An Iterative MapReduce Approach to Frequent Subgraph Mining in Biological Datasets

Steven Hill
Department of Computer Science
University of Maryland
College Park, MD 20742
smhill@umd.edu

Bismita Srichandan
Department of Computer Science
Georgia State University
Atlanta, GA 30302
bsrichandan1@student.gsu.edu

Rajshekhar Sunderraman
Department of Computer Science
Georgia State University
Atlanta, GA 30302
raj@cs.gsu.edu

ABSTRACT

Mining frequent subgraphs has attracted a great deal of attention in many areas, such as bioinformatics, web data mining and social networks. There are many promising main memory-based techniques available in this area, but they lack scalability as the main memory is a bottleneck. Taking the massive data into consideration, traditional database systems like relational databases and object databases fail miserably with respect to efficiency as frequent subgraph mining is computationally intensive. With the advent of the MapReduce framework by Google, a few researchers have applied the MapReduce model on a single graph for mining frequent substructures. In this paper, we propose to make use of the MapReduce programming model which achieves multifold scalability on a set of labeled graphs. We tested our method on both real and synthetic datasets. To the best of our knowledge, this is the first attempt to implement transaction graphs using the MapReduce model.

Categories and Subject Descriptors

J.3 [**Computer Applications**]: Life and Medical Sciences; D.1.3 [**Programming Techniques**]: Concurrent Programming

General Terms

Algorithms, Design, Performance

Keywords

Subgraphs, Frequent subgraph mining, MapReduce, Hadoop

1. INTRODUCTION

Mining frequent patterns has motivated many researchers since the very first research on finding the association rules in itemsets [1]. Graphs are prevalent in many domains such as bioinformatics, Semantic Web, chemoinformatics, and social networks. Frequent subgraph mining is a very well-studied area in graph mining research [9, 11] because of its wide range of applications in the above areas. Frequent patterns can help understand different functions and relations. For example, in a protein-protein interaction network (PPI), a frequent pattern could uncover unknown functions of a protein. Similarly, in a social network, a frequent pattern could show a friend clique. There are two different aspects of mining frequent subgraphs. The first category deals with a single large graph [6, 11, 14, 16]. The second category deals with a set of graphs [2, 12, 13, 21]. Counting the frequency in a transaction setting is a little different than the single graph setting. In a transaction setting, the frequency of a substructure is determined by the number of graph transactions containing the pattern, whereas in the single graph setting, the frequency of a substructure is determined by the number of times the pattern appears in the whole graph. There are a few implementations [17, 20] for finding the frequent patterns in a single graph using MapReduce framework [8], and one for finding subgraphs in transactions graphs in a grid environment [10]. All major frequent subgraph mining algorithms are based on the assumption that the graph data fits well in memory. Memory-based algorithms do fairly well on small datasets, but as the data size increases, memory becomes a bottleneck. To overcome this problem, a few database approaches [3, 4, 19] were proposed. The issue with database approaches is that as the dataset size grows, computation time rises drastically.

Our contribution: In this paper, we are proposing a novel method to extract the significant patterns from a set of labeled graphs using MapReduce. Our method works for both directed and undirected graphs. Given a graph dataset $D = \{G_1, \cdots, G_N\}$, the support of a subgraph $S(g)$ indicates the total number of times the subgraph g appears in the whole dataset D. A subgraph g is called frequent if $S(g)$ at least satisfies the user provided support. A detailed description is given later in the paper.

The rest of the paper is organized as follows. Related work is discussed in section 2. Section 3 gives an overview of the MapReduce model. In section 4, we discuss the frequent subgraph mining process using an open-source implementation of MapReduce known as Hadoop. Section 5 discusses our programming model in a more concise, formal setting, as well as provide an example. In section 6 we show the experimental details related to both real and synthetic datasets.

ACM-BCB'12, October 7-10, 2012 Orlando, FL, USA

2. RELATED WORKS

In this section, we discuss some of the previous research done in this area. We have categorized the work in three different ways in subsequent paragraphs: main memory based, traditional database based, and distributed.

We first discuss the main memory based algorithms. Subdue and its extensions [5, 6, 7, 11] were part of the earliest research by Cook et al. to discover the best compressing structure. Inokuchi et al. [13] proposed an Apriori based algorithm to discover all frequent subgraphs. Kuramochi et al. proposed FSG [15] which represents graphs as a sparse adjacency matrix and uses canonical labeling to determine subgraph isomorphism. In a subsequent publication [16], Kuramochi et al. proposed algorithms based on horizontal and vertical pattern discovery which operates on a single graph. Han et al. proposed the gSpan algorithm [21], which uses depth-first search and generates less candidate items than FSG. Huan et al. [12] proposed FFSM which shows better performance over gSpan. Jiang et al. [14] tried to mine globally frequent subgraphs on a single labeled graph.

