DAA Assignment

Chapter 1: The worst-case time complexity for generating all maximal cliques and computational experiments [1]

1.1 INTRODUCTION

This paper presents a depth-first search algorithm for generating all the maximal cliques of an undirected graph, in which pruning methods are employed as in the Bron–Kerbosch algorithm[2]. The algorithm's worst-case running time complexity is O(3^(n/3)), for a graph with n vertices. This is the best one could hope for as a function of n, since there exist up to 3^(n/3) maximal cliques in a graph with n vertices as shown by Moon and Moser[3].

1.2 KEY PARTS OF THE ALGORITHM

- **Q**: current clique being built. Initiated as an empty set.
- **SUBG**: set of vertices that can be added to Q. Initiated as a set containing all vertices V of G(V,E).
- **CAND**: vertices from SUBG that haven't been processed (candidates for expansion of cliques). Initiated as a set containing all vertices V of G(V,E).
- **FINI**: Vertices already processed at a given recursion level (finished nodes). Initiated as an empty set.

The core of the algorithm is a recursive procedure named EXPAND(Q, SUBG, CAND, FINI)

<u>If SUBG is empty:</u>
The current clique being built is optimal. Increase num_max_cliques, build clique_size_distr by increasing the count of the corresponding clique size.

<u>Else:</u>
First we choose a pivot vertex u from SUBG that maximizes $|CAND \cap \Gamma(u)|$, where $\Gamma(u)$ denotes neighbors of vertex u and |S| denotes the number of elements in set S. The intuition behind choosing a pivot is to minimize the number of recursive calls by reducing the size of searching space(candidate set - CAND-Γ(u))

Build the candidate set(CAND-Γ(u)). Vertices in the candidate set are considered for expansion of Q.

For each vertex q in candidates set, create SUBG_q = SUBG∩Γ(q), CAND_q = CAND∩Γ(q), FINI_q as empty set. Add q to Q recursively call EXPAND(Q, SUBG_q, CAND_q, FINI_q). After the recursion call remove q from Q and move it from CAND to FINI.

By placing Vertices already processed at a given recursion level in FINI and vertices that need to be considered for further expansion in CAND we prevent redundant generation of cliques.

1.3 TIME COMPLEXITY BREAKDOWN

1. Choosing a pivot from SUBG - **O(n^2)**
2. Generating Candidate set - **O(n)**
3. Process each vertex in candidate set**(|CAND-Γ(u)|)**
   For each vertex a recursive call is made on a subproblem with fewer vertex(worst case scenario subproblem is of size n-1) **O(n-1)**

$$T(n)=k \cdot T(n-1)+O(n^{\wedge}2)+O(n)$$

Where k = |CAND-Γ(u)| represents the branching factor
The Moon-Moser theorem[3] states that the maximum number of maximal cliques in an undirected graph with n vertices is 3^(n/3). when every vertex belongs to exactly three maximal cliques. Therefore in the worst case the branching factor is 3.The new recurrence relation becomes.

$$T(n)=3 \cdot T(n-1)+O(n^{\wedge}2)$$

On solving this, we get T(n)=O(3^(n/3))

1.4 EXPERIMENTAL RESULTS

Dataset	Total Maximal Cliques	Execution Time (seconds)	Largest Clique Size
Wiki-Vote	4,59,002	27	17
Email-Enron	2,26,859	142	20
As-Skitter	3,73,22,355	56,891	67

1.5 REFERENCES

[1] Tomita, Etsuji & Tanaka, Akira & Takahashi, Haruhisa. (2006). The Worst-Case Time Complexity for Generating All Maximal Cliques and Computational Experiments. Theoretical Computer Science. 363. 28-42. 10.1016/j.tcs.2006.06.015.

[2] C. Bron, J. Kerbosch, Algorithm 457, finding all cliques of an undirected graph, Comm. ACM 16 (1973) 575–577.

[3] J.W. Moon, L. Moser, On cliques in graphs, Israel J. Math. 3 (1965) 23–28.

Chapter 2: Listing All Maximal Cliques in Sparse Graphs in Near-optimal Time

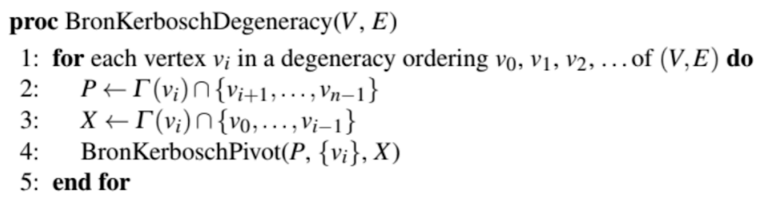
2.1 INTRODUCTION

The Bron-Kerbosch Algorithm is an algorithm used for finding all the maximal cliques in an undirected graph. The paper delves into a variant of the algorithm that involves using a degeneracy ordering in order to run more efficiently with fewer recursion calls.

