Implementing Maximal Quasi Clique in a Distributed Environment.

Vishwas Shanbhog , Rustom Shroff
Department of Computer Science
At University at Buffalo
Email: {vishwass , rustomsh }@buffalo.edu

Instrutor:
Prof. Jaroslaw Zola

1 ABSTRACT

Maximal Quasi Clique is an important clustering algorithm. It has application in various fields like classifying molecular sequences in genome projects by using a linkage graph of their pairwise similarities, analysis of massive telecommunication data sets, as well as various data mining and graph mining applications, such as cross-market customer segmentation. All these problems generally have very noisy data and there is no similarity score threshold which can accurately distinguish between two clusters. Hence we use Maximal Quasi Clique algorithm to identify clusters which are almost cliques and by varying the value of γ we can obtain clusters of different sizes.

2 QUASI CLIQUE

Let G = (V, E) be an undirected graph and let U ⊆ V. The U-induced subgraph G'= (U, E_U) is a γ -quasi-clique if $|E_U| \geq \gamma \cdot \binom{|U|}{2}$ where, 0 < $\gamma \leqslant 1$. G' is maximal if no larger γ -quasi-clique exists that would contain G'. The following are the results obtained when our code was run. As per our implementation we are ignoring the γ for quasi cliques which have less than 3 edges.

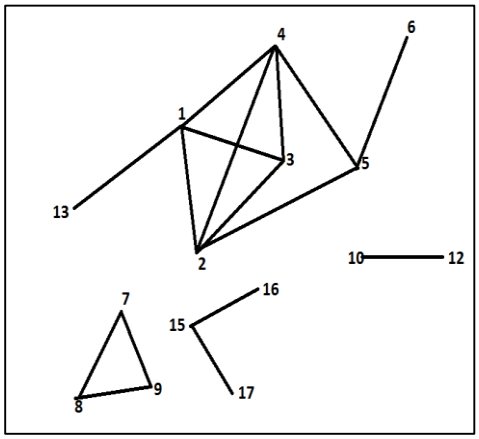


Figure 1: Original graph

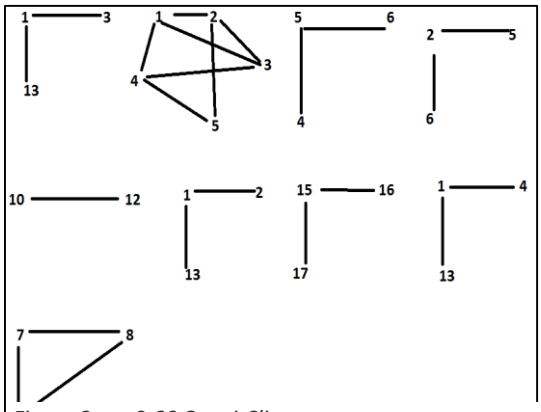


Figure 2: γ =0.66 Quasi-Cliques

3 ALGORITHM

This algorithm is made to work with sparse data, the type that occurs in genome sequences. The algorithm that we have implemented does not guarantee that all maximal quasi-cliques will be found at the end of the computation.

Input:
List/set of edges E
Parameter γ
Output: Set C of maximal γ-quasi-cliques

1: $C \leftarrow \emptyset$
2: **for** each $e_i \in E, e_i = (u, v)$ **do**
3: $c.key \leftarrow \{u, v\}$
4: $c.value \leftarrow \{(u, v)\}$
5: $C \leftarrow C \cup \{c\}$
6: **end for**
7: **repeat**
8: $change \leftarrow false$
9: find $c_i, c_j \in C$ such that $c_i.key \cap c_j.key \neq \emptyset$
10: $c.key \leftarrow c_i.key \cup c_j.key$
11: $c.value \leftarrow c_i.value \cup c_j.value$
12: **if** $|c.value| \geq \gamma \cdot \binom{|c.key|}{2}$ **then**
13: $C \leftarrow C \setminus \{c_i, c_j\}$
14: $C \leftarrow C \cup \{c\}$
15: $change \leftarrow true$
16: **end if**
17: **until** $change = false$
18: **return** C

4 OPTIMIZATIONS PERFORMED

Our source of inspiration to implement Gamma Quasi Cliques enumeration algorithm has been the Implementation of CLOSET . The implementation of CLOSET can be found at http://aluru-sun.ece.iastate.edu/doku.php?id=closet.

