# Graph Algorithms Project Report

Group 40

March 23, 2025

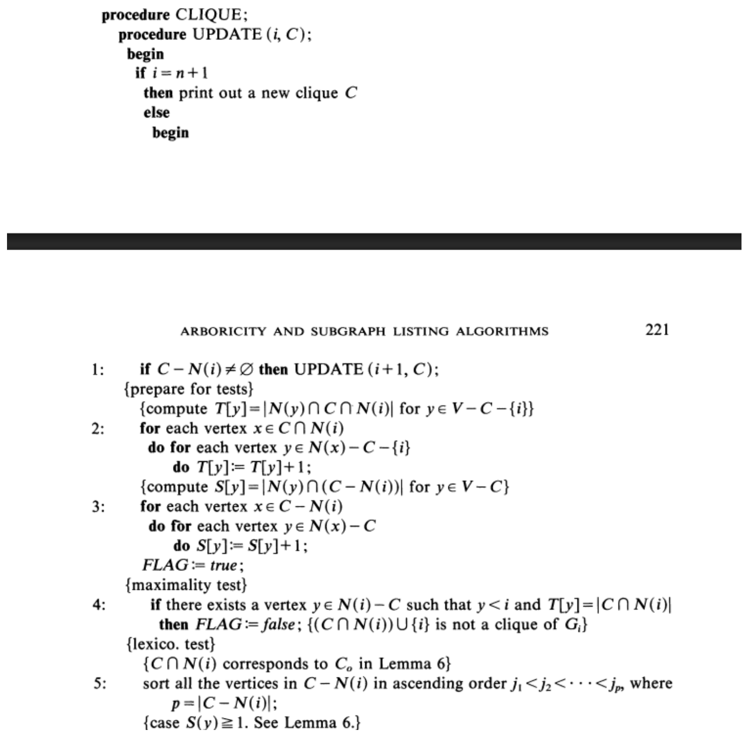
## 1. Arboricity and Subgraph Listing Algorithms (Chiba & Nishizeki, 1985)[?]

The paper gives different algorithms for four subgraphs listing algorithms. The algorithms are: listing all triangles in the graph, listing all quadrangles in the graph, listing all complete subgraphs and listing all maximal cliques in the graph. We focus only on the fourth, that is, listing all maximal cliques in the graph. Moroever, we shall also see how many maximal cliques are present in the graph and what is the size of the largest maximal clique. Also, we'll test this algorithm on 3 datasets as discussed below and compare the execution times for the algorithm on the 3 datasets.

### 1.1 Time Complexity

The paper proves that the following algorithm runs in O(a(G)*m) time. Here, a(G) is the arboricity of the graph, which is equal to the minimum number of edge-disjoint spanning forests into which G can be decomposed; m is the number of edges.

## 1.2 Algorithm



```
procedure CLIQUE;
  procedure UPDATE ($i$, $C$);
    begin
     if $i = n+1$
       then print out a new clique $C$
       else
         begin
```

ARBORICITY AND SUBGRAPH LISTING ALGORITHMS 221

```
1:        if $C - N(i) \neq \emptyset$ then UPDATE ($i+1$, $C$);
        {prepare for tests}
          {compute $T[y] = |N(y) \cap C \cap N(i)|$ for $y \in V - C - \{i\}$}
2:        for each vertex $x \in C \cap N(i)$
            do for each vertex $y \in N(x) - C - \{i\}$
                do $T[y] := T[y] + 1$;
          {compute $S[y] = |N(y) \cap (C - N(i))|$ for $y \in V - C$}
3:        for each vertex $x \in C - N(i)$
            do for each vertex $y \in N(x) - C$
                do $S[y] := S[y] + 1$;
          $FLAG := true$;
        {maximality test}
4:          if there exists a vertex $y \in N(i) - C$ such that $y < i$ and $T[y] = |C \cap N(i)|$
              then $FLAG := false$; {$(C \cap N(i)) \cup \{i\}$ is not a clique of $G_i$}
        {lexico. test}
          {$C \cap N(i)$ corresponds to $C_o$ in Lemma 6}
5:        sort all the vertices in $C - N(i)$ in ascending order $j_1 < j_2 < \cdots < j_p$, where
              $p = |C - N(i)|$;
          {case $S(y) \geqq 1$. See Lemma 6.}
```

Figure 1: Algorithm part 1

The algorithm is based on Tsukiyama's algorithm which is as follows: If G is a graph with vertex set $V = \{0, 1, 2, \ldots, n-1\}$ and edge set $E$, $G_i$ is the subgraph of $G$ induced by vertices $0, 1, \ldots, i$ and $N(i)$ is the set of vertices adjacent to $i$ in the given graph $G$. If we know that $I_{i-1}$ is a maximal independent set of $G_{i-1}$ then:

1. $I_{i-1} \cap N(i)$ is not empty, then $I_{i-1}$ is a maximal independent set of $G_i$.
2. If there is no independent set $I$ of $G_{i-1}$ such that $I - N(i)$ is not a superset of $I_{i-1} - N(i)$, then $I_{i-1} - N(i) \cup \{i\}$ is a maximal independent set of $G_i$.

The problem of listing all the cliques in a graph $G$ is equivalent to that of listing all the maximal independent sets of the complement $G^c$ of $G$.
So we can use Tsukiyama's algorithm to generate all maximal cliques in the given graph.

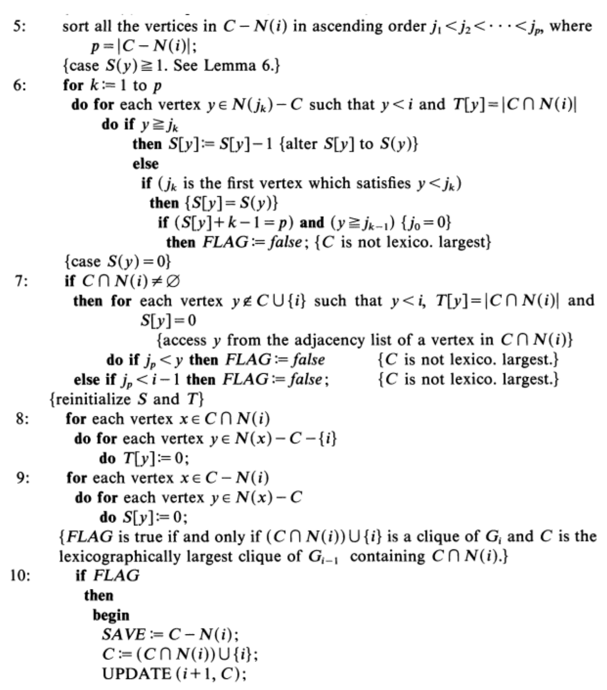
Thus the algorithm first defines the UPDATE function, which is based on recursion.

It takes two parameters $i$, and $C$. Here, $C$ is a maximal clique on the subgraph of $G_{i-1}$ induced by vertices $0, 1, \ldots, i-1$.

