Assignment Report

On

IMPLEMENTATION OF MAXIMAL CLIQUE ENUMERATION ALGORITHMS

BY

Venkata Saketh Dakuri (2022A7PS0056H),
Kavya Ganatra (2022A7PS0057H), Kalash Bhattad (2022A7PS0065H),
Pratyush Bindal (2022A7PS0119H), R V S Aashrey Kumar (2022A7PS0160H)

SUBMITTED IN FULFILLMENT OF THE REQUIREMENTS OF

ASSIGNMENT - I, CS F364: DESIGN & ANALYSIS OF ALGORITHMS

BIRLA INSTITUTE OF TECHNOLOGY AND SCIENCE PILANI
(RAJASTHAN) HYDERABAD CAMPUS (MARCH 2025)

Table of Contents

Introduction

Maximal Clique Enumeration

Maximal clique enumeration (MCE) is a fundamental problem in graph theory with extensive applications in bioinformatics, social network analysis, and data mining. A clique is a subset of vertices in an undirected graph where every pair of vertices is connected by an edge. A maximal clique is a clique that cannot be extended by adding more vertices without losing its completeness. Since the number of maximal cliques in a graph can be exponential in the number of vertices, finding an efficient algorithm for MCE remains a central challenge in theoretical and practical graph analysis.

Algorithms Explored

Numerous algorithms have been developed to address the problem of MCE, with a focus on both worst-case optimality and practical efficiency. Three prominent approaches which were explored are those introduced by Eppstein, Löffler, and Strash; Tomita, Tanaka, and Takahashi; and Chiba and Nishizeki. Each of these algorithms adopts a distinct strategy to improve computational efficiency, particularly in sparse graphs where the number of maximal cliques is significantly lower than in dense graphs.

Description

Eppstein, Löffler, and Strash (2010) introduced an algorithm that parametrizes the problem based on the degeneracy of the graph, a measure of its sparsity. By leveraging the degeneracy order, the algorithm significantly reduces redundant computations and ensures near-optimal performance for sparse graphs.

Tomita, Tanaka, and Takahashi (2006) developed an algorithm that achieves the worst-case optimal time complexity of $O(3^{n/3})$ for an n-vertex graph. Their approach improves recursive branching by employing an efficient pivot selection strategy, which minimizes the number of recursive calls and accelerates clique enumeration in practice. This algorithm has been widely recognized for its balance between worst-case optimality and practical efficiency.

Chiba and Nishizeki (1985) proposed an alternative method focusing on triangle listing and adjacency structure analysis. Their algorithm efficiently enumerates maximal cliques by leveraging the sparsity of real-world graphs, particularly those with low arboricity. This method is particularly effective for graphs where edge density varies significantly across different regions.

This report aims to explore and implement these three approaches, analyzing their computational efficiency and performance on various graph datasets. By comparing their effectiveness, we can gain deeper insights into the trade-offs between worst-case complexity and practical runtime efficiency in maximal clique enumeration.

Eppstein, Löffler, and Strash (2010)

Background and Theory

The Bron–Kerbosch Algorithm

The classical Bron–Kerbosch algorithm is a recursive backtracking method for enumerating all maximal cliques. It maintains three sets during recursion:

- R: a (possibly partial) clique under construction.
- P: candidate vertices that can be added to R (neighbors common to all vertices in R).
- X: vertices already processed (to avoid duplicate work).

A clique is maximal when both P and X are empty.

Pivoting for Efficiency

A standard improvement (due to Tomita et al.) chooses a pivot vertex u (from $P \cup X$) so that one only recurses into vertices in P that are not neighbors of u; this limits the number of recursive calls. The theory guarantees a worst-case time of $O(3^{(n/3)})$ (ignoring the output size).

Degeneracy Ordering

A key observation in the paper is that many practical graphs are sparse and exhibit low degeneracy.
- The degeneracy of a graph is the smallest number d such that every nonempty subgraph has a vertex of degree at most d.
- A degeneracy ordering is an ordering of vertices such that each vertex has at most d neighbours later in the order.

By processing vertices in this order, the "outer" level of the recursion is limited so that each vertex's candidate set P (later neighbours) always has size at most d. This leads to the overall running time of $O(d \cdot n \cdot 3^{(d/3)})$—nearly optimal in terms of the worst-case output size.

Algorithm

The algorithm essentially has two parts:

a. Preprocessing – Compute a degeneracy ordering.

b. Clique enumeration – For each vertex, use a modified version of Bron–Kerbosch with pivoting.

The outer loop processes each vertex v in the degeneracy order. For each such vertex:
- P is set to the "later" neighbors (those that appear after v in the degeneracy order).
- X contains "earlier" neighbors.
- R starts as $\{v\}$.

Then, the BronKerboschPivot routine is invoked to recursively expand R into all maximal cliques that contain v as the earliest vertex in the order. This guarantees that every maximal clique is reported exactly once.

Code Implementation

ELS Basic Implementation

```
#include <iostream>
#include <vector>
#include <unordered_set>
#include <fstream>
#include <sstream>
#include <algorithm>
#include <limits>
#include <chrono>
using namespace std;
using namespace std::chrono;


//Eppstein, Löffler & Strash (2010) algorithm for finding all maximal cliques in an undirected graph
void addEdge(int u,int v,vector<unordered_set<int> >& adj) {
    if (u>=adj.size()||v>=adj.size()){
        int newSize=max(u,v)+1;
        adj.resize(newSize);
    }
    adj[u].insert(v);
    adj[v].insert(u);
}


unordered_set<int> setintersect(const unordered_set<int>& A,const unordered_set<int>& B){
    unordered_set<int> res;
    for(int a:A){
        if(B.find(a)!=B.end()) res.insert(a);
    }
    return res;
}


unordered_set<int> setdiff(const unordered_set<int>& A,const unordered_set<int>& B) {
    unordered_set<int> res;
    for(int a:A){
        if(B.find(a)==B.end()) res.insert(a);
    }
    return res;
}


int maxCliqueSize = 0;
int totalMaximalCliques = 0;
unordered_map<int, int> cliqueSizeDistribution;


void BronKerboschPivot(unordered_set<int> P,unordered_set<int> R,unordered_set<int> X,const
vector<unordered_set<int> >& adj) {
    unordered_set<int> unionPX=P;
    unionPX.insert(X.begin(),X.end());
    if (unionPX.empty()){
        int cliqueSize = R.size();
        if (cliqueSize > 1) {
            maxCliqueSize = max(maxCliqueSize, cliqueSize);
            totalMaximalCliques++;
            cliqueSizeDistribution[cliqueSize]++;
        }
        return;
    }
    int u = *unionPX.begin();
    unordered_set<int> diff=setdiff(P,adj[u]);
    for(int v:diff){
        unordered_set<int> newP=setintersect(P,adj[v]);
        unordered_set<int> newX=setintersect(X,adj[v]);
        unordered_set<int> newR=R;
        newR.insert(v);
        BronKerboschPivot(newP, newR, newX, adj);
        P.erase(v);
        X.insert(v);
    }
```

```
}


vector<int> degeneracyorder(const vector<unordered_set<int> >& adj){
    int n=adj.size();
    vector<bool> used(n,false);
    vector<int> degree(n,0);
    for(int i=0;i<n;i++) degree[i]=adj[i].size();
    vector<int> ordering;
    for(int k=0;k<n;k++) {
        int u=-1;
        int minDeg=INT_MAX;
        for(int i=0;i<n;i++) {
            if(!used[i]&&degree[i]<minDeg) {
                minDeg=degree[i];
                u=i;
            }
        }
        if(u==-1) break;
        used[u]=true;
        ordering.push_back(u);
        for(int w:adj[u]){
            if (!used[w]) degree[w]--;
        }
    }
    reverse(ordering.begin(), ordering.end());
    return ordering;
}


void BronKerboschDegeneracy(const vector<unordered_set<int> >&adj) {
    int n=adj.size();
    vector<int> ordering=degeneracyorder(adj);
    vector<int> pos(n,0);
    for(int i=0;i<n;i++) pos[ordering[i]] = i;
    for (int i=0;i<n;i++) {
        int vi=ordering[i];
        unordered_set<int> P;
        for(int w:adj[vi]){
            if(pos[w]>pos[vi]) P.insert(w);
        }
        unordered_set<int> X;
        for(int w:adj[vi]){
            if(pos[w]<pos[vi]) X.insert(w);
        }
        unordered_set<int> R;
        R.insert(vi);
        BronKerboschPivot(P, R, X, adj);
    }
}


int main(int argc, char* argv[]) {
    ifstream infile(argv[1]);
    ofstream outfile("output.txt");
    string line;
    vector<pair<int, int>> edgeList;
    int maxVertex = 0;
    while(getline(infile, line)){
        if(line.empty() || line[0] == '#')
            continue;
        istringstream iss(line);
        int u, v;
        if(iss >> u >> v){
            edgeList.push_back({u, v});
            maxVertex = max(maxVertex, max(u, v));
        }
    }
    infile.close();

    vector<unordered_set<int>> adj(maxVertex + 1);
    for(auto &edge : edgeList)
        addEdge(edge.first, edge.second, adj);

    auto start = high_resolution_clock::now();
    BronKerboschDegeneracy(adj);
    auto stop = high_resolution_clock::now();
    auto duration = duration_cast<milliseconds>(stop - start);
```

```
    outfile << "Largest size of the clique: " << maxCliqueSize << endl;
    outfile << "Total number of maximal cliques: " << totalMaximalCliques << endl;
    outfile << "Execution time (ms): " << duration.count() << endl;
    outfile << "Distribution of different size cliques:" << endl;

    for(const auto& pair : cliqueSizeDistribution)
        outfile << "Size " << pair.first << ": " << pair.second << endl;
    outfile.close();

    return 0;
}
```

