Large-Scale Frequent Subgraph Mining in MapReduce

Wenqing Lin[#], Xiaokui Xiao[#], Gabriel Ghinita[*]

[#]*Nanyang Technological University, Singapore*
{wlin1, xkxiao}@ntu.edu.sg

[*]*University of Massachusetts Boston*
Gabriel.Ghinita@umb.edu

***Abstract*— Mining frequent subgraphs from a large collection of graph objects is an important problem in several application domains such as bio-informatics, social networks, computer vision, etc. The main challenge in subgraph mining is efficiency, as *(i)* testing for graph isomorphisms is computationally intensive, and *(ii)* the cardinality of the graph collection to be mined may be very large. We propose a two-step *filter-and-refinement* approach that is suitable to massive parallelization within the scalable MapReduce computing model. We partition the collection of graphs among worker nodes, and each worker applies the filter step to determine a set of candidate subgraphs that are *locally frequent* in its partition. The union of all such graphs is the input to the refinement step, where each candidate is checked against all partitions and only the *globally frequent* graphs are retained. We devise a statistical threshold mechanism that allows us to predict which subgraphs have a high chance to become globally frequent, and thus reduce the computational overhead in the refinement step. We also propose effective strategies to avoid redundant computation in each round when searching for candidate graphs, as well as a lightweight graph compression mechanism to reduce the communication cost between machines. Extensive experimental evaluation results on several real-world large graph datasets show that the proposed approach clearly outperforms the existing state-of-the-art and provides a practical solution to the problem of frequent subgraph mining for massive collections of graphs.**

I. Introduction

Applications from several areas such as bio-informatics, computational chemistry, social networks, the semantic web and computer vision, make use of large amounts of data encoded as graphs. For instance, in the bio-informatics domain, graphs can naturally model protein structures. By looking at large sample sets of such graphs and determining common formations among them, researchers are able to understand what is the role of a certain protein-protein interaction network. Frequent subgraph patterns in social networks can help identify relationships within different groups, and help understand the mechanics of social behavior and interactions. The necessity to search for patterns within massive amounts of graph data, coupled with the computationally-intensive nature of testing graph isomorphism relationships (the fundamental operation in graph mining) makes the graph mining problem a very challenging one from a performance standpoint.

There are two broad categories of large-scale frequent subgraph mining scenarios: in the first case, there is one single large graph of massive scale, in the order of terabytes of data, and frequent subgraph patterns must be found in different regions of the graph. In the second case, frequent subgraphs must be found within a large-scale collection of moderate-sized graphs. The former case is relevant to the social network domain, whereas the latter finds many applications in the areas of bio-informatics and computational chemistry. Both scenarios share a number of common challenges, such as large data input size, which may exceed the memory resources of a single machine, and vast amounts of CPU time required to compute frequent patterns. Given these characteristics, cloud computing and the widespread MapReduce framework represent a promising direction to solve these challenging problems.

Several solutions have been proposed for the single-graph scenario in either a sequential [17], [19], [24], [25] or parallel computing (MapReduce, MPI) framework [21], [33], [38]. However, our focus is on the equally-important case of mining a large collection of individual, moderate-sized graphs. This problem is also known in literature as subgraph mining in a *transaction setting* [15]. The objective is to find subgraphs that occur with *support* higher than a *threshold* θ expressed as a fraction of the collection cardinality, i.e., $0 \leq \theta \leq 1$. The early solutions to this problem are memory-based [16], [18], [23], [29], [32], [40], and assume that the entire graph collection fits in memory. However, as data size increases, the assumption no longer holds. To address the limited amount of main memory, some disk-based graph database solutions have been proposed [37]. These approaches do solve the memory limitation, but they incur significant overhead for accessing the data, as the number of disk I/Os is very high. The work in [15] is the only one so far to employ MapReduce for mining a large collection of graphs. That solution takes an incremental approach, similar in concept to the *Apriori* algorithm [18] for graph mining. Specifically, in the first step, a fraction of the graph collection is mapped to each worker, which determines the local support (on its data partition) for all possible *single-edge* subgraphs. A subsequent reduction phase determines the global support for each such subgraph, and candidates that do not meet the global support threshold are discarded. The program continues to the next step where all *two-edge* subgraphs are generated from the set of retained candidates, and so on until all frequent subgraphs are found. Although this method uses MapReduce, the large number of resulting steps still creates significant performance problems, as we will prove experimentally in Section VII.

We propose a two-step *filter-and-refinement* approach that uses MapReduce, but considers a completely different computation workflow than the work in [15]. The proposed workflow

is more suitable to massive parallelization, and works as follows: first, in the *filter* step, the collection of graphs is partitioned among worker nodes, and each worker determines a set of *locally frequent* subgraphs on its local partition. As opposed to [15], there is no restriction on the size of such subgraphs. Next, the union of all local candidates is processed in the *refinement* step, where each candidate is evaluated across *all* partitions, not only that where it originated, and only the *globally frequent* subgraphs are retained. The benefit of the two-step approach is that it reduces the amount of communication among worker nodes, and at the same time it allows for a high degree of parallelism within each step.

Recall that, an important source of computational overhead in graph mining is testing for subgraph isomorphism (a NP-hard problem). To improve efficiency, we devise a statistical model that predicts which subgraphs have a high chance to become globally frequent, and thus reduce the overhead of redundant subgraph isomorphism testing in the refinement step. Furthermore, we propose effective strategies for reusing computation at each worker in the process of isomorphism testing for subgraphs that share edges.

Communication cost is also an important concern, as large amounts of intermediate data may be generated and transferred among workers. Excessive network transmission increases the overall execution time of graph mining, and may also lead to bottlenecks and failures. To reduce the amount of communication, we devise a lightweight graph compression scheme which reduces the amount of information that needs to be transferred between machines, while at the same time keeping the encoding/decoding computational overhead low.

In summary, our contributions are:

(i) A novel two-step filter-and-refinement computation workflow which uses MapReduce for frequent subgraph mining, is highly parallelizable and avoids excessive amounts of data communication.

(ii) A statistical model for predicting which locally frequent subgraphs are likely to also be globally frequent. The model increases slightly the amount of computation in the filter step, but has as benefit significant gains in the refinement step.

(iii) A strategy for reusing computation in the expensive process of isomorphism testing for graphs that share edges. We investigate both a top-down and a bottom-up approach targeted at reducing computational cost at each worker node.

(iv) A lightweight technique for graph compression that reduces significantly the amount of data transmission between the worker nodes and the file system, and hence decreases further the overall runtime of the mining algorithm.

(v) An extensive experimental evaluation on several real-world graph datasets which shows that the proposed approach clearly outperforms the existing state-of-the-art.

The remainder of the paper is organized as follows: Section II introduces fundamental concepts and definitions. Section III gives an overview of the proposed approach, whereas Sections IV and V provide specific details of the filter and refinement steps, respectively. The graph compression scheme to reduce communication cost is presented in Section VI. An extensive experimental evaluation is presented in Section VII, followed by a review of related work in Section VIII and conclusions in Section IX.

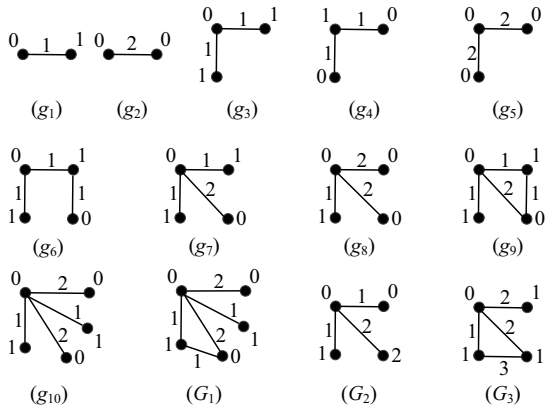


Fig. 1. Frequent Subgraph Computation.

II. Preliminaries

Section II-A focuses on *mining frequent subgraphs*, whereas Section II-B provides a brief MapReduce primer.

A. Mining Frequent Subgraphs

Let $\mathcal{G}$ be a set of n graphs where each node and edge is labeled. We denote each graph $G \in \mathcal{G}$ as a quadruple $G = (V, E, L, l)$, where V and E are the set of vertices and edges in G, respectively, and l is a labeling function that maps each vertex and edge in G to a label in a finite alphabet L. For ease of exposition, we assume that all edges in G are undirected and connected; however, our results can be easily extended to the case of directed or disconnected graphs.

Given any graph $G' = (V', E', L', l')$, we say G' is a *subgraph* of another graph $G = (V, E, L, l)$, if there exists an injective function $\mu : V' \rightarrow V$ such that $\forall (u, v) \in E'$ it holds that

$$(l(u) = l'(\mu(u))) \wedge (l(v) = l'(\mu(v))) \wedge (l(u, v) = l'(\mu(u), \mu(v))).$$

In other words, the labels for each edge as well as the labels for the edge's endpoints are identical. We use $G' \subseteq G$ to denote that G' is a subgraph of G, and we refer to G as a *super-graph* of G'.

