On Using Cached Results to Enumerate Maximal k-Cliques in GeneWeaver

Qi Yang
Department of Computer Science and Software Engineering
University of Wisconsin-Platteville
Platteville, Wisconsin 53818

Greg Speegle and Erich J. Baker
School of Engineering and Computer Science
Baylor University
Waco, TX 76798-7141

Abstract—The online biological data analytics tool GeneWeaver [1] uses a fast algorithm to directly compute k-cliques between different sets of data. By caching such results, we are able to compute k-clique faster in some cases. We derived a formula to determine whether or not to cache a result. In order to know if the cached results can be used to compute a desired k-clique, we also created a new algorithm to solve the generalized set coverage problem.

Keywords: k-Partite Graphs, Maximal Cliques, Set Covering Problem, NP-Hard

I. INTRODUCTION

GeneWeaver [1] provides a set of web-accessible computational tools for the analysis of sets of genes derived from biological data repositories. While GeneWeaver was originally built on a relational database model with a web-based front end for data and analytics visualization, recent efforts have been made to move GeneWeaver to a big data environment in order to leverage parallel processing and provide enhanced analytical capabilities.

The underlying GeneWeaver data structure uses a bipartite data model to represent relationships between genes and related biological processes, diseases or functions, and, by extension, enables exploration of k-partite graphs through discovery of maximal k-cliques.

The k-clique finding algorithm [9] is used to compare multiple partitioned gene sets in GeneWeaver. While this algorithm is currently the best such algorithm for finding k-cliques, the functional implementation and use cases represented in GeneWeaver allows an algorithm which can be orders of magnitude faster than [9].

The new algorithm, called the Natural Set Join, exploits the fact that k-clique finding in GeneWeaver can be exploratory, meaning that previous results can be cached and exploited to provide significant performance improvement in some cases. To determine if a set of cached results can be used, we designed an algorithm for a variation of the set coverage problem with degree d.

This paper’s contributions are:

- a clear caching strategy for maximal k-clique finding in GeneWeaver

The rest of the paper proceeds as follows. Section II maps k-partite graphs to tables and describes the Natural Set Join used to enumerate maximal k-cliques. Section III reviews the Set Coverage Problem and defines the Set Coverage problem with degree d. Section III-A presents the fast, approximate algorithm. Section III-C shows the experimental results of the set coverage algorithm. Section IV shows the applicability to GeneWeaver. Section V contains related work and Section VI summarizes the paper.

II. NON-1NF RELATIONAL ALGEBRA FOR ENUMERATING ALL MAXIMAL k-CLIQUEs

A. Graph Tables

An undirected graph $G = (V, E)$ is called a k-partite graph if $V$ can be divided into $k$ distinct sets (partitions) such that any two nodes in the same partition are not connected. We denote the set of partitions as $P$.

For a k-partite graph $G = (\bigcup_{p \in P} p, E)$, a $k$-clique $v^k$ is a subset of $V$ such that it contains at least one node from every partition and every two nodes in $v^k$ are connected if they are not in the same partition. A $k$-clique $v^k$ is maximal if no node can be added to $v^k$ and it remains a $k$-clique.

Consider Figure 1. Some example 3-cliques are $\{1, A, \}$ and $\{2, A, *\}$ while the only maximal 3-clique is $\{1, 2, A, *, \}$.

We use non-1NF tables and extended relational algebra operations to represent k-partite graphs and maximal k-cliques. For a k-partite graph $G$, a graph table $T_G$ corresponding to $G$ has $k$ attributes, one attribute for each partition, and all rows of $T_G$ represent maximal k-cliques of $G$. Notice that the domain of each attribute is the power set of the corresponding partition.

Table I is the graph table representing the k-partite graph in Figure 1.

<table>
<thead>
<tr>
<th>$A_0$</th>
<th>$A_1$</th>
<th>$A_2$</th>
</tr>
</thead>
<tbody>
<tr>
<td>${1, 2}$</td>
<td>${A}$</td>
<td>${$$}$</td>
</tr>
</tbody>
</table>

Table I

THE GRAPH TABLE CORRESPONDING TO FIGURE 1.
Finally, we use a subgraph to explicitly refer to the removal of an entire partition from a graph. The partition removed is generally clear from the context.