ELS Optimised Implementation

```
#include <iostream>
#include <vector>
#include <fstream>
#include <sstream>
#include <algorithm>
#include <limits>
#include <chrono>
using namespace std;
using namespace std::chrono;


//Eppstein, Löffler & Strash (2010) algorithm for finding all maximal cliques in an undirected graph - Optimised


int maxCliqueSize = 0;
long long totalMaximalCliques = 0;
vector<int> cliqueSizeDistribution;


void addEdge(int u, int v, vector<vector<int>>& adj) {
    if(u >= (int)adj.size() || v >= (int)adj.size()){
        int newSize = max(u, v) + 1;
        adj.resize(newSize);
    }
    adj[u].push_back(v);
    adj[v].push_back(u);
}


vector<int> intersectVectors(const vector<int>& A, const vector<int>& B) {
    vector<int> res;
    res.reserve(min(A.size(), B.size()));
    auto itA = A.begin(), itB = B.begin();
    while(itA != A.end() && itB != B.end()){
        if(*itA < *itB)
            ++itA;
        else if(*itB < *itA)
            ++itB;
        else {
            res.push_back(*itA);
            ++itA; ++itB;
        }
    }
    return res;
}


vector<int> diffVectors(const vector<int>& A, const vector<int>& B) {
    vector<int> res;
    res.reserve(A.size());
    auto itA = A.begin(), itB = B.begin();
    while(itA != A.end()){
        if(itB == B.end() || *itA < *itB){
            res.push_back(*itA);
            ++itA;
        } else if(*itA == *itB) {
            ++itA; ++itB;
        } else {
            ++itB;
```

```
        }
    }
    return res;
}


void BronKerboschPivot(vector<int> P, vector<int> R, vector<int> X,
                       const vector<vector<int>>& adj,
                       int &localMax, long long &localCount, vector<int>& localDist) {
    if(P.empty() && X.empty()){
        int cliqueSize = R.size();
        if(cliqueSize > 1){
            localMax = max(localMax, cliqueSize);
            localCount++;
            if(localDist.size() <= (size_t)cliqueSize)
                localDist.resize(cliqueSize+1, 0);
            localDist[cliqueSize]++;
        }
        return;
    }
    vector<int> unionPX = P;
    unionPX.insert(unionPX.end(), X.begin(), X.end());
    sort(unionPX.begin(), unionPX.end());
    int bestPivot = unionPX.front();
    size_t bestDegree = 0;
    for (int candidate : unionPX) {
        vector<int> common = intersectVectors(P, adj[candidate]);
        if(common.size() > bestDegree){
            bestDegree = common.size();
            bestPivot = candidate;
        }
    }
    vector<int> diffP = diffVectors(P, adj[bestPivot]);
    for (int v : diffP) {
        vector<int> newR = R;
        newR.push_back(v);
        vector<int> newP = intersectVectors(P, adj[v]);
        vector<int> newX = intersectVectors(X, adj[v]);
        BronKerboschPivot(newP, newR, newX, adj, localMax, localCount, localDist);
        P.erase(remove(P.begin(), P.end(), v), P.end());
        X.push_back(v);
        sort(X.begin(), X.end());
    }
}


vector<int> degeneracyOrder(const vector<vector<int>>& adj) {
    int n = adj.size();
    vector<int> d(n);
    for (int i = 0; i < n; i++) {
        d[i] = adj[i].size();
    }
    int maxDeg = *max_element(d.begin(), d.end());
    vector<vector<int>> bucket(maxDeg + 1);
    for (int i = 0; i < n; i++) {
        bucket[d[i]].push_back(i);
    }
    vector<bool> removed(n, false);
    vector<int> order;
    order.reserve(n);
    for (int k = 0; k < n; k++) {
        int currDeg = 0;
        while (currDeg <= maxDeg && bucket[currDeg].empty())
            currDeg++;
        if (currDeg > maxDeg)
            break;
        int u = bucket[currDeg].back();
        bucket[currDeg].pop_back();
        removed[u] = true;
        order.push_back(u);
        for (int v : adj[u]) {
            if (!removed[v]) {
                int oldDeg = d[v];
                auto &bkt = bucket[oldDeg];
                auto it = find(bkt.begin(), bkt.end(), v);
                if (it != bkt.end())
                    bkt.erase(it);
                d[v]--;
```

```
                bucket[d[v]].push_back(v);
            }
        }
    }
    reverse(order.begin(), order.end());
    return order;
}


void BronKerboschDegeneracy(const vector<vector<int>>& adj) {
    int n = adj.size();
    vector<int> ordering = degeneracyOrder(adj);
    vector<int> pos(n);
    for (int i = 0; i < n; i++) {
        pos[ordering[i]] = i;
    }
    maxCliqueSize = 0;
    totalMaximalCliques = 0;
    cliqueSizeDistribution.clear();
    for (int i = 0; i < n; i++){
        cout << "[DEBUG] Processing node " << ordering[i]
             << " (loop index " << i << ")" << endl;
        int vi = ordering[i];
        vector<int> P, X, R;
        for (int w : adj[vi]) {
            if (pos[w] > pos[vi])
                P.push_back(w);
        }
        for (int w : adj[vi]) {
            if (pos[w] < pos[vi])
                X.push_back(w);
        }
        sort(P.begin(), P.end());
        sort(X.begin(), X.end());
        R.push_back(vi);
        int localMaxClique = 0;
        long long localCliqueCount = 0;
        vector<int> localDist;
        BronKerboschPivot(P, R, X, adj, localMaxClique, localCliqueCount, localDist);
        maxCliqueSize = max(maxCliqueSize, localMaxClique);
        totalMaximalCliques += localCliqueCount;
        if(localDist.size() > cliqueSizeDistribution.size())
            cliqueSizeDistribution.resize(localDist.size(), 0);
        for (size_t j = 0; j < localDist.size(); j++)
            cliqueSizeDistribution[j] += localDist[j];
    }
}


int main(int argc, char* argv[]) {
    if(argc < 2){
        cerr << "Usage: " << argv[0] << " [input file]" << endl;
        return 1;
    }
    ifstream infile(argv[1]);
    ofstream outfile("output.txt");
    if(!infile){
        cerr << "Failed to open input file." << endl;
        return 1;
    }


    string line;
    vector<pair<int,int>> edgeList;
    int maxVertex = 0;
    while(getline(infile, line)){
        if(line.empty() || line[0] == '#') continue;
        istringstream iss(line);
        int u, v;
        if(iss >> u >> v){
            edgeList.push_back({u,v});
            maxVertex = max(maxVertex, max(u,v));
        }
    }
    infile.close();
    vector<vector<int>> adj(maxVertex + 1);
    for(auto &edge : edgeList)
        addEdge(edge.first, edge.second, adj);
```

```
    for(auto &neighbors : adj){
        sort(neighbors.begin(), neighbors.end());
        neighbors.erase(unique(neighbors.begin(), neighbors.end()), neighbors.end());
    }


    auto start = high_resolution_clock::now();
    BronKerboschDegeneracy(adj);
    auto stop = high_resolution_clock::now();
    auto duration = duration_cast<milliseconds>(stop - start).count();


    outfile << "Largest size of the clique: " << maxCliqueSize << "\n";
    outfile << "Total number of maximal cliques: " << totalMaximalCliques << "\n";
    outfile << "Execution time (ms): " << duration << "\n";
    outfile << "Distribution of different size cliques:\n";
    for(size_t i = 0; i < cliqueSizeDistribution.size(); i++){
        if(cliqueSizeDistribution[i] > 0)
            outfile << "Size " << i << ": " << cliqueSizeDistribution[i] << "\n";
    }
    outfile.close();
    return 0;
}
```

Walkthrough for the Code Implementation

The C++ code implements the idea. Let's walk through its main components of the basic ELS implementation:

Graph Representation and Utility Functions

- The graph is stored as a vector of unordered sets (adjacency list).
  - Function addEdge(u, v, adj) ensures the vector is resized if necessary and then adds v to u's neighbor set and vice versa.
- Set operations are implemented via helper functions:
  - setintersect(A, B) returns the intersection of two sets.
  - setdiff(A, B) returns the set difference A \ B.

The BronKerboschPivot Function

- This is the core recursive procedure that implements Bron–Kerbosch with pivoting. Its parameters are:
  - P: Candidate vertices for further expansion.
  - R: The current clique being built.
  - X: Vertices that have been considered already.
  - adj: The global adjacency list.
- Key steps in the function:
  - The union of P and X is calculated. If this union is empty, then R is a maximal clique. The clique's size is recorded, and statistics (total count and distribution by size) are updated.
  - A pivot (u) is chosen arbitrarily from $P \cup X$. (In some versions, one chooses the pivot that maximizes $|P \cap \Gamma(u)|$.)
  - The recursion then proceeds for every vertex v in $(P - \Gamma(u))$:

        - New candidate set (newP) is computed as the intersection of P with the neighbors of v.
        - New exclusion set (newX) is obtained by intersecting X with the neighbors of v.
        - R is extended by v.
        - After the recursive call, v is removed from P and added to X to avoid duplication.

Computing the Degeneracy Order

- The degeneracyorder(adj) function computes the ordering by:
    - Initializing an array of vertex degrees.
    - Repeatedly selecting the vertex with minimum degree among those not yet removed.
    - After removal, the degrees of its (not yet used) neighbors are decremented.
    - The resulting order is then reversed; the reverse order is the desired degeneracy order.
- This simple greedy strategy runs in O(n + m) time.

The BronKerboschDegeneracy Function

- This procedure ties everything together:
    - It computes the degeneracy ordering and also an auxiliary "position" vector that maps each vertex to its order.
    - For each vertex v (in degeneracy order), it builds:
        - P as all neighbors of v that come later in the order.
        - X as all neighbors of v that come earlier.
        - It then calls BronKerboschPivot with R initially {v}.
- By ensuring that every maximal clique is reported only when its earliest vertex (according to the degeneracy order) is processed, the algorithm avoids repeating cliques.

The Main Function

- The main function performs the following:
    - It opens an input file (with graph data).
    - It reads the graph (number of vertices and edges) and adds edges accordingly.
    - It then calls BronKerboschDegeneracy to enumerate all maximal cliques.
    - Execution time is measured using the chrono library.
    - Finally, statistics such as the largest clique size, the total number of maximal cliques, and the clique size distribution are written to an output file.

Optimisations Performed

1. **Pivot Strategy**
    - In Bron–Kerbosch, choosing a good pivot can significantly reduce the number of recursive calls.

- In the optimized version (ELS Optimisation), it calculates a best pivot by checking each candidate (in union(P, X)) and measuring how many neighbors it shares with P. The one with the greatest overlap is chosen as the pivot, thereby minimizing the number of branches.
- By contrast, the earlier version (ELS Basic) just takes the first element from union(P, X) as the pivot. This can lead to a bigger branching factor, because the chosen pivot may not be the one that maximally reduces the size of P.

2. **Degeneracy Ordering**
   - "Degeneracy ordering" sorts the vertices so that each vertex has a limited number of neighbors later in the ordering. Processing vertices in this sequence further constrains the search.
   - The optimized version fully implements degeneracy ordering by placing vertices into buckets based on degrees and then taking them one by one in ascending degree order, updating neighboring degrees as it goes.
   - The original version (ELS Basic) uses a simpler approach to degeneracy, essentially picking the vertex with the smallest current degree among those unremoved—but it does not employ a bucket-based approach. The bucket-based approach in ELS Optimisation can be more precise and efficient for large graphs.
3. **Sorting & Unique**
   - The optimized code (ELS Optimisation) sorts vectors P and X before performing set intersection or difference. It then uses pointer-based iteration (e.g., intersectVectors), which can be faster than repeated hash lookups in an unordered_set.
   - The earlier code uses set operations on unordered_set, which might be simpler to implement but can have larger overhead when sets are big.
4. **Data Structure Choice**
   - The optimized version uses arrays/vectors (e.g., vector<int>) combined with sorted set operations. Sorting once and performing merge-like intersections is typically O(|A| + |B|).
   - The original version uses unordered_set, which can give constant-time membership checks but may incur overhead from hashing—especially in intersections or difference operations where you must loop through the entire set.
5. **Local vs. Global Statistics**
   - In the optimized version, each recursive call manages "localDist," "localMax," and "localCount," which are then merged into global (or outer) statistics at a higher level. This consolidates updates and often avoids frequent global data structure resizing.
   - The original version updates global variables (maxCliqueSize, totalMaximalCliques, cliqueSizeDistribution) directly in each call, which may lead to more overhead—each clique discovered immediately changes the unordered_map or global counters.

