Knowl Inf Syst (2018) 56:663–690
https://doi.org/10.1007/s10115-017-1104-7

REGULAR PAPER

Mining frequent subgraphs from tremendous amount of small graphs using MapReduce

Zhe Peng[1] · Tongtong Wang[1] · Wei Lu[1] · Hao Huang[2] · Xiaoyong Du[1] · Feng Zhao[3] · Anthony K. H. Tung[3]

Received: 28 July 2016 / Revised: 15 May 2017 / Accepted: 20 August 2017 /
Published online: 6 October 2017

Abstract Frequent subgraph mining from a tremendous amount of small graphs is a primitive operation for many data mining applications. Existing approaches mainly focus on centralized systems and suffer from the scalability issue. Consider the increasing volume of graph data and mining frequent subgraphs is a memory-intensive task, it is difficult to tackle this problem on a centralized machine efficiently. In this paper, we therefore propose an efficient and scalable solution, called MRFSE, using MapReduce. MRFSE adopts the breadth-first search strategy to iteratively extract frequent subgraphs, i.e., all frequent subgraphs with $i+1$ edges are generated based on frequent subgraphs with i edges at the ith iteration. In our design, existing frequent subgraph mining techniques in centralized systems can be easily extended and integrated. More importantly, new frequent subgraphs are generated without performing any *isomorphism test* which is costly and imperative in existing frequent subgraph mining techniques. Besides, various optimization techniques are proposed to further reduce the communication and I/O cost. Extensive experiments conducted on our in-house clusters demonstrate the superiority of our proposed solution in terms of both scalability and efficiency.

Keywords Frequent subgraph mining · MapReduce · Isomorphism-testing-free

1 Introduction

Graphs are ubiquitous and explored in a wide spectrum of applications such as cheminformatics [20], bioinformatics [2,14], and object recognition in image processing [18,22]. As

✉ Wei Lu
lu-wei@ruc.edu.cn

1 School of Information and Key Lab of Data Engineering and Knowledge Engineering, MOE, Renmin University of China, Beijing, China

2 State Key Laboratory of Software Engineering, Wuhan University, Wuhan, China

3 School of Computing, National University of Singapore, Singapore, Singapore

a fundamental data structure, there have been extensive studies on graphs and their properties. One of the most important research topics for graphs is frequent subgraph mining, which returns all subgraphs that are contained in at least T graphs from a tremendous amount of small graphs, where T is a user-defined support. Frequent subgraph mining is a primitive operation for a variety of data mining applications, such as extracting frequent patterns from chemical compounds, identifying the relationship between chemical compounds, and building graph indexes to facilitate subgraph containment searches.

Existing frequent subgraph mining approaches in centralized systems can be classified into two categories: Apriori-based approaches [13,15] and pattern-growth approaches [4,11,12,21,24]. Both Apriori-based and pattern-growth approaches follow the generation-and-verification paradigm to mine frequent subgraphs. Apriori-based approaches iteratively extract the frequent subgraphs in a breath first search fashion. New frequent subgraphs with size i edges (or vertices) are generated by frequent subgraphs with size $i - 1$ edges (or vertices). Pattern-growth approaches extract the frequent subgraphs in a depth-first search (DFS) fashion. New frequent subgraphs with size i edges[1] are enumerated by appending one edge to a frequent $(i - 1)$-subgraph. Since a new subgraph can be generated by multiple subgraphs (pattern-growth approaches) or subgraph pairs (Apriori-based approaches), various optimization techniques in existing approaches are proposed to reduce the generation spaces. Besides, they also focus on designing effective labeling techniques. In this way, new generated subgraphs with the same label are considered to be isomorphic, and hence, frequent subgraphs can be verified by aggregating and counting the number of graphs that contain them.

Existing frequent subgraph mining approaches are *memory-intensive*. Considering the AIDS Antiviral Screen[2] dataset consisting of only 43,905 chemical compounds, these pattern-growth approaches including gSpan [24], Mofa [4], FFSM [11,12] and Gaston [21] consume 300, 600 MB, 1.2 and 1.3 GB memory usage, respectively, when the frequent threshold is set to 5% of the cardinality of the dataset [9]. Consider the forever increasing data size. For example, SCIFinder,[3] which provides the world's largest collection of chemistry and related science information, reports that about 4000 new compound structures are added each day. Obviously, it is not appropriate to employ these in-memory pattern-growth approaches, and somehow difficult to perform frequent subgraph mining on a centralized machine efficiently.

MapReduce [8] was recently proposed as a programming model for processing data-intensive applications in a distributed computing environment with clusters of computers. It has gained wide acceptance and been applied in various application domains due to its simplicity, flexibility, fault tolerance and scalability. It is therefore a natural choice platform for extracting frequent subgraphs over large-scale graph datasets. In this paper, we study how to mine frequent subgraphs using MapReduce. We propose a robust, efficient and scalable solution, called MRFSE. MRFSE mines frequent subgraphs iteratively, i.e., MRFSE mines all frequent i-subgraphs at the ith iteration $(i \geq 1)$. Execution of MRFSE at each iteration follows the generation-and-verification paradigm and employs a single MapReduce job to mine frequent i-subgraphs. The *mappers* generate a superset of frequent i-subgraphs based on frequent $i - 1$-subgraphs obtained in the previous round $(i > 1)$. The *reducers* verify each candidate in the superset and output all frequent i-subgraphs, which are then taken as the input of next MapReduce Job. We iterate until no new frequent subgraphs can be found.

[1] In the remainder of this paper, an i-subgraph is referred to as a subgraph with i edges.

[2] http://dtp.nci.nih.gov/docs/aids/aids_data.html.

[3] http://www.cas.org.

As discussed before, existing frequent subgraph mining approaches are memory-intensive. To tackle this problem, we do not process the whole dataset in main memory simultaneously. Instead, we process every graph g separately, from which we mine every possible frequent subgraph that g contains. Specifically, the whole dataset D is first divided into m disjoint subsets and each subset D_i is sent to a single machine M_i. During iterations, M_i is responsible for mining frequent subgraphs from graphs in D_i unchangeably. In our design, each graph g is represented as a set of key/value pairs. These pairs are considered as a whole and taken as the input of map function. The key of the pair is the label of a frequent i-subgraph that g contains. The value of the pair consists of a set of mappings that record every subgraph in g that is graph isomorphic to the key. In this way, at the ith iteration, all possible frequent $(i-1)$-subgraphs that g contains and their corresponding mappings are maintained. We coordinate machines to mine all size-i frequent subgraphs using a single MapReduce Job. At the ith iteration, mappers in M_i issue parallel scan over graphs in D_i. The map function takes each g as the input. Existing Apriori-based and pattern-growth approaches can be easily extended to update g by replacing $(i-1)$-subgraphs with i-subgraphs associated with corresponding mappings. Besides, labels of new generated i-subgraphs are shuffled to reducers. Reducers verify and output frequent i-subgraphs by aggregating the same labels of subgraphs.

In this paper, we make the following contributions:

- We propose a generic solution, called MRFSE, to iteratively mine frequent subgraphs. Existing techniques can be easily extended and integrated into our solution.
- MRFSE mines frequent subgraphs over each graph in the dataset separately. This mechanism brings three benefits. First, new frequent subgraphs are generated without performing graph isomorphism test which is imperative and expensive in existing approaches. Second, it is unnecessary to shuffle mappings from mappers to reducers. Consider that often these mappings are prohibitively large. MRFSE is able to save a large amount of network bandwidth. Third, as processing each graph separately, compared with existing approaches, MRFSE consumes less memory cost.
- We conduct extensive experiments on our in-house clusters to demonstrate the superiority of our proposed solution in terms of both scalability and efficiency. The preliminary study of this work is reported in [19].

The remainder of the paper is organized as follows. Section 2 discusses related work. Section 3 introduces the necessary background knowledge and provides the problem definition. Section 4 describes an overview of the solution. Section 5 presents the extension of existing techniques to generate new subgraphs in our solution. Section 6 reports the experimental results, and Sect. 7 concludes the paper.

2 Related work

Frequent subgraph mining is a well-studied research area in the centralized systems. Typical Apriori-based algorithms include AGM [13] and FSG [15]. In AGM, all frequent subgraphs with $i+1$ vertices are generated by two frequent subgraphs with i vertices that share a common subgraph with $i-1$ vertices. While in FSG, frequent subgraphs with $i+1$ edges are generated by two frequent subgraphs with i edges that share a common subgraph with $i-1$ edges. Instead of following the breadth-first mechanism in Apriori-based algorithms, pattern-growth algorithms generate frequent subgraphs following the depth-first mechanism. At each iteration, they first enumerate all frequent i-subgraph candidates by appending one edge in frequent 1-subgraphs to all frequent $(i-1)$-subgraphs, then remove duplicated candidates by performing graph isomorphism test operation, and finally verify the frequent i-subgraphs by collecting the number of graphs that contain them. The most important work in this direction is gSpan [24]. gSpan builds a lexicographic order (termed as DFS code) among subgraphs. Based on this order, first, new subgraphs are generated by its parent subgraph with the minimum DFS code; second, generation of new subgraphs are restricted based on a certain rule so as to reduce the enumeration space. For this reason, gSpan mines frequent subgraphs without generating candidates. Although other pattern-growth algorithms, such as MOFA [4], FFSM [11], SPIN [12] and GASTON [21], propose various labeling techniques and optimization strategies to reduce the enumeration space of mining frequent subgraphs, gSpan remains as one of the most widely used approaches to extracting frequent subgraphs. For example, gSpan is popularized to build graph indices in gIndex [25], FG-Index [7] and FG*-Index [6]. In our work, we focus on the extension of gSpan while other techniques can be similarly adapted in our solution as well.

