sensors

MDPI

Article

Detecting Incremental Frequent Subgraph Patterns in IoT Environments

Kyoungsoo Bok, Jaeyun Jeong, Dojin Choi and Jaesoo Yoo *

Department of Information and Communication Engineering, Chungbuk National University, Chungdae-ro 1, Seowon-Gu, Cheongju, Chungbuk 28644, Korea; ksbok@chungbuk.ac.kr (K.B.); jjjeong@chungbuk.ac.kr (J.J.); mycdj91@gmail.com (D.C.)

* Correspondence: yjs@cbnu.ac.kr; Tel.: +82-43-261-3230

Received: 7 October 2018; Accepted: 15 November 2018; Published: 18 November 2018

Abstract: As graph stream data are continuously generated in Internet of Things (IoT) environments, many studies on the detection and analysis of changes in graphs have been conducted. In this paper, we propose a method that incrementally detects frequent subgraph patterns by using frequent subgraph pattern information generated in previous sliding window. To reduce the computation cost for subgraph patterns that occur consecutively in a graph stream, the proposed method determines whether subgraph patterns occur within a sliding window. In addition, subgraph patterns that are more meaningful can be detected by recognizing only the patterns that are connected to each other via edges as one pattern. In order to prove the superiority of the proposed method, various performance evaluations were conducted.

Keywords: graph stream; IoT; subgraph pattern; frequent pattern detection; incremental

---

1. Introduction

A graph is a data structure consisting of vertices and edges connecting the vertices. These graphs have been used to represent many-to-many relationships between objects, where a vertex represents an object, and an edge represents a relationship between objects [1–4]. Graphs are widely used in various fields, such as road networks, bioinformatics, and social networks [5–7]. For example, in a traffic network, regions are represented by vertices, and roads are expressed by edges. In a social network, vertices represent users, and edges express the relationships of followers and friends. In bioinformatics, graphs are used to model interactions between biomolecules, and graphical frequent pattern detection is used for protein function prediction, mutant gene discrimination, disease type identification, and so on [8–10]. Now, graph data change in real time due to the activation of Internet of Things (IoT) along with the advances in network technologies. Stream data in which vertices and edges that make up a graph continuously change are called graph streams. Graph streams have been used in various fields for different applications, such as abnormal detection, real-time trend analysis, and event detection [11–16]. As the graph stream has been applied in various fields, a large number of studies on various techniques for the analysis of graph streams have been conducted [17–20].

As the vertices and edges are continually added, deleted, and updated in IoT environments, studies on the detection or analysis of changes in graphs have also been conducted. Various approaches, such as graph clustering, graph stream classification, subgraph mining, and frequent subgraph pattern detection, have been proposed for graph analysis [5,21–27]. Frequent subgraph pattern, which detects a subgraph frequently occurring during a specific period is a widely used analysis method for graph streams [28–32]. In the IoT environment, frequent subgraph pattern is used for analyzing interactions among various objects or for determining anomalies [33–35]. For example, in anomaly detection,

the process of data transmission between IoT devices is modeled as graph data, and then a frequent subgraph pattern is generated. If an infrequent subgraph pattern occurs, it is determined as an anomaly.

The frequent subgraph detection in static graphs identifies frequently occurring subgraphs in the whole graph. However, graph streams continue to change vertices or edges over time. A graph matching finds a correspondence between the vertices and the edges of two graphs that satisfies some constraints. Since there are various subgraph patterns that can occur in the graph stream, all possible subgraphs should be compared when using graph matching. Therefore, it takes a lot of comparison time to detect frequent subgraphs by using graph matching in the graph stream. As the utilization of frequent subgraph pattern detection in recent graph streams has increased, various approaches have been actively conducted [29,36–38] in order to deal with this. In Reference [30], Data Stream Tree (DSTree) was proposed to store graph streams in memory efficiently during frequent subgraph pattern detection. Once the graph stream data are inputted, a DSTree is constructed, using whichever FP-tree was constructed to detect the frequent pattern. In Reference [31], Data Stream Matrix (DSMatrix) was proposed for storing graphs more efficiently than the DSTree proposed in Reference [30]. The DSMatrix is a two-dimensional array, so it can be constructed at a lower cost than that of DSTree. In addition, the frequency level of each pattern is calculated with a depth-first search after constructing a new FP-tree. In Reference [32], a frequent detection that considers connectivity was proposed. In this method, a simple frequent subgraph pattern detection using the AND operation was used, and only connected patterns are detected by managing adjacent edge information. However, the existing methods have several problems. First, Reference [30] uses many pointers because it constructs a DSTree, which is time-consuming. In addition, References [30,31] detect insignificant frequent patterns since these methods do not consider connectivity. Furthermore, References [30–32] employs a sliding window to detect all frequent patterns. However, since they do not process duplicate operations, their operation time is degraded.

Real-time processing is required to detect frequent subgraphs in the graph stream. Real-time processing of graph streams needs to reduce processing time. Since real-time processing techniques use memory, it is necessary to minimize memory usage. In this paper, we propose a new method that can detect a frequent subgraph pattern incrementally in graph stream data inputted in real time. The proposed method models the relationship between IoT devices in the IoT environment as a graph and detects frequently occurring subgraphs above the threshold value as vertices and edges constantly change over time. Frequent subgraphs that are detected through the proposed method are used to determine anomalies in the IoT environment or to analyze interactions among IoT devices. The proposed method is aimed at reducing processing time and memory usage to process graph streams in real time. The proposed method constructs DSMatrix for graph streams and does not construct FP-tree to reduce the memory usage. The proposed method determines whether re-calculation is needed after calculating whether a subgraph pattern detected from the previous sliding window will be frequent or not in the future. By doing so, only necessary calculations are performed, thereby reducing the total computation. In addition, more meaningful patterns can be detected by recognizing the patterns that are connected through edges between the patterns as one pattern. In addition, more meaningful patterns can be detected by recognizing patterns that are connected to each other through their edges as one pattern.

This paper is organized as follows. Section 2 analyzes the existing methods that detect a frequent subgraph pattern in graphs and presents the limitations of the previous studies. Section 3 explains the proposed frequent subgraph pattern detection method, and Section 4 presents performance evaluation results that verify the superiority of the proposed method. Finally, Section 5 describes the conclusions of this study and future research.

2. Related Works

In Reference [30], a frequent subgraph detection was proposed in which graph streams are rapidly stored in memory using a tree structure called DSTree. Once the graph streams are inputted, each edge

is arranged in order, and then a tree is constructed. Since each of the nodes in a DSTree maintains edge information by batch, it can be maintained by just changing the relevant information, even if a window slide has moved. Each node in a DSTree stores the names of the graph edges and the count of the appearances for each batch, each of which is divided by a semicolon. If there is a scarce graph, a tree is constructed only with regard to the edges newly created. Thus, efficient graph storage can be achieved.

