

An Iterative MapReduce Based Frequent Subgraph Mining Algorithm

Mansurul A. Bhuiyan and Mohammad Al Hasan

Abstract—Frequent subgraph mining (FSM) is an important task for exploratory data analysis on graph data. Over the years, many algorithms have been proposed to solve this task. These algorithms assume that the data structure of the mining task is small enough to fit in the main memory of a computer. However, as the real-world graph data grows, both in size and quantity, such an assumption does not hold any longer. To overcome this, some graph database-centric methods have been proposed in recent years for solving FSM; however, a distributed solution using MapReduce paradigm has not been explored extensively. Since MapReduce is becoming the de-facto paradigm for computation on massive data, an efficient FSM algorithm on this paradigm is of huge demand. In this work, we propose a frequent subgraph mining algorithm called FSM-H which uses an iterative MapReduce based framework. FSM-H is complete as it returns all the frequent subgraphs for a given user-defined support, and it is efficient as it applies all the optimizations that the latest FSM algorithms adopt. Our experiments with real life and large synthetic datasets validate the effectiveness of FSM-H for mining frequent subgraphs from large graph datasets. The source code of FSM-H is available from www.cs.iupui.edu/~alhasan/software/

Index Terms—Frequent sub-graph mining, iterative MapReduce

1 INTRODUCTION

IN recent years, the “big data” phenomenon has engulfed a significant number of research and application domains including data mining, computational biology [1], environmental sciences, e-commerce [2], web mining, and social network analysis [3]. In these domains, analyzing and mining of massive data for extracting novel insights has become a routine task. However, traditional methods for data analysis and mining are not designed to handle massive amount of data, so in recent years many such methods are re-designed and re-implemented under a computing framework that is better equipped to handle big-data idiosyncrasies.

Among the recent efforts for building a suitable computing platform for analyzing massive data, the MapReduce [4] framework of distributed computing has been the most successful. It adopts a data centric approach of distributed computing with the ideology of “moving computation to data”; besides it uses a distributed file system that is particularly optimized to improve the IO performance while handling massive data. Another main reason for this framework to gain attention of many admirers is the higher level of abstraction that it provides, which keeps many system level details hidden from the programmers and allow them to concentrate more on the problem specific computational logic.

MapReduce has become a popular platform for analyzing large networks in recent years. However, the majority of

such analyses are limited to estimating global statistics (such as diameter) [5], spectral analysis [6], or vertex-centrality analysis [5]. There also exist some works that mine (and count) sub-structures from a large network. For instance, Suri and Vassilvitskii [7] and Pagh and Tsourakakis [8] use MapReduce for counting triangles, Afrati et al. [9] use MapReduce for enumerating the instances of a query graph in a large network, and Bahmani et al. [10] mine densest subgraphs in a large graph. However, mining frequent subgraphs from a graph database has received the least attention. Given the growth of applications of frequent subgraph mining (FSM) in various disciplines including social networks, bioinformatics [11], cheminformatics [12], and semantic web [13], a scalable method for frequent subgraph mining on MapReduce is of high demand.

Solving the task of frequent subgraph mining on a distributed platform like MapReduce is challenging for various reasons. First, an FSM method computes the support of a candidate subgraph pattern over the entire set of input graphs in a graph dataset. In a distributed platform, if the input graphs are partitioned over various worker nodes, the local support of a subgraph in the respective partition at a worker node is not much useful for deciding whether the given subgraph is frequent or not. Also, local support of a subgraph in various nodes cannot be aggregated in a global data structure, because, **MapReduce programming model does not provide any built-in mechanism for communicating with a global state**. Also, the support computation cannot be delayed arbitrarily, as following Apriori principle [14], future candidate frequent patterns¹ can be generated only from a frequent pattern.

• The authors are with the Department of Computer Science, Indiana University—Purdue University, Indianapolis, IN.
E-mail: {mbhuiyan, alhasan}@cs.iupui.edu.

Manuscript received 17 Oct. 2013; revised 30 June 2014; accepted 14 July 2014. Date of publication 4 Aug. 2014; date of current version 28 Jan. 2015.

Recommended for acceptance by B. Cooper.

For information on obtaining reprints of this article, please send e-mail to: reprints@ieee.org, and reference the Digital Object Identifier below.

Digital Object Identifier no. 10.1109/TKDE.2014.2345408

1. In data mining, the word *pattern* is generally used to denote a combinatorial object, such as a set, a sequence, a tree or a graph. In this paper, we use pattern to denote a graph object only.

In this paper, we propose, FSM-H², a distributed frequent subgraph mining method over MapReduce. Given a graph database, and a **minimum support threshold**, FSM-H generates a complete set of frequent subgraphs. To ensure completeness, it constructs and retains all patterns in a partition that has a non-zero support in the map phase of the mining, and then in the reduce phase, it decides whether a pattern is frequent by aggregating its support computed in all partitions from different computing nodes. To overcome the dependency among the states of a mining process, FSM-H runs in an iterative fashion, **where the output from the reducers of iteration $i - 1$ is used as an input for the mappers in the iteration i .** The mappers of iteration i generate candidate subgraphs of size i (number of edge), and also compute the local support of the candidate pattern. The reducers of iteration i then find the true frequent subgraphs (of size i) by aggregating their local supports. They also write the data in disk that are processed in subsequent iterations.

We claim the following contributions:

- We introduce, FSM-H, a novel iterative MapReduce based frequent subgraph mining algorithm, which is complete.
- We design novel data structures to save and consequently propagate the states of the mining process over different iterations.
- We empirically demonstrate the performance of FSM-H on synthetic as well as real world large datasets.

2 RELATED WORKS

There exist many algorithms for solving the in-memory version of frequent subgraph mining task, most notable among them are AGM [15], FSG [16], gSpan [17], Gaston [18], and DMTL [19]. These methods assume that the dataset is small and the mining task finishes in a reasonable amount of time using an in-memory method. To consider the large data scenario, a few traditional database based graph mining algorithms, such as, DB-Subdue [20], and DB-FSG [21] and OO-FSG [22] are also proposed.

For large-scale graph mining tasks, researchers considered shared memory parallel algorithms for frequent subgraph mining. Cook et al. presented a parallel version of their frequent subgraph mining algorithm Subdue [23]. Wang et al. developed a parallel toolkit [24] for their Motif-Miner [25] algorithm. Meinel et al. created a software named Parmol [26] which includes parallel implementation of Mofa [27], gSpan [17], FFSG [28] and Gaston [18]. ParSeMis [29] is another such tool that provides parallel implementation of gSpan algorithm. To deal with the scalability problem caused by the size of input graphs, there are couple of notable works, PartMiner [30] and PartGraphMining [31], which are based on the idea of partitioning the graph data. There also exists an work [32] on adaptive parallel graph mining for CMP Architectures.

MapReduce framework has been used to mine frequent patterns where the transactions in the input database are simpler combinatorial objects such as, a set [33], [34], [35],

[36], or a sequence [37]. In [38], the authors consider frequent subgraph mining on MapReduce, however, their approach is inefficient due to various shortcomings. The most notable is that in their method they do not adopt any mechanism to avoid generating duplicate patterns. This cause an exponential increase in the size of the candidate subgraph space; furthermore, the output set contains duplicate copy of the same graph patterns that are hard to unify as the user has to provide a subgraph isomorphism routine for this amendment. Another problem with the above method is that it requires the user to specify the number of MapReduce iterations. Authors did not mention how to determine the total iteration count so that the algorithm is able to find all frequent patterns for a given support. One feasible way might be to set the iteration count to be the edge count of the largest transaction but that will be an overkill of the resources. FSM-H does not suffer from any of the above limitations. During the revision phase of this journal, we became aware of another work [39] of frequent subgraph mining on MapReduce. It is a non-iterative method which runs Gaston [18] on each partition of the graph database.

There exist some works [40], [41] that mine subgraphs that are frequent considering their induced occurrences in a single large graph. However, they are different as the objective of FSM-H is to mine subgraphs that are frequent over a collection of graphs in a graph database.

3 BACKGROUND

3.1 Frequent Sub-Graph Mining

Let, $\mathcal{G} = \{G_1, G_2, \dots, G_n\}$ be a graph database, where each $G_i \in \mathcal{G}, \forall i = \{1 \dots n\}$ represents a labeled, undirected, simple (no multiple edges between a pair of vertices), and connected graph. For a graph g , its size is defined as the number of edges it contains. Now, $\mathbf{t}(g) = \{G_i : g \subseteq G_i \in \mathcal{G}, \forall i = \{1 \dots n\}\}$, is the *support-set* of the graph g (here the subset symbol denotes a subgraph relation). Thus, $\mathbf{t}(g)$ contains all the graphs in \mathcal{G} that has a subgraph isomorphic to g . The cardinality of the *support-set* is called the *support* of g . g is called frequent if $\text{support} \geq \pi^{\min}$, where π^{\min} is predefined/user-specified *minimum support* (*minsup*) threshold. The set of frequent patterns are represented by \mathcal{F} . Based on the size (number of edges) of a frequent pattern, we can partition \mathcal{F} into a several disjoint sets, \mathcal{F}_i such that each of the \mathcal{F}_i contains frequent patterns of size i only.