Recently, there is an increasing interest on designing new approaches to mining frequent subgraphs using MapReduce. In this field, the state-of-the-art approaches follow the generation-and-verification paradigm to mine frequent subgraphs. Bhuiyan and Hasan [3], Hill et al. [10] , Liu et al. [17] and Lu et al. [19] mine frequent subgraphs iteratively, and at the ith iteration, all frequent i-subgraphs are extracted ($i > 1$). Hill et al. [10] and Liu et al. [17] issue two MapReduce jobs at each iteration. In the first MapReduce job, for every graph g of the dataset, the *mappers* shuffle all frequent $(i-1)$-subgraphs that g contains and corresponding mappings to *reducers*, and *reducers* generate frequent i-subgraph candidates of g. In the second MapReduce job, all i-subgraphs from different *mappers* that are graph isomorphic to each others are aggregated in *reducers* and all frequent i-subgraphs are extracted. Hill et al. [10] and Liu et al. [17] are inefficient mainly due to three reasons. First, no rules are adopted to restrict the generation of new subgraphs, leading to an exponential increase in the size of the generation space. Second, the size of mappings could be prohibitively large, especially in a large-scale dataset, and shuffling these mappings causes an expensive communication cost. Third, due to the lack of unified lexicographic order of subgraphs, it is necessary to perform graph isomorphism test for subgraphs when aggregating the occurrence of isomorphic subgraphs. Bhuiyan and Hasan [3] and Lu et al. [19] utilize the techniques in gSpan to overcome the first and third limitations of [10]. First, each subgraph is represented as a label, which is termed as minimum DFS code, and two subgraphs with the same minimum DFS code mean they are isomorphic. For this reason, [1,3,19] collect all isomorphic subgraphs by simply aggregating the same minimum DFS code, and hence avoid expensive isomorphism test cost. Second, subgraphs are ordered based on their corresponding minimum DFS codes and new subgraphs are generated merely from its parent graph with the minimum DFS code, hence reducing the size of the generation space. The main difference between [19] and [3] is the way, i.e., following the bottom-up and top-down paradigm, respectively, to generate the frequent subgraph candidates. By well organizing necessary information used in gSpan, [3] enumerates frequent i-subgraph candidates by appending every possible frequent 1-subgraph to frequent $(i-1)$-subgraphs, removes duplicated candidates by performing graph isomorphism test, and verifies every frequent i-subgraph s by counting the number of graphs that contain s. While [19] generates frequent i-subgraph candidates for every graph g separately by appending every possible frequent 1-subgraph to frequent $(i-1)$-subgraphs that both g contain, duplicated subgraphs can be detected by simply comparing their vertex IDs, and hence avoiding the graph isomorphism test which is a costly operation. Different from the above three works that iteratively mine frequent subgraphs, [1,16] issues two MapReduce jobs, with one to identify all frequent subgraph candidates, with the other to verify whether each candidate is frequent or not. The whole dataset is first divided into m disjoint subsets, and each subset is sent to a single machine. The rational behind [16] is that if a subgraph s is a frequent, s must be reported as a locally frequent subgraph in some machine. For this reason, [16] extracts every possible frequent subgraph as a candidate in the first MapReduce job. To collect the complete frequency of a subgraph s, it is necessary to further compute the frequency for s that is locally infrequent in other machines. To verify whether s is frequent or not, [16] issues another MapReduce Job to compute the frequency of every candidate. To avoid generating a huge number of candidates, the frequency of s that is locally infrequent in some machines is still computed. In this way, the upper bound of the frequency of s can be refined and if the upper bound is less than the frequency support, s can be pruned. The shortcomings of [16] are twofold. First, it is difficult to select such a set of subgraphs that are locally infrequent but required to compute the frequency. Improper selection will cause either a huge number of candidates or unnecessary computations (the frequency of a subgraph is computed across multiple machines but finally verified as an infrequent subgraph). Second, the selection problem will become increasingly difficult if the number of machines grows (image the worst case that the number of machines equals that of the graphs). Aridhi et al. [1] propose a density-based data partitioning strategy to balance the workload. We do not compare MRFSE with [1] because [1] mines frequent subgraphs approximately.

This paper is an extension of our previous work [19]. As pointed out before, all existing iterative approaches are necessary to shuffle mappings, the size of which is prohibitively large, from *mappers* to *reducers*, causing an expensive communication cost. In this work, we introduce a new solution that is able to avoid shuffling mappings and hence improve the mining performance.

3 Preliminaries

In this section, we first formally define the problem of frequent subgraph mining and then present a brief review of the gSpan [24]. Table 1 summarizes the notations used throughout this paper.

3.1 Frequent subgraph mining

For ease of presentation, we model each complex structure as an undirected, labeled graph. Typically, a labeled graph g is able to be represented as a quadruple (V_g, E_g, L_g, l_g), in which V_g is the set of vertices, E_g is the set of undirected edges, L_g is a set of labels, and l_g is a labeling function that maps every vertex and edge to a single label in L_g. Given a graph g, we use $l_g[v]$ to denote the label of vertex v and $l_g[u, v]$ to denote the label of edge (u, v) connecting vertices u and v in g. For the sake of brevity, we use $g.id$ to denote the identifier of graph g.

Given two graphs g and s, we can verify whether s is a subgraph of g (i.e., g is a supergraph of s or g contains s) by performing *subgraph isomorphism*, which is defined as follows:

Definition 1 (*Subgraph Isomorphism* $\subseteq$) Given a graph $s = (V_s, E_s, L_s, l_s)$ and a graph $g = (V_g, E_g, L_g, l_g)$, s is said to be subgraph isomorphic to g (denoted as $s \subseteq g$) if and only if there exists an injective function $f : V_s \rightarrow V_g$ such that (1) $\forall v \in V_s$, we can have $f(v) \in V_g$, and $l_s(v) = l_g(f(v))$; (2) $\forall (u, v) \in E_s$, we can have $(f(u), f(v)) \in V_g$, and $f_s[u, v] = f_g[f(u), f(v)]$.

Table 1 Symbols and definitions

Symbols	Definitions
D	A collection of graphs
g	A graph in D
V_g	The set of vertices in g
E_g	the set of edges in g
$g.id$	The graph identifier of g in D
$l_g[v]$	The label of vertex v in g
$l_g[u, v]$	The label of edge (u, v) connecting u and v in g
s	A frequent subgraph candidate
$s \subseteq g$	s is a subgraph of g (or g is a supergraph of q)
g_e^s	A subgraph in g built by embedding e and subgraph s
$s.D$	An ID list of graphs in D containing s
T	The frequent threshold
F_i	All size-i frequent subgraphs contained in the dataset
F_i^g	Subgraphs in F_i that graph g contains
$\mathbb{F}_i^g$	A superset of F_i^g
$\mathcal{E}_s^g$	All embeddings in g, $\forall e \in \mathcal{E}_s^g$, g_e^s is graph isomorphic to subgraph s

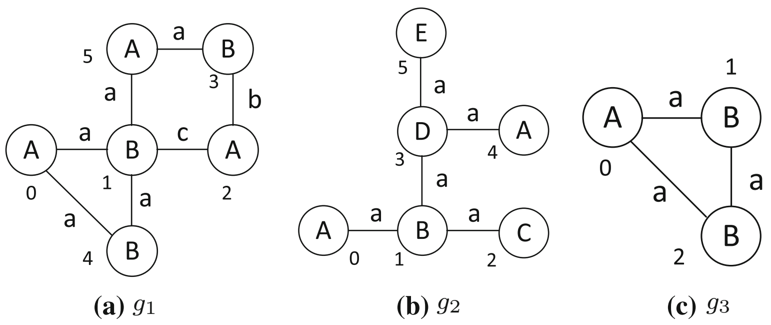


Fig. 1 An Example of Subgraph Isomorphism. **a** g_1, **b** g_2 and **c** g_3

Example 1 **(Subgraph Isomorphism)** Figure 1 shows an example of three undirected, labeled graphs. According to Definition 1, we can find that $g_3 \subseteq g_1$ since there exists an injective function f: $\{0 \to 0, 1 \to 1, 2 \to 4\}$, while $g_3 \nsubseteq g_2$.

Particularly, given two graphs s and g, if $s \subseteq g$ and $|V_s| = |V_g|$, then we say s is *graph isomorphic* to g. That is, graph isomorphism is a special case of subgraph isomorphism.

Definition 2 (*Embedding*) Given a subgraph s and a graph g, suppose $s \subseteq g$ and f is an injective function. An **embedding** e is a sequence of vertices in V_g that are mapped to the vertices of V_s using f, i.e., $e = [f(v_1), \ldots, f(v_i), \ldots, f(v_{|V_s|})]$, where $v_i \in V_s$ and $f(v_i) \in V_g$.

Figure 2 shows two subgraph s_1, s_2 and the embeddings in g_1 (shown in Fig. 1a) for them, respectively. In practice, an embedding acts like an injective function. Given a subgraph s

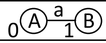
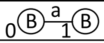
Subgraph	Embeddings				Subgraph	Embeddings	
s_1 	[0 1]	[0 4]	[5 1]	[5 3]	s_2 	[1 4]	[4 1]

Fig. 2 An example of embeddings

Order	0	1	2	3	4
Sequence	(0,1,A,a,B)	(1,2,B,a,C)	(1,3,B,a,D)	(3,4,D,a,A)	(3,5,D,a,E)

Fig. 3 An example of DFS code

and a graph g, we can build a unique subgraph g_e^s in g based on an embedding e such that g_e^s is graph isomorphic to s. It is worth mentioning that there might exist multiple embeddings for a subgraph sg in g for s in that several vertices and edges are attached with the same labels (taking s_2 in Fig. 2 for example).

A subgraph s is called *connected* if for any two vertices $u, v \in V_s$, there exists a path from u to v in s. Given a graph collection D and a subgraph s, the *posting list* of s is defined as an ID list of graphs in D containing s, i.e., $s.D = \{g.id | g \in D, s \subseteq g\}$. Given a subgraph s, s is said to be *frequent* if $|s.D| \geq T$, where T is called *frequency threshold* which is a user-defined number. In many applications, users are more concerned with the frequent recurring components of graphs [23]. Hence, in this paper, we focus on identifying the frequent subgraphs that are connected. The frequent unconnected subgraphs can be derived based on these frequent connected subgraphs using a frequent itemset mining algorithm [15].
Problem Definition Given a graph collection D and a frequency threshold T, the problem of *Frequent subgraph mining* is to identify all connected subgraphs which are contained in at least T graphs of D.

In the remainder of this paper, we refer to a subgraph as a connected subgraph. For the sake of brevity, we also use i-subgraph to denote a subgraph with i edges, and F_i to denote all frequent subgraphs in D with i edges. Given an i-subgraph s, we say s' is a *child* of s when $s \subseteq s'$ and $|E_{s'}| = i + 1$; similarly, we say s' is a *parent* of s when $s' \subseteq s$ and $|E_{s'}| = i - 1$. An enumeration operation of an i-subgraph s is to enumerate all children of s by appending one edge from every possible vertex in V_s.