The base case is -
If $i = n$, then $C$ is a clique on the subgraph of $G_{n-1}$ induced by vertices $0, 1, \ldots, n-1$. Thus, $C$ is a maximal clique in the graph $G$.

Otherwise, -
If $C-N(i)$ is not empty, then $C$ is a clique for $G_i$ as well. So we now call UPDATE(i+1,C).

Then, we conduct maximality test and lexico. test. Maximality test checks whether the candidate of a new clique $(C \cap N(i)) \cup \{i\}$ is indeed a clique.

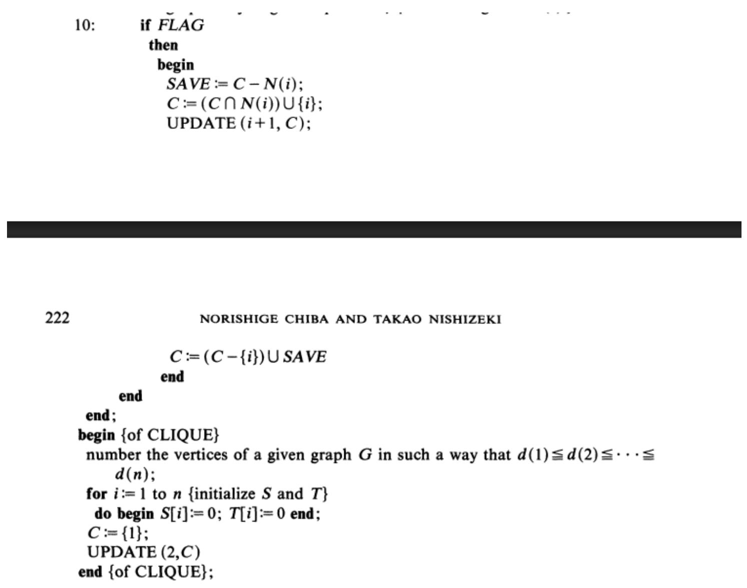


```
5:   sort all the vertices in $C - N(i)$ in ascending order $j_1 < j_2 < \cdots < j_p$, where
         $p = |C - N(i)|$;
     {case $S(y) \geqq 1$. See Lemma 6.}
6:   for $k := 1$ to $p$
       do for each vertex $y \in N(j_k) - C$ such that $y < i$ and $T[y] = |C \cap N(i)|$
           do if $y \geqq j_k$
               then $S[y] := S[y] - 1$ {alter $S[y]$ to $S(y)$}
               else
                 if ($j_k$ is the first vertex which satisfies $y < j_k$)
                   then {$S[y] = S(y)$}
                     if $(S[y] + k - 1 = p)$ and $(y \geqq j_{k-1})$ {$j_0 = 0$}
                       then $FLAG := false$; {$C$ is not lexico. largest}
     {case $S(y) = 0$}
7:   if $C \cap N(i) \neq \varnothing$
       then for each vertex $y \notin C \cup \{i\}$ such that $y < i$, $T[y] = |C \cap N(i)|$ and
                 $S[y] = 0$
                   {access $y$ from the adjacency list of a vertex in $C \cap N(i)$}
            do if $j_p < y$ then $FLAG := false$          {$C$ is not lexico. largest.}
       else if $j_p < i - 1$ then $FLAG := false$;        {$C$ is not lexico. largest.}
    {reinitialize $S$ and $T$}
8:   for each vertex $x \in C \cap N(i)$
       do for each vertex $y \in N(x) - C - \{i\}$
           do $T[y] := 0$;
9:   for each vertex $x \in C - N(i)$
       do for each vertex $y \in N(x) - C$
           do $S[y] := 0$;
     {$FLAG$ is true if and only if $(C \cap N(i)) \cup \{i\}$ is a clique of $G_i$ and $C$ is the
     lexicographically largest clique of $G_{i-1}$ containing $C \cap N(i)$.}
10:    if $FLAG$
         then
           begin
             $SAVE := C - N(i)$;
             $C := (C \cap N(i)) \cup \{i\}$;
             UPDATE $(i+1, C)$;
```

Figure 2: Algorithm part 2

The lexico. test checks whether C is the lexicographically largest clique of $G_{i-1}$ containing $C \cap N(i)$. This is to avoid generating duplicate cliques.

Thus, if the candidate clique $(C \cap N(i)) \cup i$ is indeed maximal and lexicographically greatest, then we call UPDATE(i+1,C).



```
10:       if FLAG
            then
              begin
                $SAVE := C - N(i)$;
                $C := (C \cap N(i)) \cup \{i\}$;
                UPDATE $(i+1, C)$;
```

222 NORISHIGE CHIBA AND TAKAO NISHIZEKI

```
                $C := (C - \{i\}) \cup SAVE$
              end
          end
  end;
begin {of CLIQUE}
  number the vertices of a given graph $G$ in such a way that $d(1) \leqq d(2) \leqq \cdots \leqq$
      $d(n)$;
  for $i := 1$ to $n$ {initialize $S$ and $T$}
    do begin $S[i] := 0$; $T[i] := 0$ end;
  $C := \{1\}$;
  UPDATE $(2, C)$
end {of CLIQUE};
```

Figure 3: Algorithm part 3

Now, in our main function, an important preprocessing step is numbering the vertices of a given graph $G$ in such a way that its arranged in a non-decreasing order of degree. This is what reduces the time complexity of the algorithm from what Tsukiyama proposed, $O(n * m)$ $n$=number of vertices, $m$=number of edges, to $O(a(G) * m)$ $a(G)$ = arboricity, $m$ = number of edges.

Then we set $C = \{0\}$. This is because $\{0\}$ is always a maximal clique in the graph $G_0$ induced by the vertex 0. And thus now we can use this recursive algorithm to generate all maximal cliques in the graph $G$ by calling UPDATE(1,C).

[Please note: the difference in the nubmering used in the algorithm and the explanation & code comes from the fact that while the vertices given in paper are 1-indexed, i.e. the vertices start from 1, what we have implemented follows 0-indexing, i.e. the vertices start from 0.]

## 1.3 Implementation

- $n$ is defined as the number of vertices that appear in the edge list. This means that if there are any vertices that have no edge incident on them, they're ignored. This is because a clique is of size $> 2$ and is completely connected. So a vertex that has no edge incident on it, can neither be a part of a clique, nor form a clique by itself.
- $C$ is stored as a vector of size $n$. Thus, if a vertex $i$ is present in $C$, $C[i] = 1$ and for any vertex $j$ not present in C, $C[j] = 0$.
- *adj* stores the adjacency list and is a vector of unordered sets. Here, for a vertex $i$, its neighbors $N(i)$ are stored in $adj[i]$ as an unordered_set.

