<u>CS F364 ASSIGNMENT 1</u>

COMPARATIVE ANALYSIS OF MAXIMAL CLIQUE LISTING ALGORITHMS

BY

Name(s) of Student(s)   ID.No.(s)   Email
Nishit Nilay 2022A7PS0230H f20220230@hyderabad.bits-pilani.ac.in
Rohan Kakade 2022A7PS0216H
f20220216@hyderabad.bits-pilani.ac.in Rithvik Raajha 2022A7PS1361H
f20221361@hyderabad.bits-pilani.ac.in Vatsal Tyagi 2022A7PS1388H
f20221388@hyderabad.bits-pilani.ac.in Aditya Narwania 2022A7PS1540H
f20221540@hyderabad.bits-pilani.ac.in

BIRLA INSTITUTE OF TECHNOLOGY AND SCIENCE, PILANI

Introduction

Enumerating all **maximal cliques** in an undirected graph is a fundamental problem in graph theory, with a wide range of applications in social network analysis, bioinformatics, and beyond. Over the years, various algorithms have been proposed to address the combinatorial explosion that arises in the worst case. Three especially noteworthy approaches are:

1. **Chiba–Nishizeki Algorithms**
   These algorithms leverage the concept of **arboricity** (or equivalently **degeneracy**, up to constant factors) to list subgraphs, including maximal cliques, in time proportional to parameters that capture how "sparse" a graph is. Their methods demonstrate that even though enumerating maximal cliques can be exponential in the worst case, it can be done more efficiently for many sparse graphs.

2. **Tomita–Tanaka–Takahashi (TTT) Algorithm**
   This approach refines the classic **Bron–Kerbosch** depth-first search framework by adding pruning rules that avoid redundant expansions. The TTT algorithm attains an optimal worst-case running time of O(3n/3)O(3^{n/3})O(3n/3) for an nnn-vertex graph when output is printed in a condensed form. It has been shown both theoretically and empirically to perform very efficiently in practice.

3. **Pivot-Based Bron–Kerbosch Algorithm**
   The Bron–Kerbosch method is a classic backtracking procedure that systematically grows candidate cliques. The "pivoting" enhancement, popularized by Tomita et al., picks a carefully chosen vertex (the pivot) in each recursive step, thereby reducing the branching factor. With the right pivot selection strategy, this algorithm achieves worst-case–optimal performance among pivot-based backtracking methods.

Together, these algorithms illustrate the breadth of techniques—from graph degeneracy exploitation to careful pivoting and pruning—that make **maximal clique enumeration** tractable on many real-world or otherwise structured graphs.

Abstract

This paper presents an overview of three influential algorithms for enumerating all maximal cliques in undirected graphs, with a focus on applications to sparse graphs. The first approach, stemming from the work of Chiba and Nishizeki, exploits the graph's arboricity—a measure closely related to its degeneracy—to efficiently list subgraphs and maximal cliques in time proportional to the sparsity parameter. Next, the Tomita–Tanaka–Takahashi (TTT) algorithm refines the classic Bron–Kerbosch procedure by incorporating effective pruning strategies, achieving worst-case optimal performance of O(3n/3)O(3^{n/3})O(3n/3) for an nnn-vertex graph when a compact output representation is used. Finally, the pivot-based Bron–Kerbosch algorithm is examined; by selecting a pivot vertex at each recursive step, it significantly reduces the branching factor and ensures that each maximal clique is reported exactly once. Together, these methods highlight a range of techniques—from leveraging degeneracy and arboricity to strategic pivoting—that render maximal clique enumeration tractable in many practical settings, particularly for graphs exhibiting sparse structure.

Tomita’s Algorithm for Maximal Clique Enumeration

Overview

Tomita’s algorithm is a practical method for finding all maximal cliques in an undirected graph. A **clique** is a set of vertices where every two distinct vertices are connected, and a **maximal clique** is one that cannot be enlarged by adding another vertex. This algorithm uses a depth-first search approach along with a clever pivoting strategy to reduce unnecessary work.

Key Concepts

1. **Current Clique (R):**
   The set of vertices that have been chosen so far to form a clique.

2. **Candidates (P):**
   The vertices that can still be added to the current clique (they are adjacent to all vertices in RRR).

3. **Excluded (X):**
   Vertices that have already been considered (or “processed”) in the current branch to avoid duplicates.

At the start, the current clique RRR is empty, and all vertices are available as candidates:

- R=∅R = \varnothingR=∅
- P=VP = VP=V (all vertices)
- X=∅X = \varnothingX=∅

Main Idea

The algorithm works recursively. At each step:

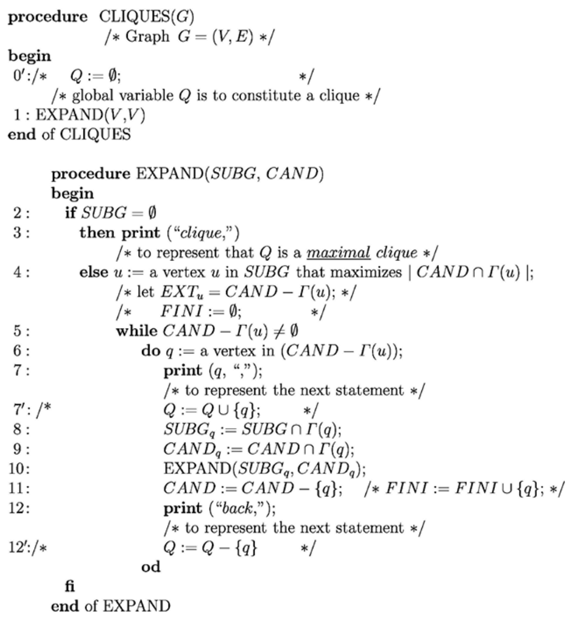
1. **If** both PPP and XXX are empty, then RRR is a maximal clique, and it is output.

