William Beninger
Krysti Bouttell-Bonnar

November 25, 2013

FINAL REPORT
COMPUTER SCIENCE 4310
Mining Colossal Frequent Patterns

Summary of Components Being Implemented:

Given that the amount of data that is available to us at any given time has been steadily increasing along with the sheer size of the individual pieces of data, it becomes clear that mining frequent patterns has become more difficult to do in a timely manner. To solve this issue many have looked at how to efficiently mine what are called colossal frequent patterns. Colossal patterns are described as patterns that are exceedingly long (Zhu et. al, 2007) . Mining colossal patterns with any previously known frequent pattern algorithm presents the problem of the downward closure property. Which is that for every frequent itemset all of it's subsets will also be frequent. This property causes issues when mining colossal patterns as a colossal frequent pattern by definition is a large pattern which implies that it will have a substantial number of subsets that will also be frequent. Here in lies the problem, when building frequent patterns most algorithms will start from the bottom, or 1-itemsets, and linearly work their way up to the frequent patterns. However, when dealing with colossal patterns and a linear bottom up approach the number of midsized frequent patterns needed to construct larger patterns becomes exponential (Zhu et. al, 2007). This causes algorithms to get stuck at the mid point while trying to analyze the exponential number of sub patterns. The algorithms that deal with mining colossal frequent patterns try to alleviate the problem of dealing with the large number of sub patterns.

The first algorithm that we looked at was from “Mining Colossal Frequent Patterns by Core Pattern Fusion” written by Zhu et. al. The core pattern fusion algorithm attempts to bypass the previously mentioned mid point of exponential sub patterns. It does this using what the authors call

core patterns. Core patterns are patterns that tend to be 'clustered' together with other similar patterns. Due to this clustering, core patterns are usually the sub patterns that make up larger patterns. By using a distance measure to determine which patterns are close to one another, the algorithm then merges those patterns that are within a specific distance from a chosen pattern. It is this merge step that allows core pattern fusion to skip the mid point of exponential sub patterns, as it is possible to jump from an itemset with four elements to an itemset with eight elements. Thus, it is not a linear algorithm and avoids the pit fall of the downward closure property.

The second algorithm to handle colossal patterns was from “An Efficient Approach to Colossal Pattern Mining” written by Dabbiru and Shashi. The algorithm is called colossal pattern miner. Colossal pattern miner uses a vertical representation of the data as opposed to the more commonly used horizontal representation (Dabbiru and Shashi, 2012). This is done to reduce the number of scans done on the database. Colossal pattern miner starts by using any existing efficient pattern mining algorithm to get the 2-itemsets, this becomes the initial pool. It then uses a clustering algorithm to cluster the itemsets based on how many transactions contain that itemset. For example if an itemset gamma corresponded to six transactions that contained gamma then gamma would go in the cluster with a cluster mean of six. The algorithm then starts from the largest cluster working in descending order through the clusters. Starting with the largest cluster it randomly chooses a pattern alpha. For each alpha it loops through the remaining patterns in the initial pool, beta. Similar to the first algorithm it then calculates the distance between alpha and beta and checks to see if that distance is smaller than the chosen core ratio. If the distance is smaller than store beta in alpha's list of core patterns. For each chosen alpha merge alpha with it's list of core patterns. In the merge step merge alpha with it's first core pattern. If the support of alpha drops significantly once merged with the first core pattern undo the merge and start over with alpha's second core pattern. Continue doing the merge until all of alpha's core patterns have attempted to be merged with alpha always undoing those merges that cause a significant

drop in the support count.

Implementation Details:

The core pattern fusion algorithm begins by using any efficient algorithm to create 3-itemsets, in the implementation we use Apriori, this is then used as the initial pool. From the initial pool we randomly choose an itemset, we will call this itemset alpha. The algorithm then loops through the remaining itemsets, called beta, from the initial pool and calculates the distance between alpha and beta. If the distance between alpha and beta is smaller than a specific core ratio, we use 0.5 as our core ratio, then itemset beta is a core pattern to itemset alpha and should be stored in alpha's list of core patterns. Core pattern fusion loops through the process of choosing alpha and finding alpha's core patterns until a specific number of alpha's are chosen, this number is usually user defined, it is known as the maximum number of patterns to find, we use 2 in our implementation. Once we have the desired number of alpha's chosen, for each alpha we fuse alpha with with it's list of core patterns. These lists of fused core patterns become the new initial pool and the algorithm starts again until we have found the desired number of patterns to find.

