2018 IEEE International Conference on Data Mining Workshops (ICDMW)

SparkFSM: A Highly Scalable Frequent Subgraph Mining Approach using Apache Spark

Bismita S. Jena, Cynthia Khan, Rajshekhar Sunderraman
Department of Computer Science, Georgia State University
[bsrichandan1, ckhan3]@student.gsu.edu, raj@gsu.edu

Abstract

Knowledge mining from graph data has attracted many researchers over the past several years. With the evolution of internet, computer technology, social networking sites, and web logs, graphs have become a very crucial dataset for mining and finding appropriate knowledge. Based on the application, graphs take different forms, such as airline flight information graph is mostly directed and smaller graphs, chemical compound structures are small and undirected, and social network graphs are very large single graphs based on the different types of associations between people. Earlier, our first attempt to use Hadoop (MapReduce model) to mine the directed frequent subgraphs from a large group of smaller graphs, which is famously known as transaction graphs, proved to be very scalable over the memory-based or database-oriented approaches. In this paper, we introduce SparkFSM, which not only handles undirected and directed graphs, but is also very scalable and efficient in handling isomorphism with the relatively new technology in industry (Spark/Scala). The combination of Spark with the functional style language Scala has established to be a de Facto standard while dealing with in-memory large data processing.

Index Terms — Frequent Subgraphs, Isomorphism, Spark, Scala

I. Introduction

Data mining has been one of the most studied area in computing. A very interesting study found from the article [41] shows that about ninety percent of all data existing right now in the world has been generated in the recent few years. Even though the web has been around for several years now, the credit for the massive amount of data generation may be given to the advancement of software technologies and social networks. Among all kinds of structured data, graph data is a major kind. Graph data falls under two categories, single large graphs and transaction graphs consisting of multiple small/medium sized graphs. Mining knowledge from each category differs significantly, mainly based on the requirement and type of graph data available.

In this paper, our major focus is to study transaction graphs which cover a very wide domain in industry such as airline flight information, chemical compound structure in drug analysis, fraud detection by studying the behavior of certain individuals etc. Frequent subgraph mining techniques on these graphs derive the knowledge out of the transaction graphs and provide further information for research studies. In our 2012 paper [9], for the first time we introduced Big Data Mining biological and synthetic graphs showed tremendous improvement over the in-memory [34, 37, 14, 15, 16, 17, 18, 19] and database [21, 22, 23] counterparts . Many authors have attempted to use the MapReduce model [3, 4, 5, 7, 8] and significantly benefited with its performance.

Authors in DIMSpan [1], introduced Apache Flink and Java language implementation on directed multigraphs and have achieved significant performance improvement over the MapReduce model [9]. MapReduce model had a few drawbacks like disk I/O, and especially due to the iterative style requirement for subgraph mining, it proved to be inefficient. Spark evolved based on the shortcomings of MapReduce model (though MR model is still one of the best models for huge batch processing). Over the past years, Spark [40] has become the major industry standard for its in-memory processing of big data. An additional benefit Spark brought for the users was Scala [39] as its base programming language. Scala, being a functional style language, offers much more flexibility over other verbose programming languages. In this paper, we introduce the next generation high performance, highly scalable, efficient way of handling isomorphism, and a model which can handle both undirected and directed structures. Both directed and undirected structures are important and based on the application, each has different semantic meaning. For example, a directed subgraph mining suits well for the airline graphs, but for PubChem data which contains the bioassay records for anti-cancer screen tests, the direction may not carry much meaning. The compounds contain atoms that are bound by the single or double bonds.

Our contributions in this paper are summarized as follows:

1. To our knowledge, this is the very first implementation on frequent subgraph mining on transaction graphs using Spark/Scala, handling both undirected and directed graphs.
2. Due to Scala's functional style, subgraph isomorphism capture became relatively easy by maintaining a unique code for the subgraphs.
3. Spark's in-memory processing capability helps in eliminating disk I/O to a larger extent by caching the RDDs.
4. We have conducted comparison with the DIMSpan [1] and our previous MapReduce Model [9], the results show tremendous performance improvements over both prior works.

The rest of the paper is organized in several sections. Section 2 discusses the background on FSM concepts and related work in the area of frequent subgraph mining. Section 3 explains Apache Spark, its major components and a small introduction to functional style language Scala. Section 4, explains frequent subgraph mining in undirected graphs. Section 5 focuses on the directed graphs leveraging the same utility. Section 6 presents experimental details conducted on very large graph datasets on the YARN cluster. Section 7 concludes the paper with some future prospects of graph data mining with faster and efficient computing techniques.

II. Frequent subgraph mining Background and related work

Frequent subgraph mining has a very long research history starting with in-memory single machine implementations. As the technology progressed, many researchers tried to explore the latest techniques. In this section, we will introduce a few definitions and key terminologies used in frequent subgraph mining (FSM) and later introduce the related research in the subgraph mining domain.