2.2 KEY PARTS OF THE ALGORITHM

The algorithm detailed in Fig. 4 just orders the vertices in a degeneracy ordering, i.e. the degree of the vertices, and then applies the Bron-Kerbosch Algorithm with pivoting with each vertex from the ordering as the pivot.

1. Any preprocessing in order to get an edge list with indexed nodes from 0 to N - 1 along with N, the number of nodes.
2. Generating an adjacency list utilising a vector of unordered sets. This will help in looking up adjacent nodes in O(1) time as compared to using vectors.
3. Compute the degeneracy ordering of the graph by repeatedly removing the vertex of minimum degree then updating its neighbours' degrees.
4. We have three sets of vertices:
    a. R, which is the current clique
    b. P, which is a list of vertices that can be added to R to increase the clique size
    c. X, which is a list of vertices that can't be used as they've already been processed.
5. Using the degeneracy ordering, process the vertices and initialise each set accordingly for a certain vertex v:
    a. R, will have only v
    b. P will have all the vertices after v
    c. X will have all the vertices before v
6. For each call of the function, find a pivot to maximise the neighbours of the pivot, and then process only the vertices in P not including the neighbours of the pivot.
7. Base Case: If both P and X are empty, then no more vertices can be added to the clique and every vertex has been processed and so this is a maximal clique and should be recorded.
8. For each vertex in P \ neighbours(pivot), add the vertex to R, and update P to the neighbours of vertex in P and X to neighbours of vertex in X.
9. Keep recursing until the entire degeneracy list is covered.



```
proc BronKerboschDegeneracy($V$, $E$)
1: for each vertex $v_i$ in a degeneracy ordering $v_0, v_1, v_2, \dots$ of $(V,E)$ do
2:     $P \leftarrow \Gamma(v_i) \cap \{v_{i+1}, \dots, v_{n-1}\}$
3:     $X \leftarrow \Gamma(v_i) \cap \{v_0, \dots, v_{i-1}\}$
4:     BronKerboschPivot($P$, $\{v_i\}$, $X$)
5: end for
```

Fig. 4. Our algorithm.

2.3 TIME COMPLEXITY BREAKDOWN

Computing Degeneracy Ordering

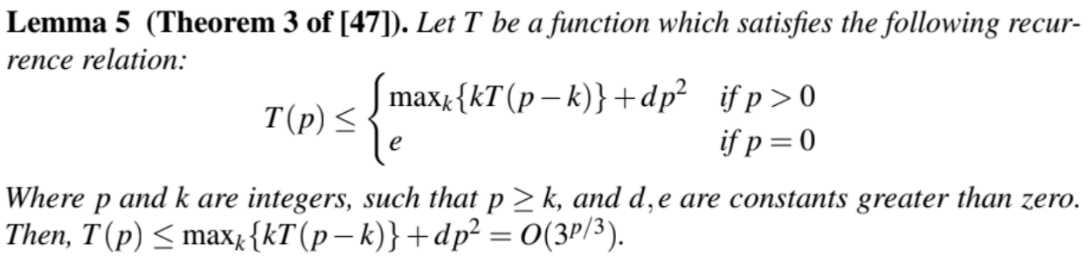
This step involves iteratively removing the vertices of minimum degree one by one, and then modifying the neighbours with updated degrees.

This step is O(n + m) where n is the number of vertices and m is the number of edges.

This is because every vertex will be covered while removing and every edge will be covered while updating the degrees.

Recursive Calls

The most time consuming part of the process is to recursively search for maximal cliques.