3.2 Minimum DFS code and gSpan

The key technique of the gSpan algorithm is its *DFS coding*, by which each subgraph can be translated to a collection of DFS codes. Each DFS code is an edge sequence, generated by performing a depth first search (DFS) on the edges of the subgraph, i.e., edges in the DFS code are ranked by the discovery order in the DFS search. Correspondingly, each vertex in the subgraph is also marked by its discovery order. In the DFS coding technique, an edge in the DFS code is represented by a 5-tuple, $(i, j, l_s[i], l_s[i, j], l_s[j])$, where $l_s[i], l_s[i, j], l_s[j]$ are the labels of vertex i, edge (i, j), and vertex j in the subgraph s, respectively. We show a DFS code consisting of five ordered edges in Fig. 3 for the graph g_2 shown in Fig. 1b. Since there exist multiple depth-first search trees for a given subgraph s, s can have many DFS codes. Therefore, a lexicographic order is used so that every two DFS codes can be compared with each other [24].

Property 1 (*Minimum DFS Code*) [24] Given an i-subgraph s, let parent$(s).min$ be the parent in F_{i-1} with the minimum DFS code. Then, the minimum DFS code of s is the concatenation of parent$(s).min$ and the DFS code of the new added edge.

Algorithm 1: gSpan(D, F, s)

```
1 if s.code is not the minimum DFS code of s then
2     return;
3 $F \leftarrow F \bigcup \{s\}$ // $F$ is the collection of frequent subgraphs ;
4 compute $Enu(s)$;
5 foreach $s' \in E(s)$ do
6     if $|s'.D| \geq T$ then
7         gSpan($D, F, s'$);
```

At first, each different 1-subgraph is generated with its minimum DFS code. This generation can be implemented by just scanning the dataset once. By introducing the minimum DFS code for the same subgraph, where the minimum DFS code can be obtained based on Lemma 1, gSpan is able to avoid redundant enumeration and restrict the extension of s in a proper way.

Property 2 (*Redundant Enumeration Elimination*) [24] Given a frequent subgraph s, children of s will be enumerated by extending one edge from every possible vertex in V_s if and only if it takes the minimum DFS code.

Property 2 eliminates redundant enumeration for the same subgraph in order to avoid many unnecessary computations. Consider the graph g_2 shown in Fig. 1b as an example. Subgraph (0, 3, 5) of g_2 can be extended from either (0, 3) or (3, 5). Since subgraph (0, 3, 5) that has been extended from (3, 5) does not take the minimum DFS code, no enumeration is required in this case. On the other hand, the same subgraph extended from (0, 3) takes the minimum DFS code, and hence, enumeration of all its children will be performed by extending one edge. In this manner, we can ensure that enumeration for the same subgraph is executed only once.

Property 3 (*Enumeration Restriction*) [24] Given a frequent subgraph s associated with the minimum DFS code, suppose L and R are the last and first discovery vertices in the depth-first search. Let $P_{R \to L}$ be the path from R to L. We enumerate children of s by appending one edge in one of the following two cases: (1) *backward extension:* from L to a vertex in $P_{R \to L}$; (2) *forward extension:* from a vertex in $P_{R \to L}$ to a new vertex.

Property 3 restricts the enumeration of a frequent subgraph by extending one edge from its partial vertices instead of the complete vertices. For example, enumerating children of g_2 shown in Fig. 1 constrains the extension of edges from vertices 0, 3, 5. We use $Enu(s)$ to denote the children of s by extending one edge using Property 3. After generating all distinct 1-subgraphs, the details of extracting all frequent subgraphs and their posting lists based on each 1-subgraph are shown in Algorithm 1.

4 Overview of the solution

We propose a solution, called MRFSE (MapReduce-based frequent subgraph extraction), to iteratively mine frequent subgraphs using a MapReduce-based platform such as Hadoop in the level growth fashion. In our design, the whole dataset D is first divided into m disjoint subsets and each subset D_i is sent to a single machine M_i. In particular, M_i is responsible

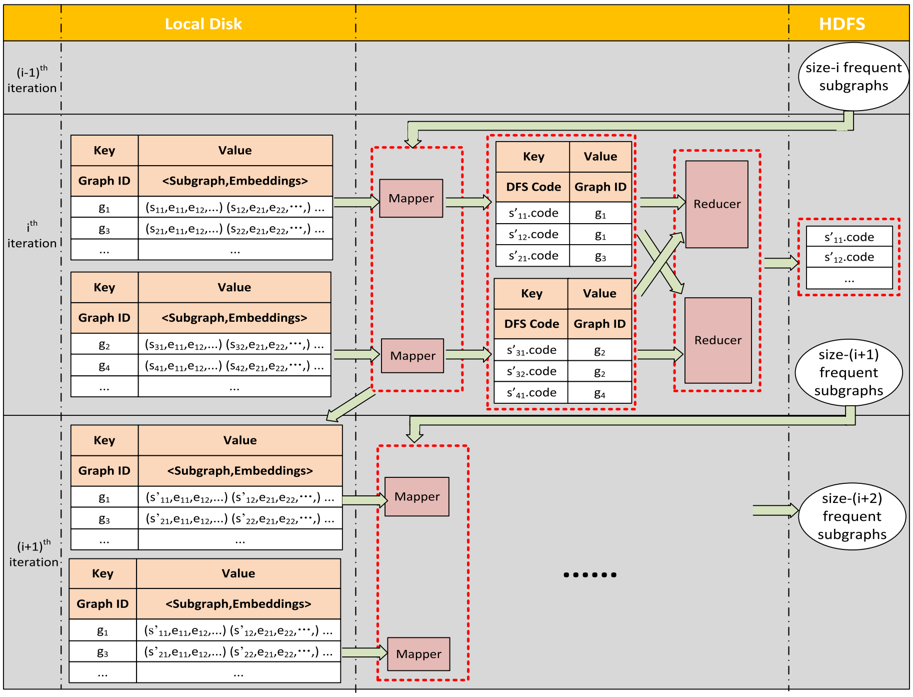

Local Disk
HDFS
(i-1)th iteration
size-i frequent subgraphs
ith iteration
Key | Value
Graph ID | <Subgraph,Embeddings>
g1 | (s11,e11,e12,...) (s12,e21,e22,⋯,) ...
g3 | (s21,e11,e12,...) (s22,e21,e22,⋯,) ...
g2 | (s31,e11,e12,...) (s32,e21,e22,⋯,) ...
g4 | (s41,e11,e12,...) (s42,e21,e22,⋯,) ...
Mapper
Key | Value
DFS Code | Graph ID
s'11.code | g1
s'12.code | g1
s'21.code | g3
s'31.code | g2
s'32.code | g2
s'41.code | g4
Reducer
s'11.code
s'12.code
size-(i+1) frequent subgraphs
(i+1)th iteration
g1 | (s'11,e11,e12,...) (s'12,e21,e22,⋯,) ...
g3 | (s'21,e11,e12,...) (s'22,e21,e22,⋯,) ...
size-(i+2) frequent subgraphs
......

Fig. 4 Overview of the solution to mine frequent subgraphs

for mining frequent subgraphs from graphs in D_i unchangeably during iterations. At the ith iteration, we coordinate machines to mine all frequent i-subgraphs using a single MapReduce Job. Figure 4 shows the overview of our solution for mining frequent subgraphs in details. Specifically, each mapper in M_i generates frequent i-subgraph candidates from D_i which are then shuffled to reducers. Reducers verify and output all frequent i-subgraphs by aggregating and counting all isomorphic i-subgraphs.

For ease of illustration, we denote as F_i^g all frequent i-subgraphs that g contains, and for each subgraph $s \in F_i^g$, we denote as $\mathcal{E}_s^g$ all the embeddings in g where $\forall e \in \mathcal{E}_s^g$, g_e^s is graph isomorphic to s. To continuously generate frequent subgraph candidates during iterations, we maintain $\bigcup_{g \in D} F_i^g$ and $\bigcup_{g \in D} \bigcup_{s \in F_i^g} \mathcal{E}_s^g$,[4] which are generated at the end of iteration i, and then taken as the input of iteration $i+1$. As discussed before, in our design, each machine M_i is responsible for mining frequent subgraphs from graphs in a subset D_i. In this way, by configuring one or multiple mappers in each machine, each mapper in M_i is responsible for mining frequent subgraphs from a certain number of graphs in D_i. *Without loss of generality, we assume each machine is configured to have a single mapper.* Next, we elaborate how MRFSE mines frequent subgraphs at each iteration.

Initially, the whole dataset D is first divided into m disjoint subsets and each subset D_i is sent to a single machine M_i. Note D_i is maintained in the local disk of M_i. The first job mines frequent 1-subgraphs. The map function in M_i takes every $g \in D_i$ as the input and generates two outputs. The first output is $\mathbb{F}_1^g$ (a superset of F_1^g) and $\bigcup_{s \in \mathbb{F}_1^g} \mathcal{E}_s^g$. *Note that every subgraph in* $\mathbb{F}_1^g$ is stored using its minimum DFS code. The first output is written to the local

[4] For every $g \in D$, we maintain all frequent i-subgraphs associated with the corresponding embeddings.

disk and taken as the input of the mapper in the next MapReduce Job. The second output consists of a key value pair, highlighted in the rectangle with a red dashed line of Fig. 4. The key is the minimum DFS code of each subgraph, and the value is the graph identifier. The mapper shuffles the minimum DFS code of each subgraph associated with a graph identifer. Given two subgraphs s_1, s_2, if the minimum DFS codes of s_1, s_2 are equal, then s_1,s_2 are graph isomorphic [24]. For this reason, reducers verify and output all frequent 1-subgraphs to HDFS by aggregating and counting all isomorphic size-1 subgraphs that share the same minimum DFS codes.

The second job mines frequent 2-subgraphs. The map function in M_i takes F_1, $\mathbb{F}_1^g$, $\bigcup_{s\in\mathbb{F}_1^g} \mathcal{E}_s^g$ as the input, and generates two outputs as well. F_1 contains all frequent 1-subgraphs and is used to filter out non-frequent subgraphs in $\mathbb{F}_1^g$ and their corresponding embeddings by examining their minimum DFS codes. The first output is $\mathbb{F}_2^g$ (a superset of F_2^g) and $\bigcup_{s\in\mathbb{F}_2^g} \mathcal{E}_s^g$. Existing Apriori-based and pattern-growth approaches can be extended to generate F_2^g and $\bigcup_{s\in\mathbb{F}_2^g} \mathcal{E}_s^g$ based on F_1^g and $\bigcup_{s\in F_1^g} \mathcal{E}_s^g$. In the next section, we will describe the details of the above generation. *Similarly, every subgraph in* $\mathbb{F}_2^g$ *i*s stored using its minimum DFS code. The first output is written to the local disk and taken as the input of the mapper in next MapReduce Job. The second output consists of a key value pair, which is the minimum DFS code of each 2-subgraph in $\mathbb{F}_2^g$ and the graph identifier, respectively. The iteration goes on similarly to iteration 2 until no new frequent subgraphs are generated. Compared with existing frequent subgraph mining approaches using MapReduce, MRFSE has the following properties:

– Each graph g is taken as the process unit, and we generate new subgraphs by appending one edge based on the frequent subgraphs that g contains. For this reason, existing approaches in centralized systems can be easily extended in our solution.
– In our design, no embeddings are shuffled among machines as each machine is responsible for mining frequent subgraphs from a separate set of graphs unchangeably during iterations. For this reason, MRFSE is able to save a large amount of network bandwidth.
– In the map function, only F_i^g and $\bigcup_{s\in F_1^g} \mathcal{E}_s^g$ are maintained in main memory. Obviously, MRFSE reduces the memory requirement significantly.