Example. Fig. 1a shows a database with three vertex labeled graphs (G_1, G_2 and G_3). With $\pi^{\min} = 2$, there are 13 frequent subgraphs as shown in Fig. 1b. Note that, For the sake of simplicity in this example we assume that the edges of the input graphs are unlabeled. But FSM-H is designed and developed to handle labels both on vertices and edges.

3.2 MapReduce

MapReduce is a programming model that enables distributed computation over massive data [4]. The model provides two abstract functions: *map*, and *reduce*. Map corresponds to the “map” function and “reduce” corresponds to the “fold” function in functional programming. Based on its role, a worker node in MapReduce is called a

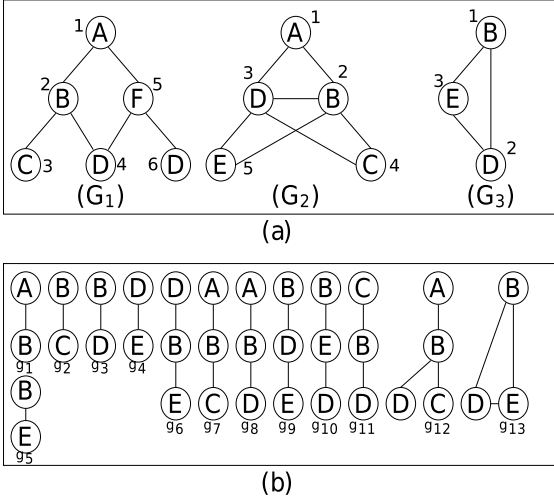


Fig. 1. (a) Graph database with three graphs with labeled vertices (b) Frequent subgraph of (a) with $minsup = 2$.

mapper or a reducer. A mapper takes a collection of (key, value) pairs and applies the map function on each of the pairs to generate an arbitrary number of (key, value) pairs as intermediate output. The reducer aggregates all the values that have the same key in a sorted list, and applies the reduce function on that list. It also writes the output to the output file. The files (input and output) of MapReduce are managed by a distributed file system. Hadoop is an open-source implementation of MapReduce programming model written in Java language.

3.2.1 Iterative MapReduce

Iterative MapReduce [42] can be defined as a multi staged execution of map and reduce function pair in a cyclic fashion, i.e. the output of the stage i reducers is used as an input of the stage $i + 1$ mappers. An external condition decides the termination of the job. Pseudo code for iterative MapReduce algorithm is presented in Fig. 2.

4 METHOD

FSM-H is designed as an iterative MapReduce process. At the beginning of iteration i , FSM-H has at its disposal all the frequent patterns of size $i - 1$ (\mathcal{F}_{i-1}), and at the end of iteration i , it returns all the frequent patterns of size i , (\mathcal{F}_i). Note that, in this work, the size of a graph is equal to the number of edges it contains. For a mining task if \mathcal{F} is the set of frequent patterns, FSM-H runs for a total of l iterations, where l is equal to the size of the largest graph in \mathcal{F} .

To distribute a frequent subgraph mining task, FSM-H partitions the graph dataset $\mathcal{G} = \{G_i\}_{i=1..n}$ into k disjoint partitions, such that each partition contains roughly equal number of graphs; thus it mainly distributes the support

Iterative_MapReduce():

1. While(Condition)
2. Execute MapReduce Job
3. Write result to DFS
4. Update condition

Fig. 2. Iterative MapReduce algorithm.

```
//  $\mathcal{G}$  is the database
//  $k$  is initialize to 1
Mining Frequent Subgraph( $\mathcal{G}, minsup$ ):
0. Populate  $\mathcal{F}_1$ 
1. while  $\mathcal{F}_k \neq \emptyset$ 
2.    $\mathcal{C}_{k+1} = \text{Candidate generation}(\mathcal{F}_k, \mathcal{G})$ 
2.   forall  $c \in \mathcal{C}_{k+1}$ 
3.     if isomorphism checking( $c$ ) = true
4.       support counting( $c, \mathcal{G}$ )
5.       if  $c.support \geq minsup$ 
6.          $\mathcal{F}_{k+1} = \mathcal{F}_{k+1} \cup \{c\}$ 
7.        $k = k + 1$ 
8. return  $\bigcup_{i=1..k-1} \mathcal{F}_i$ 
```

Fig. 3. Frequent subgraph mining algorithm with breadth-first candidate enumeration.

counting (discussed in details later) subroutine of a frequent pattern mining algorithm. Conceptually, each node of FSM-H runs an independent FSM task over a graph dataset which is $1/k$ 'th of the size of $|\mathcal{G}|$. Note that, k is an important parameter and details on how to choose it optimally is explained in Sections 4.3.1 and 5.5. The FSM algorithm is an adaptation of the baseline algorithm shown in Fig. 3, which runs in a sequential machine. Below we provides more details of this algorithm.

4.1 Baseline Frequent Subgraph Mining Algorithm

The pseudo-code shown in Fig. 3 implements an FSM algorithm that follows a typical candidate-generation-and-test paradigm with breadth-first candidate enumeration. In this paradigm, the mining task starts with frequent patterns of size one (single edge patterns), denoted as \mathcal{F}_1 (Line 0). Then in each of the iterations of the while loop (Line 1-6), the method progressively finds $\mathcal{F}_2, \mathcal{F}_3$ and so on until the entire frequent pattern set (\mathcal{F}) is obtained. If \mathcal{F}_k is non-empty at the end of an iteration of the above while loop, from each of the frequent patterns in \mathcal{F}_k the mining method creates possible candidate frequent patterns of size $k+1$ (Line 2). These candidate patterns are represented as the set \mathcal{C} . For each of the candidate patterns, the mining method computes the pattern's support against the dataset \mathcal{G} (Line 5). If the support is higher than the minimum support threshold ($minsup$), the given pattern is frequent, and is stored in the set \mathcal{F}_{k+1} (Line 6). Before support counting, the method also ensures that different isomorphic forms of a unique candidate patterns are unified and only one such copy is processed by the algorithm (Line 4). Once all the frequent patterns of size $k + 1$ are obtained, the while loop in Line 1 to 7 continues. Thus each iteration of the while loop obtains the set of frequent patterns of a fixed size, and the process continues until all the frequent patterns are obtained. In Line 8, the FSM algorithm returns the union of $\mathcal{F}_i : 1 \leq i \leq k - 1$.

Below, we provide a short description of each of the sub-routines that are called in the pseudo-code.

4.1.1 Candidate Generation

Given a frequent pattern (say, c) of size k , this step adjoins a frequent edge (which belongs to \mathcal{F}_1) with c to obtain a

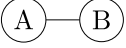
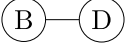
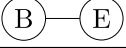


Pattern	Occurrence List (OL)
	1 : [(1, 2)] ; 2 : [(1, 2)]
	1 : [(2, 4)] ; 2 : [(2, 3)] ; 3 : [(1, 2)]
	2 : [(2, 5)] ; 3 : [(1, 3)]
	1 : [(1, 2), (2, 4)] ; 2 : [(1, 2), (2, 3)]
	2 : [(1, 2), (2, 5)]

Fig. 6. Support counting.

$label_1, label_2$). Using the same coding scheme, the pattern in Fig. 5b maps to the string $(1, 2, A, B)(2, 3, B, D)(2, 4, B, C)$. However, the min-dfs-code of the pattern $B - \{A, C, D\}$ is $(1, 2, A, B)(2, 3, B, C)(2, 4, B, D)$, which matches with the isomorphic form shown in Fig. 5a; thus the pattern will only be generated by extending $A - B - C$. Other generation paths, including the one that extends $A - B - D$ are invalid and hence are ignored after performing isomorphism checking.

4.1.3 Support Counting

Support counting of a graph pattern g is important to determine whether g is frequent or not. To count g 's support we need to find the database graphs in which g is embedded. This mechanism requires solving a subgraph isomorphism problem, which is \mathcal{NP} -complete. One possible way to compute the support of a pattern without explicitly performing the subgraph isomorphism test across all database graphs is to maintain the occurrence-list (OL) of a pattern; such a list stores the embedding of the pattern (in terms of vertex id) in each of the database graphs where the pattern exists. When a pattern is extended to obtain a child pattern in the candidate generation step, the embedding of the child pattern must include the embedding of the parent pattern, thus the occurrence-list of the child pattern can be generated efficiently from the occurrence list of its parent. Then the support of a child pattern can be obtained trivially from its occurrence-list.