The *frequency* of a graph G' in $\mathcal{G}$, denoted as $f(G')$, is defined as the number of graphs in $\mathcal{G}$ that contain G' as subgraph. That is,

$$f(G') = \big|\, \{G \mid G \in \mathcal{G} \wedge G' \subseteq G\} \,\big|.$$

Meanwhile, the *support* of G' in $\mathcal{G}$ is defined as $f(G')/n$, i.e., the fraction of graphs in $\mathcal{G}$ that are super-graphs of G'. We say that G' is a *frequent subgraph* in $\mathcal{G}$, if the support of G' is not less than a *support threshold* θ ($0 \leq \theta \leq 1$).

Example 1: Figure 1 shows 13 graphs, $g_1, \cdots, g_{10}$, and G_1, G_2, G_3, where the label of each vertex is in the domain

TABLE I
SUMMARY OF NOTATIONS

Notation	Description
$\mathcal{G}$	input collection of graphs
$n = \|G\|$	number of graphs in $\mathcal{G}$
θ	given support threshold
$f(G)$	frequency of G in $\mathcal{G}$
$s(G)$	support of G in $\mathcal{G}$
m	number of MapReduce machines (workers)
M_i	i-th machine (worker), $(i = 1, \ldots, m)$
$\mathcal{G}_i$	subset of graphs in $\mathcal{G}$ that is distributed to M_i
$n_i = \|\mathcal{G}_i\|$	number of graphs in $\mathcal{G}_i$
$f_i(G)$	frequency of G in $\mathcal{G}_i$
$f_i^{\top}(G)$	upper-bound of $f_i(G)$ (see Eq. (2))
$f^{\top}(G)$	upper-bound of $f(G)$, defined as $\sum_{i=1}^{m} f_i^{\top}(G)$
ρ	probability threshold (see Section IV-B)

$[0, 2]$, and the label of each edge is in the domain $[1, 3]$. For each g_i $(1 \leq i \leq 10)$, there exists a graph G_j where $1 \leq j \leq 3$ such that g_i is a subgraph of G_j. For instance, g_{10} is a subgraph of G_1, since we can injectively map all the vertices and edges of g_{10} to the vertices and edges of G_1. Consider a graph set $\mathcal{G} = \{G_1, G_2, G_3\}$, the frequency of g_1 is $f(g_1) = \left|\{G_1, G_2, G_3\}\right| = 3$, and, similarly, the frequency of g_{10} is $f(g_{10}) = \left|\{G_1\}\right| = 1$. That is, the support of g_1 is $s(g_1) = 1$, and the support of g_{10} is $s(g_{10}) = 1/3$. ■

Given θ and $\mathcal{G}$, our objective is to identify all frequent subgraphs in $\mathcal{G}$, as well as the frequency of each frequent subgraph in $\mathcal{G}$. We aim to accomplish this task using a *MapReduce* program on m machines.

B. MapReduce

A MapReduce program, also referred to as a *workflow*, consists of several *rounds*, each of which contains three *phases*: *map*, *shuffle*, and *reduce*, as follows:

1) *Map.* In this phase, each machine reads data from a *distributed file system (DFS)*, and applies a *map function* on the data to convert them into a set of *pairs*. Each pair consists of a *key* and a *value*.

2) *Shuffle.* In this phase, the key-value pairs are aggregated by keys, and the values in the pairs with the same key are grouped as a list. Each key, along with the corresponding list of values, is then sent to one of the machines.

3) *Reduce.* Each machine examines the keys and lists that it receives in the shuffle phase, and then applies a *reduce function* on each list of values. The function transforms each list into new key-value pairs, which are then stored in the DFS and can be utilized by subsequent rounds of the MapReduce program.

In subsequent sections, we focus on the design of the Map and Reduce functions, as the shuffle phase is automatically handled by the MapReduce infrastructure (e.g., Hadoop [5]).

Table I summarizes the notations used throughout the paper.

III. SOLUTION OVERVIEW

The proposed filter-and-refinement MapReduce solution is illustrated in Figure 2 and consists of three rounds of computation:

Round 1: *Filter*. In the map phase of this round, each machine M_i $(i = 1, \ldots, m)$ reads a *disjoint* subset $\mathcal{G}_i$ of $\mathcal{G}$ and identifies a set of graphs H_i, such that *(i)* each graph $G \in H_i$ is the subgraph of at least one graph in $\mathcal{G}_i$, and *(ii)* G is likely to be a frequent subgraph in $\mathcal{G}$. Then, for each graph $G \in H_i$, M_i outputs a key-value pair where the key is G and the value indicates $f_i(G)$, i.e., the number of graphs in $\mathcal{G}_i$ that are super-graphs of G. Next, in the shuffle phase (not explicitly shown in the diagram), all key-value pairs having G as key are sent to the same machine, say M_j. Finally, in the reduce phase, M_j inspects the list of values with key G, and computes the sum of all individual frequencies. Based on the sum, M_j evaluates whether G is likely to be a frequent subgraph in $\mathcal{G}$. If G cannot be a frequent subgraph, then it is discarded; otherwise, M_j outputs a key-value pair with key G and value equal to the sum of frequencies (as we will discuss in Section IV the value contains some additional information as well).

Round 2: *Sorting*. The set of key-value pairs obtained from the filter round is sorted in ascending order of *graph size* (defined as number of graph edges) in the key. This is accomplished by invoking a single-round MapReduce sorting algorithm such as *TeraSort* in *Hadoop* [5]. The sorted sequence is then stored in the DFS.

Round 3: *Refinement*. In the map phase, each machine M_i reads $\mathcal{G}_i$ and the sorted sequence S from the DFS. Then, for each graph G that appears in S, M_i determines $f_i(G)$, the frequency of G in $\mathcal{G}_i$. Next, M_i outputs a key-value pair $\langle G, f_i(G)\rangle$. Such key-value pairs are re-distributed among the machines in the shuffle phase. Then, for each key G, the reduce phase computes the sum of the values corresponding to G. (Note that this sum equals $f(G)$, the exact frequency of G in $\mathcal{G}$.) If $f(G)/n$ is not less than the support threshold θ, then a key-value pair $\langle G, f(G)\rangle$ is written to the DFS to indicate that G is a frequent subgraph in $\mathcal{G}$.

Since there is nothing specific to the studied problem within the sorting round, we omit it further from consideration, and focus on the filter and refinement rounds, where the challenges reside. First, in the filter round, given that each machine M_i sees only a subset $\mathcal{G}_i$ of the graphs in $\mathcal{G}$, we focus on how can M_i identify subgraphs that are likely to be frequent in $\mathcal{G}$ (i.e., globally frequent). We refer to such subgraphs as frequent subgraph *candidates*. This must be done in a manner that ensures both correctness and efficiency of the overall solution (Section IV). Second, in the refinement round, when each machine M_i computes the frequency of candidate graphs in $\mathcal{G}_i$, we focus on how to lower the cost of subgraph isomorphism tests required to identify super-graphs of the candidates (Section V). Third, as each machine may generate a relatively large set of candidate subgraphs, we study how to reduce the communication overhead in re-distributing and manipulating

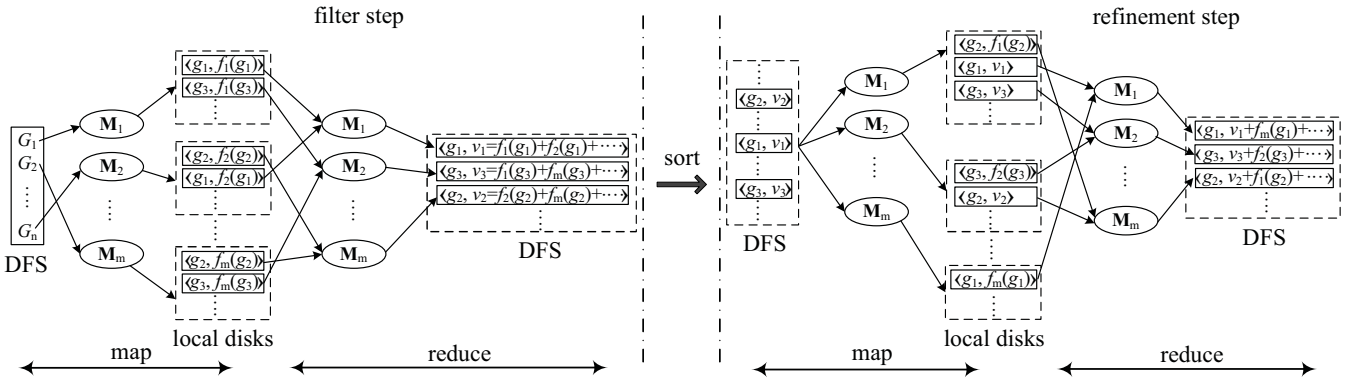


Fig. 2. Proposed Filter-and-Refinement MapReduce Workflow for Frequent Subgraph Mining.

the key-value pairs for the candidates (Section VI).

IV. Filter Round

The filter round distributes graphs in $\mathcal{G}$ onto the m machines, and asks each machine to report candidates for frequent subgraphs. To ensure completeness of the result, the set of candidates reported should not incur any *false negative*, i.e., every frequent subgraph in $\mathcal{G}$ must be reported as a candidate by at least one machine.

A. A Preliminary Approach

One simple approach to implement the filter round is as follows:

1) In the map phase, divide $\mathcal{G}$ into m disjoint subsets $\mathcal{G}_1, \mathcal{G}_2, \ldots, \mathcal{G}_m$, and send $\mathcal{G}_i$ $(i = 1, \ldots, m)$ to machine M_i. Next, each M_i reports a graph G as a candidate whenever $f_i(G)/n_i \geq \theta$.