In our solution, to provide enough number of mappers in the subgraph enumeration job, we split the dataset into partitions at the beginning, each of which will be held and processed by a separate mapper at each iteration. In order to guarantee that each mapper processes equal amount of computations at each iteration, we propose two partitioning strategies as follows.

– *Random partitioning* In this strategy, we randomly assign each graph to a machine.
– *Equal size partitioning* In this strategy, we assign each graph to the machine with the minimum number of edges for the graphs inside this machine.

5 Subgraph enumeration

In this section, we present the map task of enumerating frequent $(i+1)$-subgraph candidates associated with their embeddings in a separate graph g. For the ease of illustration, in what follows, we assume that a subgraph is associated with a set of embeddings whenever there is no ambiguity in our discussion.

The input of subgraph enumeration is F_i, $\mathbb{F}_i^g$ and $\bigcup_{s\in\mathbb{F}_i^g} \mathcal{E}_s^g$. The output of subgraph enumeration is $\mathbb{F}_{i+1}^g$ and $\bigcup_{s\in\mathbb{F}_{i+1}^g} \mathcal{E}_s^g$. The main idea of our approach is that $\forall s \in F_i^g$, we enumerate children of s by appending one edge based on $\bigcup_{s\in F_i^g} \mathcal{E}_s^g$. Formally, we can have:

Algorithm 2: Subgraph Enumeration Task

```
1  map-setup                                          /* load frequent $i$-fragments */
2      Load $F_i$ from HDFS;
3  map ($k_1, v_1$)                                   /* generate $(i+1)$-subgraphs */
4      parse $\mathbb{F}_i^g$ from $v_1$;
5      foreach $s \in \mathbb{F}_i^g$ do
6          if contain($F_i$, $s.code$) then
7              add $s$ to $F_i^g$;
8      $\mathbb{F}_{i+1}^g \leftarrow$ enumeration ($F_i^g$);
9      write $\mathbb{F}_{i+1}^g$ to the local disk;
10     foreach $f \in \mathbb{F}_{i+1}^g$ do
11         emit($f.code$, $k_1$)
12         $k_1 = g.id$;
```

$$\mathbb{F}_{i+1}^g = \bigcup_{\forall s \in F_i^g} Enu(s) \tag{1}$$

The pseudo-code for subgraph enumeration is presented in Algorithm 2. Before executing the map function, we first set F_i to the frequent i-subgraphs which have been identified in the previous iteration (line 2). Note that we only load the minimum DFS code of each frequent i-subgraph into main memory rather than the subgraphs themselves. It is feasible because the number of frequent i-subgraphs remains fairly constant when the cardinality of the dataset increases. In practice, when F_i is too large to be held in main memory, we only load the hash keys of the minimum DFS codes or just choose not to load F_i at all. Although it incurs unnecessary enumerations for some non-frequent i-subgraphs, the correctness is still guaranteed because non-frequent i-subgraphs will be pruned in the frequent subgraph extraction job. At each map function, we first parse $\mathbb{F}_i^g$ from the input value (line 4), and filter non-frequent subgraphs that are not contained in F_i (lines 5–7). We generate a superset, $\mathbb{F}_{i+1}^g$, of F_{i+1}^g based on F_i^g (line 8) and write $\mathbb{F}_{i+1}^g$ to the local disk (line 9). To help count frequent $(i+1)$-subgraphs, we shuffle the minimum DFS code of each subgraph and the identifier of the graph containing this subgraph separately, which are then aggregated in reducers.

Existing techniques can be easily extended and integrated into our framework to extract frequent subgraphs. In the following, we extend pattern-growth approaches (called *extension-based enumeration*), and Apriori-based approaches (called *join-based enumeration*) to help generate new subgraphs at each iteration.

5.1 Extension-based enumeration

Consider an example by taking s_1 shown in Fig. 2 as s and g_1 shown in Fig. 1 as g. According to Lemma 3, we first identify the vertices in path $P_{R\rightarrow L}$ for s, which are vertex 0 labeled with A and vertex 1 labeled with B. Since there exist only two vertices, we do not need to perform the backward extension. We begin from the vertex 1 of s to conduct the forward extension by appending a new edge. Since we have already identified all embeddings in g for s, we sequentially check the vertex v in each embedding e where v in g_e^s is mapped to vertex 1 of s. Specifically, we first collect all frequent edges which start from v but not contained in g_e^s, and append the edge one by one. For example, for embedding [0 1] (emphasized in yellow of embeddings of s_{11} in Fig. 5), starting from vertex 1, we can append edges (1, 5), (1, 4), (1, 2),

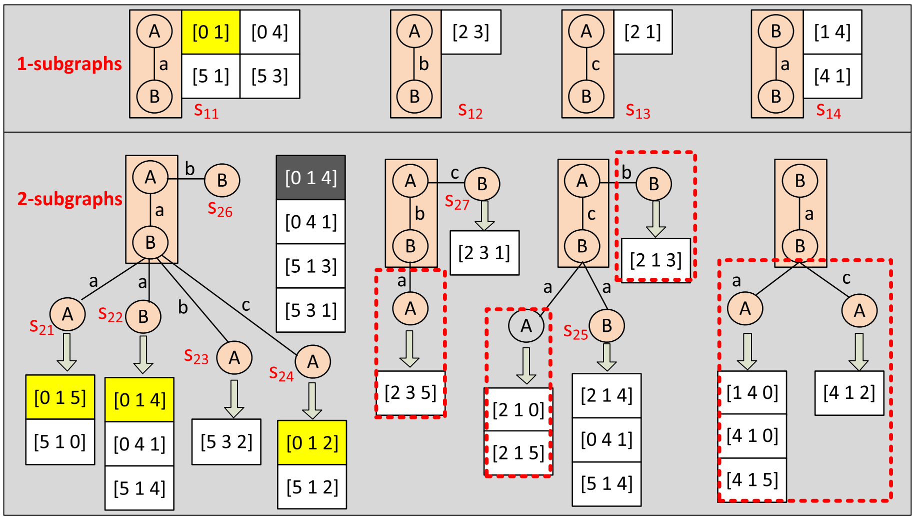


Fig. 5 An example of extension-based enumeration

Algorithm 3: E-Enumeration(F_i^g)

```
1  $\mathbb{F}_{i+1}^g \leftarrow \emptyset$; $genG \leftarrow \emptyset$;
2  $\mathbb{E} \leftarrow$ collect distinct edges from $F_i^g$;
3  foreach $s \in F_i^g$ do
4      foreach extension do
5          foreach $e \in s.E$ do
6              $\mathbb{S} \leftarrow$ ext($e$, $\mathbb{E}$, extension);
7              foreach $s' \in \mathbb{S}$ do
8                  let $e'$ be an embedding of $s'.E$;
9                  if !contain($genG$, $g_{e'}^{s'}$) then
10                     $genG \leftarrow genG \cup \{g_{e'}^{s'}\}$;
11                     if !contain($\mathbb{F}_{i+1}^g$, $s'.code$) then
12                         $\mathbb{F}_{i+1}^g \leftarrow \mathbb{F}_{i+1}^g \cup \{s'\}$;
13                     else
14                         $\bar{s} \leftarrow$ get($\mathbb{F}_{i+1}^g$, $s'$);
15                         $\bar{s}.E \leftarrow \bar{s}.E \cup s'.E$;
16 return $\mathbb{F}_{i+1}^g$;
```

based on which three new 2-subgraphs are generated and the corresponding embeddings are highlighted in yellow in the figure. Similarly, we can perform the forward extension for the other embeddings associated with the same subgraph. After extending one edge from vertex 1, we perform with the forward extension from the vertex 0 of s. For embedding [0 1], we append one edge $(0, 4)$ that starts from 0. A new 2-subgraph is generated and the corresponding embedding is highlighted in black, and the complete extension for s in g is shown at the bottom of Fig. 5.

Lemma 1 (Isomorphism-free Verification) *Given a subgraph s with an embedding e, and another subgraph s' with an embedding e', if $|E_{g_e^s}| = |E_{g_{e'}^{s'}}|$, and $\forall (i, j) \in E_{g_e^s}$, $(i, j) \in E_{g_{e'}^{s'}}$, then g_e^s and $g_{e'}^{s'}$ are the identical subgraphs. Therefore, s is graph isomorphic to s'.*

To avoid unnecessary enumeration of the same subgraphs, we need to remove redundant subgraphs with non-minimum DFS code. Take the subgraph s_{12} shown in Fig. 5 as an example. By appending edge (3, 5) for embedding [2 3], we can generate a subgraph s with embedding [2 3 5]. Although we can perform either a graph isomorphism testing of s on the generated subgraphs, or a minimum DFS code verification of s, these two operations are expensive. In practice, we can verify whether s is a redundant subgraph simply based on Lemma 1. For example, by constructing the subgraph from embedding [2 3 5] of s, we find that it is identical to the subgraph constructed from embedding [5 3 2] of subgraph s_{23} shown in Fig. 5. Hence, s is subgraph isomorphic to s_{23}, and we do not need to enumerate s.

According to Lemma 1, for each subgraph s, s generated by $parent(s).min$ takes the minimum DFS code. Hence, we rank the subgraphs in $\mathbb{F}_i^g$ in the ascending order of their minimum DFS codes. When a subgraph is verified as a duplicate, we can safely discard this subgraph and its embeddings since the associated DFS code is not minimum. After ranking and enumerating al 1-subgraphs, we can detect the other redundant subgraphs and their embeddings surrounded by dashed red lines.