Example. In Fig. 6 we illustrate a simple scenario of support counting based on occurrence list. In the top three rows of Fig. 6, we show the OL of the pattern $A - B$, $B - D$ and $B - E$. The Pattern $A - B$ occurs in Graph G_1 and G_2 in vertex pair $(1, 2)$ and $(1, 2)$, respectively; so its OL is: 1:[(1,2)]; 2:[(1,2)]. If we adjoin $B - D$ with the pattern $A - B$ and form the pattern $A - B - D$, then we can construct the OL of $A - B - D$ (shown in 4th row) by intersecting the OLs of $A - B$ and $B - D$. Note that, the intersection considers both the graph ids and the vertex ids in the OLs. By counting the graph ids present in an OL we can compute the support of that pattern. In Fig. 6, the pattern $A - B - D$ is frequent given minimum support 2 but the pattern $A - B - E$ is not frequent.

```

Mapper_FSG( $\mathcal{F}_k^p \langle x.min\text{-}dfs\text{-}code, x.obj \rangle$ ):
1.  $\mathcal{C}_{k+1} = \text{Candidate\_generation}(\mathcal{F}_k^p)$ 
2. forall  $c \in \mathcal{C}_{k+1}$ 
3.   if isomorphism\_checking( $c$ ) = true
4.     populate\_occurrence\_List( $c$ )
5.     if  $length(c.occurrence\_List) > 0$ 
6.       emit ( $c.min\text{-}dfs\text{-}code$  ,  $c.obj$ )

```

Fig. 7. Mapper of distributed frequent subgraph mining algorithm.

4.2 Distributed Paradigm of FSM-H

An important observation regarding the baseline FSM algorithm (Fig. 3) is that it obtains all the frequent patterns of size k in one iteration of while loop from Line 1 to Line 6. The tasks in such an iteration comprise to one MapReduce iteration of FSM-H. Another observation is that, when the FSM algorithm generates the candidates of size $k + 1$, it requires the frequent patterns of size k (\mathcal{F}_k). In an iterative MapReduce, there is no communication between subsequent iterations. So, $k + 1$ 'th iteration of FSM-H obtains \mathcal{F}_k from the disk which is written by the reducers at the end of the k 'th iteration. A final observation is that, deciding whether a given pattern is frequent or not requires counting its support over all the graphs in the dataset (\mathcal{G}). However, as we mentioned earlier each node of FSM-H works only on a disjoint partition of \mathcal{G} . So, FSM-H requires to aggregate the local support from each node to perform the task in Line 5. From the above observations we identify the distribution of mining task of FSM-H among the mappers and the reducers.

Fig. 7 shows the pseudo-code of a mapper. The argument \mathcal{F}_k^p represents the set of size- k frequent subgraphs having non-zero support in a specific partition p . The mapper reads it from Hadoop Distributed File Systems (HDFS). Each pattern (say x) in \mathcal{F}_k^p is read as a key-value pair. The key is the min-dfs-code of the pattern ($x.min\text{-}dfs\text{-}code$) and the value is a pattern object ($x.obj$); here "object" stands for its usual meaning from the object oriented programming. This pattern object contains all the necessary information of a pattern i.e., its support, neighborhood lists, and occurrence list within partition p . It also contains additional data structure that are used for facilitating candidate generation from this pattern in the same partition. We will discuss the pattern object in details in a later section. The mapper then generates all possible candidates of size $k + 1$ (Line 1) by extending each of the patterns in \mathcal{F}_k^p . For each of the generated candidates (say, c), the mapper performs isomorphism checking to confirm whether c is generated from a valid generation path; in other words, it tests whether c passes the min-dfs-code based isomorphism test (Line 3). For successful candidates, the mapper populates their occurrence list (Line 4) over the database graphs in the partition p . If the occurrence list of a candidate pattern is non-empty, the mapper constructs a key-value pair, such as, ($c.min\text{-}dfs\text{-}code, c.obj$) and emits the constructed pair for the reducers to receive (Line 6).

Fig. 8 shows the pseudo code for a reducer in distributed frequent subgraph mining. The reducer receives a set of key-value pairs, where the key is the min-dfs-code of a

Reducer_FSG($c.min\text{-}dfs\text{-}code, \langle c.obj \rangle$):

1. **forall** $obj \in \langle c.obj \rangle$
2. $support += length(obj.OL)$
3. **if** $support \geq minsup$
4. **forall** $obj \in \langle c.obj \rangle$
5. **write** ($c.min\text{-}dfs\text{-}code, obj$) **to HDFS**

Fig. 8. Reducer of distributed frequent subgraph mining algorithm.

pattern namely $c.min\text{-}dfs\text{-}code$ and the value is a list of $c.obj$'s constructed from all partitions where the pattern c has a non-zero support. Reducer then iterates (Line 1) over every $c.obj$ and from the length of the occurrence list of each $c.obj$ it computes the aggregated support of c . If the aggregated support is equal or higher than the minimum support threshold (Line 3), the reducer writes each element in the list paired with the min-dfs-code of c in HDFS for the mappers of the next iteration.

Fig. 9 illustrates the execution flow of FSM-H. The execution starts from the mappers as they read the key-value pair of size k patterns in partition p from the HDFS. As shown in the Fig. 9, the mappers generate all possible $k+1$ -size candidate patterns and perform the isomorphism checking within partition p . For a pattern of size $k+1$ that passes the isomorphism test and has a non-zero occurrence, the mapper builds its key-value pair and emits that for the reducers. These key-value pairs are shuffled and sorted by the key field and each reducer receives a list of values with the same key field. The reducers then compute the support of the candidate pattern by aggregating the support value computed in the partitions where the respective pattern is successfully extended. If a pattern is frequent, the reducer writes appropriate key-value pairs in the HDFS for the mappers of the next iteration. If the number of frequent $k+1$ size pattern is zero, execution of FSM-H halts.

4.3 Framework of FSM-H

FSM-H has three important phases: data partition, preparation phase and mining phase. In data partition phase FSM-H creates the partitions of input data along with the omission of infrequent edges from the input graphs. Preparation

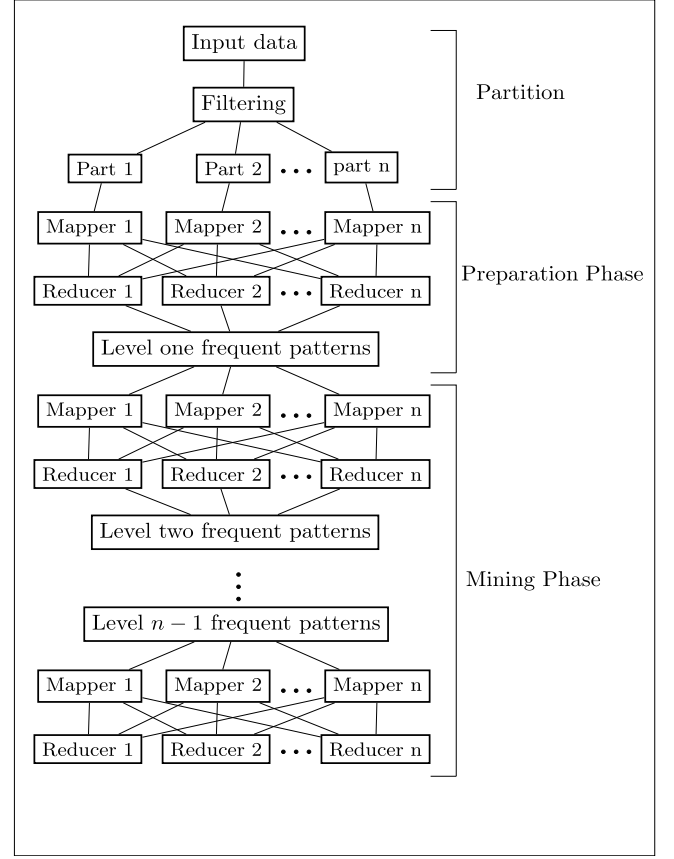


Fig. 10. (a) Framework of FSM-H.

and mining phase performs the actual mining task. Fig. 10 shows a flow diagram of different phases for a frequent subgraph mining task using FSM-H.

Below, we present an in-depth discussion of each of the phases.

4.3.1 Data Partition

In data partition phase, FSM-H splits the input graph dataset (\mathcal{G}) into many partitions. One straightforward partition scheme is to distribute the graphs so that each partition contains the same number of graphs from \mathcal{G} . This works well for most of the datasets. However, for datasets where the size (edge count) of the graphs in a dataset vary substantially, FSM-H offers another splitting option in which the total number of edges aggregated over the graphs in a partition are close to each other. In experiment section, we show that the latter partition scheme has a better runtime performance as it improves the load balancing factor of a MapReduce job. For FSM-H, the number of partition is also an important tuning parameter. In experiment section, we show that for achieving optimal performance, the number of partitions for FSM-H should be substantially larger than the number of partitions in a typical MapReduce task.