Labeled Graph: A labeled graph is represented by a tuple consisting of 4 elements, G = (V, E, L, l), where V is the set of Vertices, E is the set of directed or undirected edges where E is a subset of V x V. L is the set of labels, and l is the function assigning labels to the vertices and edges.

Subgraph: A graph G^1 whose vertices and edges are subsets of the Graph G, is called subgraph of G, and G is called the supergraph of G^1.

Frequency/Support: The number of occurrences of a subgraph in a graph is called the frequency. In case of transaction graphs, the frequency is counted by the number of occurrences of a subgraph in the set of graphs. If a subgraph appears more than once in a graph, that is captured as an isomorphic structure and is eliminated from the count. We will explain more on this in section 4 and 5 while describing the directed and undirected graphs.

Threshold Frequency: There is no set standard to measure threshold, but normally a marker is set as minimum threshold considering the type of graphs and expected outcome. If the threshold frequency goes below the minimum level, we have to decide whether that subgraph is meaningful for the application or to be rejected. We explain it with an example below. The subgraphs highlighted in Fig. 2, are substructures of G1 and G2 from Fig. 1. The subgraph appears in both G1 and G2, but not in G3, but since the minimum frequency of 50% is met, the structure is retained. Note here, the subgraph B-b-C-b-B is retained in undirected graphs, but not in directed graphs. We are providing a small comparison here to guide the readers through the process, in chemical compounds, ex. water (H_2O), two hydrogen atoms share one electron each with the oxygen atom forming the single covalent bond structure, and this is preserved in our experiment in undirected biological graphs. Similar is the case with NH_3, a compound consisting of Nitrogen and three Hydrogen atoms. For the directed graphs, B-b-C-b-B is an isomorphic structure and must be eliminated.

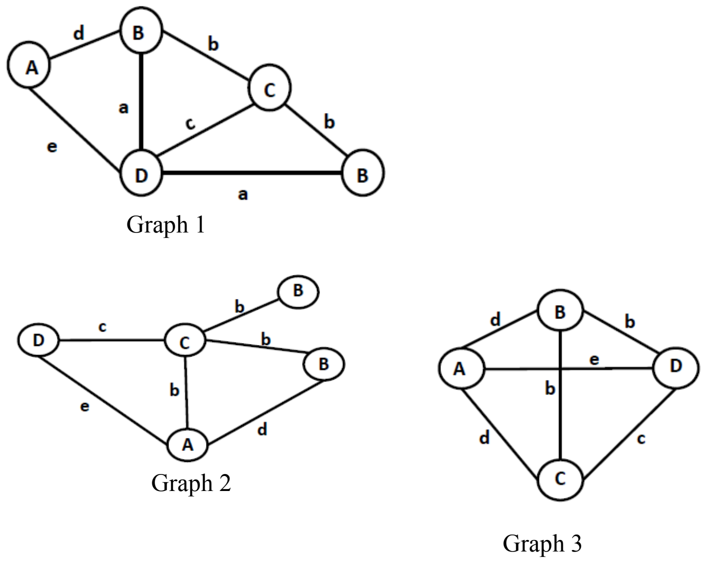


Figure 1: Undirected graphs

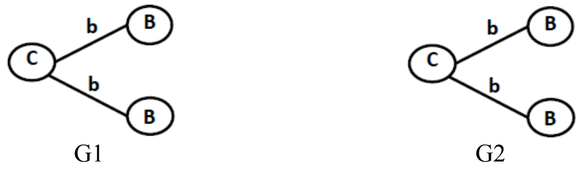


Figure 2: Structures retained

Subgraph Pruning: Pruning of structures happen when they don't meet the criteria for minimum threshold. Consider the 3-edge subgraphs derived from Fig. 1, shown below in Fig. 3. Both the subgraphs are pruned before we reach the four-edge structure. They don't satisfy the minimum threshold criteria and appear only in one graph.

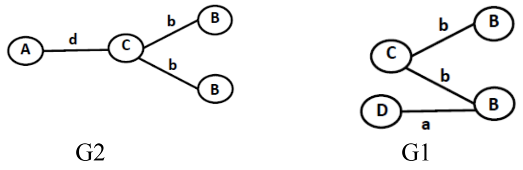


Figure 3: Pruned subgraphs

Isomorphic Subgraphs: In undirected graph setting, a subgraph is considered isomorphic if the instance appears multiple times in the same graph. The example given in Fig. 4, shows that the structure appears in graph 1 twice. Please note the difference here, even though we retained the subgraphs in Fig. 2, we are eliminating one of the subgraphs in Fig. 4. It is highly unlikely that the chemical compounds will need the exact duplicate to form a compound (this is one assumption we considered after observing the graphs' pattern). From the below subgraph, only one instance of B-b-C-c-D-a-A is retained. This is a little tricky here to determine which one to keep and which one to eliminate from next computation. If 1 is pruned, the entire next generation will be lost. Instead of pruning, which was a challenging decision, we decided to keep both the structures and their unique codes. While considering for the number of occurrences, we counted this as one. This way the next generation structures are not impacted.

