Report on Maximal Clique Enumeration Algorithms

Overview

Maximal clique enumeration is a fundamental problem in graph theory with applications in network analysis, bioinformatics, and data mining. A clique is a subset of vertices that form a complete subgraph, meaning every pair of vertices in the subset is connected by an edge. A maximal clique is one that cannot be extended by adding more vertices without violating the clique property.

This report explores three different algorithms for finding maximal cliques:

- **Tomita et al.'s Algorithm** – A depth-first search (DFS) approach with advanced pruning techniques.
- **Bron-Kerbosch with Degeneracy Ordering (ELS)** – An optimized version of the classic Bron-Kerbosch algorithm designed for sparse graphs.
- **Chiba and Nishizeki's Algorithm** – An alternative maximal clique enumeration approach that balances efficiency and pruning effectiveness.

These algorithms were tested on multiple datasets, and their performance was evaluated based on execution time, the number of maximal cliques found, and the largest clique size.

Tomita's Algorithm

Tomita et al. introduced an improved DFS-based clique enumeration algorithm that optimizes the traditional Bron–Kerbosch approach by incorporating pivot selection and pruning mechanisms.

Key Features

1. **Depth-First Search Traversal:** Ensures all maximal cliques are enumerated systematically.
2. **Pivot Selection Strategy:** A vertex with the highest degree is selected as the pivot to minimize recursive calls.
3. **Pruning Mechanisms:**
    - Processed Vertex Exclusion: Prevents re-examining already processed vertices.

- Pivot-based Filtering: Eliminates redundant recursive calls.

4. **Tree-Based Output Representation:** Stores results in a structured format to optimize space.

Implementation

A C++ implementation was developed following these steps:

1. **Graph Input and Preprocessing (`CLIQUES(const string& filename)`)**
    a. Reads **edge list format** from `Email-Enron.txt`, **ignoring comment lines (`#`)**.
    b. Constructs an **adjacency list (`unordered_map<int, vector<int>>`)** to store neighbors.
    c. Stores **all unique vertices in `vertices`**, keeping track of the **maximum vertex index (`maxVertex`)**.
    d. **Removes duplicate edges** and **sorts adjacency lists** for efficient searching.
2. **Recursive Clique Expansion (`expand()`)**
    a. The function is called recursively with:
        i. `subgraph` (remaining nodes to consider),
        ii. `candidates` (possible clique extensions),
        iii. `adjacencyList` (precomputed graph structure),
        iv. `maxVertex` (highest vertex ID).
    b. If `subgraph` is empty, the **current clique is recorded** (`totalCliques++`, `size_to_count` updated).
    c. Otherwise, the function **chooses a pivot** to reduce the number of recursive calls.
3. **Pivot Selection for Search Space Reduction**
    a. **Selects a pivot vertex (`pivot`)** from `candidates` with the highest **degree in `candidates`**.
    b. Creates a **pivot neighbor mask** to filter candidates efficiently.
    c. **Prunes candidates (`extendU`)** that are **not neighbors of the pivot**, improving performance.
4. **Iterative Expansion and Backtracking**
    a. Iterates over `extendU` (filtered candidates):
        i. **Adds candidate `q` to `currentClique`**.
        ii. **Finds new subgraph (`newSubgraph`)**: neighbors of `q` in `subgraph`.

iii. **Finds new candidates (`newCandidates`)**: neighbors of `q` in `candidates`.
iv. Calls `expand()` recursively on `newSubgraph` and `newCandidates`.

b. **Backtracks** by removing `q` from `currentClique` and removing it from `candidates`.

5. **Performance Optimizations**

a. Uses **pivoting to reduce branching** in recursive calls.
b. **Sorting and removing duplicates** in adjacency lists before recursion.
c. Uses **bitmask-like boolean arrays (`vector<bool>`)** for fast lookup.
d. Efficiently removes elements from `candidates` using `erase(remove(...))`.