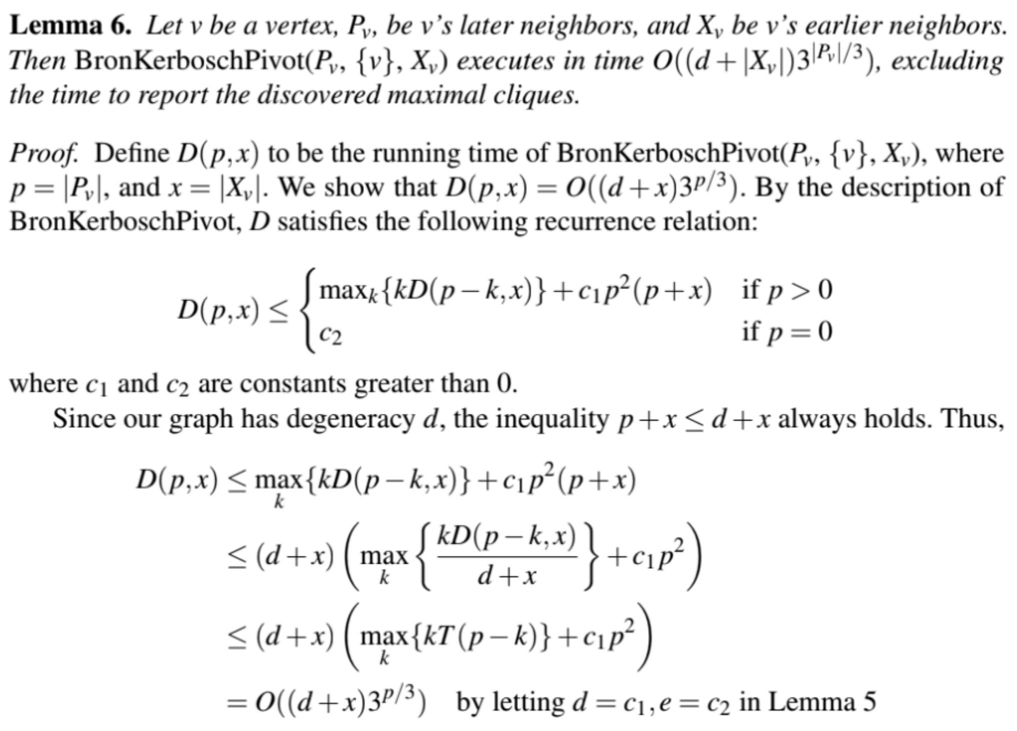
Lemma 5 (Theorem 3 of [47]). *Let T be a function which satisfies the following recurrence relation:*

$$T(p) \leq \begin{cases} \max_k\{kT(p-k)\} + dp^2 & \text{if } p > 0 \\ e & \text{if } p = 0 \end{cases}$$

Where p and k are integers, such that $p \geq k$, and d, e are constants greater than zero. Then, $T(p) \leq \max_k\{kT(p-k)\} + dp^2 = O(3^{p/3})$.

The paper details properly how the time complexity of this algorithm has been calculated. First, Lemma 5 is shown to give a big O time complexity of 3^p/3 for the particular recurrence relation shown above.

This has been taken as a result from Tomita et. al., 2006. The paper has been referenced below for any further inspection.



Lemma 6. *Let v be a vertex, P_v, be v's later neighbors, and X_v be v's earlier neighbors. Then* BronKerboschPivot(P_v, $\{v\}$, X_v) *executes in time $O((d+|X_v|)3^{|P_v|/3})$, excluding the time to report the discovered maximal cliques.*

Proof. Define $D(p,x)$ to be the running time of BronKerboschPivot(P_v, $\{v\}$, X_v), where $p = |P_v|$, and $x = |X_v|$. We show that $D(p,x) = O((d+x)3^{p/3})$. By the description of BronKerboschPivot, D satisfies the following recurrence relation:

$$D(p,x) \leq \begin{cases} \max_k\{kD(p-k,x)\} + c_1p^2(p+x) & \text{if } p > 0 \\ c_2 & \text{if } p = 0 \end{cases}$$

where c_1 and c_2 are constants greater than 0.

Since our graph has degeneracy d, the inequality $p+x \leq d+x$ always holds. Thus,

$$\begin{aligned} D(p,x) &\leq \max_k\{kD(p-k,x)\} + c_1p^2(p+x) \\ &\leq (d+x)\left(\max_k\left\{\frac{kD(p-k,x)}{d+x}\right\} + c_1p^2\right) \\ &\leq (d+x)\left(\max_k\{kT(p-k)\} + c_1p^2\right) \\ &= O((d+x)3^{p/3}) \quad \text{by letting } d = c_1, e = c_2 \text{ in Lemma 5} \end{aligned}$$

The next part of the proof details exactly how Bron-Kerbosch algorithm with pivoting and degeneracy ordering leads to the recurrence relation detailed above in Lemma 5.

Hence, holding that true, we finally get the time complexity of **O((d+x) * 3^p/3) for each node.** Where p is the size of neighbours after v, and x is the size of earlier neighbours of v. And d is the degeneracy of the graph.

Combining Everything

As p can at most be d, and x is ultimately limited by the graph structure, and cancels to zero on larger datasets with more nodes.

