Graph Based Anomaly Detection using MapReduce on Network Records

Brandon Joyce
Dept. of Mathematics and Computer Science
University of North Carolina at Greensboro
Greensboro, NC 27412, USA
bwjoyce@uncg.edu

Enyue Lu
Dept. of Mathematics and Computer Science
Salisbury University
Salisbury, MD 21801, MD
ealu@salisbury.edu

Matthias K. Gobbert
Dept. of Mathematics and Statistics
University of Maryland, Baltimore County
Baltimore, MD, USA
gobbert@umbc.edu

***Abstract*—It has become increasingly complex to detect intrusion through network records due to large volumes of network traffic data. In order to detect anomalies and do it quickly, parallel strategies such as MapReduce can be used to improve the performance. In this project, network data was modeled as a graph in order to apply a graph compression algorithm. A star graph configuration model allows for the use of a pattern recognition algorithm to identify network intrusions. We have developed a MapReduce algorithm that uses a SUBDUE graph compression method. We tested our algorithm on the 1999 KDD Cup network intrusion dataset. It has shown that our MapReduce algorithm is capable of processing large amounts of data with high sensitivity (99.21%) and accuracy (94.65%).**

***Index Terms*—Graph theory, Intrusion detection, Pattern recognition**

I. Introduction

Network Intrusion Detection Systems (NIDS) have been used in order to monitor network traffic. There are two types of strategies that NIDS use, namely misuse and anomaly detection. Misuse detection relies on known intrusive patterns, whereas anomaly detection tries to detect any communication that is anomalous. Thus, new types of attacks could be identified with anomaly detection and then the newly identified attack pattern could be used in misuse detection [1].

The 1999 KDD Cup Dataset is a labeled dataset that has been used by many researchers to train and test their network intrusion methods. It was created by MIT Lincoln Labs as a way to simulate a U.S. Air Force local-area network [2]. The training dataset is compressed of nearly five million network records, and the testing dataset has over three hundred thousand records. Each record has forty one attributes (such as connection type, duration, etc.) and a label that marks the record as either normal or as a particular type of intrusion/attack (portsweep, guess password, satan, etc.).

In [3], a graph based approach was used for anomaly detection. They tested their algorithm on the 1999 KDD Cup network intrusion dataset. Their algorithm modeled network records as subgraphs. They used a SUBDUE algorithm to perform graph compression in order to identify anomalous records. For each record, an anomaly score was calculated. The anomaly score for the KDD records was used to determine the accuracy of their algorithm. They showed that for small KDD record samples, many anomalous records would have a higher anomaly score.

Our goal is to identify network anomalies given a network record log file, such as the 1999 KDD Cup Dataset. To accomplish this, we designed a MapReduce algorithm based on the graph compression model presented in [3]. Our MapReduce algorithm examines each record in order to find the most frequent pair of attributes that occurred in all of the records. We then compress all records that contain this attribute-pair or pattern. This "find and compress" process is repeated iteratively. To determine the accuracy of our method, we will calculate anomaly scores for each record as in [3].

We will use the next two sections to explain a Confusion Matrix, special types of statistical percentages, MapReduce, SUBDUE Graph Compression, and the Star Graph Model. Section 4 explains our MapReduce algorithm. Section 5 explains our analysis on the KDD dataset using the Star-Graph Model. Section 6 examines our results. The last section is the conclusion of our work.

II. Background

A. Confusion Matrix

A Confusion Matrix is created by calculating the True Positives, True Negatives, False Positives, and False Negatives of a test. These values are calculated (as shown in [4]) as follows:

1) *True Positives (TP)*: is the percentage of attack patterns which are correctly identified as the intrusion
2) *True Negatives (TN)*: is the percentage of normal patterns which are correctly identified as the normal activities
3) *False Positives (FP)*: is the percentage of normal patterns which are incorrectly identified as the intrusion
4) *False Negatives (FN)*: is the percentage of attack patterns which are incorrectly identified as the normal activities

From these four values, we can derive other statistical values [4]:

1) *True positive rate (TPR) or sensitivity*: is the proportion of attack patterns which are correctly identified as the intrusion to all the exactly attack patterns:

$$\mathit{TPR(Sensitivity)} = \frac{TP}{TP + FN}$$

2) *False positive rate (FPR)*: is the proportion of normal patterns which are incorrectly identified as the intrusion to all the exactly normal patterns:

$$FPR = \frac{FP}{FP + TN}$$

3) *True negative rate (TNR) or specificity*: is the proportion of normal patterns which are correctly identified as the normal activities to all the exactly normal patterns:

$$TNR(Specificity) = \frac{TN}{TN + FP}$$

4) *False negative rate (FNR)*: is the proportion of attack patterns which are incorrectly identified as the normal activities to all the exactly attack patterns:

$$FNR = \frac{FN}{FN + TP}$$

5) *Accuracy*: is the proportion of all the correctly identified patterns to all the correctly and incorrectly identified patterns:

$$Accuracy = \frac{TP + TN}{TP + TN + FP + FN}$$

6) *Positive predictive value (PPV or Detection rate)*: is the proportion of attack patterns which are correctly identified as the intrusion to all the predicted attack patterns:

$$PPV = \frac{TP}{TP + FP}$$

7) In [5], one other useful value is also used: *Negative Predictive Value (NPV)*: is the proportion of normal patterns which are correctly identified as normal to all the predicted normal patterns:

$$NPV = \frac{TN}{TN + FN}$$

The PPV is used to evaluate the chance that a record that is flagged as intrusive is actually intrusive, whereas the NPV is used to determine the chance that a record that is flagged as normal is actually normal. Also, it should be noted that most anomaly detection methods usually yield a high number of false positives, since the definition of "normal" is usually dynamic [1]. We used these statistical measurements to evaluate our anomaly detection results.

B. MapReduce

Hadoop MapReduce is a popular open source parallel paradigm maintained by Apache. This paradigm relies on two methods: a "Map" and a "Reduce." Both methods use key-value pairs. During the Map phase, the input key-value pair is mapped to 0, 1, or many output key-value pair(s). The output for the Map method is used as the input for Reduce method. Importantly, every unique key is mapped to the same Reduce bin and the associated values are combined into a list that can be analyzed. Furthermore, the Hadoop Distributed File System (HDFS) allows for petabytes of data to be analyzed by splitting larger files into several smaller files [6]. An example of how MapReduce works is shown in Figure 1 and an example of how to use MapReduce to count letters in shown in Figure 2.

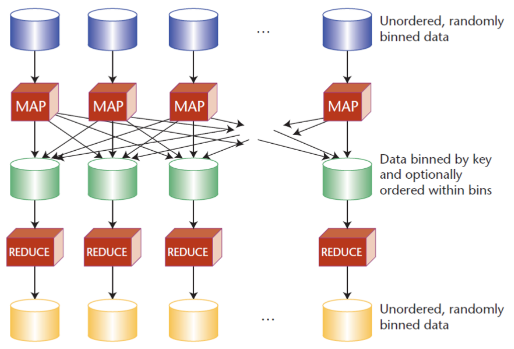

Unordered, randomly binned data
MAP
MAP
MAP
MAP
Data binned by key and optionally ordered within bins
REDUCE
REDUCE
REDUCE
REDUCE
Unordered, randomly binned data

Fig. 1: MapReduce Paradigm

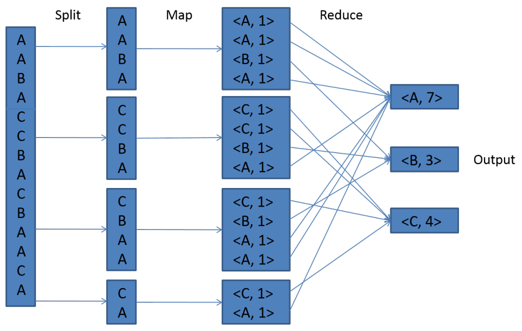

Split
Map
Reduce
A A B A C C B A C B A A C A
A B A
C C B A
C B A A
C A
<A, 1> <A, 1> <B, 1> <A, 1>
<C, 1> <C, 1> <B, 1> <A, 1>
<C, 1> <B, 1> <A, 1> <A, 1>
<C, 1> <A, 1>
<A, 7>
<B, 3>
<C, 4>
Output

Fig. 2: MapReduce Letter Count Example

III. Related Work

A. SUBDUE Graph Compression

In [3], a graph-based knowledge discovery system called SUBDUE was used. SUBDUE finds graph patterns through graph compression. For a given graph, SUBDUE initially creates several subgraphs by making each node in the graph a subgraph. SUBDUE then extends each subgraph by either adding an adjacent node or edge until the subgraphs reach a desired size. SUBDUE also calculates the number of occurrences of each subgraph in the entire graph. In this manner, SUBDUE can identify the most common subgraph with n nodes in a given graph structure. After identifying the most common n-node subgraph, SUBDUE then compresses all occurrences of that subgraph into a single node. By repeating this compression algorithm until every node in the graph is unique (i.e. there are no more patterns in the graph that occur more than once), SUBDUE can significantly compress a graph and identify common subgraph patterns. For instance, Figure 3 shows an example of how SUBDUE would compress a graph