B. Natural Set Join

The natural set join (NSJ) is an extension to the relational algebra natural join operator. It is a binary operator over two relations. For every attribute in common between the two relations, NSJ takes the intersection of the attributes. If the intersection of all common attributes is non-empty, the NSJ returns a row with one copy of each attribute. Note that if the relations are in 1NF, the NSJ and the natural join are the same operation. However, for non-1NF relations where the domains of the attributes are sets, the results of the natural join and NSJ are different.

Let $\tau_{f(x)}(r)$ be an extended relational algebra operator indicating the application of update $f(x)$ to every row in $r$.

We apply the natural set join, $\cong$ to graph tables. Let $r(R)$ and $s(S)$ represent two graph tables over schemas $R$ and $S$.

$$r \cong s = \Pi_{R \cup (S-R)} \sigma_R(\tau_{f(A)}((r \times s)))$$

where $f(A)$ is $r.A \leftarrow r.A \cap s.A$ applied to all attributes in $R \cap S$ and $P$ is $r.A \neq \emptyset$ for all attributes in $R \cap S$.

Given the non-1NF structure of graph tables, the notion of duplicate elimination in relational algebra (denoted $\delta$) must be extended by the maximal operator denoted $\tilde{\delta}$. For a graph table $T$, $\tilde{\delta}(T)$ removes all rows $t$ such that there exists another row $t'$ such that for all attributes $A$, $t.A \subseteq t'.A$. Note that the maximal natural set join is required to enumerate all maximal k-cliques from cached graph tables.

C. k-Clique Algorithm

Given a set of graph tables $\tau$, the maximal natural set join of all elements in $\tau$ is a representation of a maximal k-clique graph $G$ provided $\tau$ has the following properties:

1) no attribute in any element of $\tau$ represents a partition not in $G$
2) every partition in $G$ is represented by at least two tables in $\tau$.

For example, if $G$ has six partitions, then every table in $\tau$ can only contain attributes representing those six partitions. A table based on partitions outside of $G$ cannot be used. Note also that at least 3 tables must be in $\tau$.

In general, the fewer tables in $\tau$, the more efficient the natural set join. Thus, we want to cover the set of partitions in $G$ (the set of attributes in $T$) with the fewest number of tables.

The NSJ algorithm requires the existence of subgraphs in order to compute the maximal k-cliques. Thus, we assume all bicliques are directly computed by some other means (such as the algorithm in [9]).

D. Preservation of k-Cliques over Subgraphs

While obvious, an important property of k-cliques is the fact a set of nodes completely connected over k-partitions are also completely connected in a subgraph with fewer partitions. We call this property the preservation of k-cliques of subgraphs.

Lemma 1. Given a k-partite graph $G = (\bigcup_{p \in P} p, E)$ and a k-clique $v^k$ on $G$, every j-partite subgraph of $G$ ($2 \leq j \leq k-1$) contains a j-clique $v^j$ such that all of the nodes in $v^k$ are either in $v^j$ or the partitions excluded from the subgraph.

Proof. Obvious from the definition of a k-clique. Given a set of nodes such that every pair is either connected or in the same partition, removing nodes from the set does not change the status of any remaining node.

E. Proof of Completeness

The proof that only maximal k-cliques are generated by the maximal NSJ of an appropriate set of graph tables is straightforward but lengthy. Here we show the more interesting result that all maximal k-cliques are generated,

Theorem 1. Given $G = (\bigcup_{p \in P} p, E)$ and a set of graph tables $T_0, T_1, T_2$ such that each partition in $G$ appears in at least two tables, then $\tau = \max(T_0 \cong T_1 \cong T_2)$ is the graph table for $G$.

Proof. We want to show that every k-clique in $G$ is represented by a row in $\tau$.

We induct over $k$.