The following are the traditional database based approaches for finding frequent subgraphs. DB-Subdue [3] and HDB-Subdue [18] are the SQL versions of Subdue that follow a minimum description length (MDL) principle for graph compression. DB-FSG [4] is a relational database approach applied on transaction graphs. OO-FSG [19] uses an object oriented database approach to find the frequent subgraphs in a set of labeled graphs.

There are a few distributed approaches which make use of the MapReduce framework for finding frequent patterns. Wu et al. [20] proposed a parallel subgraph mining algorithm in which the motif network diameter and number of vertices are taken as standard for motif matching. Liu et al. [17] proposed a method MRPF to find motifs in prescription compatibility networks. Fatta et al. [10] use a search tree partitioning strategy, along with dynamic load balancing.

3. MAPREDUCE

MapReduce, proposed by Google, is a distributed model for processing large-scale data. Users specify a map function and a reduce function. MapReduce takes in a list of key-value pairs, splits them among the possible map tasks, and then each map function produces any number of intermediate key-value pairs. Pairs with similar keys are gathered together at the reduce tasks, and then each reduce function performs computations before outputting values, which are either the final results, or possibly input for the next iteration. Ideally, MapReduce frameworks consist of several computers, usually referred to nodes, on the scale of tens to thousands. Processing occurs on data stored in the filesystem. Computation should be parallelized across the cluster, fault tolerant, and scheduled efficiently. We now go into some of the specifics of the map and reduce functions. *Figure 1* shows the MapReduce model.

3.1 Map Function

The mapper's job is to take in a key-value pair. This key-value pair often comes from a partition of data specified by the MapReduce architecture. After processing, the map function will emit another key-value pair. An added bonus comes in the form of an in-mapper combiner, which can do local computations to lessen the burden on the filesystem by acting as a mini-reducer. After all mappers have finished, all of the results are shuffled, sorted, and sent to the reducers.

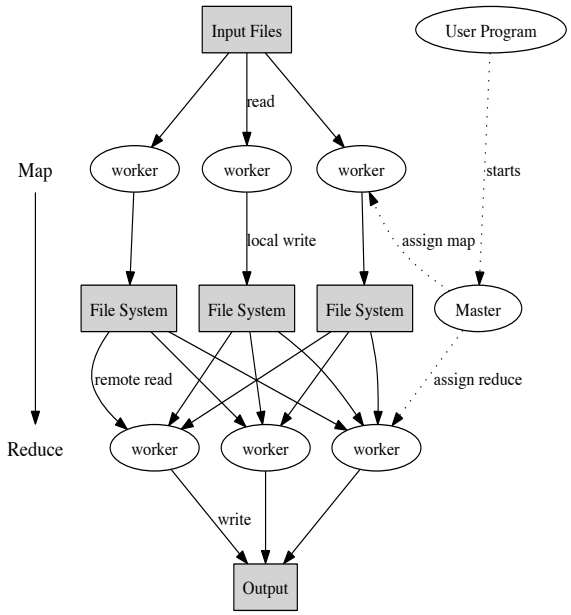

Figure 1: a MapReduce model

3.2 Reduce Function

The reducer takes in a list of values corresponding to a specific key. Here, the reduce function can perform many operations, such as aggregations and summations. Since all the values we need have been grouped, bulk computations on those values becomes trivial.

4. FREQUENT SUBGRAPH MINING USING MAPREDUCE

We propose using Apache Hadoop[1], an open source framework derived from Google's MapReduce and Google File System, to generate the frequent subgraphs. Hadoop has become a popular approach for distributed and parallel computing due its top-level status within Apache, as well as being widely supported by the community. Computations through Hadoop are highly scalable and reliable, making Hadoop a very powerful tool for processing large datasets, or in the context of this paper, large graph datasets. Using Hadoop iteratively, we can construct all isomorphic subgraphs that exceed a user defined support. We have two heterogeneous MapReduce jobs per iteration: one for gathering subgraphs for the construction of the next generation of subgraphs, and the other for counting these structures to remove irrelevant data. *Figure 2* shows this workflow. We will describe the process in more detail.

4.1 Definitions and FSG Determination

In this section, we will discuss the subgraph construction process, as well as the determination of frequent subgraphs. We start with a definition for the support in a transaction graphs setting.