The embedding-based enumeration algorithm is outlined in Algorithm 3. At first, we initialize $\mathbb{F}_{i+1}^g$, and a hash set $genG$ which maintains all distinct g_e^s for each new subgraph $s \in \mathbb{F}_{i+1}^g$ associated with an embedding $e \in s.E$ (line 1). We then sort the subgraphs in F_i^g in the ascending order of their minimum DFS codes and collect the distinct edges from their minimum DFS codes (lines 2–3). For each subgraph s in F_i^g, we enumerate all children of s by appending one edge from every possible vertex, which is described in Lemma 3 (lines 4–16). For each possible vertex v, we sequentially check the embeddings of s and extending one edge from the mapping vertex v' of each embedding to v by probing the edges starting from v' in $\mathbb{E}$ (line 7). For each newly generated subgraph s', according to Lemma 1, if $g_{e'}^{s'}$ is contained in $genG$, we can verify that the associated DFS code of s' is not the minimum. We note that s' might have already been contained in F_i^g, since the same s' is generated by s but with another embedding. In this case, we merge their embeddings together (lines 14–16). Finally, we return $\mathbb{F}_{i+1}^g$.

5.2 Join-based enumeration

As noted in Sect. 1, Apriori-based approaches generate new subgraphs by joining two frequent subgraphs. Directly employing Apriori-based approaches poses three problems:

– Redundant elimination is required for the newly generated subgraphs by performing graph isomorphism testing.
– Subgraph isomorphism testing is required to verify whether the newly generated subgraphs are contained in g.
– It is costly to identify join pairs of subgraphs.

To tackle these problems, we extend the Apriori-based approaches by joining two embeddings rather than the subgraphs. For join-based enumeration, we do not need to maintain all embeddings for a subgraph s in g. According to Lemma 2, we join two embeddings once their corresponding subgraphs in g satisfy a certain condition. However, if we maintain multiple embeddings for the same subgraph, that will incur unnecessary computations for performing the same join. Hence, in our design, for each subgraph sg contained in g, we maintain only one embedding of sg for s if sg is graph isomorphic to s.

Lemma 2 *Given an embedding $e \in s.E$ and another embedding $e' \in s'.E$, where s and s' are contained in F_i^g, we join e and e' if and only if one of the following conditions is satisfied: (1) if $i = 1$, g_e^s and $g_{e'}^{s'}$ share one vertex; (2) else, g_e^s and $g_{e'}^{s'}$ share one connected child, i.e., they share $i - 1$ edges.*

For a subgraph with ring or star structures, the number of embeddings can be reduced significantly as a result. Consider two subgraphs s and sg in Fig. 6 for example. We can observe that sg is graph isomorphic to s. Since each vertex labeled with B of s can be mapped to every vertex labeled with B of sg, the total number of embeddings in sg for s is 24. However, in our case, we only maintain one embedding. Compared with the extension-based enumeration, join-based enumeration has the following properties:

1. $\forall s \in \mathbb{F}_i^g$, it maintains less number of embeddings for s;
2. It generates smaller size of $\mathbb{F}_{i+1}^g$. That is, some non-frequent subgraphs in $\mathbb{F}_{i+1}^g$ are pruned only based on F_i^g.

Hence, the I/O cost incurred in identifying frequent $(i + 1)$-subgraphs can be reduced, by maintaining less number of both embeddings and subgraphs. We illustrate how to generate new subgraphs by join embedding pairs and take graph g_1 (shown in Fig. 1) as g for example. The generated 1-subgraphs and their embeddings for g are shown in the top left corner of Fig. 7.

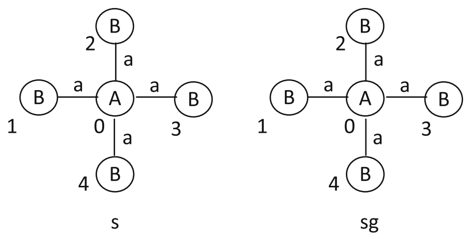
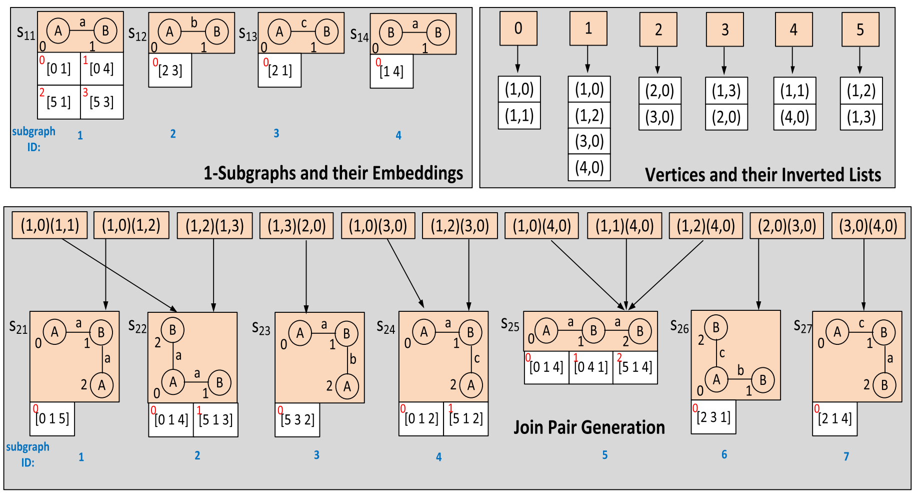


Fig. 6 Two subgraphs with star structures



1-Subgraphs and their Embeddings
Vertices and their Inverted Lists
Join Pair Generation
subgraph ID:

Fig. 7 A join example for 1-subgraphs

Based on Lemma 2, we join two embeddings if they satisfy the first condition. Hence, we sequentially scan each embedding, starting from the subgraph (s_{11}) with the smallest minimum DFS code to that (s_{14}) with the largest minimum DFS code, and build the inverted list for each vertex, as shown in the top right corner of Fig. 7. Each item in the inverted list consists of a pair (id_1, id_2), where id_1 and id_2 represent the subgraph identifier in F_i^g, and the embedding identifier in the subgraph, respectively, and are highlighted in red and blue colors, respectively. Then, we join every item pair that belong to the same inverted list. Since an $(i+1)$-subgraph could be generated by two different item pairs, to guarantee that each $(i+1)$-subgraph is generated by the i-subgraph in F_i^g with the minimum DFS code, we join the item pairs according to an order. We first sort the subgraphs in F_i^g according to the ascending order of their minimum DFS codes and then sequentially process each subgraph s in F_i^g that is involved in the join pairs.

Lemma 3 *Given an embedding $e \in s.E$ and another embedding $e' \in s'.E$, suppose $s.code \leq s'.code$ and $s, s' \in F_i^g$, let (φ_1, φ_2) be the edge contained in $g_{e'}^{s'}$ but not in g_e^s, and φ_1 be the vertex contained in both $g_{e'}^{s'}$ and g_e^s. If e and e' satisfy the join condition, then we generate a new subgraph sg based on g_e^s by adding edge (φ_1, φ_2) in $g_{e'}^{s'}$ to g_e^s from its vertex φ_1.*

For the example shown in Fig. 7, by sorting the subgraphs in F_1^g, which are $s_{11}, s_{12}, s_{13}, s_{14}$, we will first process join pairs that s_{11} involves in, which are (1,0)(1,1), (1,0)(1,2), (1,0)(3,0), (1,0)(4,0), (1,2)(3,0), (1,2)(4,0), (1,3)(2,0), (1,1)(4,0), (1,2)(1,3). The item pairs are joined according to Lemma 3. For example, to join item pair $(1, 0)(1, 1)$, which corresponds to embeddings [0 1] and [0, 4], we can identify that φ_1 and φ_2 are vertices 0 and 4. By appending the edge $(0, 4)$ to $g_{s_{11}\cdot e_0}^{s_{11}}$, we can generate a 2-subgraph sg which is graph isomorphic to s_{22}. However, in our case, the generated DFS code from sg may not be the minimum for s_{22}, because we only maintain a single embedding while omitting the other possible embeddings of sg for s_{22}. We use the mechanism proposed in [24] to identify the minimum DFS code of s_{22} based on sg. One observation is that the subgraphs joined by items pairs (1,0)(1,1) and (1,2)(1,3) are graph isomorphic to s_{22}, and hence, by first identifying the minimum DFS code of s_{22} based on (1,0)(1,1), subsequent identification of s_{22} can be omitted.

Similarly, based on Lemma 1, we can discard s if there exists an identical subgraph to a new generated $(i+1)$-subgraph g_e^s. Another important observation is that although an $(i+1)$-subgraph g_e^s is generated by joining two frequent i-subgraphs, we may verify that s is not a frequent subgraph without scanning the dataset. To illustrate this point, we consider the four frequent 3-subgraphs that are shown in Fig. 8. We first build the inverted lists and then generate the new subgraphs by joining item pairs. Note that given an $(i+1)$-subgraph s, the necessary condition of qualifying s to be frequent is that all its connected children are frequent. Therefore, s_{41} and s_{43} are not frequent since one of their subgraphs is not frequent, and this can be simply verified based on Lemma 4.

Lemma 4 *Given a subgraph s, let $\tau(s)$ be the number of its connected children and $\phi(s)$ be the item pairs from which s can be joined. The necessary condition of qualifying s to be frequent is that $\phi(s) = \tau(s) * (\tau(s) - 1)/2$.*

We present the join-based enumeration algorithm in Algorithm 4. The initialization is the same as that of Algorithm 3 (line 1). By scanning the embeddings in F_i^g, we build the inverted lists $\mathbb{L}$ (line 2). We then sequentially scan each subgraph s in F_i^g and collect the join pairs $\mathbb{J}$ that s is involved in (line 4). For each join pair $p \in \mathbb{J}$, we generate the subgraph sg based on p (line 6). If sg is verified as a new subgraph, then we compute its minimum DFS

Algorithm 4: J-Enumeration(F_i^g)

```
1  $\mathbb{F}_{i+1}^g \leftarrow \emptyset$; $genG \leftarrow \emptyset$;
2  sort subgraphs of $F_i^g$ in the order of minimum DFS codes;
3  build the inverted lists $\mathbb{L}$ by scanning embeddings in $F_i^g$;
4  foreach $s \in F_i^g$ do
5      find the join pairs $\mathbb{J}$ which $s$ is involved in;
6      foreach $p \in \mathbb{J}$ do
7          generate subgraph $sg$ contained in $g$ based on $p$;
8          if $!contain(genG, sg)$ then
9              identify $s'$ with the minimum DFS code for $sg$;
10             if $!contain(\mathbb{F}_{i+1}^g, s'.code)$ then
11                 $\mathbb{F}_{i+1}^g \leftarrow \mathbb{F}_{i+1}^g \cup \{s'\}$;
12             else
13                 $\bar{s} \leftarrow$ get($\mathbb{F}_{i+1}^g, s'.code$);
14                 $\bar{s}.E \leftarrow \bar{s}.E \cup s'.E$;
15             $genG \leftarrow genG \cup \{sg\}$;
16 foreach $s \in \mathbb{F}_{i+1}^g$ do
17     if $\phi(s) <> \tau(s) * (\tau(s) - 1)/2$ then
18         remove $s$ from $\mathbb{F}_{i+1}^g$;
19 return $\mathbb{F}_{i+1}^g$;
```

