Prefix Tree Based MapReduce Approach for Mining Frequent Subgraphs

Supriya Movva, Saketh Prata(✉), Sai Sampath, and R. G. Gayathri

Department of Computer Science and Engineering, Amrita School of Engineering, Amritapuri, Amrita Vishwa Vidyapeetham, Coimbatore, India
supriya.movva.212@gmail.com, pratasaketh@gmail.com

Abstract. The frequent subgraphs are the subgraphs which appear in a number, more than or equal to a user-defined threshold. Many algorithms assume that the apriori based approach yields an efficient result for finding frequent subgraphs, but in our research, we found out that Apriori algorithm lacks scalability with the main memory. Frequent subgraph mining using Apriori algorithm with FS tree uses adjacency list representation. FS tree is a prefix tree data structure. It implements the algorithm in two phases. In the first phase, it uses the Apriori algorithm to find frequent two edge subgraphs. In the second phase, it uses FS-tree algorithm to search all the frequent subgraphs from frequent two edge subgraphs. Scanning the dataset for every candidate is the drawback of the Apriori algorithm, so the Apriori algorithm with FS-tree is used to overcome the multiple scanning. This algorithm is also implemented in an assumption that the data set fits well in memory. In this paper, we propose parallel map-reduce based frequent subgraph mining technique performed in a distributed environment on the Hadoop framework. The experiments validate the efficiency of the algorithm for generating frequent subgraphs in large graph datasets.

Keywords: Frequent subgraph mining · Subgraph · Support threshold · Hadoop framework · Map-reduce

1 Introduction

Data mining is a process of discovering patterns in large data sets. One of our research interests is finding frequent itemsets that occur in large graph datasets. With the increasing demand on the analysis of large amounts of complex data, graph mining has become an active and essential theme in data mining. The task of graph mining is extracting interesting patterns from graphs that describe the underlying data and could be used further. Mining patterns in biochemical structures [1,2], anomaly detection, chemoinformatics [3], network flow analysis and social network analysis [4] are some of the applications of graph mining.

© ICST Institute for Computer Sciences, Social Informatics and Telecommunications Engineering 2019
Published by Springer Nature Switzerland AG 2019. All Rights Reserved
N. Kumar and R. Venkatesha Prasad (Eds.): UBICNET 2019, LNICST 276, pp. 218–232, 2019.
https://doi.org/10.1007/978-3-030-20615-4_17

Frequent subgraph mining is an important research in graph mining to find all the subgraphs that appear frequently in database according to given frequency threshold. There are two different ways for finding frequent subgaphs. (1) Transaction setting [5] (2) Single graph setting [6]. The number of transactions containing the frequent pattern is said to be transaction setting and the number of times the pattern appears in the whole graph is said to be single graph setting. There are many proposed algorithms for finding frequent subgraphs, but all these algorithms are on the assumption that the graph data fits well in memory. As the data size grows memory becomes a problem and computational time rises drastically. Hence, an efficient way to compute large datasets is to process them in parallel by distributing the data among several nodes and combining the results. Distributed computing frameworks like Pregel, Hadoop are popular of their kind.

Map Reduce programming model [7] has been the most successful for mining frequent subgraphs on a distributed computing platform. Hadoop framework uses a distributed file system that is particularly optimized to improve the IO performance while handling Big data. The main reason behind using Hadoop framework for mining frequent subgraphs is its computational performance and efficiency. Another reason is, the higher level of abstraction that it provides, which keeps many system levels hidden from the programmers and allow them to concentrate more on problem specific computational logic.

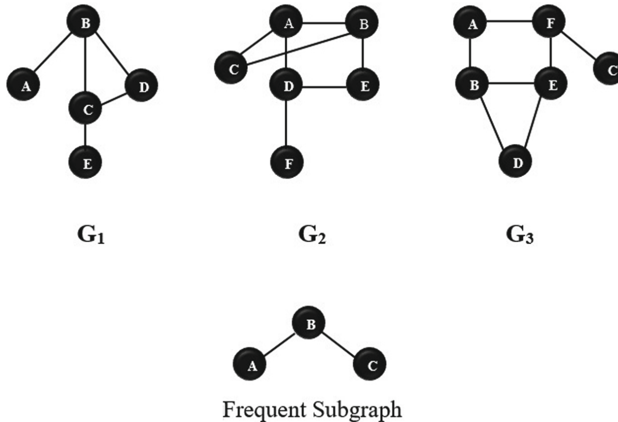


Fig. 1. Frequent subgraph

Solving the task of Frequent Subgraph Mining (FSM) using Hadoop framework is challenging. In this paper we use Improvised Apriori with frequent subgraph tree Algorithm in distributed computing framework to find the frequent subgraphs. If the input graphs are partitioned over different data nodes, calculating the support count in the data nodes and then finding the frequent subgraphs

does not yield an efficient result because of the data set partition. Here, we propose an algorithm that generates all the one-edge candidates and the occurrence count from the given graph data set in the data nodes and then find the frequent subgraphs using support count.