- We define $n$, $C$, $T$, $S$ and *adj* (the adjacency list) as global variables to prevent multiple reallocations and improve efficiency. This also means that these don't need to be passed in the function call each time UPDATE is called recursively. ($S$ and $T$ are vectors that are used to conduct the maximality and lexico. tests.)

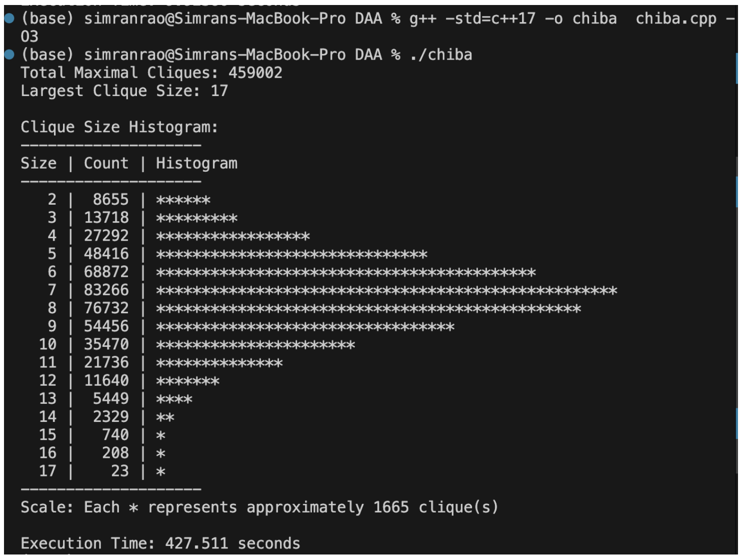
- To number the vertices according to the non-decreasing order of degree, the following steps are followed-

  1. The graph is read into a temporary adjacency list *tempadj* which is an unordered_map from the vertex number to the unordered_set containing its neighbors. For each edge $(a, b)$ read, we insert $b$ into $adj[a]$ as well as insert $a$ into $adj[b]$. This is to say that we treat the graph as undirected. The map and set ensure that if the input database gives the graph as a directed graph, no duplicates are inserted.
  2. Next, we transfer this into a vector *adjvec* and sort the vertices according to the size of the set of neighbors of each vertex. This means that the vertices appear in the vector in the non-decreasing order of degree.
  3. Now, we rename each vertex as $0, 1, \ldots, n-1$ in the order of their appearance. We store this mapping in the unordered_map *renumber*.
  4. Finally, we build the adjacency list *adj* by renaming each vertex in *adjvec* according to the mapping created.

- We use the `<ctime>` library to measure the execution time of this algorithm when run against the 3 datasets provided: Wiki-Vote, Email-Enron and Skitter.

### 1.4 Results

While the algorithm runs for both Wiki-Vote and Email-Enron and produces the following results, it fails to run on Skitter due to memory constraints. Because the algorithm is recursive, on large datasets like Skitter (which contains 1696415 vertices and 11095298 edges), it leads to stack-overflow.

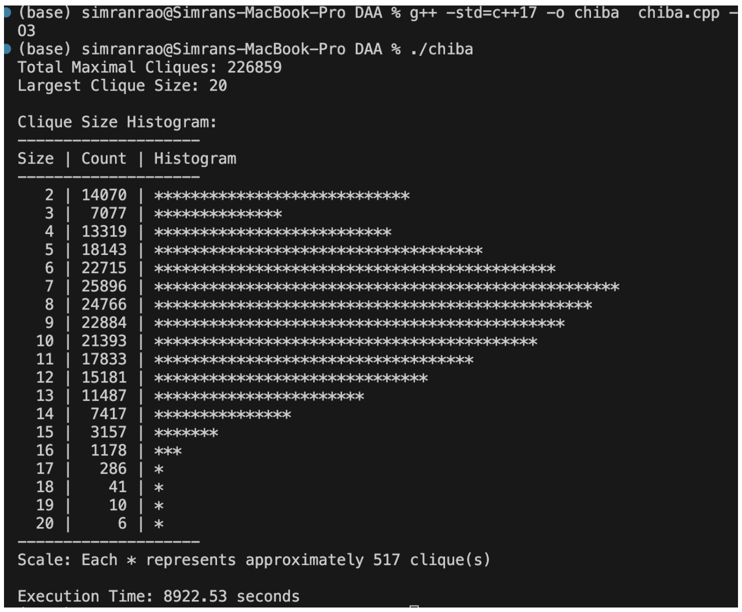


```
(base) simranrao@Simrans-MacBook-Pro DAA % g++ -std=c++17 -o chiba  chiba.cpp -
O3
(base) simranrao@Simrans-MacBook-Pro DAA % ./chiba
Total Maximal Cliques: 459002
Largest Clique Size: 17

Clique Size Histogram:
--------------------
Size | Count | Histogram
--------------------
   2 |  8655 | ******
   3 | 13718 | *********
   4 | 27292 | *****************
   5 | 48416 | ******************************
   6 | 68872 | ******************************************
   7 | 83266 | ***************************************************
   8 | 76732 | ***********************************************
   9 | 54456 | *********************************
  10 | 35470 | **********************
  11 | 21736 | **************
  12 | 11640 | *******
  13 |  5449 | ****
  14 |  2329 | **
  15 |   740 | *
  16 |   208 | *
  17 |    23 | *
--------------------
Scale: Each * represents approximately 1665 clique(s)

Execution Time: 427.511 seconds
```

Figure 4: Output when ran on Wiki-Vote dataset



```
(base) simranrao@Simrans-MacBook-Pro DAA % g++ -std=c++17 -o chiba  chiba.cpp -
O3
(base) simranrao@Simrans-MacBook-Pro DAA % ./chiba
Total Maximal Cliques: 226859
Largest Clique Size: 20

Clique Size Histogram:
--------------------
Size | Count | Histogram
--------------------
   2 | 14070 | ****************************
   3 |  7077 | **************
   4 | 13319 | **************************
   5 | 18143 | ************************************
   6 | 22715 | ********************************************
   7 | 25896 | ***************************************************
   8 | 24766 | ************************************************
   9 | 22884 | *********************************************
  10 | 21393 | ******************************************
  11 | 17833 | ***********************************
  12 | 15181 | ******************************
  13 | 11487 | ***********************
  14 |  7417 | ***************
  15 |  3157 | *******
  16 |  1178 | ***
  17 |   286 | *
  18 |    41 | *
  19 |    10 | *
  20 |     6 | *
--------------------
Scale: Each * represents approximately 517 clique(s)

Execution Time: 8922.53 seconds
```

Figure 5: Output when ran on Email-Enron dataset

## 2. The worst-case time complexity for generating all maximal cliques and computational experiments (Tomita et al, 2006).

This algorithm presents a depth-first search (DFS) approach for generating all maximal cliques in an undirected graph. It employs pruning techniques similar to those used in the Bron–Kerbosch algorithm. The algorithm processes input from a file containing the number of nodes and edges, followed by the edge list.