NOTE : An isolated vertex is by definition a clique, but in the ELS paper it has been mentioned that in practical use-cases such as MCE(Maximal Clique Enumeration), it is only of practical use to check for maximal cliques of size >=2.

Results Obtained

Basic ELS Results

Wiki-Vote Dataset

```
No. of maximal cliques: 459002
Number of nodes: 7115
No. of elements in degeneracy ordering: 7115
Time taken: 10413 milliseconds
Clique size distribution:
Clique size: 17, Frequency: 23
Clique size: 15, Frequency: 740
Clique size: 16, Frequency: 208
Clique size: 14, Frequency: 2329
Clique size: 13, Frequency: 5449
Clique size: 12, Frequency: 11640
Clique size: 11, Frequency: 21736
Clique size: 10, Frequency: 35470
Clique size: 9, Frequency: 54456
Clique size: 8, Frequency: 76732
Clique size: 5, Frequency: 48416
Clique size: 6, Frequency: 68872
Clique size: 4, Frequency: 27292
Clique size: 3, Frequency: 13718
Clique size: 7, Frequency: 83266
Clique size: 2, Frequency: 8655
Maximum clique size: 17
```

Email-Enron Dataset

```
No. of maximal cliques: 226859
Number of nodes: 36692
No. of elements in degeneracy ordering: 36692
Time taken: 8239 milliseconds
Clique size distribution:
Clique size: 20, Frequency: 6
Clique size: 19, Frequency: 10
Clique size: 18, Frequency: 41
Clique size: 17, Frequency: 286
Clique size: 16, Frequency: 1178
Clique size: 15, Frequency: 3157
Clique size: 14, Frequency: 7417
Clique size: 13, Frequency: 11487
Clique size: 12, Frequency: 15181
Clique size: 10, Frequency: 21393
Clique size: 11, Frequency: 17833
Clique size: 9, Frequency: 22884
Clique size: 6, Frequency: 22715
Clique size: 5, Frequency: 18143
Clique size: 4, Frequency: 13319
Clique size: 8, Frequency: 24766
Clique size: 3, Frequency: 7077
Clique size: 7, Frequency: 25896
Clique size: 2, Frequency: 14070
Maximum clique size: 20
```

As-Skitter Dataset

```
No. of maximal cliques: 37322355
Number of nodes: 1696415
No. of elements in degeneracy ordering: 1696415
Time taken: 31717463 milliseconds
Clique size distribution:
Clique size: 67, Frequency: 4
Clique size: 65, Frequency: 49
Clique size: 66, Frequency: 22
Clique size: 64, Frequency: 84
Clique size: 62, Frequency: 242
Clique size: 61, Frequency: 283
Clique size: 63, Frequency: 146
Clique size: 60, Frequency: 405
Clique size: 55, Frequency: 2042
Clique size: 56, Frequency: 1080
Clique size: 57, Frequency: 546
Clique size: 58, Frequency: 449
```

```
Clique size: 59, Frequency: 447
Clique size: 51, Frequency: 25637
Clique size: 52, Frequency: 17707
Clique size: 53, Frequency: 9514
Clique size: 54, Frequency: 3737
Clique size: 48, Frequency: 62437
Clique size: 49, Frequency: 39822
Clique size: 46, Frequency: 158352
Clique size: 47, Frequency: 99522
Clique size: 45, Frequency: 222461
Clique size: 44, Frequency: 275995
Clique size: 43, Frequency: 321829
Clique size: 42, Frequency: 367879
Clique size: 41, Frequency: 420796
Clique size: 39, Frequency: 520239
Clique size: 38, Frequency: 583922
Clique size: 40, Frequency: 474301
Clique size: 37, Frequency: 663650
Clique size: 36, Frequency: 744634
Clique size: 35, Frequency: 809485
Clique size: 33, Frequency: 946717
Clique size: 32, Frequency: 1017560
Clique size: 34, Frequency: 878552
Clique size: 31, Frequency: 1055653
Clique size: 29, Frequency: 999860
Clique size: 30, Frequency: 1034106
Clique size: 28, Frequency: 955212
Clique size: 27, Frequency: 892719
Clique size: 26, Frequency: 818353
Clique size: 25, Frequency: 753633
Clique size: 24, Frequency: 706753
Clique size: 23, Frequency: 673924
Clique size: 22, Frequency: 640890
Clique size: 21, Frequency: 611976
Clique size: 20, Frequency: 600192
Clique size: 19, Frequency: 639413
Clique size: 18, Frequency: 729601
Clique size: 17, Frequency: 839330
Clique size: 16, Frequency: 939987
Clique size: 15, Frequency: 980831
Clique size: 14, Frequency: 945194
Clique size: 13, Frequency: 877282
Clique size: 12, Frequency: 798073
Clique size: 11, Frequency: 728098
Clique size: 9, Frequency: 608937
Clique size: 10, Frequency: 665661
Clique size: 8, Frequency: 588889
Clique size: 7, Frequency: 598284
Clique size: 6, Frequency: 684873
Clique size: 5, Frequency: 939336
Clique size: 4, Frequency: 1823321
Clique size: 50, Frequency: 30011
Clique size: 3, Frequency: 3171609
Clique size: 2, Frequency: 2319807
Maximum clique size: 67
```

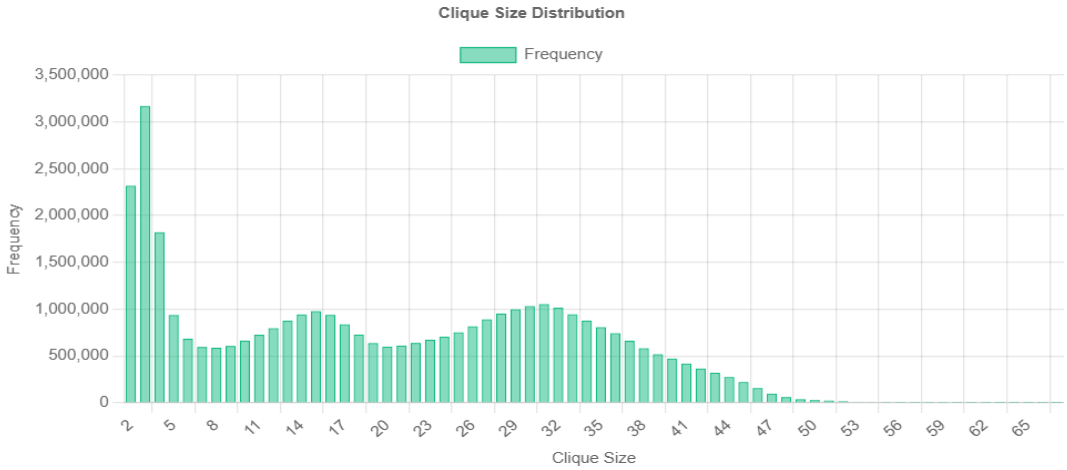

Fig: As-Skitter Dataset Results

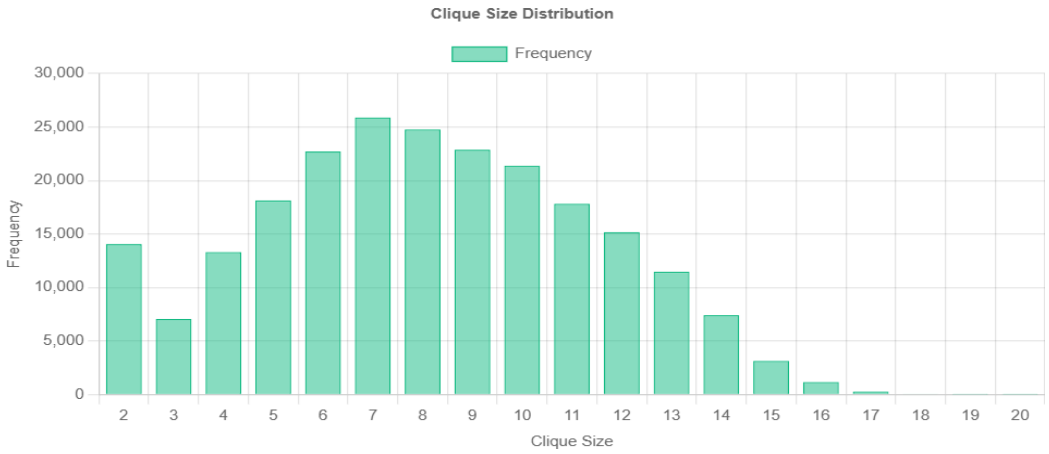

Fig: Email-Enron Dataset Results

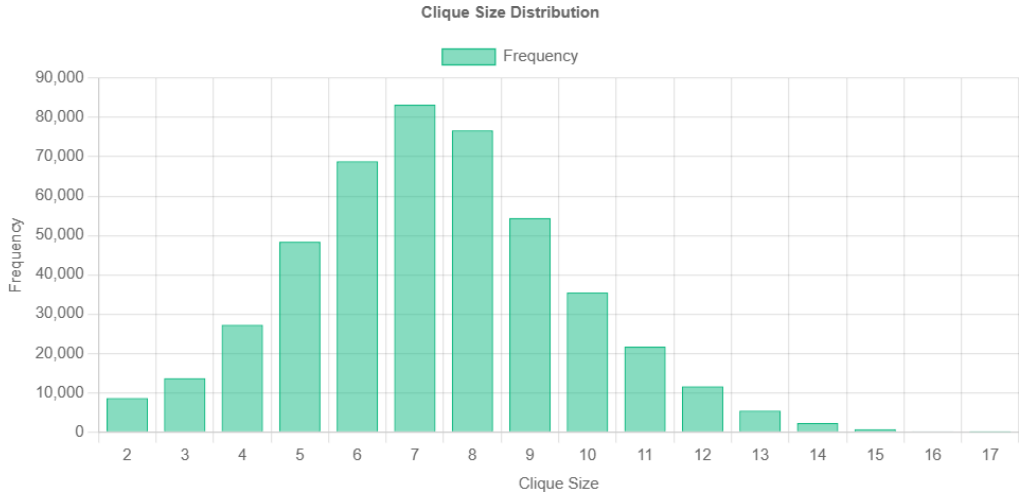

Fig: Wiki-Vote Dataset Results

Optimised ELS Results

Wiki-Vote Dataset

```
Largest size of the clique: 17
Total number of maximal cliques: 459002
Execution time (ms): 2156
Distribution of different size cliques:
Size 2: 8655
Size 3: 13718
Size 4: 27292
Size 5: 48416
Size 6: 68872
Size 7: 83266
Size 8: 76732
Size 9: 54456
Size 10: 35470
Size 11: 21736
Size 12: 11640
Size 13: 5449
Size 14: 2329
Size 15: 740
```

```
Size 16: 208
Size 17: 23
Maximal Clique Size: 17
```

Email-Enron Dataset

```
Largest size of the clique: 20
Total number of maximal cliques: 226859
Execution time (ms): 1540
Distribution of different size cliques:
Size 2: 14070
Size 3: 7077
Size 4: 13319
Size 5: 18143
Size 6: 22715
Size 7: 25896
Size 8: 24766
Size 9: 22884
Size 10: 21393
Size 11: 17833
Size 12: 15181
Size 13: 11487
Size 14: 7417
Size 15: 3157
Size 16: 1178
Size 17: 286
Size 18: 41
Size 19: 10
Size 20: 6
Maximal Clique Size: 20
```