Thus applying this for every node in the function gives us a final time complexity of **O(d3^d/3)**

2.4 EXPERIMENTAL RESULTS

Dataset	Maximal Cliques	Execution Time (sec)	Largest Clique Size
Wiki-Vote	4,59,002	212 sec	17
Enron	2,26,859	582 sec	20
As-Skitter	3,73,22,355	~75,750 sec	67

*time for As-Skitter has been approximated

2.5 POSSIBLE OPTIMIZATIONS

The main problem with this implementation is the inefficient way of taking unions and intersections of sets. Due to the nature of vectors and iteration, we'll have to iterate over both the sets every time.

A possible optimisation for smaller graphs at least, is to use bitsets which use bitwise operations in order to calculate unions and intersections, which is as simple as using AND and OR operations. This can help if we have small graphs, but as seen for Skitter, this strategy wouldn't help too much, unless we can handle bitsets of huge size.

Another would be to use iterators as Prahlad mentioned in his section. Copying has a lot of overhead every time and contributes a great deal to the overall runtime and efficiency of the algorithm. Passing by reference to somehow evade this would be a nice boost to the efficiency of this algorithm.

2.6 REFERENCES

Eppstein, D., Löffler, M., & Strash, D. (2010). Listing all maximal cliques in sparse graphs in Near-Optimal time. In *Lecture notes in computer science* (pp. 403–414). https://doi.org/10.1007/978-3-642-17517-6_36

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, https://doi:10.1016/j.tcs.2006.06.015.

Chapter 3: Arboricity and Subgraph Listing Algorithms

3.1 INTRODUCTION

The algorithm for listing all maximal cliques in a graph, as described in the paper, was implemented and tested on three real-world datasets: the **Wiki-Vote dataset**, **Email-Enron dataset, Autonomous systems by Skitter datasets**. This report presents an analysis of the algorithm's performance, including the total number of maximal cliques found, execution time, and additional metrics for evaluating computational efficiency.

3.2 KEY PARTS OF THE ALGORITHM

The Chiba-Nishizeki algorithm on Pg 220-222, lists all maximal cliques in a graph using a vertex elimination approach combined with adjacency list traversal. The key steps are:

1. **Vertex Sorting**: We sort the adjacency list such that the nodes with least degree appear first and thus are ordered based on degree.

2. **Clique Expansion**: For each node, the algorithm finds cliques that have the vertex by searching its neighborhood.
3. **Maximality test [1, pp. 218-220]:** For a given clique C, we try to add a node and check if it can be made maximal by adding that node. Doing this test for all nodes ensures that the current clique is maximal and no more edges can be added to it.
4. **Backtracking & Pruning[1, pp. 219-221]**: Unnecessary computations are avoided by eliminating processed vertices, ensuring(using the lexicographical test) each clique is enumerated exactly once.

3.3 TIME COMPLEXITY BREAKDOWN

The algorithm operates with a worst-case time complexity of **O(α(G) * m) per maximal clique**, where:

- **m** is the number of edges in the graph.
- **α(G)** is the arboricity of the graph

Step-wise Complexity Analysis

1. **readGraph (O(m)):** Simply reading all edges from file and updating the adj lists.
2. **Vertex Sorting (O(n))**:
    - In my implementation I have used the bucket sort algorithm as degrees and node numbers are discrete integer values.
3. **Building Neighborhood Subgraphs (O(m))**:
    - Before a node is added, it goes through **maximalityTest()** and **lexicographicTest()** each of which take **(O(m))** time.

4. **Recursive Clique Enumeration (O(α(G) * m))**:
    - The recursion ensures that each clique is found exactly once, using edge-based expansion limited by arboricity α(G). This is implemented in the **UPDATE()** function.
5. **Pruning & Backtracking (O(m))**:
    - By removing already processed vertices, the search space is reduced.

Combining these steps, the overall complexity **per** maximal clique remains **O(α(G) * m)**.

3.4 EXPERIMENTAL RESULTS

Dataset	Total Maximal Cliques	Execution Time (seconds)	Largest Clique Size
Wiki-Vote	459,002	484	17
Email-Enron	226,859	697	20
As-Skitter	37,322,355	90,437	67

3.5 POSSIBLE OPTIMIZATIONS

1) **Early vertex elimination** [1, p. 213]
We found that there are many 2 sized cliques in the dataset. Since, a 2 sized clique is just an edge which has 2 nodes both of which have a degree 1, we can detect all 2-Cliques in the bucket sort itself. These nodes and their edges can be removed safely and the clique algorithm will still work. We argue that if β $is\ the\ fraction\ of\ 2 - Cliques$ in the entire set of maximal cliques, then
New time complexity is

$$O(\beta \cdot \alpha(G) \cdot m) \text{ per maximal clique.}$$

This would be helpful in the skitter dataset as it has $\beta = 2319807/37322355 = 0.062$.
We admit this is a small improvement, but still thought it's worth mentioning.