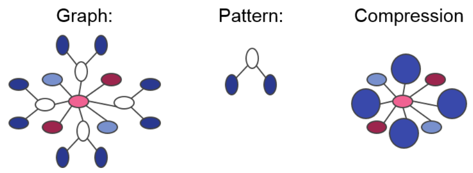


Fig. 3: Using SUBDUE for Graph Compression

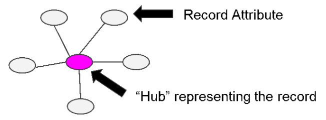


Fig. 4: Star Graph Network Representation

after finding the most frequent 3-node subgraph/pattern.

B. Star Graph Model

In [3], each individual network record is modeled as a graph. The record attributes are all connected to a central "hub" representing the record itself, as shown in Figure 4 . In essence, each subgraph/record is modeled as a star graph. Anomalies can be found by identifying frequent record attribute pairs and compressing them into a single vertex using a SUBDUE algorithm, as shown in Figures 5 and 6. Furthermore, we can model each star graph as a list, where each list represents an individual record and each list element represents a record feature. By using the SUBDUE compression strategy, common record patterns can be compressed by combining list elements. Furthermore, we can use the MapReduce paradigm to accomplish this task in parallel.

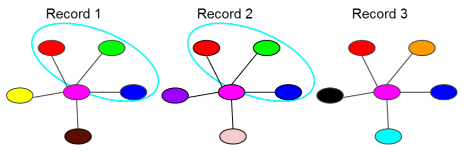


Fig. 5: Identify Network Record Patterns

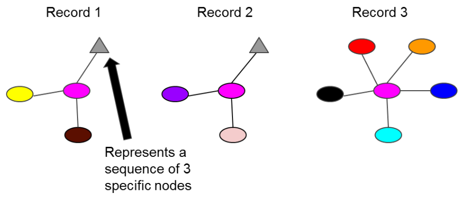
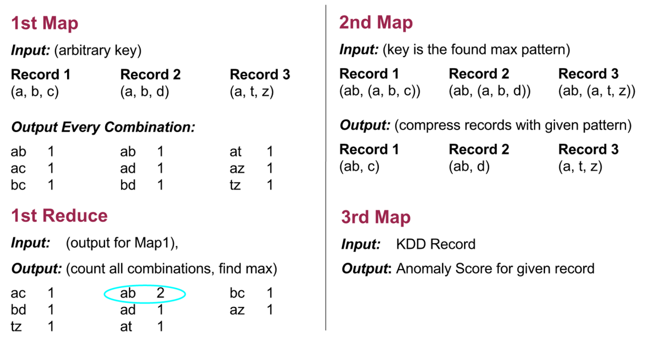


Fig. 6: Reduce Patterns to a Single Node

IV. MapReduce Algorithm

To identify common patterns in the network data, we iteratively examine record attribute pairs. For each individual record, we created all 2-node combinations of record attributes. For example, given a record with attributes *connection type=tcp, duration=0, bytes transferred=10*, the record combination would be *(connection type=tcp, duration=0), (connection type=tcp, bytes transferred=10), (bytes transferred=10, duration=0)*. We choose to examine combinations of length 2 because of the relative small number of combinations/patterns to examine. In essence, the KDD Cup dataset has 41 attributes. Thus each record has 820 combinations to examine. If we choose to also examine combinations of length three, then an additional 10,660 patterns would have to be examined per



1st Map

Input: (arbitrary key)

Record 1	Record 2	Record 3
(a, b, c)	(a, b, d)	(a, t, z)

Output Every Combination:

ab	1	ab	1	at	1
ac	1	ad	1	az	1
bc	1	bd	1	tz	1

1st Reduce

Input: (output for Map1),

Output: (count all combinations, find max)

ac	1	ab	2	bc	1
bd	1	ad	1	az	1
tz	1	at	1		

2nd Map

Input: (key is the found max pattern)

Record 1	Record 2	Record 3
(ab, (a, b, c))	(ab, (a, b, d))	(ab, (a, t, z))

Output: (compress records with given pattern)

Record 1	Record 2	Record 3
(ab, c)	(ab, d)	(a, t, z)

3rd Map

Input: KDD Record

Output: Anomaly Score for given record

Fig. 7: MapReduce Algorithm Example

record. Also, by using compression, the number of elements in many of the records will be reduced on each iteration of our algorithm; consequently, the number of patterns to examine on subsequent iterations will be decreased. In [3], they examined patterns with lengths of 2 or 3; however, our choice to only examine patterns of length 2 allowed us to eventually find patterns with lengths of 3 or more. For example, to find the 3-length pattern abc in the list (a, b, c, d) would only take two compressions. The first compression could yield the list (ab, c, d) and a second compression would yield the list (abc, d).

