Available online at www.sciencedirect.com

ScienceDirect

Procedia Computer Science 35 (2014) 338 – 347

Procedia Computer Science

18th International Conference on Knowledge-Based and Intelligent Information & Engineering Systems - KES2014

Effectively and efficiently mining frequent patterns from dense graph streams on disk

Peter Braun[a], Juan J. Cameron[a], Alfredo Cuzzocrea[b,*], Fan Jiang[a], Carson K. Leung[a,*]

[a]*Department of Computer Science, University of Manitoba, Winnipeg, MB, R3T 2N2, Canada*
[b]*ICAR-CNR and University of Calabria, Via P. Bucci, 41C, I-87036, Rende (CS), Italy*

Abstract

In this paper, we focus on *dense graph streams*, which can be generated in various applications ranging from sensor networks to social networks, from bio-informatics to chemical informatics. We also investigate the problem of *effectively and efficiently mining frequent patterns* from such streaming data, in the targeted case of dealing with *limited memory environments* so that *disk support* is required. This setting occurs frequently (e.g., in mobile applications/systems) and is gaining momentum even in advanced computational settings where social networks are the main representative. Inspired by this problem, we propose (i) a specialized data structure called DSMatrix, which captures important data from dense graph streams onto the disk directly and (ii) stream mining algorithms that make use of such structure in order to mine frequent patterns effectively and efficiently. Experimental results clearly confirm the benefits of our approach.
© 2014 The Authors. Published by Elsevier B.V. This is an open access article under the CC BY-NC-ND license (http://creativecommons.org/licenses/by-nc-nd/3.0/).
Peer-review under responsibility of KES International.

Keywords: Data mining; frequent pattern mining; graph streams; knowledge-based and intelligent information & engineering systems; knowledge discovery; limited memory; stream mining

1. Introduction

Over the past two decades, numerous studies[19,21,26] have been proposed for the research problem of frequent pattern mining from traditional static databases[3]. Examples include (i) the FP-growth algorithm[17] that uses an in-memory structure called *Frequent Pattern tree* (*FP-tree*) to capture the content of the transaction database, as well as (ii) algorithms that use disk-based structures for mining[5,16].

The automation of measurements and data collection has produced high volumes of valuable data at high velocity in many application areas. The increasing development and use of a large number of sensors has added to this situation. These advances in technology have led to streams of data in road, sensor and social networks[11,20]. These kinds of data share in common the property of being modeled in terms of graph-structured data so that *graph streams*[1,2,10] are generated. In order to be able to make sense of streaming data, stream mining algorithms are needed[9,24]. When

* Corresponding authors.
E-mail address: cuzzocrea@si.deis.unical.it (A. Cuzzocrea), kleung@cs.umanitoba.ca (C.K. Leung)

1877-0509 © 2014 The Authors. Published by Elsevier B.V. This is an open access article under the CC BY-NC-ND license (http://creativecommons.org/licenses/by-nc-nd/3.0/).
Peer-review under responsibility of KES International.
doi:10.1016/j.procs.2014.08.114

comparing with the mining from traditional *static* databases, mining from *dynamic* data streams is more challenging due to the following *properties of data streams*. First, *data streams are continuous and unbounded*. To find frequent patterns from streams, we no longer have the luxury of performing multiple scans of the streams. Once the streams flow through, we lose them. Hence, we need some data structures to capture the important contents of the streams (e.g., recent data—because users are usually more interested in recent data than older ones [12,13]). Second, *data in the streams are not necessarily uniformly distributed; their distributions are usually changing with time*. A currently infrequent pattern may become frequent in the future, and vice versa. So, we have to be careful not to prune infrequent patterns too early; otherwise, we may not be able to get complete information such as frequencies of certain patterns (as it is impossible to retract those pruned patterns).

To mine frequent patterns from data streams, several approximate and exact algorithms have been proposed. For example, FP-streaming [15] and TUF-streaming [22] are *approximate* algorithms, which focus mostly on efficiency. However, due to approximate procedures, these algorithms may find some infrequent patterns or miss frequency information of some frequent patterns (i.e., some false positives or negatives). To mine only truly frequent patterns (i.e., no false positives and no false negatives), an *exact* algorithm (i) constructs a *Data Stream Tree* (*DSTree*) [25] to capture contents of the streaming data and then (ii) recursively builds FP-trees for projected databases based on the information extracted from the DSTree.

While the two aforementioned properties of streams play an important role in mining data streams, they play a more challenging role in the mining of a special class of data streams—namely, *graph streams*. Nowadays, various graph data sources can easily generate high volumes of streams of graphs (e.g., direct acyclic graphs representing human interactions in meetings [14], social networks representing connections or friendships among social individuals [8,29], semantic graphs linking web documents [28]). Problems and state-of-the-art solutions are highlighted in recent studies. For instance, Aggarwal et al. [2] studied the research problem of mining dense patterns in graph streams, and they proposed probabilistic algorithms for determining such structural patterns effectively and efficiently. Bifet et al. [4] mined frequent closed graphs on evolving data streams. Their three algorithms work on coresets of closed subgraphs, compressed representations of graph sets, and maintenance of these sets in a batch-incremental manner. Moreover, Aggarwal [1] explored a relevant problem of *classification* of graph streams. Along this direction, Chi et al. [10] proposed a fast graph stream classification algorithm that uses discriminative clique hashing, which can be applicable for OLAP analysis over evolving complex networks. Furthermore, Valari et al. [30] discovered top-k dense subgraphs in dynamic graph collections by means of both exact and approximate algorithms. As a preview, while these studies focus on graph mining, our mining algorithms in the current paper work on both graph-structured data and other non-graph data.