[1]http://hadoop.apache.org/

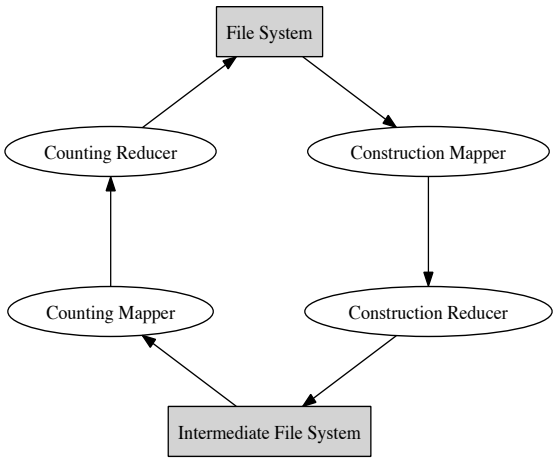

Figure 2: Frequent subraph mining using MapReduce

Definition 1. A *labeled graph* is represented by a 4-tuple, $G = (V, E, L, l)$, where

- V is a set of vertices
- $E \subseteq V \times V$ is a set of edges, they can directed or undirected
- L is a set of labels
- $l : V \cup E \rightarrow L$, l is a function assigning labels to the vertices and the edges

4.1.1 Subgraph Support

In *Figure 3*, three transaction graphs are shown. The numbers 1, 2, 3, 4, and 5 are numbers assigned for programming purposes. The labels A, B, C, D, E, F, G, H, and J are important to the algorithm. Let *nGraph* be the total number of graphs in the dataset and *nSubGraph* be the number of times a particular subgraph appears in the dataset. Then the support *Sup* of a particular subgraph is defined as :

$$Sup = \frac{nSubGraph}{nGraph}$$

The graph isomorphism problem needs to be tackled while counting the support of subgraphs in the dataset. Two instances are isomorphic if the vertex and edge labels are same and the directions are the same. In our method, graphs can be directed and undirected, and the node numbers help identify cases of repeated labels. For example, if we count the number of occurrences of the subgraph E-D-C in the three graphs, the count is 4, but we only take the unique counts, so it is actually 3. Graph 2 contains the subgraph E-D-C twice. Note that although we count it as one instance of the subgraph, we do not discard the other instance before pruning. Omitting that instance could be a potential problem when we construct the next generation of subgraphs. If the user support is taken as 2, then E-D-C/A-B-C are frequent subgraphs whereas E-R-J is not.

4.2 Subgraph Construction

This section elaborates on the process of subgraph construction. We explain in detail the process of map functions and reducer functions within each job of each iteration.

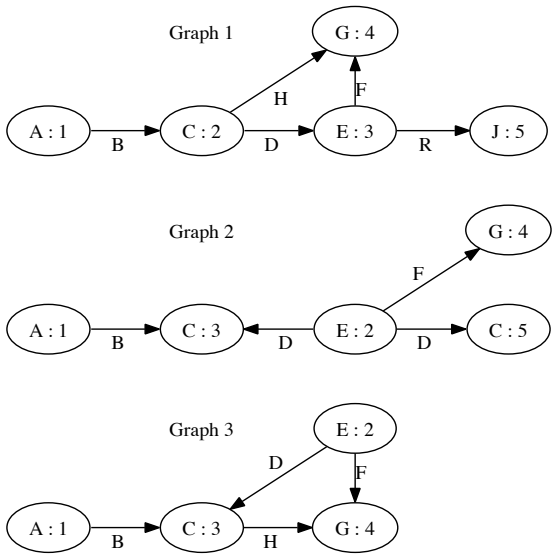

Figure 3: Example graphs in the dataset

4.2.1 Map Function for Gathering Subgraphs with Similar Graph ID

Hadoop sends single lines from the input file to the mappers, to which each applies a map function to those lines. This initial map function will have the responsibility of sending the subgraph encoded in the input string to the correct reducer using the graph id. For the first iteration, the encoded input string will represent a single edge of the graph. For all other iterations, we have an encoded input string representing a subgraph of size $k - 1$.

input key : offset of the input file for the string
input value : string representing a subgraph of size-$(k-1)$ and graph id
output key : graph id
output value : string representing the input subgraph

4.2.2 Reducer for Constructing Subgraphs

All of the subgraphs of size $k - 1$ with the same graph id are gathered for the reducer function. We note all of the single edges in these subgraphs and use that information to generate the next generation of possible subgraphs of size k. We encode this subgraph as a string just as was outputted from the previous map function. We keep all labels alphabetized and use special markers to designate differing nodes with the same labels. The results of this step are written out to the Hadoop File System.

input key : graph id
input values : list of subgraphs of size-$(k-1)$ encoded with graph id
output key : encoded subgraph of size-k and graph id
output value : *none*

4.2.3 Map Function for Gathering Subgraph Structures

Similar to the process involving the first map function, Hadoop sends lines of input to the mappers. This second map function will have the responsibility of outputting the

label-only subgraph encodings as a key and the node identification numbers and graph ids as values.