2. **Otherwise,** a **pivot** vertex uuu is chosen from the set P∪XP \cup XP∪X. The pivot is selected so that many vertices in PPP are its neighbors.

3. The algorithm then considers only those vertices in PPP that are *not* neighbors of uuu. The intuition is that if a maximal clique does not include the pivot, then it must include a

vertex that is not adjacent to the pivot.

4. For each such vertex vvv, the algorithm:
    - Adds vvv to the current clique RRR.
    - Updates the candidate set PPP and excluded set XXX by keeping only those vertices that are also neighbors of vvv.
    - Recursively repeats the process with the new sets.
5. After processing vvv, it is removed from PPP and added to XXX to prevent it from being reconsidered in the same branch.



```
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
```

Fig. 2. Algorithm CLIQUES.

Explanation of the Pseudocode

6. **Base Case:**
   When both PPP and XXX are empty, no further vertices can be added to RRR. At this point, RRR is a maximal clique and is output.
7. **Pivot Selection:**
   The `choose_pivot` function selects a pivot uuu from P∪XP \cup XP∪X that maximizes the number of neighbors in PPP. This choice helps limit the recursive

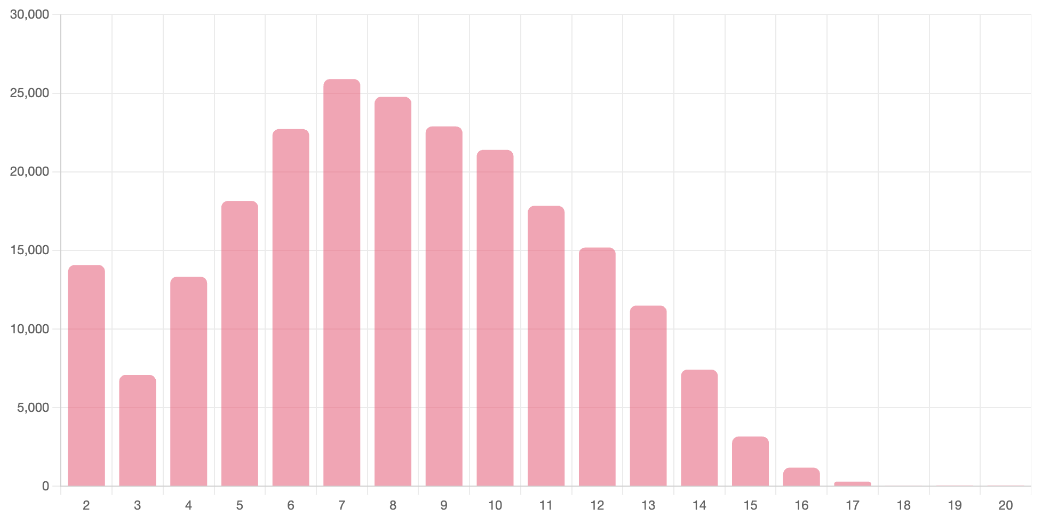
calls to only those vertices that are not neighbors of uuu.

8. **Recursive Expansion:**
   For each vertex vvv in P∖Γ(u)P \setminus \Gamma(u)P∖Γ(u):
   - vvv is added to the current clique RRR.
   - The candidate set PPP and the excluded set XXX are updated by taking their intersection with the neighbors of vvv. This ensures that all vertices in the new sets remain adjacent to every vertex in the new clique R∪{v}R \cup \{v\}R∪{v}.
   - The algorithm recurses with these new sets.
9. **Backtracking:**
   After exploring all cliques containing vvv, vvv is removed from PPP and added to XXX to ensure it is not considered again in the same branch.