Although memory is not too expensive nowadays, the volume of data generated in data streams (including graph streams) also keeps growing at a rapid rate. Hence, algorithms for mining frequent patterns with limited memory are still in demand, so as to deal with the case of streams generated by graph data sources. For instance, Cameron et al. [7] studied this topic and proposed an algorithm that works well for *sparse* data streams in limited memory space. In contrast, the mining algorithms we propose in the current paper are designed to mine *dense* graph streams in limited memory space. Our algorithms can be viewed as complements to the sparse stream mining algorithm.

Preliminary results of our recent study [6] show the feasibility of dense-graph stream mining. Here, we detail how to effectively and efficiently mine frequent patterns from dense graph streams on disk. Our key contributions include (i) a simple yet powerful on-disk data structure called *Data Stream Matrix* (*DSMatrix*), which effectively captures and maintains relevant data found in the data streams (especially, dense graph streams) in a "matrix" form; (ii) its corresponding tree-based *mining algorithms*, which build the DSMatrix and efficiently mine frequent patterns from streaming data with sliding window models; and (iii) a *frequency counting technique*, which effectively avoids the recursive building of FP-trees for projected databases and efficiently saves space. As the proposed DSMatrix can generally be applicable to different kinds of streaming data, it can be used in graph streams where memory requirements are very demanding.

This paper is organized as follows. Background is provided in Section 2. Section 3 presents our DSMatrix structure for summarizing dense graph streams. Then, Section 4 explains how we make use of the DSMatrix for tree-based frequent pattern mining from dense graph streams. Section 5 focuses on an analytical evaluation on the complexity of the DSMatrix structure in comparison with other similar structures for stream mining of frequent patterns. Section 6 shows experimental results. Finally, conclusions and future work are given in Section 7.

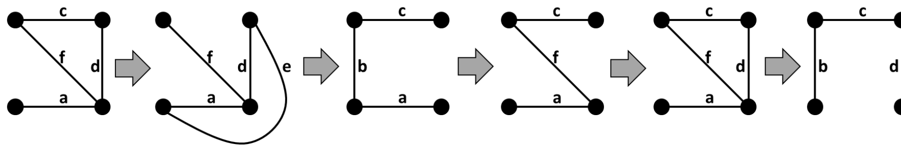


Fig. 1. A sample graph stream.

2. Background on frequent pattern mining from graph streams

In this section, we provide background on frequent pattern mining from graph streams (see Example 1), with a focus on (i) ⟨global *DSTree*, local FP-trees⟩ and (ii) ⟨global *DSTable*, local FP-trees⟩ mining options.

Example 1. Fig. 1 shows a stream of six graphs, where each graph $G_i = (V_i, E_i)$ consists of $|V_i| = 4$ vertices and $|E_i| \leq 6$ edges, where each edge is denoted by a symbol a, b, c, d, e or f. These graphs may represent some interactions in meetings, changes of friendships among social individuals, or reactions among chemical components. For illustrative purposes, let us consider a sliding window of size w=2 batches (i.e., only two batches are kept): (i) edge sets $E_1 = \{a, c, d, f\}$, $E_2 = \{a, d, e, f\}$ and $E_3 = \{a, b, c\}$ are kept in the first batch B_1; and (ii) edge sets $E_4 = \{a, c, f\}, E_5 = \{a, c, d, f\}$ and $E_6 = \{b, c, d\}$ are kept in the second batch B_2. □

2.1. The data stream tree (*DSTree*) and the ⟨*global DSTree, local FP-trees*⟩ mining option

An exact stream mining algorithm mines frequent patterns by first constructing a DSTree[25], which is then used as a *global tree* for recursively generating smaller FP-trees (as *local trees*) for projected databases. Due to the aforementioned properties of data streams, frequencies of "items" (i.e., *edges* in graph streams) are continuously affected by the insertion of new batches (and the removal of old batches) of data. Hence, when mining graph streams, the DSTree arranges edges according to some canonical order (e.g., alphabetical order), which can be specified by the user prior to the tree construction or mining process, rather than a frequency-dependent order (which may lead to swapping, merging, and/or splitting of tree nodes when frequencies change). As such, the DSTree can be constructed using only a *single* scan of the streams. Note that the DSTree is designed for processing streams within a sliding window. So, for a window size of w batches, each tree node keeps (i) an edge and (ii) a *list* of w frequency values (instead of a single frequency count in each node of the FP-tree for frequent pattern mining from *static* databases). Each entry in this list captures the frequency of the edges in each batch of dynamic streams in the current window. By so doing, when the window slides (i.e., when new batches are inserted and old batches are deleted), frequency information can be updated easily. Consequently, the resulting DSTree preserves the usual tree properties that (i) the total frequency (i.e., sum of w frequency values) of any node is at least as high as the sum of total frequencies of its children and (ii) the ordering of edges is unaffected by the continuous frequency changes.

Once the DSTree is constructed, it is always kept up-to-date when the window slides. The mining is delayed until it is needed. Specifically, the mining algorithm first traverses relevant paths of the DSTree upwards and sums the frequency values of each list in a node representing an edge (or a set of edges)—to obtain its frequency in the current sliding window—for forming an appropriate projected database. Afterwards, the algorithm constructs an FP-tree for the projected database of each of these frequent patterns of only 1 edge (i.e., 1-itemset) such as an $\{x\}$-projected database (in a similar fashion as in the FP-growth algorithm for mining static data[17]). Thereafter, the algorithm recursively forms subsequent FP-trees for projected databases of frequent k-itemsets where $k \geq 2$ (e.g., $\{x, y\}$-projected database, $\{x, z\}$-projected database, etc.) by traversing paths in these FP-trees. As a result, the algorithm finds all frequent patterns. Note that, as edges are consistently arranged according to some canonical order, the algorithm guarantees the inclusion of all *frequent* edges using just upward traversals. Moreover, there is also no worry about possible omission or double-counting of edges during the mining process. Furthermore, as the DSTree is always kept up-to-date, all frequent patterns—which are embedded in batches within the current sliding window—can be found effectively. In the remainder of this paper, we call this exact algorithm that uses the DSTree as the global tree—from which FP-trees for subsequent projected databases can be constructed recursively—the ⟨*global DSTree*, *local*

Table 1. DSTable and DSMatrix.