In Reference [31], a DSMatrix, which can store graph streams more efficiently than the previous approaches were proposed. A DSMatrix is a 2D structure that represents whether an edge in the graph is generated by one or zero to store graph stream data in a small memory usage. To detect a frequent pattern, two approaches are used: Recursive-FP-tree and FP-trees for only frequent singletons. Using the recursive-FP-tree method, graph streams are stored in the DSMatrix and then an FP-tree is constructed for all edges. All frequent patterns are detectable by recursively constructing the FP-tree. This method can easily extract preferred subgraph by constructing multiple FP-trees. In the FP-trees for only frequent singletons, an FP-tree is first constructed with regard to all edges, and then each node is visited using the depth-first search method to calculate the number of occurrences of frequent patterns. This method can detect frequent patterns at a lower cost than that of the recursive-FP-tree since FP-trees are constructed for only a singleton edge.

In Reference [32], a frequent subgraph pattern using a simple AND operation without FP-tree was proposed. This method detects meaningful frequent patterns considering the connectivity between edges. It uses two approaches: one excludes unconnected patterns after generating candidates of frequent patterns using the AND operation for frequent subgraph pattern detection, and the other verifies whether the connection is made prior to starting the AND operation, and then it performs the operation.

A DSTree can store data in memory efficiently during frequent subgraph pattern detection [30]. Once graph stream data are inputted, a DSTree is constructed using whichever FP-tree was constructed to detect the frequent pattern. A DSMatrix can store graphs efficiently [31]. The DSMatrix is a two-dimensional array, so it can be constructed at a lower cost than that of the DSTree. In addition, a frequency level of each pattern is calculated with a Depth-First Search (DFS) after constructing a new FP-tree. In Reference [32], a frequent detection method that considers connectivity was proposed. In this method, a simple frequent subgraph pattern detection using the AND operation is used, and only connected patterns are detected by managing adjacent edge information in a table. However, the existing methods have some limitations as Table 1. Reference [30] has a shortcoming when a graph's structure changes significantly because the structure of the DSTree also changes, which takes a lot of time to reconstruct. In addition, when a DSTree is constructed, a large number of pointers are used when graph data are dense, entailing a large management cost to maintain the tree. To solve this problem effectively, a DSMatrix was proposed [31]. However, it requires many operations and a lot of memory because a large number of FP-trees are constructed during frequent subgraph pattern detection. It requires a lot of computation time because it travels all over the FP-trees to generate frequent patterns. Reference [32] uses AND operation to detect frequent patterns. However, it has a problem with performing comparative operations on the edges that are not likely to be detected in the future. In addition, References [31,32] do not solve the duplicate calculation problem, which is one of the drawbacks when using a sliding window, thereby degrading the performance.

In this paper, we propose an incremental frequent pattern detection to solve the problems of existing methods. The proposed method reduces the amount of computation by reusing the results of analysis from the previous sliding window as the window slide moves. It stores the input graph streams in DSMatrix to reduce the cost of building a DSTree according to the changes of the graph stream. It reduces memory usage because it only manages previously detected frequent pattern information. The detected patterns are calculated and managed separately for the next few sliding windows. This calculated value reduces the overall computation because the next sliding window only performs AND operations on subgraph patterns that are likely to occur in a frequent pattern. It

also reduces unnecessary comparison operations because only the connections between patterns are determined in one pattern.

Table 1. Characteristics and limitation of the existing methods.

Methods	Characteristics	Limitations
[30]	• Use of less space because building a tree with only the main lines from the sparse graph. • fast access because it is constructed by using trees.	• Increased DSTree construction time when the structure of the graph changes significantly. • A large number of comparison operations occur because it detects frequent patterns after the full scan of DSTree.
[31]	• Deployed at less cost than DSTree since DSMatrix is a two-dimensional array. • Construct FP-Trees to detect frequent patterns and calculate the frequency of each pattern using the DFS.	• Require a lot of computation and memory while constructing many FP-Trees for frequent pattern detection. • DFS takes a long time to detect frequent patterns. • Fail to resolve duplicate calculations, a problem caused by the use of sliding window techniques.
[32]	• Fast and frequent patterns detection using simple AND computations. • Significant frequent pattern detection based on connectivity.	• Significant performance degradation because AND operations are performed on all patterns. • Fail to resolve duplicate calculations, a problem caused by the use of sliding window techniques.

3. The Proposed Frequent Subgraph Pattern Detection

3.1. Preliminary

A graph is a data structure to express a multiple-relationship among objects. The graph consists of the vertex representing the object and the edge representing the relationship among the objects. Definition 1 represents the definition for the graph. If data transmission between IoT devices is represented as a graph in the IoT environment, a vertex represents IoT device and an edge represents data transmission status between IoT devices. Similarly, when a human network on a social network is represented as a graph, the vertex is the user and the edge is the friendship. Graphs are divided into directed and undirected graphs. The undirected graph does not take into account the direction of the edge between the vertices, but the directed graph displays the arrow lines along the connected direction between the vertices. When the vertices or edges that make up the graph change continuously over time, they are called graph streams. Graph streams are graphs that vertices and edges change dynamically over time. The graph streams occur frequently in IoT environments where the relationships of objects change. Definition 2 defines the graph stream G_t that occurs in time t.

Definition 1. *Graph G*

Given a set of vertices V and a set of edges E which are subsets of $V \times V$, a graph is defined as an order pair $G = (V, E)$.

Definition 2. *Graph stream G_t*

Given a set of vertices V_t at time t and a set of edges E_t which are subsets of $V_t \times V_t$ at time t, stream graph is defined as order pair $G_t = (V_t, E_t)$.

In a static graph with no change, a frequent subgraph represents a subgraph that appears above the threshold within the given graph. Since vertices and edges change continuously in the graph stream over time, a frequent subgraph is defined as a subgraph with more than a threshold within a continuous time interval, as defined in Definition 3.

Definition 3. *Frequent subgraph in graph stream*

Given a stream graph, frequent subraph is defined as $Freq(SG) \geq \theta$, where SG is a subgraph, $Freq(SG)$ is the occurrence percentage of graph that contain SG.

We incrementally detect a frequent subgraph in the graph stream. It constructs the DSMatrix proposed in Reference [31] to determine whether or not each edge occurs in the input graph stream.

With a two-dimensional array called DSMatrix, it is possible to store a large amount of edge information in a small space. The proposed method generates subgraphs if neighboring edges occur frequently and manage their occurrence information in the frequent subgraph management table by using DSMatrix. It determines incrementally whether a frequent subgraph has occurred by using FiB, FiS, and slideNum in the frequent subgraph management table. Table 2 shows the notations used in the proposed method.

Table 2. Notations.

Notation	Description
DSMatrix	Matrix representing the occurrence of the edge as 1 or 0
$< v_i, v_j >$	edge connecting the vertices v_i and v_j
FiB	Number of occurrences of the edge in one batch processing
FiS	Total number of edges in the sliding window
slideNum	Value to calculate whether Edge will be frequent or not during future window slide
AND operation	Operations that determine whether two subgraphs occur simultaneously

3.2. Overall Procedure

As graph streams have been applied to IoT, including anomaly detection, real-time trend analysis, and event detection, a large number of studies have been performed on various methods that analyze graph streams [39,40]. There are three considerations when a frequent subgraph pattern is detected in graph stream data. First, frequent subgraph pattern detection should be fast and should utilize limited storage space efficiently. Since stream data are supplied constantly and no end point is specified, the graph data to be analyzed change in real time. Thus, when graph are inputted and analyzed, graph should be deleted, to some extent, to ensure sufficient memory for analyzing the next input graph. Second, input graph differ as time passes. That is, the currently frequent patterns may not continue for the next input graph, and vice versa. Finally, patterns should be generated considering the connectivity of graphs. Here, connectivity means that detected patterns are connected to one another.

