In maximum clique problem, it is desired to find maximum number of vertices, any two of which are adjacent. The maximum clique problem falls into category of NP-hard problems. It is, therefore, often avoided to detect maximum clique by practitioners in many applications despite the fact that it has significant applications in the field of information retrieval, data mining, network analysis etc. Community Detection in social networks is one of the recent trends in computer science. Maximum Clique and community in social networks have overlapping definitions in respective domains. Thus problem of community detection in social networks reduces to finding cliques in graphs, provided social networks are represented as graphs. Several exact algorithms to find maximum clique already exist in literature that promise acceptable runtimes on certain graphs. But problem arises when these algorithms are applied on real world graphs which are massive in size. In this work, a novel branch and bound exact algorithm to find maximum clique, Maximum Clique Finder (MCF) has been presented with new pruning steps. This algorithm has been tested on real world graphs and DIMACS benchmark graphs, where it exhibits runtimes several times better than other existing algorithms and it performs notably well on large sparse real world graphs.

**Keywords:** Community Detection in Social Networks, Information Retrieval, Data Mining, Graphs, NP-Hard problems.

1. INTRODUCTION

An undirected graph is denoted by $G = (V, E)$, where $V$ is set of vertices and $E$ is set of edges. A clique in the graph is set of vertices in which any two vertices are adjacent to each other. Two vertices are adjacent to each other if there is an edge between them. Therefore clique is a complete sub-graph of a given undirected graph. Finding the clique that has maximum number of vertices in a graph is called maximum clique problem [1].

There is a wide range of prominent applications of maximum clique problem such as community detection in networks [2, 3, 4], data mining in biometrics [5], information retrieval [6], data mining [7], symptoms correlation based disease classification [8], computer vision [9], coding theory [10], and pattern recognition [11]. Many more applications are listed in [12, 13].

Finding maximum clique is an NP-Hard problem [14]. An independent set is set of vertices in which no two vertices are adjacent. Finding the largest such set in a graph is called maximum independent set problem. A problem similar to this is vertex cover problem. A vertex cover is set of vertices which covers all the edges in the graph. To find smallest such set in the graph is called maximum vertex cover problem. The maximum clique problem is computationally equivalent to these two problems. Since all of these problems are NP-Hard problems, no polynomial time exact algorithm is expected to be found. Nevertheless, maximum clique problem finds several significant applications in prominent fields of computer science, it is of great interest to try to develop fast and exact algorithms.

Almost every exact algorithm employs branch and bound approach which continuously optimizes the search of solution by discarding (pruning) the branches which will not lead to solutions any better than previously acquired solutions.


This paper presents an algorithm to find maximum clique in an undirected graph with novel pruning steps. Some very promising exact algorithms and the development in finding maximum clique are briefly discussed in section 2. In section 3 our algorithm Maximum Clique Finder (MCF) is described in detail. Section 4 discusses implementation and result analysis. Section 4 concludes this article.

2. RELATED WORK

A simple approach to find maximum clique in an undirected simple graph $G$ is to find all the cliques present in the graph and then select the largest one. But enumerating all the cliques requires infeasible time. Hence a simple algorithm is presented in [15], which reduces enumerations significantly. By pruning the fruitless branches, the search space is tremendously reduced.
reduced. The algorithm finds largest clique containing vertex $v_i$ at each step $i$ by performing depth first search from vertex $v_i$. At each depth $i$, if the member of remaining vertices, which can possibly constitute a clique containing vertex $v_i$, is smaller than the size of largest clique found so far, the algorithm backtracks by pruning this branch of enumeration. Algorithm proposed in [16] incorporates an additional pruning in algorithm presented in [15] with the help of some auxiliary bookkeeping. Algorithm proposed in [16] is faster than algorithm proposed in [15] on random and DIMAX benchmark graphs [22]. However the order in which vertices are processed majorly affects the pruning strategy used in this algorithm.

Many algorithms to find maximum clique use vertex coloring to define upper bound on the maximum clique. MCQ algorithm [17] is one of the latest and popular methods which uses this idea. MaxCliqueDyn [23] is the improved version of MCQ with the variants MCQD and MCQD&CS. It uses computationally more expensive tighter upper bounds, which are applied on a part of search space. BBMC [24] is another enhanced version of MCQ which uses efficient methods to compute graph transitions and bounds. It uses bit strings to sort vertices in constant time.