2) In the reduce phase, the union of the sets of candidates produced by the m machines is written to the DFS.

In other words, we report a graph as a candidate whenever it is *locally frequent* on some machine. This avoids false negatives because if a graph G' is not reported as a candidate by any machine, then the frequency of G' in $\mathcal{G}$ must satisfy the following inequality:

$$f(G') = \sum_{i=1}^{m} f_i(G') < \sum_{i=1}^{m} (\theta \cdot n_i) = \theta \cdot n,$$

i.e., we have $s(G') = f(G')/n < \theta$, and hence, G' cannot be a frequent subgraph.

However, this approach is inefficient as it may lead to a large number of *false positives*, i.e., candidate graphs that are reported even though they are actually infrequent in $\mathcal{G}$. Consider a graph G such that $f(G) < n \cdot \theta$. For G not to be reported, it must hold that on each of the m machines

$$\forall i \in [1, m], f_i(G) < n_i \cdot \theta. \tag{1}$$

For Eq. (1) to hold, the graphs in $\mathcal{G}$ that are super-graphs of G must be distributed uniformly to the m machines. However, such an even distribution is difficult to obtain, especially when $f(G)$ is close to $n \cdot \theta$, as it requires prior knowledge of the super-graphs of G in $\mathcal{G}$, which is not available before the map phase of the filter round.

To reduce the number of false positives, one can utilize the map phase of the filter round. As a naive strategy, assume that in the map phase each machine M_i outputs the frequency for *all* subgraphs G in $\mathcal{G}_i$ (regardless of whether G is locally frequent or not). Then, in the reduce phase, we can sum up the frequency of G on each machine to obtain its global frequency $f(G)$, based on which we can precisely decide whether G is globally frequent. As such, we can eliminate all false positives in the candidate graphs, but at the cost of computing local frequencies for an excessive number of graphs. Next, we develop a more advanced approach for candidate graph generation that reduces false positives without incurring the prohibitive cost of frequency computation for all subgraphs.

B. An Improved Approach

The improved approach for the filter round works as follows:

1) In the map phase, we distribute each graph $G \in \mathcal{G}$ to a randomly selected machine, i.e., each machine M_i receives a sample set of $\mathcal{G}_i$ with a sampling rate $1/m$.

2) Then, M_i outputs $f_i(G)$ of any graph G that is locally frequent in $\mathcal{G}_i$ (i.e., $f_i(G) \geq n_i \cdot \theta$). In addition, for a *selected set* of graphs that are locally infrequent in $\mathcal{G}_i$, M_i also outputs key-value pairs that record the local frequencies of those graphs.

3) In the reduce phase, for each graph G that is locally frequent on some machine, we derive an upper-bound of $f(G)$ (i.e., the global frequency of G). If the upper-bound is at least $\theta \cdot n$, then we output a key-value pair for G to indicate that it is a candidate frequent subgraph.

Specifically, the upper-bound of $f(G)$ that we use in the reduce phase is $f^{\top}(G) = \sum_{i=1}^{m} f_i^{\top}(G)$, where

$$f_i^{\top}(G) = \begin{cases} f_i(G), & \text{if } \mathsf{M}_i \text{ reports } f_i(G) \text{ in the map phase} \\ \lceil n_i \cdot \theta \rceil - 1, & \text{otherwise} \end{cases} \tag{2}$$

That is, whenever $f_i(G)$ is unknown, we use $\lceil n_i \cdot \theta \rceil - 1$ as an optimistic estimation of $f_i(G)$.

Selection of locally infrequent graphs. For the above approach to be efficient, we need to carefully select the set

of locally infrequent graphs output by each machine M_i in the map phase. If this set is too large in size, performance will decrease. On the other hand, if it is too small, then the reduce phase may leave a large number of false positives in the candidate graphs, since the frequency upper-bound $f^\top(G)$ for a candidate graph G tends to be loose when the local frequencies of G are not reported by a large portion of the machines. This calls for a method that can *selectively* choose locally infrequent graphs from each machine while maintaining the pruning power of $f^\top(G)$.

To obtain a good trade-off, one needs to determine when will a locally infrequent graph be useful for pruning false positives in the candidate graphs. Consider a graph G' that is locally infrequent on machine M_i. Assume that M_i reports the local frequency of G' (i.e., $f_i(G')$) in the map phase. Then, $f_i(G')$ will help eliminate false positives in the reduce phase only if G' is reported as a candidate graph by at least one machine M_j, in which case $f_i(G')$ will contribute to the calculation of the upper-bound $f^\top(G')$ that decides whether G' should be pruned. In other words, M_i should report $f_i(G')$ only if G' is locally frequent on some other machine M_j. Nevertheless, as M_i only sees its local data $\mathcal{G}_i$ in the map phase, it cannot determine whether G' might be locally frequent on other machines. To address this issue, we propose a probabilistic inference method as follows.

First, given that *(i)* $\mathcal{G}_i$ is a random sample set of $\mathcal{G}$ and *(ii)* a $f_i(G')/n_i$ fraction of the graphs in $\mathcal{G}_i$ are super-graphs of G', M_i estimates that the frequency of G' in $\mathcal{G}$ is roughly $n \cdot f_i(G')/n_i$. Given this estimation, M_i infers that the local frequency of G' on any other machine M_j follows a binomial distribution:

$$Pr\{f_j(G') = k\} = \binom{n_j}{k} p^k (1-p)^{n_j-k},$$

where $p = f_i(G')/n_i$. Accordingly, the event that G' is locally frequent on M_j should occur with the following probability:

$$Pr\{f_j(G') \geq \lceil \theta \cdot n_j \rceil\} = \sum_{k=\lceil \theta \cdot n_j \rceil}^{n_j} \binom{n_j}{k} p^k (1-p)^{n_j-k}.$$

By the union bound, M_i gets the following upper-bound on the probability that G' is locally frequent on at least one machine besides M_i:

$$\begin{aligned} & Pr\{\exists j \neq i, f_j(G') \geq \lceil \theta \cdot n_j \rceil\} \\ \leq & \sum_{j \neq i} Pr\{f_j(G') \geq \lceil \theta \cdot n_j \rceil\} \\ = & \sum_{j \neq i} \sum_{k=\lceil \theta \cdot n_j \rceil}^{n_j} \binom{n_j}{k} p^k (1-p)^{n_j-k}. \end{aligned} \quad (3)$$

Note that this upper-bound is easy to compute given the cumulative distribution function of the binomial distribution.

Based on the above upper-bound, M_i can then decide whether the local frequency of G' should be output. In particular, if the upper-bound falls below a threshold ρ which is a system parameter (e.g., $< 5\%$), then M_i infers that G' is unlikely to be locally frequent on any other machines, and hence, it would not output any key-value pair pertinent to G'. As such, M_i can avoid reporting a large number of key-value pairs that are useless for the pruning procedure in the reduce phase. In Section VII we discuss a methodology for setting the value of ρ in practice.

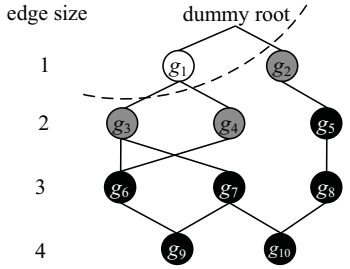


Fig. 3. Subgraph lattice.

Computing graph frequencies. To compute the set of locally frequent subgraphs, one of several techniques [16], [18], [23], [29], [32], [37], [40] that address subgraph mining in a sequential setting can be used. All such techniques follow a branch-and-bound approach. All subgraphs are organized in a lattice, where each node denotes a subgraph. We add a *dummy root* node above the single-edge subgraphs (corresponding to an empty subgraph). Each node in the lattice extends the graph of its parent node(s) by adding a single edge, i.e., a child node is a super-graph of all its parent nodes. The lattice is traversed top-down, and the support of each visited node is computed. If a visited graph is frequent, the corresponding graph and its frequency are output, and the traversal continues with the child nodes. Otherwise, all its child nodes and their descendants are pruned.

Example 2: Figure 3 shows the lattice formed by the graphs $g_1, \cdots, g_{10}$ in Figure 1. Each node in the lattice denotes a subgraph. Let $\theta = 0.5$. In the beginning of the mining process, we visit the single-edge subgraphs, i.e., g_1 and g_2. Then, we compute their supports, as $s(g_1) = 1$ and $s(g_2) = 1/3$. Since $s(g_2) < \theta$, we stop visiting all descendants of g_2 in the lattice. As $s(g_1) > \theta$, we output g_1, as well as its frequency, and continue the traversal on the branch of g_1 by visiting g_3 and g_4. Next, we compute $s(g_3) = s(g_4) = 1/3$, and the branches of g_3 and g_4 are pruned. The traversal stops since there are no more nodes to visit. ■

For illustration purposes, we use white nodes to denote frequent subgraphs, grey nodes to denote visited nodes which are infrequent, and black nodes to denote unvisited subgraphs. Existing algorithms return only white nodes (i.e., frequent subgraphs). However, to improve the efficiency of the refinement step, we choose to also return some grey nodes, if their probability to be globally frequent is above threshold ρ. Note that, even though existing techniques do not output grey nodes, they still have to compute their frequencies, as required by the termination condition of the traversal algorithm. Therefore, our decision to output grey subgraphs as well comes at no additional computational cost in the filter step.