   -

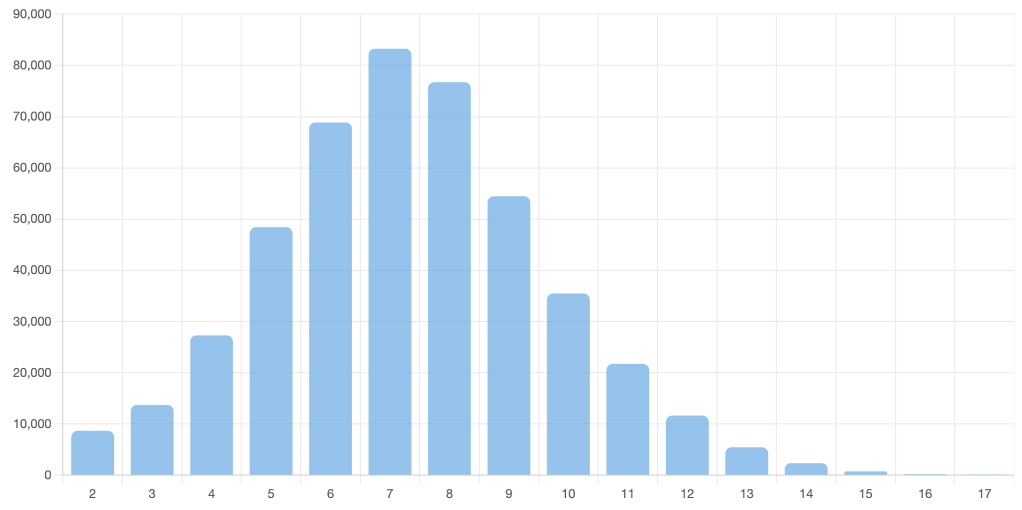
Email-Enron Output

1. Largest Clique Size: 20
2. Total Max Cliques: 226859
3. Time Taken: 150.52 seconds
4. Distribution of clique sizes:



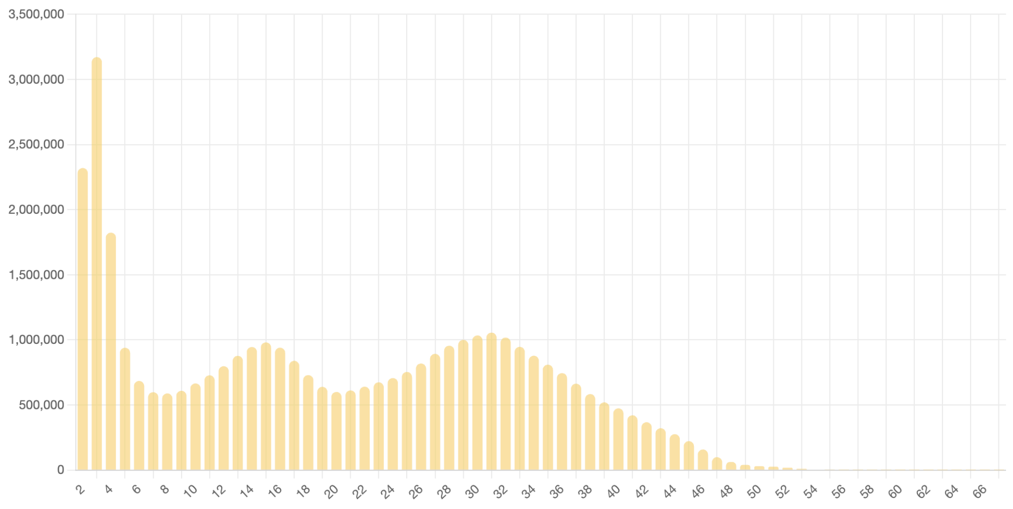
Wiki-Vote Output

1. Largest Clique Size: 17
2. Total Max Cliques: 459002
3. Time Taken: 30.58 seconds
4. Distribution of clique sizes:



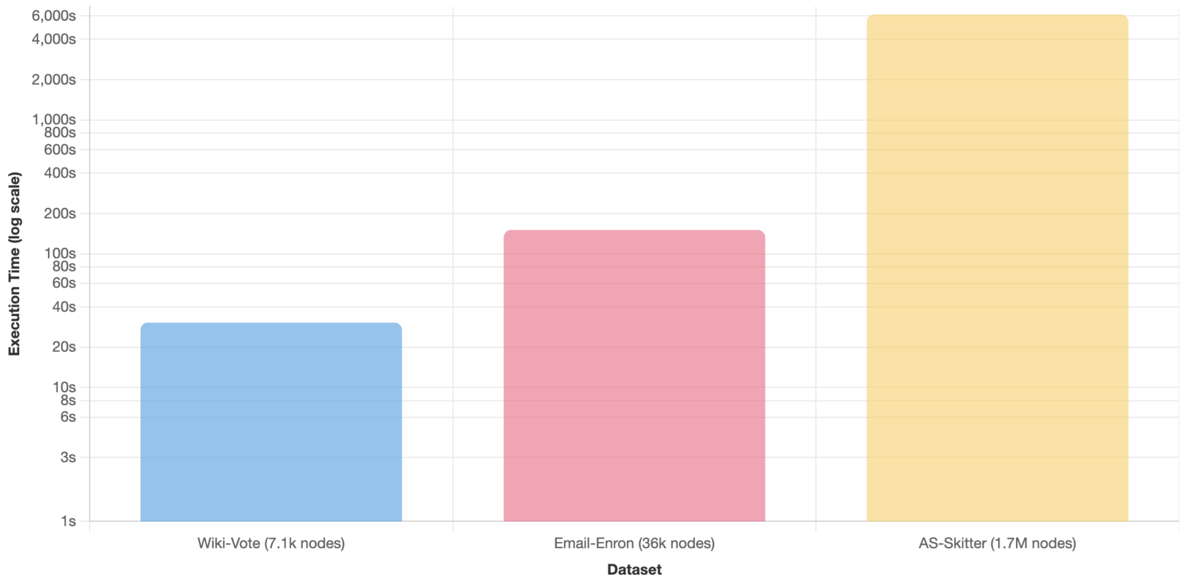
AS-Skitter Output

1. Largest Clique Size: 67
2. Total Max Cliques: 37322355
3. Time Taken: 6127.37 seconds
4. Distribution of clique sizes:



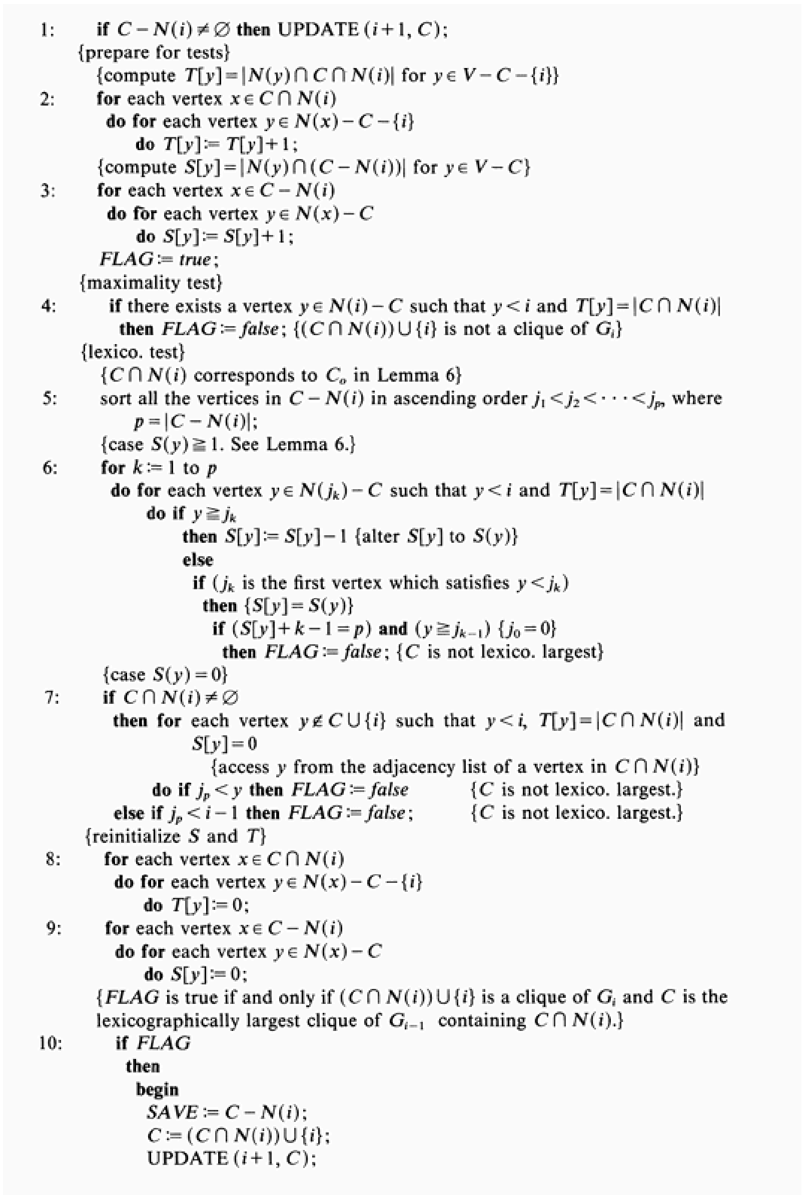
Execution Time Comparison

This chart compares the total execution time of the Tomita's EXPAND algortihm across different datasets.



<u>Chiba:</u>

The algorithm is designed to **enumerate all maximal cliques** in an undirected graph G=(V,E)G = (V, E)G=(V,E). A **maximal clique** is a set of vertices that are all mutually adjacent and cannot be enlarged by adding any other vertex from the graph. Unlike simpler backtracking methods, this algorithm incorporates two crucial checks—**maximality** and **lexicographic ordering**—to avoid duplicates and to systematically explore all possibilities.



```
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.}
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 $C \cap N(i) \neq \emptyset$
        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 $(i+1, C)$;
```

Key Features

1. **Vertex Ordering**
   The algorithm first **sorts** or **indexes** the vertices in a nondecreasing degree order:
   d(1)≤d(2)≤⋯≤d(n).d(1) \le d(2) \le \cdots \le d(n).d(1)≤d(2)≤⋯≤d(n).
   This ordering helps reduce the branching factor and provides a consistent framework for the subsequent checks.

2. **Candidate Clique CCC**
   A set CCC (the "candidate clique") is maintained throughout the recursion. At each step, the algorithm decides whether to include a particular vertex iii into CCC or skip it.

3. **Recursive Function `UPDATE(i, C)`**

   - **Parameters:**

     1. iii: The current vertex index being considered.

     2. CCC: The current clique we have built so far.

   - **Behavior:**

     1. **Exclude iii**: The algorithm can first explore the possibility of not adding iii at all (essentially calling `UPDATE(i+1, C)`).

     2. **Include iii**: If certain tests pass (detailed below), the algorithm adds iii to CCC (with some modifications to ensure all vertices in CCC are neighbors of iii) and recurses to the next vertex.

4. **Maximality and Lexicographic Tests**

   - **Maximality Test:** Ensures that adding iii to CCC indeed yields a larger clique that is not already accounted for by some other vertex's adjacency. If a vertex yyy outside CCC can mimic the same connections, that might mean the clique is not unique or truly maximal.

   - **Lexicographic Test:** Imposes a strict ordering condition so each maximal clique is generated **only once** in a consistent "lexicographic" order. If the candidate clique fails this ordering criterion, the algorithm does not proceed along that branch.