### 2.1 Preprocessing

Since the input vertices are not guaranteed to be between 0 and $n-1$, where $n$ is the total number of vertices, an unordered map is used to reindex the nodes between 0 and $n-1$. After this, the function `EXPAND` is invoked with parameters `SUBG` and `CAND`, both initially set to $V$ (the original set of vertices).

### 2.2 Algorithm Description

The `EXPAND` function operates as follows:

- If `SUBG` is empty, a maximal clique has been found. The algorithm updates the maximum clique size found so far and increments the clique count.
- Otherwise, a vertex $u \in$ `SUBG` is selected to maximize the size of `CAND` $-$ neighbors$(u)$. This set consists of vertices not adjacent to the vertex with the highest degree.
- Define $\mathrm{EXT}_u =$ `CAND` $-$ neighbors$(u)$. While $\mathrm{EXT}_u$ is non-empty:
    1. For each vertex $q \in \mathrm{EXT}_u$, add it to the clique by incrementing $Q$.
    2. Construct $\texttt{SUBG}_q$ and $\texttt{CAND}_q$ by intersecting `SUBG` and `CAND` with the neighbors of $q$.
    3. Remove $q$ from `CAND` to ensure that previously processed nodes are not reconsidered.
    4. Remove $q$ from the current clique.

### 2.3 Time Complexity

The worst-case time complexity for enumerating all maximal cliques is $O(3^{n/3})$, as shown in theoretical studies on the Bron–Kerbosch algorithm. The pruning strategies implemented reduce unnecessary recursive calls, improving practical performance.

### 2.4 Algorithm

The algorithm uses a global variable $Q$ representing a set of vertices that constitutes a complete subgraph found up to a particular time. The algorithm begins by letting $Q$ be an empty set, and expands $Q$ step by step by applying a recursive procedure `EXPAND` to $V$ and its succeeding induced subgraphs to search for larger and larger complete subgraphs until they reach maximal ones.

If $Q = \{p_1, p_2, \ldots, p_d\}$ is a complete subgraph found at some stage, we consider the set of vertices

$$\mathrm{SUBG} = V \cap (p_1) \cap (p_2) \cap \cdots \cap (p_d),$$

where SUBG $= V$ and $Q = \emptyset$ at the initial stage. We apply the procedure `EXPAND` to SUBG to search for larger complete subgraphs. If SUBG $= \emptyset$, then $Q$ is clearly a maximal complete subgraph, or a maximal clique. Otherwise, $Q \cup \{q\}$ is a larger complete subgraph for every $q \in$ SUBG.

Now, we consider the smaller subgraphs $G(\mathrm{SUBG}_q)$ that are induced by new sets of vertices $\mathrm{SUBG}_q = \mathrm{SUBG} \cap (q)$ for all $q \in$ SUBG, and we further apply recursively the same procedure `EXPAND` to $\mathrm{SUBG}_q$ to find larger complete subgraphs containing $Q \cup \{q\}$.

To prune unnecessary parts of the search forest, we regard the previously described set SUBG (initially empty) as an ordered set of vertices, and we continue to generate maximal cliques from the vertices in SUBG stepwise in this order. We let FINI be a subset of vertices of SUBG that have already been processed by the algorithm. We then denote the set of remaining candidates for expansion by CAND:

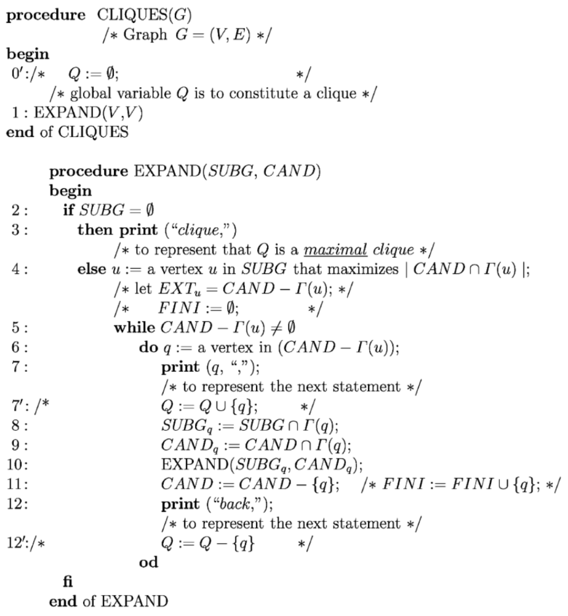
$$\mathrm{CAND} = \mathrm{SUBG} - \mathrm{FINI}.$$

Hence, we have SUBG = FINI$\cup$CAND with FINI$\cap$CAND $= \emptyset$. Thus, the pivot is selected from CAND and not SUBG, allowing us to optimize the algorithm by reducing the search space.

The subgraph $G(\mathrm{SUBG}_q)$ with $\mathrm{SUBG}_q$ is defined as

$$\mathrm{SUBG}_q = \mathrm{FINI}_q \cup \mathrm{CAND}_q \quad (\mathrm{FINI}_q \cap \mathrm{CAND}_q = \emptyset),$$

where $\mathrm{FINI}_q = \mathrm{FINI} \cap (q)$ and $\mathrm{CAND}_q = \mathrm{CAND} \cap (q)$. Thus, only the vertices in $\mathrm{CAND}_q$ can be candidates for expanding the complete subgraph $Q \cup \{q\}$ to find new larger cliques.



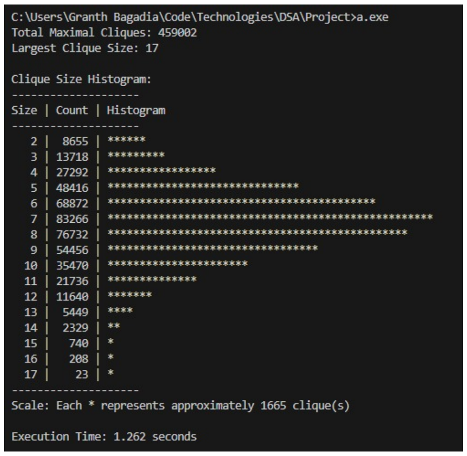
```
procedure  CLIQUES($G$)
                /* Graph  $G = (V, E)$ */
begin
 0′:/*    $Q := \emptyset$;                          */
        /* global variable $Q$ is to constitute a clique */
 1 : EXPAND($V$,$V$)
end of CLIQUES

        procedure EXPAND($SUBG$, $CAND$)
        begin
 2 :      if $SUBG = \emptyset$
 3 :         then print ("clique,")
                  /* to represent that $Q$ is a maximal clique */
 4 :         else $u$ := a vertex $u$ in $SUBG$ that maximizes $| CAND \cap \Gamma(u) |$;
                  /* let $EXT_u = CAND - \Gamma(u)$; */
                  /*     $FINI := \emptyset$;           */
 5 :              while $CAND - \Gamma(u) \neq \emptyset$
 6 :                  do $q$ := a vertex in $(CAND - \Gamma(u))$;
 7 :                     print ($q$, ",");
                         /* to represent the next statement */
 7′: /*                  $Q := Q \cup \{q\}$;        */
 8 :                     $SUBG_q := SUBG \cap \Gamma(q)$;
 9 :                     $CAND_q := CAND \cap \Gamma(q)$;
10:                      EXPAND($SUBG_q, CAND_q$);
11:                      $CAND := CAND - \{q\}$;    /* $FINI := FINI \cup \{q\}$; */
12:                      print ("back,");
                         /* to represent the next statement */
12′:/*                   $Q := Q - \{q\}$        */
                     od
          fi
        end of EXPAND
```

Figure 6: Algorithm

### 2.5 Results

The algorithm runs for all datasets and produces the following results.