For our algorithm, it was important to be able to construct and deconstruct patterns of length 2. In other words, if we create a pattern abc on a particular iteration, and we want to know the two elements used to construct this pattern, we cannot tell if the two elements used were a and bc or ab and c. To remedy this problem, we surrounded each pattern we created with parenthesis and also initially surrounded each element/attribute in a given KDD Record with parenthesis. Thus, instead of examining a pattern like abc, we would examine patterns like $((a)((b)(c)))$. Here, it can easily be seen that this pattern was constructed using (a) and $((b)(c))$. We logically construct and more importantly deconstruct patterns by counting the number of open and closed parenthesis. It was also important to avoid symmetric patterns, such as ab and ba. To avoid this, we modeled each star graph as a list and took advantage of the ordered property of a list. For our MapReduce algorithm, we have three pairs of MapReduce tasks. We use the following subsections to describe each stage of our algorithm using some of the techniques described above.

A. First Map

The first Map job is used to create all KDD attribute patterns of length 2. To accomplish this, The first Map method accepts an arbitrary key with an individual record as its value. The Mapper then computes all of the 2-length attribute combinations for the KDD records. Each 2-combination attribute pair is outputted as a key with the value of 1 (similar to the MapReduce Letter Count example in Figure 2).

B. First Reduce

The first Reduce method counts these combinations so that the most common pattern can be found. Each unique pattern is mapped to the same bin with the associated values combined into a list; thus, the Reducer only needs to sum the items in this list to determine the frequency of a given pattern. Also, to decrease run time associated with I/O, we specify that each reducer should only return the most frequent key value. Thus, finding the most frequent pattern has a run time of $O(r)$, where r is the number of reducers.

C. Second Map

The second Map accepts the most common pattern as its key and an individual record as its value. If the record contains both of the attributes in this pattern, then they are removed and replaced with the full pattern. For example, given three arbitrary records *(a, b, c)*, *(a, b, d)*, and *(a, t, z)*, the most common pattern is *ab*, which would be found through the first MapReduce job. During the second Map method, the first two records could be compressed; the results would be *(ab, c)*, *(ab, d)*, and *(a, t, z)*. There is no need for a second Reduce. Notice that *ab* is a new list element. In essence, if the pattern *abc* was found on another iteration, then the list *(ab,c)* would be compressed and the resulting list would be *(abc)*.

D. Repetition of Algorithm and Termination Method

The first two MapReduce jobs will iterate until there are no more discovered patterns with a frequency > 1 (i.e. any two nodes from any two different star-graphs formed a unique pattern that was distinct from all other 2-node star-graph patterns), as this is the termination method in [3]. Each iteration finds the most frequent pattern or attribute pair and performs one compression.

The termination method we used in our initial tests required us to compress all star-graphs until any 2-node pattern had a frequency of 1. However, we found that our algorithm was taking several iterations to terminate. So we consider other possible other termination methods:

1) Base termination method on ratio of iterations and number of records
2) Base termination on ratio of number of records and the frequency of the maximum pattern discovered on a particular iteration
3) Base termination on ratio of number of attributes in the dataset and length of the max pattern discovered on a particular iteration
4) Base termination on ratio of iterations and number of attributes in the dataset

At present, the latter item in this list seems the most likely, since all of our experiments seem to generate the best results, as we will show, around the 30th iteration regardless of sample size. However the second item in this list could be used as an upper bound for the number of iterations. This can be seen in Figure 11. The upper bound intuitively makes sense because once we have eliminated all patterns with a certain percentage of occurrence, the remaining patterns should be (in theory) anomalous by virtue of their infrequent occurrence, so there seems to be no need to compress/discover all patterns in the star-graph records.