5. **Counters (T[y]T[y]T[y] and S[y]S[y]S[y])**
 Two arrays (or mappings) are used internally for the tests:
   - T[y]T[y]T[y]: Tracks adjacency relationships between yyy and vertices in CCC (or C∩N(i)C \cap N(i)C∩N(i)). Helps confirm if adding iii remains a valid extension.
   - S[y]S[y]S[y]: Tracks adjacency relative to the vertices in (C−N(i))(C - N(i))(C−N(i)). Helps ensure the resulting clique meets the lexicographic ordering rules.
6. **FLAG Variable**
 A boolean flag (`FLAG`) is set to **false** if either the maximality test or the lexicographic test fails. Only if `FLAG` remains **true** does the algorithm actually update CCC to include iii.
7. **Backtracking**
 After recursively exploring the branch where iii is included in CCC, the algorithm **backtracks**, restoring CCC to its original state. This ensures all possible branches of the search are eventually explored without mixing partial states.

How It All Fits Together

1. **Initialize:** Sort the vertices by degree and start with C=∅C = \varnothingC=∅.
2. **Recursive Exploration:** The procedure `UPDATE(i, C)` tries both “excluding iii” and “including iii” (if valid).
3. **Checks:** Each time we consider including iii, the algorithm does:

- **Maximality Test:** Uses T[y]T[y]T[y] to verify no outside vertex yyy duplicates or supersedes the new clique.
- **Lexicographic Test:** Uses S[y]S[y]S[y] to maintain a strict ordering so no duplicates appear in the output.
- If these tests fail, the algorithm skips adding iii. Otherwise, it updates CCC to (C∩N(i))∪{i}(C \cap N(i)) \cup \{i\}(C∩N(i))∪{i} and recurses further.

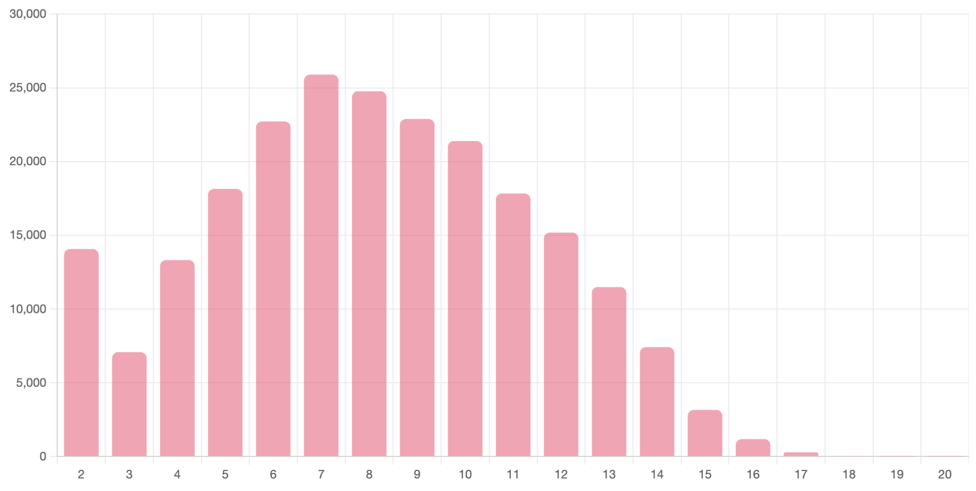
4. **Output:** Whenever the recursion moves beyond the last vertex or confirms no more vertices can be added, the algorithm checks if CCC is maximal and then **prints** or **records** CCC.
5. **Backtrack:** The algorithm reverts CCC to its original form and continues exploring other branches.

Why It Enumerates All Maximal Cliques Exactly Once

- **Complete Coverage:** The recursion ensures that every vertex is considered in both “excluded” and “included” modes, guaranteeing that no clique is overlooked.
- **No Duplication:** The lexicographic ordering test, combined with removing or adding vertices in a consistent manner, prevents the same maximal clique from being generated in multiple ways.
- **Maximality Guarantee:** The algorithm’s final check ensures that no additional vertex can extend CCC, so only truly maximal cliques are output.