The rest of the paper is framed as follows: Sect. 2 exchanges its views on several works related to Frequent Subgraph Mining. Section 3 presents the proposed approach, important concepts related to the algorithm. Section 4 presents the implementation of the proposed algorithm and the experimental results.

2 Related Work

Extracting or finding frequent subgraphs in a single large graph or set of graphs has been popular in recent times. If the graph dataset is extensive the time and space complexity will be high. Many researchers have implemented many algorithms to find frequent subgraphs in both single and multiple graph settings in an efficient way. Single graph setting has one large graph, and multiple graph setting comprises of a collection of graphs. These frequent itemsets are used for discovering association rules, for extracting common patterns and for classification. Developing algorithms for finding frequent subgraphs is computationally intensive as subgraph isomerism play a vital role throughout the computation. There are two types of approaches for finding frequent subgraphs. (1) Apriori-based algorithms and (2) Pattern-based approach. In this section, we present different Apriori and Pattern based algorithms with many advantages and drawbacks.

Apriori Graph-based Mining (AGM) is an algorithm to find frequent subgraphs [8]. It uses Apriori-based approach. This algorithm generates candidate graphs, merges any two candidate graphs at an instant and finds whether the obtained graph is frequent or not. Two graphs of size k are merged together to form a resultant graph of size $(k+1)$. A level-wise search is used to find the frequent subgraphs. The AGM algorithm is used to identify the set of graphs which are connected and the set of graphs which are not connected. It efficiently found all the frequent subgraphs, but the complexity was high.

FSG is another Apriori-based algorithm which uses edge based candidate generation method [5]. Candidate subgraphs are generated by adding edge to the previous subgraph. So, in every iteration, the generated candidate subgraph size is exactly one greater than the previous frequent ones. Candidate pruning is also done if the generated candidate does not satisfy minimum threshold. It is very costly because it uses isomorphic testing and generates multiple candidates.

Edge-disjoint path join algorithm [9] abides Apriori-based approach which uses edge disjoint paths as building blocks. The number of disjoint paths is the measuring factor of this algorithm. It also efficiently finds frequent subgraphs, but it is computation intensive.

Fast Frequent subgraph mining (FFSM) considers large dense graphs with fewer labels for finding frequent itemsets. Data sets used here are chemical [10]. It uses the vertical level search strategy to reduce the number of candidate generation. Limitation for FFSM algorithm is that it is an NP-Complete problem. Experimentation showed that FFSM outperformed gSpan.

Molecular Fragments Identification Technique [11,12] is a pattern based approach algorithm used to find regular core structures which are found in all given molecular structures and generates an embedding list. In this algorithm, in every iteration, one more edge is added to the previous frequent subgraph which leads to duplicate candidate generation.

Spanning Tree-based Maximal Graph Mining (SPIN) is a pattern based algorithm. The main aim of this algorithm is to mine subgraphs that are not part of any frequent subgraph [13]. It uses spanning tree approach to discover maximal frequent subgraphs. Pruning technique used in the SPIN algorithm is bottom-up pruning. It saves space by using these techniques.

Temporal pattern subgraph Mining (TSM) finds patterns having temporal information [14]. It uses forward unnecessary checking scheme and backward unnecessary checking scheme to find a frequent temporal graph. It does not generate unnecessary candidates and scans the database once. It is an extension of the DFS search strategy. It is an efficient algorithm for finding patterns which have temporal information.

gSpan algorithm [16] generates a tree-like structure (DFS code tree) over all possible patterns, in which every node represents a DFS code for a graph pattern. The ith level of a code tree contains DFS of all the subgraphs of size (i-1). Each subgraph is generated by adding one extra edge to subgraphs which are present in the previous level of the tree. It preserves the transaction list for discovered graph and pruning is done by deleting nodes which do not satisfy minimal DFS code.

All these algorithms are in assumption that data set fits well in memory, but if data set size is huge, all these algorithms do not give an efficient result, and even though if it can find all the frequent subgraphs, it takes a lot of time and space to find frequent item sets. Hence, some of the researchers implemented a few algorithms in a parallel or distributed environment to find frequent subgraphs in multiple systems in parallel, which give efficient output and reduce both time and space complexities.

There exist some algorithms on adaptive parallel mining for CMP Architectures [18]. Map-Reduce programming model has been used to mine frequent patterns where the transactions in input database are simpler combinatorial objectives such as set or sequence [19,20,22–24]. These algorithms do not implement any method to avoid duplicate candidates generated. Another problem for the iterative approach of Map-Reduce is that it requires many iterations to get the final output.