Base case: $k = 2$. All maximal bicliques are directly computed, so $\tau$ must exist as a maximal biclique table.

Inductive Hypothesis: All maximal cliques with fewer than $k$ partitions can be computed as stated in the theorem.

WLOG, let the partitions in $G$ be labeled $P_0 \ldots P_{k-1}$ and the corresponding attributes in the graph tables be labeled $A_0 \ldots A_{k-1}$. Furthermore, let $T_0 = (A_0 \ldots A_{k-2})$, $T_1 = (A_1 \ldots A_{k-1})$ and $T_2 = (A_0, A_{k-1})$. Let $v^K$ represent an arbitrary k-clique in $G$.

By the Preservation of k-Cliquess over Subgraphs, there exists a maximal (k-1)-clique $v^{k-1}$ over the partitions $P_0 \ldots P_{k-2}$ such that every node in $v^k$ is in $v^{k-1}$ or in partition $P_{k-1}$. By the inductive hypothesis, we can construct $T_0$ which contains $t_0$ which has all of $v^{k-1}$ (and perhaps additional nodes). Similarly, there must exist $T_1 = (P_1 \ldots P_{k-1})$
and $t_1$. Finally, the biclique table $T_2 = (A_0, A_{k+1})$ contains $t_2$, which contains the nodes of $v^k$ on the attributes $A_0$ and $A_{k-1}$.

Consider $\max(T_1 \triangledown v^k T_1 \triangledown T_2)$ with respect to $t_0, t_1, t_2$. Call this row $t'$. Consider an arbitrary node $v \in v^k$. Assume $v$ is in one of the partitions $P_1 \ldots P_{k-2}$. Then $v$ is in $t_0$ and $t_1$, and as such, is in the intersection. The remaining cases follow analogously. Therefore, $v \in t'.A_v$. So $t'$ subsumes $t$, but since $t$ represents a maximal $k$-clique, $t' = t$.

III. Set Coverage Problem

Given $N$ a set of sets, let $X = \bigcup_N$ be the union of all the sets. A cover $C$ of $X$ is a subset of $N$ such that $\bigcup_C = X$. The Set-Covering Problem is to find the smallest $C$, i.e., the least $|C|$. The problem is discussed in detail in [5], a widely adapted textbook on algorithms, as an example of NP hard problems. This means that no known computer algorithms can solve the problem in less than exponential time. For NP hard problems, we usually look for practical approximate algorithms that run much faster (polynomial time in the size of $N$) and generate solutions that are close to an optimal solution. For the Set-Covering problem, we are looking for a cover that may not be of the minimum size but is not too far above the minimum size. The Set-Covering problem has a lot of applications. A good example is the job scheduling problem. In this case, each element represents a desired job skill and each set represents an employee with the skills in the set. Then a cover would be a group of employees such that, for any required skill, at least one employee in the group has it and the group can complete the job, and a minimum cover would be the smallest group of employees that have all the required skills and can complete the job.

In this paper, we consider a variation of the Set-Covering problem. We call the new variation the Set-Covering problem of degree $D$ and it can be defined as follows:

Given $N$ sets, let $X = \bigcup_N$ be the union of all the sets. A cover $C$ of $X$ of degree $D$, where $D$ is a positive integer, is a subset of $N$ such that every element of $X$ belongs to at least $D$ sets in the group.

The Set-Covering problem of degree $D$ is to find the smallest $C$, i.e., the least $|C|$.

The Set-Covering problem has several variations, but, as far as the principal investigator knows, this one is new and has not been studied before. Clearly, it becomes the original Set-Covering problem when $D = 1$. Thus the Set-Covering problem of degree $D$ is also NP hard and we should look for approximate solutions. Also, in order for a cover of degree $D$ to exist, each element needs to be in at least $D$ sets among the original $N$ sets. For the job scheduling application, a cover of degree $D$ is a group of employees such that, for any desired skill, at least $D$ employees in the group have it. This makes a lot of sense in practice, because we need multiple employees for each required skill in case some of them cannot work for various reasons.