(a) DSTable

Row	Boundaries	Contents
Edge a:	Columns 3 & 5	$(c, 1), (d, 2), (b, 1); (c, 3), (c, 4)$
Edge b:	Columns 1 & 2	$(c, 2); (c, 5)$
Edge c:	Columns 2 & 5	$(d, 1)$, end; $(f, 3), (d, 3), (d, 4)$
Edge d:	Columns 2 & 4	$(f, 1), (e, 1); (f, 4)$, end
Edge e:	Columns 1 & 1	$(f, 2)$;
Edge f:	Columns 2 & 4	end, end; end, end

(b) DSMatrix

Row	Contents
Edge a:	1 1 1; 1 1 0
Edge b:	0 0 1; 0 0 1
Edge c:	1 0 1; 1 1 1
Edge d:	1 1 0; 0 1 1
Edge e:	0 1 0; 0 0 0
Edge f:	1 1 0; 1 1 0
Boundaries:	Columns 3 & 6

FP-trees⟩ mining option. This option works well when memory space is not an issue. The success of this algorithm mainly relies on the assumption—usually made for many tree-based algorithms [17]—that all tree (i.e., the global tree together with subsequent FP-trees) fit into the memory. For example, when mining frequent patterns from the $\{x, y, z\}$-projected database, the global tree and three subsequent FP-trees (for the $\{x\}$-, $\{x, y\}$- and $\{x, y, z\}$-projected databases) are all assumed to fit into memory.

2.2. *The data stream table (DSTable) and the ⟨global DSTable, local FP-trees⟩ mining option*

To handle situations where the memory is so limited that not all the trees can fit into memory (e.g., the case of streaming generated from graph data sources), the *Data Stream Table* (*DSTable*) [7] was proposed. The DSTable is a two-dimensional table that captures on the disk the contents of streaming data in all batches within the current sliding window. Each row of the DSTable represents an edge. Like the DSTree, edges in the DSTable are arranged according to some canonical order (e.g., alphabetical order), which can be specified by the user prior to the construction of the DSTable. As such, table construction requires only a single scan of the stream. Each entry in the resulting DSTable is a "pointer" that points to the location of the table entry (i.e., which row and which column) for the "next" edge in the same "transaction" (i.e., the same *edge set* in the graph stream). The DSTable also keeps w boundary values (to represent the boundary between w batches in the current sliding window) for each edge. By doing so, when the window slides, data in the old batch can be removed and data in the new batch can be added easily.

Like its counterpart with the DSTree, mining with this DSTable is also delayed until it is needed. Once the DSTable is constructed, it is kept up-to-date when the window slides. The mining algorithm first traverses relevant transactions from the DSTable to construct an FP-tree for the projected database of each of the 1-itemsets. Subsequent FP-trees for projected databases of frequent k-itemsets (where $k \geq 2$) are then recursively formed by traversing the paths of these FP-trees. As a result, the algorithm finds all frequent patterns. In the remainder of this paper, we call this the ⟨*global DSTree*, *local FP-trees*⟩ mining option.

Example 2. Continue with Example 1. Table 1(a) shows the information captured by a DSTable for the graph streams shown in Fig. 1. The first entry in Row a with value $(c, 1)$—which captures a set starting edge a and having c as the second edge—points to the 1st column of Row c. Its value $(d, 1)$ points to the 1st column of Row d, which captures the value $(f, 1)$. This indicates the third and fourth edges are d and f, respectively. Then, the 1st column of Row f with value "end" indicates the end of the set containing $\{a, c, d, f\}$. Similarly, start traversing from the second entry in Row a with value $(d, 2)$ to entries with values $(e, 1)$, $(f, 2)$ and "end" reveals the set containing $\{a, d, e, f\}$.

Let the user-specified *minsup* threshold be 2. Based on the contents of the entire DSTable, the mining algorithm first finds frequent singletons $\{a\}, \{b\}, \{c\}, \{d\}$ and $\{f\}$. The algorithm then constructs an FP-tree for the $\{a\}$-projected database (i.e., edge sets containing a) to get frequent 2-itemsets $\{a, c\}$, $\{a, d\}$ and $\{a, f\}$. From this FP-tree, the algorithm recursively constructs subsequent FP-trees (e.g., for $\{a, c\}$-, $\{a, c, d\}$- and $\{a, d\}$-projected databases). Afterwards, the algorithm constructs an FP-tree for the $\{b\}$-projected database, from which subsequent FP-trees are constructed. Similar steps apply to $\{c\}$- and $\{d\}$-projected databases.