Frequent subgraph mining using Apriori algorithm with FS tree uses adjacency list representation. It implements the algorithm in two phases. In the first phase, it uses the Apriori algorithm to find frequent two edge subgraphs. In second phase it uses FS-tree algorithm to find all the frequent subgraphs from frequent two edge subgraphs. Scanning the dataset for every candidate is the drawback of the Apriori algorithm, so the Apriori algorithm with FS-tree is used to overcome the multiple scanning. This algorithm is also implemented in an assumption that the data set fits well in memory. In this paper, we implemented

the same Apriori algorithm with FS-tree in a distributed environment using the Map-Reduce programming model in a distributed environment to reduce execution time and enhance efficiency.

3 Proposed Work

When a large number of graphs are given as input to the Apriori algorithm, it can lead to inefficient computation of the desired solution. So, dividing the graphs into several non-empty sets with a limited number of graphs in each set and computing them in distributed environment improves time efficiency. In this paper, we propose a two-phase approach to generate all the subgraphs that frequently occur in the graph data set containing a large number of graphs. The proposed method starts with the Data pre-processing, Candidate Generation, Support Counting followed by the phases where these concepts are used.

Let $D = \{G_1, G_2 ..., G_n\}$ be a graph data set where each G_i represents an undirected graph. Let the data set be divided into three sets say, D_1, D_2, D_3 such that $D_1 = \{G_1, G_2 ..., G_i\}$, $D_2 = \{G_{i+1}, G_2 ..., G_k\}$, $D_3 = \{G_{k+1}, G_2 ..., G_n\}$ where i, j, k are any arbitrary values representing the graphs in the data set.

3.1 Data Pre-processing

The graph data set is pre-processed and stored in the form of an adjacency list. Let $D = \{G_1, G_2 ..., G_n\}$ be a graph data set where each G_i represents an undirected graph. Now, each G_i is of the form (V, E) where V is a set of Vertices and E is the set of edges in the graph. Each graph is represented as an adjacency list. The idea of computing the frequent subgraphs is based on the iterative approach of the Apriori algorithm computes the Frequent subgraphs in two steps. One is Candidate Generation, and the other is Support Counting.

3.2 Candidate Generation

A subgraph in a graph is said to be a candidate. To find all frequent subgraphs in a given graph data set, we need to generate all the subgraphs and check whether each one is a frequent subgraph or not. Thus candidate generation is an important step in Frequent Subgraph mining. The candidates are generated as k-edge subgraphs starting from $k = 0$ which are the vertices. The candidates are thus generated by adding one edge to the previous frequent subgraph. Thus to form a candidate with $k + 1$ edges, we combine two k-edge subgraphs. The two k-edge subgraphs are thus selected such that they have the same $k - 1$ size subgraph. This common subgraph is often referred to as Core of the subgraph. The frequency of each subgraph is also calculated. Algorithm 1 shows the stepwise implementation of Candidate Generation function.

Algorithm 1. Candidate Generation

Input : F_{k+1}, set of k+1 edge frequent subgraphs
Output: C_{k+2}, set of k+2 edge candidate subgraphs along with their count
//s_1 and s_2 are subgraphs such that $s_1 \in F_{k+1} and s_2 \in F_{k+1}$
//C_i is the list of candidate subgraphs of i-edge length and corresponding count
//$s_1 \cup s_2$ defines an operation where a k+2 edge subgraph is formed using k+1 edge subgraphs
//If k<0, s_1 and s_2 are vertices and have no common edges, then $s_1 \cup s_2$ is a valid combination iff $s_1 \cup s_2 \in g$

1. **for** all $(s_1, s_2) \in F_{k+1}$
2. **if** $(s_1 \cup s_2 \in C_{k+2})$
3. $C_{k+2}(s_1 \cup s_2) \leftarrow C_{k+2}(s_1 \cup s_2) + 1$
4. **else**
5. $C_{k+2} \leftarrow C_{k+2} \cup (s_1, s_2)$
6. $C_{k+2}(s_1 \cup s_2) \leftarrow 1$
7. **end for**
8. return C_{k+2}

The candidate generation can be illustrated using an example. Let the value of k be one. To form a two-edge candidate, Let A-B-C be a two edge subgraph with B as the root. This two edge subgraph can be formed from two one-edge subgraphs A-B and B-C. These subgraphs have a $k-1$ size subgraph, i.e. a vertex B in common. Thus, to form a $k+1$ edge candidate, two K-edge candidates are to be combined with a $k-1$ edge as the core. Figure 2 shows the detailed explanation of candidate generation for the above-stated example.