As-Skitter Dataset

```
Largest size of the clique: 67
Total number of maximal cliques: 37322355
Execution time (ms): 651343
Distribution of different size cliques:
Size 2: 2319807
Size 3: 3171609
Size 4: 1823321
Size 5: 939336
Size 6: 684873
Size 7: 598284
Size 8: 588889
Size 9: 608937
Size 10: 665661
Size 11: 728098
Size 12: 798073
Size 13: 877282
Size 14: 945194
Size 15: 980831
Size 16: 939987
Size 17: 839330
Size 18: 729601
Size 19: 639413
Size 20: 600192
Size 21: 611976
Size 22: 640890
Size 23: 673924
Size 24: 706753
Size 25: 753633
Size 26: 818353
Size 27: 892719
Size 28: 955212
Size 29: 999860
Size 30: 1034106
Size 31: 1055653
Size 32: 1017560
Size 33: 946717
Size 34: 878552
Size 35: 809485
Size 36: 744634
Size 37: 663650
Size 38: 583922
Size 39: 520239
Size 40: 474301
Size 41: 420796
Size 42: 367879
Size 43: 321829
Size 44: 275995
Size 45: 222461
Size 46: 158352
Size 47: 99522
Size 48: 62437
Size 49: 39822
Size 50: 30011
```

```
Size 51: 25637
Size 52: 17707
Size 53: 9514
Size 54: 3737
Size 55: 2042
Size 56: 1080
Size 57: 546
Size 58: 449
Size 59: 447
Size 60: 405
Size 61: 283
Size 62: 242
Size 63: 146
Size 64: 84
Size 65: 49
Size 66: 22
Size 67: 4
Maximal Clique Size: 67
```

Tomita, Tanaka, and Takahashi (2006)

Background and Theory

The primary purpose of this paper is to overcome the limitations of earlier algorithms for maximal clique enumeration by delivering both theoretical optimality and enhanced practical efficiency. Earlier methods, such as the Bron–Kerbosch algorithm and the algorithm by Tsukiyama et al., have been widely used but suffer from key drawbacks:

Lack of Worst-Case Guarantees: Many previous approaches do not provide a rigorous worst-case time complexity analysis. This is problematic given that an nnn-vertex graph can potentially contain up to 3^{n/3} maximal cliques (as established by Moon and Moser). Without matching this theoretical bound, algorithms can exhibit suboptimal performance in worst-case scenarios.
Redundant Exploration: Prior algorithms often generate the same maximal clique multiple times or explore redundant branches in the search space. This inefficiency not only wastes computational resources but also complicates the overall enumeration process.

Inefficient Output Representation: Some methods require outputting entire cliques explicitly, leading to high memory usage and processing time, especially when the cliques themselves are large or numerous.

Tomita, Tanaka, and Takahashi address these issues with several key improvements:

Optimal Worst-Case Time Complexity: The algorithm achieves a worst-case running time of $O(3^{n/3})$, which is optimal given the worst-case number of maximal cliques in an nnn-vertex graph. This result aligns with the Moon–Moser bound, ensuring that the algorithm performs as well as theoretically possible in the worst case.
Effective Pruning Strategies: By partitioning the search space into candidate (CAND) and finished (FINI) sets and applying a smart pivot selection strategy, the algorithm avoids redundant exploration. It only expands vertices that are necessary to generate new maximal cliques, thereby eliminating duplicate work.
Compact Output Representation: Instead of explicitly listing every maximal clique—which could be very resource-intensive—the algorithm outputs a tree-like sequence that implicitly represents the clique structure. This approach significantly reduces both time and memory overhead associated with output generation.
Enhanced Practical Performance: Computational experiments demonstrate that the algorithm not only meets the theoretical bounds but also runs considerably faster than previous algorithms on a variety of graph instances, making it highly effective for real-world applications.

In summary, this paper makes a substantial contribution by presenting an algorithm that addresses the shortcomings of earlier methods—namely, the lack of worst-case optimality, redundant exploration of the search space, and inefficient output representation—while delivering both robust theoretical guarantees and superior practical performance.

Algorithm

- There are few terms that are important in the algorithm SUBG,FINI,CAND and Γ(x). Γ(x) represents the set of vertices that are connected to vertex x in the graph G.
- If Q is the set of vertices (p1,p2,p3,. .... ,pn) represent a complete subgraph at any point,
- SUBG represents the set of vertices V ∩ Γ(p1) ∩ Γ(p2) ∩ Γ(p3) ∩ Γ(p4) ∩ Γ(p5) …….∩Γ(pn).
- Basically SUBG is the set of those vertices in V that are adjacent to vertices in our current set Q.
- Goal is to expand Q recursively exploring all possibilities but optimally. Formally, goal is to find candidate vertices q such that Q U {q} is also a complete subgraph.
- So, when SUBG is an empty set, this means that the Q at that point is in fact a maximal clique.
- FINI is a subset of SUBG that are vertices that are already expanded.
- CAND is a subset of SUBG that are vertices that are yet to be expanded.

How does EXPAND work?

- For every q that belongs to SUBG, SUBGq represents the set of vertices that are in SUBG and also are adjacent to q. (SUBGq = SUBG ∩ Γ(q))
- FINIq is FINI ∩ Γ(q), CANDq is CAND ∩ Γ(q)
- Now at every point in the algorithm, every element in SUBG is a potential vertex to be expanded. But, should we expand all the vertices recursively? Or can we make a smarter choice?
- The answer is yes. The first obvious pruning would be to only expand vertices from CAND to expand Q to Q U {q}.
- Even now, we can make a smarter choice by not expanding all elements of CAND .
- A key observation is that, if there exists a vertex u in SUBG such that all maximal cliques Q U {u} have been enumerated, we do not need to expand those vertices that are adjacent to u (Γ(u)) and also are in CAND. Mathematically, we only expand |CAND - Γ(u)| number of vertices.
- The next question that arises is what ‘u’ do we choose?
- For the sake of optimality, we will choose that ‘u’ that minimises CAND - Γ(u). In order to minimise CAND - Γ(u), we need to maximise CAND ∩ Γ(u). That means we have to find that vertex in SUBG that has most number of vertices in CAND that are adjacent to it.This is the vertex that we will expand i.e., q.
- We will recursively call EXPAND on SUBGq and CANDq.
- After this vertex has been explored/expanded we will remove it from Q.

Code Implementation

TTT Basic Implementation

```
#include <iostream>
#include <vector>
#include <unordered_set>
#include <unordered_map>
#include <fstream>
#include <sstream>
#include <algorithm>
#include <limits>
#include <chrono>
#include <map>
using namespace std;
using namespace std::chrono;


// Tomita, Tanata & Takahashi (2006) algorithm for finding all maximal cliques in an undirected graph


vector<int> Q;
int maxCliqueSize = 0;
int totalMaximalCliques = 0;
map<int, int> cliqueSizeDistribution;


void addEdge(int u, int v, vector<unordered_set<int>>& adj) {
    if(u >= adj.size() || v >= adj.size()){
        int newSize = max(u, v) + 1;
        adj.resize(newSize);
    }
    adj[u].insert(v);
    adj[v].insert(u);
}


void EXPAND(unordered_set<int> SUBG, unordered_set<int> CAND, vector<unordered_set<int>>& adj) {
    if(SUBG.empty()){
        int cliqueSize = Q.size();
        if (cliqueSize > 1) {
            maxCliqueSize = max(maxCliqueSize, cliqueSize);
            totalMaximalCliques++;
            cliqueSizeDistribution[cliqueSize]++;
        }
        return;
    } else {
        int u = -1;
        int maxCount = 0;
        for (int x : SUBG) {
            int cnt = 0;
            for (int y : CAND) {
                if(adj[x].find(y) != adj[x].end())
                    ++cnt;
            }
            if(cnt > maxCount){
                maxCount = cnt;
                u = x;
            }
        }
        unordered_set<int> Extu;
        for (int v : CAND) {
            if(adj[u].find(v) == adj[u].end())
                Extu.insert(v);
        }
        unordered_set<int> FINI;
        while(!Extu.empty()){
            int q = *Extu.begin();
            Q.push_back(q);
            unordered_set<int> SUBGq;
            for (int v : SUBG) {
                if(adj[q].find(v) != adj[q].end())
                    SUBGq.insert(v);
            }
            unordered_set<int> CANDq;
            for (int v : CAND) {
                if(adj[q].find(v) != adj[q].end())
```

```
                    CANDq.insert(v);
                }
                EXPAND(SUBGq, CANDq, adj);
                CAND.erase(q);
                FINI.insert(q);
                Q.pop_back();
                Extu.clear();
                for (int v : CAND) {
                    if(adj[u].find(v) == adj[u].end())
                        Extu.insert(v);
                }
            }
        }
}


void CLIQUES(vector<unordered_set<int>>& adj, int V) {
    unordered_set<int> Vset;
    for(int i = 0; i < V; i++)
        Vset.insert(i);
    EXPAND(Vset, Vset, adj);
}


int main(int argc, char* argv[]) {
    ios_base::sync_with_stdio(false);
    cin.tie(nullptr);
    cout.tie(nullptr);
    ifstream infile(argv[1]);
    ofstream outfile("output.txt");
    string line;
    vector<pair<int, int>> edgeList;
    int maxVertex = 0;
    while(getline(infile, line)){
        if(line.empty() || line[0] == '#')
            continue;
        istringstream iss(line);
        int u, v;
        if(iss >> u >> v){
            edgeList.push_back({u, v});
            maxVertex = max(maxVertex, max(u, v));
        }
    }
    infile.close();
    vector<unordered_set<int>> adj(maxVertex + 1);
    for(auto &edge : edgeList)
        addEdge(edge.first, edge.second, adj);

    auto start = high_resolution_clock::now();
    CLIQUES(adj, maxVertex + 1);
    auto stop = high_resolution_clock::now();
    auto duration = duration_cast<milliseconds>(stop - start);
    outfile << "Largest size of the clique: " << maxCliqueSize << endl;
    outfile << "Total number of maximal cliques: " << totalMaximalCliques << endl;
    outfile << "Execution time (ms): " << duration.count() << endl;
    outfile << "Distribution of different size cliques:" << endl;
    for(const auto& pair : cliqueSizeDistribution) outfile << "Size " << pair.first << ": " << pair.second
<< endl;
    outfile.close();
    return 0;
}
```