Efficient computation of upper-bound probabilities. Each machine M_i may produce a large number of subgraphs, so

computing the probability in Eq. (3) separately for each subgraph would be costly. Due to the fact that Eq. (3) is monotonic in the range $[1, \lceil \theta \cdot n_j \rceil]$, given a user-defined probability upper-bound ρ, we can employ *binary search* to determine an appropriate value for frequency f_i that is equal or close to the frequency of a graph G whose upper-bound probability is ρ. Let $\alpha = \lceil \theta \cdot n_j \rceil$. The binary search first considers the range $[1, \alpha]$ and computes the probability for frequency $f_i = \lceil \alpha/2 \rceil$. If the probability is larger than ρ, we change the range to $[1, \lceil \alpha/2 \rceil]$, and compute the probability at $\lceil \alpha/4 \rceil$; otherwise, the range of $[\lceil \alpha/2 \rceil + 1, \alpha]$ is considered, and we compute the probability at $\lceil (\lceil \alpha/2 \rceil + 1 + \lceil \alpha \rceil)/2 \rceil$ to determine the next range. The search will stop if the obtained probability value converges close enough to ρ. Let f_p denote the frequency value obtained as above. For a given graph G, if $f_i(G)$ is smaller than f_p, the probability that G is globally infrequent is high, hence we prune G from further computation.

Implementation Issues. In the map phase, each key-value pair that is sent out by a machine M_i has a graph G as the key, and the frequency of G as well as the *machine id* i of M_i as the value. In the key, G is represented in a graph canonical form, which will be compressed as discussed in Section VI. The canonical form of each graph G allows us to aggregate the graphs in the shuffle phase so as to obtain the upper-bound of $f(G)$ in the reduce phase. As to the value, the frequency of G in each machine will be calculated as a sum in the reduce phase, while the machine ids will be retained as one part of the new value. Note that, the machine id in the value enables us to identify the graphs that are sent out by all machines, i.e., a subset of globally frequent subgraphs. These graphs as well as their frequencies are written to DFS as part of the final result. Furthermore, when the key-value pairs are read by machine M_i in the refinement round, if M_i already appears in the list of machine ids then the frequency of the graph in the key does not need any refinement on behalf of M_i, so redundant computation is avoided.

V. Refinement Round

At the start of the refinement round, each machine M_i receives the sorted sequence S of candidate frequent graphs from the DFS. Machine M_i constructs $\mathcal{S}_i$ by removing from S all key-value pairs that contain the machine id of M_i (if a value contains the id of M_i, it means that there is no refinement required in the current partition $\mathcal{G}_i$ of the data). Each candidate graph G in $\mathcal{S}_i$ is locally infrequent in M_i, i.e., $f_i(G) < \lceil \theta \cdot n_i \rceil$, and its frequency needs to be determined.

Given a candidate graph G, we denote as $\mathcal{A}^i(G)$ the set of super-graphs of G in $\mathcal{G}_i$. To compute the frequency $f_i(G) = \left|\mathcal{A}^i(G)\right|$, a naive approach is to perform a subgraph isomorphism test between G and each graph in $\mathcal{G}_i$. However, due to the computational complexity of subgraph isomorphism testing, such an approach will not scale well. Recall that, inclusion relationships exist among candidate graphs, captured by a lattice structure. A child node G in the lattice is a super-graph of a parent node G', hence $\mathcal{A}^i(G) \subseteq \mathcal{A}^i(G')$.

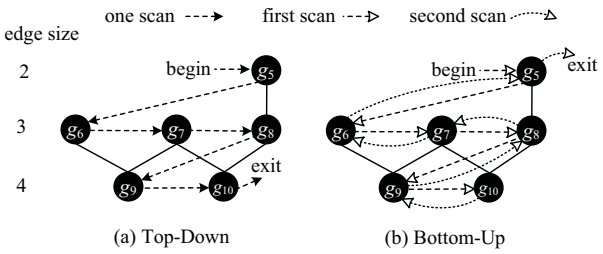


Fig. 4. Top-Down vs. Bottom-Up Strategy.

Such dependencies help save redundant computation. We devise two strategies for frequency computation, *Top-Down* and *Bottom-Up*, named after the direction in which the strategy is traversing the lattice.

A. The Top-Down Approach

In the Top-Down approach, we scan $\mathcal{S}_i$ once in ascending order of graph size (i.e., from top to bottom of the lattice), and compute $\mathcal{A}^i(G)$ for each candidate graph G as soon as we visit node G. Specifically, if there is no candidate graph G' in $\mathcal{S}_i$ such that G' is a subgraph of G (i.e., G has no parents), we perform isomorphism testing against every graph in $\mathcal{G}_i$ to compute $\mathcal{A}^i(G)$. Otherwise, we first compute an *upper-bound* of $\mathcal{A}^i(G)$ by intersecting $\mathcal{A}^i(G')$ for all $G' \subseteq G$ and $G' \in \mathcal{S}_i$. We denote the upper-bound of $\mathcal{A}^i(G)$ as $ub(G)$, i.e.,

$$ub(G) = \bigcap_{G' \subseteq G \wedge G' \in \mathcal{S}_i} \mathcal{A}^i(G').$$

Note that, for any graph $G' \in S_i$ which is a subgraph of G,

$$\left|\mathcal{A}^i(G)\right| \leq \left|ub(G)\right| \leq \left|\mathcal{A}^i(G')\right| < \lceil \theta \cdot n_i \rceil \ll n_i,$$

if θ is sufficiently small. Besides, since we process the candidate graphs in $\mathcal{S}_i$ in ascending order of graph size, we have all $\mathcal{A}^i(G')$ before visiting G. Therefore, to compute $\mathcal{A}^i(G)$, we perform isomorphism testing only against the graphs in $ub(G)$, instead of the entire $\mathcal{G}_i$.

The computational overhead can be reduced further if we consider only the set of *maximum* subgraphs of G. We say that a graph G' in $\mathcal{S}_i$ is a maximum subgraph of G if there does not exist another graph G^+ in S_i such that G^+ is a subgraph of G and also a super-graph of G'. To explain that, consider that there exists a graph $G^+ \in S_i$ such that G^+ is a subgraph of G', then we have $\mathcal{A}^i(G') \subseteq \mathcal{A}^i(G^+)$. Therefore, $\mathcal{A}^i(G^+)$ is not useful to reduce the size of $ub(G)$ if we already visited G'. In the lattice, the parent nodes of G are the maximum subgraphs of G, which are easy to determine.

Example 3: Continuing Example 2, Figure 4(a) shows the graph processing order of the Top-Down strategy with dashed arrows. We begin by processing g_5, which does not have any parents, so $\mathcal{A}^i(g_5)$ is computed by performing isomorphism testing against every graph in $\mathcal{G}_i$. The traversal then continues as shown, with $g_6, \ldots, g_{10}$. In the case of nodes that have parents, it is possible to reuse some of the earlier computation. For instance, when visiting g_{10}, we compute $ub(g_{10}) = \mathcal{A}^i(g_7) \cap \mathcal{A}^i(g_8)$, and examine only the graphs in $ub(g_{10})$ to obtain $\mathcal{A}^i(g_{10})$. ■

B. The Bottom-Up Approach

The drawback of the Top-Down approach is that it still performs some redundant computations. Since the super-graphs of G are also the super-graphs of G's subgraphs, the super-graphs of G will be examined for each of G's subgraphs in the Top-Down approach. To avoid the redundancy, we devise the Bottom-Up approach, which consists of two scans of $\mathcal{S}_i$.

The first scan processes the candidate graphs in $\mathcal{S}_i$ in ascending order of graph size. Given a candidate graph G, if G does not have subgraphs in $\mathcal{S}_i$, we compute $\mathcal{A}^i(G)$ by examining every graph in $\mathcal{G}_i$. That is, we obtain a tight upper-bound as $ub(G) = \mathcal{A}^i(G)$. Otherwise, if G has subgraphs G' in $\mathcal{S}_i$, the upper-bound of $\mathcal{A}^i(G)$ is then computed by intersecting the upper-bounds of all $\mathcal{A}^i(G')$, i.e.,

$$ub(G) = \bigcap_{G' \subseteq G \wedge G' \in \mathcal{S}_i} ub(G').$$

However, the isomorphism testing against graphs in $ub(G)$ is not performed right away, instead it is deferred until later on in the algorithm. According to the analysis in Section V-A, $ub(G')$ is obtained before processing G, and $\left|ub(G)\right| < \lceil \theta \cdot n_i \rceil$. In addition, we only need to consider the maximum subgraph of G to compute $ub(G)$.

The second scan processes the candidate graphs in $\mathcal{S}_i$ in *descending* order of graph size. Given a candidate graph G, if G has no super-graphs in $\mathcal{S}_i$, we examine every graph in $ub(G)$, which is computed in the first scan, to obtain $\mathcal{A}^i(G)$. Otherwise, a *lower bound* of $\mathcal{A}^i(G)$ is computed by merging $\mathcal{A}^i(G')$ for all G's super-graphs $G' \in S_i$. We denote the lower bound of $\mathcal{A}^i(G)$ as $lb(G)$, i.e.,

$$lb(G) = \bigcup_{G \subseteq G' \wedge G' \in \mathcal{S}_i} \mathcal{A}^i(G').$$

Since we scan the graphs in descending order of graph size, the frequency of G' has been computed before G. Note that, the graphs in $lb(G)$ are super-graphs of G, therefore, to compute the frequency of G, we only need to examine the graphs in a subset of $ub(G)$, i.e.,

$$ub(G) \setminus lb(G).$$

To avoid re-computing the frequency of G in the second scan in cases where $\mathcal{A}^i(G)$ has been computed in the first scan, we mark G in the first scan. Before processing a graph in the second scan, we first check its mark to determine whether we need to compute its frequency or not.