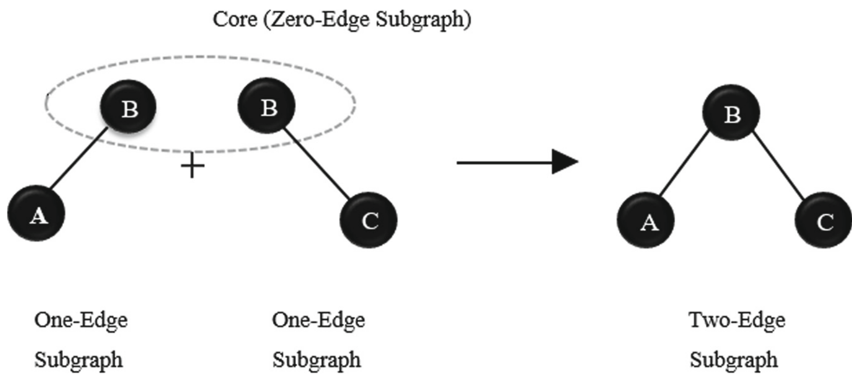


Fig. 2. Example of candidate generation

3.3 Support Counting

Support of a graph is the parameter used to determine whether a graph is a frequent subgraph or not. The total number of times a subgraph is appearing in the graph dataset is termed as the frequency of that graph. Support of a given graph is defined as the frequency of each graph divided by the total number of graphs in the graph data set. To check if a graph is frequent or not, the parameter

defined by the user to compare is the threshold support. If the calculated support is greater than or equal to the user-defined threshold support, that graph can be called as a Frequent subgraph. These frequent subgraphs are used to find the subgraphs with more edges added to them.

Mapper

Algorithm 2. Mapper

Input : *G: a graph dataset, σ : minimum support.*
Output: *Set of one-edge graphs with their frequency, adjacency list.*
//C_i: Candidate subgraph set of i length edges
//cg(F_i): Candidate generation function using i-edge frequent subgraphs
//adjacency_list(g): Create adjacency list for graph g
//Reducer(a,b): Send parameters a , b to reducer

1. $C_1 \leftarrow NULL$
2. adj_list $\leftarrow NULL$
3. **while** g ε G **do**
4. $F_0 \leftarrow set\ of\ vertices\ in\ g$
5. $C_1 \leftarrow C_1 \cup\ cg(F_0)$
6. adj_list $\leftarrow adj_list \cup adjacency_list(g)$
7. **end while**
8. Reducer(adj_list, C_1)

The Mapper operates on the block of data given to it and finds the one edge subgraphs along with their repeating count using a level-by-level expansion of the Apriori algorithm. For each sub-graph in a set, the Mapper calculates its frequency. The calculation of the support at this stage will not give the desired solution because the support of a graph calculated at one node may not be greater than or equal to the threshold support but as a whole when the support of sub-graph combined from all the nodes may be greater than or equal to the threshold support.

The set of vertices is formed from the graph. Each vertex is taken, combined with another vertex so as to form a one edge subgraph. Now the presence of this edge can be checked from the adjacency list. Consider an edge A-B. We can check if A-B is an edge by traversing through the list in the Adjacency list whose first element is A. In this way, the one-edge candidates are formed. The frequency of each one-edge is calculated which forms a (key, value) pair where the one-edge candidate is the key, and the frequency of the candidate is the value. Since several blocks of data are being processed at the same time; we achieve computational efficiency. This (key, value) is the output of the Mapper and is given as the input to the Reducer. Algorithm 2 shows the step-wise implementation of Mapper.

Reducer

Algorithm 3. Reducer

Input : *adj_list,* C_1, σ
Output: *Set of frequent two-edge subgraphs.*
//C_i: Candidate subgraph set of i length edges
//cg(F_i): Candidate generation function using i-edge frequent subgraphs
//$\delta(C_i)$:Count of the graph C_i
//σ : Minimum support threshold
//F.add(C_i): Add element C_i to set F

1. F $\leftarrow NULL$
2. $F_1 \leftarrow NULL$
3. $F_2 \leftarrow NULL$
4. **for** all candidate $C_i \ \varepsilon \ C_1$ **do**
5. Support(C_i) = $\delta(C_1)/T$
6. **if** Support(C_i) $\geq \sigma$ **then**
7. F_1.add(C_i)
8. **end if**
9. **end for**
10. F$\leftarrow F \cup F_1$
11. $C_2 \leftarrow cg(F_1)$
12. **for all** candidate $C_i \ \varepsilon \ C_2$ **do**
13. Support(C_i) = $\delta(C_i)/T$
14. **if** Support(C_i) $\geq \sigma$ **then**
15. F_2.add(C_i)
16. **end if**
17. **end for**
18. F$\leftarrow F \cup F_2$
19. FS-Tree()

The (key, value) pairs received from Mapper nodes are combined in the reducer. It aggregates the frequency of a sub-graph using the key which is the one-edge candidate. Thus the support of each one-edge candidate is calculated. The support calculated is then compared with the threshold support. The candidates whose support is greater than or equal to the threshold are made into a set and are said to be frequent one-edge subgraphs.

