Large Maximal Cliques Enumeration in Large Sparse Graphs

Natwar Modani

IBM India Research Lab
Vasant Kunj Institutional Area
New Delhi
India
namodani@in.ibm.com

Kuntal Dey

IBM India Research Lab
Vasant Kunj Institutional Area
New Delhi
India
kuntadey@in.ibm.com

Abstract

Identifying communities in social networks is a problem of great interest. One popular type of community is where every member of the community knows all others, which can be viewed as a clique in the graph representing the social network. In several real life situations, finding small cliques may not be interesting as they are large in number and low in information content. Hence, in this paper, we propose a variant of maximal clique enumeration problem where we try to enumerate only large maximal cliques. We describe a novel preprocessing technique to reduce the graph size before enumerating the large maximal cliques. This is of great practical interest since enumerating maximal cliques is a computationally hard problem and the execution time increases rapidly with the input size. We also present a new maximal clique enumeration algorithm SELMaC2, which exploits the constraint on minimum size of the desired maximal cliques. We present experimental results on several real life social networks. Our results show that the preprocessing methods achieve significant reduction in the graph size. Also our algorithm has fewer intermediate steps and is faster than the competing algorithms adapted from the literature by incorporating the minimum size criterion. Our results also show the scalability of our approach.

1 Introduction

Finding communities is one of the fundamental problems in social network analysis [4, 7, 9, 5, 6]. Social networks found to form in real life come from different domains, and understanding the communities within these social networks is often of interest to different parties in more than one context. Some example social networks are formed by people to people interaction (online or telephonic) and website link farms. In social networks formed by people to people interaction, communities often represent people with shared interest (including friendship or relations). An example of such social network in the online world is based on email and instant messaging. Another such example is the social network formed by telephonic calling patterns of people [15]. Link farm communities [8] are often formed to artificially inflate the page ranks of the participating websites, and they intend to promote the whole community of websites through mutual recommendation.

Various definitions have been proposed for communities in a social network. One interesting type of community in any social network is where every member of the group *knows* every other member of the group. If one thinks of a person in the social network as a vertex and a relation between two persons (*knows*) as an edge, then a social network can be viewed as a graph. We will concentrate on the social networks where the relation is bidirectional (i.e., if *a knows b*, then *b* also *knows a*) and between any pair of people there can be at most one relation. This would result in a simple undirected graph describing the social network. One can think of the communities mentioned above as cliques in the graph which describes the social network.

Enumerating maximal cliques is a well-studied problem [10, 16, 20, 2, 12]. Cliques have been viewed as one type of community in telecom social networks [15, 1]. The graphs in such applications are very large and sparse and the number of maximal cliques increase exponentially as the clique size decreases. The information content or value of a smaller clique is obviously less than a larger clique. Hence finding small maximal cliques may not yield enough value to justify the effort. Motivated by these real-life problems, we study a variant of the maximal clique enumeration problem by incorporating a minimum clique size criterion. To the best of our knowledge, this is the first

15th International Conference on Management of Data
COMAD 2009, Mysore, India, December 9–12, 2009
©Computer Society of India, 2009

systematic study for this variation of the problem.

Our approach is especially suited for sparse graphs. The minimum size criterion allows us to speed up the maximal clique enumeration by enabling preprocessing to reduce the graph size. Our novel filtering criterion uses the observation that if the two vertices on the two ends of an edge are to be in the same clique of size at least L, then they must share at least L vertices in their respective neighbourhoods[1]. Additionally, if a vertex is to participate in a clique of size at least L, then it must share at least L neighbours with at least $L-1$ of its other neighbours.

The new clique finding algorithm SEL*MaC*2 (which stands for *S*equential *E*xploration for *L*arge *Ma*ximal *C*liques with *Ma*ximality *C*onditions) attaches conditions to every candidate which must be satisfied in order for it to be a large maximal clique. These conditions allow us to reuse computations performed at one stage to all further stages by successively refining the critia for the candidate to be a maximal clique and thereby improving the run time of the algorithm. Also, the execution behaviour of our algorithm does not depend heavily on the presence of a few nodes with large degree.

Our experiments show that the preprocessing method we propose achieves significant reduction in the graph size and that our algorithm SEL*MaC*2 has fewer intermediate steps and is faster than the competing algorithms. To the best of our knowledge, the prior art attempts to perform maximal clique enumeration on graphs with several thousand nodes only. In contrast, our approach finds all maximal cliques of size 16 and above (1,291 in number) in a graph with ~ 7.75 million vertices and ~ 37 million edges in ~ 12 minutes and all maximal cliques of size 8 and above (2,520 in number) in a graph with ~ 2.2 million vertices and ~ 3.55 million edges in less than a minute.

Our main contributions in this paper are the following:

1. We pose the problem of community finding as a variant of maximal clique enumeration problem by introducing a minimum size criterion.

2. We present a novel preprocessing technique that exploits the desired minimum clique size and the extent of overlap in the neighbourhood of adjacent vertices to filter the graph before enumerating maximal cliques.

3. We present SEL*MaC*2, a novel clique enumeration algorithm, which is specially suitable for large maximal clique enumeration.

4. We present a graph filtering technique to reduce the graph size by exploiting the minimum size criterion without having to load the entire graph in memory.

5. We analyze several real life social networks formed by call detail records of one of the world's largest telecom service providers.

[1]where the neighbourhood of a node v is defined as v and all nodes that are adjacent to it

The rest of the paper is organized as follows. In Section 2 we define the problem formally and introduce some notations. In Section 3, we present our preprocessing and large maximal clique enumeration algorithm. We contrast our approach with the two other approaches in the literature which are most relevant to the problem at hand in Section 5 while surveying the related work. In Section 6, we describe our experiments and discuss our results and also compare with other approaches. Finally, we conclude in Section 7.

2 Problem Definition

Let $G = (V, E)$ be a simple undirected graph, where V is the set of vertices and E is the set of edges. Let K be a subset of vertices V. K is called as a clique if $\forall u, v \in K$, $(u, v) \in E$. Further, if there is no set $K' \subset V$ such that K' is a clique and $K \subset K'$ then K is called as a maximal clique. We will denote the size of a set of vertices K by $|K|$. We let the user specify a threshold L such that the user is interested only in the maximal cliques K such that $|K| \geq L$.

Large Maximal Clique Enumeration Problem: Given a simple undirected graph $G(V, E)$, enumerate all maximal cliques of size at least L.