Example 4: Figure 4(b) illustrates the Bottom-Up strategy, which first visits g_5, and then visits all the other nodes following the dashed arrows. The upper bounds are constructed in this traversal, but no isomorphism tests are performed (except for nodes without parents). When g_{10} is reached, it is determined that $ub(g_{10}) = ub(g_7) \cap ub(g_8)$, and the actual subgraph isomorphism testing is performed for graphs in $ub(g_{10})$. Then, the traversal returns following the dotted arrows, towards the first node g_5. During the second traversal, when re-visiting some node, only graphs in the difference between the upper bound and lower bound of the respective node must be included for isomorphism testing. For instance, when re-visiting g_8, we only examine the graphs in $ub(g_8) \setminus \mathcal{A}^i(g_{10})$. ■

Recall that for each graph G, $ub(G)$ is already sufficiently small if θ is small. Removing the graph in the lower bound from the upper bound further shrinks the total number of graphs for subgraph isomorphism test, especially when $\mathcal{S}_i$ is large. Therefore, a significant amount of computation can be saved with this strategy.

VI. Reducing Communication Overhead

Frequent subgraph mining with MapReduce produces a large set of key-value pairs which must be transferred via network communication. This may incur significant overhead, since keys are canonical labelings of graphs, which are sizable. We propose a lightweight compression technique that is able to represent the canonical labeling of a graph $G = (V, E, L, l)$ using only $O(|E| \log_w |L|)$ bits, whereas the computational complexity of encoding/decoding is only $O(|E|^2 \log_w |L|)$, where w is the size of a machine word.

A. Compression of Canonical Labeling

The canonical labeling of a graph corresponds to a unique permutation of its edges or vertices. For example, *CAM (Canonical Adjacency Matrix)* [18] computes a permutation of vertices such that the code produced from the adjacency matrix for that permutation is maximal, whereas the DFS code [40] finds a permutation of edges such that the ordering obtained is a maximal code. We adopt the CAM approach, but our technique can be extended to other canonical labeling methods.

For real-world graph datasets, trees are very common, and most of frequent subgraphs are trees [29], [32]. Therefore, our approach first focuses on spanning trees, and then extends to the case of non-tree edges.

Encoding Spanning Trees. Given graph $G = (V, E, L, l)$, denote its spanning tree by $T = (V, E', L, l)$, where $E' \subseteq E$. Let the permutation of vertices be $V = (v_0, v_1, \cdots, v_{d-1})$, $d = |V|$, where v_0 is the root of T and each other vertex v_i is incident to one and only one vertex v_j for $j < i$ (i.e., its parent). To store T in an adjacency list, for each vertex v_i, we record its label $l(v_i)$ as well as the edge $\langle v_{e(i)}, l(v_i, v_{e(i)})\rangle$ that connects v_i to its parent $e(i)$. We use a vector to store the column of vertex labels, denoted as $L_v = (l(v_0), l(v_1), \cdots, l(v_{d-1}))$. Similarly, we denote $I_v = (e(1), e(2), \cdots, e(d-1))$ as the set of parent nodes for vertices v_i ($i = 1, \cdots, d-1$), and $L_e = (l(v_1, v_{e(1)}), l(v_2, v_{e(2)}), \cdots, l(v_{d-1}, v_{e(d-1)}))$ as the corresponding vertex-parent edges (note that $e(0)$ is not defined).

Compressing tree T boils down to compressing the vectors L_v, I_v, and L_e. Note that, the elements in those vectors are often from a small domain. For instance, the elements in L_v and L_e are chosen from the set L of distinct labels in the dataset, and the elements of I_v are positive numbers smaller than the number of vertices in T. Furthermore, each vector is also small in size, where $|L_v| = d$ and $|L_e| = |I_v| =$

$d-1$. Based on these observations, we devise a lightweight compressing technique that can represent each vector as an integer. Let L_e^{max} be the maximum element in L_e. Using the numeration base $b = L_e^{max} + 1$, we represent L_e in an integer

$$\widehat{L_e} = \sum_{i=0}^{d-1} l(v_i, v_{e(i)}) \times b^i. \quad (4)$$

I_v^{max}, L_v^{max}, $\widehat{I_v}$ and $\widehat{L_v}$ are similarly defined. A labeling of T is uniquely determined by the values:

$$|V|, I_v^{max}, L_v^{max}, L_e^{max}, \widehat{I_v}, \widehat{L_v}, \widehat{L_e}.$$

To decompress $\widehat{L_e}$, we use the *base conversion algorithm* [22] to reconstruct L_e. Maintaining a dynamic vector $\mathcal{L}$, we divide $\widehat{L_e}$ by b with a remainder r and append r to the end of $\mathcal{L}$. The process repeats until $\widehat{L_e}$ is zero. If $|\mathcal{L}| < |L_e| = d-1$, we append $d - 1 - |\mathcal{L}|$ zeros to $\mathcal{L}$. L_v and I_v are decoded similarly.

Extension to non-tree edges. Consider graph $G = (V, E, L, l)$ and one of its spanning trees $T = (V, E', L, l)$. A non-tree edge is denoted as a triple $\langle i, j, l(v_i, v_j)\rangle$, where $(v_i, v_j) \in E \setminus E'$ and $l(v_i, v_j)$ is the label on the edge. We order v_i and v_j in the triple such that $i > j$. Hence, the set of non-tree edges of G can be denoted as

$$\widetilde{T} = \{\langle i, j, l(v_i, v_j)\rangle \mid (v_i, v_j) \in E \setminus E' \wedge i > j\}.$$

To compress $\widetilde{T}$, we extend the vectors of T to accommodate the non-tree edges. Specifically, we append all vertex labels of each element in $\widetilde{T}$ to I_v. Similarly, we append each label in $\widetilde{T}$ to L_e. Denote the newly obtained sets by $\widehat{I_v}$ and $\widehat{L_e}$, respectively. We have that $|L_e| = |E|$ and $|I_e| = 2|E|-|V|+1$. Eq. (4) still holds, by replacing I_v and L_e with $\widehat{I_v}$ and $\widehat{L_e}$, respectively.

Therefore, the graph G can be represented as

$$\mathcal{C}(G) = \left\{|V|, |E|, I_v^{max}, L_v^{max}, L_e^{max}, \widehat{I_v}, \widehat{L_v}, \widehat{L_e}\right\}. \quad (5)$$

In case that the size of $\widehat{I_v}$, $\widehat{L_v}$, or $\widehat{L_e}$ is larger than the size of a machine word, we can use an array of machine words to store each value, where the size of each array is $\lceil \log_w \widehat{I_v} \rceil$, $\lceil \log_w \widehat{L_v} \rceil$, and $\lceil \log_w \widehat{L_e} \rceil$, respectively.

B. Theoretical Analysis

Correctness analysis. The correctness condition consists of two parts: (i) the compression result of canonical labeling must be unique, and (ii) the decompressing technique must correctly reconstruct the canonical labeling of a graph. The first part ensures that aggregating graphs in the filter phase is correct, whereas the second part guarantees the correctness in the refinement phase.

Lemma 1: Given the canonical labeling of a graph G, $\mathcal{C}(G)$ in Eq. (5) is unique.

Proof: Given the canonical labeling of G, it can be equivalently represented by three vectors I_v, L_v, and L_e. Since $\mathcal{C}(G)$ is the combination of the compression of I_v, L_v, and L_e, we are to prove that the compression of each vector is unique. Consider the vector I_v: we compress I_v as three values $|V|$, $b = I_v^{max} + 1$, and $\widehat{I_v}$. By contradiction, assume that there exist two different vectors $I_v = (v_0, v_1, \cdots, v_{d-1})$ and $I_u = (u_0, u_1, \cdots, u_{d-1})$ with the same compressed image, that is, for some $i \in [0, d-1]$, we have $I_v[i] \neq I_u[i]$. According to Eq. (4), $\widehat{I_v} \neq \widehat{I_u}$, which contradicts the assumption that I_v and I_u have the same compressed image. Therefore, the canonical labeling of a graph G is unique. ■

Lemma 2: Given $\mathcal{C}(G)$ of a graph G, the decompression of $\mathcal{C}(G)$ is the canonical labeling of G.