Two-edge subgraphs can be formed by combining the frequent one-edge subgraphs with zero-edge subgraph as the core. The two-edge candidates are generated by checking their presence in the adjacency list. The set of frequent two-edge subgraphs is formed by choosing the candidates whose support is greater than equal to the threshold support. This set is given to a tree data structure named as FS-TREE which is built to determine the further frequent subgraphs. Algorithm 3 shows the step-wise implementation of Reducer.

The main disadvantage of the Apriori algorithm is its memory usage. A large number of candidates are generated, and it becomes costly to store and scan them. Hence a tree data structure is used to store the frequent subgraphs. The tree data structure uses the concept of a prefix-tree. The prefix tree is a tree data structure where the children of a node share the common prefix with that

node, and the root node is associated with null. The main aim of this tree is to save some amount of memory to store the candidates. Traversal through the tree extracts the frequent subgraphs.

Algorithm 4. FS-TREE

Input : *Frequent two-edge subgraphs.*
Output: *FS-Tree with all frequent subgraphs.*
//F: Set of frequent two-edge subgraphs and one-edge subgraph
//insert(S,T):If T has a child C and S is the one edge expansion of C, then insert S as the child of C.

1. root ← $NULL$
2. **for** all $C_i \in F_2$
3. insert(C_i,root)
4. **end for**

The input to the data structure is the set of frequent two-edge subgraphs. The root node is created. The frequent two-edge subgraphs are added to the root as children. Thus, the first level of the tree contains all the frequent two-edge subgraphs along with their frequencies. To add a subgraph of more number of edges, those are added as children to the nodes which have the same prefix and then the frequency is incremented. Once, the tree is constructed, each level contains all the frequent subgraphs having the same number of edges along with their support value. On traversing to the higher levels, we can find the frequent subgraphs of a higher number of edges. Thus all the frequent subgraphs can be found. Algorithm 4 shows the step-wise implementation of FS Tree.

The above algorithm can be well represented using an example. Consider the frequent subgraph stated in the Fig. 1. There A-B-C is a frequent subgraph. Let us assume A-B-D is a frequent subgraph. The FSG A-B-D can be obtained by expanding the FSG A-B-C at B, i.e., the edge B-D can be added to A-B-C. This reduces the necessity of two different nodes, and those can be represented using a single node as shown in Fig. 3.

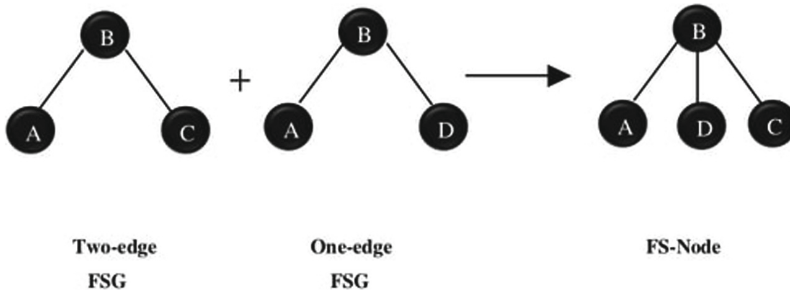


Fig. 3. Implementation of FS-TREE

4 Experimental Validation and Verification

The algorithm is implemented in Python. The operating system is Ubuntu 16.04. The Hadoop version used is 3.0.0. The experiments were performed on a CPU which possesses a 3.1 GHz quad-core Intel processor with 4 GB memory and 1 TB of storage.

All the values are experimented thrice, and the average values are taken. Following are the results of our experiments, implemented in a Hadoop cluster having four nodes. One node serves as the master node, and the remaining three nodes are the data nodes also known as slave nodes [22].

A set of synthetic graphs is generated to evaluate the performance of the proposed approach. The number of graphs in the data set range from 20 to 5000. Each graph contains 10–14 vertices. Each graph contains 25–30 edges.

Table 1. Number of FSG with varying threshold

Number of graphs in data set	Number of FSG with threshold 0.1	Number of FSG with threshold 0.2	Number of FSG with threshold 0.3
20	417	184	85
50	463	161	75
100	417	141	71
500	344	137	64
2000	312	138	60

Initially, a data set containing 20 graphs is taken. Each graph has at most 13 vertices and 29 edges. The threshold value was set to 0.1. A total of 417 frequent subgraphs including both the one edge and two edge subgraphs are obtained. When the threshold value is increased to 0.2, only 184 frequent subgraphs are obtained in total.