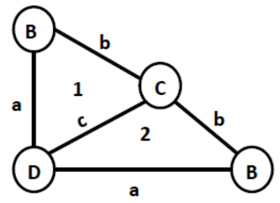


Figure 4: Isomorphic structures in Graph 1 of Fig. 1

Duplicate Elimination: While creating the next level subgraphs, the join criteria yields many duplicates, we use a unique coding to eliminate them. Only one instance is kept and moves to the next level. From Fig. 2, we can see that the edges can be joined in a few ways, so the unique code helps us to eliminate those duplicates. Ex., B-b-C-b-B can also appear as B-b-C-B-b-C, therefore maintaining the unique weight codes help in getting rid of the duplicates.

Related Research: In this section, we provide some of the related research work done leveraging Big Data technologies as well as the pioneer papers who initially introduced the concept. The research on graph mining has taken two major routes, mining single graphs and transaction graphs. We would like to introduce the readers to a few publications focusing on the transaction graphs. Since the inception of MapReduce [6], it has become an industry standard for large batch processing. We made the very first attempt [9] on transaction graphs and it was quite successful in performance improvement over the database versions [21, 22, 23] and the in-memory versions [34, 37, 14, 15, 16, 17, 18, 19]. Later, many authors tried and successfully achieved the desired result using MapReduce [3, 4, 5, 7, 8]. MapReduce had a few drawbacks for iterative style algorithm and frequent disk I/O created a bottleneck, adding more processing time. Spark became the new normal for large volume processing with its in-memory computation. The RDDs support caching mechanism which eliminated the disk I/O entirely. The lineage helps in keeping track of the previous iteration. In-memory cluster computing approach was adopted in [36]. DIMSpan [1] introduced Apache Flink implementation in Java on directed multi graphs.

Some papers tried to handle the subgraph mining using other distributed computing techniques [2, 24, 25, 28, 29, 31, 32]. In [27], authors have provided a survey on the FSM techniques. The paper [33] provides very good analysis and comparison on several pioneer algorithms. There are some very significant contributions [20, 11, 12, 13] towards this frequent subgraph mining domain. A few others focused on a different aspect of mining frequent subgraphs [10, 30, 35, 26].

III. Spark and Scala for High Performance Computing

Hadoop brought revolution to high performance computing with the introduction of MapReduce model and HDFS [38]. This worked as magic for big data batch processing jobs. Though the model is best for large batch processing, there are a few drawbacks when it comes to disk I/O, and iterative algorithms. MapReduce model didn't claim capability for real-time processing, but that's the need of the hour with the age of the internet and instant transactions. Spark was developed keeping in mind all of the above points. We discuss below a few key aspects of Spark that also proved to be very useful for our research due to its support for iterative style, in-memory data computation. Additionally, as Spark is built in Scala, the compatibility helped us in many ways while dealing with isomorphism.

RDD: Spark's core concept relies on the Resilient Distributed Datasets (RDDs). Spark RDD supports lazy evaluation, and maintains lineage. So if data is lost at one step, it can be retrieved from the previous RDD. The RDDs can be cached from a previous operation and can be reused in a subsequent step by applying the proper action on it.

Key Functions: The functions *map, flatmap, filter*, and *reduce* play a major role in the programming. Memory overhead related issues are nicely handled by the logical partition function ***repartition***. Sometimes, *coalesce* helps when the entire data need to be co-located to a few nodes only. *Cache* is another crucial functionality used to store intermediate results. This particularly helps before writing the RDD to disk for specific requirements.

Core Components: The major core functionalities Spark offers are: **Spark SQL**, uses Dataframe as data abstract which handles any SQL/HQL queries. **Spark Streaming**, the near real-time aspect of Spark, where data comes in mini-batches and a little latency is associated with this due to the small-batches. **GraphX**, a very powerful graph library best suited for the graph needs with its built-in capabilities. It is based on RDDs which are immutable and helps in processing due to the lineage tracking. **MLib**, the machine learning library that has successfully been applied on many machine learning algorithms.

Scala: This is purely a functional style programming language. There are several advantages of using Scala over Java for the Spark implementation purposes. Since Spark framework is built in Scala, many of its functionalities are very well compatible. Due to Scala's functional style, apparently, several lines of Java code can be represented in just a few lines in most cases. We could see the differences between our Java MapReduce [9] and

Scala Spark implementation 'SparkFSM'.

IV. FSM ON UNDIRECTED TRANSACTION GRAPHS

This section discusses on the algorithm and isomorphic structure identification in undirected graphs. As we noticed in Fig. 2, the structures are retained during single and double edge formations due to the nature of chemical compounds. As discussed in section II, under isomorphic structure determination, instead of pruning subgraph 2, we retained it, but while doing frequency evaluation, its unique code is dropped. The unique code is the key factor for the undirected graphs. Each node and edge has been assigned a specific weight for programming purposes. In the algorithm 1, step 5 explains on the core components of the undirected algorithm.