Proof: Given $\mathcal{C}(G)$ in Eq. (5) after reconstruction, the sizes of I_v, L_v, and L_e are $|V|$, $2|E| - |V| + 1$, and $|E|$, respectively. Let us prove the case of one vector, the same proof can be applied to the other ones. Consider I_v: by contradiction, assume that the constructed vector I'_v is different from I_v. Since the base and size of I'_v and I_v are the same, there exists an $i \in [0, d-1]$ such that $I'_v[i] \neq I_v[i]$. According to Eq. (4), $\widehat{I'_v} \neq \widehat{I_v}$, which contradicts that I'_v is reconstructed from $\widehat{I_v}$. Consequently, the decompression of $\mathcal{C}(G)$ is the canonical labeling of G. ■

Complexity analysis. Assume that each element in I_v, L_v, and L_e is stored in a machine word. Therefore, the total size of these three vectors is

$$(2|E| - |V| + 1 + |V| + |E|) \times w = (3|E| + 1)w.$$

Now, consider the size of the compressed image: according to Eq. (4), we have

$$\widehat{I_v} \leq (I_v^{max} + 1)^{|V|},$$
$$\widehat{L_v} \leq (L_v^{max} + 1)^{2|E|-|V|+1},$$
$$\text{and } \widehat{L_e} \leq (L_e^{max} + 1)^{|E|}.$$

Let $b = \max\{I_v^{max} + 1, L_v^{max} + 1, L_e^{max} + 1\}$. The labels in L can be mapped to a consecutive range $[0, |L|)$, and assume that $O(|V|) = O(|L|)$ for moderate-sized graphs, hence we have $b = O(|L|)$. As such, the total size of the compressed image is at most

$$(|V| + 2|E| - |V| + 1 + |E|) \times \lceil \log_w b \rceil$$
$$= (3|E| + 1)\lceil \log_w b \rceil = O(|E| \log_w |L|).$$

The compression ratio is at least $w/\lceil \log_w b \rceil$.

As to the computational complexity of compression, let us first consider the computation of L_e. Since the size of $\widehat{L_e}$ is at most $|E| \log_w |L|$, and there are $|E|$ elements in L_e, the complexity of computing $\widehat{L_e}$ is $O(|E|^2 \log_w |L|)$. A similar analysis applies to I_v and L_v, whose upper-bound is also $(|E|^2 \log_w |L|)$. That is, the complexity of compression is $O(|E|^2 \log_w |L|)$. Similarly, the complexity of decompression is also $O(|E|^2 \log_w |L|)$.

VII. Experimental Evaluation

We evaluate experimentally the proposed algorithm *MRFSM* (MapReduce Frequent Subgraph Mining) against two baselines on several real-world large graph datasets. In Section VII-A

TABLE II
TABLE OF DATASETS

dataset	number of graphs	disk size	$\overline{\lvert E\rvert}$	$\overline{\lvert V\rvert}$
Pubchem [7]	46,703,496	41.8 GB	52.32	50.37
Akos [1]	15,720,753	6.6 GB	24.00	22.52
ChemDB [3]	7,100,106	6.7 GB	53.23	50.77
Ambinter [2]	6,551,088	3.4 GB	29.40	27.07
Enamine [4]	1,378,907	0.7 GB	27.86	25.78
NCI [6]	265,242	0.2 GB	41.76	40.47

we describe the experimental setup, followed by an analysis of how to tune the value of probability threshold ρ in Section VII-B. Section VII-C presents the head-to-head comparison with competitor techniques.

A. Experimental Setup

We deploy a *MRFSM* prototype in Amazon EC2[1] using up to 121 large instances. Each instance has 2 CPUs and 7.5GB RAM and runs Hadoop (version 0.20.203). One instance is set up as the master node and the others as worker nodes. We use the default configuration of Hadoop, i.e., $dfs.replication = 3$ and $fs.block.size = 64$MB.

Implementation. We use a public C++ implementation of *GASTON* [32] to mine the locally frequent subgraphs in the filter round (we emphasize that our framework can be used in conjunction with any other frequent subgraph mining algorithm, not only GASTON). We implement *MRFSM* using *Hadoop Streaming API*[2], which places no restrictions on the programming language.

As competitors, we consider the *Iterative Frequent Subgraph Mining* (*IFSM*) [15] algorithm, with source code made available by the authors. We also adapt the partitioning-based method from [31] (*PGM*) to run on MapReduce. Note that, since MapReduce is a share-nothing architecture, the sequential data partition method in *PGM* is performed on a single machine, and the result is fed to the map phase of the filter round.

Datasets. We use seven real-world datasets listed in Table II. We use as default dataset $Pubchem$, the largest dataset which consists of more than 46 million graphs. The average number of edges and vertices are 52.32 and 50.37, respectively. Note that, none of these datasets (except *NCI*) can be handled by a single machine with up to 12GB RAM. For each dataset, given the set L of all its labels, we assign each label $l \in L$ a value $i \in [0, |L|)$, such that (i) each label l is injectively mapped to a value in the range $[0, |L|)$, and (ii) the more frequent label gets the smaller value.

Methodology. We perform several types of measurements: first, we record the total running time (wall clock) of each algorithm, i.e., the time elapsed from starting the MapReduce program until the program returns the result. Second, we report the running time of the filter and refinement rounds independently. Third, we report the communication cost of transferring data in MapReduce, as well as the number of candidate graphs generated in the filter round. We run each experiment three times, and report the average reading.

[1]http://aws.amazon.com/ec2

[2]http://hadoop.apache.org/docs/stable/streaming.html

TABLE III
EXPERIMENTAL PARAMETER VALUES

parameter	values
θ	0.025, 0.05, **0.1**, 0.2, 0.4
m	80, 90, **100**, 110, 120
ρ	0, 0.00625, 0.0125, **0.05**, 0.1, 0.2, 0.4, 0.8, 1

The parameter values used are shown in Table III, with default values in bold.

B. Tuning Probability Upper-Bound ρ

Recall from Section IV-B that the probability ρ has an important role in pruning the search space. We show next how to tune the value of ρ in practice. Figure 5 shows performance results when varying ρ from 0 to 1. When $\rho = 0$, each worker outputs all the subgraphs that are visited in the filter round, whereas when $\rho = 1$, *MRFSM* only outputs the locally frequent subgraphs.

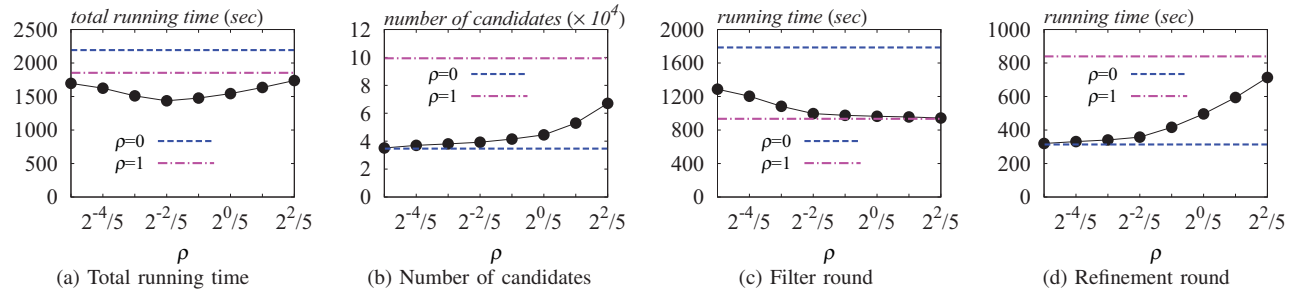
Figure 5(a) shows the total running time of *MRFSM*. When ρ grows from 0 to 0.05, the total running time decreases, as the total number of subgraphs that are generated by each worker decreases, and the overhead of the filter round is lower (Figure 5(c)). On the other hand, when ρ increases further towards 1, the total running time increases, because the number of candidates increases (Figure 5(b)), and so does the overhead in the refinement round (as shown in Figure 5(d)). A clear trade-off can be observed between the costs of the filter and refinement rounds: for smaller ρ, the number of subgraphs that are generated in each worker is larger, but the number of candidates sent to the refinement round is smaller. A good setting for the probability threshold is $\rho = 0.05$, when the overhead is minimized. In the rest of the experiments, we set $\rho = 0.05$.

C. Comparison with competitor techniques

We compare *MRFSM* against *PGM* and *IFSM*. First, we vary support threshold θ from 0.025 to 0.4. Figures 6(a)(b) show that *IFSM* does not terminate within reasonable time (one day) when $\theta \leq 0.05$. Furthermore, *PGM* does not terminate when $\theta = 0.025$, owing to the significant computational overhead in the refinement round (Figure 6(d)). Even though the filter time of *MRFSM* is slightly higher than that of *PGM*, due to producing locally infrequent subgraphs, this cost is offset by the significant improvement in the refinement round. *MRFSM* significantly outperforms the other algorithms in terms of total running time. Due to its poor performance, we no longer consider *IFSM* in the rest of the section.

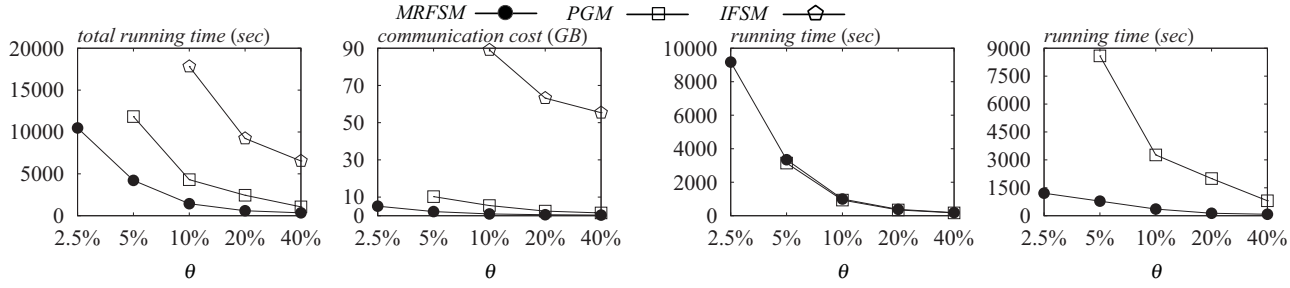
Next, we evaluate the impact of number of available workers, and we vary m from 80 to 120. Figure 7(a) demonstrates that *MRFSM* clearly outperforms *PGM* in terms of total running time for all settings. Furthermore, communication cost increases only slightly as m grows.