Algorithm 1 MapReduce Star-Graph Compression

```
1:  ▷ key $k$ is arbitrary
2:  procedure MAP1((key $k$, record $r$)
        ▷ Create all attribute 2-length combinations
3:      patterns  $P \leftarrow$ combinations($r, 2$)
4:      for all pattern $p \in P[p_1, p_2, ...]$ do
5:          emit: (pattern $p$, count 1)
6:      end for
7:  end procedure
8:
9:  ▷ Local variables used in each Reducer
10: $max \leftarrow 0$
11: $maxPattern \leftarrow null$
12:
13: ▷ Determine frequency of each pattern
14: procedure REDUCE1((pattern $p$, counts[$c_1, c_2, ...$]))
15:     $sum \leftarrow 0$
16:     for all count $c \in counts[c_1, c_2, ...]$ do
17:         $sum \leftarrow sum + c$
18:     end for
19:     if $sum > max$ then
20:         $max \leftarrow sum$
21:         $maxPattern \leftarrow p$
22:     end if
23: end procedure
24:
25: ▷ Method runs after REDUCE1 is finished
26: procedure REDUCE1CLEANUP
27:     emit: (pattern $maxPattern$, count $max$)
28: end procedure
29:
30: ▷ Determine the global max among all Reducers
31: $max \leftarrow$ findMaxFreq()
32: $maxPattern \leftarrow$ findMaxPattern($max$)
33:
34: procedure MAP2((pattern $maxPattern$, record $r$))
35:     ▷ Deconstruct Pattern into Original Elements
36:     element $e_1 \leftarrow maxPattern.element1$
37:     element $e_2 \leftarrow maxPattern.element2$
38:     if $r$.contains($e_1$ and $e_2$) then
39:         $r$.remove($e_1$ and $e_2$)
40:         $r$.add($maxPattern$)
41:         $r$.iterationList.add($r$.iteration)
42:     end if
43:     $r$.iteration++
44:     emit: (key $k$, record $r$)
45: end procedure
46: ▷ Repeat lines 1-45 until termination condition is reached
47:
48: ▷ Calculate anomaly scores for each record
49: procedure MAP3((key $k$, record $r$))
50:     ▷ Calculate anomaly score with Equation 1
51:     $a \leftarrow$ anomalyScore($r$.iteration, $r$.iterationList)
52:     emit: (String $r$.ID, Value a)
53: end procedure
```

E. Third Map

A final Map job was used to calculate the anomaly score for each KDD record. The anomaly score formula we present with Equation 1 requires that the total number of iterations used be known. However, this value might not be known if the number of iterations is not fixed. For example, when we used the termination method in [3] that required us to continue finding patterns until the frequency of was < 2, we did not know how many iterations we needed. To remedy this, we saved certain information in each star graph on each iteration. For instance, if a star graph was compressed on on the ith iteration, then we would store the value of i in a list. We also stored and updated the current iteration number in each star graph. When our algorithm terminated, we knew the total number of iterations and we also knew what iteration each star-graph was compressed on. We will use the next section to explain how we calculated anomaly scores. An example of our algorithm is shown in Figure 7.

V. Star Graph Analysis

In [3], an anomaly score is calculated for each record. This score is based on whether or not a record was compressed during the ith iteration of the fist two MapReduce jobs. Thus, records that can contain common patterns will experience more compression and therefore have a lower anomaly score. Whereas anomalous records will have less frequent patterns and therefore experience less compression and have a higher anomaly score. The equation used in [3] is the following:

$$A = 1 - \frac{1}{n}\sum_{i=1}^{n}(n - i + 1) * c_i \qquad (1)$$

Each record starts with an anomaly score of 1 and is updated with the formula inside the summation operator: n is the number of iterations, i is the current iteration number, c_i is the percentage of the star graph that was compressed/reduced on the ith iteration. It should be noted that the $n - i$ term in the above equation has the effect of weighting common patterns discovered at earlier iterations higher than the patterns discovered at later iterations. This is because patterns that are discovered at later iterations will become increasingly anomalous since the most common patterns will be removed at earlier iterations. As for the c_i term, since we are only observing patterns that have a length of two nodes/attributes, a star graph will only be compressed by a maximum of one element. Thus, c_i can be defined on a list with x number of elements/attributes to be:

$$c_i = \frac{x - (x - 1)}{number\ of\ attributes} = \frac{1}{number\ of\ attributes}$$

For the KDD dataset, the number of iterations is 41. If no compression takes place on the ith iteration, then c_i will be zero. Specifically:

$$c_i = \left\{ \begin{array}{cc} \frac{1}{number\ of\ attributes}, & \text{graph was compressed} \\ 0, & \text{graph was not compressed} \end{array} \right\}$$