During the partition phase, input dataset also goes through a filtering procedure that removes the infrequent edges from all the input graphs. While reading the graph database for partitioning, FSM-H computes the support-list of each of the edges from which it identifies the edges that are infrequent for the given minimum support threshold.

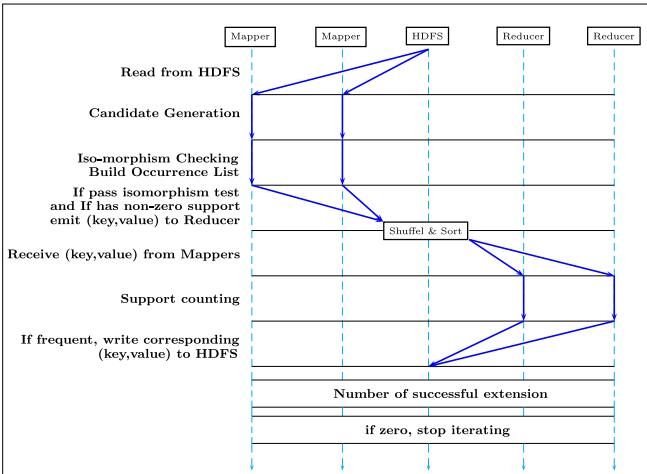


Fig. 9. Execution flow of FSM-H.

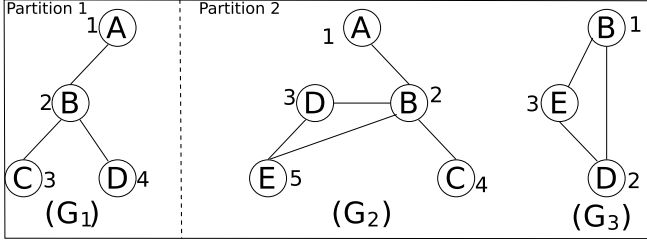


Fig. 11. Input data after partition and filtering phase.

Example. For the graph dataset in Fig. 1, for a minimum support threshold of 2, the edges $A - B$, $B - C$, $B - D$, $D - E$ and $B - E$ are frequent and the remaining edges are infrequent. Now suppose FSM-H makes two partitions for this dataset such that the first partition contains G_1 , and the second partition contains G_2 and G_3 . While making these partitions FSM-H filters the infrequent edges. Fig. 11 shows the partitioning where the infrequent edges are stripped off from the database graphs.

4.3.2 Preparation Phase

The mappers in this phase prepare some partition specific data structures such that for each partition there is a distinct copy of these data structures. They are static for a partition in the sense that they are same for all patterns generated from a partition. The first of such data structure is called *edge-extension-map*, which is used for any candidate generation that happens over the entire mining session. It stores the possible extension from a vertex considering the edges that exists in the graphs of a partition. For example, the graphs in partition two have edges such as $B - D$, $B - C$, $B - A$, and $B - E$. So, while generating candidates, if B is an extension stub, the vertex A , C , D or E can be the possible vertex label of the vertex that is at the opposite end of an adjoined edge. This information is stored in the Edge-extension-map data structure for each of the vertex label that exists in a partition. The second data structure is called *edge-OL*, it stores the occurrence list of each of the edges that exists in a partition; FSM-H uses it for counting the support of a candidate pattern which is done by intersecting the OL of a parent pattern with the OL of an adjoin edge.

Example. Figs. 12a and 12b shows these data structures for the Partition 1 and 2 defined in Fig. 11. In partition 1, the edge-extension choice from a vertex with label D is only B (shown as $D : (B)$), as in this partition $B - D$ is the only frequent edge with a B vertex. On the other hand, the corresponding choice for partition 2 is B and E (shown as, $D : (B; E)$), because in partition 2 we have two edges, namely $B - D$ and $D - E$ that involve the D vertex. In partition 1, the edge $B - D$ occurs in G_1 at vertex id (2, 4); on the other hand in partition 2, the same edge occurs in G_2 and G_3 at vertex id (2, 3) and (1, 2), respectively. These information are encoded in the *edge-OL* data structures of these partitions as shown in this figure.

The mappers in the preparation phase also start the mining task by emitting the frequent single edge patterns as key-value pair. Note that, since the partition phase have filtered out all the infrequent edges, all single edges that exist in any

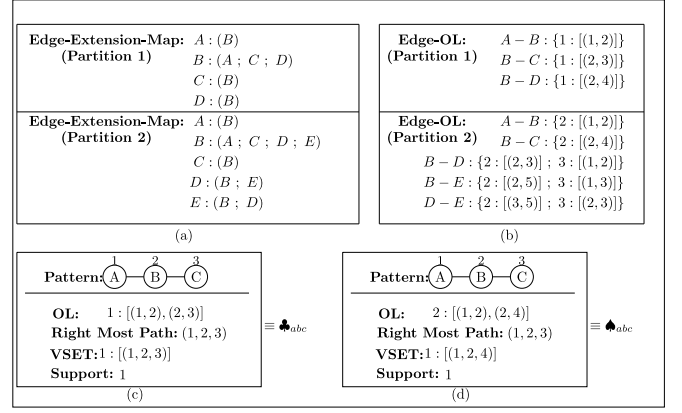


Fig. 12. The static data structures and $A - B - C$ pattern object in partition 1 and 2 (a) Edge-extension Map (b) Edge-OL (c) and (d) $A - B - C$ Pattern object.

graph of any partition is frequent. As we mentioned earlier the key of a pattern is its min-dfs-code and the value is the pattern object. Each pattern object have four essential attributes: (a) Occurrence List that stores the embedding of the pattern in each graph in the partition, (b) Right-Most-Path (c) VSET that stores the embedding of the Right Most Path in each graph in the partition, and (d) support value. Mappers in the preparation phase compute the min-dfs-code and create the pattern object for each single-edge patterns. While emitting a key-value pair to a reducer, the mappers also bundle the static data structures (SDS), edge-extension-map and edge-OL with each of the pattern object. FSM-H uses Java serialization to convert these objects in to byte stream while sending them as value in a key-value pair.

The reducers of this phase actually do nothing but writing the input key-value pairs in HDFS since all the single length patterns that the mappers send are frequent. In Fig. 10, the second block portrays the preparation phase.

Example. Figs. 12c and 12d exhibits the *Pattern* object along with their attributes for the pattern $A - B - C$ in partition 1 and 2, respectively. The attribute OL records the occurrence of this pattern in the corresponding database graphs; if a pattern has multiple embeddings in a database graph all such embeddings are stored. Right-Most-Path records the id of the right-most-path vertices in the pattern object and VSET stores the corresponding ids in the database graphs. Like OL, VSET is also a set and it stores information for multiple embedding if it applies. Finally, Support stores the support value of the pattern. In the following discussion, we use \clubsuit and \spadesuit to denote a pattern object from partition 1 (G_1) and 2 (G_2, G_3), respectively. For example, \spadesuit_{abc} identifies the pattern $A - B - C$ from partition 2 as shown in Fig. 12d.

4.3.3 Mining Phase

In this phase, mining process discovers all possible frequent subgraphs through iteration. Preparation phase populates all frequent subgraphs of size one and writes it in the distributed file system. Iterative job starts by reading these from HDFS. Each of the mappers of an ongoing iteration is responsible for performing the mining task over a particular chunk of the data written in HDFS by the preparation

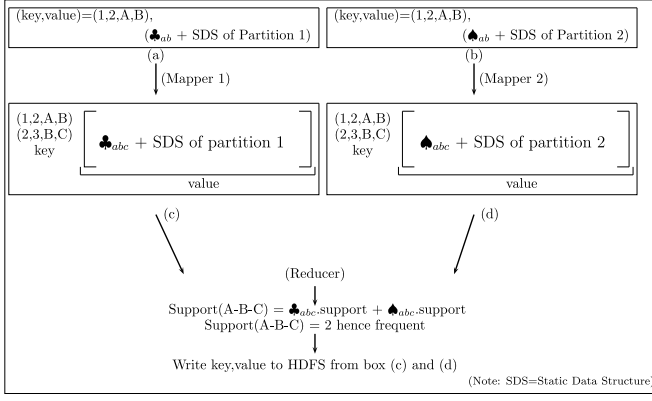


Fig. 13. One iteration of the mining phase of FSM-H with respect to pattern A-B.

phase. The map function of mining phase reconstructs all the static data structures that are required to generate candidate patterns from the current pattern. Using the static data structures and the pattern object, the mappers can independently execute the subroutine that is shown in Fig. 7. The reducers in this phase simply execute the routine in Fig. 8.