Algorithm 1: Undirected Graphs

```
Input: Graph (G1), Frequency (f)
Output: Qualified Subgraph Edge list

  Process: 1) G1.map => Load RDD1
           2) RDD1.filter(count >= f) => RDD_1
           3) RDD_1.map => SingleEdgeRDD
                (For each single edge in RDD_1,
              append reverse_single_edge to RDD_1)
           4) Assign unique code to each unique node label
           5) k EdgeRDD.join(SingleEdgeRDD) =>
               k+1_EdgeRDD
                 • Unidirection – join $RDD_A$.secondNode
                   === $RDD_B$.firstNode
                 • Filter ($RDD_A$.graphID ===
                   $RDD_B$.graphID)
                 • Generate unique code for each edge
                 • Filter isomorphic structures

           6) k+1_EdgeRDD.groupby(code).count()
           7) k+1_EdgeRDD.filter(count >= f) =>
               k+1EdgeEDD
           8) Repeat steps 5 – 7 for k+1EdgeRDD
           9) Repeat step 8 for 1 to n edge subgraphs
```

*RDD_A and RDD_B represent the alias for SingleEdgeRDD for initial round, and it represents the future n-egde RDDs as RDD_A and SingleEdgeRDD as RDD_B for subsequent steps.

As per our observation, the intermediate subgraphs meeting frequency threshold are very high. It is because of the isomorphic structure retention in the early stages like stage 1 and 2. We have explained the reason behind this in section II under threshold frequency. We are not showing all the intermediate frequent subgraphs due to the page limitations. There are many four-edge subgraphs, but we show only the important subgraphs that are unique for the undirected structures. From our observation, there are many subgraphs common across undirected and directed, but the directed graphs do not produce the four-edge substructure shown in Fig. 5. Fig. 5a shows the pattern that appears in graphs G1 and G2 as we retained the B-b-C-b-B until 2-edges.

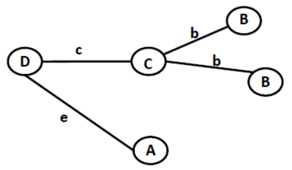


Figure 5a: G1, G2

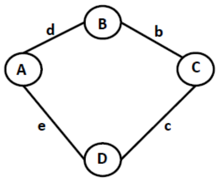


Figure 5b: G1, G2, and G3

Figure 5: Four-edge subgraphs

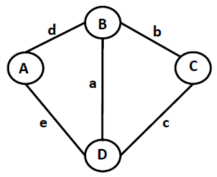


Figure 6a: G1, G3

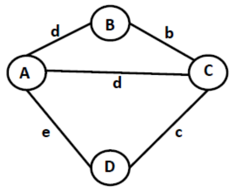
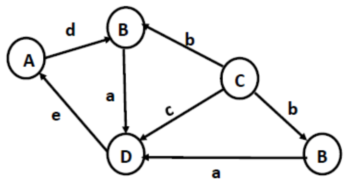


Figure 6b: G2, G3

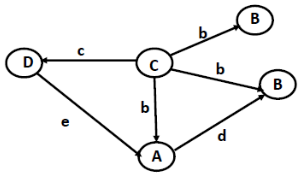
Figure 6: Five-edge subgraphs

V. FSM ON DIRECTED GRAPHS LEVERAGING THE UTILITY

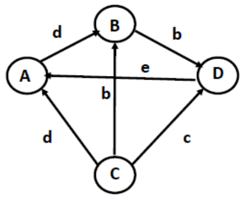
This section describes the algorithm for directed graphs. The isomorphic structure determination for directed graphs are a little straight forward as it is based on the direction. As per our observation, many subgraphs that appear in the undirected results, don't appear in the directed structure. Many structures are pruned at very early stage. Fig. 7 shows the directed graphs with the same three graphs used in undirected section, having direction attached to the nodes and edges.



Graph 1



Graph 2



Graph 3

Figure. 7: Directed graphs

One important thing to note here is, in the very early stage, single-edge filter prunes the C-b-B directed edge as part of isomorphic structure elimination. When we go to the next level subgraphs, the structure generated in Fig. 2 for undirected graphs does not exist. We observed that many substructures get pruned in the directed graphs. The graphs shown in Fig. 5 don't appear in directed graphs. Fig. 8 shows the four-edge subgraphs from the directed structures (these graphs also appear in undirected structures as well, but we have not shown it in Fig. 5).