We define $\Gamma(v)$, the neighbourhood of a node v, as

$$\Gamma(v) = \{v\} \cup \{u : (u, v) \in E\} \tag{1}$$

Let us define $\delta(v) = |\Gamma(v)|$ and $\delta_{max} = \max_{v \in V} \delta(v)$. Also denote the number of nodes with neighbourhood size exactly δ as n_δ. Clearly, $\Sigma_{\delta=0}^{\delta_{max}}\ n_\delta = |V|$ and $\Sigma_{\delta=1}^{\delta_{max}}\ (\delta - 1) * n_\delta = 2 * |E|$. We define a graph parameter Δ as $\Delta = max(\delta)$ s.t. $\Sigma_{j=\delta}^{\delta_{max}}\ n_j \geq \delta$. Here Δ is the largest maximal clique size that can be supported by a graph with the given degree distribution.

3 Algorithm

Now we describe our algorithm to enumerate large maximal cliques in a very large graph. The computational effort required to find the maximal cliques increases very quickly with the graph size, hence we first attempt to filter the graph to reduce the graph size. We achieve this goal based on the threshold parameter supplied by the user. Ours is a three phase approach. In the first phase, we try to reduce the graph size without loading the entire graph in main memory. After this, we apply another level of filtering on the filtered graph obtained from the first phase which reduces the graph size further. Finally on this smaller graph, we enumerate the large maximal cliques. Note that each

```
DegreeBasedFilter (infile, outfile, $L$, $D_p$){
    init $D_c = \emptyset$
    $(\forall v \in V)${
        read a node $v$ and its neighbourhood $\Gamma(v)$
            from the infile,
        $\gamma(v) = \Gamma(v) - (D_c \cup D_p)$
        if $(|\gamma(v)| < L)$ then $D_c = D_c \cup \{v\}$
        else write $v, \gamma(v)$ to outfile
    }
    // if a node was dropped in this iteration make
        a
    // recursive call with input and output files
        switched
    if $(D_c \neq \emptyset)$
        DegreeBasedFilter(outfile, infile, $L$, $D_c$)
}
```

Figure 1: Algorithm *DBFilter*: Degree Based Filtering Algorithm.

of the preprocessing techniques and clique enumeration algorithm can be used independent of each other.

3.1 Preprocessing

The first part of our algorithm uses the observation that a node $v \in V$ cannot participate in a clique of size L unless $\delta(v) \geq L$. Once we drop the nodes that do not satisfy the neighbourhood size criterion, we may have some vertices which satisfied the criterion earlier, but do not satisfy it any more. Hence, we apply this criterion recursively on the resulting graph as long as we can drop some nodes from the graph. We refer to this procedure as degree based filtering. Some times the input graph may be too big to hold in the memory in order to perform the degree based filtering, especially with programming languages like Java. In Figure 1, we present a recursive algorithm which will require at most $|V| + \delta_{max}$ amount of random access memory and two files (i.e., sequential access storage) to perform the degree based filtering. We assume that the input graph is given in form of neighbourhood list, i.e., each row in the input represents a node and all its neighbours including v itself.

Let D_p denote the set of vertices dropped in previous iteration and D_c denote the set of vertices dropped in current iteration. The first call to the degree based filtering routine is made with $D_p = \emptyset$. This preprocessing is similar to filtering technique, termed as peeling, introduced in [1] to reduce the graph size while attempting to find large cliques in a social network. However, our technique can handle much larger graphs as we do not require to store the entire graph in memory. Some points worth noting in this algorithm are:

```
SharedNeighboursFilter $(G(V, E), L)${
    initialize droppedSomeNodeOrEdge = false
    $(\forall v \in V)${
        $\forall u \in \Gamma(v)${
            if $(|\Gamma(v) \cap \Gamma(u)| < L)${
                $E = E - \{(u, v)\}$
                droppedSomeNodeOrEdge = true
            }
        }
        if$(|\Gamma(v)| < L)$ {
            $V = V - v$
            $E = E - \{(w, v)\ \forall w \in \Gamma(v)\}$
            droppedSomeNodeOrEdge = true
    }
    if (droppedSomeNodeOrEdge)
        SharedNeighboursFilter $(G(V, E), L)$
}
```

Figure 2: Algorithm *SNNFilter*: Shared Neighbourhood Filtering Algorithm.

- It takes at most $|V| + \delta_{max}$ amount of random access memory ($\leq |V|$ amount of memory to hold the dropped vertices, and $\leq \delta_{max}$ to hold the neighbourhood of a vertex while processing it) irrespective of $|E|$.
- We need to keep track of the nodes dropped only for two level of recursion, hence, practically the memory requirement is even less.
- Each vertex in the filtered graph has a minimum neighbourhood size of L in the filtered graph, i.e., the filtered graph consists of k-cores [18] with $k = L$.

The next step is to filter the graph further based on the following criteria. In order for an edge $e = (u, v)$ to participate in a clique of size L, the nodes on the two sides of the edge (u, v) must share at least L neighbours in their neighbourhood, i.e., $|\Gamma(u) \cap \Gamma(v)| \geq L$. Furthermore, if a vertex v is to participate in a clique of size $\geq L$, it must have at least $L - 1$ neighbours u (besides itself), such that $|\Gamma(v) \cap \Gamma(u)| \geq L$.

We use these criteria to first eliminate edges and then to eliminate the vertices which cannot participate in a clique of size at least L. Figure 2 shows the algorithm for the same, which we refer to as *Shared Neighbourhood Filtering* (*SNNFiltering* for brevity). This is a far stricter criterion than the degree based filtering and results in significant reduction in the graph size.

Note that while the degree based filtering results in a induced subgraph of the remaining vertices, in shared neighbourhood filtering, some of the edges between the surviving nodes may also be dropped if they cannot participate in the same maximal clique.

3.2 Large Maximal Clique Enumeration

Now, we describe our algorithm SELMaC2 to find all maximal cliques of size at least L. The guiding thought behind the algorithm is that we expect potentially a very large but fairly sparse graph and do not expect a huge number of large maximal cliques. This fact allows us to explore the graph vertices in a sequential manner as against the depth-first technique used in [10, 16]. We assume the input graph to be in the form of neighbourhood list and process these neighbourhood lists one by one. SELMaC2 does not presume any specific ordering of node arrival.

A clique is a subset of neighbourhoods of the member nodes. Hence, we start with the neighbourhood of a node, say v, as a candidate for being clique. Consider another node u which is a neighbor of v. If both of them participate in a clique, then the clique will be a subset of the intersection of the neighbourhoods of u and v. On the other hand, a clique may only contain v and not u or vice versa. We need to identify cliques of all three types while processing the vertices. As we are interested in only large maximal cliques, we retain only those candidates for further processing which may lead to large maximal cliques.

We introduce some more notations used in our algorithm. Let U be the set of nodes already processed. We define a *candidate* as a set of vertices which may form a large maximal clique and denote it by c. The candidate may also have associated maximality conditions discussed below. The set of all candidates is denoted by C. A candidate c consists of confirmed node set F_c and alive node set A_c. F_c is a subset of U and is a clique by itself. A_c is the set of nodes not yet processed but are neighbours of all the nodes in F_c.