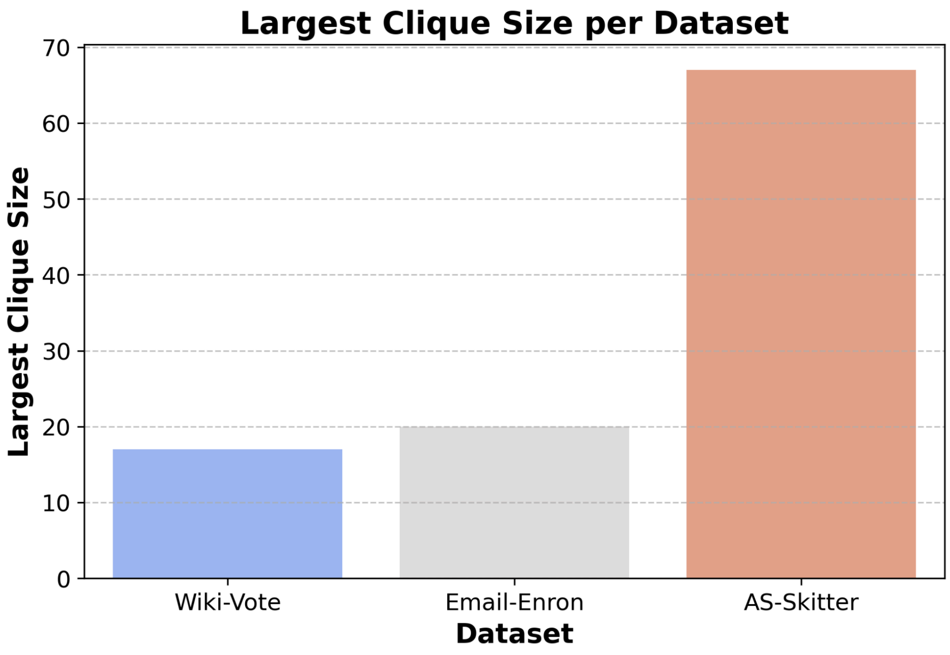
2) We were not able to completely prevent the 'copy' operation during a recursive call. So another optimization could be to use iterators to save time.

3.6 REFERENCES

[1] Chiba, N., & Nishizeki, T. (1985). *Arboricity and Subgraph Listing Algorithms*. SIAM Journal on Computing, 14(1), 210-223.