A. Algorithm

In [13], three algorithms are presented for the original Set-Covering problem: Greedy, Check And Remove (CAR), and List And Remove (LAR). Algorithm Greedy produces good results, it runs too slow. Algorithm CAR runs much faster than Greedy, but the resulting cover sets are larger sizes. Algorithm LAR produces smallest cover sets than Greedy and runs at the same speed as CAR, and hence is the best of the three.

We used a doubly linked matrix to implement the three algorithms. In the matrix in Tables II- VII, there are six sets, S1 to S6, and six elements, a to f. A row represents a set and a column represents an element. A one or zero in a cell indicates whether or not the element is in the set. For the Natural Join application, all sets have the same size. That is why each set has three elements. In general, the sets can have different sizes. Each set has an integer variable UncoveredCount for the number of its elements that are not covered yet; each element has a Boolean variable Covered that indicates whether the element is covered or not.

![Table II](image)

In order to modify algorithm LAR to provide a solution to the generalized set covering problem with degree $D$, we change the Boolean variable Covered to an integer variable CoveredCount to indicate how many times the element is covered. There are also two global variables ResultCover and UncoveredCount for the total number of uncovered elements. When a set is added to ResultCover, its UncoveredCount is set to -1. The modified algorithm is in Figure 2.

We now show how the algorithm works on the sample matrix for a degree of 2, assuming the sets are examined sequentially to find the set with the largest UncoveredCount. In the first iteration, S1 is added to ResultCover and its CoveredCount is updated to -1. The CoveredCount becomes one for the three elements in S1 (a, b, and c), but no elements are covered, because the degree is 2. Thus ResultCover = S1, Uncovered remains six and the matrix is in Table III.

During the second iteration, S2 is added to ResultCover, CoveredCount is incremented by one for element b, c, and d, and elements b and c are covered. Then UncoveredCount of S3 is decremented from 3 to 2, since S3 contains element b; UncoveredCount of S4 is also decremented from 3 to 2, since S4 contains element c. Now ResultCover = S1, S2, Uncovered = 4 and the matrix is in Table IV.

In the third iteration, S5 is added to ResultCover, CoveredCount is incremented by one for element a, e and f, and
element a is covered. Then UncoveredCount for S3 and S6 are decremented by one, since they both contain element a. Now ResultCover = S1, S2, S5, Uncovered = 3 and the matrix is in Table V.

In the fourth iteration, S4 is added to ResultCover. CoveredCount is incremented by one for element e with CoveredCount(e) = d then

Remove s from ResultCover
for all e ∈ s do
Decrement CoveredCount(e)
end for
end if
end while

Uncovered = 0 and the matrix is in Table VII. A cover is found.

Notice that the value of UncoveredCount for the sets containing an element is updated only when the value of CoveredCount becomes the same as the degree. The value of CoveredCount can be larger than degree, because the value will be used in the removing phase (the for loop starting on line 28 in the algorithm) to remove redundant sets from ResultCover. However, we will ignore the removing phase here, since it is straightforward.

B. Proof of Correctness

Lemma 2. The value of CoveredCount for each element equals the number of sets in ResultCover that contain the element.

Proof. At the beginning, ResultCover is empty and CoveredCount is zero for each element. The claim is true. During an iteration of the main while loop (lines 9-27), if the execution terminates because no set is selected, then ResultCover does not change and CoveredCount will not change for any elements either. If a set is selected and added to ResultCover during an iteration of the main while loop, then the value of CoveredCount of an element is incremented by one if and only if the set added to ResultCover contains the element. Therefore the claim is still true after each iteration if it is true before the iteration.

Lemma 3. The value of Uncovered is the number of elements that are not covered by the sets in ResultCover.