In this paper, we propose a frequent subgraph pattern detection incrementally for graph stream data in IoT. The proposed method determines a frequent pattern for performing anomaly detection or for analyzing the cooperative relationship between IoT devices. For example, anomaly detection is determined as an anomaly when a subgraph that is not detected as a frequent pattern occurs, or cooperative relationship analysis determines that mutual cooperation is very high in the event of a frequent pattern among IoT devices. A frequent subgraph pattern is a connected subgraph that occurs above a threshold for a particular time interval. The proposed method reuses the analyzed results from the previous window when the window is moved to reduce the amount of computation. It stores the input graph streams in a DSMatrix; then, frequent patterns can be detected through the simple AND operation. Here, the detected patterns are calculated for determining whether they are frequent or infrequent in the several sliding windows in the future, and then managed in a table separately. Through the calculated values, only necessary calculations are performed in the next sliding window to reduce the overall computation.

The overall processing procedure of the proposed method is shown in Figure 1. The preprocessing efficiently stores the input graph in the memory and generates a DSMatrix. Here, a DSMatrix can store a large amount of graph data in a small space using a 2D structure. During frequent subgraph pattern detection, an operation to generate the actual frequent patterns is performed. When graph patterns are detected, graph pattern occurrences are summed to check whether frequent patterns occur, and the occurrences of two patterns are summed again via the AND operation, thereby enabling frequent subgraph pattern detection consisting of multiple edges. Here, the information on the frequent patterns generated from the previous window slide is employed through the use of a frequent subgraph pattern management table. In the frequent subgraph pattern management table, whether the previously detected patterns will be frequent or infrequent in the next several sliding windows is calculated and stored; through this value, only necessary calculations are performed in the next sliding window, thereby reducing the overall computation. Finally, the detected frequent patterns are delivered to the user and are simultaneously stored in the frequent subgraph pattern management table to be utilized in the next window slide.

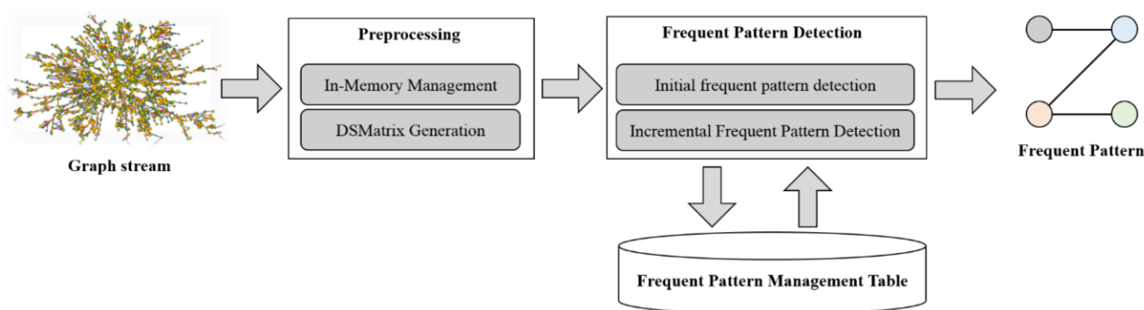

Figure 1. Overall processing procedure.

3.3. Preprocessing

Preprocessing stores graph streams in the memory and configures the DSMatrix. Since graph streams are continuously generated, graphs should be analyzed and deleted to ensure sufficient memory for processing the next input graph. In addition, all patterns can be generated for frequent patterns, considering not only currently generated graph but also previous and future graph. We use a 2D array structure called a DSMatrix in the preprocessing considering this characteristic. Since DSMatrix is a Boolean-type array, it can store more data in a smaller space than the DSTree uses in the existing method [30]. In addition, since the presence of an edge in a graph is expressed by one or zero, data can be added and deleted rapidly, which is suitable for the sliding window.

In a DSMatrix, a single window slide consists of the number of batches, which is set by the user. Table 3 shows the DSMatrix that results when the graph streams $G_1 \sim G_9$ are inputted, as shown in Figure 2. Here, one batch consists of three graphs, and three batches comprise a single sliding window. Each edge is represented by the names of the two connecting vertices. For example, the edge connected by vertices v_i and v_j is expressed as $< v_i, v_j >$. With the edges expressed in this way, the contents parts in Table 3 represent whether the edges occur or not as one or zero, respectively.

Table 3. DSMatrix.

Edge	Contents (Batch1)	Contents (Batch2)	Contents (Batch3)
$< v_1, v_2 >$	1 1 1	1 1 1	1 1 1
$< v_1, v_3 >$	1 1 1	0 1 1	1 1 1
$< v_1, v_4 >$	1 0 0	1 1 0	0 1 0
$< v_2, v_4 >$	1 0 1	0 0 0	1 0 0
$< v_3, v_4 >$	1 0 1	0 1 0	1 1 1

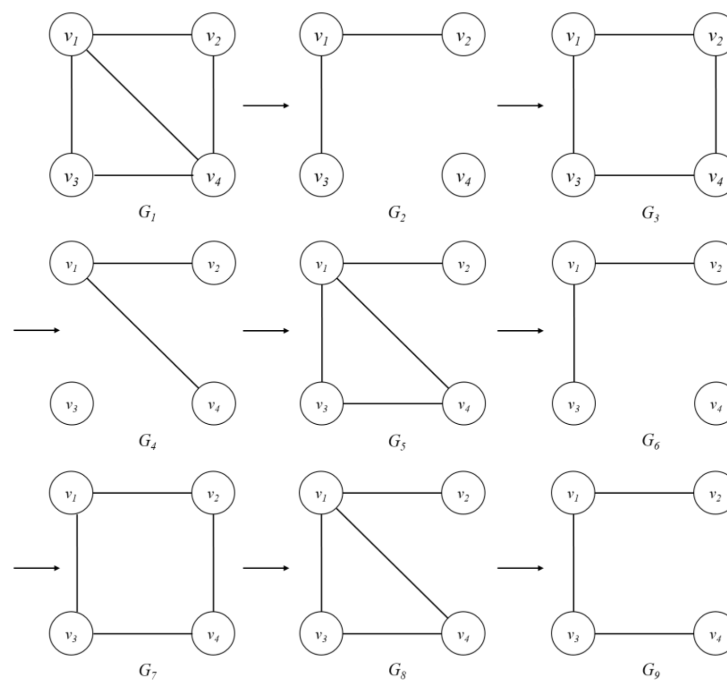

Figure 2. Example of a graph stream, where G_t is a graph at time t.