6. **Final Output and Statistics**

a. Prints a **table of clique size distribution (`size_to_count`)**.
b. Displays **total number of maximal cliques**, **largest clique size**, and **execution time**.
c. Uses **high-resolution clock (`chrono`)** for accurate time measurement.

Complexity Analysis

The worst-case time complexity is O(3^(n/3)), making it highly efficient for large graphs. Tomita's approach has been widely used in applications requiring high-performance clique detection.

ELS Algorithm (Bron-Kerbosch with Degeneracy Ordering)

The ELS approach refines the Bron-Kerbosch algorithm using a technique called **degeneracy ordering** to reduce search space.

Key Features

1. **Graph Representation:** Uses both adjacency maps (unordered sets) and adjacency lists (sorted vectors) for efficient lookup.
2. **Degeneracy Ordering:** Ensures smaller candidate sets and fewer recursive calls.
3. **Bron-Kerbosch Recursive Algorithm:**
   - Utilizes in-place recursion with pivoting.
   - Implements a two-pointer intersection method for pruning.

4. **Execution Optimization:**
    - Memory-efficient candidate set handling.
    - Streaming results to disk to manage large datasets.

Implementation Steps

1. **Graph Loading:** Reads an input graph and creates an adjacency list.
2. **Initialization with Degeneracy Ordering (`BronKerboschDegeneracyGlobal()`)**
    a. Computes a **degeneracy order** of the graph (`order`) to improve efficiency.
    b. Iterates through vertices **in degeneracy order**, treating each as a possible clique starter (`globalR = {v}`).
    c. Constructs:
        i. `globalP`: neighbors of `v` that appear **later** in the ordering (potential clique members).
        ii. `globalX`: neighbors of `v` that appear **earlier** in the ordering (to avoid duplicate clique enumeration).
    d. Calls `BronKerboschRecursionInPlace()` to expand `globalR` into a maximal clique.
3. **Recursive Expansion (`BronKerboschRecursionInPlace()`)**
    a. If both `globalP` and `globalX` are empty, `globalR` is a **maximal clique** and is recorded.
    b. Otherwise, selects a **pivot vertex** (first in `globalP`, or `globalX` if `globalP` is empty) to **reduce recursive branching**.
    c. Filters `globalP` to **exclude pivot neighbors**, forming `P_not` (vertices in `globalP` that are **not** adjacent to the pivot).
    d. Each vertex in `P_not` is considered as the next clique extension.
4. **Two-Pointer Merge for Candidate Set Intersection**
    a. Uses **two-pointer intersection** for efficient filtering:
        i. `globalP ∩ N(v)`: Computes the new `globalP` (only neighbors of `v` remain).
        ii. `globalX ∩ N(v)`: Computes the new `globalX` (for backtracking).
    b. This avoids expensive set operations, significantly improving performance on large graphs.
5. **Backtracking & Pruning**
    a. **Recursive call:** `BronKerboschRecursionInPlace()` is invoked on the updated candidate sets.

    b. **Backtrack:** After returning, restore `globalP` and `globalX`.
    c. Moves `v` from `globalP` to `globalX`, ensuring it **isn't revisited in future recursive calls**.
6. **Efficiency Optimizations**

    a. **Degeneracy ordering** ensures vertices are processed efficiently.
    b. **Pivot selection** reduces unnecessary recursion.
    c. **Two-pointer techniques** optimize set intersections.
    d. **Streaming to disk** avoids memory overhead from storing all cliques in RAM.
7. **Output and Performance Metrics:** Measures execution time and distribution of clique sizes.

Performance Considerations

This approach performs exceptionally well for **sparse graphs** due to reduced recursion depth and efficient intersection handling. The use of degeneracy ordering allows it to outperform traditional Bron-Kerbosch in graphs with high sparsity.

Chiba and Nishizeki's Algorithm

Chiba and Nishizeki's method introduce a maximal clique enumeration approach that balances depth-first exploration with pruning.