To discard a candidate which may not give rise to a maximal clique, we make the following observation. Exactly one of the two conditions will be satisfied by any given maximal clique K and any vertex v:

1. $v \in K$
2. $\exists u : u \in K$ and $u \notin \Gamma(v)$

It is easy to see that if both are true, then K will not be a clique. On the other hand if both are false, K will not be maximal since $K \cup v$ will also be a clique. To exploit this observation, whenever we drop a vertex from a candidate, we attach a list of vertices that are not neighbours of the dropped vertex (amongst the candidate member vertices) as a taboo list to this candidate. If all the nodes in a taboo list are dropped, then the above given observation implies that this candidate cannot evolve into a maximal clique.

Our algorithm is presented in Figure 3. This algorithm finds all the maximal cliques that are at least L in size. In our algorithm, when we process a node v, and are considering the candidate c, if $v \in c$, we create two candidates, one which includes v, and other which does not. For the second candidate, we attach a new taboo list λ as a list of vertices that are not neighbours of v amongst A_c. If at least one of these nodes is not part of the candidate at any given point, the candidate cannot lead to a maximal clique as noted above. We attach one such taboo list for every such node which we do not include in the candidate. We denote the set of taboo lists attached to a candidate c as Λ_c. Another possibility is that $\Gamma(v)$ contains a maximal clique which does not include any of the nodes which have been processed up to now. To cover this possibility, we create a new candidate with $A = \Gamma(v) - U - \{v\}$ and $F = \{v\}$ and attach taboo list for each node in $U \cap \Gamma(v)$.

When we include a node from A into F, we may drop some vertices in A (as they are not neighbour of nodes included in F_c). We drop such vertices from the taboo lists also. On the other hand, if the vertex that is included from A to F is a member of the taboo list, then the taboo list is satisfied and we discard that taboo list. Similarly, when we choose to drop a node, we again remove that vertex from the taboo lists for the candidate. If at any stage, any of the taboo lists for a given candidates become empty, then we drop the candidate as it cannot lead to a maximal clique. If a taboo list is so large that its violation will lead to the candidate becoming smaller than desired size L, then we don't need to keep this taboo list. This optimization cannot be performed easily in any other clique enumeration algorithm. Also, if a taboo list, say λ_i, is a superset of another, say λ_j, we can drop λ_i since if a node from λ_j is present in the candidate, clearly a node from λ_i is present in the candidate and hence λ_i is satisfied.

4 Correctness and Complexity

Now we will give informal arguments to show the correctness of our algorithm and provide a bound its complexity.

4.1 Correctness

First, it is easy to see that the filtering schemes do not drop any edge or vertex which may participate in a maximal clique of desired size. Hence, it does not affect the correctness of the overall approach.

Now, we will show that at the end of the algorithm, the candidate set C corresponds to the set of all maximal cliques of size at least L, and that none of the maximal cliques of size at least L are missing from it.

For the purpose of this section, we will denote the confirmed and alive set together by the candidate name, i.e., c would mean $F_c \cup A_c$. Observe that

```
findLargeMaximalClique(graph $G(V, E)$, size threshold $L$){

    init $C = \emptyset$, $U = \emptyset$
    $(\forall v \in V)$\{
        // Create a candidate $\check{c}$ from $\Gamma(v)$ without the vertices already considered
        create a new candidate $\check{c}$ as follows
        $F_{\check{c}} = \{v\}$, $A_{\check{c}} = \Gamma(v) - \{v\} - U$ and attach taboo lists defined as
        for $(\forall w \in U \cap \Gamma(v))$
            create a taboo list $\lambda_w = A_{\check{c}} - \Gamma(w)$
        if (processCandidateTabooLists ($\check{c}$) = *true*) then $C = C \cup \{\check{c}\}$
        $U = U \cup \{v\}$
        $(\forall c \in C)$\{
            if $(v \in A_c)$\{
                $C = C - \{c\}$

                // Create a candidate $\hat{c}$ with intersection of $c$ and $\Gamma(v)$
                create a new candidate $\hat{c}$ with:
                $F_{\hat{c}} = F_c \cup \{v\}$, $A_{\hat{c}} = (A_c - \{v\}) \cap \Gamma(v)$ and $\Lambda_{\hat{c}} = \Lambda_c$
                if (processCandidateTabooLists ($\hat{c}$) = *true*) then $C = C \cup \{\hat{c}\}$

                if$(A_c \not\subset \Gamma(v))$ \{
                  // Create a candidate $\tilde{c}$ from $c$ without the new vertex $v$
                  create a new candidate $\tilde{c}$ with:
                  $F_{\tilde{c}} = F_c, A_{\tilde{c}} = A_c - \{v\}$ and $\Lambda_{\tilde{c}} = \Lambda_c$
                  add a new taboo list $\lambda = A_c - \Gamma(v)$
                  if (processCandidateTabooLists ($\tilde{c}$) = *true*) then $C = C \cup \{\tilde{c}\}$
                }
            } // end if $v \in A_c$
        }
    }

}

processCandidateTabooLists (candidate $c$) {

    for each taboo list $\lambda_i(c) \in \Lambda_c$ {
        $\lambda_i(c) = \lambda_i(c) \cap A_c$
        if $((\lambda_i \cap F_c \neq \emptyset)$ OR $(|\lambda_i| > |F_{\hat{c}}| + |A_{\hat{c}}| - L)$ OR $(\exists \lambda_j \in \Lambda_c \; s.t. \; \lambda_i \supset \lambda_j))$
            discard $\lambda_i$
        if $((|\lambda_i| = 0)$ OR $(|F_c \cup A_c| < L))$
            return *false* // this candidate cannot lead to a maximal clique of size$\geq L$
    }
    return *true* // this candidate *may* lead to a maximal clique of size$\geq L$

}
```

Figure 3: SEL*MaC*2: Our Large Maximal Clique Enumeration Algorithm.

if a vertex set $K = \{k_1, k_2, \ldots, k_r\}$ is a clique then $K \subseteq \cap_{k_i \in K} \Gamma(k_i)$. Now, if K is a maximal clique, then no other vertex can be a neighbor of all the vertices included in K, i.e.,

$$K = \bigcap_{k_i \in K} \Gamma(k_i) = \bigcap_{k_i \in K} \Gamma(k_i) - (V - K)$$

Hence, if K is a maximal clique, it will be of the form $K = \cap_{k_i \in K} \Gamma(k_i) - (V - K)$, and if $K = \cap_{k_i \in K} \Gamma(k_i) - (V - K)$, then K will be a clique.