3.4. Initial Frequent Subgraph Pattern Detection

The initial frequent subgraph pattern detection generates a frequent subgraph pattern as a reference pattern during the incremental frequent subgraph pattern detection. We considers two features when a frequency pattern is detected. First, since stream data are continuously inputted, graphs should be analyzed and deleted, to some extent, when inputted to ensure sufficient memory for analyzing the next input graph. This requires a technique that detects a frequent subgraph pattern rapidly. The pattern occurrences are summed to check whether frequent patterns occur, and the occurrences of two patterns are summed again via the AND operation, thereby detecting frequent patterns consisting of multiple edges continuously. Here, since the operation that sums the number of occurrences and the AND operation are simple, patterns can be generated quickly. Second, we consider connectivity. If two patterns are far away from each other, it is difficult to see these two patterns as a single pattern, even if they occur simultaneously. Thus, our method detects more meaningful frequent patterns by considering connectivity.

A frequency level of a single edge is checked first to detect a frequent pattern. The number of occurrences in a single batch per edge in the DSMatrix, which is made in the preprocessing, is calculated and inputted to the Frequency in Batch (FiB) column in the frequent subgraph pattern management table, as presented in Table 4. FiB refers to the number of occurrences of the corresponding edge in a single batch. In addition, Frequency in Sliding window (FiS) refers to a value that sums all FiBs calculated previously, by which the determination of whether the current edge is frequent in the current sliding window can be verified. For example, the edge $< v_1, v_2 >$ occurred three times in every single batch and occurred a total of nine times in the current window slide.

Table 4. Frequent subgraph pattern management table.

Edge	slideNum	FiB (Batch1)	FiB (Batch2)	FiB (Batch3)	FiS
$< v_1, v_2 >$	1	3	3	3	9
$< v_1, v_3 >$	1	3	2	3	8
$< v_1, v_4 >$	0	1	2	1	4
$< v_2, v_4 >$	-1	2	0	1	3
$< v_3, v_4 >$	0	2	1	3	6

slideNum is a calculated value determining whether a single edge will be frequent or not during the next several slides. For example, even if zero is inputted to all $< v_1, v_2 >$ in the next window slide, the frequency count is still six, which is a frequent edge. However, if zero is input to all $< v_1, v_2 >$ again in the next window slide, the frequency count is three, which is not a frequent edge. Thus, slideNum becomes one. The algorithm to calculate slideNum is shown in Algorithm 1. slideNum is calculated as FiS and is divided into two cases, i.e., when it is larger or smaller than the threshold. If it is larger than the threshold, the remaining number, after removing the first batch from the current sliding window, is calculated as batchCount, and it is then determined whether batchCount exceeds the threshold, assuming that all new input batches are zero. This process is iterated until batchCount becomes smaller than the threshold. Then, the result is returned. If FiS is smaller than the threshold, whether batchCount exceeds the threshold is checked, assuming that the next input batches are all one, and the count is returned.

Algorithm 1. Algorithm to calculate slideNum.

```
Input:
FiB[] – array of FiB, which is the number of edges in a batch
FiS - The number of edges in sliding window
th - threshold
Output: slideNum
slideNum ← 0
if FiS >= th then
  batchCount ← FiS–FiB[0]
  while batchCount >= th and slideNum < slidingWindowSize do
    batchCount ← batchCount–FiB[slideNum+1]
    slideNum ← slideNum+1
  return slideNum
else
  batchCount ← FiS–FiB[0]+batchSize
  while batchCount < th and slideNum < slidingWindowSize do
    batchCount ← batchCount–FiB[SlideNum+1] + BatchSize
    slideNum ← slideNum + 1
  return -slideNum
```

After single edges, which are frequent, are all detected, patterns consisting of multiple edges are detected. If the AND operation is applied to the detected two patterns, it can identify the number of frequent occurrences of two patterns at the same time. A pattern is expanded continuously using this process. That is, if a pattern consisting of two edges is detected, a single frequent edge is employed. In the previous example, if the AND operation is applied to $< v_1, v_2 >$ and $< v_1, v_3 >$ among the single edge patterns, it produces 111;011;111 so that the number of occurrences becomes eight, which indicates that $< v_1, v_2, v_3 >$ is also a frequent pattern.

Additionally, when detecting patterns consisting of multiple edges, connectivity should be taken into consideration. An edge name is used to determine whether two patterns are connected. That is, if there is a duplicate vertex ID in the pattern's name, it indicates that the two patterns are connected. If two patterns are not connected, they are not included in the frequent patterns. That is, as $< v_1, v_2 >$ and $< v_3, v_4 >$ have no duplicate vertex ID, the AND operation is not performed. The detected frequent patterns are stored in the frequent subgraph pattern management table, as presented in Table 5. Here, FiB, FiS, and slideNum have the same as those used for the single edge.

Table 5. Frequent subgraph pattern management table where patterns are added.

Subgraph	slideNum	FiB (Batch1)	FiB (Batch2)	FiB (Batch3)	FiS
$< v_1, v_2 >$	1	3	3	3	9
$< v_1, v_3 >$	1	3	2	3	8
$< v_1, v_4 >$	0	1	2	1	4
$< v_2, v_4 >$	−1	2	0	1	3
$< v_3, v_4 >$	0	2	1	3	6
$< v_1, v_2, v_3 >$	1	3	2	3	8
$< v_1, v_3, v_4 >$	0	2	1	3	6
$< v_1, v_2, v_3, v_4 >$	0	2	1	2	6

3.5. Incremental Frequent Subgraph Pattern Detection

The incremental frequent subgraph pattern detection is a technique to resolve duplicate calculations, which is a problem in the sliding window. Since graph are duplicated in sliding windows, performance is degraded. To resolve this problem, we store previously detected frequent subgraph pattern information in the frequent subgraph pattern management table and generate a new frequent pattern using the stored frequent pattern information. In the frequent subgraph pattern management table, whether the previously detected patterns will be frequent or infrequent in the next several sliding windows is calculated and stored; through this value, only necessary calculation is performed in the next sliding window, thereby reducing the overall computation cost.

When new graphs $G_{10} \sim G_{12}$ are inputted as shown in Figure 3, a new batch is added to the DSMatrix. If the new batch is added, FiB and FiS are calculated and stored, and the same is done in the basic frequent subgraph pattern detection. However, if slideNum is not zero, slideNum is decreased and FiB and FiS are not calculated. That is, $< v_1, v_2 >$, $< v_1, v_3 >$, and $< v_2, v_4 >$ do not calculate FiB and FiS, as presented in Table 6.

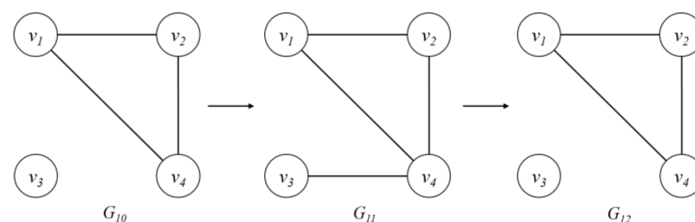

Figure 3. New input graph stream, where G_{10} is a graph at time 10, G_{11} is a graph at time 11, and G_{12} is a graph at time 12.