In [3], they used “anomaly rank” to determine the quality of their results. They only used one or two anomalies in their samples, so we wrote a simple formula to generalize their method. We calculated the anomaly rank for a sample of records with the following formula:

$$Anomaly\ Rank = \frac{\sum_{i=1}^{n} \text{rank}(n_i)}{n}$$

where n is the number of intrusive records, n_i is the ith anomaly in the dataset, and rank(n_i) is is calculated by sorting KDD records based on anomaly scores. Rank 1 means the highest likelihood for the anomaly. A lower anomaly rank indicates that anomalous records had higher anomaly scores. A lower anomaly rank is desirable because it indicates that many anomalous records are being ranked higher than normal records, so it can be easier to distinguish between the normal and intrusive records. To calculate the average anomaly rank for a particular attack, we took the average of all the anomaly scores with that particular attack label. To calculate the average anomaly rank for all of the attacks, we took the average of all the anomaly scores with an attack label.

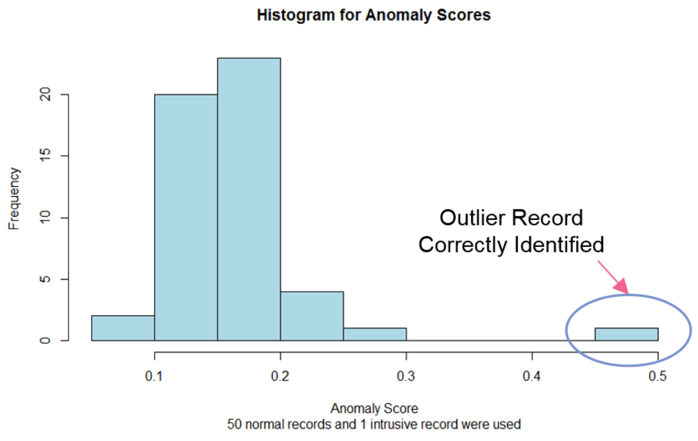


Fig. 8: Anomaly Detection Results

VI. Results

For our initial test, we chose fifty random records that were labeled as normal and one record labeled as intrusive. This is similar to the experiment performed in Figure [3]. We ran our algorithm using Hadoop MapReduce in non-distributed mode. Our algorithm terminated after eighty iterations and three minutes of run time. Portsweep was used as the type of network intrusion. As can be seen in Figure 8, the network anomaly was clearly identified.

We then ran a test in an attempt to duplicate the results in [3]. We chose one subset for each attack with fifty records: forty-nine normal records and one attack record. We randomly generated each subset ten times with replacement, so duplicates were possible. Our results are shown in Figure 9. Our average anomaly rank for all attack records used was 4.6; the average anomaly rank in [3] for all attack records tested was about 4.75. The possible reason for our improved results could be that we only examined patterns with two nodes, whereas [3] examined patterns of two and three nodes, so it could be possible that some patterns in the SUBDUE algorithm were

TABLE I: Values of Confusion Matrix for 1,001,289 Network Records

True Positives	True Negatives	False Positives	False Negatives
28282	919456	53325	226

TABLE II: Statistical Percentages for 1,001,289 KDD Network Records

Sensitivity	False Positive Rate	Specificity	False Negative Rate	Accuracy	Detection Rate	Negative Predictive Value
99.21%	5.48%	94.52%	0.79%	94.65%	34.65%	99.98%