Let U be the set of vertices processed up to now. Then, it is easy to see that the candidates in our algorithm are of the form

$$c = \bigcap_{p \in P} \Gamma(p) - Q$$

where (P, Q) form a partition of U. Hence, when $U = V$, each candidate will be a clique if it is not empty. Since we are checking for maximality using the taboo lists, we would be discarding the non-maximal cliques. Hence, the output of the algorithm is a set of maximal cliques.

Now, we want to show that we will not miss any maximal clique of size $\geq L$. To show this, note that we drop the candidates based on two conditions.

1. The size of the candidate has become less than L, i.e., $|F_c \cup A_c| < L$. Clearly, this candidate cannot lead to maximal clique of size $\geq L$.

2. One of the taboo lists has become empty for the candidate c. Let us say this taboo list is created due to dropping the node u. This implies that in current situation, $F_c \cup A_c - \Gamma(u) = \emptyset$. Hence, we can bring back u, and form a clique which will have the clique resulting from processing c and u. Hence, clearly the clique found by processing c will not be maximal.

We retain all the other candidates of the form $c = \bigcap_{p \in P} \Gamma(p) - Q$ where (P, Q) form a partition of U. Hence, we will not miss any maximal clique of size $\geq L$.

4.2 Complexity

Now, we will show the time complexity of our algorithm. For a vertex and a candidate, we need to check for the existence of the vertex in the candidate. If the vertex is a part of the candidate, then we need to perform an intersection. The checking for existence of an element in a set can be performed in constant time (with a hashset based implementation), and the intersection can be performed in time linear in the size of the smaller set.

To process the taboo lists of a candidate, we require time that is product of the maximum taboo list size and number of taboo lists for that candidate. It is easy to see that since no more than $\delta_{max} - L$ number of nodes can be dropped to form a candidate (otherwise the candidate becomes too small), and each taboo list is bounded in size by $\delta_{max} - L$ (since if the list is any longer then it cannot be violated without reducing the size of the clique below the desired limit). Hence the bound on time to process one candidate for a given vertex is $O((\delta_{max} - L)^2)$, and the overall complexity of our clique enumeration algorithm is:

$$O(|V| * |C| * (\delta_{max} - L)^2) \tag{2}$$

Since the complexity of our algorithms depends on the number of candidates, we now compute the maximum number of candidates. Consider any existing candidate. When it is processed with another node in its alive node set, there are two possibilities. The first possibility is that it is split in to two candidates, one which contains the next node and the one which does not. If the candidate which contains the node retains all the members of the original node than the other candidate which does not contain the node, will be a subset of the first candidate. Hence we will end up with the same candidate after processing the node. On the other hand, if the candidate containing the node does not have all members of the parent candidate, then the size of both the candidates will be at least 1 less than the original candidate. Also note that when a candidate contains multiple vertices in the confirmed part, the size of the candidate is bounded from above by the smallest neighbourhood size. Now, by definition, there are less than Δ nodes with neighbourhood size more than Δ. If they interact with nodes with neighbourhood size $\leq \Delta$, then they do not determine the number of candidates. Hence, to find the worst case, we assume that they do not interact with nodes with neighbourhood size $\leq \Delta$. From such nodes (of neighbourhood size $> \Delta$), we can get at most 2^{Δ} candidates (which will happen when there is one starting point and rest of the nodes cause splits).

For convenience, we will denote the nodes with a neighbourhood size $> \Delta$ as V_L and the other nodes as V_S. If we consider the candidates which start from nodes in V_S, the candidates can undergo at most $\Delta - L$ splits (after that it becomes too small to be interesting). Hence, from a candidate of starting size $\leq \Delta$, we can get up to $2^{(\Delta - L)}$ candidates. To compute the maximum number of candidates, now we only need to compute how many starting candidates of size $\leq \Delta$ we can have. But each node can at most start one new candidate, hence the new starting candidates can be at most $|V|$. Hence, the maximum number of candidates can be:

$$|C| = O(2^{\Delta} + |V| 2^{(\Delta - L)}) \tag{3}$$

In sparse graphs with $|E| \sim O(|V|)$, it is easy to see that $\Delta \leq \sqrt{|E|} \Rightarrow \Delta \sim O(\sqrt{|V|})$. Hence, the number of candidates in sparse graphs are $O(2^{\sqrt{|V|}} +$

$|V|2^{(\sqrt{|V|}-L)})$, and the overall complexity of our algorithm is $O(|V|*(2^{\sqrt{|V|}}+|V|2^{(\sqrt{|V|}-L)})*(\delta_{max}-L)^2)$.

5 Comparison And Related Work

Finding communities is one of the fundamental problems in social network analysis [4, 7, 9, 5, 6]. Various definitions have been proposed for communities in a social network [11, 17, 3] including cliques. [15, 1] attempt to enumerate maximal cliques in social networks formed by telecom interactions. In [1], the authors also present a filtering technique, termed as peeling, to reduce the graph size while attempting to find large cliques in a social network. Peeling is similar to *DB-Filter* algorithm presented in this paper. However, our technique can handle much larger graphs as we do not require to store the entire graph in memory. Besides this, the authors are not aware of any graph filtering technique for clique enumeration. One might attempt to use clustering coefficients, defined in [21], as the ratio of number of triangles in the immediate neighbourhood of the node divided by the number of possible triangles based on the degree of the vertex. However it cannot be used to filter the graph meaningfully while not affecting any interesting maximal cliques.

Maximal clique enumeration is a well known and well studied problem [10, 16, 20, 2, 12, 14]. There are primarily two lines of work for this problem, of which one guarantees a run time that is polynomial in output size and the other trades this guarantee for fast running time for practical problems. Some examples of output polynomial solutions are [20, 12, 19] which are amongst theoretically the best possible results. These methods are based on augmentation of a clique in such a way that one would avoid going in direction of a non-maximal clique, making it hard to know if a clique under construction is likely to have the required minimum size until very late. Hence it is hard to effectively exploit the largeness constraint of the interesting maximal cliques.

The other line of work utilizes neighbourhood based approaches, starting with a large sized vertex set and iteratively or recursively pruning it to a maximal clique. This type of approach is explored in [10, 16]. It is hard to provide output polynomial guarantee in these approaches. However, it is relatively easier to exploit the restriction on the minimum size of the clique to speed up the computations. Our work falls in this category. A sketch of some of the ideas discussed in this paper were presented in [13] as a poster.

In our experiments, we compare SELMaC2 with [16, 10]. To provide a fair comparison, we modified their approaches to exploit the minimum size criteria in our experiments. In essence, the approaches of these and our algorithm are similar and they essentially differ in the bounding condition. We can consider the approach as constructing a binary decision tree, with thinking of including a vertex as a left child of a given node, and dropping a vertex as right child. However, unlike the other two approaches which perform a depth first search for maximal cliques, we explore the graph vertices in a sequential fashion.