input key : offset of the input file for the string
input value : encoded string representing subgraph of size-k and graph id
output key : label-only string encoding subgraph
output value : corresponding node ids and graph id

4.2.4 Reducer for Determining Frequent Subgraphs

The last reducer function per iteration will gather on label-only subgraph structures. The main task is to count the unique instances of the specific subgraph, which is done by iterating through the input values, incrementing a count, and ignoring subgraphs with previously seen graph ids. The label markers are removed at this point. At the end, if the count agrees with the given user defined support, it is written out to the Hadoop File System for the next iteration, otherwise it is ignored, effectively pruned. The output of iteration k is all subgraphs of size k that meet the support.

input key : label-only string encoding subgraph of size-k
input values : list of corresponding node ids and graph ids
output key : the encoded subgraph and graph id
output value : *none*

5. DETAILS OF MAPREDUCE-FSG

MapReduce-FSG is an iterative algorithm that relies on two heterogeneous MapReduce jobs. The first job (denoted as A_k) constructs size-k subgraphs from size-$(k-1)$ subgraphs, while the second job (denoted as B_k) will check whether or not a subgraph meets the user defined support. The algorithm starts with single edges, and runs until there are no longer any frequent subgraphs constructed. *Algorithms 1 and 2* highlight the tasks of A_k.

Algorithm 1 Map A_k

```
Input: (offset, subgraph)
parse subgraph for graph id
EMIT: (graph id, subgraph)
```

Algorithm 2 Reduce A_k

```
Input: (graph id, subgraphs s_1, s_2, s_3, ···)
Edges ← φ
newSubgraphs ← φ
for all s ∈ subgraphs do
    Retrieve all edges from s and add to Edges
end for
for all s ∈ subgraphs do
    Construct k-sized subgraphs from (k − 1)-sized s using
    edges from Edges that are eligible and add the new
    subgraph to newSubgraphs
end for
for all s ∈ newSubgraphs do
    EMIT: (encoding for subgraph, empty text)
end for
```

Algorithms 3 and 4 outline the important steps of B_k. These algorithms are essential for pruning unnecessary subgraphs for the next iteration. Without them, we would quickly weigh down the disk and network.

Algorithm 3 Map B_k

```
Input: (offset, encoded subgraph)
parse encoded subgraph for label-only subgraph
EMIT: (label-only subgraph, subgraph)
```

Algorithm 4 Reduce B_k

```
Input: (label-only subgraph, subgraphs s_1, s_2, s_3, ···)
GraphIDs ← φ
count ← 0
for all s ∈ subgraphs do
    if s.graphid ∉ GraphIDs then
        count ← count + 1
        GraphIDs ← GraphIDs ∪ s.graphid
    end if
end for
if count ≥ user support then
    for all s ∈ subgraphs do
        EMIT: (subgraph, empty text)
    end for
end if
```

5.1 Canonical Ordering of Elements

As we are using Hadoop's Text to encapsulate a string object representing a subgraph, it is important to be able to differentiate between repetitive labels. We sort the outgoing nodes lexicographically based on label, and then use the unique id numbers if ambiguity still remains. The sorting helps us with key matching, which is essential for our MapReduce approach. Reducer A will dynamically mark all node labels in the encoding Text so that we may distinguish between identical labels that belong to different nodes during Reducer B.

5.2 Illustrative Example

Here we illustrate our implementation of the algorithm by showing outputs generated in each step. We use the three sample graphs from *Figure 3*. User-support is taken as 2. The output generated by the A_i and B_i steps are coded as three-part strings separated by "_". The first represents the graph id, the second represents a label-only subgraph, such as (A:B-C) standing for "node A has an edge B to node C", and the third part represents node id numbers, such as (1:3) standing for "node with id 1 has an edge to node with id 3."

5.2.1 Step B_1

As we are using single edges as the initial input, we do not need an A_1, and can proceed to B_1. At this step, all single edges that do not meet our support of 2 have been removed. We show the output below,represented in *Figure 4*.

2_(A:B-C)_(1:3)
3_(A:B-C)_(1:3)
1_(A:B-C)_(1:2)
3_(C:H-G)_(3:4)
1_(C:H-G)_(2:4)
3_(E:D-C)_(2:3)
2_(E:D-C)_(2:3)
2_(E:D-C)_(2:5)
2_(E:F-G)_(2:4)
3_(E:F-G)_(2:4)
1_(E:F-G)_(3:4)