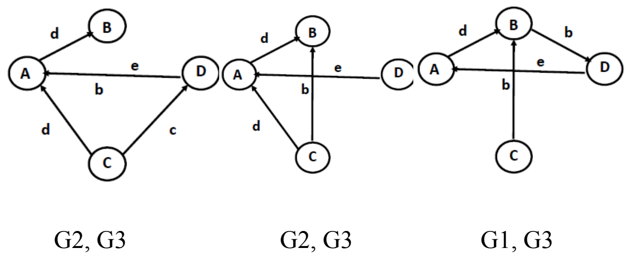


Figure 8: Four-edge directed subgraphs

We found a very interesting pattern from the three example graphs; both directed and undirected graphs yield the same five-edge subgraphs.

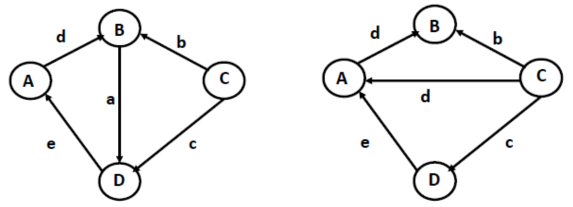


Figure 9a: G1, G3 Figure 9b: G2, G3

Figure 9: Five-edge directed subgraphs

Algorithm 2 describes the directed graph subgraph finding process. Step 4 provides the helper methods used in the process. The converge keyword represents the structures where both subgraphs meet at one node w.r.t. their direction pointing to the node. Diverge keyword is used to represent the subgraphs whose edges depart from a node and point in opposite directions. Unidirection is used for the subgraphs whose edges form a path.

Algorithm 2: Directed Graphs

```
Input: Graph (G1), Frequency (f)
Output: Qualified Subgraph Edge list

Process:
    1) G1.map => Load RDD1
    2) RDD1.filter(count >= f) => RDD_1
    3) RDD_1.filter(duplicate edges) => SingleEdgeRDD
    4) kEdgeRDD.join(SingleEdgeRDD) => k+1_EdgeRDD
        • Unidirection - join $RDD_A$.secondNode ===
          $RDD_B$.firstNode
        • Converge – join $RDD_A$.secondNode ===
          $RDD_B$.secondNode
        • Diverge – join $RDD_A$.firstNode ===
          $RDD_B$.firstNode
        • Filter($RDD_A$.graphID === $RDD_B$.graphID)
        • Eliminate isomorphic structures
        • Eliminate duplicates within same graphID
        • Assign Node labels according to the orientation of
          the join to maintain directional pattern
    5) k+1_EdgeRDD.groupBy(NodeLabel and edge
       pattern).count()
    6) k+1_EdgeRDD.filter(count >= f) => k+1EdgeRDD
    7) Repeat steps 4 – 7 for k+1EdgeRDD
    8) Repeat step 7 for 1 to n edge subgraphs
```

*RDD_A and RDD_B represent the alias for SingleEdgeRDD for initial round, and it represents the future n-Edge RDDs as RDD_A and SingleEdgeRDD as RDD_B for subsequent steps.

VI. EXPERIMENTAL DETAILS

All our tests were conducted on AWS EMR [43] with 1 master node and 2 slave nodes with m4 large configuration. We used Spark 2.3 for all our experiments. Both directed and undirected jobs ran in parallel on the same cluster and this was an evaluation criterion for the experiments.

Dataset Preparation: We used the chemical compound dataset retrieved from the PubChem website [42]. The dataset contains the bioassay records for anti-cancer screen tests with different cancer cell lines; they are categorized as active and inactive. Our initial round of experiments are conducted on the graphs as they appear on the site. Later, the data preparation was the most important criteria to test the scalability. A few authors concatenated the graphs from biological set to produce the larger sizes. After our analysis, we found that the graph sizes won't help much for proper evaluation if concatenated as is. The isomorphic subgraphs will be eliminated during the very first step as the graph numbers remain same across the larger set. We wrote a script to generate the larger graphs like OVCAR8I and OVCAR8HI. The script reads the last graph number and generates the next generation single edges and produce equal number of graphs. This way we can make sure that the evaluation is accurate for frequency determination. Also, as the biological graphs contain only vertices, edge numbers and labels, we have written a Perl script which helps with the preprocessing steps to create the single edges. After the initial load, the data load is not required for the several runs, so the time taken by the initial load is ignored (approx. 15-20 seconds).

Comparison: Exact comparison with DIMSpan[1] would not be appropriate as we have covered the undirected graphs in this research. The graphs generated for our evaluation are very complex due to the way they are created. It is not mere concatenation, rather every graph has millions of unique edges and the frequencies of new undirected graphs are massive. We did one level comparison with the biological directed graphs that shows somewhat comparable results. But we see improvements over DIMSpan. Since the original biological graph sizes are not very large, the time between DIMSpan and SparkFSM would not differ much. Matching the MRFSM [9] computation time with the SparkFSM would not be fair as the technologies are different and Spark is in-memory computation. The table 1 below provides the computation time in seconds, size of graphs, number of approximate edges present. As observed, the original graphs take a few seconds for frequencies 10%, 20%, 25% and 50%. The comparison is based on both the undirected and directed implementations.