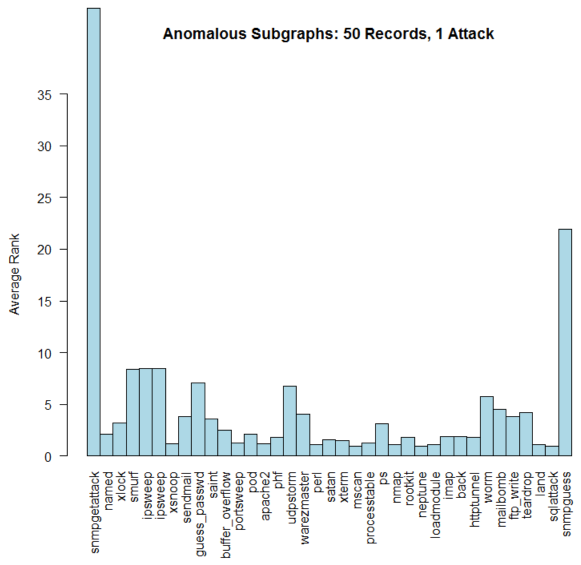


Fig. 9: Results using MapReduce

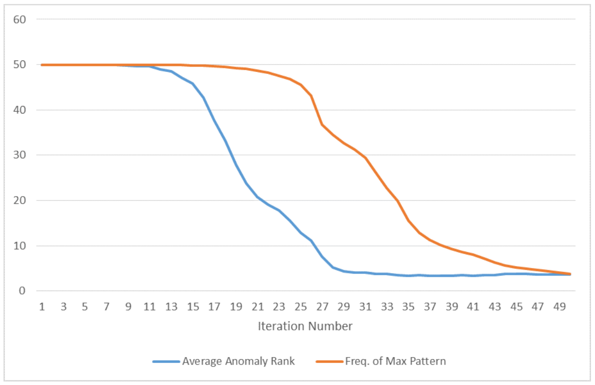


Fig. 10: Graph of the average anomaly rank and frequency of maximum occurring pattern through first 50 iterations of 50 Records

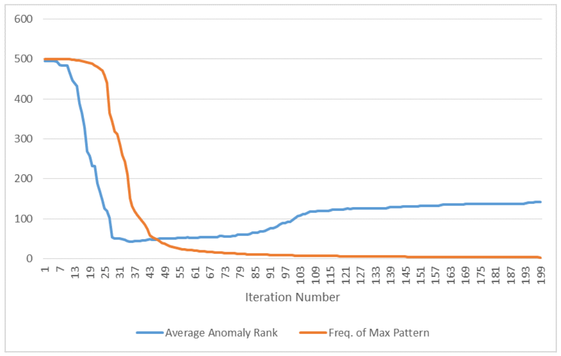


Fig. 11: Graph of the average anomaly rank and frequency of maximum occurring pattern pattern through the first 200 iterations of 500 Records

excluded. In other words, our algorithm could create a pattern of length four after three iterations/compressions, whereas their SUDBUE algorithm might create a pattern of length six after two graph compressions and may have skipped many of the patterns of length four. Thus, our algorithm is capable of enumerating all possible patterns. For example, given a list (a, b, c, d, e, f), our MapReduce algorithm could find the pattern $abcd$ after three iterations. The intermediate lists could be (ab, c, d, e, f) and (abc, d, e, f), which could finally yield the discover of pattern $abcd$ in the list $(abcd, e, f)$. Whereas for the SUBDUE algorithm, it is possible that the patterns abc and def would be discovered during the compression process. The intermediate list could be (abc, d, e, f), which would subsequently yield the list (abc, def). In this case, the pattern $abcd$ would not be found be SUBDUE.

Our algorithm took well over 41 iterations of our MapReduce algorithm to terminate. This was much higher than we expected, so we checked the results of our algorithm after each iteration. As shown in Figure 10, our algorithm performs slightly better at early iterations than at later iterations. The orange line shows the frequency of the maximum pattern on a particular iteration, as we originally thought this might be a reason for our improvement at earlier iterations. The blue line represents the overall average anomaly rank for all attacks. The average anomaly rank drops as low as 3.6, compared to a final average rank of 4.6. We also ran 30 tests of 500 records, with the same ratio of normal versus attack records (i.e. attack rate was 2%). For two of our tests, we combined attacks that had a low frequency of occurrence. In Figure 11, it can be seen that our algorithm performs much better at early iterations.

To see if this phenomenon existed with larger datasets, we performed a test on over a million records with an attack rate of approximately 2.8%; in essence, there were approximately 28,000 anomalies. We examined a histogram of anomaly scores shown in Figure 12. We flagged records with an anomaly score > 0.5 as anomalous. Our results are listed in Tables I and II. As with most anomaly detection methods, our test yielded a high number of false positives [1]. In essence, many abnormal but legitimate network communication records were flagged as anomalous.

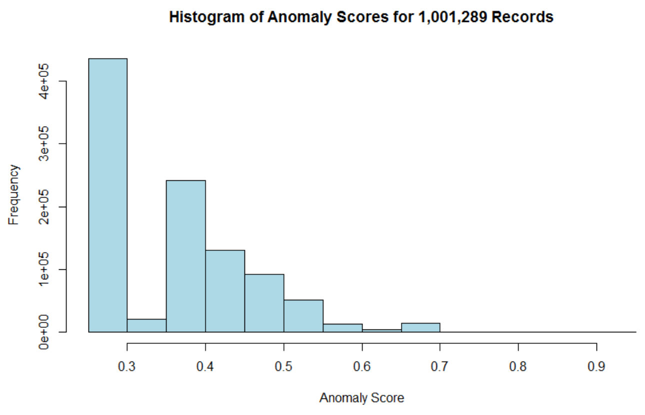


Fig. 12: Anomaly Detection using a Larger Dataset

TABLE III: Algorithm Runtimes at UMBC HPCF Termination parameter was 32 iteration