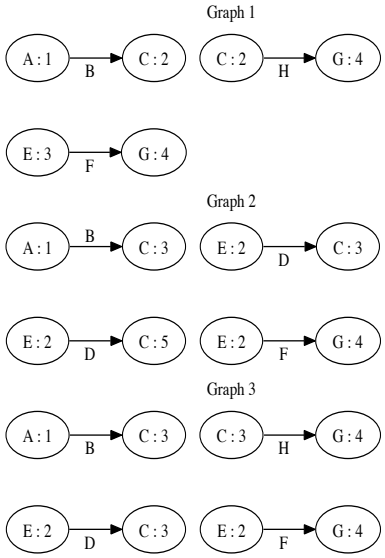

Figure 4: Single edge subgraphs that meet support

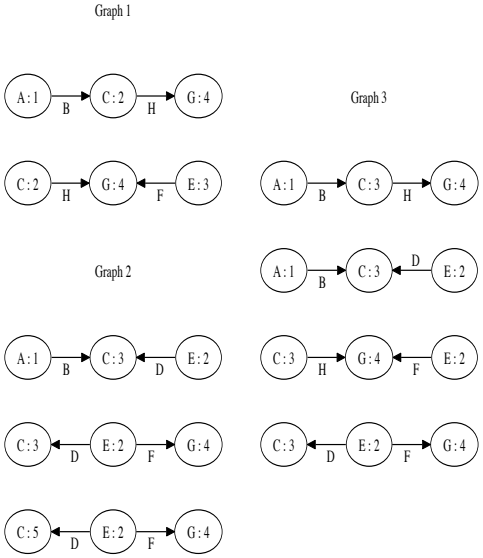

Figure 5: Double edge subgraphs that meet support

5.2.2 *Step* A_2

The worker for A_2 will read input from the filesystem corresponding to the job of B_1 and construct all subgraphs of size 2 that do not necessarily agree with our support. The output:

1_(Aˆ1:B-Cˆ1)(Cˆ1:H-Gˆ1)_(1:2)(2:4)
1_(Cˆ1:H-Gˆ1)(Eˆ1:F-Gˆ1)_(2:4)(3:4)
2_(Aˆ1:B-Cˆ1)(Eˆ1:D-Cˆ1)_(1:3)(2:3)
2_(Eˆ1:D-Cˆ1,D-Cˆ2)_(2:3,5)
2_(Eˆ1:D-Cˆ1,F-Gˆ1)_(2:3,4)
2_(Eˆ1:D-Cˆ1,F-Gˆ1)_(2:5,4)
3_(Aˆ1:B-Cˆ1)(Cˆ1:H-Gˆ1)_(1:3)(3:4)
3_(Aˆ1:B-Cˆ1)(Eˆ1:D-Cˆ1)_(1:3)(2:3)
3_(Cˆ1:H-Gˆ1)(Eˆ1:D-Cˆ1)_(3:4)(2:3)
3_(Cˆ1:H-Gˆ1)(Eˆ1:F-Gˆ1)_(3:4)(2:4)
3_(Eˆ1:D-Cˆ1,F-Gˆ1)_(2:3,4)

Notice the "ˆ" used above. These are markers for the correct placement of labels. Dealing with repetitive labels and subgraphs, we have to deal with a lot of ambiguity. In graph 2, we have 2_(Eˆ1:D-Cˆ1,D-Cˆ2)_(2:3,5). Without the marker, we would have (E:D-C,D-C). To make sure we are following the substructure through multiple graph ids, we need those markers to remove confusion.

5.2.3 *Step* B_2

The worker for B_2 will read input from the filesystem corresponding to the job of A_2. This input is an unfiltered group of size-2 subgraphs, and B_2 will filter out results that do not agree with the user-support, as well as remove special markers. The output is a pruned group of subgraphs of size-2 which meets the support. As a result, we obtain:

1_(A:B-C)(C:H-G)_(1:2)(2:4)
3_(A:B-C)(C:H-G)_(1:3)(3:4)
2_(A:B-C)(E:D-C)_(1:3)(2:3)
3_(A:B-C)(E:D-C)_(1:3)(2:3)
1_(C:H-G)(E:F-G)_(2:4)(3:4)
3_(C:H-G)(E:F-G)_(3:4)(2:4)
2_(E:D-C,F-G)_(2:3,4)
2_(E:D-C,F-G)_(2:5,4)
3_(E:D-C,F-G)_(2:3,4)

This output corresponds to *Figure 5*

5.2.4 *Step* A_3

Similar to A_2, we read from the results of the preceding B step.