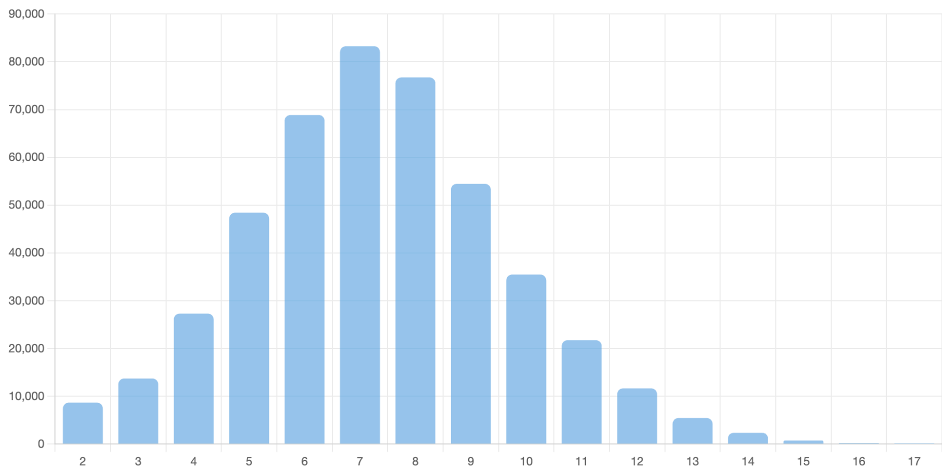
Email-Enron Output

1. Largest Clique Size: 20
2. Total Max Cliques: 226859
3. Time Taken: 1564.35 seconds
4. Distribution of clique sizes:



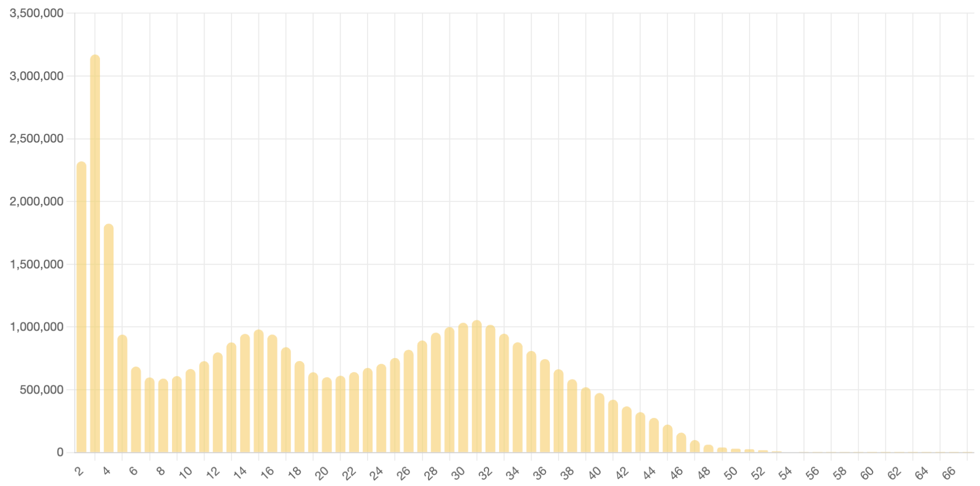
Wiki-Vote Output

1. Largest Clique Size: 17
2. Total Max Cliques: 459002
3. Time Taken: 978.25 seconds
4. Distribution of clique sizes:



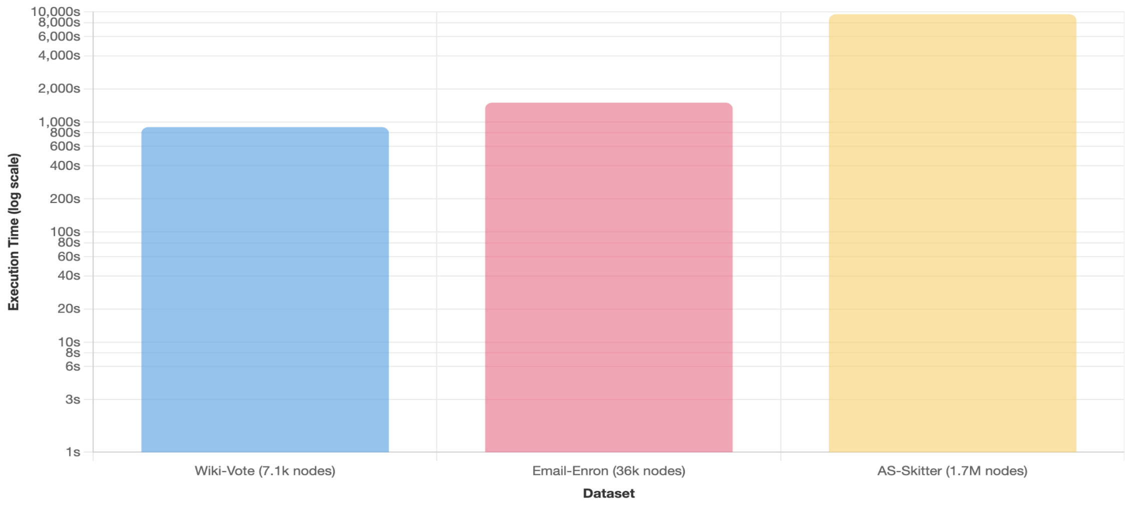
AS-Skitter Output

1. Largest Clique Size: 67
2. Total Max Cliques: 37322355
3. Time Taken: 9532.35 seconds
4. Distribution of clique sizes:



Execution Time Comparison

This chart compares the total execution time of the ARBORICITY AND SUBGRAPH LISTING ALGORITHMS by NORISHIGE CHIBA AND TAKAO NISHIZEKV



ELS :

The algorithm is designed to list all maximal cliques in an undirected graph G=(V,E)G = (V, E)G=(V,E). A **maximal clique** is a set of vertices where every pair is connected, and no additional vertex from VVV can be added without breaking this complete connectivity. The algorithm uses a recursive, backtracking strategy with three mutually disjoint sets:

- **RRR:** The current clique being constructed.
- **PPP:** The set of vertices that can still be added to RRR (these vertices are all adjacent to every vertex in RRR).
- **XXX:** The set of vertices that have already been processed in the current branch (to prevent duplicates).

Initially, the algorithm starts with R=∅R = \varnothingR=∅, P=VP = VP=V (the entire vertex set), and X=∅X = \varnothingX=∅.

Key Steps of the Algorithm

1. **Check for Completion:**
   At the start of a recursive call, the algorithm checks if both PPP and XXX are empty:
   - If **P∪X=∅P \cup X = \varnothingP∪X=**∅, there are no further vertices that can extend RRR. This means RRR is a maximal clique, so the algorithm outputs (or reports) RRR.
2. **Pivot Selection:**
   To reduce the number of recursive calls, a **pivot vertex** uuu is chosen from the set P∪XP \cup XP∪X.
   - The pivot is typically chosen so that ∣P∩Γ(u)∣|P \cap \Gamma(u)|∣P∩Γ(u)∣ is maximized (where Γ(u)\Gamma(u)Γ(u) denotes the neighbors of uuu).