Graph	Size	Graph Nos.	Edge	Time(s) undirected	Time(s) directed
MCF7A	1.3M	2293	18	5	30
MCF7HA	2.3M	2293	31	34	49
MCF7I	12M	25475	36	40	74
MCF7HI	20M	25475	59	91	77
MOLT4A	1.7M	3139	43	33	55
MOLT4HA	3M	3139	60	76	37
MOLT4I	17M	36624	36	84	63
MOLT4HI	29M	36624	59	74	75
NCIH23I	18M	38295	36	78	65
NCIH23HI	31M	38295	59	73	86
OVCAR8I	18M	38436	36	56	56
OVCAR8HI	20M	38436	48	45	33

Table 1: Performance analysis biological graphs (time in **seconds**, threshold frequency: **50%**)

MRFSM vs SparkFSM: We skipped the synthetic graphs' performance evaluation for this work as those graph generators do not produce proper transaction after a certain point. As we observed, beyond 1000K limit, the number of new edge and vertices combination was very limited. The minimum support level was not able to match beyond 7% which is not very practical in real life graph scenarios. Table 3 indicates our previous evaluation MRFSM [9] on the biological graphs with 2/4 node cluster using MapReduce model [9]. It is evident from table 1, with similar number of nodes (3 nodes) in SparkFSM, the time has reduced to 5 seconds compared to the 587 seconds in the MRFSM approach.

Dataset	Active: 2	Active: 4	Inactive: 2	Inactive: 4
MCF-7	833	587	1092	683
MOLT-4	922	556	1279	815
NCI-H23	815	516	1537	889
OVCAR-8	861	552	1257	844

Table 2: Comparison with MRFSM [9] and SparkFSM

DIMSpan vs SparkFSM: We used DIMSpan [1] as one of our evaluation standard, but DIMSpan has focused on the multi directed graphs as opposed to our SparkFSM which is more focused on undirected graphs. From their Data Sets section 5.2, we noticed that they are simply copying the graphs several times to create the larger volume. For this reason, the comparison between DIMSpam and SparkFSM will not provide any valuable insight.

The table 3 below shows our evaluation on undirected graphs. As described in the dataset preparation section, the graphs span from 50-100 edges. It became more complex after the graphs were duplicated with a new number assigned to each graph. We created graphs up to 4 million and captured the time in minutes. Graph sizes range from 124MB to 2.1GB.

Graph	Support	Graph Nos.	Time (min)
OVCAR8HI	75%	153,180	2.2
OVCAR8HI	90%	153,180	0.7
OVCAR8HI	75%	306,366	4.0
OVCAR8HI	90%	306,366	0.96
OVCAR8HI	75%	1,225,465	13
OVCAR8HI	90%	1,225,465	2
OVCAR8HI	75%	2,450,931	26
OVCAR8HI	90%	2,450,931	4

Table 3: Performance analysis on large undirected datasets (time in **minutes**)

VII. CONCLUSION

In this paper, our major effort was to mine the frequent subgraphs from undirected transaction graphs. Ever since the MapReduce based FSM [9] was published, many researchers have attempted various techniques on the directed transaction graphs. So far, we have not encountered any group actively working on the undirected transaction graphs using the Spark/Scala framework. Spark with its extraordinary in-memory distributed capability for Big Data has proved to be very efficient and highly scalable. Based on the comparison made on directed graphs, we could achieve much better performance improvement over our own version of MRFSM [9] as well as DIMSpan [1]. Since the original biological graphs are relatively small, and Spark being a heavy computing engine, the time taken is in seconds. The undirected FSM test we conducted on the large complex graphs shows the scalability of the application. It takes only a few minutes to process such complicated graphs with so high volume. The major improvement in the algorithm design and code development was comparatively easier due to Scala's functional style. Many issues like verbosity could easily be avoided in Scala unlike its Java counterpart. As part of the future work, we are focusing on the single large graph mining and developing algorithms that can reduce the isomorphic structures in the early stages and help with performance gain.

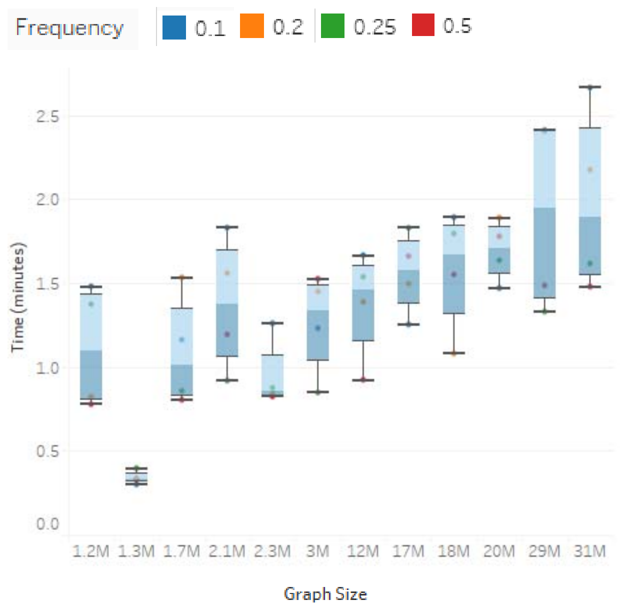


Figure 10: Box and Whisker plot showing time required to compute each undirected graph size at the varying frequencies