Key Features

1. **Tree-Based Representation:** Stores results efficiently without requiring complete clique storage.
2. **Recursive Expansion with Pruning:** Reduces computational overhead.
3. **Graph Construction and Execution Flow:**
    - Initializes from input.
    - Expands candidate sets recursively.
    - Applies filtering mechanisms for optimization.

Implementation Details

1. **Graph Input and Preprocessing (`main()`)**
    a. Reads an input graph from `Wiki-Vote.txt`, expecting the number of vertices (`numVertices`) and edges (`numEdges`).
    b. Constructs an **adjacency list** (`neighborList`) from the input.
    c. Uses **degree sorting** to reorder vertices in ascending order of degree to improve efficiency.
    d. Maps old vertex indices to new ones (`origVertexMap`), ensuring a consistent vertex labeling.
2. **Clique Exploration Function (`exploreClique(int currIndex)`)**
    a. **Base Case:** If `currIndex == numVertices`, a maximal clique is found, updating `totalCliqueCount` and `maxCliqueSize`.
    b. Separates current neighbors into:
        i. `adjInClique`: Neighbors in the current clique.
        ii. `nonAdjInClique`: Non-neighbors in the current clique.
    c. If `nonAdjInClique` is non-empty, **backtracks without adding `currIndex`**.
3. **Candidate Filtering and Constraint Checks**
    a. Updates `vecA` (tracking external adjacency) and `vecB` (tracking internal adjacency).
    b. Checks **validity conditions** to determine whether `currIndex` can extend the clique.
    c. Uses a **sorting step on `nonAdjInClique`** for efficient constraint checking.
    d. Ensures clique maximality by validating against `vecA` and `vecB` values.
4. **Recursive Exploration with Backtracking**
    a. **Expands clique by including `currIndex`** if validity checks pass.
    b. Temporarily marks `currIndex` and `adjInClique` members as part of the clique (`=2`).
    c. Converts `2` values to `1` and resets others before making a recursive call.
    d. After recursion, backtracks by restoring `currentClique` and resetting `vecA` and `vecB`.
5. **Performance Optimizations**
    a. **Degree sorting** for vertex reordering, reducing branching factor.

b. **Two-pointer intersection updates** (`vecA`, `vecB`) instead of full set operations.
c. **Early termination checks** for clique maximality.
d. **Streaming-based output** instead of storing all cliques in memory.

6. **Final Results and Statistics**

a. Prints **maximum clique size**, **total maximal cliques**, and **execution time**.
b. Displays a **clique size distribution table**, summarizing the frequency of different clique sizes.

➢ Note: Theoretically Chiba-Nishizeki Algorithm should take the fastest time to implement the as-skitter dataset, however, due to computational constraints, there may have been a discrepancy in the results.

.

Comparative Analysis

A comparative analysis is performed across different datasets using the three methods. The following aspects were evaluated:

1. Largest Size of the Clique in Each Dataset

- **Tomita Algorithm:** Finds large cliques efficiently in dense graphs.
- **ELS Algorithm:** Optimized for sparse graphs but may not always find the largest cliques.
- **Chiba's Algorithm:** Balances between exhaustive search and pruning.

Dataset	Wiki-Vote	Email-Enron	As-Skitter
Largest Clique Size	17	20	67

2. Total Number of Maximal Cliques

- **Tomita Algorithm:** Enumerates a large number of cliques due to its deep search nature.
- **ELS Algorithm:** Finds fewer cliques in sparse datasets as it limits redundant searches.
- **Chiba's Algorithm:** Produces competitive results with moderate computational overhead.