- **Intuition:** Any maximal clique that does **not** include uuu must include at least one vertex that is not adjacent to uuu. Thus, it is sufficient to consider only those vertices in PPP that are not neighbors of uuu (i.e., in P∖Γ(u)P \setminus \Gamma(u)P∖Γ(u)).

3. **Branching on Candidates:**
   The algorithm then iterates over each vertex vvv in the set P∖Γ(u)P \setminus \Gamma(u)P∖Γ(u):

   - **Extend the Clique:** For each such vertex vvv, the algorithm forms a new candidate clique by setting
     R′=R∪{v}.R' = R \cup \{v\}.R′=R∪{v}.
   - **Update the Sets:** The candidates and the excluded set are then updated to only include those vertices that are neighbors of vvv:
     P′=P∩Γ(v)andX′=X∩Γ(v).P' = P \cap \Gamma(v) \quad \text{and} \quad X' = X \cap \Gamma(v).P′=P∩Γ(v)andX′=X∩Γ(v).
     This update ensures that any vertex in P′∪X′P' \cup X'P′∪X′ is adjacent to every vertex in R′R'R′, so the clique property is maintained.

   - **Recursive Call:** The algorithm recursively explores this new branch by calling itself with R′R'R′, P′P'P′, and X′X'X′.

4. **Backtracking:**
   After processing the branch for vertex vvv:

   - vvv is removed from PPP and added to XXX.

   - **Purpose:** This step ensures that vvv is not reconsidered in the same branch of the recursion, thereby avoiding duplicate output of the same clique.

Intuitive Rationale

- **Why Use a Pivot?**
  The pivot uuu is chosen to narrow the search space. If a maximal clique does not include uuu, it must contain some vertex that is not adjacent to uuu. By only exploring vertices in P∖Γ(u)P \setminus \Gamma(u)P∖Γ(u), the algorithm skips

many unnecessary recursive calls, making the search more efficient.

- **Disjoint Sets Ensure Uniqueness:**
  By maintaining RRR, PPP, and XXX as disjoint sets and carefully moving vertices from PPP to XXX once they've been processed, the algorithm guarantees that each maximal clique is reported exactly once.

- **Recursive Exploration and Backtracking:**
  The recursion allows the algorithm to explore all possible extensions of RRR until no more vertices can be added (thus reaching a maximal clique), and backtracking ensures that all branches of the search tree are eventually explored.

Summary

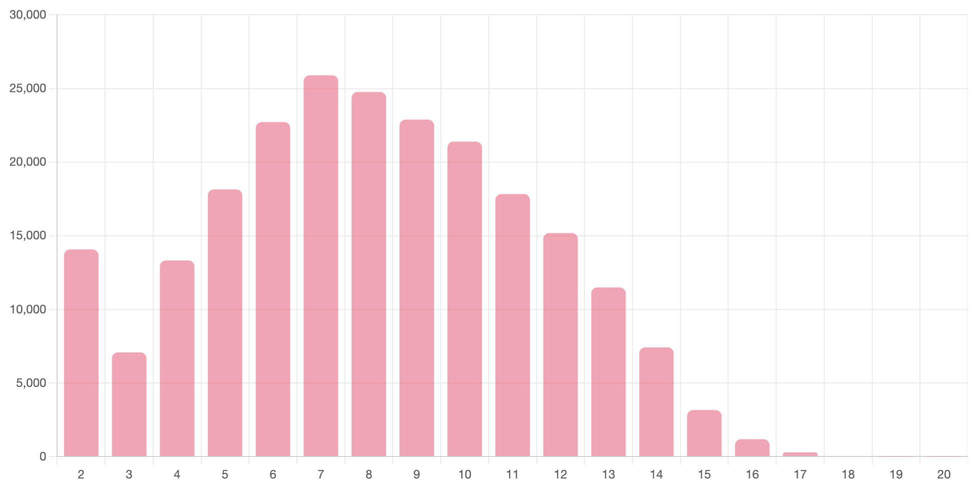
The algorithm begins with an empty clique and the entire vertex set available for expansion. It recursively adds vertices to build a clique RRR while maintaining two auxiliary sets—PPP (potential vertices) and XXX (already processed vertices)—to ensure that:

- The clique property is maintained,
- Each maximal clique is found, and
- No maximal clique is reported more than once.

The use of a pivot vertex helps to minimize the number of recursive calls by focusing only on those candidates that could lead to new maximal cliques. This careful orchestration of candidate selection, recursive expansion, and backtracking is what makes the algorithm both correct and efficient in enumerating all maximal cliques in a graph.