The approach described in [16] takes a neighbourhood and decides to include or drop one vertex at a time to construct a clique, deferring the check about its maximality until the clique is formed. Then it is compared with all the cliques found to the left of it in the decision tree to make sure no clique found left of it is a superset of this. Here, the operations at each intermediate node is relatively inexpensive, but it leads to a large number of leaf nodes which need to be checked against other leaf nodes to the left of them. Also, in order to check the maximality, one needs to keep all the cliques found in memory, in addition to the graph, and hence it is a memory intensive algorithm.

The approach described in [10] also takes a neighbourhood and decides to include or drop one vertex at a time to construct a clique. At each node, it checks if any of the dropped nodes can be brought back to extend this clique. This check ensures that if a candidate is leading to a non-maximal clique, one would be able to detect it soon and save needless computations. However, the processing at intermediate nodes becomes more expensive. Also, one has to keep checking at each intermediate node on one path for a node that is dropped. This may lead to repetitive computation.

In our approach, we construct a binary decision forest as against a single decision binary tree in the approaches mentioned above. At each stage, we want to ensure that we stop as soon as we know if the candidate leads to a non-maximal clique. For this purpose, we maintain the 'taboo lists' as described in Section 3. We only need to update the taboo lists to determine the maximality possibility. Here, the work we do in one iteration is useful in all the nodes below it too, as the taboo lists percolate down the tree. The reduction in execution time is somewhat accompanied by increase in space requirement as we need to maintain the taboo list for each dropped node. However, we have presented some checks in the algorithm which can reduce the amount of space taken by these lists. Also, since we are largely concerned with very sparse graphs (with say maximum degree of δ_{max}), the maximum amount of space we may need for a candidate's taboo lists is upper bounded by $(\delta_{max} - L)^2$.

Cliques may sometimes be too restrictive a definition [11] for communities in social networks. Several clique relaxations have been proposed in literature, notably quasi-cliques, k-cores [18], k-cliques [11], k-club and k-plex [17]. However, some of these definitions are too difficult to compute while some others do not have the desired properties of a social community.

DataSet	Region	Call / SMS	# CDRs	Vertices	Edges	δ_{max}	Δ	Max Clique Size
DS1	Mixed	SMS	46,401,569	2,212,016	3,555,345	3,796	183	13
DS2	Urban	SMS	87,333,241	4,769,284	10,811,702	44,554	309	15
DS3	Mixed	Call	195,871,787	5,806,784	28,951,874	4,623	646	19
DS4	Urban	Call	386,692,792	7,747,671	37,271,744	5,212	1,080	20

Table 1: Some statistics about the datasets

6 Experiments

We now describe our experiments and discuss our results. We build a social network from the Call Detail Records (CDRs - transaction detail about one phone call/SMS) of one of the largest telecom service providers in the world.

We used Java 1.6 to implement all three algorithms in a single thread program. We set the maximum memory size in Java to be 2GB and the programs were run on a 2 CPU running at 3GHz 4GB memory machine. While we report one number each for number of nodes in decision trees and execution time, we had performed several (≥ 5 in each case) runs of the experiments. The number of nodes in the repeated runs were (expectedly) the same and the variation in the execution time was bounded by about 5%. The execution time reported is obtained by making calls to system clock and is in seconds.

A Call Detail Record is the transaction detail about one phone call/SMS. It includes information about caller, callee and call duration among other things. We treated each phone number as a vertex and created an edge between a pair of vertices if there was a phone call/SMS between the two phone numbers corresponding to the vertices. This process provides us a simple unweighted undirected graph for each social network. We collected the CDRs for two different regions for a period of one month each and segregated data into SMS and voice call data. We discarded the CDRs corresponding to inter-circle calls (i.e., we considered only the local calls / SMSs). This gives us four data sets. One of the regions was an urban area and the other was mixed (urban+rural) population. Hence our study covers various prototypical social networks. Some statistics about the data sets is presented in Table 1.

Effectiveness of filtering techniques: First we study the effectiveness of the filtering techniques presented in this paper. After the *SNNFiltering*, the number of vertices and edges come down by several orders of magnitude. Table 2 shows our results. Figure 4 shows the number of vertices and number of edges for $DS4$ as a function filtering level. One can see that the degree based filtering achieves a factor of ~ 11 reduction in $|V|$ and ~ 4 reduction in $|E|$ for $L = 13$ on the dataset $DS4$. The reduction factor crosses ~ 1000 for $|V|$ and ~ 220 for $|E|$ for $L = 20$. However, the shared neighbourhood based filtering far outperforms the degree based filtering by achieving a reduction by factor of ~ 190 for $|V|$ and ~ 178 for $|E|$ *over and beyond* what the degree based filtering achieves for $L = 13$. For $L = 20$, the further reduction ratio (from degree based to SNN filtering) is ~ 64 for $|V|$ and ~ 86 for $|E|$. Please note that the graph size resulting from SNN filtering for $L = 13$ itself is much smaller than the size of the graph resulting from degree based filtering for $L = 20$, the size of the maximal clique. After SNN filtering, the parameters which influence the time and space complexity (namely Δ and δ_{max}) are also considerably lower (as compared to degree based filtering). This implies that the performance of clique enumeration algorithms become far better due to SNNFiltering.

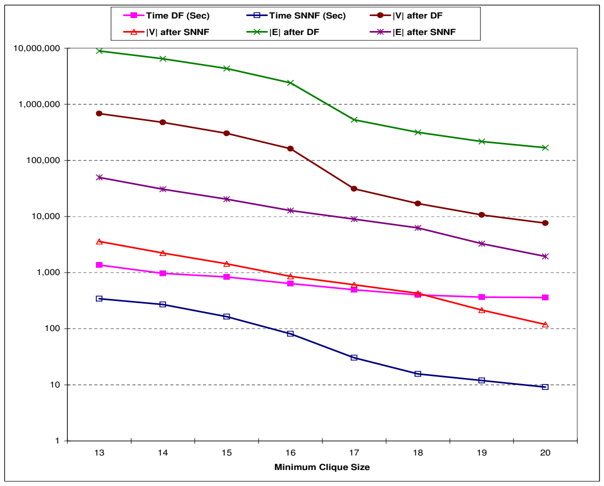

Figure 4: Effectiveness of filtering on DS4: Without filtering $|V| = 5,806,784$ and $|E| = 28,951,874$

Key Observation: *The shared neighbourhood criterion for filtering the graph is very effective and fairly efficient.*

Execution behavior of clique enumeration algorithms: We now present the execution behavior of the three algorithms, our taboo list based algorithm presented in this paper, neighbourhood based algorithm presented in [16] and the depth first search approach presented in [10]. We modified the other two algorithms to incorporate check for minimum size criterion at appropriate place and we also modified the DFS algorithm for maximal clique enumeration rather than the maximal independent set enumeration.