Proof. At the beginning, ResultCover is empty and Uncovered is the total number of elements from all sets. The claim is true. During an iteration of the main while loop, if the execution terminates because no set is selected, then ResultCover does not change, no additional elements are covered, and Uncovered remains unchanged. If a set is selected and added to
ResultCover during an iteration of the main while loop, then CoveredCount will increment for each element in the set. By Lemma 2, the CoveredCount of an element is the number of sets in ResultCover that contain the element, and hence the element is a newly covered element if the incremented CoveredCount equals the degree $d$. The value of NewCovered is incremented if and only if CoveredCount equals $d$, so the value of NewCovered is the number of elements in the set that become newly covered after the for loop (for all $e$ in $s$). Since the value of Uncovered is decremented by the value of NewCovered after the for loop, the claim is still true after the iteration if it is true before the iteration. Notice that the value of CoveredCount of such elements may increase later, but that will not affect NewCovered and then Uncovered either.

Lemma 4. The value of UncoveredCount for each set is either -1 to indicate it is in ResultCover or the number of elements in the set that are not covered by the sets in ResultCover.

Proof. At the beginning, ResultCover is empty and UncoveredCount is the number of elements in the set for each set. The claim is true. During an iteration of the main while loop, if the execution terminates because no set is selected, then ResultCover does change, and UncoveredCount remains unchanged for each set. If a set is selected and added to ResultCover during an iteration of the main while loop, then CoveredCount will increment for each element in the set, and, if CoveredCount equal $d$, UncoveredCount is decremented by one for all sets that also contain the element. By Lemma 3, UncoveredCount is still the number of elements that are not covered by the sets of ResultCover. Notice that the value of UncoveredCount will never be updated if its value is 0 or -1 at the beginning of the iteration. Therefore, the claim is still true after each iteration if it is true at the beginning.

Theorem 2. Algorithm GLAR will always find a cover if one cover exists.

Proof. Assume a cover exists but algorithm GLAR terminates without finding a cover. By Lemma 2, Uncovered must be positive; otherwise ResultCover is a cover already. The value of UncoveredCount must be -1 or 0 for each set; otherwise, a set should be selected and the execution will not be terminated. For any given cover, one or more sets must have a value of zero for their UncoveredCount; otherwise, they are all in ResultCover, which then must be a cover. For each such set, by Lemma 3, all elements it contains are covered by the sets in ResultCover already. That is, if all such sets are added to ResultCover, Uncovered will remain unchanged. This contradicts the assumption that there exists a cover.

C. Experiments

The time complexity for algorithm GLAR is $O(N^2 + \sum_{i=1}^{N} |S_i|)$, where $N$ is the number of sets. This is the same as LAR. The total time for selecting sets into ResultCover is $O(N^2)$, since UncoveredCount of all sets need to be examined to determine the set to be added in each iteration and the while loop could run $N$ iterations. A row of the matrix is traversed when the set is added to ResultCover to update CoveredCount for all elements in the set, and a column is traversed when the element is covered to update UncoveredCount for all sets containing the element. So the total time for main while loop (lines 9-27) is $O(\sum_{i=1}^{N} |S_i|)$, since the matrix is doubly linked and the cells with a value of zero will not be traversed. The for loop in lines 28-35 traverses the row for each set in ResultCover to determine if the set is redundant, and, if it is, the inner for loop traverses the row one more time to update the CoveredCount. So the total time for the for loop is $O(\sum_{i=1}^{N} |S_i|)$. Note that the degree does not impact the big-O running time, as in the worst case, the algorithm has to examine every set regardless of degree.

We implemented algorithm GLAR and ran the program for the Natural Set Joins application. Algorithm CAR is also modified to GCAR for the purpose of comparison. In this application, there are at most 12 partitions and we could have k-Cliques for k between 2 and 6. We present the case with k being 6, since the result from all other cases are similar. All 924 subsets of size 6 from 12 elements are generated first, then 10, 50, 100, 200, 300, 400, 500, 600, 700, 800 and 900 sets are randomly selected. Since all sets have a very small size of six, both programs completed execution within a small fraction of a second, but GLAR produced covers of smaller or equal sizes in all cases.
We also generated some arbitrary data sets to compare the two algorithms and present the results from two cases.

Case UU5000 has 1000 sets of the same size of 1000, and elements between 0 and 4999 are selected uniformly into the sets. The max possible cover degree is 149. Algorithm GLAR always produces covers of smaller sizes compared with algorithm GCAR, while the running time is slightly longer.