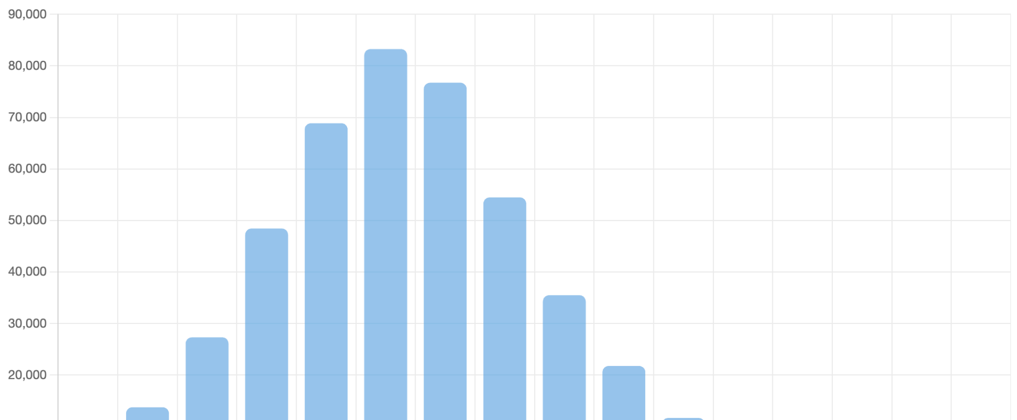
Email-Enron Output

1. Largest Clique Size: 20
2. Total Max Cliques: 226859
3. Time Taken: 9.56 seconds
4. Distribution of clique sizes:



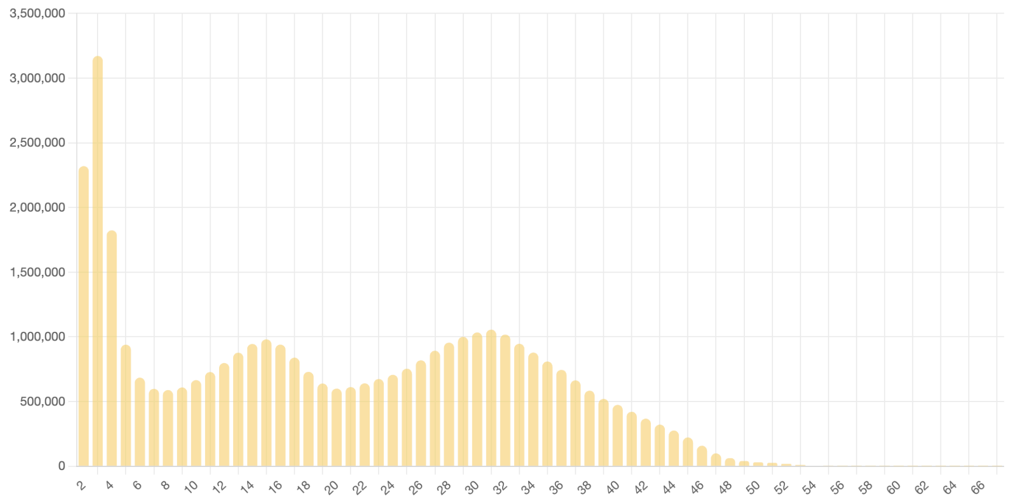
Wiki-Vote Output

1. Largest Clique Size: 17
2. Total Max Cliques: 459002
3. Time Taken: 7.38 seconds
4. Distribution of clique sizes:



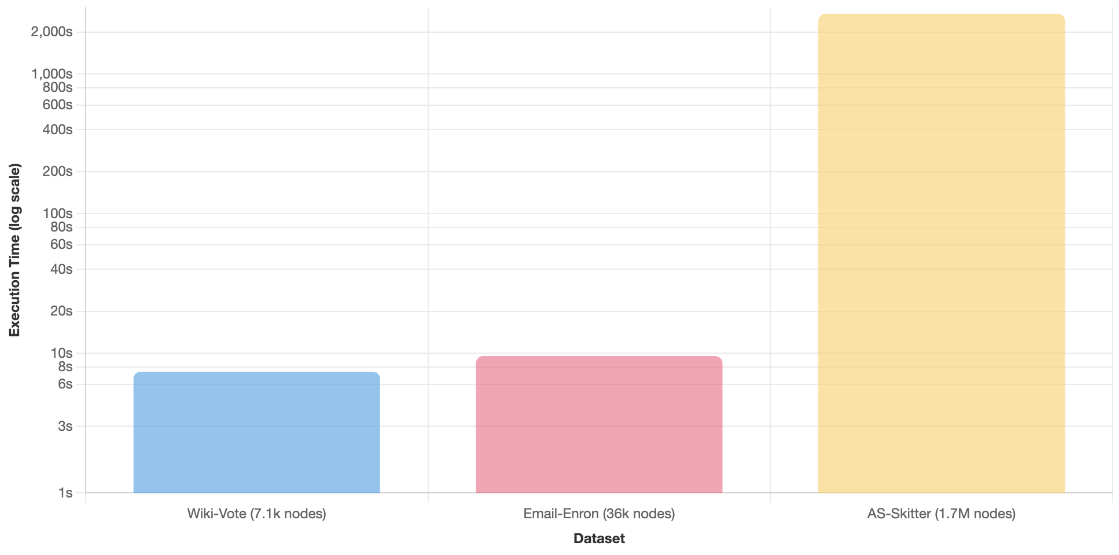
AS-Skitter Output

1. Largest Clique Size: 67
2. Total Max Cliques: 2698.67 seconds
3. Time Taken: 679599 ms
4. Distribution of clique sizes:



Execution Time Comparison

This chart compares the total execution time of the Bron–Kerbosch algorithm (with degeneracy ordering and pivoting) across different datasets.


Execution Time (log scale)
2,000s
1,000s
800s
600s
400s
200s
100s
80s
60s
40s
20s
10s
8s
6s
3s
1s
Wiki-Vote (7.1k nodes)
Email-Enron (36k nodes)
AS-Skitter (1.7M nodes)
Dataset