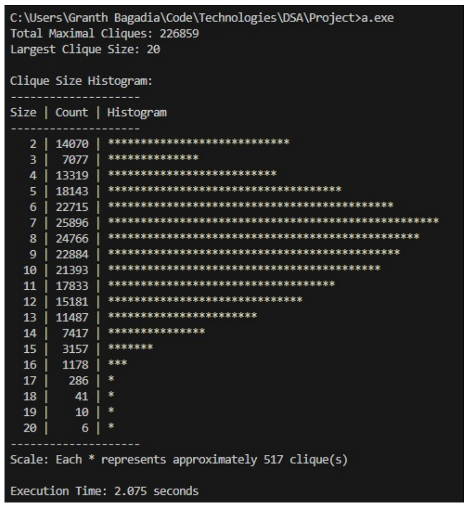


```
C:\Users\Granth Bagadia\Code\Technologies\DSA\Project>a.exe
Total Maximal Cliques: 459002
Largest Clique Size: 17

Clique Size Histogram:
--------------------
Size | Count | Histogram
--------------------
   2 |  8655 | ******
   3 | 13718 | *********
   4 | 27292 | *****************
   5 | 48416 | ******************************
   6 | 68872 | ******************************************
   7 | 83266 | ***************************************************
   8 | 76732 | ***********************************************
   9 | 54456 | *********************************
  10 | 35470 | **********************
  11 | 21736 | **************
  12 | 11640 | *******
  13 |  5449 | ****
  14 |  2329 | **
  15 |   740 | *
  16 |   208 | *
  17 |    23 | *
--------------------
Scale: Each * represents approximately 1665 clique(s)

Execution Time: 1.262 seconds
```

Figure 7: Output when ran on Wiki-Vote dataset

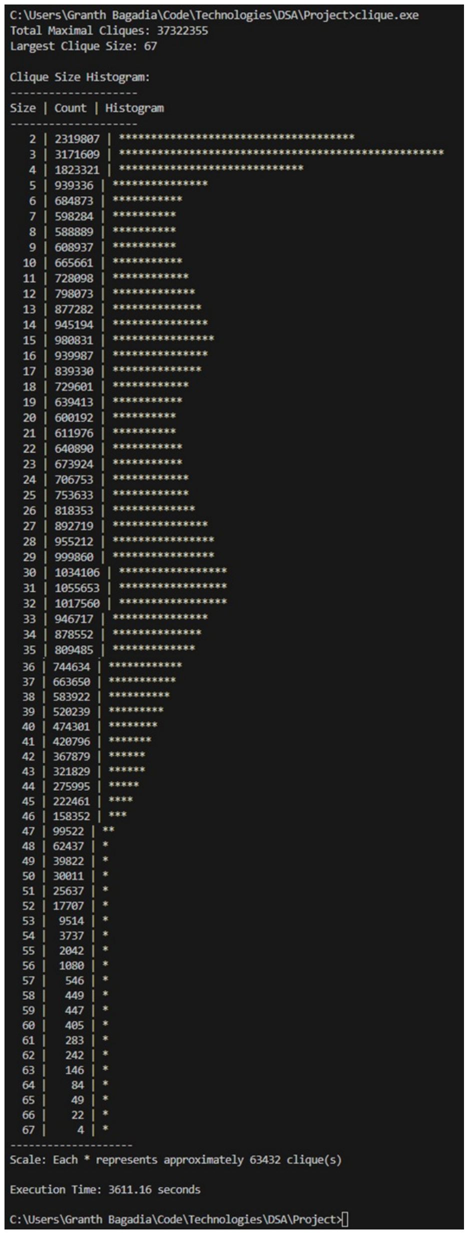


```
C:\Users\Granth Bagadia\Code\Technologies\DSA\Project>a.exe
Total Maximal Cliques: 226859
Largest Clique Size: 20

Clique Size Histogram:
--------------------
Size | Count | Histogram
--------------------
   2 | 14070 | ****************************
   3 |  7077 | **************
   4 | 13319 | **************************
   5 | 18143 | ************************************
   6 | 22715 | ********************************************
   7 | 25896 | ***************************************************
   8 | 24766 | ************************************************
   9 | 22884 | *********************************************
  10 | 21393 | ******************************************
  11 | 17833 | ***********************************
  12 | 15181 | ******************************
  13 | 11487 | ***********************
  14 |  7417 | ***************
  15 |  3157 | *******
  16 |  1178 | ***
  17 |   286 | *
  18 |    41 | *
  19 |    10 | *
  20 |     6 | *
--------------------
Scale: Each * represents approximately 517 clique(s)

Execution Time: 2.075 seconds
```

Figure 8: Output when ran on Email-Enron dataset



```
C:\Users\Granth Bagadia\Code\Technologies\DSA\Project>clique.exe
Total Maximal Cliques: 37322355
Largest Clique Size: 67

Clique Size Histogram:
--------------------
Size | Count | Histogram
--------------------
   2 | 2319807 | *************************************
   3 | 3171609 | **************************************************
   4 | 1823321 | *****************************
   5 | 939336 | ***************
   6 | 684873 | ***********
   7 | 598284 | **********
   8 | 588889 | **********
   9 | 608937 | **********
  10 | 665661 | ***********
  11 | 728098 | ************
  12 | 798073 | *************
  13 | 877282 | **************
  14 | 945194 | ***************
  15 | 980831 | ****************
  16 | 939987 | ***************
  17 | 839330 | **************
  18 | 729601 | ************
  19 | 639413 | ***********
  20 | 600192 | **********
  21 | 611976 | **********
  22 | 640890 | ***********
  23 | 673924 | ***********
  24 | 706753 | ************
  25 | 753633 | ************
  26 | 818353 | *************
  27 | 892719 | ***************
  28 | 955212 | ****************
  29 | 999860 | ****************
  30 | 1034106 | *****************
  31 | 1055653 | *****************
  32 | 1017560 | *****************
  33 | 946717 | ***************
  34 | 878552 | **************
  35 | 809485 | *************
  36 | 744634 | ************
  37 | 663650 | ***********
  38 | 583922 | **********
  39 | 520239 | *********
  40 | 474301 | ********
  41 | 420796 | *******
  42 | 367879 | ******
  43 | 321829 | ******
  44 | 275995 | *****
  45 | 222461 | ****
  46 | 158352 | ***
  47 | 99522 | **
  48 | 62437 | *
  49 | 39822 | *
  50 | 30011 | *
  51 | 25637 | *
  52 | 17707 | *
  53 |  9514 | *
  54 |  3737 | *
  55 |  2042 | *
  56 |  1080 | *
  57 |   546 | *
  58 |   449 | *
  59 |   447 | *
  60 |   405 | *
  61 |   283 | *
  62 |   242 | *
  63 |   146 | *
  64 |    84 | *
  65 |    49 | *
  66 |    22 | *
  67 |     4 | *
--------------------
Scale: Each * represents approximately 63432 clique(s)

Execution Time: 3611.16 seconds

C:\Users\Granth Bagadia\Code\Technologies\DSA\Project>
```