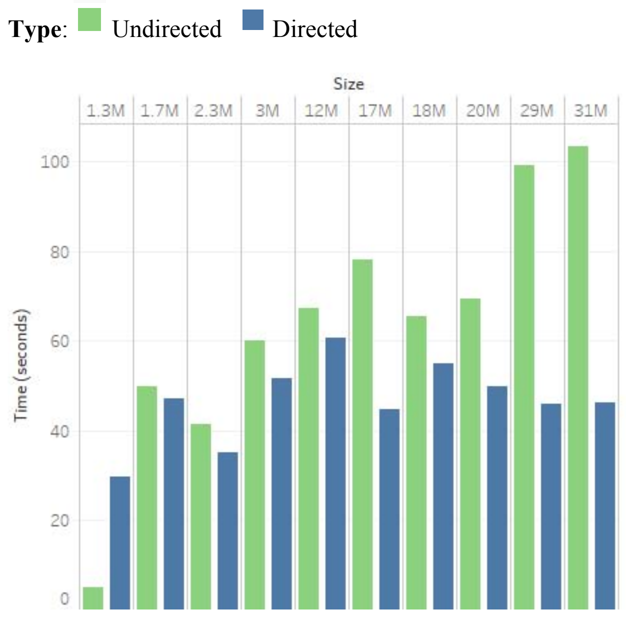


Figure 11: Performance comparison between Directed and Undirected on Biological graph dataset

References

[1] A. Petermann, M. Junghanns, and E. Rahm. DIMSpan - transactional frequent subgraph mining with distributed in-memory dataflow systems. BDCAT '17 Proceedings of the Fourth IEEE/ACM International Conference on Big Data Computing, Applications and Technologies, 2017.

[2] C. H. Teixeira et al. Arabesque: a system for distributed graph mining. In Proc. of the 25th Symposium on Operating Systems Principles, pages 425–440. ACM, 2015.

[3] M. A. Bhuiyan and M. Al Hasan. FSM-H: Frequent Subgraph Mining Algorithm in Hadoop. IEEE International Congress on Big Data, 2014.

[4] W. Lin, X. Xiao, and G. Ghinita. Large-scale frequent subgraph mining in mapreduce. In Inter-national Conference on Data Engineering (ICDE), pages 844–855. IEEE, 2014.

[5] W. Lu, G. Chen, A. Tung, and F. Zhao. Efficiently extracting frequent subgraphs using mapreduce. In IEEE Int. Conf. on Big Data, pages 639–647, 2013.

[6] J. Dean and S. Ghemawat. Mapreduce: simplified data processing on large clusters. Communications of the ACM, 51(1):107–113, January 2008.

[7] M. A. Bhuiyan and M. Al Hasan. An iterative mapreduce based frequent subgraph mining algorithm. Knowledge and Data Engineering, IEEE Transactions on, 27(3):608–620, 2015.

[8] S. Aridhi, L. D'Orazio, M. Maddouri, and E. Mephu. A novel mapreduce-based approach for distributed frequent subgraph mining. In Reconnais-sance de Formes et Intelligence Artificielle (RFIA), 2014.

[9] S. Hill, B. Srichandan, and R. Sunderraman. An iterative mapreduce approach to frequent subgraph mining in biological datasets. In Proc. ACM Conf. on Bioinformatics, Computational Biology and Biomedicine, pages 661–666, 2012.

[10] L. T. Thomas, S. R. Valluri, and K. Karlapalem. Margin: Maximal frequent subgraph mining. ACM Transactions on Knowledge Discovery from Data (TKDD), 4(3):10, 2010.

[11] S. Nijssen and J. N. Kok. The gaston tool for frequent subgraph mining. Electronic Notes in Theo-retical Computer Science, 127(1):77–87, 2005.

[12] S. Nijssen and J. N. Kok. The gaston tool for frequent subgraph mining. Electronic Notes in Theo-retical Computer Science, 127(1):77–87, 2005.

[13] A. Inokuchi, T.Washio, and H. Motoda. An apriori based algorithm for mining frequent substructures from graph data. In European Conference on Prin-ciples of Data Mining and Knowledge Discovery, pages 13–23. Springer, 2000.

[14] L. Dehaspe, H. Toivonen, and R. D. King. Finding frequent substructures in chemical compounds. In 4th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, pages 30–36. AAAI Press, August 1998.