Runtime is in Seconds	4 Nodes	8 Nodes	16 Nodes
50 Records	1,244.35	not tested	not tested
45,214 Records	2,287.01	1,956.85	2,011.65
667,370 Records	8,120.59	8,005.19	7,927.97
1,001,289 Records	9,112.92	8,313.58	8,365.27

A. Runtimes

We ran our algorithm at the University of Maryland, Baltimore County High Performance Computing Facility (UMBC HPCF). We ran our code on a varying number of nodes, with each node containing eight cores. For each different configuration, we only ran our code once. Our results are in Table III. Our runtimes do not seem to indicate a very desirable speedup. Possible reasons for this is the overhead associated with setting up each Hadoop MapReduce job. Also, patterns that took longer to count may have been processed by the same node, which would require other nodes that finished sooner to wait on the lagging node before continuing. Also, we modified our algorithm to delete directories in the HDFS as they became obsolete. This seemed to decrease the overhead associated with the HDFS managing large amounts of files. After this modification, we ran a test using a four node Hadoop cluster with 1,001,289 records in 8,207.25 seconds. Although our runtimes did not achieve the desired speedup usually associated with parallel algorithms, we were still able to process large amounts of data. Whereas [3] only used sample sizes of 50 or 100 records, we were easily able to process datasets with over one million records.

VII. Possible Future Work and Conclusions

In the future, we plan to run more tests with our algorithm on a Hadoop cluster. To improve performance, we may be able to reduce the overhead associated with Hadoop by implementing our algorithm in Apache Spark. Also, we may be able to develop a real time algorithm by adding KDD records to the histogram in Figure 12. Basically, once we have calculated our initial results for a given dataset, we can add additional records to our results. This can be accomplished by saving a list of frequent record patterns discovered using our MapReduce algorithm on particular iterations. We could then compress an individual record using this list of patterns and quickly calculate an anomaly score for that record to determine if it is likely to be a normal or an anomalous record. In conclusion, our algorithm has yielded a highly sensitive anomaly detection method; however, this seems to come at the cost of a low positive predictive value. We have been able to process large amounts of data by utilizing the Apache Hadoop framework. In addition, our algorithm works best on discrete or categorical datasets, whereas more than half of the KDD attributes are continuous variables [2]. Nonetheless, we have been able to show that using graph compression for detecting network anomalies can achieve highly accurate results.

Acknowledgment

This research was funded by the National Science Foundation, Award# CCF-1460900 under the Research Experience for Undergraduates Program. We would like to thank Professor Matthias K. Gobbert at University of Maryland, Baltimore County (UMBC) for allowing us to use the UMBC High Performance Computing Facility .

References

[1] I. Aljarah and S. A. Ludwig, "Mapreduce intrusion detection system based on a particle swarm optimization clustering algorithm," *IEEE Congress on Evolutionary Computation*, pp. 955–962, Jun. 2013.

[2] S. D. Bay, D. F. Kibler, M. J. Pazzani, and P. Smyth, "The uci kdd archive of large data sets for data mining research and experimentation," 2000.

[3] C. C. Noble and D. J. Cook, "Graph-based anomaly detection," *Proceedings of the ninth ACM SIGKDD international conference on Knowledge discovery and data mining.*, pp. 631–636, Aug. 2003.

[4] J. Therdphapiyanak and K. Piromsopa, "An analysis of suitable parameters for efficiently applying k-means clustering to large tcpdump data set using hadoop framework," *Electrical Engineering/Electronics, Computer, Telecommunications and Information Technology (ECTI-CON), 2013 10th International Conference on*, pp. 1–6, May 2013.

[5] H. Kelly, A. Bull, P. Russo, and E. McBryde, "Estimating sensitivity and specificity from positive predictive value, negative predictive value and prevalence: Application to surveillance systems for hospital-acquired infections," *Journal of Hospital Infection*, vol. 69, no. 2, pp. 164–168, 2008.

[6] Taylor and R. C., "An overview of the hadoop/mapreduce/hbase framework and its current applications in bioinformatics," *BMC bioinformatics*, vol. 11, no. Suppl 12, p. S1, Dec. 2010.