<table>
<thead>
<tr>
<th>Case</th>
<th>Degree</th>
<th>Cover Size</th>
<th>Running Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>GCAR</td>
<td>2</td>
<td>0.83</td>
<td>GLAR</td>
</tr>
<tr>
<td>GCAR</td>
<td>3</td>
<td>0.83</td>
<td>GLAR</td>
</tr>
<tr>
<td>GCAR</td>
<td>5</td>
<td>0.86</td>
<td>GLAR</td>
</tr>
<tr>
<td>GCAR</td>
<td>6</td>
<td>0.86</td>
<td>GLAR</td>
</tr>
</tbody>
</table>

Case NN5000 also has 1000 sets, but the set size is a normal distribution between 800 and 1200, and elements between 0 and 4999 are selected into the sets based on a normal distribution. The max possible cover degree is 10. In this case, the cover sizes from algorithm GLAR are significantly smaller than that from algorithm GCAR, and the running time is shorter.

<table>
<thead>
<tr>
<th>Case</th>
<th>Degree</th>
<th>Cover Size</th>
<th>Running Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>GLAR</td>
<td>2</td>
<td>103</td>
<td>GCAR</td>
</tr>
<tr>
<td>GLAR</td>
<td>3</td>
<td>138</td>
<td>GCAR</td>
</tr>
<tr>
<td>GLAR</td>
<td>5</td>
<td>211</td>
<td>GCAR</td>
</tr>
<tr>
<td>GLAR</td>
<td>6</td>
<td>387</td>
<td>GCAR</td>
</tr>
<tr>
<td>GLAR</td>
<td>10</td>
<td>440</td>
<td>GCAR</td>
</tr>
</tbody>
</table>

Table IX
RESULT FROM CASE UU5000

IV. EXTENSION TO GENEWEAVER

In order to use NSJ with cached results in GeneWeaver, we had to consider two issues. The first is whether or not a cover exists for the desired k-clique, and the second is whether or not NSJ would be faster than computing the k-clique directly. In each case, we considered only relatively high numbers of partitions (7 or more). For smaller number of partitions, it is unlikely the NSJ will outperform direct computation, since the number of k-cliques tends to be much larger with small k.

The first issue is addressed by a series of experiments using our cover algorithm. First, we mathematically determined the smallest number of graph tables that might generate the desired k-clique. Second, we experimented to determine the smallest number of graph tables to give us a different percent chances of having a cover. Tables XI-XIII show the results.

Table XI

<table>
<thead>
<tr>
<th>Case</th>
<th>Degree</th>
<th>Cover Size</th>
<th>Running Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>GLAR</td>
<td>2</td>
<td>103</td>
<td>GCAR</td>
</tr>
<tr>
<td>GLAR</td>
<td>3</td>
<td>138</td>
<td>GCAR</td>
</tr>
<tr>
<td>GLAR</td>
<td>5</td>
<td>211</td>
<td>GCAR</td>
</tr>
<tr>
<td>GLAR</td>
<td>6</td>
<td>387</td>
<td>GCAR</td>
</tr>
<tr>
<td>GLAR</td>
<td>10</td>
<td>440</td>
<td>GCAR</td>
</tr>
</tbody>
</table>

Table X
RESULT FROM CASE NN5000

IV. EXTENSION TO GENEWEAVER

In order to use NSJ with cached results in GeneWeaver, we had to consider two issues. The first is whether or not a cover exists for the desired k-clique, and the second is whether or not NSJ would be faster than computing the k-clique directly. In each case, we considered only relatively high numbers of partitions (7 or more). For smaller number of partitions, it is unlikely the NSJ will outperform direct computation, since the number of k-cliques tends to be much larger with small k.