We also evaluate the performance of the graph compression technique proposed in Section VI. We denote the *MRFSM*



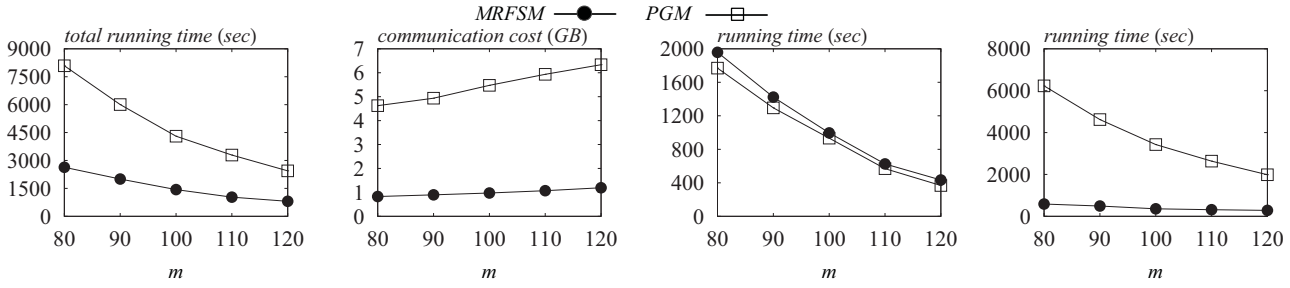
(a) Total running time (b) Number of candidates (c) Filter round (d) Refinement round

Fig. 5. Varying probability threshold ρ



(a) Total running time (b) Communication cost (c) Filter round (d) Refinement round

Fig. 6. Varying support threshold θ



(a) Total running time (b) Communication cost (c) Filter round (d) Refinement round

Fig. 7. Varying number of workers m

approach that uses compression as *MRFSM-C*, and the one without compression as *MRFSM-NC*. Figure 8(a) shows that the total running time is improved by compression. Figure 8(b) shows the size of data communicated among workers. *MRFSM-C* has much smaller cost in communication, which demonstrates the effectiveness of our compression technique. As expected, *MRFSM-NC* has slightly higher communication cost than *PGM*, since in the filter round, *MRFSM-NC* also outputs the locally infrequent subgraphs on each machine, while *PGM* only outputs the locally frequent subgraphs. Figure 8(c) shows the breakdown of the filter running time into communication time and CPU time. *MRFSM-C* has slightly higher CPU cost than *MRFSM-NC* because of encoding/decoding, but this is offset by the significant gain due to the reduction in communication time. Figure 8(d) compares the running time for the refinement round: *MRFSM-C* is again slightly worse than *MRFSM-NC*, owing to the overhead of decompression.

Next, we evaluate the Top-Down and Bottom-Up traversal strategies employed in the refinement round. Figure 9(a) presents the running time of Top-Down and Bottom-Up with support threshold θ varying from 0.025 to 0.4. Top-Down is slower than Bottom-Up, since the number of *subgraph isomorphism tests* is larger, as shown in Figure 9(b). However,

TABLE IV

TABLE OF RUNNING TIME ($\theta = 0.1$)

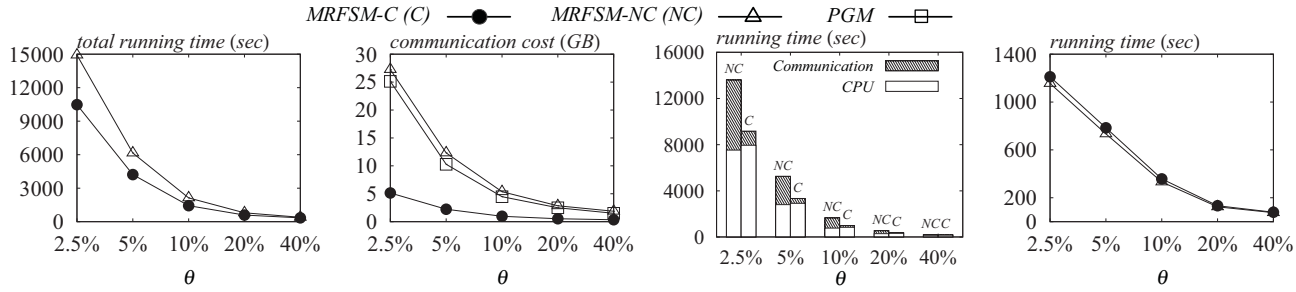
dataset	*MRFSM*	*PGM*	*IFSM*	improved ratio
Pubchem	1436.02	4303.41	11818.49	**3.00**
Akos	284.37	1025.14	1946.84	**3.60**
ChemDB	376.42	2197.56	4674.77	**5.84**
Ambinter	235.19	926.17	1298.07	**3.94**
Enamine	227.34	734.69	993.24	**3.23**
NCI	142.81	293.27	413.54	**2.05**

Bottom-Up needs to maintain all candidates in memory while Top-Down releases the memory once the candidates are no longer useful. As shown in Figure 9(c), Bottom-Up is more memory-efficient than Top-Down.

Finally, we present the total running time results on all considered real-world datasets in Table IV. *MRFSM* outperforms competitors by more than two times on all datasets (improved ratio compared to closest competitor is shown in the last column). In some cases, the improvement is close to six-fold.

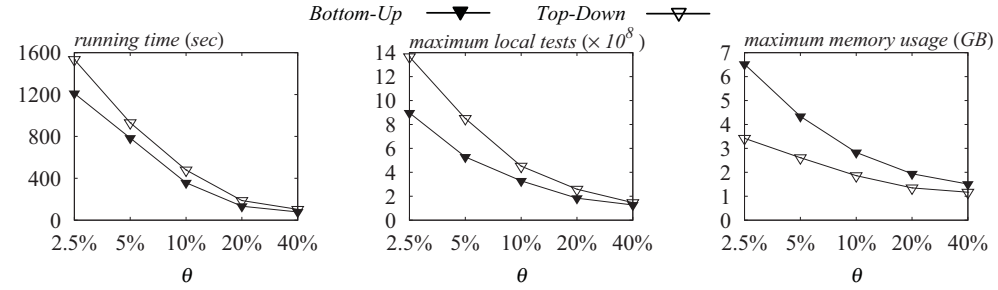
VIII. RELATED WORK

The problem of frequent subgraph mining has been studied for more than a decade. However, how to overcome the rapid



(a) Total running time (b) Communication cost (c) Filter round (d) Refinement round

Fig. 8. Impact of compression



(a) Running time (b) Number of isomorphism tests (c) Memory usage

Fig. 9. Strategy comparison in refinement round

growth of graph data remains open. In the following, we first discuss the existing work on frequent subgraph mining in transaction setting, and then discuss the parallel techniques in data mining, especially the techniques in MapReduce.

In an early work [18], Inokuchi et al. propose an *Apriori Graph Mining* (*AGM*) algorithm, which generates graph candidates by adding a vertex at a time, and computes the frequency of each candidate by scanning the database. As an improvement over *AGM*, Kuramochi et al. [23] proposed the *FSG* technique which utilizes edge-growth mining and computes the frequency of candidates based on the frequent subgraphs that have been mined already. All the aforementioned approaches employ breadth-first search, i.e., they first compute all k-size frequent subgraphs, based on which the $(k+1)$-size frequent subgraphs are computed next. In contrast, several other approaches [9], [16], [29], [32], [40] are based on depth-first search. Among these, Yan et al. [40] develop *gSpan* which uses a novel canonical graph representation to facilitate the pruning of the search space. Later on, Huan et al. [16] employ another graph representation to reduce the overhead in subgraph isomorphism testing. Nijssen et al. [32] proposed *GASTON* which categorizes the graphs into paths, trees, and cyclic graphs, and develops techniques for each category to speed up the running time. A follow-up work by Maunz et al. [29] gives more insight into the categorization of graphs for accelerating the mining process.

All the solutions discussed so far are in-memory algorithms that load the entire dataset into main memory. Such techniques cannot deal with datasets that are larger than the main memory size. To address this problem, Wang et al. [37] proposed *ADI-Mine*, an approach to facilitate frequent subgraph mining from a disk-based graph database. Different from *ADI-Mine*, Nguyen et al. [31] propose a data partition approach, which was earlier introduced for frequent itemset mining [28], [34]. In this approach, the entire dataset is partitioned into several portions in the first scan, and each portion is independently processed to obtain a set of candidates whose frequency is computed in a second scan of the dataset.

Due to the computation and I/O intensive characteristic of data mining problems, more and more efforts are geared towards solving it with the aid of parallel techniques. Cheung et al. [11] developed an approach for mining association rules in a distributed system. In that approach, several iterations are performed, and each iteration requires a broadcast of locally frequent itemsets to all machines. In a multi-core system, Li et al. [26] proposed a parallelizable FP-Growth [14] method, which partitions the database based on the frequent items. More recently, Li et al. [27] propose to parallelize FP-Growth in MapReduce with the aim of finding the k most frequent itemsets. In the area of parallel frequent sequence mining, Miliaraki el at. [30] describe *MG-FSM* which is built on MapReduce and follows the ideas of *projected database*.