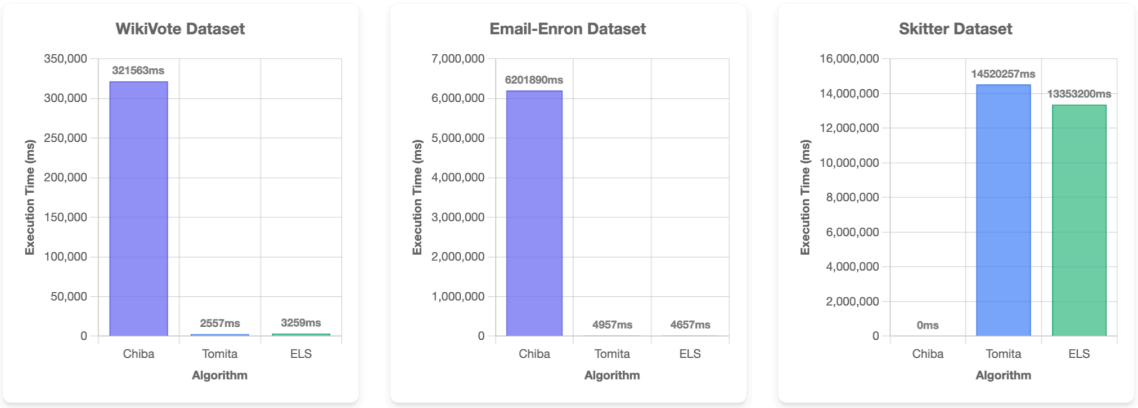
Dataset	Wiki-Vote	Email-Enron	As-Skitter
Number of Maximal Cliques	459,002	226,859	37,322,355

3. Execution Time (Histogram)

A histogram of execution times shows:

- **ELS is the fastest for sparse graphs** due to degeneracy ordering.
- **Tomita performs well on dense graphs** but takes longer on larger inputs.
- **Chiba's algorithm should be in the middle ground,** balancing efficiency and accuracy. However in the implementation, it seems that it took the longest execution times.