Chapter 4: Conclusions

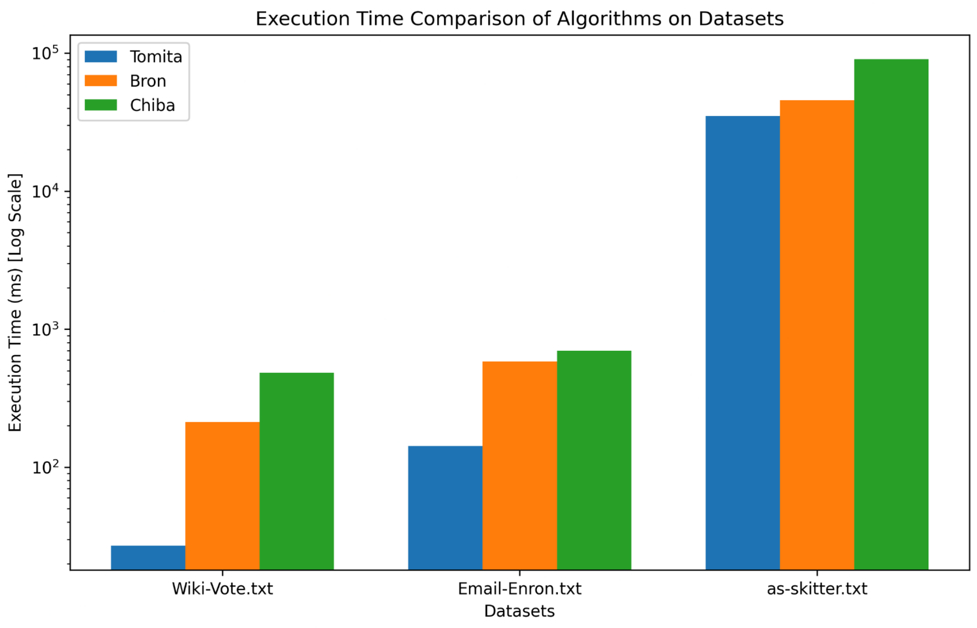
4.1 LARGEST SIZE OF THE CLIQUE



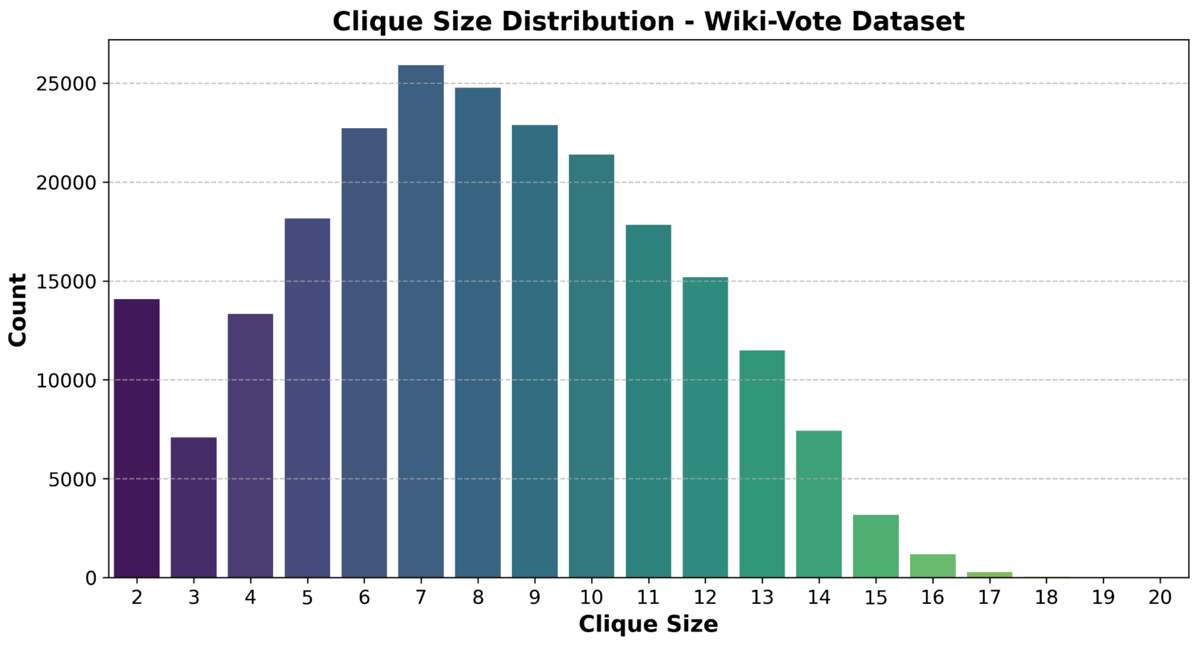
4.2 TOTAL NUMBER OF MAXIMAL CLIQUES

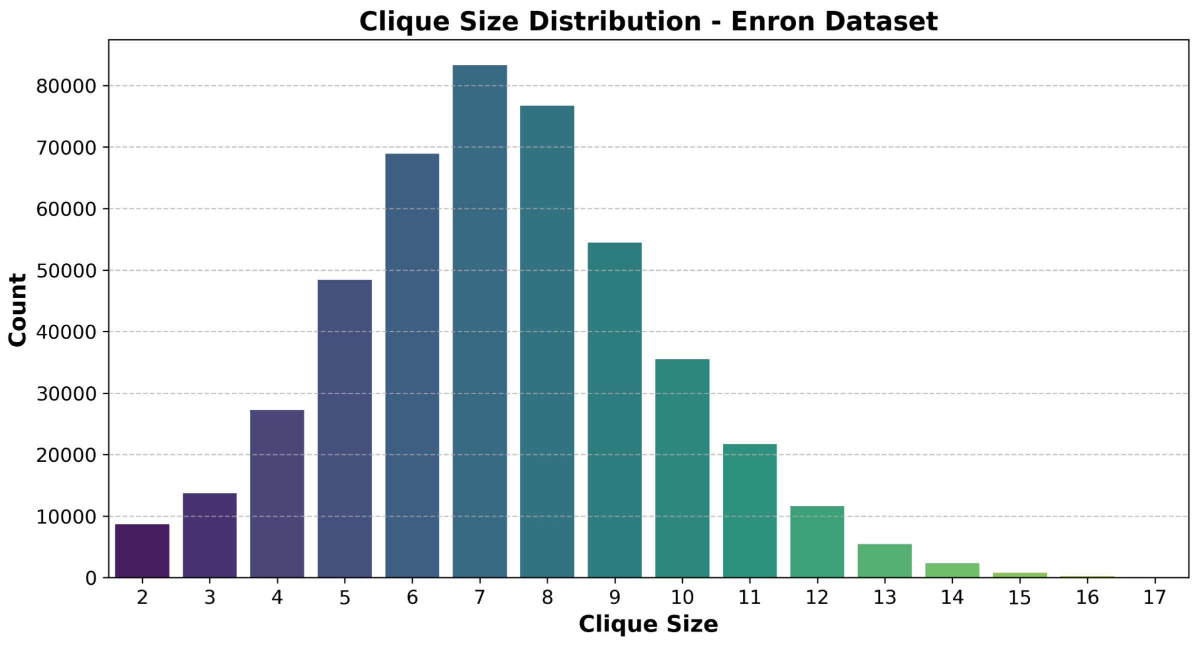
Dataset	Total Maximal Cliques
Wiki-Vote	4,59,002
Email-Enron	2,26,859
As-Skitter	3,73,22,355