1_(Aˆ1:B-Cˆ1)(Cˆ1:H-Gˆ1)(Eˆ1:F-Gˆ1)_(1:2)(2:4)(3:4)
2_(Aˆ1:B-Cˆ1)(Eˆ1:D-Cˆ1,D-Cˆ2)_(1:3)(2:3,5)
2_(Aˆ1:B-Cˆ1)(Eˆ1:D-Cˆ1,F-Gˆ1)_(1:3)(2:3,4)
2_(Eˆ1:D-Cˆ1,D-Cˆ2,F-Gˆ1)_(2:3,5,4)
3_(Aˆ1:B-Cˆ1)(Cˆ1:H-Gˆ1)(Eˆ1:D-Cˆ1)_(1:3)(3:4)(2:3)
3_(Aˆ1:B-Cˆ1)(Cˆ1:H-Gˆ1)(Eˆ1:F-Gˆ1)_(1:3)(3:4)(2:4)
3_(Aˆ1:B-Cˆ1)(Eˆ1:D-Cˆ1,F-Gˆ1)_(1:3)(2:3,4)
3_(Cˆ1:H-Gˆ1)(Eˆ1:D-Cˆ1,F-Gˆ1)_(3:4)(2:3,4)

5.2.5 *Step* B_3

We arrive at the final result (represented in *Figure 6*) given our example:

1_(A:B-C)(C:H-G)(E:F-G)_(1:2)(2:4)(3:4)
3_(A:B-C)(C:H-G)(E:F-G)_(1:3)(3:4)(2:4)
2_(A:B-C)(E:D-C,F-G)_(1:3)(2:3,4)
3_(A:B-C)(E:D-C,F-G)_(1:3)(2:3,4)

6. EXPERIMENTAL DETAILS

The experiments were conducted on 4 Linux machines, each with 16 GB of memory and 2-4 quad core processors. The MapReduce-FSG algorithm was coded in Java as to work with Hadoop. Although not mentioned, results on both the synthetic and real datasets performed substantially better than the database methods [4, 19], which were sup-

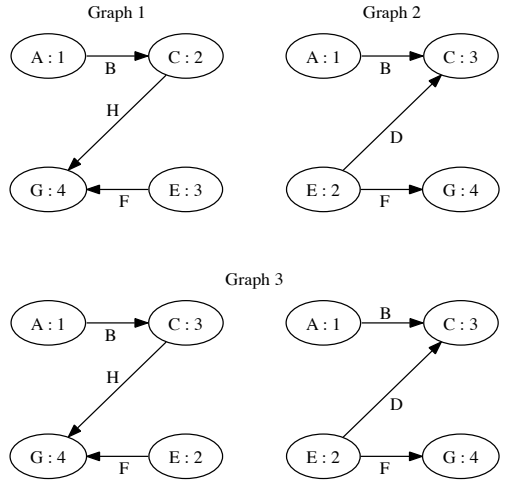

Figure 6: Triple edge subgraphs that meet support

posedly the fastest methods for general purpose subgraph mining prior to this paper.

6.1 Synthetic datasets

The experimental results are shown in *Table 1* as well as in graphical format. The graphs in *Figures 7, 8, and 9* present a sense of scalability of our method by comparing 2 and 4 sized clusters with varying supports. Even with our bantom setup, we managed to make substantial gains.

For our method, we performed experiments on datasets ranging from 100,000 to 1,000,000 transaction graphs. Each graph contains 30-50 edges and 30-50 vertices. The synthetic datasets were generated using a graph generator kindly provided by the authors[2]. Tests were conducted with varying minimum support values 1%, 4%, and 7%. The maximum-substructures is taken as four, and so we only iterate four times. Jumping from 2 nodes to 4 scaled very well for both increases in datasets, as well as number of nodes. Although we only have access to a modest cluster, it is easy to see the potential gains from large-scale clusters.

Table 1: Performance of MapReduce-FSG (in seconds)

Dataset size	Support	2 Nodes	4 Nodes
100K	1%	2471	1332
100K	4%	1718	1002
100K	7%	1203	721
400K	1%	2704	1559
400K	4%	2134	1217
400K	7%	1778	1018
1000K	1%	3702	2021
1000K	4%	3282	1809
1000K	7%	2786	1559

[2]http://www.cse.ust.hk/graphgen/

Table 2: Performance on biological datasets using a support of 50% and clusters of size 2 and 4 (in seconds)

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
P388	743	483	976	683
PC-3	857	546	1150	752
SF-295	936	528	1217	817
SN12C	813	502	1474	883
SW-620	959	568	1454	898
UACC257	836	536	1333	883
Yeast	710	607	1282	812

Dataset	Size	Tumor description
MCF-7	27770	Breast
MOLT-4	39765	Leukemia
NCI-H23	40353	Non-Small Cell Lung
OVCAR-8	40516	Ovarian
P388	41472	Leukemia
PC-3	27509	Prostate
SF-295	40271	Central Nerv Sys
SN12C	40532	Colon
SW-620	40004	Renal
UACC257	39988	Melanoma
Yeast	79601	Yeast anticancer

6.2 Biological datasets

The real datasets are taken from an online source[3], which contains data extracted from the PubChem website[4]. PubChem provides information on the biological activities of small molecules. The dataset is comprised of bioassay records for anti-cancer screen tests with different cancer cell lines, the outcome of which are either active or inactive. We first ran our method on a cluster of size 2, and then again on a cluster of size 4. Results are shown in *Table 2*, and graphically in *Figure 10. Figures 11 and 12* present speedup.