Example. Fig. 13 provides a detail walk-through of a Map-Reduce job in iteration phase with respect to the partitioned toy dataset mentioned in Fig. 11. Figs. 13a and 13b indicates the key-value pairs for pattern $A - B$ from partition 1 and 2. Suppose these key-value pairs are fed as input for Mapper 1 and Mapper 2 in Fig. 13. The mappers first extract all data structures from the value field of key-value pair including the partition-specific static data structures (in Fig. 13) such as edge-extension-map and edge-OL. Then they perform the steps mentioned in Fig. 7. Figs. 13c and 13d show the key-value pairs of $A - B - C$ that are generated by Mapper 1 and Mapper 2 by extending the pattern $A - B$. The Reducers collect all the values for a key and compute the support of the pattern by adding the supports from individual partitions. In this example, the support of the pattern $A - B - C$ is 2; since $minsup = 2$, this pattern is frequent. Reducers then write the key-value pairs corresponding to the pattern $A - B - C$ in HDFS.

4.4 Pseudo Code

In this section, we present the pseudo code of each phase using the syntax of Hadoop framework. Figs. 14, 15 and 16 demonstrate data partitioning, preparation and mining phase, respectively. In Lines 3 and 4 of Fig. 14, FSM-H performs the filtering and partitioning of the input dataset and

Partition_data(\mathcal{D}):

1. create a data directory in distributed file system
2. **while** data available in \mathcal{D}
3. $partition = \text{create_partition}()$
4. write $partition$ to file $partition_i$ in data directory

Fig. 14. Data partitioning phase.

```
// key = offset
// value = location of partition file in data directory
Mapper_preparation(Long key, Text value):
1. Generate_Level_one_OL(value)
2. Generate_Level_one_MAP(value)
3.  $P = \text{get\_single\_length\_patterns}()$ 
4. forall  $P_i$  in  $P$ :
5.    $intermediate\_key = \text{min\_dfs\_code}(P_i)$ 
6.    $intermediate\_value = \text{serialize}(P_i)$ 
7.    $\text{emit}(intermediate\_key, intermediate\_value)$ 

// key = min-dfs-code
// values = List of Byte-stream of a pattern object in all partitions
Reducer_preparation(Text key, BytesWritable { values }):
1. for all  $value$  in  $values$ :
2.    $\text{write\_to\_file}(key, value)$ 
```

Fig. 15. Preparation phase.

writes each partition and write in the HDFS. In Lines 1-7 of Fig. 15 the mappers generate static data structure and emit key-value pair of all single length patterns to reducer. Since all patterns are frequent, Reducers just relay the input key-value pair to the output file in HDFS. In Lines 1-3 of Mapper_mining function in Fig. 16, the mappers reconstruct the pattern object of size k along with the static data structures and generates the candidates from the current pattern. In Lines 4-9, the mappers iterate over all possible candidates of size $k + 1$ and on success in isomorphism and occurrence list test mappers emit the key-value pairs for the reducer. Reducer_mining function computes the aggregate support (Line 2) of the pattern and if the pattern is frequent, reducers write back the key-value pairs to HDFS for the mappers of the next iteration.

4.5 Implementation Detail

In this section, we explain some of the implementation details of FSM-H. We use Hadoop 1.1.2³ a open source implementation of MapReduce framework written in Java. We use Java to write the baseline mining algorithm as well as the map and the reduce function in the preparation and the mining phase. We override the default input reader and write a custom input reader for the preparation phase. To improve the execution time of MapReduce job, we compress the data while writing them in HDFS. We used global counter provided by Hadoop to track the stopping point of the iterative mining task.

4.5.1 MapReduce Job Configuration

In this section, we provide a detail snapshot of the Map-Reduce job configuration for FSM-H in Hadoop. Success of a job depends on the accurate configuration of that job. Since the type of the value that is read by a mapper and emit by a reducer is BytesWritable, FSM-H sets input and output format of each job as SequenceFileInputFormat and SequenceFileOutputFormat. Another job property,

3. <http://hadoop.apache.org/releases.html>


```

// key = min-dfs-code
// value = Byte-stream of pattern object for
iteration i-1
Mapper_mining(Long key, BytesWritable
value):
1. p = reconstruct_pattern(value)
2. reconstruct_all_data-structures(value)
3. P = Candidate_generation(p)
4. forall Pi in P:
5.   if pass_isomorphism_test(Pi) = true
6.     if length(Pi.OL) > 0
7.       intermediate_key = min-dfs-code(Pi)
8.       intermediate_value = serialize(Pi)
9.       emit(intermediate_key, intermediate_value)

// key = min-dfs-code
// values = List of Byte-stream of a pattern
object in all partitions
Reducer_mining(Text key, BytesWritable
values):
1. for all value in values:
2.   support += get_support(value)
3. if support ≥ minimum_support
4.   for all value in values:
5.     write_to_file(key, value)

```

Fig. 16. Mining phase.

named `mapred.task.timeout` also need to be set properly for better execution of FSM-H. This parameter controls the duration for which the master node waits for a data node to reply. If the mining task that FSM-H commence is computationally demanding, the default timeout which is 10 minutes may not be enough. To be on the safe side, FSM-H sets the timeout of a job to 5 hour (300 minutes). FSM-H also sets the `mapred.output.compress` property to true. This configuration lets the output of a MapReduce job to be compressed which eventually decreases network load and improves the overall execution time. The codec that is used for compression is `BZip2Codec`. FSM-H also increases the heap size of a job using `mapred.child.java.opts` property. The same configuration is used for both the preparation and mining phase of FSM-H.

5 EXPERIMENTS AND RESULTS

In this section, we present experimental results that demonstrate the performance of FSM-H for solving frequent subgraph mining task on large graph datasets. As input, we use six real-world graph datasets which are taken from an online source⁴ that contains graphs extracted from the PubChem website.⁵ PubChem provides information on biological activities of small molecules and the graph datasets from PubChem represent atomic structure of different molecules.

We also build a graph dataset from DBLP Computer Science Bibliography data⁶ considering publications in the year range 1992-2003. Each graph in this dataset is the ego-network of a distinct author (say, *u*); vertices in this graph is

TABLE 1
Statistics of Real Life Datasets

Transaction Graph dataset	# of Transactions (Graphs)	Average Size of each Graph
NCI-H23	40,353	28.6
OVCAR-8	40,516	28.1
SN12C	40,532	27.7
P388	41,472	23.3
Yeast	79,601	22.8
DBLP	129,073	14.70

u and his collaborators, and an edge between a pair of vertices represents one or more co-authorship events between the corresponding authors. Mining frequent subgraphs from this dataset is interesting, because it will generate subgraph patterns that correspond to a group of authors who collaborate more frequently. By keeping track of these groups over the time we will understand the dynamics of author collaboration. In Table 1, we provide the statistics of all real-life datasets.

We also create four synthetic datasets using a tool called Graphgen [43]. The number of graphs in these datasets range from 100 to 1000 K and each graph contains on average 25–30 edges. We conduct all experiments in a 10-node Hadoop cluster, where one of the node is set to be a master node and the remaining nine nodes are set to serve as data node. Each machine possesses a 3.1 GHz quad core Intel processor with 16GB memory and 1 TB of storage.

5.1 Runtime of FSM-H for Different Minimum Support

In this experiment, we analyze the runtime of FSM-H for varying minimum support threshold. We conduct this experiment for the real world datasets mention above. Here we fix the number of data nodes to 9 and keep track of the running time of FSM-H for minimum support thresholds that vary between 10 to 20 percent, except for DBLP dataset for which this is set between 3 – 6 percent. In Figs. 17a, 17b, 17c, 17d, 17e, and 17f, we show the result. As expected, the runtime decreases as minimum support threshold increases. In Figs. 18a, 18b, 18c, and 18d, we show the overall running time of FSM-H distributed over “Map” and “Reduce” phase. We pick “yeast” and “dblp” dataset for this experimentation. In all cases (different supports), “Reduce” phase takes large time to execute than the “Map” phase. Specifically, the running time of “Reduce” phase is approximately 70 percent of overall running time.