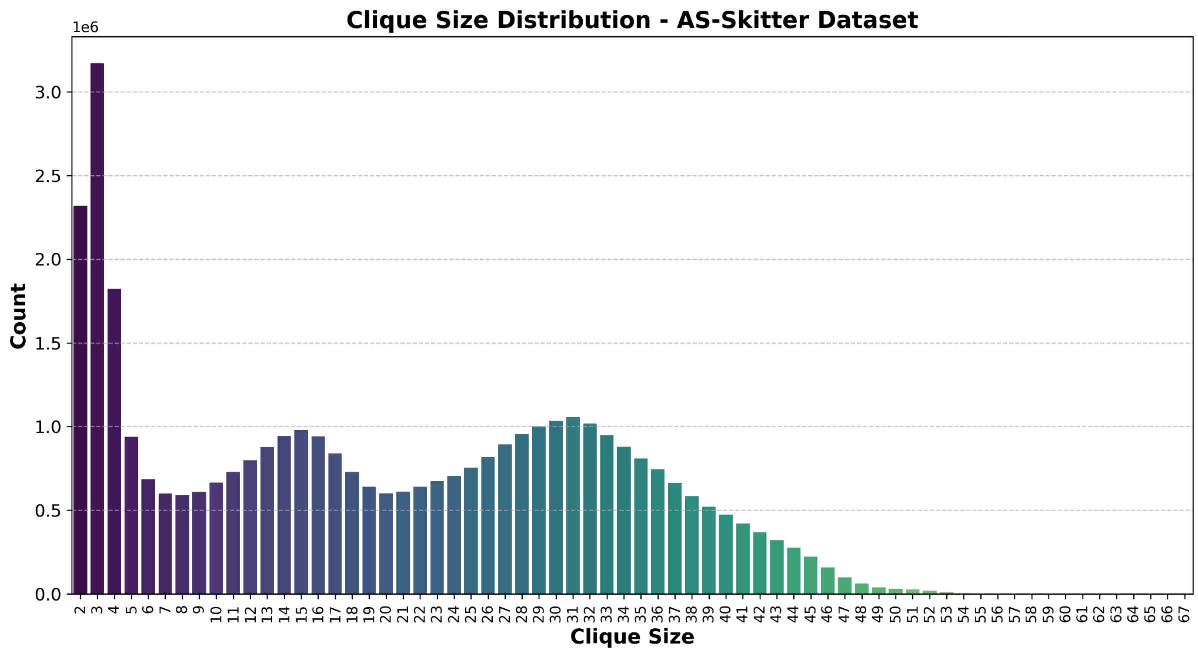
4.3 EXECUTION TIME



4.4 DISTRIBUTION OF DIFFERENT SIZE CLIQUES





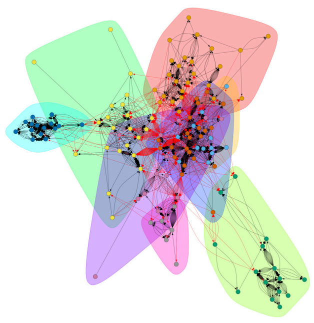


Chapter 5: Analysis of the Dataset

5.1 Enron email network

Enron Corporation is a company that collapsed in 2001 due to an accounting scandal known to many as the Enron scandal. As a consequence of this scandal, all the emails generated by the employees were declared public by the Federal Energy Regulatory Commission.

Thus, since this data is publicly available, it has been a famous source for data scientists and computer scientists to analyse, particularly regarding graph algorithms and such.



Seen here is a visualisation of the email network, representing the eight communities.

Our Thoughts

So, if the Enron network contains 226859 maximal cliques, that implies that there are interconnected subgroups of employees in Enron who regularly work amongst each other and frequently contact each other on a regular basis.