3. MAXIMUM CLIQUE FINDER ALGORITHM (MCF)

In this section a new algorithm, MCF, is presented which overcomes the limitations of other algorithms mentioned earlier by the use of additional pruning strategies. Following notations are used in the algorithm. The graph $G(V,E)$ contains $n$ vertices as $\{v_1, v_2, \ldots, v_n\}$. $\text{Adj}(v_i)$ is set of vertices adjacent to vertex $v_i$. And the cardinality of $\text{Adj}(v_i)$ e.i. degree of $v_i$ is denoted by $\text{deg}(v_i)$. Degree of each vertex is computed once in the beginning of the algorithm.

Maximum clique in a graph can be found by enumerating the largest clique containing each vertex and then selecting the largest among these. Most significant point of our algorithm is that the search space is reduced by pruning the vertices which cannot form cliques larger than the current maximum clique in an incremental fashion. Algorithm 1 and algorithm 2 shown in figure 1 and figure 2 respectively outline this method. The variable max stores size of the largest clique found so far. Initially it is set as 0 or any other positive lower bound, if cliques smaller than the lower bound, are insignificant.

Maximum clique containing a vertex $v_i$ cannot be larger than the degree of $v_i$, hence only the adjacent vertices of $v_i$ are considered to obtain the largest clique containing $v_i$. The main procedure MCF therefore generates a set $U \subseteq \text{Adj}(v_i)$ for each vertex $v_i$, which contains those neighbors of $v_i$ which could survive the prunings. Subroutine Find_Clique is then invoked on $U$. Subroutine Find_Clique presented in Algorithm 2 enumerates every possible clique containing vertex $v_i$ in a recursive fashion and returns the largest clique containing $v_i$. The size of clique found at any point of execution of Find_Clique is stored in 'size'. Initial value of size is set 1 as we start with a clique having just one vertex.

Algorithm 1: MCF($G=(V,E)$)

/*max, size, C and M are global variables.*/
begin
Sort all the vertices in $V$ non-increasing order of degree./*Pruning 1*/
max = 0;
for $i \leftarrow 1$ to $n$ /*to iterate $n$ times*/
size = 1;
$C \leftarrow \emptyset$;
$v_i \leftarrow \text{Select_First}(V)$; /* $v_i$ is highest degree node in $V$.*/
$V \leftarrow V \setminus \{v_i\}$; /*Pruning 2*/
current_deg = $\text{deg}(v_i)$;
$C \leftarrow C \cup \{v_i\}$; /*$C$ is current clique containing $v_i$*/
$U \leftarrow V \cap \text{Adj}(v_i)$;
Find_Clique($U$, $C$);
if size > max then
  max = size;
  $M \leftarrow C$; /*$M$ is max-clique found so far*/
end if
if max = current_deg then /*Pruning 3*/
  return $M$;
end if
end for
return $M$;
end

Fig. 1 Algorithm for finding maximum clique

Algorithm 2: Find_Clique($U$, size, $C$);
begin
while $U \neq \emptyset$ do
  if size + |$U$| ≤ max then /*Pruning 4*/
    return;
  end if
  select highest degree node $u$ from $U$;
  $U \leftarrow U \setminus \{u\}$;
  $C \leftarrow C \cup \{u\}$;
  Find_Clique($U \cap \text{Adj}(u)$, size+1, $C$);
end while
end

Fig. 2 Subroutine for algorithm 1
chances to be in max-clique. Pruning 1 emphasizes this idea as
the vertices are already sorted in non-increasing order of their
degrees. Lower degree nodes are pruned in very obvious
manner. Pruning 2 avoids re-computation of already found
cliques by including only those vertices in neighbors list of
vertex \( v_i \) whose cliques are not yet found. Pruning 3 works on
the same analogy used in pruning 1. If the largest clique found
so far is of size \( k \), then any vertex of degree smaller than or equal
to \( k \) cannot form a clique of size greater than \( k \). Therefore at this
point all the vertices having degree less than or equal to \( k \) are
ignored for further search of the largest clique. Pruning 4 says if
all the vertices of \( U \) were added to get the clique, its size cannot
be more than the size of largest clique found so far (max). Pruning 4 is most frequent pruning, pruning 1 and 2 are
moderate and pruning 3 is used just once.

\[
\text{max} = \text{deg}(2) /*\text{Pruning 3 prunes rest of the}
\text{vertices}*/
\text{return \{4, 2, 3, 5\} /*Max Clique*/}
\]

It is clearly evident from the above demonstration that, the
maximum clique found by MCF is correct. MCF finds
maximum clique in single iteration of the main algorithm, due to
pruning 3 which is extremely effective and gives MCF upper
hand as compared to other exact algorithms. We shall prove this
fact with the help of experimental results in next section.