5.2 Runtime of FSM-H for Different Number of Database Graphs

For this experiment, we generate four synthetic datasets each having 100 to 1,000 K input graphs. Each of these graphs has 25 vertices and their edge density is around 0.5 or smaller. Here, for a graph $G(V, E)$ the edge density is defined as $|E|/\binom{|V|}{2}$. Table 2 shows the runtime of FSM-H for these datasets for 30 percent minimum support threshold using 1,000 partitions for each dataset. As we can see that, for all the cases mappers roughly consume 30 percent of the overall running time, and the reducers consume the

4. <http://www.cs.ucsb.edu/xyan/dataset.htm>

5. <http://pubchem.ncbi.nlm.nih.gov>

6. <http://www.informatik.uni-trier.de/~ley/db/>

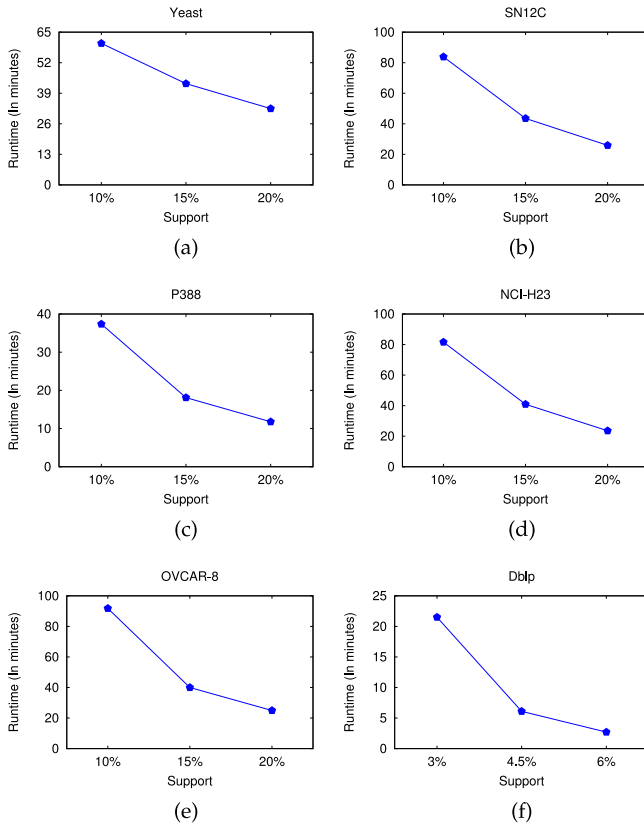


Fig. 17. Line plot showing the relationship between the minimum support threshold (percent) and the running time (in minutes) of FSM-H in (a) Yeast (b) NCI-H23 (c) P388 (d) SN12C and (e) OVCAR-8 (f) DBLP.

remaining time. One possible reason for a reducer to be slow is that in the reduce phase, a significant amount of data need to be shuffled and sorted. Moreover, a reducer sometimes requires to wait for a mapper to finish in order to collect the values corresponding to a key. As we can see, the overall running time increases at a sub-linear rate as the number of graphs in the database increases. Which shows the scalability of FSM-H.

Last column of Table 2 presents the total data communication over the networks in GigaBytes for different number of graphs. As we can see, communication size increases at a much larger rate with the number of graphs in the datasets, which is a bottleneck of FSM-H in its current implementation. One possible solution to improve the communication complexity is to use distributed cache for storing the static data structures so that they do not require to be piggybacked with the pattern objects over the networks. We plan to make this chance in a subsequent release of FSM-H software.

TABLE 2
Statistics of FSM-H on Synthetic Transaction Graph Datasets

Number of transaction	Runtime (in minute)	Map Time	Reduce Time	Comm. over Network (GB)
100 K	27.4	6.57	20.83	2.3
250 K	69.1	16.91	52.19	5.75
750 K	86.1	28.4	57.7	61
1000 K	98.7	33	65.7	78.76

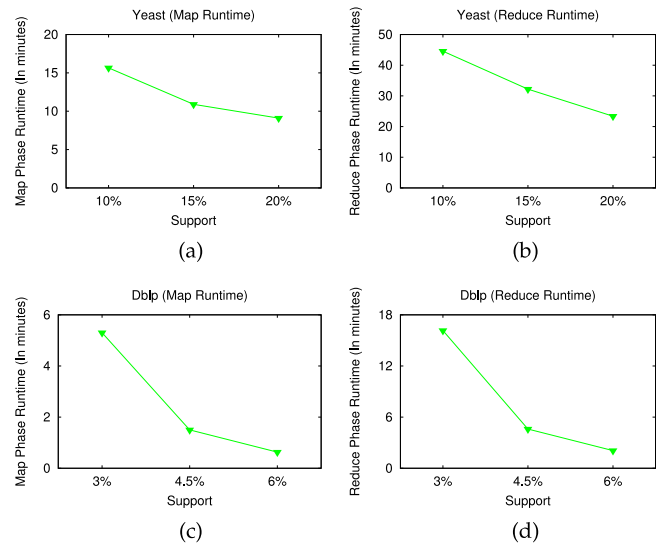


Fig. 18. Line plot showing the relationship between the minimum support threshold (percent) and the running time of Map and Reduce phase of FSM-H in (a) (b) Yeast (c) (d) DBLP.

5.3 Runtime of FSM-H on Varying Number of Data Nodes

In this experiment, we demonstrate how FSM-H's runtime varies with the number of active data nodes (slaves). We use the Yeast dataset and 20 percent minimum support threshold and keep 100 database graphs in one partition. We vary the count of data nodes among 3, 5, 7 and 9 and record the execution time for each of the configurations. As shown in Fig. 19a the runtime reduces significantly with an increasing number of data nodes. In Fig. 19b we plot the speedup that FSM-H achieves with an increasing number of data nodes, with respect to the three-data-nodes configuration. We can see that the speedup increases almost linearly except for the last data point.

5.4 Runtime of FSM-H for Varying Number of Reducer

The number of reducer plays an important role in the execution of MapReduce job in Hadoop. While writing data (output) in HDFS, a MapReduce job follows a convention of naming the output file with the key word "part". Reducer count determines how many "part" files will be generated to hold the output of a job. If the number of reducer is set to 1, entire output will be written in a single file. Since FSM-H is an iterative algorithm, where output of the current job is used as an input of the next job, the number of reducer has

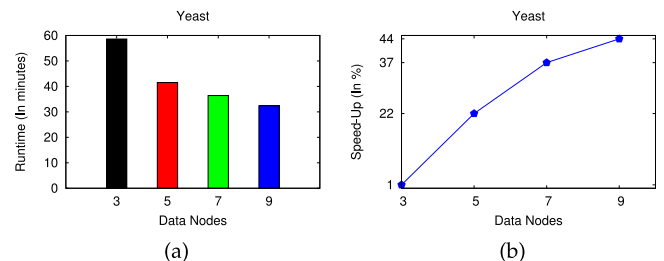


Fig. 19. Relationship between the execution time and the number of data nodes: (a) Bar plot shows the execution time; (b) Line plot shows the speedup with respect to the execution time using three data nodes configuration.

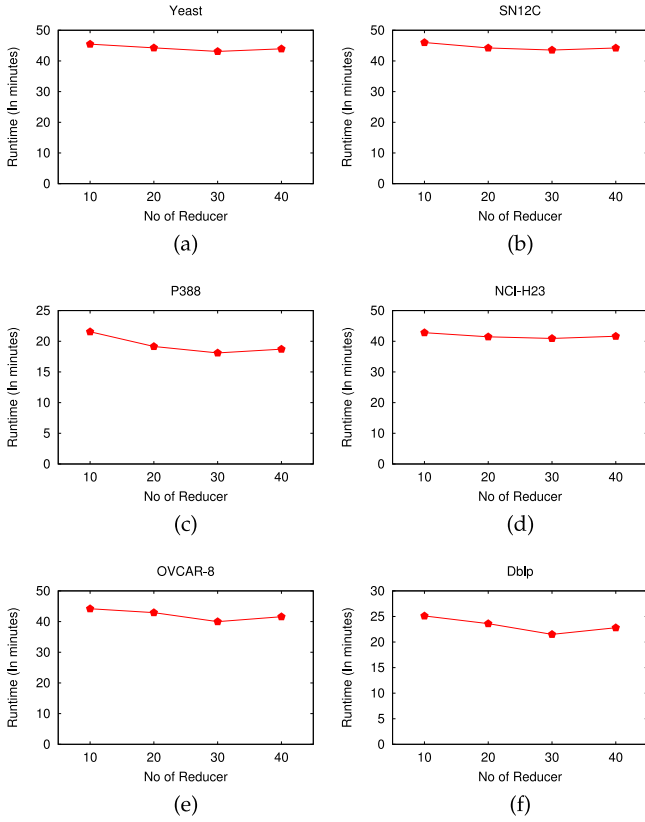


Fig. 20. Line plot showing the relationship between the number of Reducer and the running time of FSM-H in (a) Yeast, (b) NCI-H23, (c) P388, (d) SN12C, (e) OVCAR-8, and (f) DBLP.

a significant effects on the execution time of FSM-H. If we set reducer count to a small value then there will be fewer number of output files that are large in size; these large files will be a burden over the network when they are transferred between data nodes. On the other hand, large number of reducers might create many output files of zero size (reducer is unable to output any frequent pattern for the next stage Mapper). These zero size output files will also become an overhead for the next stage mappers as these files will still be used as input to a mapper. Note that, loading an input file is costly in Hadoop.