For the core pattern fusion algorithm there are a few numbers that must be known prior to running the algorithm. The first is the maximum number of patterns that the algorithm should search for. This is very important because if the number of patterns is set to a high value the algorithm will return patterns that may not be colossal. If the number of patterns to search is a low value, say one, then occasionally the algorithm will return with no patterns. This is due to the random nature of how the algorithm picks alpha. Another value that must be known prior to running the algorithm is the minimum support that is to be used.

The first paper “Mining Colossal Frequent Patterns by Core Pattern Fusion” (Zhu et. al, 2007) was fairly straight forward thus there was nothing that we had to infer from it. Although we did make

one minor alteration to the core pattern fusion algorithm. After the merge step occurs there was nothing in place to ensure that the new merged pattern was frequent. Due to this the algorithm would occasionally return a colossal pattern that did not meet minimum support. Therefore, we added a few lines to ensure that any pattern returned by the core pattern fusion algorithm had minimum support.

The implementation of the core pattern fusion algorithm required the use of maps, vectors and structs to store the various data. It also required six functions. The first being a function to create the data we were going to be testing the algorithm on, which was Diag40. Diag40 is a 40x40 matrix where each row contains the numbers 1 through 40 in increasing order. The numbers on the diagonal are then removed giving us a 40x39 matrix where no two rows are the same. We then append twenty more rows that contain the numbers 41 through 79 in increasing order. This now results in a matrix or two-dimensional array that is 60x39 where the last 20 rows are all identical. This gives us a minimum support of 20. We take each row to be a transaction.

The second function that was needed was a function to run apriori and get the 3-itemsets. This function is a variant of Apriori as it was only designed to calculate the 3-itemsets. The third function is also a helper function to get things set up before core pattern fusion can be called. It is the contains function it allows us to test whether or not a specific value is in a vector. There is also the distanceSet function which is used to calculate the distance between pattern alpha and pattern beta. This function calculates the intersection of the transactions containing alpha and the transactions containing beta divided by the union of the transactions containing alpha and the transactions containing beta this is then all minused from the number one.

The next function is patternFusion which is used to randomly select a pattern alpha until the number of alpha's chosen is the maximum number of patterns to find. For each alpha loop through the initial pool and look at every other pattern, called beta. It then calls distanceSet with alpha and beta to get the distance between alpha and beta. If the distance of alpha and beta is less than the core ratio, we implemented 0.5 as the core ratio, then store beta in a list of alpha's core patterns. It then loops through

all the alpha's and calls fusionSet with alpha's list of core patterns. The last function used to implement core pattern fusion is fusionSet. This function unions alpha with each of it's core patterns to create a larger pattern.

The second paper “An Efficient Approach to Colossal Pattern Mining” (Dabbiru and Shashi, 2010) was not that straight forward and required a few educated guesses to implement. They will be pointed out as the implementation is described. The implementation of colossal pattern miner required the use of maps, vectors, structs and nine different functions. The first function was createDiag40 which works the same as in the previous algorithm. The next function to be used was createVerticalData this function took the information in the two-dimensional array and stored it in a vertical representation. The third function is apriori which uses Apriori to create the 2-itemsets. The Apriori function for this implementation is slightly different than that of the previous as the data is stored in a vertical representation. The colossal pattern miner implementation also uses the contains function to return whether a specific value is in a vector. The next function implemented was the distanceSet function which is similar to that of the above algorithm, which computes the distance between pattern alpha and pattern beta.

The next function is frequencyClusters which takes as input the initial pool and sorts all the patterns into clusters based on the number of transactions that contain that pattern. From there the next function is sortClusters which goes through the clusters and sorts them in descending order based on the number of patterns that fall into each cluster. That is all of the functions that get everything set up for colossal pattern Miner.

The next function is colossalPatternMining which starts the process of finding colossal patterns. This function is called from the main function. One of the assumptions we made was in the main function as the original algorithm required the use of a remove subsets function after colossalPatternMining was called. This seemed redundant given that when in the colossal pattern

mining function we could ensure the returned pool did not have any subsets. In colossalPatternMining select a pattern alpha from the largest cluster. At this point in the function we made a few alterations to do if not statements as opposed to if statements as we found the code more readable with the changes. The algorithm checks to make sure that the chosen alpha has minimum support. If it does we then loop through the rest of the pool getting our pattern beta. Now that we have alpha and beta we call distanceSet to determine the distance between alpha and beta. If the distance is less than the core ratio we put beta into alpha's list of core patterns. Then for each alpha we call patternMerge on alpha's core list.