Figure 9: Output when ran on as-skitter dataset

## 3. Listing All Maximal Cliques in Sparse Graphs in Near-optimal Time

We present an algorithm for listing all maximal cliques in sparse graphs with improved time complexity. While the worst-case time complexity for this problem is generally $O(3^{n/3})$ for a graph with $n$ vertices, our approach achieves near-optimal time in sparse graphs characterized by their degeneracy. Our algorithm builds on the Bron-Kerbosch algorithm with pivoting and offers practical improvements for real-world graph analysis.

### 3.1 Time Complexity

The worst-case time complexity for enumerating all maximal cliques in any graph is $O(3^{n/3})$, matching the maximum possible number of maximal cliques in an $n$-vertex graph. However, for sparse graphs with degeneracy $d$, our algorithm achieves $O(d \cdot n \cdot 3^{d/3})$ time complexity.

This improvement results from:

- Processing each vertex exactly once in the outer loop ($n$ vertices).
- Restricting the candidate set $P$ to at most $d$ vertices per call due to the degeneracy ordering.
- The Bron-Kerbosch algorithm with pivoting requires $O(3^{|P|/3})$ time in the worst case.

For graphs with bounded degeneracy (e.g., planar graphs where $d \leq 5$), this results in near-optimal time complexity that is significantly better than the general case.

### 3.2 Algorithm

Our algorithm begins with a preprocessing step that orders the vertices of the graph in a degeneracy ordering. The degeneracy ordering arranges vertices such that each vertex has a limited number of neighbors that come later in the ordering, which is crucial for bounding the algorithm's complexity.

**Algorithm 1** ComputeDegeneracyOrdering

```
Input: Graph G = (V, E)
Output: Degeneracy ordering of vertices and the graph's degeneracy value k

1. Initialize array bin[0...max_degree] where bin[i] contains vertices of degree i
     • This bin-based structure allows efficient retrieval of the minimum degree vertex
2. Initialize array pos[1...$|V|$] to store position of each vertex in ordering
     • Initially all values are 0, indicating vertices are not yet in the ordering
3. Initialize array deg[1...$|V|$] to store current degree of each vertex
     • This will be updated as vertices are removed from consideration
4. Initialize array ordering[1...$|V|$] to store the degeneracy ordering
5. k ← 0                                   // k will track the degeneracy of the graph
6. For each vertex v ∈ V:
   (a) deg[v] ← degree of v
   (b) Add v to bin[deg[v]]
7. For i = 0 to $|V|$ - 1: // Process all vertices in order of increasing current degree
   (a) While bin[i] is empty:
         i. i ← i + 1                                          // Skip empty bins
   (b) v ← any vertex from bin[i]          // Select a vertex of minimum degree
   (c) Remove v from bin[i]
   (d) ordering[$|V|$ - i] ← v               // Add v to the end of current ordering
   (e) pos[v] ← $|V|$ - i                     // Record position of v in the ordering
   (f) k ← max(k, i)                           // Update degeneracy if necessary
   (g) For each neighbor u of v:
         i. If pos[u] = 0:                   // If u has not been processed yet
            A. Remove u from bin[deg[u]]
            B. deg[u] ← deg[u] - 1        // Removal of v reduces degree of u
            C. Add u to bin[deg[u]]          // Place u in the appropriate bin
8. Return ordering, k                                 // k is the degeneracy of G
```

The main algorithm leverages this degeneracy ordering to efficiently enumerate all maximal cliques by using a modified version of the Bron-Kerbosch algorithm with pivoting:

**Algorithm 2** DegeneracyMaximalCliques

**Input:** Graph G = (V, E)
**Output:** All maximal cliques in G

```
1. ordering, d ← ComputeDegeneracyOrdering(G)
2. For i = 1 to $|V|$:                 // Process vertices in degeneracy order
    (a) v ← ordering[i]
    (b) R ← {v}                        // Initialize current clique with vertex v
    (c) P ← {u ∈ N(v) — pos[u] ¿ pos[v]}  // Consider only neighbors that come later
    (d) X ← {u ∈ N(v) — pos[u] ¡ pos[v]}  // Neighbors already processed
    (e) BronKerboschPivot(R, P, X)     // Find all maximal cliques containing v
```

**Algorithm 3** Procedure BronKerboschPivot(R, P, X)

**Input:** R - current clique being built, P - candidate vertices, X - excluded vertices
**Output:** All maximal cliques that can be formed from R and vertices in P

```
1. If P = ∅ and X = ∅:      // Base case: no more candidates and exclusions
    (a) Report R as a maximal clique    // R cannot be extended further
    (b) Return
2. Choose a pivot vertex u from P ∪ X to maximize $|P \cap N(u)|$    // Optimization to reduce recursive calls
3. For each vertex v in P \ N(u):      // Consider only non-neighbors of the pivot
    (a) BronKerboschPivot(R ∪ {v}, P ∩ N(v), X ∩ N(v))    // Extend R with v
    (b) P ← P \ {v}                    // Remove v from candidates
    (c) X ← X ∪ {v}                    // Add v to exclusions for future iterations
```

### 3.3 Pivot Selection

The efficiency of our algorithm depends significantly on the pivot selection strategy. A good pivot minimizes the number of recursive calls by excluding as many vertices as possible from consideration. We select a pivot vertex $u$ from $P \cup X$ that maximizes the size of $P \cap N(u)$ - the set of candidate vertices adjacent to u.