We perform three experiments about the execution behavior of the algorithms. In the first experiment, we want to find the execution behavior of the algorithms for appropriately filtered graphs. This would tell us how fast we can get the cliques of a given minimum size using a particular algorithm with appropriately filtering. Here, both the input graph and cliques of interest are different in different cases. In the second experiment, we want to study the sensitivity of the clique finding algorithms on L for a fixed input graph (i.e., the input graph is same, but the cliques of interest are different in different cases). Finally, we want to characterize the effect of input graph size on the execution behavior of the clique finding algorithms when the algorithms are trying to find the cliques of fixed minimum size (i.e., the input graphs are different, but the cliques of interest are the same in different cases).

Now we describe our experiments in more detail. First, we choose different values of L and filter each dataset using the *DBFilter* and *SNNFilter* algorithms for this value of L, resulting in smaller graphs with different sizes. We report the number of cliques and the number of nodes in the binary decision tree and the execution time for each algorithm for each value of L in Table 3. Here MaxCand means the maximum number of valid candidates at any point of execution of algorithm. We do not report this number for *DFS* algorithm since it explores one candidate at a time. TotalCand refers to the number of candidates generated by the algorithm in all. For neighbourhood based algorithm we report the number of leaf nodes and total number of nodes generated in the binary decision tree.

As we go from higher values of L to the lower values, we see that at first, the neighbourhood based algorithm performs slightly better than the other two. However, when the value of L is a little away from the maximum clique size, our taboo list based algorithm starts performing the best. Also, the rate of increase in the execution time is the slowest for our algorithm, followed by the DFS algorithm, and it increases very sharply for Neighbourhood based algorithm as the value of L decreases (and consequently, the number of maximal cliques increases). The reason for this behavior is that when we set L to be close to the maximum clique size, there are very few opportunities for the algorithms to attempt dropping a vertex as it would lead to a clique candidate smaller than the desired size L. Hence, the Neighbourhood based algorithm and DFS based algorithms are not penalized heavily for carrying extra candidates or for repetitive computations. However, once the value of L is reduced, the number of maximal cliques become far larger and the penalty for carrying the non-maximal cliques for too long is stiff for Neighbourhood based algorithm. Also, the size of decision subtrees rooted at the dropped nodes become large as L reduces and the DFS algorithm starts performing repetitive computation leading to slow down. In our approach, since we reuse the computations performed at a node in all its successors, we do not suffer from this type of a slow down. Another fact worth mentioning is that the neighbourhood based recursive algorithm starts to run out of heap space due to the deep calls putting too much data on to the heap. Although with an implementation in other programming languages may enable it to process these data sets, it is clear that this algorithm is more memory intensive in a comparable implementation platform.

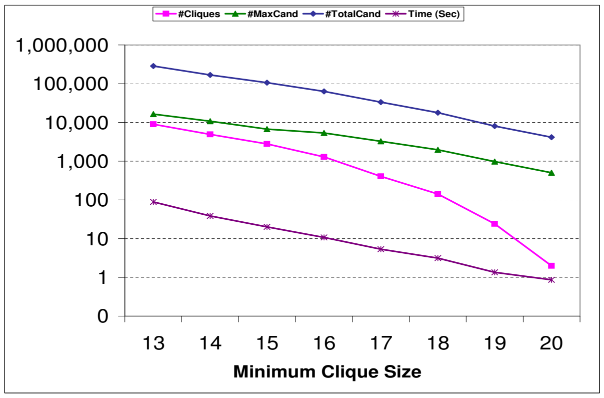

Figure 5: Execution behaviour of SELMaC2 on DS4

Figure 5 characterizes the execution behaviour of our algorithm on $DS4$. We plot the number of cliques, time taken by our algorithm, the total number of candidates generated and maximum number of candidates at any given point of time as a function of minimum desired clique size.

In order to see the effect of change in L on the behavior of the three clique enumeration algorithms on the same graph, we fix our input graph and its filtering level. We then vary L for the clique enumeration algorithms and the results are reported in Table 4. It is clear that as L grows, the run time and number of candidates etc go down exponentially. It is interesting to note that the Neighbourhood-based approach failed to produce any output for all the runs within the same resource constraints in this set of experiments. Hence it is not included in the table.

Finally, we choose different values of L and filter the dataset using the *DBFilter* and *SNNFilter* algorithms for this value of L, resulting in smaller graphs with different sizes. Now, we find the same set of cliques by running the clique enumeration algorithms with a fixed value of L. We report the number of cliques and the number of nodes in the binary decision tree and the execution time for each algorithm in Table 5. Also, based on this trend, it is clear that one can not hope to run the maximal clique enumeration algorithms on the graphs of such scale using comparable resources without the preprocessing techniques discussed in the paper.

Key Observations:

1. *Our algorithm is the fastest when the number of cliques is reasonably large. It is also less sensitive to the input size compared to the other algorithms.*

2. *Total number of candidates is least in our algorithm in all the experiments.*

7 Conclusions

In this paper, we examined a variation of maximal clique enumeration problem with a constraint on the minimum size of the maximal cliques, presented preprocessing technique to reduce the effective graph size and presented a new large maximal clique enumeration algorithm. We evaluated our approach on real life social networks formed by CDRs of one of the largest telecom service providers in the world. We found that the filtering techniques reduce the graph size by orders of magnitude and our large clique enumeration algorithm outperforms the methods adopted from literature in significant number of cases.

As we found in our experiments, in real life the seemingly daunting graph sizes may be handled in very practical times (within minutes) to find the cliques, hence it may be worthwhile to try and characterize the expected run times of community finding algorithms based on graph properties.

References

[1] J. Abello, M. G. C. Resende, and S. Sudarsky. Massive quasi-clique detection. In *Latin American Theoretical INformatics*, pages 598–612, 2002.

[2] E. A. Akkoyunlu. The enumeration of maximal cliques of large graphs. *SIAM Journal of Computing*, 2(1):1–6, March 1973.

[3] R. Alba. A graph-theoritic definition of a sociometric clique. *Journal of Math Sociology*, 3:113–126, 1973.

[4] A. Clauset, M. E. J. Newman, and C. Moore. Finding community structure in very large networks. *Phys. Rev.*, E 70(066111), 2004.

[5] Y. Dourisboure, F. Geraci, and M. Pellegrini. Extraction and classification of dense communities in the web. In *WWW*, pages 461–470, 2007.