TTT Optimised Implementation

```
#include <iostream>
#include <vector>
#include <unordered_set>
#include <unordered_map>
#include <fstream>
#include <sstream>
#include <algorithm>
#include <limits>
#include <chrono>
#include <map>
using namespace std;
using namespace std::chrono;

// Tomita, Tanata & Takahashi (2006) algorithm for finding all maximal cliques in an undirected graph -
Optimised

vector<int> Q;
int maxCliqueSize = 0;
int totalMaximalCliques = 0;
map<int, int> cliqueSizeDistribution;

void addEdge(int u, int v, vector<unordered_set<int>>& adj) {
   if(u >= adj.size() || v >= adj.size()){
       int newSize = max(u, v) + 1;
       adj.resize(newSize);
   }
   adj[u].insert(v);
   adj[v].insert(u);
}

void EXPAND(unordered_set<int> SUBG, unordered_set<int> CAND, vector<unordered_set<int>>& adj) {
   if(SUBG.empty()) {
       int cliqueSize = Q.size();
       if(cliqueSize > 1) {
           maxCliqueSize = max(maxCliqueSize, cliqueSize);
           totalMaximalCliques++;
           cliqueSizeDistribution[cliqueSize]++;
       }
       return;
   }
   int u = -1, maxCount = 0;
   for(auto it = SUBG.begin(); it != SUBG.end(); ++it) {
       int cnt = 0;
       for(auto jt = CAND.begin(); jt != CAND.end(); ++jt) {
           if(adj[*it].find(*jt) != adj[*it].end())
               ++cnt;
       }
       if(cnt > maxCount) {
           maxCount = cnt;
           u = *it;
       }
   }
   unordered_set<int> Extu;
   Extu.reserve(CAND.size());
   for(auto it = CAND.begin(); it != CAND.end(); ++it) {
       if(adj[u].find(*it) == adj[u].end())
           Extu.insert(*it);
   }
   while(!Extu.empty()){
       int q = *Extu.begin();
       Q.push_back(q);
       unordered_set<int> SUBGq;
       SUBGq.reserve(SUBG.size());
       for(auto it = SUBG.begin(); it != SUBG.end(); ++it) {
           if(adj[q].find(*it) != adj[q].end())
               SUBGq.insert(*it);
       }
       unordered_set<int> CANDq;
       CANDq.reserve(CAND.size());
       for(auto it = CAND.begin(); it != CAND.end(); ++it) {
           if(adj[q].find(*it) != adj[q].end())
               CANDq.insert(*it);
       }
       EXPAND(std::move(SUBGq), std::move(CANDq), adj);
       CAND.erase(q);
```

```
        Extu.erase(q);
        Q.pop_back();
    }
}

void CLIQUES(vector<unordered_set<int>>& adj, int V) {
    unordered_set<int> Vset;
    vector<int> ordering;
    for (int i = 0; i < V; i++){
        Vset.insert(i);
        ordering.push_back(i);
    }
    for(auto it = ordering.begin(); it != ordering.end(); ++it) {
        cout << "[DEBUG] Processing node " << *it
             << " (loop index " << distance(ordering.begin(), it) << ")" << endl;
    }
    EXPAND(Vset, Vset, adj);
}

int main(int argc, char* argv[]) {
    struct rlimit rl;
    rlim_t stack_size = 512 * 1024 * 1024;
    int result = getrlimit(RLIMIT_STACK, &rl);
    if (result != 0) {
        cerr << "Error getting stack limit: " << strerror(errno) << endl;
        return 1;
    }
    if (rl.rlim_cur < stack_size) {
        rl.rlim_cur = stack_size;
        if (rl.rlim_max < rl.rlim_cur) {
            rl.rlim_max = rl.rlim_cur;
        }
        result = setrlimit(RLIMIT_STACK, &rl);
        if (result != 0) {
            cerr << "Error setting stack limit: " << strerror(errno) << endl;
        } else {
            cerr << "Stack size increased to " << (stack_size / (1024 * 1024)) << " MB" << endl;
        }
    }

    ios_base::sync_with_stdio(false);
    cin.tie(nullptr);
    cout.tie(nullptr);

    if(argc < 2){
        cerr << "Usage: " << argv[0] << " [input file]" << endl;
        return 1;
    }

    ifstream infile(argv[1]);
    ofstream outfile("output.txt");
    string line;
    vector<pair<int, int>> edgeList;
    int maxVertex = 0;
    while(getline(infile, line)){
        if(line.empty() || line[0] == '#')
            continue;
        istringstream iss(line);
        int u, v;
        if(iss >> u >> v){
            edgeList.push_back({u, v});
            maxVertex = max(maxVertex, max(u, v));
        }
    }
    infile.close();
    vector<unordered_set<int>> adj(maxVertex + 1);
    for(auto &edge : edgeList)
        addEdge(edge.first, edge.second, adj);

    auto start = high_resolution_clock::now();
    CLIQUES(adj, maxVertex + 1);
    auto stop = high_resolution_clock::now();
    auto duration = duration_cast<milliseconds>(stop - start);

    outfile << "Largest size of the clique: " << maxCliqueSize << "\n";
    outfile << "Total number of maximal cliques: " << totalMaximalCliques << "\n";
    outfile << "Execution time (ms): " << duration.count() << "\n";
    outfile << "Distribution of different size cliques:\n";
```

```
    for(auto it = cliqueSizeDistribution.begin(); it != cliqueSizeDistribution.end(); ++it) {
        outfile << "Size " << it->first << ": " << it->second << "\n";
    }
    outfile.close();
    return 0;
}
```

Walkthrough for the Code Implementation

Graph Representation and Utility Functions

The graph is stored as a vector of unordered sets (adjacency list):

- Each index in the vector represents a vertex, and its corresponding unordered set contains its adjacent vertices.
- This allows efficient edge lookups and set operations.

Function addEdge(u, v, adj)

- Ensures the adjacency list vector is resized if necessary, based on the maximum vertex index.
- Adds vertex v to the adjacency list of u, and vice versa.
- Uses unordered sets for fast lookup and insertion.

The EXPAND Function

This is the core recursive procedure that implements Tomita's pivot-based clique enumeration. Its parameters are:

- SUBG: The candidate set of vertices that can extend the current clique.
- CAND: The set of vertices still available for expansion.
- adj: The adjacency list representing the graph.

Key steps in the function:

- **Base Case:** If SUBG is empty, Q forms a maximal clique:
    - The size of Q is recorded, and statistics (total count and distribution by size) are updated.
    - If the clique size is greater than 1, it contributes to the output.
- **Pivot Selection:**
    - The algorithm selects a pivot u from SUBG, choosing the vertex with the maximum number of neighbors in CAND.
    - This helps reduce the number of recursive calls.
- **Handling Non-Neighbors (Extu)**:
    - Extu is initialized to the vertices in CAND that are **not** neighbors of u.

- These are processed first, ensuring efficient branching.

- **Recursive Expansion:**
  - For each vertex q in Extu:
    - Q is extended with q.
    - A new SUBGq and CANDq are computed by intersecting SUBG and CAND with the neighbors of q.
    - The function is recursively called on this new set.
    - After recursion, q is moved from CAND to FINI to prevent duplicate processing.

The CLIQUES Function

This function initializes the search process:

- Creates a set Vset containing all vertices in the graph.
- Calls EXPAND(Vset, Vset, adj), starting with all vertices as both candidates (CAND) and potential clique members (SUBG).

The main Function

The main function performs the following:

- **Graph Input Handling:**
  - Reads edge pairs from an input file.
  - Ignores empty lines and comment lines (starting with #).
  - Uses maxVertex to determine the highest vertex index, ensuring correct adjacency list sizing.
  - Calls addEdge(u, v, adj) for each valid edge.
- **Maximal Clique Enumeration:**
  - Calls CLIQUES(adj, maxVertex + 1), initiating the recursive enumeration process.
- **Performance Measurement:**
  - Uses chrono to measure execution time in milliseconds.
- **Output Results:**
  - Writes the largest clique size found.
  - Writes the total number of maximal cliques detected.
  - Writes execution time.
  - Writes the distribution of cliques by size to output.txt.

Optimisations Performed

1. **Stack Limit Increase:**

- The optimized version raises the stack size (via setrlimit) to prevent stack overflows for deeper recursion.
- The basic version does not modify stack limits.

2. **Memory Reservation:**

- Both use the same Tomita, Tanata & Takahashi approach, but in the optimized version EXPAND procedure, the optimized code uses reserve calls before inserting. The basic version does not, which can cause more frequent allocations.
- The basic version has a similar pivoting strategy but re-derives Extu in every iteration rather than caching or reusing sets.

3. **Reserve & Move Semantics:**

- The optimized version uses reserve on some containers (e.g., Extu) and utilizes std::move for SUBG and CAND in recursive calls, reducing allocations and copies.
- The basic version repeatedly inserts into new sets without reusing the existing memory as efficiently.

4. **Clearing Extu - Part of Reserve Semantics explained more clearly:**

- The basic code manually re-creates Extu after each removal, while the optimized one simply erases elements from it. This can reduce overhead if Extu is used repeatedly.

NOTE : An isolated vertex is by definition a clique, but in the ELS paper it has been mentioned that in practical use-cases such as MCE(Maximal Clique Enumeration), it is only of practical use to check for maximal cliques of size >=2.

Results Obtained

Basic TTT Results

Wiki-Vote Dataset

```
Largest size of the clique: 17
Total number of maximal cliques: 459002
Execution time (ms): 4068
Distribution of different size cliques:
Size 2: 8655
Size 3: 13718
Size 4: 27292
Size 5: 48416
Size 6: 68872
Size 7: 83266
Size 8: 76732
Size 9: 54456
Size 10: 35470
Size 11: 21736
Size 12: 11640
Size 13: 5449
Size 14: 2329
Size 15: 740
Size 16: 208
Size 17: 23
```

Email-Enron Dataset

```
Largest size of the clique: 20
Total number of maximal cliques: 226859
Execution time (ms): 39666
Distribution of different size cliques:
Size 2: 14070
Size 3: 7077
Size 4: 13319
Size 5: 18143
Size 6: 22715
Size 7: 25896
Size 8: 24766
Size 9: 22884
Size 10: 21393
Size 11: 17833
Size 12: 15181
Size 13: 11487
Size 14: 7417
Size 15: 3157
Size 16: 1178
Size 17: 286
Size 18: 41
Size 19: 10
Size 20: 6
```

As-Skitter Dataset

```
Largest size of the clique: 67
Total number of maximal cliques: 37322355
Execution time (ms): 62572973
Distribution of different size cliques:
Size 2: 2319807
Size 3: 3171609
Size 4: 1823321
Size 5: 939336
Size 6: 684873
Size 7: 598284
Size 8: 588889
Size 9: 608937
Size 10: 665661
Size 11: 728098
Size 12: 798073
Size 13: 877282
Size 14: 945194
Size 15: 980831
Size 16: 939987
Size 17: 839330
Size 18: 729601
Size 19: 639413
Size 20: 600192
Size 21: 611976
Size 22: 640890
Size 23: 673924
Size 24: 706753
Size 25: 753633
Size 26: 818353
Size 27: 892719
Size 28: 955212
Size 29: 999860
Size 30: 1034106
Size 31: 1055653
Size 32: 1017560
Size 33: 946717
Size 34: 878552
Size 35: 809485
Size 36: 744634
Size 37: 663650
Size 38: 583922
Size 39: 520239
Size 40: 474301
Size 41: 420796
Size 42: 367879
Size 43: 321829
Size 44: 275995
Size 45: 222461
Size 46: 158352
Size 47: 99522
Size 48: 62437
Size 49: 39822
Size 50: 30011
Size 51: 25637
Size 52: 17707
```

```
Size 53: 9514
Size 54: 3737
Size 55: 2042
Size 56: 1080
Size 57: 546
Size 58: 449
Size 59: 447
Size 60: 405
Size 61: 283
Size 62: 242
Size 63: 146
Size 64: 84
Size 65: 49
Size 66: 22
Size 67: 4
```