The boundary information "Columns 3 & 5" for Row a indicates that (i) the boundary between batches B_1 and B_2 is at the end of Column 3 and (ii) batch B_2 ends at Column 5. Hence, when a new batch comes in, the old batch is removed. In this case, the first three columns of Row a (due to "Columns 3 & 5" in Row a), the first 1 column of

Row b (due to "Columns 1 & 2" in Row b), the first 2 columns of Rows c and d, the first 1 column of Row e, as well as the first 2 columns of Row f can be removed. □

Observation 1. As observed from the above example, mining with the ⟨global DSTable, local FP-trees⟩ option may suffer from several problems when handling data streams (especially, dense graph streams) with limited memory. Some of these problems are listed as follows:

- P1. To facilitate easy insertion and deletion of contents in the DSTable when the window (of size w batches) slides, the DSTable keeps w boundary values for each row (representing each of the m edges in the domain). Hence, the DSTable needs to keep a total of $m \times w$ boundary values (e.g., $6 \times 2 = 12$ boundary values in Example 2).
- P2. Each entry in the DSTable is a "pointer" that indicates the location in terms of row name (e.g., Row c) and column number (e.g., Column 1) of the table entry for the "next" item in the same edge set. When the data stream is sparse, only a few "pointers" need to be stored. However, when the graph stream is dense, many "pointers" need to be stored. Given a total of $|T|$ edge sets in all batches within the current sliding window, there are potentially $m \times |T|$ "pointers" (where m is the number of edges in the domain).
- P3. During the mining process, multiple FP-trees need to be constructed and kept in memory (e.g., FP-trees for all $\{a\}$-, $\{a, c\}$- and $\{a, c, d\}$-projected databases are required to be kept in memory). □

3. DSMatrix: a data structure for summarizing dense graph streams

To solve the above problems while mining frequent patterns from data streams (especially, dense graph streams) with limited memory, we propose a 2-dimensional structure called *Data Stream Matrix* (*DSMatrix*). This matrix structure captures the contents of edge sets in all batches within the current sliding window by storing them on the disk. Note that the DSMatrix is a binary matrix, which represents the presence of an item x in edge set E_i by a "1" in the matrix entry (E_i, x) and the absence of an item y from edge set E_j by a "0" in the matrix entry (E_j, y). With this binary representation of edges, each column in the DSMatrix captures an edge set. Each column in the DSMatrix can be considered as a bit vector.

Like the DSTable, our DSMatrix also keeps track of any boundary between two batches so that, when the window slides, edge sets in the older batches can be easily removed and edge sets in the newer batches can be easily added. Note that, in the DSTable, boundaries may vary from one row (representing an item) to another row (representing another item) due to the potentially different number of items present. Contrarily, in our DSMatrix, boundaries are the same from one row to another because we put a binary value (0 or 1) for each edge set.

Example 3. Let us revisit Example 2. Table 1(b) shows our DSMatrix, which captures the same information as the DSTable shown in Table 1(a) but with less space. □

Observation 2. Our DSMatrix can effectively and efficiently solve the first two problems P1–P2 associated with the DSTable (ref. Observation 1):

- S1. Recall that the DSTable needs to keep a total $m \times w$ boundary values. In contrast, our DSMatrix only keeps w boundary values (where $w \ll m \times w$) for the entire matrix, regardless how many domain items (m) are here.
- S2. Recall that each table entry in the DSTable captures both the row name and column number to represent a "pointer" to the next item in an edge set. The computation of column number requires the DSTable to constantly keep track of the index of the last item in each row representing a domain item. Moreover, each "pointer" requires two integer (row name/number and column number). For P items in $|T|$ edge sets, the DSTable requires $2 \times 32 \times P$ bits (for 32-bit integer representation). For dense data streams, the DSTable requires potentially $64m \times |T|$ bits. In contrast, our DSMatrix uses a bit vector to indicate the presence or absence of items in an edge set. The computation does not require us to keep track of the index of the last item in every row and thus incurs a lower computation cost. Moreover, given a total of $|T|$ edge sets in all batches within the current sliding window, there are $|T|$ columns in our DSMatrix. Each column requires only m bits. In other words, our DSMatrix takes $m \times |T|$ bits (cf. potentially $64m \times |T|$ bits for dense data streams required by the DSTree). □

4. Tree-based frequent pattern mining from dense graph streams

Once the DSMatrix is constructed, frequent patterns can be mined from it. Specifically, whenever a new batch of streaming data (e.g., streaming graph data) comes in, the window slides. Edge sets in the oldest batch in the sliding window are then removed from our DSMatrix so that edge sets in this new batch can be added. Following the aforementioned mining routines, (i) the mining is delayed until it is needed and (ii) the DSMatrix is kept up-to-date on the disk.

4.1. Mining with the ⟨global DSMatrix, recursive local FP-trees⟩ option

Our mining algorithm finds frequent patterns by first extracting relevant edge sets from the DSMatrix to form an FP-tree for each projected database of every frequent singleton. Key ideas of the algorithm are illustrated in Example 4.

Example 4. Let us continue with Example 3. To form the $\{a\}$-projected database, we examine Row a. For every column with a value "1", we extract its column downwards (e.g., from edges b to e if they exist). Specifically, when examining Row a, we notice that columns 1, 2, 3, 4 and 5 contain values "1" (which means that a appears in those five edge sets in the two batches of streaming graph data in the current sliding window). Then, from Column 1, we extract $\{c, d, f\}$. Similarly, we extract $\{d, e, f\}$ and $\{b, c\}$ from Columns 2 and 3. We also extract $\{c, f\}$ and $\{c, d, f\}$ from Columns 4 and 5. All these form the $\{a\}$-projected database, from which an FP-tree can be built. From this FP-tree for the $\{a\}$-projected database, we find that 2-itemsets $\{a, c\}$, $\{a, d\}$ and $\{a, f\}$ are frequent. Hence, we then form $\{a, d\}$- and $\{a, f\}$-projected databases, from which FP-trees can be built. (Note that we do not need to form the $\{a, c\}$-projected database as it is empty after forming both $\{a, d\}$- and $\{a, f\}$-projected databases.) When applying this step recursively in a depth-first manner, we obtain frequent 3-itemsets $\{a, c, d\}, \{a, c, f\}$ and $\{a, d, f\}$, which leads to FP-trees for the $\{a, d, c\}$-projected database. Again, we do not need to form the $\{a, f, c\}$- or $\{a, d, f\}$-projected databases as both of them are empty. At this moment, we keep FP-trees for the $\{a\}$-, $\{a, d\}$- and $\{a, d, c\}$-projected databases. Afterwards, we also find that 4-itemset $\{a, c, d, f\}$ is frequent. In the context of graph streams, this is a frequent collection of 4 edges—namely, Edges a, c, d and f. See Fig. 2.