[6] D. Gibson, J. Kleinberg, and P. Raghavan. Inferring web communities from link topology. In *HYPERTEXT*, pages 225–234, 1998.

[7] M. Girvan and M. E. J. Newman. Community structure in social and biological networks. *Proc. Natl. Acad. Sci*, USA 99(7821), 1973.

[8] Z. Gyongyi and H. Garcia-Molina. Web spam taxonomy. In *First International Workshop on Adversarial Information Retrieval on the Web*, 2005.

[9] R. A. Hanneman and M. Riddle. *Introduction to Social Network Methods*. University of California, Riverside, Riverside, CA, 2005.

[10] E. Loukakis and C. Tsouros. A depth first search algorithm to generate the family of maximal independent sets of a graph lexicographically. *Computing by Springer-Verlog*, 27:349–366, 1981.

[11] R. Luce. Connectivity and generalized cliques in sociometric group structure. In *Psychometrika*, volume 15, pages 169–190, 1950.

[12] K. Makino and T. Uno. New algorithm for enumerating all maximal cliques. In *SWAT*, pages 668–679, 2004.

[13] N. Modani and K. Dey. Large maximal cliques enumeration in sparse graphs. In *CIKM*, 2008.

[14] G. D. Mulligan and D. G. Corneil. Corrections to bierstone's algorithm for generating cliques. *Journal of the Assoc. for Comp. Machinery*, 19:244–247, 1972.

[15] A. A. Nanavati, S. Gurumurthy, G. Das, D. Chakraborty, K. Dasgupta, S. Mukherjea, and A. Joshi. On the structural properties of massive telecom call graphs: Findings and implications. In *CIKM*, 2006.

[16] R. E. Osteen and J. T. Tou. A clique-detection algorithm based on neighbourhoods in graph. *Intl. Journal of Computer and Information Science*, 2(4):257–268, 1973.

[17] S. Seidman and B. Foster. A graph-theoritic generalization of the clique concept. *Journal of Mathematical Sociology*, 6:139–154, 1978.

[18] S. B. Seidman. Network structure and minimum degree. *Social Networks*, 5:269–287, 1983.

[19] E. Tomita, A. Tanaka, and H. Takahashi. The worst-case time complexity for generating all maximal cliques and computational experiments. *Theor. Comput. Sci.*, 363(1):28–42, 2006.

[20] S. Tsukiyama, M. Ide, H. Ariyoshi, and I. Shirakawa. A new algorithm for generating all the maximal independent sets. *SIAM Journal of Computing*, 6(3):505–517, September 1977.

[21] D. J. Watts and S. H. Strogatz. Collective dynamics of 'small-world' networks. *Nature*, 393:440–442, June 1998.

Data		Time Taken		$\|V\|$		$\|E\|$		δ_{max}		Δ	
Set	L	Deg	SNN	Deg	SNN	Deg	SNN	Deg	SNN	Deg	SNN
	6	64.973	19.242	117,412	7,914	618,325	39,471	964	218	99	61
	7	55.416	9.722	56,868	2,981	340,615	19,196	469	200	85	55
	8	42.611	5.324	25,222	1,413	168,109	10,975	274	172	76	50
DS1	9	41.106	3.921	5,959	711	46,644	6,388	253	153	67	43
	10	36.453	1.485	2,283	397	21,916	3,940	235	138	63	37
	11	27.866	0.832	1440	220	15,686	2,320	222	103	59	32
	12	27.467	0.537	985	155	11,856	1,480	208	61	56	27
	13	26.750	0.419	692	80	9,089	790	198	45	53	24
	8	205.047	84.911	210,672	4,608	1,457,362	39,482	5,505	169	160	70
	9	147.778	18.323	53,150	2,603	446,893	24,719	1,526	141	130	61
	10	117.163	10.374	23,074	1,527	228,078	15,610	601	114	115	53
DS2	11	107.001	8.090	12,713	927	142,194	9,697	356	91	104	46
	12	97.501	4.161	7,813	573	97,210	6,062	219	60	97	39
	13	91.763	2.855	4,427	370	63,881	3,468	213	44	92	30
	14	87.978	2.404	3,107	94	49,796	838	211	37	89	21
	15	99.977	1.351	2,523	56	43,143	480	209	22	87	19
	12	1026.022	265.875	872,184	3,057	11,105,410	36,343	2,957	97	332	62
	13	907.970	243.792	702,524	2,049	9,281,099	25,106	2,601	84	307	57
	14	947.419	178.072	541,895	1,262	7,401,614	16,470	2,189	76	282	52
DS3	15	820.090	137.614	377,296	907	5,320,047	12,391	1,637	71	255	49
	16	642.502	73.659	193,925	633	2,853,552	8,902	1,098	63	218	46
	17	570.687	23.775	59,557	401	928,046	5,515	527	52	168	40
	18	368.715	10.533	25,088	211	401,956	2,846	386	49	132	36
	19	311.987	4.603	6,829	72	116,144	1,123	222	47	98	32
	13	1361.123	343.507	681,755	3,597	8,890,281	49,884	2,969	261	569	74
	14	968.530	270.186	475,173	2,237	6,487,060	30,660	2,502	136	489	63
	15	836.269	164.434	304,074	1,431	4,333,420	20,330	1,980	90	399	57
DS4	16	640.435	81.090	161,971	856	2,410,607	12,786	1,480	82	295	53
	17	497.075	30.380	31,150	607	529,290	9,015	608	73	171	47
	18	399.878	15.680	17,010	427	316,233	6,264	489	67	148	42
	19	364.323	11.960	10,651	214	216,449	3,288	482	56	141	37
	20	360.749	9.121	7,629	119	167,648	1,940	480	49	134	35