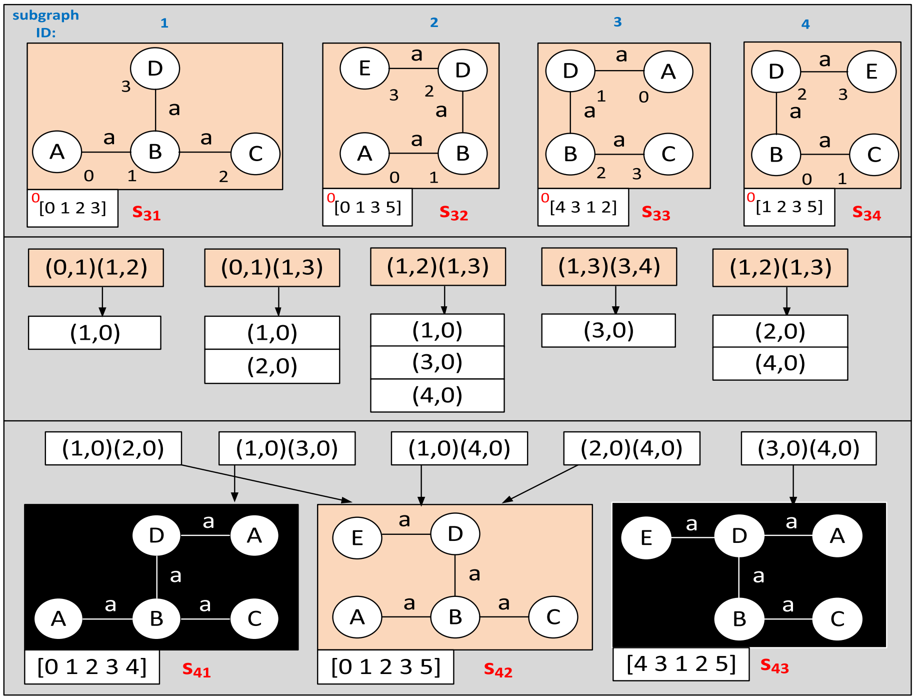


Fig. 8 Pruning non-frequent $(i + 1)$-subgraphs

code and update $\mathbb{F}^g_{i+1}$ (lines 8–14). By pruning non-frequent generated subgraphs in $\mathbb{F}^g_{i+1}$ based on Lemma 4 (lines 15–17), we return $\mathbb{F}^g_{i+1}$. In Lemma 5, we show that our join-based enumeration approach is correct and complete.

Lemma 5 (Correctness and Completeness of Join-based Enumeration) *The join-based enumeration is correct and complete.*

Proof The correctness of join-based enumeration is guaranteed by Lemma 2. Next, we prove the completeness of the join-based enumeration. Given a frequent subgraph s, based on its definition, all parents of s is frequent. Let s' be the parent with the minimum DFS code, and e be the edge which is added to s' to constitute s. By replacing e with any edge in s' (if any), we can constitute a new subgraph $\bar{s}$. Obviously, s can be joined by s' and $\bar{s}$. As we maintain all frequent subgraphs associated with their embeddings, the join-based enumeration is complete. □

Discussion [5] proposes a data structure called VAT to organize the subgraphs and their embeddings. VAT is similar to the data structure used in MRFSE-J. The main difference is that, at first step, VAT generates new frequent subgraph candidates by joining every two possible frequent subgraphs and performs a graph isomorphism test to remove duplicates; at second step, for each new generated candidate, VAT drills down to specific graphs and join two subgraphs by comparing their embeddings. In MRFSE-J, we first build the inverted index, based on which two subgraphs that occur in the same posting list are joined together. Consider that for every join pair P in MRFSE-J, VAT is also necessary to generate P. Obviously, MRFSE-J eliminates the cost of the first step of VAT.

6 Experiment

In this section, we evaluate the performance of the proposed algorithms on a 72-node cluster. Each node in the cluster has one Intel Xeon X3430 2.4 GHz Quad core Processor, 8 GB of RAM, two 500 GB SATA hard disks and gigabit ethernet. Each node is equipped with the CentOS 5.5 operating system, Java 1.6.0 with a 64-bit server VM, and Hadoop 0.20.2. We configure one node to act as the name node and job tracker and the remaining nodes as the data nodes and task nodes. All the nodes are connected via three high-speed switches. We make the following changes to the default Hadoop configuration: (1) the replication factor is set to 1; (2) each node is configured to run one map; (3) the number of mappers is set to the number of used machines instead of the number data splits. A typical example of setting a configurable number of mappers can be found in Giraph,[5] a distributed graph processing systems built on Hadoop.

We evaluate the following approaches in our experiments.

- MRFSE is our proposed approach. MRFSE employs one MapReduce job to mine F_i that is generated without performing graph isomorphism test, and no embeddings are shuffled. *By default, MRFSE uses the extension-based enumeration to generate frequent subgraph candidates.*
- P-MRFSE [19] is our previous work. P-MRFSE employs two MapReduce jobs to mine F_i that is generated without performing graph isomorphism test. Embeddings need to be shuffled once to generate candidates.

[5] https://github.com/apache/giraph.

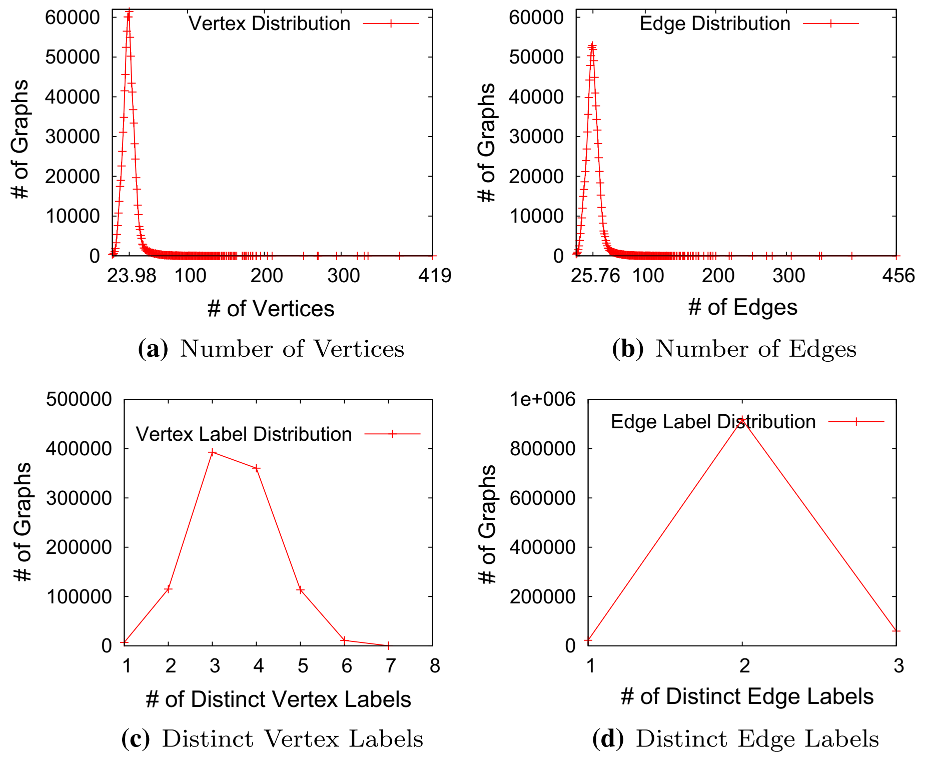


Fig. 9 Distribution of graphs in the real dataset. **a** Number of vertices, **b** number of edges, **c** distinct vertex labels and **d** distinct edge labels

- MIRAGE [3] issues one MapReduce job to mine F_i. In MIRAGE, duplicated candidates need to be detected by performing graph isomorphism test and embeddings needs to be shuffled for the generation of candidates in the next iteration.

We conduct the experiments over one real dataset and several synthetic datasets:

- PubChem[6]: PubChem is a free database which provides various interfaces for users to access and download chemical structures, substances, and biological assays. We extract one million chemical structures from PubChem as the real dataset. The distribution of graphs in PubChem is shown in Fig. 9. The graphs have 23.98 vertices, 25.76 edges, 3.5 distinct vertex labels, 2.0 distinct edge labels on average, and the total number of distinct vertex labels and distinct edge labels is 81 and 3, respectively. The size of PubChem dataset is 434 MB.
- Synthetic Datasets: We use the graph generator generously provided by [7] to generate a collection of graphs with different settings, such as the number of graphs, the average size of each graph ($|E_g|$), the number of distinct labels of vertices and edges. The size of graphs follows a normal distribution with 5 as the variance.

We evaluate the performance of the proposed approaches in terms of the running time, communication cost and I/O cost. We use PubChem as the default dataset, and adjust the

[6] http://pubchem.ncbi.nlm.nih.gov.

Table 2 Parameters for the real dataset

Parameter	Range
Maximum size of frequent subgraphs	2, 4, 6, 8, 10, 12, 14, 16
Frequency threshold (%)	4, 6, 8, **10**, 12
# of compute nodes	**20**, 30, 40, 50

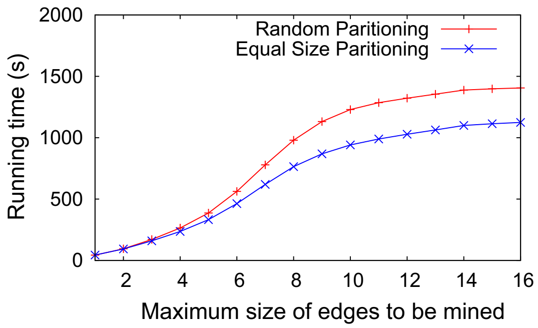


Fig. 10 Effect of data partitioning strategies

maximum size of the frequent subgraphs, the frequency support (T) and the number of machines as summarized in Table 2. The default settings of parameters are highlighted in bold.