4. EXPERIMENTS AND RESULT ANALYSIS

In this section we present comparison of performance of
our algorithm with other exact algorithms. Our experiments
were performed on 64 bit windows 7 Home Basic with 2.3 GHz
Intel Core i3 with 32 GB of main memory. Implementation is
done in C compiled using NeuTroN DoS-C++ version 0.77.0.0.
Our implementation uses a simple adjacency list representation
for graph. This is done by maintaining a reference array of size
\(|V|\), which contains references for \(|V|\) lists of vertices, each
corresponding to a particular vertex. Figure 4 shows our
representation.

We have considered graphs from two categories. First
category includes graphs originated from real world
applications. Table 1 gives brief description of those graphs.
Second category includes graphs from DIMACS
Implementation Challenge [22]. Table 2 represents structural
properties of the graphs.

### Table 1 Description of Real World Graphs Used in Experiment.

<table>
<thead>
<tr>
<th>Graph</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Email_enron [25]</td>
<td>Communication network of E-mail</td>
</tr>
<tr>
<td>dictionary28 [20]</td>
<td>Network of words</td>
</tr>
<tr>
<td>code-mat-2003 [26]</td>
<td>Collaboration network of scientists</td>
</tr>
<tr>
<td>foldoc [27]</td>
<td>Dictionary for computing related terms</td>
</tr>
<tr>
<td>web-Google [28]</td>
<td>Web graph released as part of Google</td>
</tr>
<tr>
<td>soc-wiki-vote [29]</td>
<td>programming contest in 2002</td>
</tr>
</tbody>
</table>
Three significant exact algorithms are considered for result analysis i.e. Carraghan Pardalos [15], Ostergard Algorithm [16] and MCQD [23]. For Carraghan Pardalos we have used our own implementation. For Ostergard Algorithm, we have used cliquer source code [31] that is publically available. For MCQD also we have used publically available source code available at http://insilab.org/maxclique/. Table 3 represents performance comparison of various algorithms. We have set an upper limit of 1800 seconds on runtime. The program is forcefully aborted if it fails to terminate in 1800 seconds. It is shown by an Asterisk (*) in the table. Figure 3 represents comparison of normalized runtimes of various algorithms. Figure 4 represents comparison of runtimes of various algorithms against edge density in the graphs.

<table>
<thead>
<tr>
<th>Graph</th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

![Graph Comparison](image_url)
Fig. 4. Runtimes of various algorithms vs edge density.

From figure 4 we can conclude that MCF performs excellent if the graph is sparse. Maximum clique problem has major applications in community detection in social graphs. Almost every social graph follows power law of degree distribution [1], which implies that there are small number of higher degree vertices and large number of lower degree vertices. Pruning 3 in our algorithm utilizes this characteristic and most of the lower degree vertices are pruned in initial iterations of our algorithm. For DIMACS graphs also our algorithm gives very good results.

5. CONCLUSION

A novel exact algorithm, MCF, to find maximum clique has been presented in this paper. The algorithm has been tested on real world graphs and DIMACS benchmark graphs. The results are compared against performance of other significant exact algorithms such as algorithm proposed by Carraghan and Pardalos [15], Ostergard algorithm [16], MCQ [17] and MCQD [24]. The results show that proposed algorithm performed tremendously well with real world graphs. For DIMACS benchmark dense graphs, our algorithm brings slight improvement over Carraghan and Pardalos algorithm whereas cliquer [16] performs better. For sparse real world graphs our algorithm performed significantly better than any other algorithm. Maximum clique problem falls into category of NP-Hard problems. Due to this reason, maximum clique detection is avoided in several applications. But the results shown in this research establish the fact that maximum cliques can be found in feasible time in significantly large and sparse real world graphs.

REFERENCES


AUTHOR BIOGRAPHIES

Atul Srivastava received the B. Tech. degree in Information Technology from the UP Technical University, Lucknow, India, in 2008, and M.Tech. degree in Information Technology from YMCA University of Science and Technology, Faridabad, India in 2012. He is a research student of YMCA University of Science and Technology, Faridabad, India. He is currently Assistant Professor at Pranveer Singh Institute of Technology, Kanpur, India.

Dr. Anuradha Pillai has received her M. Tech and PhD in Computer Engineering from MD University Rohtak, in the years 2004 and 2011 respectively. She has published 30 research papers in various International journals and conferences. She has more than 13 years of teaching experience. Presently she is serving as Assistant Professor at YMCA University of Science & Technology, A State Govt. University, Faridabad Haryana. Her research interests include Web Mining, Data Structures and Algorithms, Databases. Research Papers listed in database.

Dr. Dimple Juneja Gupta is working as Faculty at Department of Computer Applications at NIT Kurukshetra, India.