**Algorithm 4** ChoosePivot

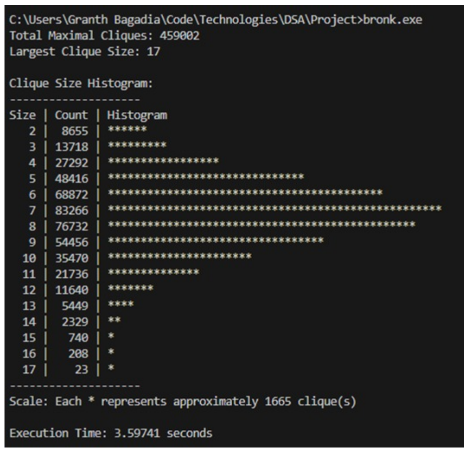
**Input:** Sets P (candidate vertices) and X (excluded vertices)
**Output:** Pivot vertex that maximizes $|P \cap N(u)|$

```
1. max_count ← -1                  // Track highest number of connections found
2. pivot ← null                    // Best pivot vertex so far
3. For each vertex u in P ∪ X:     // Try each candidate and excluded vertex
   (a) count ← 0                   // Count connections to candidate set
   (b) For each vertex v in P:
       i. If (u, v) ∈ E:           // If u is adjacent to candidate v
          A. count ← count + 1     // Increment count of neighbors
   (c) If count > max_count:       // Found better pivot
       i. max_count ← count        // Update max count
       ii. pivot ← u               // Update best pivot
4. Return pivot                    // Return vertex that will eliminate most recursive calls
```

The pivot strategy exploits a fundamental property of cliques: any maximal clique containing vertex v must either include the pivot u or contain at least one vertex not adjacent to u. Since a pivot with many connections to the candidate set P eliminates many potential branches (we only consider vertices in P that are not neighbors of the pivot), selecting the pivot that maximizes these connections provides the greatest pruning effect.

### 3.4 Results

The algorithm runs for all datasets and produces the following results.

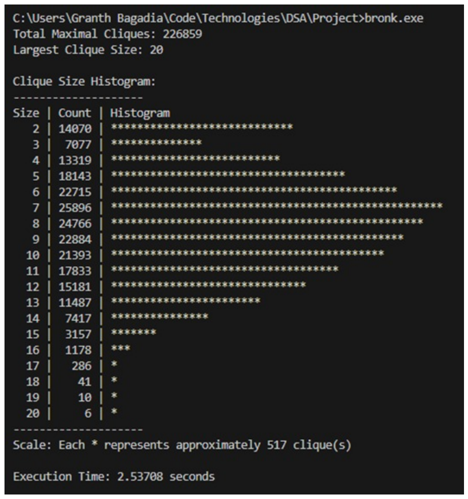


```
C:\Users\Granth Bagadia\Code\Technologies\DSA\Project>bronk.exe
Total Maximal Cliques: 459002
Largest Clique Size: 17

Clique Size Histogram:
--------------------
Size | Count | Histogram
   2 |  8655 | ******
   3 | 13718 | *********
   4 | 27292 | *****************
   5 | 48416 | ******************************
   6 | 68872 | ******************************************
   7 | 83266 | ***************************************************
   8 | 76732 | ***********************************************
   9 | 54456 | *********************************
  10 | 35470 | **********************
  11 | 21736 | **************
  12 | 11640 | *******
  13 |  5449 | ****
  14 |  2329 | **
  15 |   740 | *
  16 |   208 | *
  17 |    23 | *
--------------------
Scale: Each * represents approximately 1665 clique(s)

Execution Time: 3.59741 seconds
```

Figure 10: Output when ran on Wiki-Vote dataset

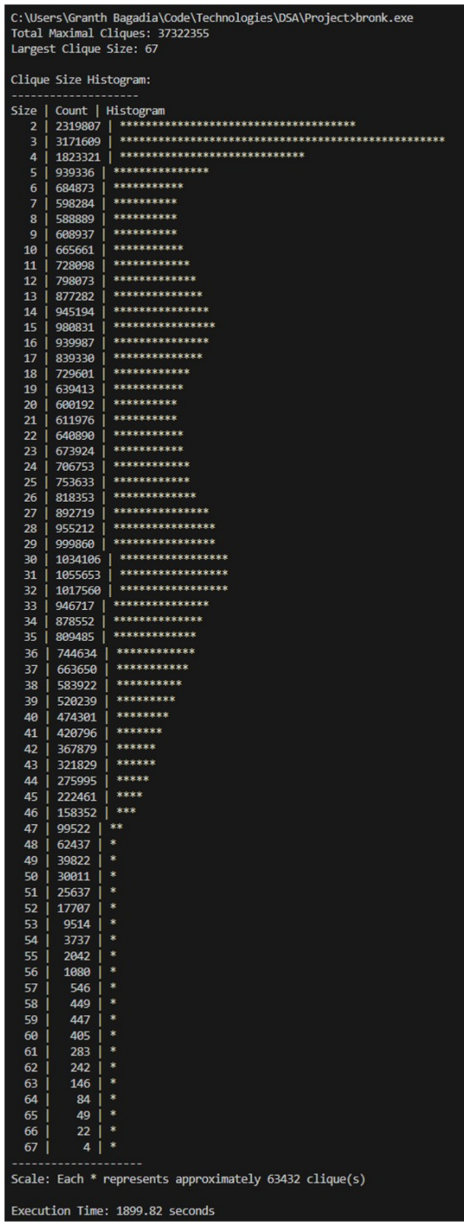


```
C:\Users\Granth Bagadia\Code\Technologies\DSA\Project>bronk.exe
Total Maximal Cliques: 226859
Largest Clique Size: 20

Clique Size Histogram:
--------------------
Size | Count | Histogram
   2 | 14070 | ****************************
   3 |  7077 | **************
   4 | 13319 | **************************
   5 | 18143 | ************************************
   6 | 22715 | ********************************************
   7 | 25896 | ***************************************************
   8 | 24766 | ************************************************
   9 | 22884 | *********************************************
  10 | 21393 | ******************************************
  11 | 17833 | ***********************************
  12 | 15181 | ******************************
  13 | 11487 | ***********************
  14 |  7417 | ***************
  15 |  3157 | *******
  16 |  1178 | ***
  17 |   286 | *
  18 |    41 | *
  19 |    10 | *
  20 |     6 | *
--------------------
Scale: Each * represents approximately 517 clique(s)

Execution Time: 2.53708 seconds
```