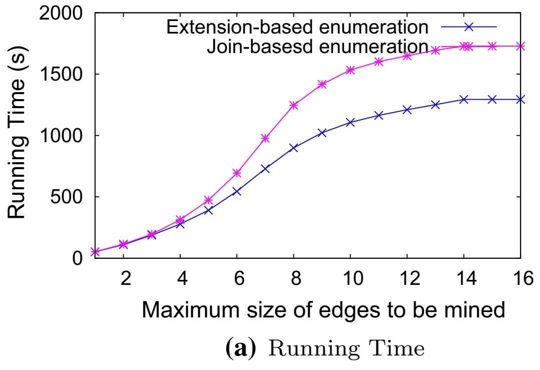
6.1 Effect of data partitioning strategies

We first study the effect of MRFSE using random partitioning and equal size partitioning to split the dataset and report the result in Fig. 10. From the figure, we can observe that performance of using equal size partitioning strategy is better than that of using random partitioning. This is because we enumerate new subgraphs by appending frequent 1-subgraphs to the latest generated frequent subgraphs. Random partitioning makes each machine process similar number of graphs, but the size of graphs could be skewed among machines. While equal size partitioning makes each machine process similar number of edges, the workload could be more balanced among machines in this way, and hence speeding up the mining process. In the remaining experiments, we employ equal size partitioning strategy to partition the data.

6.2 Effect of subgraph enumeration methods

We then evaluate the effect of using different subgraph enumeration methods, i.e., an extension-based approach and a join-based approach, to generate frequent subgraph candidates. The results are plotted in Fig. 11.

The first observation is that the join-based approach incurs less I/O cost. The reason is twofold. First, compared with the extension-based approach, the join-based approach maintains less number of embeddings. Second, the join-based approach is able to filter out a larger number of subgraphs, i.e., less number of frequent subgraph candidates and their corresponding embeddings are written to the local disk, and hence resulting in a less number of I/Os.



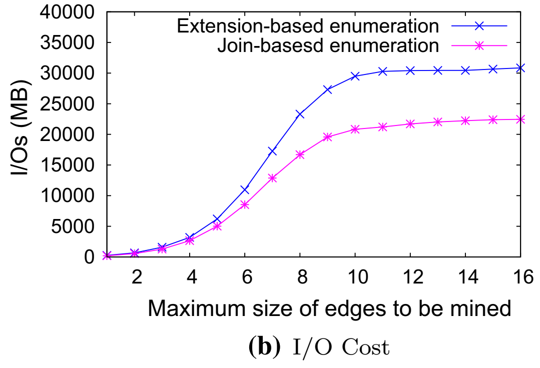


Fig. 11 Comparison of subgraph enumeration methods. **a** Running time and **b** I/O cost

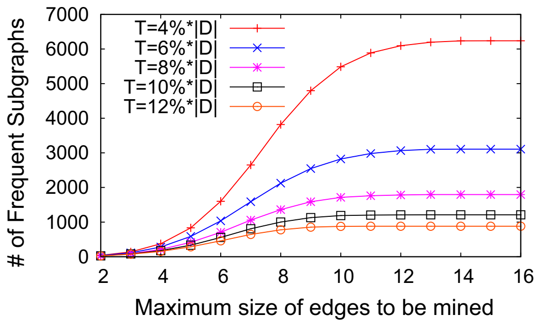
The second observation is that although the join-based approach maintains less number of embeddings, the extension-based approach achieves better performance. The reason is twofold as well. First, as described in Sect. 5, extension-based approach generates new subgraphs using gSpan, i.e., by scanning embeddings associated with each subgraph and appending frequent 1-subgraphs to every possible position of each embedding. Join-based approach generates frequent subgraphs by scanning embeddings associated with each subgraph, producing and joining the embedding pairs. Although the join-based approach generates less number of embeddings and subgraphs, the computation is more costly due to identification of the minimum DFS codes for the newly generated subgraphs. Second, Fig. 9 suggests that the data graphs are sparse, and hence, they may not contain complex subgraphs such as rings or stars. As such, the I/O cost reduction in the join-based approach does not make up for the more expensive computation, and therefore, it is less efficient than the extension-based approach as a result.

To derive a minimum running time, we adopt the extension-based approach to enumerating new subgraphs in MRFSE in the remainder of the experiments.

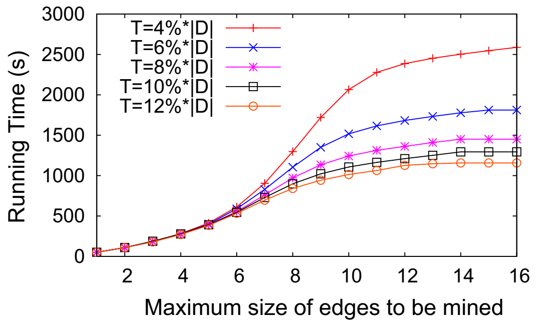
6.3 Effect of frequency support

We study the performance of MRFSE by varying the frequency threshold from 4 to 12% of $|D$, the cardinality of the data set. Figure 12 plots the experimental results in terms of the number of frequent subgraphs, running time and I/O cost of MRFSE. From 12a, we observe that at the beginning, the number of frequent subgraphs increases exponentially and then increases smoothly after an inflection point emerges. This is because frequent subgraphs

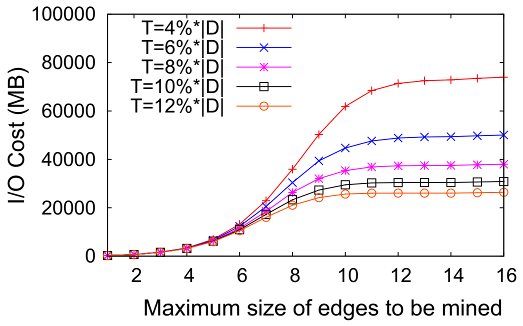
Fig. 12 Effect of frequency support. **a** # of frequent subgraphs, **b** running time and **c** I/O cost



(a) # of Frequent Subgraphs



(b) Running Time



(c) I/O Cost

with smaller size tend to have many extensions that are then able to be verified as frequent subgraphs as well. Nevertheless, frequent subgraphs with larger size may not have many valid extensions with respect to the frequency threshold. The position of the inflection point relies on the frequency threshold. Generally, a small frequency threshold leads to a larger inflection point. Another interesting finding is that the number of frequent subgraphs drops super-linearly when we enlarge the frequency threshold. Both the running time and I/O cost of MRFSE follow the same trends to the number of frequent subgraphs by varying the frequency threshold. As we can see, mining frequent subgraphs from one million chemical instances with original data size 434 MB needs to consume about 35 GB disk spaces. Like [9], this finding verifies as well that frequent subgraph mining is a memory-intensive task, and it is rather necessary to propose a distributed file system-based approach, like MFRSE, to solve this problem.

6.4 Comparison with existing approaches

We next compare MRFSE with existing approaches over PubChem real dataset and plot the results in Fig. 13. As we can see from Fig. 13a, MRFSE runs faster than P-MRFSE and MIRAGE. The reason is twofold. First, compared with both P-MRFSE and MIRAGE, MRFSE maintains embeddings locally, and hence avoiding expensive communication cost; second, compared with P-MRFSE, MRFSE merges two MapReduce jobs in P-MRFSE into one MapReduce job, eliminating the overhead of issuing an extra MapReduce job; Besides, MRFSE mines frequent subgraphs without performing any isomorphism test operation which is imperative and expensive in MIRAGE. As pointed out in Sect. 2, P-MRFSE and MIRAGE follow similar generate-and-verification paradigm to mine frequent subgraphs. Like MRFSE, P-MRFSE mines frequent subgraphs without performing any isomorphism test operation. From this perspective, P-MRFSE performs better than MIRAGE. While MIRAGE merges two MapReduce jobs in P-MRFSE into one MapReduce job, in this way, MIRAGE performs better than P-MRFSE. Considering the above two factors together, we find that the performance of P-MRFSE and MIRAGE is fairly similar.

Compared with P-MRFSE and MIRAGE, the communication cost in MRFSE is reduced by up to three orders of magnitude. This is not surprising because both P-MRFSE and MIRAGE need to shuffle tremendous amount of embeddings from mappers to reducers for generating candidates in the current iteration or ready for generating candidates in the next iteration, while MRFSE simply maintains embeddings locally.

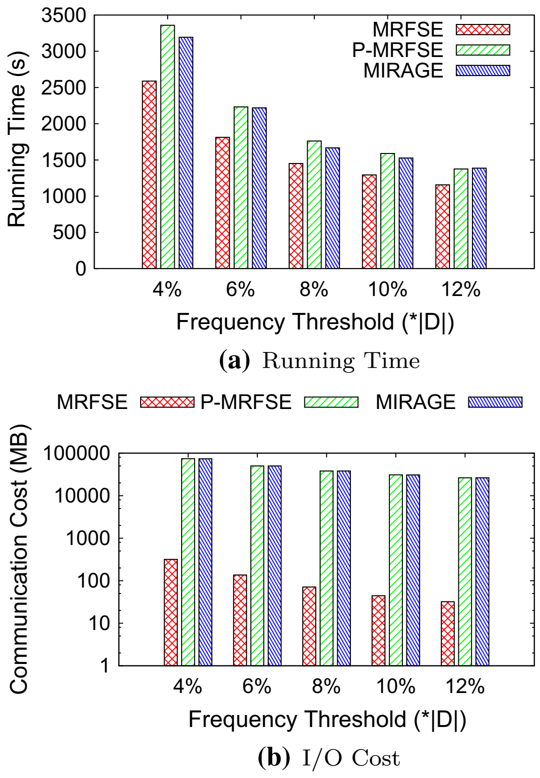


Fig. 13 Comparison over PubChem dataset. **a** Running time and **b** I/O cost

Table 3 Parameters for synthetic datasets

Parameter	Range
Data size (× 1000 k)	0.5, **1**, 1.5, 2
# of vertex labels	20
# of edge labels	1
Averaged graph size (# of edges)	10, 15, **20**, 25

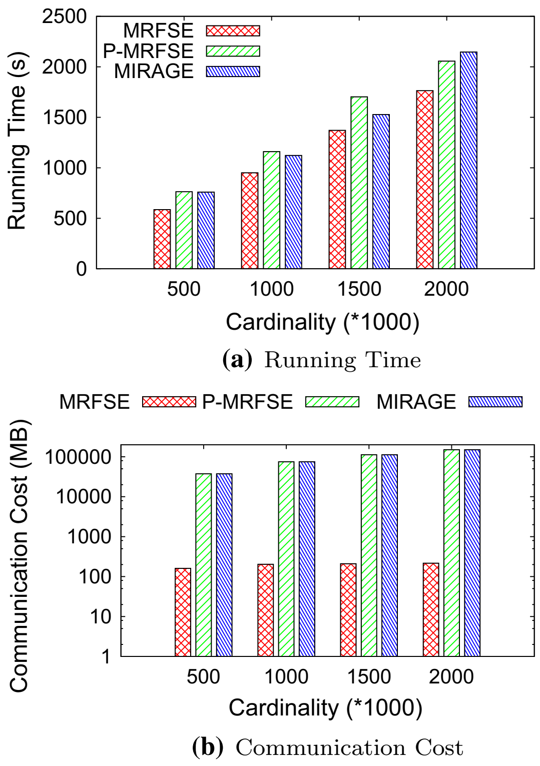
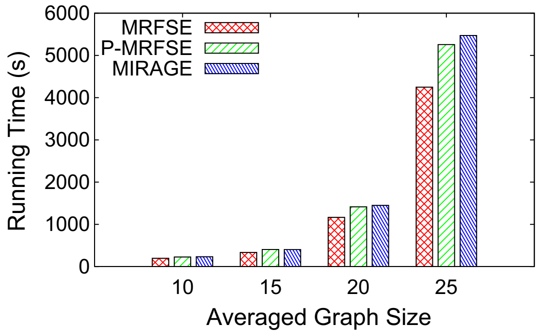