These experiments indicate the probability of computing the maximal k-clique from cached results is very high, even when only a few graph tables are available.
We next conducted experiments to determine the transition point between directly computing the maximal k-cliques and using the cached results. Each experiment constructed the underlying bipartite graphs by using a fixed number of nodes per partition and a consistent density of edges between each partition. Different experiments varied these values. A given k-partite graph is constructed by taking the union of the bipartite graphs needed to form the graph. For example, to construct the 4-partite graph 0-1-2-3, we took the union of the edges in the six bipartite graphs 0-1, 0-2, 0-3, 1-2, 1-3 and 2-3. We assumed a maximum of 12 partitions based on usage from GeneWeaver. Due to the exponential cost in finding maximal k-cliques, we kept the total number of nodes relatively small. Larger scale parallel experiments are part of future work.

All of the performance experiments followed a pattern similar to the results in Table XIV. The worst case performance for maximal clique problem is $O(3^n)$ where $n$ is the size of the graph. The exponential runtime is due to the exponential number of maximal cliques that may exist. Table XV shows the number of k-cliques found with different sizes of partitions and edge densities. Since the cached tables contain one row for each maximal k-clique, the cached tables can be exponentially larger than the graphs which generate them. Since the nested join operation is $O(n^2)$ where $n$ is the number of rows in the table, NSJ requires far more time than direct computation until the number of maximal k-cliques falls below a threshold.

<table>
<thead>
<tr>
<th>Parts</th>
<th>Graph</th>
<th>1200 @ 10%</th>
<th>1200 @ 20%</th>
<th>300 @ 30%</th>
<th>300 @ 40%</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>202,672</td>
<td>473,318,312</td>
<td>494,265</td>
<td>5,645,603,473</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>2831</td>
<td>80,810,630</td>
<td>10,317,222</td>
<td>3,577,085,465</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>3</td>
<td>6,157,021</td>
<td>1,676,103</td>
<td>1,483,504,894</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>0</td>
<td>108,518</td>
<td>120,421</td>
<td>402,225,089</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>0</td>
<td>344</td>
<td>2625</td>
<td>67,517,146</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>0</td>
<td>21</td>
<td>5,800,003</td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>202,467</td>
<td></td>
</tr>
</tbody>
</table>

Table XIV
COMPARISON OF USING THE NATURAL SET JOIN VERSUS DIRECT COMPUTATION OF K-CLIQUES. THE ROWS AND COLUMNS ARE DEFINED THE SAME AS IN TABLE XI. AN ENTRY OF D INDICATES A DIRECT COMPUTATION IS FASTER, WHILE AN ENTRY OF N INDICATES THE NATURAL SET JOIN IS FASTER. IN ALL CASES, THE OPTIMAL NUMBER OF SUBSETS IN TABLE XI ARE USED. EACH PARTITION IS ASSUMED TO HAVE 1200 NODES AND THERE IS A UNIFORM 10% CHANCE FOR ANY PAIR OF NODES TO BE CONNECTED.

<table>
<thead>
<tr>
<th>Parts</th>
<th>Graph</th>
<th>1200 @ 10%</th>
<th>1200 @ 20%</th>
<th>300 @ 30%</th>
<th>300 @ 40%</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>117</td>
<td>18,357</td>
<td>288</td>
<td>19,891</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>130</td>
<td>22,386</td>
<td>391</td>
<td>28,900</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>153</td>
<td>25,576</td>
<td>521</td>
<td>41,077</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>177</td>
<td>30,682</td>
<td>656</td>
<td>56,822</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>206</td>
<td>36,112</td>
<td>848</td>
<td>76,627</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>239</td>
<td>43,888</td>
<td>1085</td>
<td>99,839</td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>275</td>
<td>51,014</td>
<td>1570</td>
<td>129,501</td>
<td></td>
</tr>
</tbody>
</table>

Table XVI
TIME REQUIRED (IN SECONDS) TO COMPUTE THE CORRESPONDING MAXIMAL K-CLIQUE IN TABLE XV

Since the optimal case uses three cached tables of size $p$ to find the maximal k-cliques of size $p+1$, we use $s = 3$ for our results in Table XVII. Again, due to the exponential size of the cached tables, $N_p^s$ is the dominant term in the inequality. As such, when the cached tables reach the specified threshold, the NSJ algorithm becomes the faster implementation, usually by multiple orders of magnitude. However, when the cached tables are large, the NSJ algorithm can be slower by orders of magnitude as well.