1) Initially, our map reduce implementation iteration generated one set of nC2 pairs of new possible clusters. Using Closet as Reference, we were able get the idea of multiple iteration within mapreduce jobs up to a particular limit. This improved performance.

2) MEMORY ERROR: DATA Structures used initially were complex python set and a separate python class for Clusters which resulted in the termination our program due to memory error. Switched back to using simple data structure like Python tuple and list. Implementation of set was done using lists.

3) COMBINATIONS USING rdd.reduceByKey(): While using reduceByKey we maintained lists of older clusters and using them ,formed and new clusters and kept them in the seperate lists. Due to this, second mapreduce iteration processing time increased heavily. Implementation improved in performance to use pyspark rdd.groupByKey(numPartitions=x)

4) ALTERNATE IMPLEMENTATION: pyspark supports a function called Cartesian. This function was used to find the Cartesian product of clusters rdd with itself resulting all the nC2 combinations and all of them appearing twice and combination of clusters with itself. We filter all the clusters combinations which are duplicated. Also filter cluster combinations which do not have similar node. Hence for sparse there too many redundant combinations. So second level iteration combinations increased rapidly resulting in poor performance. Hence did not continue with it.

5) STORAGE-LEVEL: Considering the number of clusters formed during 2nd and higher iterations the memory required for this computation is very high.So using cache is not an option.We have opted the option Storage level : "MEMORY _AND_DISK_SER" where in the RDD are spilled to disk if memory is not sufficient for the serialized object. However the serialization of large dataset will result in large decrease in performance during re-computation.

6) MERGING CLUSTERS: Improved performance using combineByKey instead of reduceByKey for merging clusters.

5 COMPLEXITY

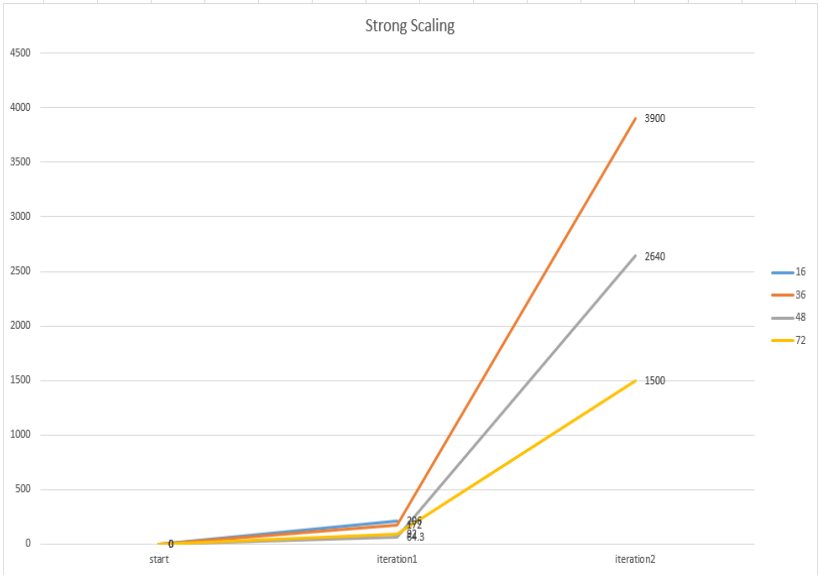
The complexity of this algorithm is $O(|S|*|V|^2)$, where S is the set of vertices of the constructed maximal quasi-clique and V is the total number of vertices in the graph. Since the complexity of graph depends on both V and S the complexity of the algorithm increases if the maximal clique size increases as well. Hence this algorithm is said to perform well with sparse data.

6 ENVIRONMENT USED

We have used the Spark environment in the Center for Computational Research in University at Buffalo to implement the quasi clique algorithm. We have used IB CPU-E5645 which have 12 cores and 48GB Ram.

7 ANALYSIS

	Executors			
.	16	36	48	72
start	0	0	0	0
iteration1	206	172	64.3	92
iteration2	NA	3900	2640	1500



We observe strong scaling in our code because when we increase the number executors we observe a decrease in the iteration times. We also observed that for the first iteration the network overhead played was not compensated by the increase in the executors as there was not much computation to be performed on each executer but in the second iteration there was sufficient computation to be done on each executor and hence we observed a performance gain.

8 CONCLUSION

Due to difficulty in carrying out the optimization of the code we were not able to run over code on large data sets until it converged. When we ran our code on large data sets with fewer resources we encountered memory issues after the first two iterations itself. Due to this we have been able to converge on small datasets in one iteration but were not be to tune Spark parameters