The final function that colossal pattern miner requires is patternMerge which takes alpha's core list and merges it with alpha. This function starts with ensuring that alpha and the first beta from alpha's core list are not subsets. If they are not subsets union alpha's pattern with beta's pattern and intersect alpha's transactions and beta's transactions. Check to see if the intersection of the two sets meets minimum support. If the intersection meets minimum support call distanceSet and check the distance between alpha and the unioned alpha pattern and beta pattern if the distance is greater than 0.5 store beta as a frequent long pattern, but not a colossal pattern and undo the union of alpha's pattern and beta's pattern. Otherwise, here we had to make another estimation as to what the paper was stating. It states that once the algorithm gets into this final else statement to update alpha with the unioned and intersected sets that were calculated earlier. However, we were unsure if this was supposed to permanently alter alpha's value. We chose to make it permanently alter alpha's value. With this new alpha union the pattern and transaction with the set colossal and return colossal.

We also had to make the same alteration as in the first algorithm. To ensure that any merged patterns had minimum support or we would occasionally get colossal patterns that did not hve minimum support.

Evaluation Strategy:

To evaluate the two algorithms we implemented they will be run on a Toshiba laptop using an Intel Pentium CPU 2020M @ 2.40 Ghz processor, with 6.00 GB of RAM and using a g++ compiler version 4.8.1.

For the evaluation of the algorithms we decided to time in seconds how long the algorithms took to find the colossal patterns. In the proposal we had stated that we would have liked to run the data from the first paper on the second algorithm and the data from the second paper on the first algorithm. However we were unable to locate the yeast data that was used by Dabbiru and Shashi. So we no longer have the option of exchanging the data of the papers. Although both papers did run their data on different lengths of $Diag_n$ thus we will do the same to see whether one algorithm outperforms another continuously or at a certain size n does one algorithm have a better run time.

Results of Evaluation:

	Core Pattern Fusion	Colossal Pattern Miner
Data size: 12 x 5 Min Support: 6 Max Patterns: 2	0 seconds	0 seconds
Data size: 18 x 8 Min Support: 9 Max Patterns: 2	0 seconds	2.02 seconds
Data size: 22 x 10 Min Support: 11 Max Patterns: 2	0.02 seconds	3.695 minutes
Data size: 26 x 12 Min Support: 13 Max Patterns: 2	0.05 seconds	> 1 hour
Data size: 38 x 18 Min Support: 19 Max Patterns: 2	0.29 seconds	> 1 hour
Data size: 60 x 39 Min Support: 20 Max Patterns: 2	10.41 seconds	> 7.5 hours
Data size: 80 x 59 Min Support: 20 Max Patterns: 2	62.69 seconds	did not attempt

For the above table, for those cells that have relavant runtime outputs. We ran each of the algorithms ten times on the same data and then averaged to runtimes. It is clear from the above table that core pattern fusion is a much superior algorithm when attempting to mine colossal patterns. Due to the extremely lengthy running time of the colossal pattern miner algorithm we were unable to get proper values for all of our data sets to use in the evaluation.

Conclusions:

It is clear that the core pattern fusion algorithm greatly outperforms the colossal pattern miner algorithm. However, this could be a direct result of some of the assumptions that we made in regards to the implementation of the colossal pattern miner algorithm. Therefore it is possible that it may run more efficiently if implemented using different assumptions. Coloassal pattern miner can also be implemented using parrallelism, which we chose not to include in our imlementation. This could also

reduce the runtime of the algorithm.

In doing this project we learnt quite a bit about how the expanding size of data is causing issues when that data needs to be mined. This is particularly evident when biology and computer science overlap, where it is necessary to mine data such as gene sequences or microarray data. These are guaranteed to have colossal patterns.

Sources:

Dabbiru, M., Shashi, M. (2010). An Efficient Approach to Colossal Pattern Mining. *International Journal of Computer Science and Network Security*, 10, 304-312.

Zhu et. al. (2007). Mining Colossal Frequent Patterns by Core Pattern Fusion *Data Engineering, 2007. ICDE 2007. IEEE 23rd International Conference*, , , 706-715. Doi: 10.1109/ICDE.2007.367916