[15] D. J. Cook and L. B. Holder. Graph-based data mining. IEEE Intelligent Systems, 15(2):32–41, May 2000.

[16] D. J. Cook, L. B. Holder, and S. Djoko. Knowledge discovery from structural data. Intelligent Information Systems, 5(3):229–248, November 1995.

[17] C. Borgelt and M. R. Berthold. Mining molecular fragments: Finding relevant substructures of molecules. In IEEE International Conference on Data Mining (ICDM), pages 51–58, 2002.

[18] X. Yan and J. Han. gspan: graph-based substructure pattern mining. In IEEE International Conference on Data Mining (ICDM) Proceedings, pages 721–724. IEEE, December 2002.

[19] L. B. Holder, D. J. Cook, and S. Djoko. Substucture discovery in the subdue system. In Workshop on Knowledge Discovery in Databases (KDD) Proceedings, pages 169–180. AAAI Workshop, July 1994.

[20] M. Kuramochi and G. Karypis. Frequent subgraph discovery. In IEEE International Conference on Data Mining (ICDM) Proceedings, pages 313–320. IEEE Computer Society, December 2001.

[21] S. Chakravarthy and S. Pradhan. Db-fsg: An sql-based approach for frequent subgraph mining. In 19th international conference on Database and Expert Systems Applications (DEXA) Proceedings, pages 684–692. Springer, September 2008.

[22] S. Padmanabhan and S. Chakravarthy. Knowledge discovery from structural data. In 11th International Conference on Data Warehousing and Knowledge Discovery (DaWaK), pages 325 – 338. Springer, August 2009.

[23] B. Srichandan and R. Sunderraman. Oo-fsg: An object-oriented approach to mine frequent subgraphs. In Australasian Data Mining Conference (AusDM) Proceedings, pages 221–228. CRPIT, December 2011.

[24] B. Wu and Y. Bai. An efficient distributed subgraph mining algorithm in extreme large graphs. In International conference on Artificial intelligence and computational intelligence: Part I (AICI) Proceedings, pages 107–115. Springer, October 2010.

[25] S. Ranu and A. K. Singh. Graphsig: A scalable approach to mining significant subgraphs in large graph databases. In IEEE Int. Conf. on Data Engineering (ICDE), pages 844–855. IEEE, 2009.

[26] B. Bringmann and S. Nijssen. What is frequent in a single graph? In PAKDD, pages 858–863. Springer, 2008.

[27] C. Jiang, F. Coenen, and M. Zito. A survey of frequent subgraph mining algorithms. The Knowledge Eng. Review, 28(01):75–105, 2013.

[28] R. Kessl, N. Talukder, P. Anchuri, and M. Zaki. Parallel graph mining with gpus. In BigMine, pages 1–16, 2014.

[29] R. R. McCune, T. Weninger, and G. Madey. Thinking like a vertex: a survey of vertex-centric frameworks for large-scale distributed graph processing. ACM Computing Surveys (CSUR), 48(2):25, 2015.

[30] A. Petermann, M. Junghanns, S. Kemper, K. G´omez, N. Teichmann, and E. Rahm. Graph mining for complex data analytics. In IEEE Int.Conf. on Data Mining Workshops (ICDMW), pages 1316–1319, 2016.

[31] A. Stratikopoulos et al. Hpc-gspan: An fpga-based parallel system for frequent subgraph mining. In IEEE Int. Conf. on Field Programmable Logic and Applications (FPL), pages 1–4, 2014.

[32] B. Vo, D. Nguyen, and T.-L. Nguyen. A parallel algorithm for frequent subgraph mining. In Advanced Computational Methods for Knowledge Engineering, pages 163–173. Springer, 2015.

[33] M. W¨orlein et al. A quantitative comparison of the subgraph miners mofa, gspan, ffsm, and gaston. In European Conference on Principles of Data Mining and Knowledge Discovery, pages 392–403. Springer, 2005.

[34] X. Yan and J. Han. gspan: Graph-based substructure pattern mining. In Technical Report UIUCDCS-R-2002.2296, 2002.

[35] X. Yan and J. Han. Closegraph: mining closed frequent graph patterns. In KDD, pages 286–295. ACM, 2003.

[36] M. Zaharia et al. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In Proc. of the 9th USENIX conference on Networked Systems Design and Implementation, pages 2–2, 2012.

[37] D. J. Cook and L. B. Holder. Substructure discovery using minimum description length and background knowledge. Artificial Intelligence Research, 1(1):231–255, August 1993.

[38] https://hadoop.apache.org/docs/current/hadoop-project-dist/hadoop-hdfs/HdfsDesign.html

[39] https://www.scala-lang.org/

[40] https://databricks.com/spark

[41] https://www.dezyre.com/article/apache-flink-vs-spark-will-one-overtake-the-other/282

[42] Graph Dataset: http://www.cs.ucsb.edu/*xyan/dataset.htm

[43] https://aws.amazon.com/emr/