In this experiment, We measure the runtime of FSM-H for various configurations such as, 10, 20, 30 and 40 reducers. We run the experiment on biological datasets for 20 percent and DBLP for 3 percent minimum support threshold. We keep approximately 100 database graphs in each partition. Figs. 20a, 20b, 20c, 20d, 20e, and 20f shows the relationship between execution time and the number of reducers using line plot. As we can see, 30 is the best choice for the number of reducers in our cluster setup. This finding is actually intuitive, because we have nine data nodes each having four reduce slots (four core processor), yielding 36 processing units. So keeping a few units for system use, 30 is the best choice for the number of reducers.

5.5 Runtime of FSM-H for Different Partition Sizes

An important requirement for achieving the optimal performance of a Hadoop cluster is to find the appropriate number of mappers. In such a cluster, the setup and scheduling

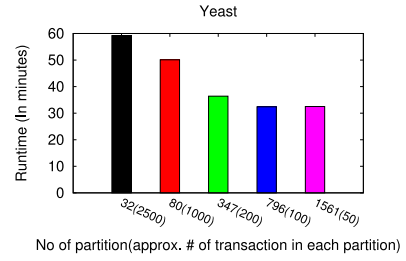


Fig. 21. Bar plot showing the relationship between the partition count and the running time of FSM-H for Yeast dataset.

of a task wastes a few seconds. If each of the tasks is small and the number of tasks is large, significant amount of time is wasted for task scheduling and setup. So the experts advise that a mapper should execute for at least 40 seconds.⁷ Another rule of thumbs is that the number of mappers should not over-run the number of partitions.

In our cluster setup, we have nine data nodes, each with four cores that allow us to have 36 mappers. Then, the perfect number of input partitions should be less than or equal to 36. But this rules does not fit well for frequent subgraph mining task. For example, the Yeast dataset has close to 80,000 graphs and if we make 30 partitions, then each partition ends up consisting of 2,666 graphs. Since, frequent subgraph mining is an exponential algorithm over its input size, performing a mining task over these many graphs generally becomes computationally heavy. In this case, the map function ends up taking more time than is expected. As a result the optimal number of partitions for an FSM task should set at a much higher value (compared to a tradition data analysis task) so that the runtime complexity of the mappers reduces significantly. Note that, the higher number of partitions also increases the number of key-value pairs for a given patterns which should be processed by the reducers. However, the performance gain from running FSM over small number of graphs supersedes the performance loss due to the increased number of key-value pairs. This is so, because the gain in execution time in the mappers follows an exponential function, whereas the loss in execution time in the reducers and the data transmission over the network follow a linear function. On the other hand, selecting the number of partitions in such a way so that, each mapper gets to process only one transaction will not be an optimal choice. The reason is, average processing time of only one transaction will be much less than average task scheduling and setup time. We will see in this experiment that increment of the number of partitions has negative effects on overall running time.

The following experiment validates the argument that we have made in the above paragraph. In this experiment, we run FSM-H on Yeast dataset for different number of partitions and compare their execution time. Fig. 21 shows the result using bar plot. The plot shows that as we increase the partition count, the performance keeps improving significantly until it levels off at around 1,000 partitions. When the partition count is 1,561, there is a slight loss in FSM-H's performance compared to the

7. <http://blog.cloudera.com/blog/2009/12/7-tips-for-improving-MapReduce-performance/>

TABLE 3
Runtime (in minutes) Comparison between FSM-H and Hill
et al.'s [38] on Three Biological Graph Datasets

Dataset	FSM-H (min)	Hill et al.'s [38] (min)
Yeast	16.6	109.2
P388	8.3	57.5
NCI-H23	17.5	117.2

scenario when the partition count is 796. The strategy of finding the optimal number of partitions depends on the characteristics of input graphs that control the complexity of the mining task, such as the density, number of edges, number of unique labels for vertices and edges in an input graph and so on.

5.6 Comparison of FSM-H with an Existing Algorithm

As discussed in Section 2, [38] proposed a MapReduce based graph mining method which is inefficient due to its naive mining approaches. In this experiment, we compare the execution time of FSM-H with that of Hill et al.'s [38] implementation that we obtain from the authors. We run both the algorithms on three real world biological dataset for 40 percent minimum support. Table 3 compares the execution time between the two algorithms. As we can see FSM-H performs significantly better than the other algorithm. The reason behind the better performance of FSM-H is that, [38]'s mining strategy is naive and it generates duplicate patterns. Like FSM-H, [38]'s algorithm mine patterns in map phase and count in reduce phase. Due to the duplicate patterns, the load over the network increases and subsequently, the reducers will have to wait for more time for the mappers to complete.

5.7 Effect of Partition Scheme on Runtime

In this experiment, we analyze how the partition heuristics discussed in Section 4.3.1 affects the execution time of FSM-H. For all datasets, we fixed minimum support to 20 percent, number of data nodes to 9 and the number of reducers to 30. The result is shown in Table 4. We find that for schema 2, which balances the total number of edges in each partition, performs somewhat better than schema 1, which only balances the number of graphs. Note that, in the real-life graph datasets most of the graphs have similar number of edges, so the partitions using schemas 1 and 2 have similar balance factors. But, for datasets where the input graphs have highly different number of vertices and edges, the schema 2 is obviously a better choice. To investigate further, we build a synthetic dataset of 50 *K* graphs where average length of half of the graphs is around 15 and for the other half it is around 30. We found that for this unbalanced dataset, scheme 1 takes about 30 percent more time than the schema 2 (see the last row of the Table 4).

6 CONCLUSIONS

In this paper we present a novel iterative MapReduce based frequent subgraph mining algorithm, called FSM-H. We show the performance of FSM-H over real life

TABLE 4
Runtime of FSM-H on Different Partition Schemes

Transaction Graph dataset	Scheme 1 (runtime in min)	Scheme 2 (runtime in min)
Yeast	32.5	30.3
SN12C	25.8	23.7
P388	11.7	10.2
NCI-H23	23.5	22
OVCAR-8	24.9	23.4
Synthetic	22.9	17.1

and large synthetic datasets for various system and input configurations. We also compare the execution time of FSM-H with an existing method, which shows that FSM-H is significantly better than the existing method.

ACKNOWLEDGMENTS

This research is supported by Mohammad Hasan's US NSF CAREER Award (IIS-1149851).

REFERENCES

- [1] A. Rosenthal, P. Mork, M. H. Li, J. Stanford, D. Koester, and P. Reynolds, "Cloud computing: A new business paradigm for biomedical information sharing," *J. Biomed. Inf.*, vol. 43, pp. 342–353, 2010.
- [2] W. Lam, L. Liu, S. Prasad, A. Rajaraman, Z. Vacheri, and A. Doan, "Muppet: Mapreduce-style processing of fast data," *Very Large Data Bases Endow.*, vol. 5, pp. 1814–1825, 2012.
- [3] G. Liu, M. Zhang, and F. Yan, "Large-scale social network analysis based on Mapreduce," in *Proc. Int. Conf. Comput. Aspects Soc. Netw.*, 2010, pp. 487–490.
- [4] J. Dean and S. Ghemawat, "Mapreduce: Simplified data processing on large clusters," *Commun. ACM*, vol. 51, pp. 107–113, 2008.
- [5] U. Kang, C. E. Tsourakakis, and C. Faloutsos, "Pegasus: A petascale graph mining system implementation and observations," in *Proc. 9th IEEE Int. Conf. Data Mining*, 2009, pp. 229–238.
- [6] U. Kang, B. Meeder, and C. Faloutsos, "Spectral analysis for billion-scale graphs: Discoveries and implementation," in *Proc. 15th Pacific-Asia Conf. Adv. Knowl. Discov. Data Mining*, 2011, pp. 13–25.
- [7] S. Suri and S. Vassilvitskii, "Counting triangles and the curse of the last reducer," in *Proc. 20th Int. Conf. World Wide Web*, 2011, pp. 607–614.
- [8] R. Pagh and C. E. Tsourakakis, "Colorful triangle counting and a mapreduce implementation," *Inf. Process. Lett.*, vol. 112, no. 7, pp. 277–281, 2012.
- [9] F. Afrati, D. Fotakis, and J. Ullman, "Enumerating subgraph instances using map-reduce," in *Proc. IEEE 29th Int. Conf. Data Eng.*, Apr. 2013, pp. 62–73.
- [10] B. Bahmani, R. Kumar, and S. Vassilvitskii, "Densest subgraph in streaming and mapreduce," *Proc. Very Large Data Bases Endow.*, vol. 5, no. 5, pp. 454–465, Jan. 2012.
- [11] J. Huan, W. Wang, D. Bandyopadhyay, J. Snoeyink, and J. Prins, "Mining protein family specific residue packing patterns from protein structure graphs," in *Proc. Int. Conf. Res. Comput. Mol. Biol.*, 2004, pp. 308–315.
- [12] S. Kramer, L. Raedt, and C. Helma, "Molecular feature mining in HIV data," in *Proc. Int. Conf. Knowl. Discov. Data Mining*, 2004, pp. 136–143.
- [13] B. Berendt, "Using and learning semantics in frequent subgraph mining," in *Proc. Adv. Web Min. Web Usage Anal.*, 2006, pp. 18–38.
- [14] R. Agrawal and R. Srikant, "Fast algorithms for mining association rules in large databases," in *Proc. 20th Int. Conf. Very Large Data Bases*, 1994, pp. 487–499.
- [15] A. Inokuchi, T. Washio, and H. Motoda, "An apriori-based algorithm for mining frequent substructures from graph data," in *Proc. 4th Eur. Conf. Principles Data Mining Knowl. Discov.*, 2000, pp. 13–23.