Further investigations can reveal what exactly these subgroups mean and how they've been made. But it is very interesting to note this as a result of finding maximal cliques of the dataset.

5.2 Wikipedia vote network

On Wikipedia, there are elections held in order to allow users to become admins. So this data has been acquired where each node is a user and an edge represents one user voting for another.

Our Thoughts

So, looking at 459 thousand maximal cliques, this purely implies that there are many subgroups of users who voted for each other amongst themselves. That's the key takeaway that finding maximal cliques can bring to this particular dataset.

Chapter 6: Potential Application in Plagiarism Analysis

6.1 Background

In 3-1, when I (Ashish) was doing the OS Assignment for CS F364 Operating Systems course, I was flagged for plagiarism although I hadn't done anything of the sort.

After inspecting the moss.txt file for finding matches, I found people I didn't know at all with whom I had major matches in the code with. We later found out that effectively my code was stolen.

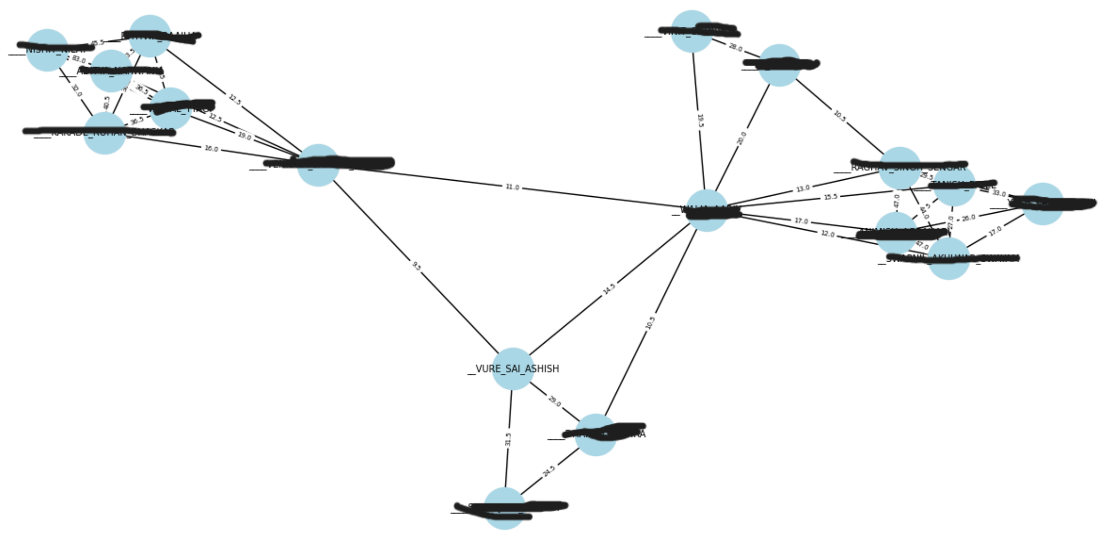
6.2 Application of Maximal Cliques

After analysing the moss file, which is roughly of the format:

```
Assignment2FinalSubmissions/____[redacted]_612_20221796/2024-11-22-23-57-29/
(71%)
Assignment2FinalSubmissions/____[redacted]_674_20220227/2024-11-22-17
-35-12/ (69%)  468
Assignment2FinalSubmissions/____[redacted]_593_20220023/2024-1
1-23-23-02-59/ (99%)
Assignment2FinalSubmissions/____[redacted]_88_202200
32/2024-11-23-23-57-09/ (99%)        450
```

Assignment2FinalSubmissions/____[redacted]_674_20220227/2024-11-22-17-35-12/ (67%)
Assignment2FinalSubmissions/__[redacted]_842_20220091/2024-11-21-18-45-07/ (73%)432
Assignment2FinalSubmissions/____[redacted]_612_20221796/2024-11-22-23-57-29/ (67%)
Assignment2FinalSubmissions/__[redacted]_842_20220091/2024-11-21-18-45-07/ (71%)427

This leads to a graph as follows:



So as seen in the graph, there are multiple maximal cliques visible. Basically what this means is that these people have gotten the code from somewhere and circulated it themselves which is why they all have links amongst each other.

So this is a valid application of cliques in real world problems.

Also P. S the plagiarism issue was resolved and I got my marks after showing this whole situation to the I. C so no issues there. However the people who stole the code did get away with it as we never got to the bottom of this situation.

Chapter 7: Contributors

1. Prahlad Reddy Ponnatata (2022A7PS0097H)
2. Sai Ashish Vure (2022A7PS0016H)
3. Nikhilesh Balla (2022A7PS0040H)