Table 6. DSMatrix after the window slide moves.

Edge	Contents (Batch2)	Contents (Batch3)	Contents (Batch4)
$< v_1, v_2 >$	1 1 1	1 1 1	111
$< v_1, v_3 >$	011	111	000
$< v_1, v_4 >$	110	010	111
$< v_2, v_4 >$	000	100	111
$< v_3, v_4 >$	010	111	010

When detecting frequent patterns consisting of multiple edges, a similar approach to the previous procedure is used. In Table 7, pattern $< v_1, v_2, v_3 >$ has one slideNum so that it can be identified as a frequent subgraph pattern without the need to calculate FiB and FiS. Thus, after slideNum is modified to zero, a frequent subgraph pattern is maintained. For the pattern $< v_1, v_3, v_4 >$, its slideNum is zero. Thus, FiB and FiS are calculated by applying the AND operation only to the new input batch of $< v_1, v_3 >$ and $< v_3, v_4 >$. That is, since $< v_1, v_3 >$ is 000 and $< v_3, v_4 >$ is 010, 000 & 010 = 000. Thus, it becomes FiB = 0 and FiS = 3 for the newly added part. Here, if FiS is larger than the threshold, slideNum is calculated.

Table 7. Frequent subgraph pattern detection after window slide moves.

Subgraph	slideNum	FiB (Batch2)	FiB (Batch3)	FiB (Batch4)	FiS
$< v_1, v_2 >$	$1 \rightarrow 0$	3	3		
$< v_1, v_3 >$	$1 \rightarrow 0$	2	3		
$< v_1, v_4 >$	0	2	1	3	6
$< v_2, v_4 >$	$-1 \rightarrow 0$	0	1		
$< v_3, v_4 >$	0	1	3	1	5
$< v_1, v_2, v_3 >$	$1 \rightarrow 0$	2	3		
$< v_1, v_3, v_4 >$	0	1	2	0	3
$< v_1, v_2, v_3, v_4 >$	0	2	1	0	3

4. Performance Evaluation

To prove the superiority of the proposed method, performance evaluation was conducted by comparing it with the existing methods [31,32]. For convenience, the method proposed in Reference [31] is called DSMatrix and the method proposed in Reference [32] is called SAND. Table 8 summarizes the performance evaluation environment. The performance evaluation program was implemented using Java. For performance data, arbitrarily created graph stream data and real data were used. For real data, as-Caida [41], one of the datasets provided by SNAP [42], was employed. The data were graphs that represent a connection relationship with network routers stored on a time basis. The data consist of 65,003 vertices and 30,000 edges. A total of 122 datasets were inputted according to the time sequence. The performance evaluation compared the processing time when the batch size, window slide size, and threshold value were changed. If the frequent subgraph pattern detection time is slower than the stream input rate, frequent patterns that cannot be discovered may be found. Thus, the accuracy of the frequent subgraph pattern detection results can be evaluated. Finally, the amount of memory used in the frequent subgraph pattern detection process was measured to evaluate the space efficiency of DSMatrix.

Table 8. Performance evaluation environment.

Feature	Contents
Processor	Intel(R) Core(TM) i5-4440 3.10 GHz
Memory	8 GB
Disk	1 TB
Program language	Java

When a frequent subgraph pattern is detected, if a graph stream input rate is faster than the processing time, a frequent subgraph pattern may occur in which a new graph may be lost before being detected. Thus, this study measured the frequent subgraph pattern detection time per sliding window to establish the right input rate. Figure 4 shows how much time is consumed in each window slide to verify whether data are lost. A total of 122 datasets were inputted according to the time sequence. A single batch consisted of 10 graphs, and the processing time according to the sliding window size was evaluated. The performance evaluation results showed that the frequent subgraph pattern detection time in each sliding window was calculated by measuring the processing time for each sliding window and taking the mean value. When a single sliding window consists of five batches, the processing time is approximately 11 ms. However, if an interval of the graph input is smaller than 11 ms, frequent patterns may be lost. Thus, frequent patterns can be detected accurately when the data size, sliding window size, and batch size are appropriately selected according to the data characteristics in application fields.

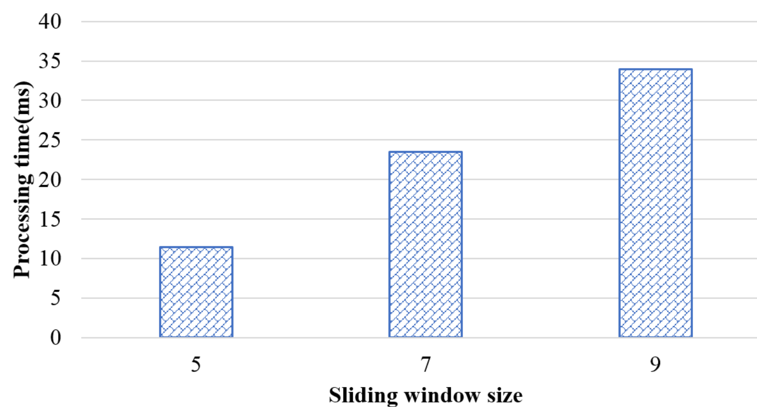

Figure 4. Processing time of the proposed method according to sliding window size.

Generally, as graph size increases, the rate of frequent subgraph pattern detection increases. As a result, if too much time is taken to detect a frequent subgraph pattern from a large number of graphs, the significance of the detected frequent subgraph pattern may be weakened. Figure 5 shows the comparison of the experimental results for data processing time between the proposed and existing methods as the number of edges increased. This experiment generated arbitrary graphs and changed the number of edges in the graphs to conduct performance evaluation. A batch consisted of 100 graphs, and a window slide consisted of five batches. In addition, a threshold was set to 80% to detect patterns that appeared more than 400 times in the window slide. In this experiment, when the number of edges was small, no significant processing time was revealed. However, when the number of edges was increased, the processing time in the proposed method was reduced by up to 60% compared to that of the existing methods. The reason for this was because duplicate processing results also increased as the number of edges increased, and the existing methods performed duplicate processing continuously.

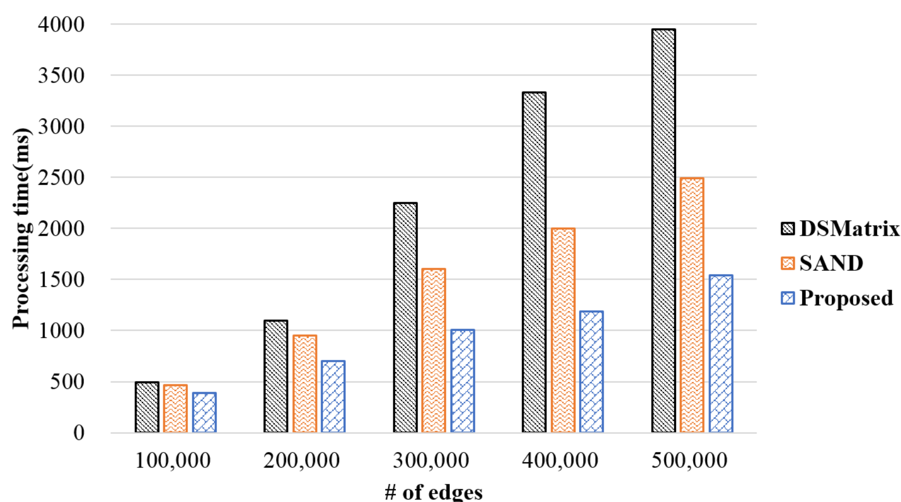

Figure 5. Processing time according to the number of edges.