Table XV
SPACE REQUIRED (NUMBER OF MAXIMAL K-CLIQUE) FOR A GRAPH TABLE WITH THE LISTED PARTITION SIZE AND EDGE DENSITY. DUE TO THE LONG TIME REQUIRED TO FIND THE MAXIMAL K-CLIQUE, ONLY ONE INSTANCE OF EACH K-PARTITE GRAPH IS PRESENTED.

The cost for doing a NSJ is based on the number of rows (maximal k-cliques) in the participating graph tables. For a given k-partite graph, we use the priority set coverage algorithm to find the tables needed to produce the results. Currently, we use the straightforward nested loop algorithm to compute NSJ, so the total number of intersections required is the product of the number of rows in the tables times $k$.

Table XVII indicates the stark contrast between using cached data and using direct computation. This means the decision to cache results is obvious with clearly defined benefits for the correct choice. Additionally, the division between cached and direct computation is so strong, an order of magnitude improvement in performance is unlikely to change the decision to cache data.

V. RELATED WORK

The Set Cover problem arises from many applications and has been studied in different scenarios. But as far as we know, the variation of the set cover problem considered in this paper has never been studied before.
Berger et al. [2] present a parallel algorithm for the set cover problem that performs well on arbitrary instances in terms of cover size, matching the sequential greed algorithm. A variation of the set cover problem, maximum coverage, has been studied in different settings [4], [10]. The problem requires a specified number of sets to be selected that maximize the total weight of the covered elements. Cormode et al. [6] study the set cover problem for very large datasets, focusing on improving disk access operation.

GeneWeaver uses the maximal k-clique enumeration algorithm from [9]. We know of only two other maximal k-clique enumeration algorithms. In [8], the concept of maximal consistent subsets can be mapped to maximal k-cliques. The CLICKS algorithm in [14] uses two stages to find clusters of categorized data.

### VI. Conclusion

The natural set join provides one opportunity for enhancing GeneWeaver by using cached results. GLAR is a new algorithm to determine the set of cached tables. The algorithm is developed for a variation of the classical set-covering problem with a degree parameter. As far as we know, this is the first time the variation is defined and being studied.

There are several directions for this research in the future. First, cached results can be modified in alternative ways. For example, the presence of an edge is frequently determined by a user-defined threshold. If the number of k-cliques is too large, a user may raise the threshold, thereby eliminating edges. If none of the edges in a k-clique are removed, then the k-clique would be preserved. A simple sequential scan of the graph table could indicate the number of k-cliques preserved. Of course, removing an edge will create new k-cliques. Efficiently finding all of the new k-cliques is ongoing work.

Second, the natural set join can easily leverage the advantages of a Hadoop-style environment, while graph algorithms, such as Bron-Kerbosch algorithm used in the current GeneWeaver implementation, are more difficult to adapt. For example, the naïve fragment and replicate algorithm for computing joins [7] can be naturally extended to implement the natural set join to achieve near linear speed up [11]. Comparisons with algorithms such as [12], which also achieves near-linear speedup, are required to determine if the caching constraints change in Big Data environments.

Additionally, natural set join performance can be improved. Currently, there are no strategies for eliminating unnecessary computations or short-circuit evaluations. Computing the intersections of attributes with a few elements (or 1) is both fast and likely to create an empty set. If an empty set is generated, the computation should proceed to the next row. Likewise, sorting the graph tables and performing a merge join [3] could outperform the simple nested-loop strategy used here.

Finally, given the cached tables, it may be possible to develop approximate algorithms which perform significantly faster. For example, a threshold for the number of single attribute partitions could be used to eliminate rows with low probability of intersecting with two other tables. Determining the statistics needed to bound the approximation error is ongoing work.

### Acknowledgment

This work was supported, in part, by NIH R01 AA18776, jointly funded by the NIAAA and NIDA.

### References


