discovery using minimum description length and background knowledge. *J. Artif. Intell. Res.*, **1**, 231–255.

[20] Lin, W., Xiao, X. and Ghinita, G. (2014) Large-Scale Frequent Subgraph Mining in MapReduce. *Proc. ICDE 14*, Chicago, IL, March 31–April 4, pp. 844–855. IEEE, CS Washington, DC, USA.

[21] Bhuiyan, M. and Hasan, M.A. (2015) An iterative mapreduce based frequent subgraph mining algorithm. *IEEE Trans. Knowl. Data Eng.*, **27**, 608–620.

[22] Aridhi, S. (2013) Distributed frequent subgraph mining in the cloud. PhD Thesis, Blaise Pascal University Aubière, France.

[23] Kuramochi, M. and Karypis, G. (2005) Finding frequent patterns in a large sparse graph. *Data Min. Knowl. Discov.*, **11**, 243–271.

[24] Reinhardt, S.P. and Karypis, G. (2007) A Multi-level Parallel Implementation of a Program for Finding Frequent Patterns in a Large Sparse Graph. *Proc. IPDPS 07*, Long Beach, CA, March 26–30, pp. 1–8. IEEE, CS Washington, DC, USA.

[25] Fiedler, M. and Borgelt, C. (2007) Support Computation for Mining Frequent Subgraphs in a Single Graph. *Proc. MLG 07*, Firence, Italy, August 1–3. ACM, New York, NY, USA.

[26] Bringmann, B. and Nijssen, S. (2008) What is Frequent in a Single Graph?. *Proc. PAKDD 08*, Osaka, Japan, May 20–23, pp. 858–863. Springer, Berlin.

[27] Elseidy, M., Abdelhamid, E., Skiadopoulos, S. and Kalnis, P. (2014) GraMi: Frequent subgraph and pattern mining in a single large graph. *PVLDB*, **7**, 517–528.

[28] Kuramochi, M. and Karypis, G. (2004) GREW-A Scalable Frequent Subgraph Discovery Algorithm. *Proc. ICDM 04*, Brighton, UK, November 1–4, pp. 439–442. IEEE, CS Washington, DC, USA.

[29] Chen, C., Yan, X., Zhu, F. and Han, J. (2007) gApprox: Mining Frequent Approximate Patterns from a Massive Network. *Proc. ICDM 07*, Omaha, NE, October 28–31, pp. 445–450. IEEE, CS Washington, DC, USA.

[30] Qin, L., Yu, J.X., Chang, L., Cheng, H., Zhang, C. and Lin, X. (2014) Scalable Big Graph Processing in MapReduce. *Proc. SIGMOD 14*, Snowbird, UT, June 22–27, pp. 827–838. ACM, New York, NY, USA.

[31] Bahmani, B., Chakrabarti, K. and Xin, D. (2011) Fast Personalized PageRank on MapReduce. *Proc. SIGMOD 11*, Athens, Greece, June 12–16, pp. 973–984. ACM, New York, NY, USA.

[32] Rastogi, V., Machanavajjhala, A., Chitnis, L. and Sarma, A.D. (2013) Finding Connected Components in Map-reduce in Logarithmic Rounds. *Proc. ICDE 13*, Brisbane, QLD, April 8–12, pp. 50–61. IEEE, CS Washington, DC, USA.

[33] Gao, J., Zhou, J., Zhou, C. and Yu, J.X. (2014) GLog: A High Level Graph Analysis System using Mapreduce. *Proc. ICDE 14*, Chicago, IL, March 31–April 4, pp. 544–555. IEEE, CS Washington, DC, USA.

[34] Tian, Y., Balmin, A., Corsten, S.A., Tatikonda, S. and McPherson, J. (2013) From 'think like a vertex' to 'think like a graph'. *PVLDB*, **7**, 193–204.

[35] Shao, B., Wang, H. and Li, Y. (2013) Trinity: A Distributed Graph Engine on a Memory Cloud. *Proc. SIGMOD 13*, New York, NY, June 22–27, pp. 505–516. ACM, New York, NY, USA.

[36] Khayyat, Z., Awara, K., Alonazi, A., Jamjoom, H., Williams, D. and Kalnis, P. (2013) Mizan: A System for Dynamic Load Balancing in Large-scale Graph Processing. *Proc. EuroSys 13*, Prague, Czech, April 14–17, pp. 169–182. ACM, New York, NY, USA.

[37] Salihoglu, S. and Widom, J. (2014) Optimizing graph algorithms on pregel-like systems. *PVLDB*, **7**, 577–588.

[38] Sun, Z., Wang, H., Wang, H., Shao, B. and Li, J. (2012) Efficient subgraph matching on billion node graphs. *PVLDB*, **5**, 788–799.

[39] Qi, Z., Xiao, Y., Shao, B. and Wang, H. (2013) Toward a distance oracle for billion-node graphs. *PVLDB*, **7**, 61–72.

[40] Wang, L., Xiao, Y., Shao, B. and Wang, H. (2014) How to Partition a Billion-node Graph. *Proc. ICDE 14*, Chicago, IL, March 31–April 4, pp. 568–579. IEEE, CS Washington, DC, USA.

[41] Yang, S., Yan, X., Zong, B. and Khan, A. (2012) Towards Effective Partition Management for Large Graphs. *Proc. SIGMOD 12*, Scottsdale, AZ, May 20–24, pp. 517–528. ACM, New York, NY, USA.

[42] Huang, J., Venkatraman, K. and Abadi, D.J. (2014) Query Optimization of Distributed Pattern Matching. *Proc. ICDE 14*, Chicago, IL, March 31–April 4, pp. 64–75. IEEE, CS Washington, DC, USA.

[43] Mondal, J. and Deshpande, A. (2012) Managing Large Dynamic Graphs Efficiently. *Proc. SIGMOD 12*, Scottsdale, AZ, May 20–24, pp. 145–156. ACM, New York, NY, USA.

[44] Leskovec, J. and Faloutsos, C. (2006) Sampling from Large Graphs. *Proc. KDD 06*, Philadelphia, PA, August 20–23, pp. 631–636. ACM, New York, NY, USA.