DSMatrix is based on sliding windows. Accordingly, its processing time depends on the sliding window size. Thus, it is important to set a sliding window size that is suitable for different applications. Figure 6 shows the difference in processing time between the proposed and two existing methods according to the window slide size. Each graph is an arbitrarily created graph that consists of 300,000 edges. In the figure, when 20 batches are included in a single sliding window, the proposed method reduces the processing time by 63% compared to that of DSMatrix and by 50% compared to that of SAND. This result verifies that, compared to the existing methods, the performance of the proposed method improves as the number of sliding windows increases. This is because, during

several graph streams, according to the value of slideNum, it was determined that no calculation will be needed in the future when using the incremental frequent subgraph pattern detection method. As a result, the overall computation cost was reduced, as patterns not needing calculation increased when the sliding window size increased.

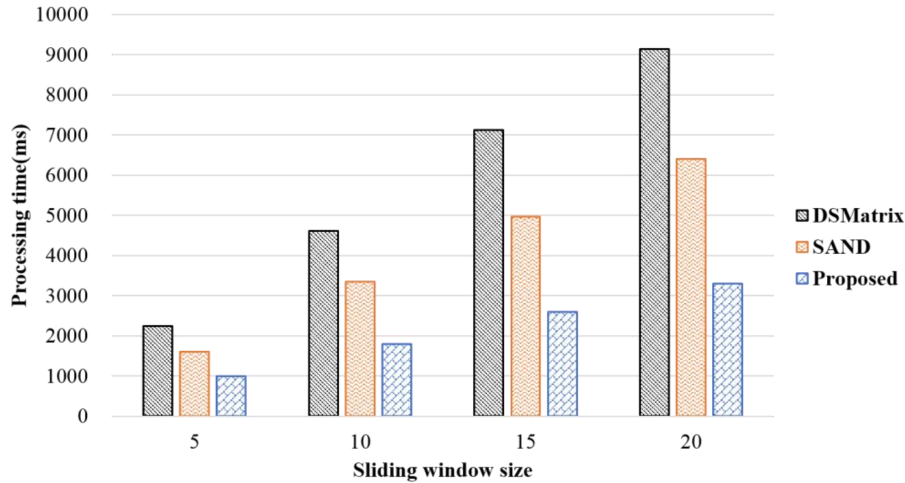

Figure 6. Processing time according to the sliding window size.

A frequent subgraph pattern means a subgraph that frequently occurs above the threshold. We confirmed through various experiments that the results of frequent pattern detection of the proposed method are identical to those of the existing methods. Therefore, we compare the proposed method with the existing methods in terms of the processing time according to the changing threshold. Figure 7 shows the processing time according to a threshold value. The graph in this performance evaluation used arbitrarily generated data, which consisted of 300,000 edges; one batch consisted of 100 graphs, and one sliding window consisted of five batches. As shown in the figure, the processing time slowed down rapidly as the threshold became smaller. This was because the number of cases to be considered when two or more frequent patterns were detected increased exponentially. However, when the threshold was set to 80%, the processing time in the proposed method was faster: 60% of that of DSMatrix and 55% of that of SAND. The reason for this is, as the threshold value became larger, slideNum was likely to increase, which improved the performance. Thus, it is important to select a threshold value that is suitable to the data and the application field.

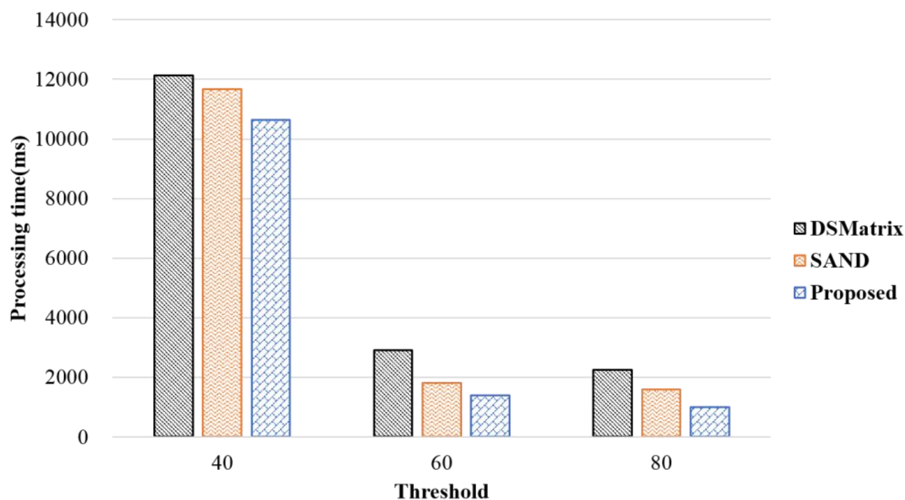

Figure 7. Processing time according to threshold.

A larger memory usage is advantageous to the frequent subgraph pattern detection of larger graph sizes. Frequent patterns may occupy more space than graph increases occupy. Thus, if the memory usage is bigger, frequent patterns of larger data can be detected. Figure 8 shows the memory usage measured using the as-Caida data. Here, the sliding window size was five, and each batch consisted of 10 graphs. The proposed method used a memory of 83 MB on average, which used more memory than SAND (62 MB on average). This was because the proposed method employs the frequent subgraph pattern management table to improve the processing time, resulting in additional memory space for pattern management. In addition, DSMatrix used 412 MB of memory on average because it employs many pointers to construct trees, and there are many duplicate data because trees are constructed for each edge. However, as the proposed method and SAND detect frequent patterns using a bit product rather than a tree, they used less memory.

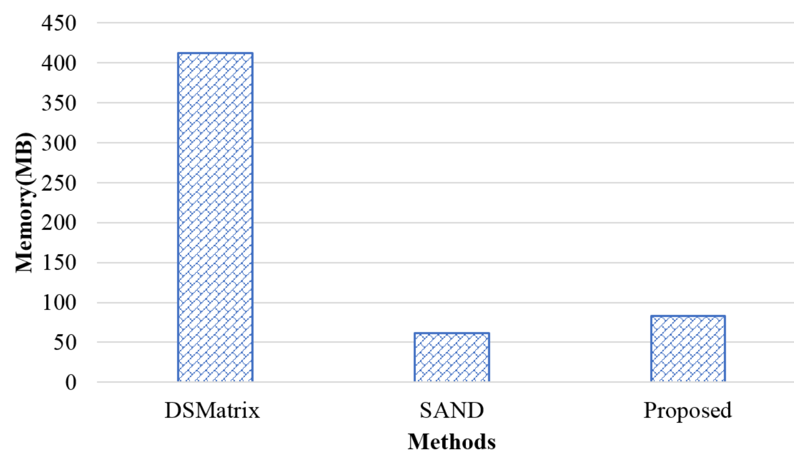

Figure 8. Memory usage.