Table 2: Effect of Filtering

			Our Algorithm			DFS		Neighbourhood		
DataSet	L	#Cliques	MaxCand	TotalCand	Time	TotalCand	Time	TotalNodes	LeafNodes	Time
	6	9,973	11,115	81,661	53.336	107,412	147.191	N/F	N/F	N/F
	7	4,903	6,020	49,432	12.779	67,875	32.216	N/F	N/F	N/F
	8	2,520	3,540	31,591	5.064	43,539	9.698	378,852	87,149	142.408
DS1	9	1,287	1,906	17,828	2.097	26,992	3.579	226,037	38,389	13.467
	10	671	1,207	11,430	1.233	16,706	1.345	117,065	12,931	2.157
	11	290	589	5,668	0.694	8,873	0.633	51,879	3,115	0.520
	12	108	272	2,516	0.264	3,346	0.251	18,565	477	0.374
	13	35	91	765	0.192	1,215	0.135	4,131	35	0.169
	8	9,066	11,936	122,413	39.02	164,380	132.711	N/F	N/F	N/F
	9	4,752	6,938	75,603	17.49	103,904	43.618	N/F	N/F	N/F
	10	2,335	3,978	44,154	7.487	58,997	12.254	562,809	79,437	96.800
DS2	11	975	2,004	21,829	3.052	29,436	4.709	268,810	27,092	10.214
	12	329	860	9,543	1.166	13,054	1.311	109,645	7,264	1.236
	13	110	255	3,131	0.313	4,852	0.442	37,742	1,412	0.252
	14	42	73	835	0.113	1,034	0.142	10,343	174	0.108
	15	10	16	227	0.058	240	0.072	1,469	10	0.071
	12	10,024	17,027	194,265	47.631	279,122	107.154	N/F	N/F	N/F
	13	5,356	10,887	127,015	25.727	184,761	45.470	N/F	N/F	N/F
	14	2,747	6,816	79,068	12.846	122,141	19.904	1,886,532	183,639	81.309
DS3	15	1,169	4,396	47,709	7.257	76,435	11.360	829,655	49,043	12.211
	16	438	2,497	27,188	4.337	42,598	5.212	338,657	10,219	2.307
	17	127	1,384	13,434	2.051	16,841	1.592	124,559	1,569	0.656
	18	27	612	6,166	1.021	6,939	0.642	47,446	159	0.276
	19	8	361	3,420	0.640	3,957	0.294	20,533	8	0.168
	13	8,926	16,387	284,706	89.172	401,455	187.817	N/F	N/F	N/F
	14	4,882	10,800	168,869	38.632	252,726	69.058	N/F	N/F	N/F
	15	2,784	6,735	105,745	20.018	162,587	31.932	2,727,889	186,997	1695.293
DS4	16	1,291	5,351	63,377	10.717	95,742	12.157	1,127,515	42,252	71.395
	17	404	3,263	33,242	5.332	50,300	5.402	438,905	7,188	3.296
	18	141	1,972	17,835	3.138	21,592	1.972	178,101	845	0.917
	19	24	974	8,032	1.355	10,037	0.795	79,425	61	0.406
	20	2	501	4,170	0.868	6,814	0.472	26,503	2	0.188

Table 3: Comparison of execution behavior: filtered for corresponding minimum clique size

			Our Algorithm			DFS	
DataSet	L	#Cliques	MaxCand	TotalCand	Time	TotalCand	Time
DS1	6	9,973	11,115	81,661	93.336	107,412	247.191
	7	4,903	6,381	57,427	74.787	81,609	200.655
	8	2,520	4,021	41,056	48.852	61,618	182.106
	9	1,287	2,658	29,197	31.656	46,256	168.277
	10	671	1,802	21,189	21.229	34,898	143.991
	11	290	1,254	15,290	14.356	25,987	115.549
	12	108	869	10,832	10.871	19,098	97.617
	13	35	567	7,750	8.477	14,146	77.120
DS2	8	9,066	11,936	122,413	39.020	164,380	162.711
	9	4,752	7,591	90,157	36.907	126,624	146.499
	10	2,335	4,944	65,452	28.984	95,241	132.694
	11	975	3,181	46,459	23.149	70,700	126.207
	12	329	1,963	32,253	17.120	52,474	106.138
	13	110	1,318	23,625	14.365	39,441	90.825
	14	42	902	17,736	12.097	30,128	77.621
	15	10	648	13,834	10.711	23,380	64.170
DS3	12	10,024	17,027	194,265	47.631	279,122	107.154
	13	5,356	11,661	142,048	41.013	212,266	90.801
	14	2,747	8,222	101,142	32.078	159,119	79.123
	15	1,169	5,714	69,847	23.645	116,479	69.469
	16	438	4,012	48,120	17.676	84,512	60.703
	17	127	2,852	33,821	14.111	61,576	51.414
	18	27	2,085	24,382	10.638	45,449	42.583
	19	8	1,511	18,086	8.300	33,832	36.853
DS4	13	8,926	16,387	284,706	89.172	401,455	187.817
	14	4,882	11,907	212,466	75.776	306,701	157.498
	15	2,784	8,381	155,752	63.650	233,055	151.485
	16	1,291	5,939	113,743	48.461	174,885	133.324
	17	404	4,290	82,468	38.021	130,410	119.857
	18	141	3,129	60,797	30.770	97,747	108.221
	19	24	2,267	45,611	25.768	73,558	90.476
	20	2	1,517	34,925	21.550	56,667	81.192

Table 4: Comparison of execution behavior on DS1, DS2, DS3 and DS4: DS1 filtered for $L = 6$, DS2 filtered for $L = 8$, DS3 filtered for $L = 12$ and DS4 filtered for $L = 13$

	Our Algorithm			DFS		Neighbourhood		
Filtering	MaxCand	TotalCand	Time	TotalCand	Time	TotalNodes	LeafNodes	Time
6	3,149	31,891	48.787	34,898	142.736	N/F	N/F	N/F
7	2,434	26,247	15.783	30,648	36.383	N/F	N/F	N/F
8	2,066	21,339	6.115	25,829	11.116	137,242	12,963	7.816
9	1,748	17,296	2.927	21,084	4.068	126,552	12,933	2.934
10	1,207	11,430	1.233	16,706	1.345	117,065	12,931	2.157
8	4,392	49,401	48.166	52,474	110.147	N/F	N/F	N/F
9	3,795	39,273	20.591	41,185	37.364	N/F	N/F	N/F
10	2,860	29,381	9.085	30,991	10.828	156,262	7,253	3.548
11	1,752	18,847	3.444	20,192	3.947	132,344	7,260	1.934
12	860	9,543	1.166	13,054	1.311	109,645	7,264	1.236
12	8,924	85,264	41.595	84,512	61.052	N/F	N/F	N/F
13	5,557	63,469	20.336	70,475	30.132	N/F	N/F	N/F
14	4,915	54,758	12.819	61,442	13.623	403,569	10,220	3.231
15	4,781	50,417	9.908	54,378	9.516	380,259	10,220	2.725
16	2,497	27,188	4.337	42,598	5.212	338,657	10,219	2.307
13	11,444	127,204	85.425	130,410	122.924	N/F	N/F	N/F
14	8,339	90,648	34.609	106,160	50.245	N/F	N/F	N/F
15	7,262	75,363	22.960	87,455	24.324	592,750	7,154	8.129
16	5,992	56,001	13.036	68,798	9.890	518,423	7,157	5.591
17	3,263	33,242	5.332	50,300	5.402	438,905	7,188	3.296

Table 5: Comparison of execution behavior on DS1, DS2, DS3 and DS4: clique enumeration algorithms were run with $L = 10$ for DS1, 12 for DS2, 16 for DS3 and 17 for DS4. The value of L used in filtering is shown in the first column.