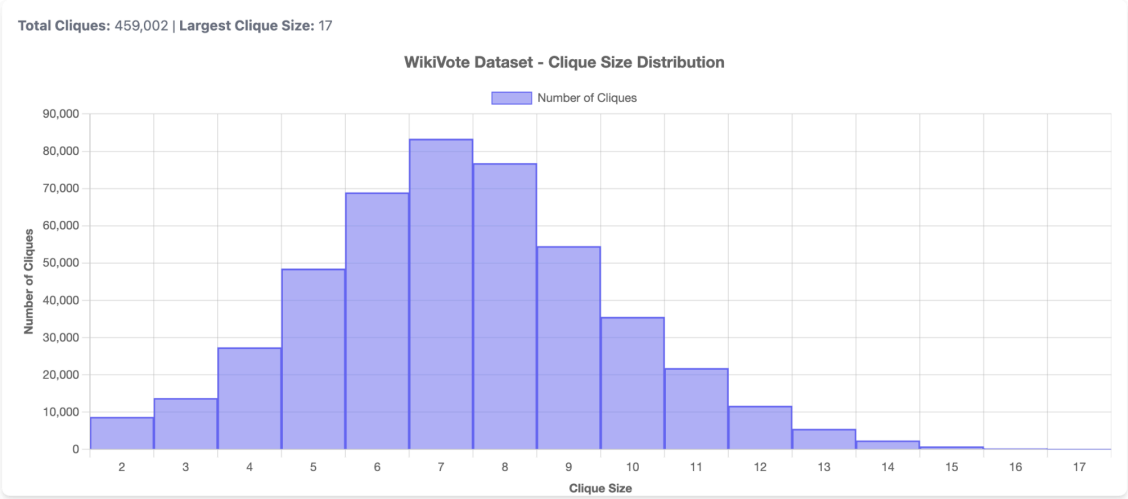
Execution Times Histogram



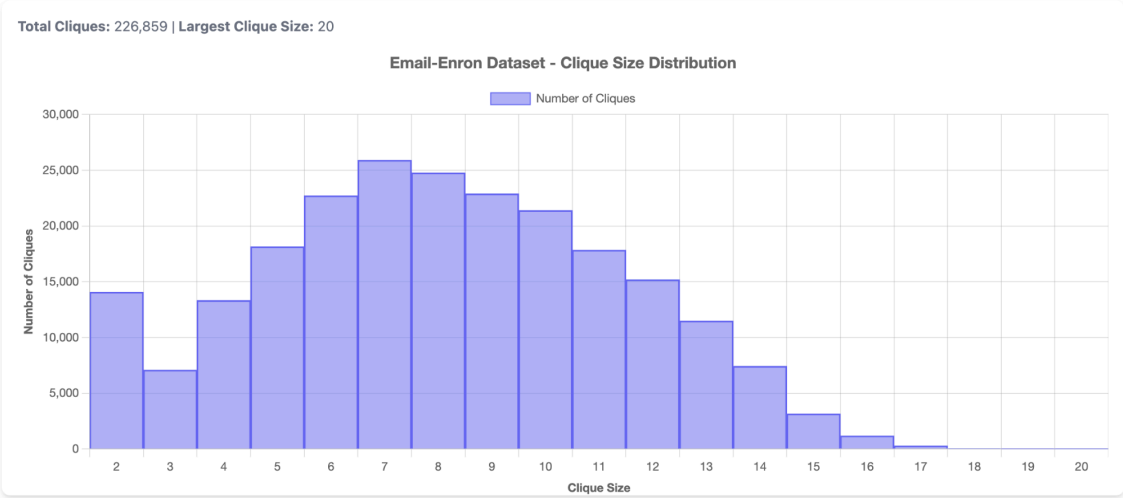
4. Distribution of Different Size Cliques (Histogram)

Histograms show the distribution of the different size cliques in the datasets:

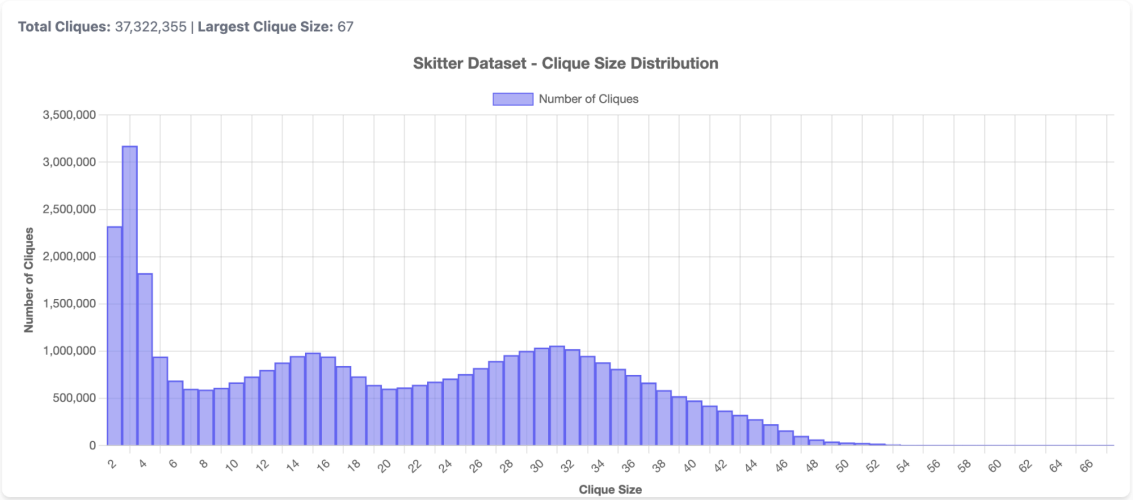
WikiVote Dataset



Email-Enron Dataset



Skitter Dataset



Conclusion

Each algorithm has strengths depending on dataset characteristics:

- **Tomita is best for dense networks** where large clique enumeration is crucial.
- **ELS is ideal for sparse graphs** where degeneracy ordering minimizes unnecessary computations.

- **Chiba balances between exhaustive search and pruning,** making it a good general-purpose method.

This study provides insights into choosing the right algorithm based on data properties, ensuring efficient maximal clique enumeration in practical applications. Further research could explore hybrid approaches that dynamically select the best algorithm based on dataset characteristics.