The existing methods require high computation costs to detect frequent patterns after the initial graph is generated. In addition, the detection of frequent subgraphs using sliding windows resulted in unnecessary comparison. The proposed method reduces processing time because it manages frequent subgraph patterns and performs comparative operations only on subgraph patterns that may occur in the future. In addition, similar to the method proposed in Reference [32], only subgraphs managed in frequent subgraph patterns perform AND operation, thereby reducing the comparison operation. The proposed method reduces the number of edges, the size of the sliding window, and the processing time as the threshold changes. Memory based processing is performed because real-time processing is required to detect frequent subgraph patterns in the graph stream. Therefore, it is important to reduce memory usage when detecting frequent subgraphs. DSMatrix [31] uses a lot of memory because it constructs DSMatrix for input graph streams and constructs the FP-Trees for frequent pattern detection. SAND [32] uses the least memory since it does not deploy FP-Trees and only constructs DSMatrix for input graph streams. The proposed method constructs DSMatrix for input graph streams, similar to SAND, but additionally manages the frequent subgraph patterns to reduce unnecessary comparison operations. Therefore, it uses more memory than SAND. However, the proposed method does not use that much memory since it only manages frequently occurring subgraphs, not all subgraphs.

5. Conclusions

In this paper, we proposed an incremental processing method to detect frequent patterns from graph streams. The proposed method can reduce the processing time by managing frequent patterns discovered in previous sliding windows in a frequent subgraph pattern management table, then utilizing the data in the table for the next sliding window. It also generates more meaningful frequent patterns by considering connectivity. The performance evaluation results verified that the proposed method could reduce duplicate operations, which was an important feature since the amount of

duplicated data increased in the sliding windows when the graph and sliding window sizes increased. As a result, the processing time was reduced by 55% on average, compared to the existing methods. The proposed method manages frequent patterns in the table, so it has the limitation of needing excessive memory space to manage frequent patterns in the table, and more time is needed to scan them as the number of frequent patterns increases. Thus, for future research, a study will be conducted using an index for direct access to the required pattern to reduce the cost of scanning when the number of patterns to be managed increases.

Author Contributions: Conceptualization, K.B., J.J., D.C., and J.Y.; Methodology, K.B., J.J., and D.C.; Validation, J.J.; Data Curation [41,42], J.J., and D.C; Writing-Original Draft Preparation, K.B.; Writing—Review & Editing, J.Y.

Funding: This research was supported by the MSIT(Ministry of Science, ICT), Korea, under the ITRC(Information Technology Research Center) support program (IITP-2018-2013-1-00881) supervised by the IITP(Institute for Information & communication Technology Promotion), by the National Research Foundation of Korea(NRF) grant funded by the Korea government(MSIP) (No. 2016R1A2B3007527), by Next-Generation Information Computing Development Program through the National Research Foundation of Korea(NRF) funded by the Ministry of Science, ICT (No. NRF-2017M3C4A7069432), and by the ICT R&D program of MSIT/IITP. [B0101-15-0266, Development of High Performance Visual BigData Discovery Platform for Large-Scale Realtime Data Analysis].

Conflicts of Interest: The authors declare no conflict of interest.

References

1. Ma, S.; Li, J.; Hu, C.; Lin, X.; Huai, J. Big graph search: challenges and techniques. *Frontiers Comput. Sci.* **2016**, *10*, 387–398. [CrossRef]
2. Zhang, L.; Gao, J. Incremental graph pattern matching algorithm for big graph data. *Sci. Program.* **2018**, *2018*, 1–8. [CrossRef]
3. Labouseur, A.G.; Birnbaum, J.; Olsen, P.W.; Spillane, S.R.; Vijayan, J.; Hwang, J.; Han, W. The G graph database: efficiently managing large distributed dynamic graphs. *Distrib. Parallel Databases* **2015**, *33*, 479–514. [CrossRef]
4. Yan, D.; Bu, Y.; Tian, Y.; Deshpande, A. Big graph analytics platforms. *Found. Trends Databases* **2017**, *7*, 1–195. [CrossRef]
5. Jiang, F.; Leung, C.K. Mining interesting “following” patterns from social networks. In Proceedings of the International Conference on Data Warehousing and Knowledge Discovery, Munich, Germany, 2–4 September 2014; pp. 308–319.
6. Tian, Y.; McEachin, R.C.; Santos, C.; States, D.J.; Patel, J.M. SAGA: a subgraph matching tool for biological graphs. *Bioinformatics* **2017**, *23*, 232–239. [CrossRef] [PubMed]
7. Fariha, A.; Ahmed, C.F.; Leung, C.K.; Abdullah, S.M.; Cao, L. Mining frequent patterns from human interactions in meetings using directed acyclic graphs. In Proceedings of the Pacific-Asia Conference on Knowledge Discovery and Data Mining, Gold Coast, Australia, 14–17 April 2013; pp. 38–49.
8. Li, P.; Heo, L.; Li, M.; Ryu, K.H.; Pok, G. Protein function prediction using frequent patterns in protein-protein interaction networks. In Proceedings of the International Conference on Fuzzy Systems and Knowledge Discovery, Shanghai, China, 26–28 July 2011; pp. 1616–1620.
9. Peng, J.; Yang, L.; Wang, J.; Liu, Z.; Li, M. An efficient algorithm for detecting closed frequent subgraphs in biological networks. In Proceedings of the International Conference on BioMedical Engineering and Informatics, Sanya, China, 28–30 May 2008; pp. 677–681.
10. Mrzic, A.; Meysman, P.; Bittremieux, W.; Moris, P.; Cule, B.; Goethals, B.; Laukens, K. Grasping frequent subgraph mining for bioinformatics applications. *BioData Mining* **2018**, *11*, 1–24. [CrossRef] [PubMed]
11. Namaki, M.H.; Lin, P.; Wu, Y. Event pattern discovery by keywords in graph streams. In Proceedings of the International Conference on Big Data, Boston, MA, USA, 11–14 December 2017; pp. 982–987.
12. Manzoor, E.A.; Milajerdi, S.M.; Akoglu, L. Fast memory-efficient anomaly detection in streaming heterogeneous graphs. In Proceedings of the ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, San Francisco, CA, USA, 13–17 August 2016; pp. 1035–1044.
13. Choudhury, S.; Holder, L.B.; Chin, G.; Agarwal, K.; Feo, J. A Selectivity based approach to continuous pattern detection in streaming graphs. In Proceedings of the International Conference on Extending Database Technology, Brussels, Belgium, 23–27 March 2015; pp. 157–168.