In the setting of parallel frequent subgraph mining, several works [33], [38] are proposed in the context of a single large graph. For the transaction setting, *SUBDUE* [12] describes a shared-memory parallel approach by partitioning tasks, and a data partitioning approach that computes the locally frequent subgraphs and broadcasts them to all machines. Frequencies are then computed in each machine, and finally a single master node aggregates the results from each machine. The data partitioning method of *SUBDUE* thus interleaves parallel and sequential computing. Buehrer el at. [10] proposed parallel frequent subgraph mining in a multi-core system, which partitions the tasks among multiple shared-memory processors. Recently, Hill el at. [15] used MapReduce for mining frequent subgraphs, which iteratively grows the searched pattern size in each round of the MapReduce job. Further details on parallel data mining can be found in the excellent survey from [41].

MapReduce [13], [35] has established itself as the candidate of choice in *big data* problems. To address the challenges that arise in large-scale graph mining, several techniques and systems have been proposed. *PEGASUS* [21] is a system based on MapReduce for graph pattern mining and graph analysis tasks in a large graph, e.g., computing the diameter [20] and counting triangles [36]. To count or enumerate the subgraphs in a large graph with MapReduce, Zhao el at. [42] proposed a *color coding* based approach, while Afrati el at. [8] devised a multi-way join method by decomposing the given graph. Xiang el at. [39] employ MapReduce to mine the maximum cliques from a large graph using a coloring based partitioning method. However, these techniques and systems cannot solve the frequent subgraph mining problem in the transaction setting, which is the focus of our work.

None of the existing work has a satisfactory solution to frequent subgraph mining in the transaction setting with MapReduce. The previous approach from [31] also uses data partitioning, but it is not able to efficiently process a large-scale dataset, as shown in our experiments. In addition, prior approaches are all evaluated on small-scale or synthetic datasets. In contrast, our technique can handle tens of millions of real-world graphs in a moderate-sized cloud environment.

IX. Conclusions

We proposed a two-step filter-and-refinement MapReduce framework for frequent subgraph mining in the transaction graph setting. Our method scales well for large datasets, and significantly outperforms competitor techniques. The gain in performance is achieved through careful selection of frequent subgraph candidates, effective strategies for avoiding redundant computation in the refinement step, and a lightweight compression scheme that reduces significant communication cost with low computational overhead of encoding/decoding. In future work, we plan to improve further the technique for candidate selection, in order to improve accuracy of frequency estimation. We also plan to investigate how to extend some of our results to the single-large-graph mining scenario.

Acknowledgement

Xiaokui Xiao was supported by the Nanyang Technological University under SUG Grant M58020016, and by Microsoft Research Asia under an Urban Informatics Research Grant. Gabriel Ghinita was supported by NSF award CNS-1111512.

References

[1] Akos. http://www.akosgmbh.de.
[2] Ambinter. http://www.ambinter.com.
[3] Chemdb. http://cdb.ics.uci.edu.
[4] Enamine. http://www.enamine.net.
[5] Hadoop. http://hadoop.apache.org.
[6] Nci. http://cactus.nci.nih.gov/download/nci.
[7] Pubchem. http://pubchem.ncbi.nlm.nih.gov.
[8] F. N. Afrati, D. Fotakis, and J. D. Ullman. Enumerating subgraph instances using map-reduce. In *ICDE*, pages 62–73, 2013.
[9] C. Borgelt and M. R. Berthold. Mining molecular fragments: Finding relevant substructures of molecules. In *ICDM*, pages 51–58, 2002.
[10] G. Buehrer, S. Parthasarathy, and Y.-K. Chen. Adaptive parallel graph mining for cmp architectures. In *ICDM*, pages 97–106, 2006.
[11] D. W.-L. Cheung, J. Han, V. T. Y. Ng, A. W.-C. Fu, and Y. Fu. A fast distributed algorithm for mining association rules. In *PDIS*, pages 31–42, 1996.
[12] D. J. Cook, L. B. Holder, G. Galal, and R. Maglothin. Approaches to parallel graph-based knowledge discovery. *J. Parallel Distrib. Comput.*, 61(3):427–446, 2001.
[13] J. Dean and S. Ghemawat. Mapreduce: Simplified data processing on large clusters. In *OSDI*, pages 137–150, 2004.
[14] J. Han, J. Pei, and Y. Yin. Mining frequent patterns without candidate generation. In *SIGMOD Conference*, pages 1–12, 2000.
[15] S. Hill, B. Srichandan, and R. Sunderraman. An iterative mapreduce approach to frequent subgraph mining in biological datasets. In *BCB*, pages 661–666, 2012.
[16] J. Huan, W. Wang, and J. Prins. Efficient mining of frequent subgraphs in the presence of isomorphism. In *ICDM*, pages 549–552, 2003.
[17] J. Huan, W. Wang, J. Prins, and J. Yang. Spin: mining maximal frequent subgraphs from graph databases. In *KDD*, pages 581–586, 2004.
[18] A. Inokuchi, T. Washio, and H. Motoda. An apriori-based algorithm for mining frequent substructures from graph data. In *PKDD*, pages 13–23, 2000.
[19] X. Jiang, H. Xiong, C. Wang, and A.-H. Tan. Mining globally distributed frequent subgraphs in a single labeled graph. *Data Knowl. Eng.*, 68(10):1034–1058, 2009.
[20] U. Kang, C. E. Tsourakakis, A. P. Appel, C. Faloutsos, and J. Leskovec. Hadi: Mining radii of large graphs. *TKDD*, 5(2):8, 2011.
[21] U. Kang, C. E. Tsourakakis, and C. Faloutsos. Pegasus: A peta-scale graph mining system. In *ICDM*, pages 229–238, 2009.
[22] H. Kettani. On the conversion between number systems. In *MSV/AMCS*, pages 317–320, 2004.
[23] M. Kuramochi and G. Karypis. Frequent subgraph discovery. In *ICDM*, pages 313–320, 2001.
[24] M. Kuramochi and G. Karypis. Finding frequent patterns in a large sparse graph. In *SDM*, 2004.
[25] M. Kuramochi and G. Karypis. Grew-a scalable frequent subgraph discovery algorithm. In *ICDM*, pages 439–442, 2004.
[26] E. Li and L. Liu. Optimization of frequent itemset mining on multiple-core processor. In *VLDB*, pages 1275–1285, 2007.
[27] H. Li, Y. Wang, D. Zhang, M. Zhang, and E. Y. Chang. Pfp: parallel fp-growth for query recommendation. In *RecSys*, pages 107–114, 2008.
[28] J.-L. Lin and M. H. Dunham. Mining association rules: Anti-skew algorithms. In *ICDE*, pages 486–493, 1998.
[29] A. Maunz, C. Helma, and S. Kramer. Large-scale graph mining using backbone refinement classes. In *KDD*, pages 617–626, 2009.
[30] I. Miliaraki, K. Berberich, R. Gemulla, and S. Zoupanos. Mind the gap: large-scale frequent sequence mining. In *SIGMOD Conference*, pages 797–808, 2013.
[31] S. N. Nguyen, M. E. Orlowska, and X. Li. Graph mining based on a data partitioning approach. In *ADC*, pages 31–37, 2008.
[32] S. Nijssen and J. N. Kok. A quickstart in frequent structure mining can make a difference. In *KDD*, pages 647–652, 2004.
[33] S. Reinhardt and G. Karypis. A multi-level parallel implementation of a program for finding frequent patterns in a large sparse graph. In *IPDPS*, pages 1–8, 2007.
[34] A. Savasere, E. Omiecinski, and S. B. Navathe. An efficient algorithm for mining association rules in large databases. In *VLDB*, pages 432–444, 1995.
[35] Y. Tao, W. Lin, and X. Xiao. Minimal mapreduce algorithms. In *SIGMOD Conference*, pages 529–540, 2013.
[36] C. E. Tsourakakis, U. Kang, G. L. Miller, and C. Faloutsos. Doulion: counting triangles in massive graphs with a coin. In *KDD*, pages 837–846, 2009.
[37] C. Wang, W. Wang, J. Pei, Y. Zhu, and B. Shi. Scalable mining of large disk-based graph databases. In *KDD*, pages 316–325, 2004.
[38] B. Wu and Y. Bai. An efficient distributed subgraph mining algorithm in extreme large graphs. In *AICI (1)*, pages 107–115, 2010.
[39] J. Xiang, C. Guo, and A. Aboulnaga. Scalable maximum clique computation using mapreduce. In *ICDE*, pages 74–85, 2013.
[40] X. Yan and J. Han. gspan: Graph-based substructure pattern mining. In *ICDM*, pages 721–724, 2002.
[41] M. J. Zaki. Parallel and distributed data mining: An introduction. In *Large-Scale Parallel Data Mining*, pages 1–23, 1999.
[42] Z. Zhao, G. Wang, A. R. Butt, M. Khan, V. S. A. Kumar, and M. V. Marathe. Sahad: Subgraph analysis in massive networks using hadoop. In *IPDPS*, pages 390–401, 2012.