We backtrack and examine the next frequent singleton $\{b\}$. When examining Row b, we notice that Columns 3 and 6 contain values "1" (which means that b appears in those two edge sets in the current sliding window). For these two columns, we extract downward to get $\{c\}$ and $\{c, d\}$ that appear together with b (i.e., to form the $\{b\}$-projected database). As shown in Fig. 2, the corresponding FP-tree contains $\{c\}$:2 meaning that c occurs twice with b (i.e., 2-itemset $\{b, c\}$ is frequent with frequency 2). Similar steps are applied to other frequent singletons $\{c\}, \{d\}$ and $\{f\}$ in order to discover all frequent patterns. □

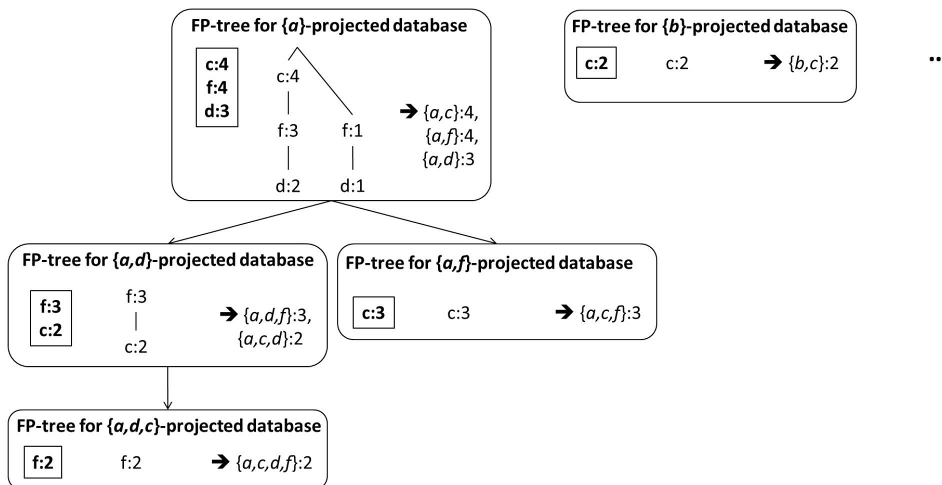


Fig. 2. Mining with the ⟨global DSMatrix, recursive local FP-trees⟩ option (Example 4).

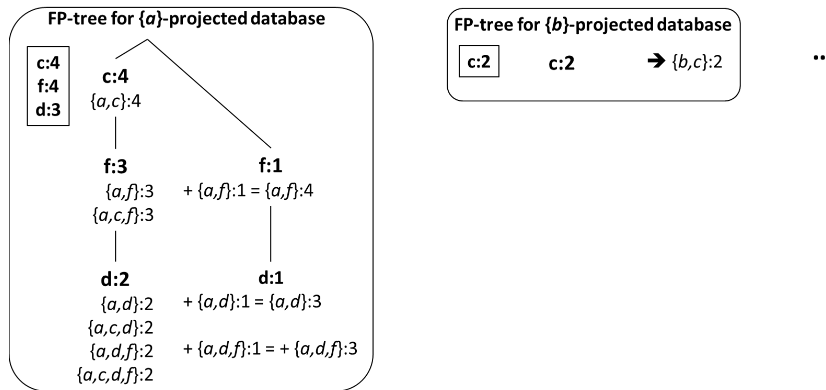


Fig. 3. Mining with the ⟨global DSMatrix, local FP-trees for only frequent singletons⟩ option (Example 5).

4.2. *Mining with the ⟨global DSMatrix, local FP-trees for only frequent singletons⟩ option*

The above mining process requires multiple FP-trees to be kept in memory during the mining process. However, when memory space is limited, *not* all of the multiple FP-trees can fit into memory. To solve this problem (as well as problem P3 in Observation 1), we propose the following effective frequency counting technique. Once an FP-tree for the projected database of a frequent singleton is built, we traverse every tree node in a depth-first manner (e.g., pre-order, in-order, or post-order traversal). For every first visit of a tree node, we generate the all itemsets represented by the node and its subsets. We also compute their frequencies.

Example 5. Based on the DSMatrix in Example 3, we first construct an FP-tree for the $\{a\}$-projected database. Then, we traverse every node in such an FP-tree. When traversing the left branch $\langle c{:}4, f{:}3, d{:}2\rangle$. By visiting nodes "$f{:}3$" and "$d{:}2$", we get $\{a, f\}$ and $\{a, c, f\}$ both with frequencies 3, as well as $\{a, d\}, \{a, c, d\}, \{a, d, f\}$ and $\{a, c, d, f\}$ all with frequencies 2. Then, we visit nodes "$f{:}1$" and "$d{:}1$" in the right branch $\langle f{:}1, d{:}1\rangle$, from which we get the frequency 1 for both $\{a, d\}, \{a, d, f\}$ and $\{a, f\}$. This frequency value is added to the existing frequency count of 2 (from the middle branch) to give the frequency of $\{a, d\}$ and $\{a, d, f\}$ equal to 3. Hence, with the *minsup* threshold set to 2, we obtain frequent patterns $\{a, c\}{:}4$, $\{a, c, d\}{:}2$, $\{a, c, d, f\}{:}2$, $\{a, c, f\}{:}3$, $\{a, d\}{:}3$, $\{a, d, f\}{:}3$ and $\{a, f\}{:}4$. Note that, during this mining process for the $\{a\}$-projected database, we count frequencies of itemsets without recursive construction of FP-trees. See Fig. 3.