- [16] M. Kuramochi and G. Karypis, "Frequent subgraph discovery," in *Proc. Int. Conf. Data Mining*, 2001, pp. 313–320.
- [17] X. Yan and J. Han, "gspan: Graph-based substructure pattern mining," in *Proc. Int. Conf. Data Min.*, 2002, pp. 721–724.
- [18] S. Nijssen, and J. Kok, "A quickstart in frequent structure mining can make a difference," in *Proc. 10th ACM SIGKDD Int. Conf. Knowl. Discov. Data Mining*, 2004, pp. 647–652.
- [19] V. Chaoji, M. Hasan, S. Salem, and M. Zaki, "An integrated, generic approach to pattern mining: Data mining template library," *Data Min. Knowl. Discov. J.*, vol. 17, no. 3, pp. 457–495, 2008.
- [20] S. Chakravarthy, R. Beera, and R. Balachandran, "Db-subdue: Database approach to graph mining," in *Proc. Adv. Knowl. Discov. Data Mining*, 2004, pp. 341–350.
- [21] S. Chakravarthy and S. Pradhan, "Db-FSG: An SQL-based approach for frequent subgraph mining," in *Proc. 19th Int. Conf. Database Expert Syst. Appl.*, 2008, pp. 684–692.
- [22] B. Srichandan and R. Sunderraman, "Oo-FSG: An object-oriented approach to mine frequent subgraphs," in *Proc. Australasian Data Mining Conf.*, 2011, pp. 221–228.
- [23] D. J. Cook, L. B. Holder, G. Galal, and R. Maglothin, "Approaches to parallel graph-based knowledge discovery," *J. Parallel Distrib. Comput.*, vol. 61, pp. 427–446, 2001.
- [24] C. Wang and S. Parthasarathy, "Parallel algorithms for mining frequent structural motifs in scientific data," in *Proc. 18th Annu. Int. Conf. Supercomput.*, 2004, pp. 31–40.
- [25] S. Parthasarathy and M. Coatney, "Efficient discovery of common substructures in macromolecules," in *Proc. IEEE Int. Conf. Data Mining*, 2002, pp. 362–369.
- [26] T. Meinl, M. Worlein, O. Urzova, I. Fischer, and M. Philippsen, "The parmol package for frequent subgraph mining," *Electron. Commun. EASST*, vol. 1, pp. 1–12, 2006.
- [27] C. Borgelt and M. Berthold, "Mining molecular fragments: finding relevant substructures of molecules," in *Proc. IEEE Int. Conf. Data Mining*, 2002, pp. 51–58.
- [28] J. Huan, W. Wang, and J. Prins, "Efficient mining of frequent subgraphs in the presence of isomorphism," in *Proc. 3rd IEEE Int. Conf. Data Mining*, 2003, pp. 549–552.
- [29] M. Philippsen, M. Worlein, A. Dreweke, and T. Werth. (2011). ParSemis—The parallel and sequential mining suite. [Online]. Available: <https://www2.cs.fau.de/EN/research/ParSeMis/index.html>
- [30] J. Wang, W. Hsu, M. L. Lee, and C. Sheng, "A partition-based approach to graph mining," in *Proc. 22nd Int. Conf. Data Eng.*, 2006, p. 74.
- [31] S. N. Nguyen, M. E. Orlowska, and X. Li, "Graph mining based on a data partitioning approach," in *Proc. 19th Australasian Database Conf.*, 2008, pp. 31–37.
- [32] G. Buehrer, S. Parthasarathy, and Y.-K. Chen, "Adaptive parallel graph mining for CMP architectures," in *Proc. 6th IEEE Int. Conf. Data Mining*, 2006, pp. 97–106.
- [33] G.-P. Chen, Y.-B. Yang, and Y. Zhang, "Mapreduce-based balanced mining for closed frequent itemset," in *Proc. IEEE 19th Int. Conf. Web Serv.*, 2012, pp. 652–653.
- [34] S.-Q. Wang, Y.-B. Yang, Y. Gao, G.-P. Chen, and Y. Zhang, "Mapreduce-based closed frequent itemset mining with efficient redundancy filtering," in *Proc. IEEE 12th Int. Conf. Data Mining Workshops*, 2012, pp. 49–53.
- [35] L. Zhou, Z. Zhong, J. Chang, J. Li, J. Huang, and S. Feng, "Balanced parallel FP-growth with Mapreduce," in *Proc. IEEE Youth Conf. Inf. Comput. Telecommun.*, 2010, pp. 243–246.
- [36] H. Li, Y. Wang, D. Zhang, M. Zhang, and E. Y. Chang, "Pfp: parallel FP-growth for query recommendation," in *Proc. ACM Conf. Recommender Syst.*, 2008, pp. 107–114.
- [37] B.-S. Jeong, H.-J. Choi, M. A. Hossain, M. M. Rashid, and M. R. Karim, "A MapReduce framework for mining maximal contiguous frequent patterns in large DNA sequence datasets," *IETE Tech. Rev.*, vol. 29, pp. 162–168, 2012.
- [38] S. Hill, B. Srichandan, and R. Sunderraman, "An iterative Mapreduce approach to frequent subgraph mining in biological datasets," in *Proc. ACM Conf. Bioinform., Comput. Biol. Biomed.*, 2012, pp. 661–666.
- [39] X. Xiao, W. Lin, and G. Ghinita, "Large-scale frequent subgraph mining in Mapreduce," in *Proc. Int. Conf. Data Eng.*, 2014, pp. 844–855.
- [40] M. Kuramochi and G. Karypis, "Finding frequent patterns in a large sparse graph*," *Data Mining Knowl. Discov.*, vol. 11, pp. 243–271, 2005.
- [41] S. Skiadopoulos, M. Elseidy, E. Abdelhamid, and P. Kalnis, "Grami: Frequent subgraph and pattern mining in a single large graph," *Proc. Very Large Database Endow.*, vol. 7, pp. 517–528, 2014.
- [42] J. Lin and C. Dyer, *Data-Intensive Text Processing with MapReduce*. San Rafael, CA, USA: Morgan and Claypool.
- [43] J. Cheng, Y. Ke, W. Ng, and A. Lu, "Fg-index: Towards verification-free query processing on graph databases," in *Proc. ACM SIGMOD Int. Conf. Manage. Data*, 2007, pp. 857–872.



Mansurul Alam Bhuiyan received the BSc (Eng.) degree in computer science and engineering from the Bangladesh University of Engineering and Technology in 2008. He is a third year PhD student in the Department of Computer and Information Science at Indiana University-Purdue University Indianapolis. Earlier, he was a lecturer of computer science at Stamford University, Dhaka, Bangladesh. Data mining is his core research interest, within data mining he works on problems related to pattern mining, and large network analysis. His research work is published in top-tier data mining conferences, including CIKM and ICDM.



Mohammad Al Hasan received the BSc degree in computer science and engineering from BUET, Dhaka, in 1998, the MS degree in computer science from the University of Minnesota, Twin Cities in 2002, and the PhD degree in computer science from Rensselaer Polytechnic Institute, NY, in 2009. He is an assistant professor of computer science at Indiana University-Purdue University, Indianapolis (IUPUI). Before that, he was a senior research scientist at eBay Research Labs, San Jose, CA. His research interest focuses on developing novel algorithms in data mining, data management, information retrieval, machine learning, social network analysis, and bioinformatics. He has published more than 30 research articles in top-tier data mining conferences and journals. He has received various awards, including PAKDD conference best paper award in 2009, SIGKDD doctoral dissertation award in 2010, US NSF CAREER award in 2012, and IUPUI School of Science Pre-tenure Research award in 2013.

► For more information on this or any other computing topic, please visit our Digital Library at www.computer.org/publications/dlib.