7. CONCLUSION AND FUTURE WORK

We have proposed a new approach for mining frequent subgraphs through MapReduce. Using Hadoop, we have obtained a scalable and efficient method. Even with a modest cluster, we have managed to outperform all previous methods. Not only does our method perform better than previous methods, but it can handle undirected graphs in addition to directed. Although we show a novel approach to subgraph mining that has promising results, due to limited facilities, further work will be needed to fully understand the scalability of our method.

8. ACKNOWLEDGMENTS

This research has been supported by the NSF REU grant # 1156733 and The Molecular Basis of Disease Program, Georgia State University, USA.

[3]http://www.cs.ucsb.edu/∼xyan/dataset.htm
[4]http://pubchem.ncbi.nlm.nih.gov

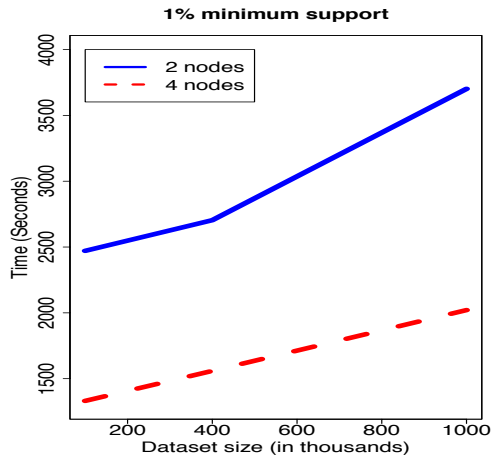

Figure 7: Comparison with 1% support

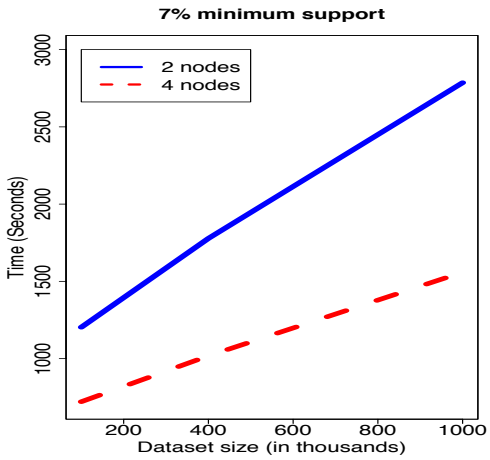

Figure 9: Comparison with 7% support

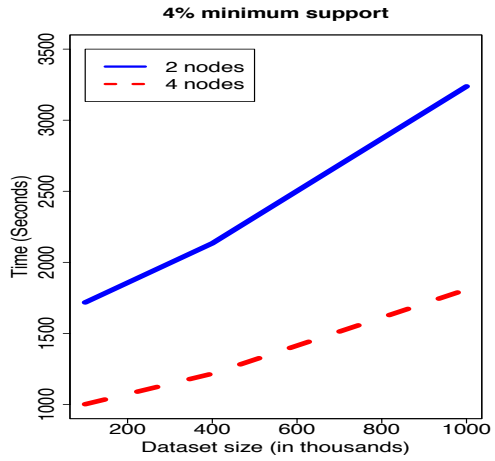

Figure 8: Comparison with 4% support

9. REFERENCES

[1] R. Agrawal and R. Srikant. Fast algorithms for mining association rules. In *20th International Conference on Very Large Data Bases (VLDB)*, pages 487–499. Morgan Kaufmann, September 1994.

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

[3] S. Chakravarthy, R. Beera, and R. Balachandran. Db-subdue: Database approach to graph mining. In *8th Pacific-Asia Conference in Knowledge Discovery and Data Mining (PAKDD) Proceedings*, pages 341–350. Springer, May 2004.

[4] 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.

[5] 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.

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

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

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

[9] 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.

[10] G. D. Fatta and M. Berthold. Dynamic load balancing for the distributed mining of molecular structures. *IEEE Transactions on Parallel and Distributed System*, 17(8):773–785, September 2006.

[11] 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.

[12] J. Huan, W. Wang, and J. Prins. Efficient mining of frequent subgraphs in the presence of isomorphism. In *Third IEEE International Conference on Data Mining (ICDM) Proceedings*, pages 549–552. IEEE, November 2003.