Fig. 14 Scalability. **a** Running time, **b** communication cost

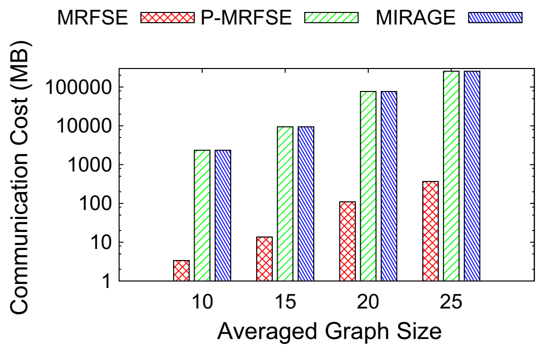
6.5 Scalability

We evaluate the scalability of MRFSE together with other existing approaches over the synthetic dataset. The generation parameters of the synthetic datasets are summarized in Table 3. In each experiment, we adjust a single parameter while keeping the rest using their default values (in bold). The frequent threshold for all experiments is set to 6%.

We plot the results in Fig. 14 by varying the number of graphs. First, MRFSE runs faster than P-MRFSE and MIRAGE by a factor of 11–22%. Second, when the number of graphs varies, the running time of all three approaches increases linearly. The reason is that although the cardinality of the dataset increases, the number of frequent subgraphs almost remains constant. For this reason, the number of frequent subgraphs that are contained in each graph almost remains the same. The running time (communication cost) of the above three approaches depends on that over every individual graph, together with the cardinality of the dataset. Therefore, both the communication cost and running time of them increase



(a) Running Time



(b) Communication Cost

Fig. 15 Effect of averaged graph size. **a** Running time, **b** communication cost

linearly with the number of graphs. Third, MRFSE reduces the communication cost by up to three orders of magnitude. The reason is that MRFSE maintains the embeddings locally and only shuffles the minimum DFS codes of frequent subgraph candidates from mappers to reducers.

6.6 Effect of averaged graph size

We finally investigate the performance by varying the averaged graph size and plot the results in Fig. 15. MRFSE runs faster than P-MRFSE and MIRAGE by a factor up to 20%, and there is an obviously increasing benefit when the averaged size of graph increases. Besides, when the averaged graph size varies, the running time and the communication cost of above three approaches increase exponentially. The reason is that the number of frequent subgraphs increases exponentially when the averaged graph size varies, which results in an exponential increase of the number of frequent subgraphs that a graph contains.

7 Conclusion

In this paper, we study the problem of efficiently mining frequent subgraphs from a tremendous amount of small graphs in a distributed environment, particularly, using the MapReduce framework. We propose an efficient and scalable solution, called MRFSE, to iteratively mine frequent subgraphs. At each iteration, in the map phase, MRFSE issues parallel scan over graphs in the dataset, extracts frequent subgraph candidates from every graph separately, and shuffles the minimum DFS codes of these candidates to reducers. Existing approaches in centralized systems are easily extended to generate new subgraphs. In the reduce phase, reducers verify and output frequent subgraphs by aggregating the same minimum DFS codes. MRFSE poses three beneficial properties. First, new frequent subgraphs are generated without performing graph isomorphism test which is imperative and expensive in existing approaches; second, MRFSE is able to save a large amount of network bandwidth by maintaining tremendous amount of embeddings locally. Third, as processing each graph separately, compared with existing approaches, MRFSE consumes less memory cost. Extensive experiments conducted on our in-house clusters demonstrate that the techniques built using proposed framework are scalable and efficient.

Acknowledgements We would like to thank the anonymous reviewers for their helpful and insightful comments. This work was in part supported by the National Natural Science Foundation of China (61502504, 61502347, 61432006), the Nature Science Foundation of Hubei Province of China (2016CFB384), the Ministry of Science and Technology of China, National Key Research and Development Program (Project Number: 2016YFB1000700) and the Fundamental Research Funds for the Central Universities, the Research Funds of Renmin University of China No. 15XNLF09.

References

1. Aridhi S, d'Orazio L, Maddouri M, Nguifo EM (2015) Density-based data partitioning strategy to approximate large-scale subgraph mining. Inf Syst 48:213–223
2. Berman HM, Westbrook J, Feng Z, Gilliland G, Bhat TN, Weissig H, Shindyalov IN, Bourne PE (2000) The protein data bank. Nucleic Acids Res 28:235–242
3. Bhuiyan M, Hasan MA (2015) An iterative mapreduce based frequent subgraph mining algorithm. IEEE Trans Knowl Data Eng 27(3):608–620
4. Borgelt C, Berthold MR (2002) Mining molecular fragments: finding relevant substructures of molecules. In: ICDM, pp 51–58
5. Chaoji V, Hasan MA, Salem S, Zaki MJ (2008) An integrated, generic approach to pattern mining: data mining template library. Data Min Knowl Discov 17(3):457–495
6. Cheng J, Ke Y, Ng W (2009) Efficient query processing on graph databases. ACM Trans Database Syst 34(1):2
7. Cheng J, Ke Y, Ng W, Lu A(2007) Fg-index: towards verification-free query processing on graph databases. In: SIGMOD conference, pp 857–872
8. Dean J, Ghemawat S (2004) MapReduce: simplified data processing on large clusters. In: OSDI, pp 137–150
9. Han J (2005) Data mining: concepts and techniques. Morgan Kaufmann Publishers Inc., San Francisco
10. Hill S, Srichandan B, Sunderraman R (2012) An iterative mapreduce approach to frequent subgraph mining in biological datasets. In: BCB, pp 661–666
11. Huan J, Wang W, Prins J (2003) Efficient mining of frequent subgraphs in the presence of isomorphism. In: ICDM, pp 549–552
12. Huan J, Wang W, Prins J, Yang J (2004) Spin: mining maximal frequent subgraphs from graph databases. In: KDD, pp 581–586
13. Inokuchi A, Washio T, Motoda H (2000) An apriori-based algorithm for mining frequent substructures from graph data. In: PKDD, pp 13–23
14. Kanehisa M, Goto S (2000) KEGG: Kyoto encyclopedia of genes and genomes. Nucleic Acids Res 28(1):27–30
15. Kuramochi M, Karypis G (2001) Frequent subgraph discovery. In: ICDM, pp 313–320
16. Lin W, Xiao X, Ghinita G (2014) Large-scale frequent subgraph mining in mapreduce. In: IEEE 30th international conference on data engineering, Chicago, ICDE 2014, IL, USA, 31 March–4 April, pp 844–855
17. Liu Y, Jiang X, Chen H, Ma J, Zhang X (2009) Mapreduce-based pattern finding algorithm applied in motif detection for prescription compatibility network. In: Advanced parallel processing technologies, 8th international symposium, APPT 2009, Rapperswil, Switzerland, Proceedings, 24–25 Aug, pp 341–355

18. Lowe DG (2001) Local feature view clustering for 3D object recognition. In: CVPR, pp 682–688
19. Lu W, Chen G, Tung AKH, Zhao F (2013) Efficiently extracting frequent subgraphs using mapreduce. In: Proceedings of the 2013 IEEE international conference on big data, Santa Clara, CA, USA, 6–9 Oct 2013, pp 639–647
20. National library of medicine. http://chem.sis.nlm.nih.gov/chemidplus
21. Nijssen S, Kok JN (2004) A quickstart in frequent structure mining can make a difference. In: KDD, pp 647–652
22. Petrakis EGM, Faloutsos C (1997) Similarity searching in medical image databases. IEEE Trans Knowl Data Eng 9(3):435–447
23. Wang C, Wang W, Pei J, Zhu Y, Shi B (2004) Scalable mining of large disk-based graph databases. In: KDD, pp 316–325
24. Yan X, Han J (2002) gspan: graph-based substructure pattern mining. In: ICDM, pp 721–724
25. Yan X, Yu PS, Han J (2004) Graph indexing: a frequent structure-based approach. In: SIGMOD conference, pp 335–346

Zhe Peng is currently working for the Ph.D. degree in School of Information and DEKE, MOE, Renmin University of China, Beijing, China. She is a student member of China Computer Federation. Her research interests include data management and analytics.

Tongtong Wang received the MS degree in Shandong Jianzhu University, China, in 2015. He is currently working toward the Ph.D. degree in School of Information and DEKE, MOE, Renmin University of China. He is a student member of China Computer Federation. His research interests include data management and analytics.

Wei Lu is currently an associate professor at Renmin University of China. He received his Ph.D. degree in Computer Science from Renmin University of China in 2011. His research interest includes query processing in the context of spatiotemporal, cloud database systems, and applications.

Hao Huang received a Ph.D. degree in Computer Science from Zhejiang University, China, in 2012. He is currently an Associate Professor at State Key Laboratory of Software Engineering, Wuhan University, China. His research interests include data management and analytics, data mining, and intelligent information systems. He is a member of the ACM and the CCF.

Xiaoyong Du is a professor at Renmin University of China. He received his Ph.D. degree from Nagoya Institute of Technology in 1997. His research focuses on intelligent information retrieval, high performance database and unstructured data management.

Feng Zhao is a Data Scientist at Expedia, London. He received his Ph.D. degree in School of Computing from National University of Singapore in 2014. His research interest includes deep learning, graph mining and recommendation system.

Anthony K. H. Tung received the BSc (Second Class Honor) and MSc degrees in computer science from the National University of Singapore (NUS) in 1997 and 1998, respectively, and the Ph.D. degree in computer sciences from Simon Fraser University (SFU) in 2001. He is currently an Associate Professor in the Department of Computer Science, National University of Singapore. His research interests involve various aspects of databases and data mining (KDD) including buffer management, frequent pattern discovery, spatial clustering, outlier detection, and classification analysis.