14. Vlassopoulos, C.; Kontopoulos, I.; Apostolou, M.; Artikis, A.; Vogiatzis, D. Dynamic graph management for streaming social media analytics. In Proceedings of the ACM International Conference on Distributed and Event-based Systems, Irvine, CA, USA, 20–24 June 2016; pp. 382–385.
15. Edouard, A.; Cabrio, E.; Tonelli, S.; Thanh, N.L. Graph-based event extraction from twitter. In Proceedings of the International Conference Recent Advances in Natural Language Processing, Varna, Bulgaria, 2–8 September 2017; pp. 222–230.
16. Eberle, W.; Holder, L. Identifying anomalies in graph streams using change detection. In Proceedings of the KDD Workshop on Mining and Learning in Graphs, San Francisco, CA, USA, 14 August 2016.
17. McGregor, A. Graph stream algorithms: A survey. *SIGMOD Record* **2014**, *43*, 9–20. [CrossRef]
18. Guo, Y.; Hong, S.; Chafi, H.; Iosup, A.; Epema, D.H.J. Modeling, analysis, and experimental comparison of streaming graph-partitioning policies. *J. Parallel Distrib. Comput.* **2017**, *108*, 106–121. [CrossRef]
19. Aridhi, S.; Montresor, A.; Velegrakis, Y. BLADYG: A graph processing framework for large dynamic graphs. *Big Data Res.* **2017**, *9*, 9–17. [CrossRef]
20. Aggarwal, C.C.; Li, Y.; Yu, P.S.; Jin, R. On dense pattern mining in graph streams. *PVLDB* **2010**, *3*, 975–984. [CrossRef]
21. Kim, K.; Seo, I.; Han, W.; Lee, J.; Hong, S.; Chafi, H.; Shin, H.; Jeong, G. TurboFlux: A fast continuous subgraph matching system for streaming graph data. In Proceedings of the ACM SIGMOD International Conference on Management of Data, Houston, TX, USA, 10–15 June 2018; pp. 411–426.
22. Riedy, J. Streaming graph analysis: new models, new architectures. In Proceedings of the ACM International Conference on Computing Frontiers, Ischia, Italy, 8–10 May 2018; p. 268.
23. Yang, M.; Rashidi, L.; Rajasegarar, S.; Leckie, C. Graph stream mining based anomalous event analysis. In Proceedings of the Pacific Rim International Conference on Artificial Intelligence, Nanjing, China, 28–31 August 2018; pp. 891–903.
24. Boobalan, M.P.; Lopez, D.; Gao, X.Z. Graph clustering using k-Neighbourhood Attribute Structural similarity. *Appl. Soft Comput.* **2016**, *47*, 216–223. [CrossRef]
25. Gao, J.; Zhou, C.; Zhou, J.; Yu, J.X. Continuous pattern detection over billion-edge graph using distributed framework. In Proceedings of the International Conference on Data Engineering, Chicago, IL, USA, 31 March–4 April 2014; pp. 556–567.
26. Sun, Z.; Wang, H.; Wang, H.; Shao, B.; Li, J. Efficient subgraph matching on billion node graphs. *PVLDB* **2012**, *5*, 788–799. [CrossRef]
27. Valari, E.; Kontaki, M.; Papadopoulos, A.N. Discovery of top-k dense subgraphs in dynamic graph collections. In Proceedings of the International Conference on Scientific and Statistical Database Management, Chania, Crete, Greece, 25–27 June 2012; pp. 213–230.
28. Abdelhamid, E.; Canim, M.; Sadoghi, M.; Bhattacharjee, B.; Chang, Y.; Kalnis, P. Incremental frequent subgraph mining on large evolving graphs. *IEEE Trans. Knowl. Data Eng.* **2017**, *29*, 2710–2723. [CrossRef]
29. Ramraja, T.; Prabhakarb, R. Frequent Subgraph Mining Algorithms—A Survey. *Procedia Comput. Sci.* **2015**, *47*, 197–204. [CrossRef]
30. Leung, C.K.; Khan, Q.I. DSTree: A tree structure for the mining of frequent sets from data streams. In Proceedings of the International Conference on Data Mining, Hong Kong, China, 18–22 December 2006; pp. 928–932.
31. Braun, P.; Cameron, J.J.; Cuzzocrea, A.; Jiang, F.; Leung, C.K. Effectively and efficiently mining frequent patterns from dense graph streams on disk. In Proceedings of the International Conference in Knowledge Based and Intelligent Information and Engineering Systems, Gdynia, Poland, 15–17 September 2014; pp. 338–347.
32. Cuzzocrea, A.; Han, Z.; Jiang, F.; Leung, C.K.; Zhang, H. Edge-based mining of frequent subgraphs from graph streams. In Proceedings of the International Conference in Knowledge Based and Intelligent Information and Engineering Systems, Singapore, 7–9 September 2015; pp. 573–582.
33. Ismail, W.N.; Hassan, M.M.; Alsalamah, H.A. Mining of productive periodic-frequent patterns for IoT data analytics. *Future Gener. Comp. Syst.* **2018**, *88*, 512–523. [CrossRef]
34. Chen, F.; Deng, P.; Wan, J.; Zhang, D.; Vasilakos, A.V.; Rong, X. Data mining for the internet of things: Literature review and challenges. *IJDSN* **2015**, *11*, 1–14. [CrossRef]
35. Tsai, C.; Lai, C.; Chiang, M.; Yang, L.T. Data mining for internet of things: A survey. *IEEE Commun. Surv. Tutor.* **2014**, *16*, 77–97. [CrossRef]

36. Elseidy, M.; Abdelhamid, E.; Skiadopoulos, S.; Kalnis, P. GRAMI: Frequent subgraph and pattern mining in a single large graph. *PVLDB* **2014**, *7*, 517–528. [CrossRef]
37. Tanbeer, S.K.; Leung, C.K.; Cameron, J.J. Interactive mining of strong friends from social networks and its applications in E-commerce. *J. Org. Comput. E. Commerce.* **2014**, *24*, 157–173. [CrossRef]
38. Bifet, A.; Holmes, G.; Pfahringer, B.; Gavaldà, R. Mining frequent closed graphs on evolving data streams. In Proceedings of the ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, San Diego, CA, USA, 21–24 August 2011; pp. 591–599.
39. Shivraj, V.L.; Rajan, M.A.; Balamuralidhar, P. A graph theory based generic risk assessment framework for internet of things (IoT). In Proceedings of the International Conference on Advanced Networks and Telecommunications Systems, Bhubaneswar, India, 17–20 December 2017; pp. 1–6.
40. Almuammar, M.; Fasli, M. Pattern discovery from dynamic data streams using frequent pattern mining with multi-support thresholds. In Proceedings of the International Conference on the Frontiers and Advances in Data Science, Xi'an, China, 23–25 October 2017; pp. 35–40.
41. Center for Applied Internet Data Analysis. Available online: http://www.caida.org (accessed on 20 July 2018).
42. Stanford Large Network Dataset Collection. Available online: https://snap.stanford.edu/data/ (accessed on 20 July 2018).

© 2018 by the authors. Licensee MDPI, Basel, Switzerland. This article is an open access article distributed under the terms and conditions of the Creative Commons Attribution (CC BY) license (http://creativecommons.org/licenses/by/4.0/).