Further, the number of graphs in the data set is increased. A data set with 100 graphs is taken. When the algorithm is implemented with the minimum threshold of 0.1, 417 frequent subgraphs are obtained in total. Upon increasing the threshold value to 0.2, the number of frequent subgraphs decreased to 141. The reason can be illustrated using an example below.

Consider a subgraph G_i. Assume, it is occurring in 15 graphs in the dataset. Let the total number of graphs in the dataset be 100. Hence the support of graph G_i is 0.15. If the minimum support threshold is 0.1, subgraph G_i has supported greater than 0.1 and hence can be stated as a Frequent subgraph. But if the minimum support threshold value is 0.2, the subgraph G_i has support less than 0.2 and hence cannot be called as a frequent subgraph. Thus, with increasing the threshold, many subgraphs may not have their support greater than or equal to the updated minimum support threshold. Thus, with an increasing threshold

Table 2. Experimental results with threshold 0.1

No of graphs in dataset	Apriori (seconds)	Apriori with FS-Tree (seconds)	Apriori with FS-Tree using Hadoop (seconds)
20	10.98321	7.6301	0.194902
50	14.5764	11.9728	0.273901
100	28.9214	19.9214	0.236587
500	178.3206	142.5293	0.925664
2000	2016.65045	2011.093	4.157381

value, the number of frequent subgraphs decrease (Table 1). The term *frequent subgraph* is denoted as FSG.

As the number of graphs increases in the data set, the time of execution increases in the traditional algorithm because of multiple scanning. But, the proposed algorithm takes comparatively very less time which can be concluded from the results in the table below.

Table 2 shows the results for the synthetic graph data set when the threshold value is set to 0.1. The time of execution for 20 graphs in traditional Apriori algorithm is 10.98321 s. When the number of graphs increased to 50, the time of execution in traditional Apriori algorithm is 14.5764 s. The same is the case when the data is tested using Apriori algorithm using FS-TREE and also the Apriori algorithm using FS-TREE implemented in Hadoop.

Table 3. Experimental results with threshold 0.2

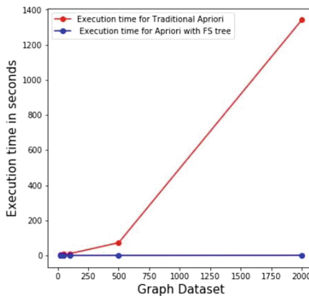
No of graphs in dataset	Apriori (seconds)	Apriori with FS-Tree (seconds)	Apriori with FS-Tree using Hadoop (seconds)
20	5.67543	3.3156	0.148771
50	10.1214	9.3213	0.216901
100	21.2494	15.1247	0.161685
500	91.6564	77.53	0.778358
2000	1865.51075	1561.328	3.242671

The time of execution shows a similar pattern for increasing threshold values. When the threshold value is increased to 0.2, the time execution increased with the increase in the number of input graphs (Table 3). The results for the time of execution with threshold 0.3 are briefed in (Table 4).

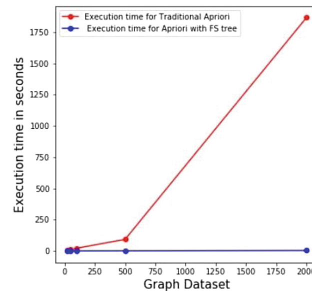
The time of execution for a given graph, at a given threshold is compared for the Apriori algorithm, improvised Apriori algorithm using FS-Tree in (Fig. 4). The time taken for a traditional Apriori algorithm is an exponential curve, whereas, the time taken by improvising the Apriori algorithm using FS-Tree

Table 4. Experimental results with threshold 0.3

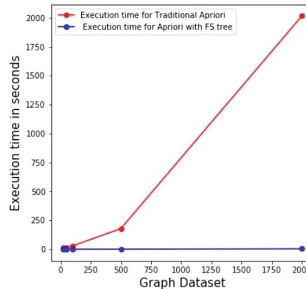
No of graphs in dataset	Apriori (seconds)	Apriori with FS-Tree (seconds)	Apriori with FS-Tree using Hadoop (seconds)
20	3.9860	1.6757	0.138207
50	7.30981	3.4252	0.184441
100	10.0919	7.54593	0.104866
500	72.4203	69.00373	0.473741
2000	1341.1567	1032.199	1.726677



a) Threshold - 0.1



b) Threshold - 0.2



c) Threshold - 0.3

Fig. 4. Execution time vs support threshold

is a linear curve. When the same algorithm is implemented in a distributed environment using Hadoop, the slope of the curve is almost zero. This implies that the time of execution reduced drastically when the algorithm is implemented in Hadoop.