[13] A. Inokuchi, T. Washio, and H. Motoda. An apriori-based algorithm for mining frequent substructures from graph data. In *4th European Conference on Principles of Data Mining and Knowledge Discovery (PKDD) Proceedings*, pages 13–23. Springer, September 2000.

[14] X. Jiang, H. Xiong, C. Wang, and A.-H. Tan. Mining globally distributed frequent subgraphs in a single labeled graph. *Data & Knowledge Engineering*, 68(10):1034–1058, October 2009.

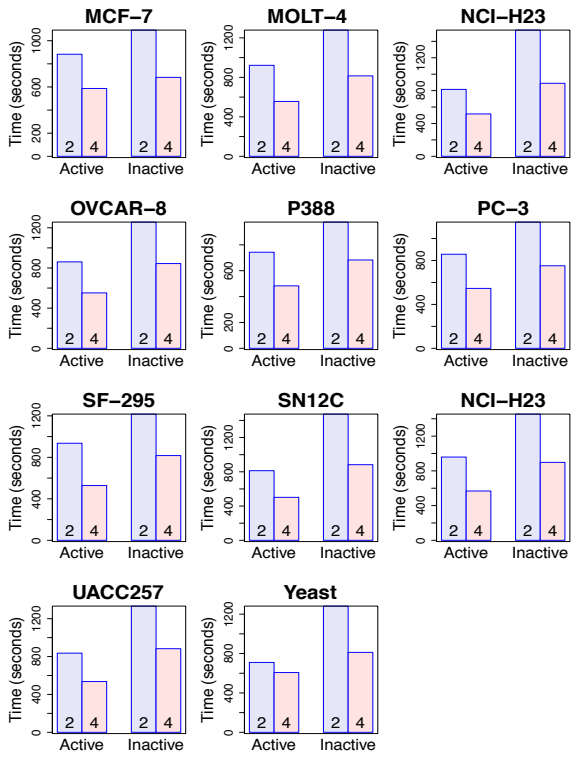

Figure 10: Results of biological datasets. Each graph shows the runtimes for active and inactive outcomes on both clusters of size 2 and 4.

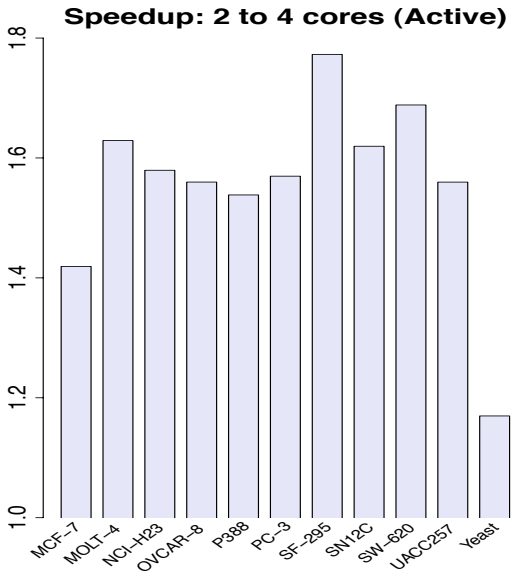

Figure 11: Speedup for active dataset

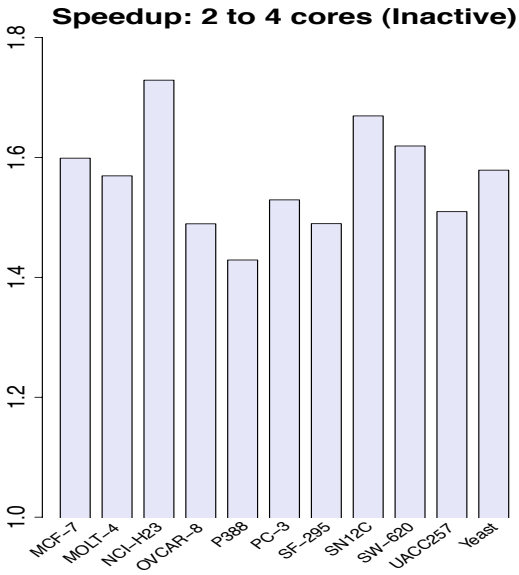

Figure 12: Speedup for inactive dataset

[15] 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.

[16] M. Kuramochi and G. Karypis. Finding frequent patterns in a large sparse graph. *Data Mining and Knowledge Discovery*, 11(3):795–825, November 2005.

[17] Y. Liu, X. Jiang, H. Chen, J. Ma, and X. Zhang. Mapreduce-based pattern finding algorithm applied in motif detection for prescription compatibility network. In *8th International Symposium on Advanced Parallel Processing Technologies (APPT)*, pages 341 – 355. Springer, August 2009.

[18] 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.

[19] 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.

[20] 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.

[21] 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.