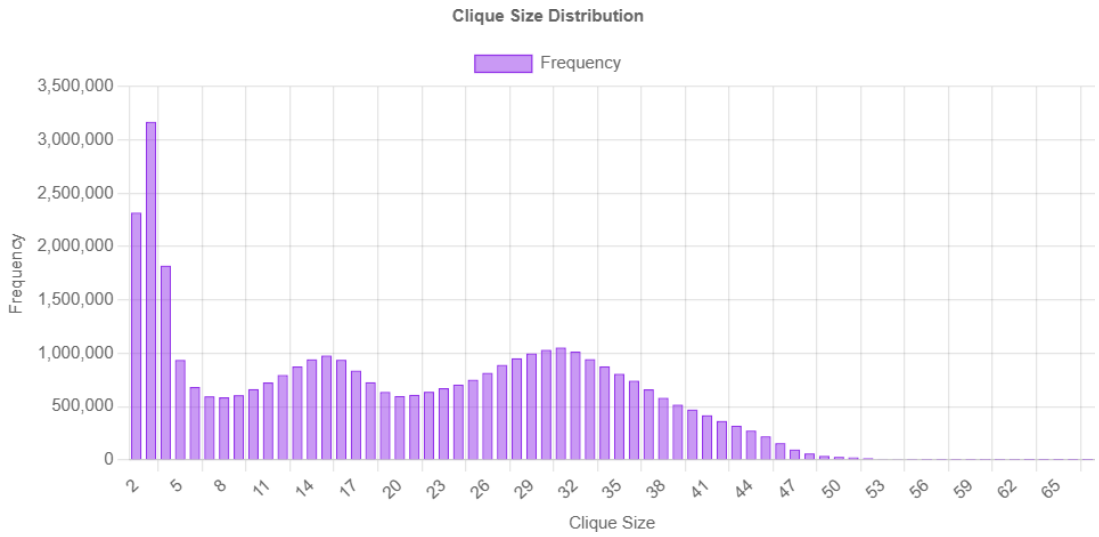

Fig: As-Skitter Dataset Results

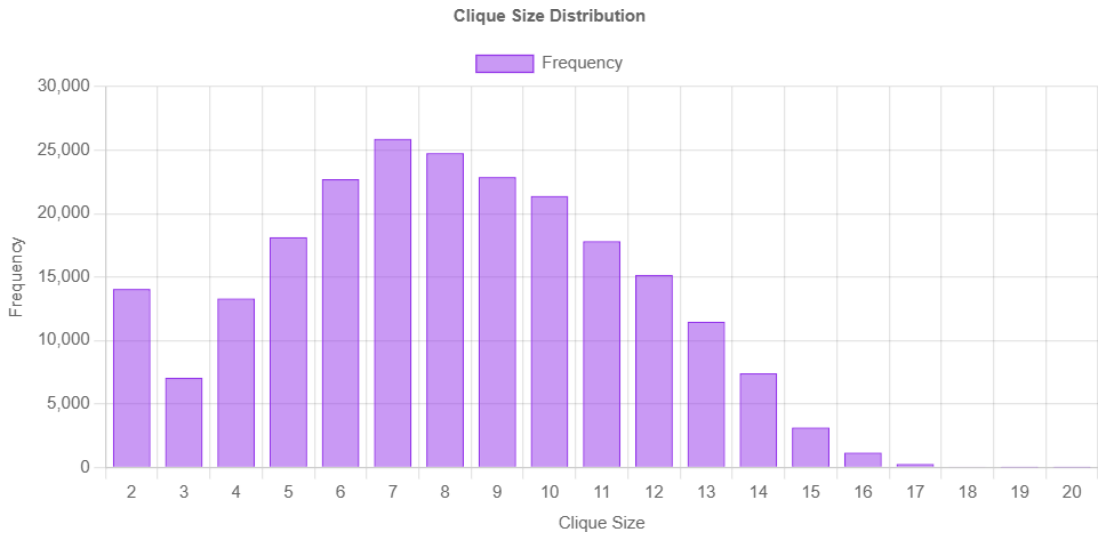

Fig: Email-Enron Dataset Results

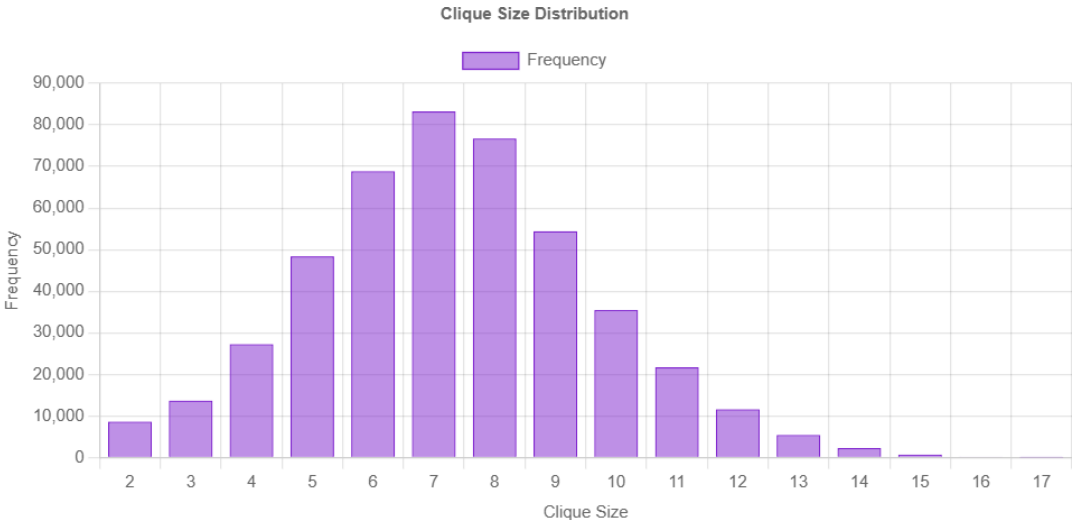

Fig: Wiki-vote Dataset Result

Optimised TTT Results

Wiki-Vote Dataset

```
Largest size of the clique: 17
Total number of maximal cliques: 459002
Execution time (ms): 1483
Distribution of different size cliques:
Size 2: 8655
Size 3: 13718
Size 4: 27292
Size 5: 48416
Size 6: 68872
Size 7: 83266
Size 8: 76732
Size 9: 54456
Size 10: 35470
Size 11: 21736
Size 12: 11640
Size 13: 5449
Size 14: 2329
Size 15: 740
Size 16: 208
Size 17: 23
```

Email-Enron Dataset

```
Largest size of the clique: 20
Total number of maximal cliques: 226859
Execution time (ms): 7890
Distribution of different size cliques:
Size 2: 14070
Size 3: 7077
Size 4: 13319
Size 5: 18143
Size 6: 22715
Size 7: 25896
Size 8: 24766
Size 9: 22884
Size 10: 21393
Size 11: 17833
Size 12: 15181
Size 13: 11487
Size 14: 7417
Size 15: 3157
Size 16: 1178
Size 17: 286
```

```
Size 18: 41
Size 19: 10
Size 20: 6
```

As-Skitter Dataset

```
Largest size of the clique: 67
Total number of maximal cliques: 37322355
Execution time (ms): 22136003
Distribution of different size cliques:
Size 2: 2319807
Size 3: 3171609
Size 4: 1823321
Size 5: 939336
Size 6: 684873
Size 7: 598284
Size 8: 588889
Size 9: 608937
Size 10: 665661
Size 11: 728098
Size 12: 798073
Size 13: 877282
Size 14: 945194
Size 15: 980831
Size 16: 939987
Size 17: 839330
Size 18: 729601
Size 19: 639413
Size 20: 600192
Size 21: 611976
Size 22: 640890
Size 23: 673924
Size 24: 706753
Size 25: 753633
Size 26: 818353
Size 27: 892719
Size 28: 955212
Size 29: 999860
Size 30: 1034106
Size 31: 1055653
Size 32: 1017560
Size 33: 946717
Size 34: 878552
Size 35: 809485
Size 36: 744634
Size 37: 663650
Size 38: 583922
Size 39: 520239
Size 40: 474301
Size 41: 420796
Size 42: 367879
Size 43: 321829
Size 44: 275995
Size 45: 222461
Size 46: 158352
Size 47: 99522
Size 48: 62437
Size 49: 39822
Size 50: 30011
Size 51: 25637
Size 52: 17707
Size 53: 9514
Size 54: 3737
Size 55: 2042
Size 56: 1080
Size 57: 546
Size 58: 449
Size 59: 447
Size 60: 405
Size 61: 283
Size 62: 242
Size 63: 146
Size 64: 84
Size 65: 49
Size 66: 22
Size 67: 4
```

Chiba and Nishizeki (1985)

Background and Theory

This paper addresses fundamental limitations in existing subgraph listing algorithms by introducing a novel edge-searching strategy that significantly improves efficiency. Traditional methods for subgraph enumeration, such as triangle and clique listing, often rely on brute-force approaches or adjacency matrix representations, leading to high time and space complexity. Chiba and Nishizeki's work leverages the concept of **arboricity**—a measure of a graph's edge density—to refine subgraph enumeration algorithms, achieving significant theoretical and practical advancements.

Key Challenges in Prior Approaches:

1. **High Computational Complexity:** Many earlier algorithms for triangle, quadrangle, and clique listing required time complexities dependent on the total number of vertices (n), which could lead to suboptimal performance.
2. **Inefficient Graph Representation:** Methods that used adjacency matrices consumed $\mathbf{O}(n^2)$ space, making them impractical for large graphs.
3. **Redundant Processing:** Prior techniques often revisited edges and vertices multiple times, leading to unnecessary computations.

Chiba & Nishizeki's Key Contributions:

1. **Arboricity-Based Complexity Reduction:**
   - Instead of general $\mathbf{O}(n^2)$ bounds, the paper introduces algorithms parameterized by **arboricity** ($\alpha(G)$), which is much smaller than n for many sparse graphs.
   - The paper establishes an upper bound for arboricity: $\alpha(G) \leq \sqrt{(2m + n)}/2$, meaning that for many real-world graphs, these algorithms run in nearly linear time.
2. **Optimized Triangle and Quadrangle Listing:**
   - The proposed **triangle listing algorithm** runs in $\mathbf{O}(\alpha(G)m)$ time, which is significantly faster than the naive $\mathbf{O}(m^{3/2})$ approach.
   - Similarly, quadrangles (4-cycles) are identified in $\mathbf{O}(\alpha(G)m)$ time using an efficient set-based representation.
3. **Efficient Clique Enumeration:**
   - The authors extend their approach to k**-clique enumeration**, achieving $\mathbf{O}(\alpha(G)^{(k-2)}m)$ complexity, which is optimal given known worst-case clique counts.
   - The **clique listing algorithm** also ensures that each maximal clique is discovered once, avoiding redundant computation.

4. **Minimal Space Usage:**
    - Unlike adjacency-matrix-based methods requiring **O(n^2)** space, the algorithms introduced in this paper use only **O(m)** space, making them highly practical.