The primary use of FS-Tree is to reduce the space complexity. The tree data structure is built to reduce the number of two-edge candidates. The two-edge candidates sharing common edges can be grouped as discussed in the earlier sections.

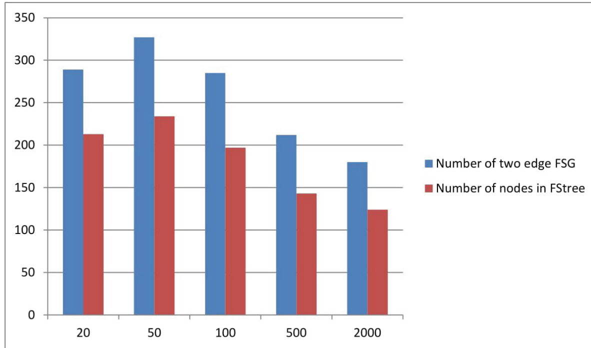


Fig. 5. Graphs with threshold 0.1

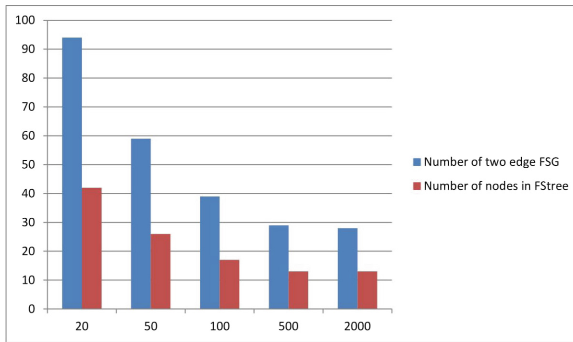


Fig. 6. Graphs with threshold 0.2

The above graph (Fig. 5) shows the number of Two-edge FSG compared with the number of nodes in the FS-Tree for a given threshold of 0.1. Consider, the graph data set with 100 graphs. The number of two-edge FSG with threshold of 0.1 is 285, and the number of nodes used to represent the two-edge FSGs using FS-Tree is 197.

With an increasing threshold, the number of two-edge FSG vs the number of nodes showed a similar pattern (Fig. 6). Consider, the threshold 0.2 for the same data set with 100 graphs. The number of two-edge FSG obtained are 39, and the number of nodes used to represent using FS-Tree is 17. In (Fig. 7), the experimental results for the number of two-edge FSG vs the number of nodes in FS-Tree for threshold 0.3 are shown.

Thus, from the above experimental results, we can state that the space complexity can be reduced using FS-Tree.

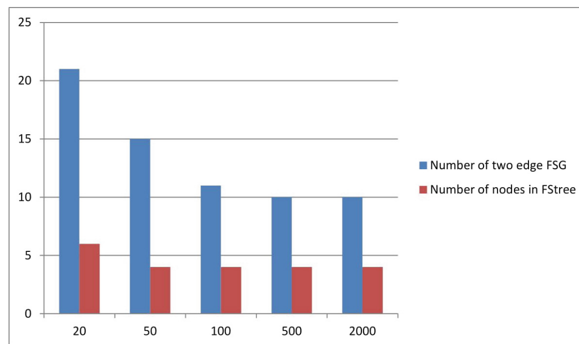


Fig. 7. Graphs with threshold 0.3

5 Conclusion

In this paper, we implemented frequent subgraph mining using FS-tree in a distributed environment using the Hadoop framework. From the experiments, we can conclude that with an increasing number of graphs in the data set, the proposed algorithm takes less time of execution than the traditional Apriori algorithm because the multiple data scans are eliminated. In the traditional algorithm, a large number of candidates are generated. With the FS-Tree approach, once the two-edge candidates are formed, each subgraph is added to the tree as a node. The subgraphs with higher number of edges can be formed using the FS-Tree. Hence the proposed approach generates comparatively less number of candidates than that of the traditional Apriori algorithm. This algorithm can be extended to other distributed frameworks like SPARK or STORM