Afterwards, we build an FP-tree for the $\{b\}$-projected database and count frequencies of all frequent patterns containing item b. Similar steps are applied to the FP-trees for the projected databases of *only* frequent singletons (i.e., FP-trees for $\{c\}$- and $\{d\}$-projected databases). □

Observation 3. Our DSMatrix with this effective frequency counting technique can solve the last problem P3 associated with the DSTable (ref. Observation 1):

S3. Recall that mining with DSTable requires recursive construction of FP-trees (e.g., not only the FP-tree for $\{a\}$-projected database but also FP-trees for $\{a, c\}$- and $\{a, c, d\}$-projected databases), which are all required to be kept in memory. In contrast, at any moment during the mining process, only one FP-tree needs to be constructed and kept in the memory for this ⟨*global DSMatrix*, *local FP-trees for only frequent singletons*⟩ mining process (cf. multiple FP-trees required for the ⟨global DSTable, recursive local FP-trees⟩ mining option). This solves problem P3. □

5. Complexity analysis

Recall from Sections 2.1 and 2.2 that mining with the DSTree[25] or DSTable[7] uses a delayed mode for mining. So, the actual mining of frequent patterns is delayed until they are needed to be returned to the user. Hence, for

S=1000 batches, the mining algorithm needs to build a global DSTree or DSTable and updates it $S-w = 1000-5 = 995$ times. Once an updated DSTree or DSTable has captured the 996th to the 1000th batches, multiple FP-trees are constructed to find frequent patterns. Note that only one set of the updated global DSTree (or DSTable) and multiple FP-trees are required. Moreover, at any time during the mining process of the ⟨*global DSTree*, *local FP-trees*⟩ option, only the global DSTree and multiple FP-trees are needed to be present. When using the ⟨*global DSTable*, *local FP-trees*⟩ option, the global DSTable is kept on disk. Thus, only multiple FP-trees are needed to be kept in the memory.

The DSMatrix, on the other hand, resides on disk. Being specialized to dense graph streams, it serves as an alternative to the global FP-tree when memory is limited. Moreover, the size of the DSMatrix is independent of the user-specified minimum support threshold *minsup*. Hence, it is useful for interactive mining, especially when users keep adjusting *minsup*, which is relevant for mining graph streams. It should be noted that the DSMatrix captures the edge sets in the current sliding window. During the mining process, the algorithm skips infrequent edges (i.e., edges having support lower than *minsup*) and only includes frequent edges when building subsequent FP-trees for projected databases.

Furthermore, when mining with our DSMatrix and our frequency counting technique, we do not even need to build too many FP-trees. Instead, we only need to build FP-trees for only frequent singletons (i.e., for $\{x\}$-projected databases, where x is a frequent edge).

Regarding disk space, the DSTable requires $64 \times P$ bits (for 32-bit integer representation), where P is the total number of items in $|T|$ edge sets in the w batches of the data streams. In the worst case, the DSTable requires potentially $64m \times |T|$ bits for dense data streams. In contrast, our DSMatrix requires only $m \times |T|$ bits, which is desirable for applications that require dense graph stream mining.