Significance and Impact:

This work provides a **unified and efficient approach** to subgraph listing by leveraging arboricity as a guiding metric. By reducing complexity from $O(n^2)$ to $O(\alpha(G)m)$ in key algorithms, the authors significantly improve computational feasibility, especially for **sparse graphs** (such as social networks, citation graphs, and planar graphs). These optimizations make Chiba and Nishizeki's algorithms foundational for modern clique detection, network analysis, and large-scale graph processing.

In summary, this paper revolutionized subgraph enumeration by introducing a **low-overhead edge-searching technique**, bounding its complexity with **arboricity**, and achieving near-optimal efficiency in triangle, quadrangle, and clique listing—laying the groundwork for numerous later advancements in graph algorithms.

Algorithm

Chiba and Nishizeki's algorithm efficiently enumerates maximal cliques in a graph using a recursive backtracking method with adjacency-based pruning. The algorithm leverages the following key concepts:

Key Definitions

1. **Adjacency List (Γ(x))**: The set of vertices adjacent to a vertex x in the graph G.
2. **Candidate Set (C)**: The set of vertices that can be part of a clique at any given stage.
3. **Maximality Test**: Ensures that a found clique is not a subset of another larger clique.
4. **Lexicographic Order Test**: Guarantees that cliques are output in a sorted order.
5. **Pruning Techniques**: Used to reduce the search space efficiently.

Algorithm Overview

- The algorithm explores potential cliques recursively by adding vertices one at a time.
- It maintains an ordered set C to track the current clique and a candidate set for possible expansions.
- At each step, the algorithm:
    1. Computes **intersection sets** to determine valid clique extensions.
    2. Applies **pruning heuristics** to reduce redundant expansions.
    3. Checks **maximality conditions** to avoid generating non-maximal cliques.
    4. Ensures **lexicographic order**, maintaining a systematic traversal of the search space.

- When no further expansion is possible, the current set C is reported as a maximal clique.

Strategies Utilised

- **Degree Sorting**: The vertices are processed in increasing order of degree to minimize the branching factor.
- **Set Intersection Computation**: Instead of checking all neighbors, the algorithm uses efficient set operations to find common neighbors.
- **Recursive Pruning**: If a vertex is found that covers all possible extensions, it is skipped.

Code Implementation

CaN Implementation

```
#include <iostream>
#include <fstream>
#include <vector>
#include <unordered_set>
#include <algorithm>
#include <chrono>
#include <sstream>
#include <cstdlib>
#include <cstring>
#include <sys/resource.h>
#include <utility>
#include <set>
using namespace std;
using namespace chrono;

static vector<int> T_, S_;
static vector<int> cliqueSizeCount;
static int numCliques = 0;
static int maxCliqueSize = 0;
static vector<unordered_set<int>> adj;
static int n = 0;
static bool FLAG = true;

static inline void ensureCliqueSize(int sz) {
   if (sz > maxCliqueSize)
       maxCliqueSize = sz;
   if ((int)cliqueSizeCount.size() <= sz)
       cliqueSizeCount.resize(sz + 1, 0);
}

static inline int getMax(const unordered_set<int>& C) {
   int mx = -1;
   for (int x : C)
       if (x > mx)
           mx = x;
   return mx;
}

static inline void recordClique(const unordered_set<int>& C) {
   vector<int> cliqueVec;
   cliqueVec.reserve(C.size());
   for (int x : C)
       cliqueVec.push_back(x);
   sort(cliqueVec.begin(), cliqueVec.end());
   int sz = static_cast<int>(cliqueVec.size());
   if (sz < 2)
       return;
   ensureCliqueSize(sz);

   ++cliqueSizeCount[sz];
   ++numCliques;
   cout << "[DEBUG] Total cliques so far: " << numCliques << endl;
}
```

```
static void UPDATE(int i, unordered_set<int>& C) {
   if (!C.empty()) {
       int m = getMax(C);
       if (i <= m) {
           UPDATE(m + 1, C);
           return;
       }
   }

   if (i == n) {
       recordClique(C);
       return;
   }

   const auto& neighborsOfI = adj[i];
   unordered_set<int> diff;
   diff.reserve(C.size());
   for (int x : C) {
       if (!neighborsOfI.count(x))
           diff.insert(x);
   }

   if (!diff.empty())
       UPDATE(i + 1, C);

   unordered_set<int> cap;
   cap.reserve(C.size());
   for (int x : C) {
       if (neighborsOfI.count(x))
           cap.insert(x);
   }

   for (int x : cap) {
       for (int y : adj[x]) {
           if ((y != i) && !C.count(y))
               ++T_[y];
       }
   }
   for (int x : diff) {
       for (int y : adj[x]) {
           if (!C.count(y))
               ++S_[y];
       }
   }

   FLAG = true;
   const int capSize = static_cast<int>(cap.size());

   for (int y : neighborsOfI) {
       if (!C.count(y) && (y < i) && (T_[y] == capSize)) {
           FLAG = false;
           break;
       }
   }

   vector<int> dv;
   dv.reserve(diff.size());
   for (int x : diff)
       dv.push_back(x);
   sort(dv.begin(), dv.end());

   const int p_int = static_cast<int>(dv.size());
   for (int k = 0; k < p_int && FLAG; k++) {
       int j_k = dv[k];
       for (int y : adj[j_k]) {
           if ((y < i) && !C.count(y) && (T_[y] == capSize)) {
               if (y >= j_k) {
                   S_[y]--;
               } else {
                   if (k == 0 || y >= dv[k - 1]) {
                       if ((S_[y] + k) == p_int) {
                           FLAG = false;
                           break;
                       }
                   }
               }
           }
       }
   }
```

```
    }

    int jp = dv.empty() ? 0 : dv.back();
    if (!cap.empty()) {
        for (int y = 0; y < i && FLAG; y++) {
            if (!C.count(y) && (T_[y] == capSize) && (S_[y] == 0)) {
                if (jp < y) {
                    FLAG = false;
                    break;
                }
            }
        }
    } else {
        if (jp < (i - 1))
            FLAG = false;
    }

    for (int x : cap) {
        for (int y : adj[x]) {
            if ((y != i) && !C.count(y))
                T_[y] = 0;
        }
    }
    for (int x : diff) {
        for (int y : adj[x]) {
            if (!C.count(y))
                S_[y] = 0;
        }
    }

    if (FLAG) {
        auto diffSave = std::move(diff);
        cap.insert(i);
        auto oldC = std::move(C);
        C = std::move(cap);
        UPDATE(i + 1, C);
        C = std::move(oldC);
        for (int val : diffSave)
            C.insert(val);
    }
}

int main(int argc, char* argv[]) {
    struct rlimit rl;
    rlim_t stack_size = 512 * 1024 * 1024;
    int result = getrlimit(RLIMIT_STACK, &rl);
    if (result != 0) {
        cerr << "Error getting stack limit: " << strerror(errno) << endl;
        return 1;
    }
    if (rl.rlim_cur < stack_size) {
        rl.rlim_cur = stack_size;
        if (rl.rlim_max < rl.rlim_cur)
            rl.rlim_max = rl.rlim_cur;
        result = setrlimit(RLIMIT_STACK, &rl);
        if (result != 0)
            cerr << "Error setting stack limit: " << strerror(errno) << endl;
        else
            cerr << "Stack size increased to " << (stack_size / (1024 * 1024)) << " MB" << endl;
    }

    ios::sync_with_stdio(false);
    cin.tie(nullptr);

    if (argc < 2) {
        cerr << "Usage: " << argv[0] << " [input file]" << "\n";
        return 1;
    }

    ifstream infile(argv[1]);
    if (!infile) {
        cerr << "Error: Unable to open input file " << argv[1] << "\n";
        return 1;
    }
    int u, v;
    int maxVertex = -1;
    vector<pair<int,int>> edges;
    edges.reserve(200000);
```

```
    string line;
    while (getline(infile, line)) {
        if (line.empty() || line[0] == '#')
            continue;
        istringstream iss(line);
        if (iss >> u >> v) {
            maxVertex = max(maxVertex, max(u, v));
            edges.emplace_back(u, v);
        }
    }
    infile.close();
    n = maxVertex + 1;
    adj.clear();
    adj.resize(n);

    for (auto &p : edges) {
        int a = p.first, b = p.second;
        if (a != b && a >= 0 && a < n && b >= 0 && b < n) {
            adj[a].insert(b);
            adj[b].insert(a);
        }
    }

    T_.assign(n, 0);
    S_.assign(n, 0);

    vector<pair<int,int>> degreeIndex;
degreeIndex.reserve(n);
for (int i = 0; i < n; i++) {
    degreeIndex.push_back({static_cast<int>(adj[i].size()), i});
}

sort(degreeIndex.begin(), degreeIndex.end());
vector<int> oldToNew(n), newToOld(n);
for (int newIndex = 0; newIndex < n; newIndex++) {
    int oldIndex = degreeIndex[newIndex].second;
    oldToNew[oldIndex] = newIndex;
    newToOld[newIndex] = oldIndex;
}
vector<unordered_set<int>> newAdj(n);
for (int i = 0; i < n; i++) {
    int new_i = oldToNew[i];
    for (int neighbor : adj[i]) {
        int newNeighbor = oldToNew[neighbor];
        if (new_i != newNeighbor)
            newAdj[new_i].insert(newNeighbor);
    }
}
adj = std::move(newAdj);
int stidx = 0;
while(stidx < degreeIndex.size() && degreeIndex[stidx].first == 0) {
    stidx++;
}
    unordered_set<int> C;
    C.insert(0);

    auto startTime = high_resolution_clock::now();
    UPDATE(1, C);
    auto stopTime = high_resolution_clock::now();
    auto duration = duration_cast<milliseconds>(stopTime - startTime).count();
    ofstream outfile("output.txt");
    if (!outfile) {
        cerr << "Error: Unable to open output.txt for writing\n";
        return 1;
    }
    outfile << "Largest size of a clique: " << maxCliqueSize << "\n";
    outfile << "Total number of maximal cliques: " << numCliques << "\n";
    outfile << "Execution time (ms): " << duration << "\n";
    outfile << "Distribution of clique sizes:\n";
    for (int i = 2; i <= maxCliqueSize; ++i) {
        int countVal = (i < (int)cliqueSizeCount.size()) ? cliqueSizeCount[i] : 0;
        outfile << "Size " << i << ": " << countVal << "\n";
    }
    outfile.close();
    return 0;
}
```

Walkthrough for the Code Implementation

1. Graph Representation and Utility Functions

- Global Variables and Data Structures
  - The graph is stored in a global variable as a vector of unordered sets (named adj), where each index represents a vertex and its set contains its adjacent vertices. Several auxiliary vectors and counters are defined:
  - T_ and S_: Vectors used to maintain counts for auxiliary conditions during recursion.
  - cliqueSizeCount: A vector that records the number of cliques found for each clique size.
  - numCliques and maxCliqueSize: Counters that track the total number of maximal cliques encountered and the size of the largest clique, respectively.
  - n: Stores the total number of vertices, and FLAG: A boolean flag used to control recursion based on pruning conditions.
- Utility Functions
  - ensureCliqueSize(sz):
    This inline function checks if the current clique size (sz) exceeds the recorded maxCliqueSize. If so, it updates maxCliqueSize and ensures that the cliqueSizeCount vector is large enough to count cliques of that size.
  - getMax(C): Traverses an unordered set C and returns its maximum element. This is used to impose an ordering on the recursive search, ensuring that the candidate set contains only vertices that are greater than those already included.
  - recordClique(C) : When a clique is identified, this function first converts the unordered set *C* into a vector, sorts it (for consistency), and then records the clique only if its size is at least 2. It updates the counters in cliqueSizeCount and numCliques and prints a debug message indicating the current total count of cliques.