References

1. Barabási, A., Oltvai, Z.: Network biology: understanding the cell's functional organization. Nat. Rev. Genet. **5**, 101–113 (2004)
2. Lacroix, V., Fernandes, C., Sagot, M.-F.: Motif search in graphs: pplication to metabolic networks. Trans. Comput. Biol. Bioinform. **3**, 360–368 (2006)
3. Borgelt, C., Berhold, M.R.: Mining molecular fragments: finding relevant substructures of molecules. In: Proceedings of International Conference on Data Mining 2002 (2002)
4. Handcock, M., Raftery, A., Tantrum, J.: Model-based clustering for social networks. J. R. Stat. Soc. Ser. (Stat. Soc.) **170**(2), 301–354 (2007)
5. Kuramochi,M., Karypis, G.: Frequent subgraph discovery. In: ICDM01. FSM (2001)
6. Cook, D.J., Holder, L.B.: Substructure discovery using minimum description length and background knowledge. J. Artif. Intell. Res. **1**, 231–255 (1994). 3rd ed
7. Praveena, A., Anitha, B., Rohini, R.: An efficient parallel iterative mapreduce based frequent subgraph mining algorithm. Middle-East J. Sci. Res. 24 (Tech. Algorithms Emerg. Technol.), 524–531 (2016)

8. Inokuchi, A., Washio, T., Motoda, H.: An apriori-based algorithm for mining frequent substructures from graph data. In: Zighed, D.A., Komorowski, J., Żytkow, J. (eds.) PKDD 2000. LNCS (LNAI), vol. 1910, pp. 13–23. Springer, Heidelberg (2000). https://doi.org/10.1007/3-540-45372-5_2
9. Vanetik, N., et al.: Computing frequent graph patterns from semi structured data. In: Proceedings 2002 IEEE International Conference on Data Mining, ICDM-2002 (2002)
10. Huan, J., Wang, W., Prins, J.: Efficient mining of frequent subgraph in the presence of isomorphism. UNC computer science Technique report TR03-021 (2003). FFSM
11. Nguyen, S.N., Orlowska, M.E., Li, X.: Graph mining based on a data partitioning. In: Nineteenth Australasian Database Conference (ADC 2008) (2008)
12. Bhuvaneswari, M., Rohini, R., Preetha, B.: A survey on privacy preserving public auditing for secure data storage. Int. J. Eng. Res. Technol. (2013)
13. Huan, J., Wang, W., Prins, J., Yang, J.: Spin: mining maximal frequent subgraphs from graph databases. In: Proceedings of the 10th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, pp. 581–586 (2004)
14. Hsieh, H.-P., Li, C.-T.: Mining temporal subgraph patterns in heterogeneous information networks. In: IEEE International Conference on Social Computing/IEEE International Conference on Privacy, Security, Risk and Trust (2010)
15. Thomas, S., Nair, J.J.: Improvised Apriori with frequent subgraph tree for extracting frequent subgraphs. J. Intell. Fuzzy Syst. **32**(4), 3209–3219 (2017)
16. Yan, X., Han, J.: gSpan: graph based sustructure pattern mining. In: Proceedings of 2nd IEEE International Conference on Data Mining, ICDM 2002 (2002)
17. Thomas, S., Nair, J.J.: A survey on extracting frequent subgraphs. In: International Conference on Advances in Computing, Communications and Informatics (ICACCI-2016) (2016)
18. Jeong, B.S., Choi, H.J., Hossain, M.A., Rashid, M.M., Karim, M.R.: A MapReduce framework for mining maximal contiguous frequent patterns in large DNA sequence datasets. IETE Tech. Rev. **29**, 162–168 (2012)
19. Hill, S., Srichandan, B., Sunderraman, R.: An iterative mapreduce approach to frequent subgraph mining in biological datasets. In: Proceedings of the ACM Conference on Bioinformatics, Computational Biology and Biomedicine (2012)
20. Wu, B., Bai, Y.L.: An efficient distributed subgraph mining algorithm in extreme large graphs. In: Wang, F.L., Deng, H., Gao, Y., Lei, J. (eds.) AICI 2010, Part I. LNCS (LNAI), vol. 6319, pp. 107–115. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-16530-6_14
21. Gayathri, S., Radhika, N.: Greedy hop algorithm for detecting shortest path in vehicular networks. Int. J. Control. Theory Appl. **9**, 1125–1133 (2016)
22. Liu, Y., Jiang, X., Chen, H., Ma, J., Zhang, X.: MapReduce-based pattern finding algorithm applied in motif detection for prescription compatibility network. In: Dou, Y., Gruber, R., Joller, J.M. (eds.) APPT 2009. LNCS, vol. 5737, pp. 341–355. Springer, Heidelberg (2009). https://doi.org/10.1007/978-3-642-03644-6_27
23. Di Fatta, G., Berthold, M.: Dynamic load balancing for the distributed mining of molecular structures. IEEE Trans. Parallel Distrib. Syst. **17**, 773–785 (2006)
24. Lin, J., Dyer, C.: Data-intensive text processing with MapReduce (2010)
25. Gayathri, R., Nair, J.J.: ex-FTCD: a novel mapreduce model for distributed multi source shortest path problem. J. Intell. Fuzzy Syst. **34**(3), 16431652 (2018)