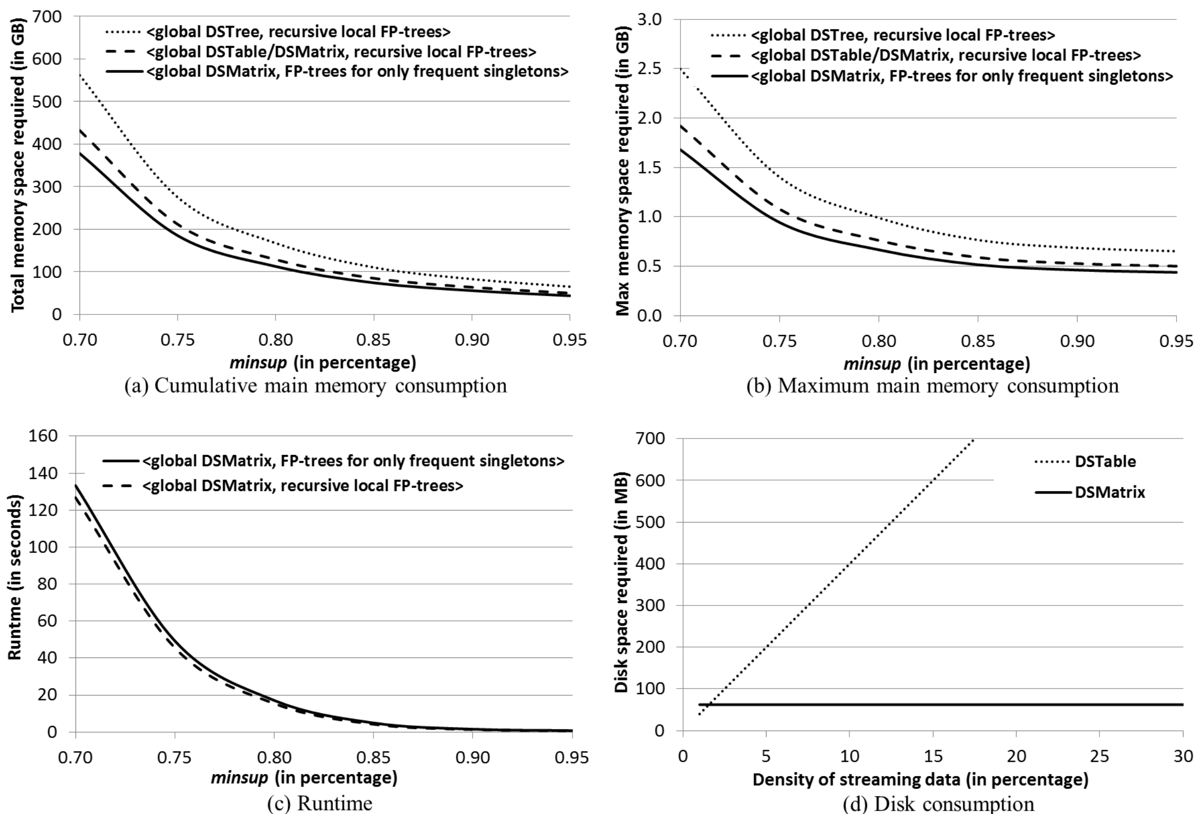
6. Experimental assessment and analysis

We first generated random graph models via a Java-based generator by varying model parameters (e.g., topology, average fan-out of nodes, edge centrality, etc.), then generated graph streams as nodes and node-edge relationships derived from the above graph models, and obtained node values from popular data stream sets available in literature (stored in the projected database). In addition, we also used many different databases including IBM synthetic data, real-life databases from the UC Irvine Machine Learning Depository, as well as those from the Frequent Itemset Mining Implementation (FIMI) Dataset Repository. For example, connect4 is a real-life dense data set containing 67,557 records with an average transaction length of 43 items, and a domain of 130 items. Each record represents a graph of legal 8-ply positions in the game of connect 4. All experiments were run in a time-sharing environment in a 1 GHz machine. We set each batch to be 6K records and the window size w=5 batches. The reported figures are based on the average of multiple runs. Runtime includes CPU and I/Os; it includes the time for both tree construction and frequent pattern mining steps. In the experiments, we mainly evaluated the accuracy and efficiency of our DSMatrix by comparing with related works such as (i) DSTree[25] and (ii) DSTable[7].

First, we measured the accuracy of the following four mining options: (i) ⟨global DSTree, (recursive) local FP-trees⟩; (ii) ⟨global DSTable, (recursive) local FP-trees⟩; (iii) ⟨global DSMatrix, recursive local FP-trees⟩; (iv) ⟨global DSMatrix, local FP-trees for only frequent singletons⟩ options. Experimental results showed that mining with any of these four options gave the same mining results.

Although these four options gave the same results, their performance varied. So, we then measured the space and time efficiency of our DSMatrix. Results show that the ⟨global DSTree, recursive local FP-trees⟩ option required the largest main memory space as it stores one global DSTree and multiple local FP-trees in main memory. Both ⟨global DSTable, recursive local FP-trees⟩ and ⟨global DSMatrix, recursive local FP-trees⟩ options required less memory as their DSTable and DSMatrix were kept on disk. Among the four mining options, the ⟨global DSMatrix, local FP-trees for only frequent singletons⟩ option required the *smallest* main memory space because at most m FP-trees needed to be generated during the entire mining process, one for each frequent domain edge. Note that not all m edges were frequent. Fig. 4(a) shows the cumulative main memory consumption. Note that, at any mining moment, only one of these FP-trees needs to be present. In other words, not all $\leq m$ FP-trees were generated at the same time. Fig. 4(b) shows the maximum main memory consumption.

Furthermore, we evaluated the effect of *minsup*. As expected, Fig. 4(c) shows that the runtime decreased when *minsup* increased.



(a) Cumulative main memory consumption (b) Maximum main memory consumption

(c) Runtime (d) Disk consumption

Fig. 4. Experimental results.

When evaluating scalability, Fig. 4(d) shows that our DSMatrix was scalable. The figure also compares the disk consumption between the DSTable and our DSMatrix, and it clearly shows that our DSMatrix requires a constant amount of disk space, where the DSTable requires different amounts depending on the density of data streams. An interesting observation that, for dense graph data, our DSMatrix is beneficial due to its bit vector representation.

As ongoing, we are conducting more extensive experiments on various datasets (including *Big data*) with different parameter settings (e.g., varying *minsup* and transaction lengths that represent the complexity of graphs).

7. Conclusions and future work

As technology advances, streams of data (including graph streams) are produced in many applications. Key contributions of this paper include (i) a simple yet powerful alternative disk-based structure—called *DSMatrix*—for efficient frequent pattern mining from streams (e.g., dense graph streams) with limited memory, (ii) tree-based frequent pattern mining algorithms, and (iii) an effective frequency counting technique, which avoids keeping too many FP-trees in memory when the space is limited. Such a technique requires only one FP-tree for a projected database to be kept in the limited memory. Analytical and experimental results show the benefits of our DSMatrix structure and its corresponding mining algorithms.

Future work is mainly oriented towards enhancing our proposed framework by means of novel advanced solutions in order to make it capable of dealing with the specific features of Big data[18,23,27].

Acknowledgements

This project is partially supported by the Natural Sciences and Engineering Research Council of Canada (NSERC) and the University of Manitoba.