Figure 11: Output when ran on Email-Enron dataset



```
C:\Users\Granth Bagadia\Code\Technologies\DSA\Project>bronk.exe
Total Maximal Cliques: 37322355
Largest Clique Size: 67

Clique Size Histogram:
--------------------
Size | Count | Histogram
   2 | 2319807 | *************************************
   3 | 3171609 | ***************************************************
   4 | 1823321 | *****************************
   5 | 939336 | ***************
   6 | 684873 | ***********
   7 | 598284 | **********
   8 | 588889 | **********
   9 | 608937 | **********
  10 | 665661 | ***********
  11 | 728098 | ************
  12 | 798073 | *************
  13 | 877282 | **************
  14 | 945194 | ***************
  15 | 980831 | ****************
  16 | 939987 | ***************
  17 | 839330 | **************
  18 | 729601 | ************
  19 | 639413 | ***********
  20 | 600192 | **********
  21 | 611976 | **********
  22 | 640890 | ***********
  23 | 673924 | ***********
  24 | 706753 | ************
  25 | 753633 | ************
  26 | 818353 | *************
  27 | 892719 | ***************
  28 | 955212 | ****************
  29 | 999860 | ****************
  30 | 1034106 | *****************
  31 | 1055653 | *****************
  32 | 1017560 | *****************
  33 | 946717 | ***************
  34 | 878552 | **************
  35 | 809485 | *************
  36 | 744634 | ************
  37 | 663650 | ***********
  38 | 583922 | **********
  39 | 520239 | *********
  40 | 474301 | ********
  41 | 420796 | *******
  42 | 367879 | ******
  43 | 321829 | ******
  44 | 275995 | *****
  45 | 222461 | ****
  46 | 158352 | ***
  47 | 99522 | **
  48 | 62437 | *
  49 | 39822 | *
  50 | 30011 | *
  51 | 25637 | *
  52 | 17707 | *
  53 |  9514 | *
  54 |  3737 | *
  55 |  2042 | *
  56 |  1080 | *
  57 |   546 | *
  58 |   449 | *
  59 |   447 | *
  60 |   405 | *
  61 |   283 | *
  62 |   242 | *
  63 |   146 | *
  64 |    84 | *
  65 |    49 | *
  66 |    22 | *
  67 |     4 | *
--------------------
Scale: Each * represents approximately 63432 clique(s)

Execution Time: 1899.82 seconds
```

Figure 12: Output when ran on as-skitter dataset

## Conclusion

This project demonstrates the implementation of key graph algorithms and their complexities while presenting them via a website.

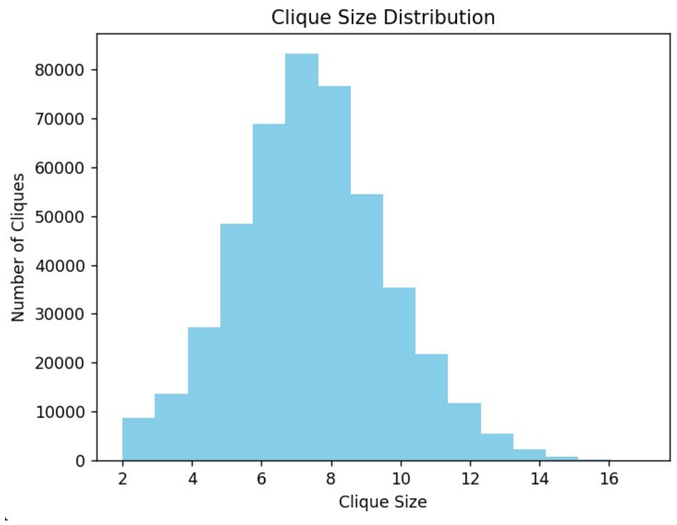


Figure 13: Histogram for Wiki-Vote dataset

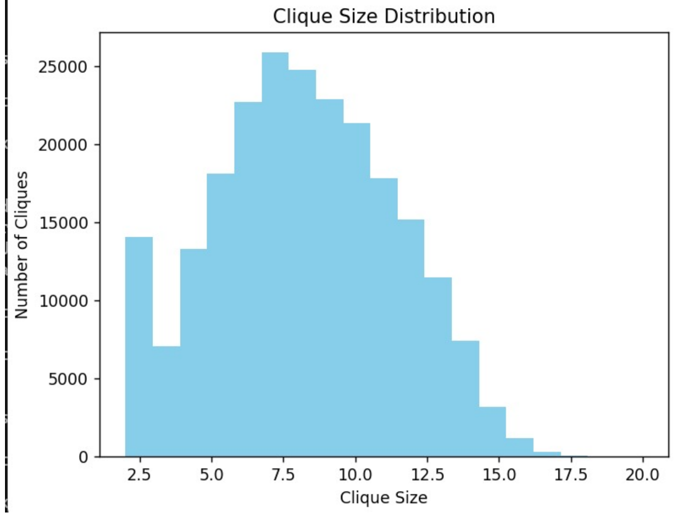


Figure 14: Histogram for Email-Enron dataset

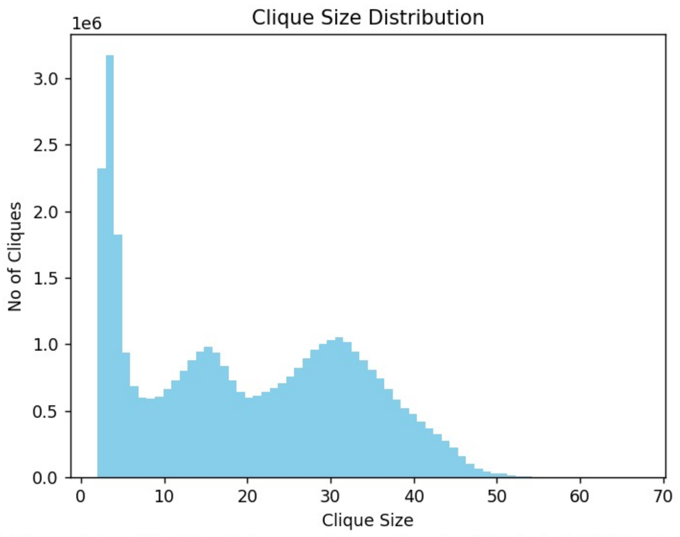


Figure 15: Histogram for Skitter dataset

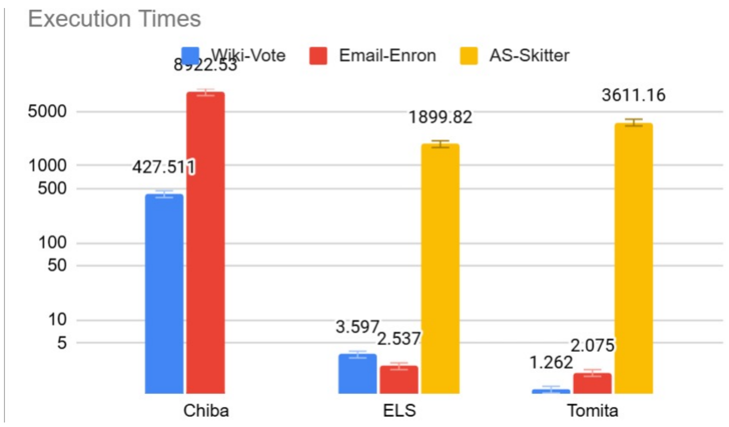


Figure 16: Histogram for the Execution times on each dataset

## Student Information

| ID | Name | Roll Number |
|---|---|---|
| 1 | Simran Sesha Rao | 2022A7PS0002H |
| 2 | Simran Singh | 2022A7PS0003H |
| 3 | Shreya Kunjankumar Mehta | 2022A7PS0115H |
| 4 | Sukhbodhanand Tripathi | 2022A7PS0187H |
| 5 | Granth Bagadia | 2022A7PS0217H |