References

1. Aggarwal CC. On classification of graph streams. In: *Proceedings of the SDM 2011*. SIAM; 2011, p. 652–663.
2. Aggarwal CC, Li Y, Yu PS, Jin R. On dense pattern mining in graph streams. *PVLDB* 2010; **3**(1–2):975–984.
3. Agrawal R, Srikant R. Fast algorithms for mining association rules in large databases. In: *Proceedings of the VLDB 1994*. Morgan Kaufmann; 1994, p. 487–499.
4. Bifet A, Holmes G, Pfahringer B, Gavaldà R. Mining frequent closed graphs on evolving data streams. In: *Proceedings of the ACM KDD 2011*. ACM; 2011, p. 591–599.
5. Buehrer G, Parthasarathy S, Ghoting A. Out-of-core frequent pattern mining on a commodity. In: *Proceedings of the ACM KDD 2006*. ACM; 2006, p. 86–95.
6. Cameron JJ, Cuzzocrea A, Jiang F, Leung CK. Frequent pattern mining from dense graph streams. In: *Proceedings of the EDBT/ICDT Workshops 2014*. CEUR-WS.org; 2014, p. 240–247.
7. Cameron JJ, Cuzzocrea A, Leung CK. Stream mining of frequent sets with limited memory. In: *Proceedings of the ACM SAC 2013*. ACM; 2013, p. 173–175.
8. Cameron JJ, Leung CK, Tanbeer SK. Finding strong groups of friends among friends in social networks. In: *Proceedings of the IEEE DASC* (*SCA*) *2011*. IEEE; 2011, p. 824–831.
9. Cao L, Yang D, Wang Q, Yu Y, Wang J, Rundensteiner EA. scalable distance-based outlier detection over high-volume data streams. In: *Proceedings of the IEEE ICDE 2014*. IEEE; 2014, p. 76–87.
10. Chi L, Li B, Zhu X. Fast graph stream classification using discriminative clique hashing. In: *Proceedings of the PAKDD 2013, Part I*. Springer; 2013, p. 225–236.
11. Cuzzocrea A. CAMS: OLAPing multidimensional data streams efficiently. In: *Proceedings of the DaWaK 2009*. Springer; 2009, p. 48–62.
12. Cuzzocrea A, Chakravarthy S. Event-based lossy compression for effective and efficient OLAP over data streams. *Data & Knowledge Engineering* 2010; **69**(7):678–708.
13. Cuzzocrea A, Furfaro F, Mazzeo GM, Saccà D. A grid framework for approximate aggregate query answering on summarized sensor network readings. In: *Proceedings of the OTM Workshops 2004*. Springer; 2004, p. 144–153.
14. Fariha A, Ahmed CF, Leung CK, Abdullah SM, Cao L. Mining frequent patterns from human interactions in meetings using directed acyclic graphs. In: *Proceedings of the PAKDD 2013, Part I*. Springer; 2013, p. 38–49.
15. Giannella C, Han J, Pei J, Yan X, Yu PS. Mining frequent patterns in data streams at multiple time granularities. In: Kargupta H, Joshi A, Sivakumar K, Yesha Y, editors. *Data mining: next generation challenges and future directions*. The MIT Press; 2004, chap. 6.
16. Grahne G, Zhu J. Mining frequent itemsets from secondary memory. In: *Proceedings of the IEEE ICDM 2004*. IEEE; 2004, p. 91–98.
17. Han J, Pei J, Yin Y. Mining frequent patterns without candidate generation. In: *Proceedings of the ACM SIGMOD 2000*. ACM; 2000, p. 1–12.
18. Leung CK. Big data mining and analytics. In: Wang J, editor. *Encyclopedia of business analytics and optimization*. IGI Global; 2014, p. 328–337.
19. Leung CK. Mining frequent itemsets from probabilistic datasets. In: *Proceedings of the EDB 2013*. KIISE Database Society; 2013, p. 137–148.
20. Leung CK, Carmichael CL. Exploring social networks: a frequent pattern visualization approach. In: *Proceedings of the IEEE SocialCom 2010*. IEEE; 2010, p. 419–424.
21. Leung CK, Carmichael CL, Johnstone P, Yuen DSHC. Interactive visual analytics of databases and frequent sets. *International Journal of Information Retrieval Research* 2013; **3**(4):120–140.
22. Leung CK, Cuzzocrea A, Jiang F. Discovering frequent patterns from uncertain data streams with time-fading and landmark models. *LNCS Transactions on Large-Scale Data- and Knowledge-Centered Systems* 2013; **8**:174–196.
23. Leung CK, Hayduk Y. Mining frequent patterns from uncertain data with MapReduce for Big data analytics. In: *Proceedings of the DASFAA 2013, Part I*. Springer; 2013, p. 440–455.
24. Leung CK, Jiang F. Frequent itemset mining of uncertain data streams using the damped window model. In: *Proceedings of the ACM SAC 2011*. ACM; 2011, p. 950–955.
25. Leung CK, Khan QI. DSTree: a tree structure for the mining of frequent sets from data streams. In: *Proceedings of the IEEE ICDM 2006*. IEEE; 2006, p. 928–932.
26. Leung CK, Tanbeer SK. PUF-tree: a compact tree structure for frequent pattern mining of uncertain data. In: *Proceedings of the PAKDD 2013, Part I*. Springer; 2013, p. 13–25.
27. Moens S, Aksehirli E, Goethals B. Frequent itemset mining for Big data. In: *Proceedings of the IEEE BigData Conference 2013*. IEEE; 2013, p. 111–118.
28. Nunes BP, Kawase R, Fetahu B, Dietze S, Casanova MA, Maynard D. Interlinking documents based on semantic graphs. In: *Proceedings of the KES 2013*. Elsevier; 2013, p. 231–240.
29. Tanbeer SK, Jiang F, Leung CK, MacKinnon RK, Medina IJM. Finding groups of friends who are significant across multiple domains in social networks. In: *Proceedings of the CASoN 2013*. IEEE; 2013, p. 21–26.
30. Valari E, Kontaki M, Papadopoulos AN. Discovery of top-k dense subgraphs in dynamic graph collections. In: *Proceedings of the SSDBM 2012*. Springer; 2012, p. 213–230.