2. The UPDATE Function: Recursive Clique Enumeration

- Core Recursion and Pruning
  The recursive function UPDATE(i, C) is the heart of the clique enumeration process. Its parameters are:
  - *i*: The current vertex index under consideration.
  - *C*: A candidate set representing the current clique.
- The function follows these major steps:
  - Pruning Based on Ordering : If *C* is not empty, the function first determines the maximum element *m* in *C*. If *i* is not greater than *m*, it calls UPDATE(m + 1, C) to skip over vertices that would violate the ordering constraint, effectively pruning redundant branches.

- Base Case : When i equals the total number of vertices n, the current set C forms a maximal clique and is recorded via recordClique(C).
- Candidate Set Adjustments : The function then retrieves the set of neighbors for the current vertex i and computes a difference set (diff) containing vertices in C that are not neighbors of i. If this difference set is non-empty, a recursive call with i+1 is made using the original candidate set.
- Computing the Intersection (cap) : A new set cap is built by collecting all vertices in C that are neighbors of i. This set represents the potential extension of the current clique.
- Updating Auxiliary Counters : For every x in cap and every x's neighbor y that is not already in C, the counter T[y] is incremented to reflect connectivity with the current clique. A similar update is done with S_ for vertices in the difference set. These counters are later used to decide if extending C further is viable.
- Flag Check and Pruning Conditions : Several loops inspect the auxiliary counters along with neighbor relations to set the boolean FLAG. If any condition fails (for instance, if certain counter values indicate that adding more vertices would lead to a duplicate exploration of a clique), FLAG is set to false to prune that branch. Apart from this, the algorithm performs maximal and lexicographical tests on all the cliques obtained to record the maximal cliques.
- Recursive Expansion : If FLAG remains true, the algorithm saves the current state of the difference set, adds the current vertex i to the candidate set, and recursively calls UPDATE(i + 1, C). After the recursive call, it restores the original candidate set, ensuring that the search space is correctly maintained when backtracking.

3. Vertex Reordering for Efficient Enumeration

Before initiating the recursion, the code performs an important vertex reordering step:

- Degree-Based Sorting:
  A vector of pairs, degreeIndex, is created where each pair consists of the degree of a vertex (the number of neighbors) and the vertex index. This vector is sorted in increasing order.
- Creating Mapping Arrays:
  Two mapping vectors (oldToNew and newToOld) are generated to translate between the original vertex order and the new ordering based on degree. Lower-degree vertices tend to be processed first, which can often reduce the search space and improve performance.
- Rebuilding the Adjacency List:
  Using the new ordering, a new adjacency list is constructed. Each vertex's neighbors are remapped into the new indexing, thereby updating the global adj variable. This

step is critical for ensuring that subsequent recursive calls operate on the optimally ordered graph.

4. **Main Function: Input, Processing, and Performance Measurement**

- Stack Limit Adjustment : The program begins by adjusting the process's stack size using getrlimit and setrlimit. This is essential for supporting deep recursion, particularly with large input graphs.
- Graph Construction : The main function reads an input file line by line, ignoring empty lines and comments. For each valid edge, it collects the edge pairs into an array and simultaneously determines the highest vertex index.
  Next, it initializes the adjacency list adj to have the appropriate size and populates it by inserting each edge in both directions (ensuring an undirected graph).
- Applying Vertex Reordering : After constructing the basic graph, the code applies the degree-based reordering as described above. This step improves the efficiency of the recursive search.
- Initiating the Recursive Search : The recursion is started by initializing a candidate set *C* with the first vertex (after reordering) and calling UPDATE(1, C). Execution time is recorded using the chrono library to measure performance.
- Output Generation : Once the recursive exploration completes, the program writes the results to an output file (output.txt). It reports:
  - The largest clique size found.
  - The total number of maximal cliques detected.
  - The execution time (in milliseconds).
  - A detailed distribution of the number of cliques for each clique size.

In summary, this code leverages a recursive backtracking strategy with aggressive pruning and vertex reordering to enumerate maximal cliques in an undirected graph. It includes several utility functions to manage clique size recording and candidate set updates, while also ensuring that deep recursion is supported through increased stack limits. The careful ordering and update of auxiliary counters contribute to the efficiency of the clique detection process.

Optimisations Performed

1. **Stack Limit Increase:**

- The code raises the stack size (via setrlimit) to prevent stack overflows for deeper recursion.

2. **Use of Static and Inline Functions & Variables :**

- The code utilizes inline and static functions and variables to optimize performance and improve code organization. By using inline functions for small, frequently called operations like ensureCliqueSize and getMax, the code reduces function call overhead

and allows for better compiler optimizations. Static variables and functions, such as T_, S_, and UPDATE, help maintain state between recursive calls without resorting to global variables, improving memory efficiency and reducing naming conflicts. These techniques contribute to faster execution, especially in the context of the recursive clique enumeration algorithm, where even small performance gains can significantly impact overall runtime due to the potentially large number of function calls and state updates involved in exploring the graph structure..

3. **Reserve and Move Semantics :**

- The code uses reserve for storing degree index and edge. And utilizes std::move for in recursive calls, reducing allocations and copies.

4. **Choice of Data Structures** :

- The code uses vectors instead of unordered_set, which can give constant-time membership checks but may incur overhead from hashing—especially in intersections or difference operations where you must loop through the entire set.

NOTE :

1) Our implementation of CaN algorithm was not able to produce output for the as_skitter dataset, due to stack overflow, since CaN uses brute force method to find all possible cliques through recursion and then prune it for obtaining maximal cliques.
2) An isolated vertex is by definition a clique, but in the ELS paper it has been mentioned that in practical use-cases such as MCE(Maximal Clique Enumeration), it is only of practical use to check for maximal cliques of size >=2.

Results Obtained

CaN Results

Wiki-Vote Dataset

```
Largest size of the clique: 17
Total number of maximal cliques: 459002
Execution time (ms): 264885
Distribution of different size cliques:
Size 2: 8655
Size 3: 13718
Size 4: 27292
Size 5: 48416
Size 6: 68872
Size 7: 83266
Size 8: 76732
Size 9: 54456
Size 10: 35470
Size 11: 21736
Size 12: 11640
Size 13: 5449
Size 14: 2329
Size 15: 740
Size 16: 208
Size 17: 23
```

Email-Enron Dataset

```
Largest size of the clique: 20
Total number of maximal cliques: 226859
Execution time (ms): 39666
Distribution of different size cliques:
Size 2: 14070
Size 3: 7077
Size 4: 13319
Size 5: 18143
Size 6: 22715
Size 7: 25896
Size 8: 24766
Size 9: 22884
Size 10: 21393
Size 11: 17833
Size 12: 15181
Size 13: 11487
Size 14: 7417
Size 15: 3157
Size 16: 1178
Size 17: 286
Size 18: 41
Size 19: 10
Size 20: 6
```

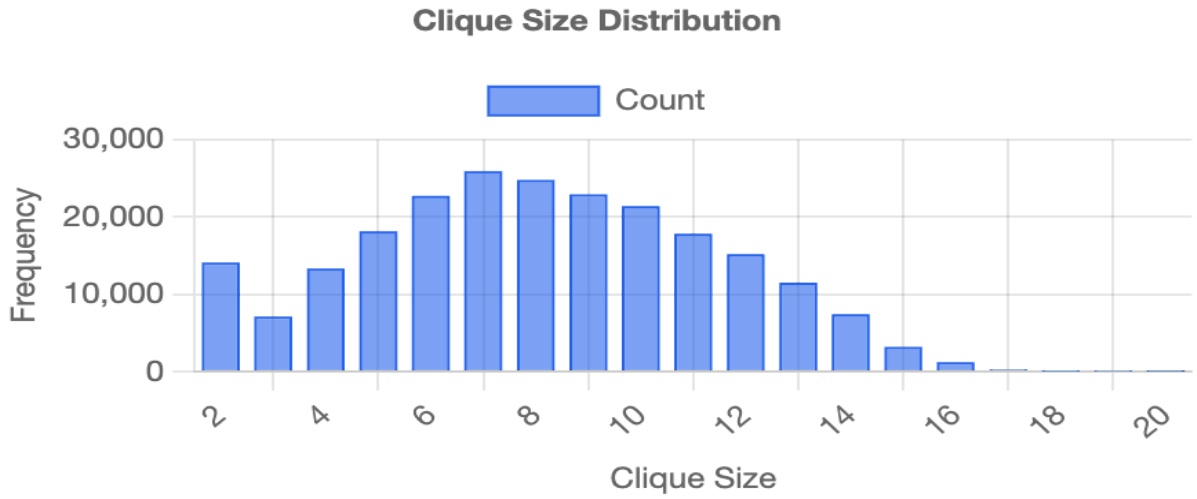

Fig: Email-Enron Dataset Results

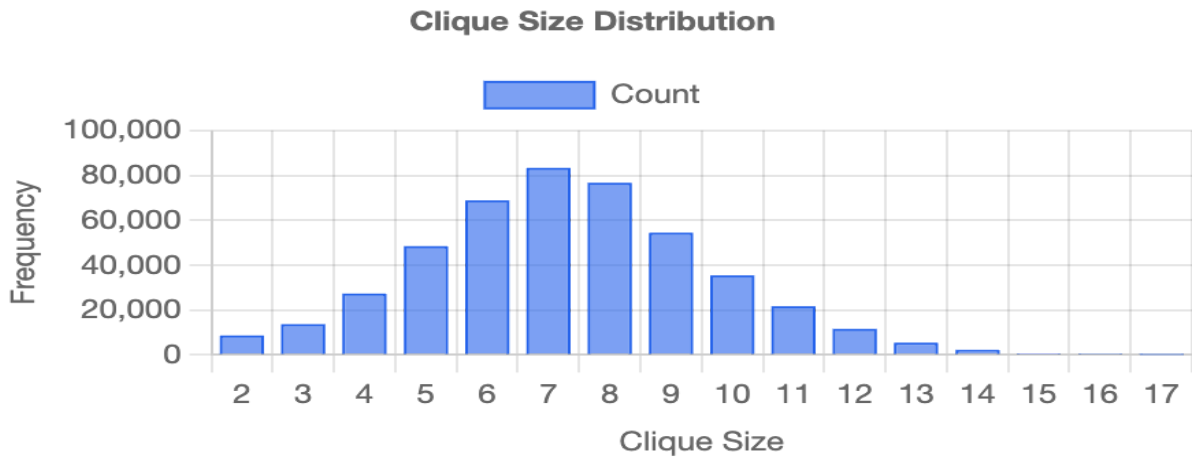

Fig: Wiki-vote Dataset Result

Compare and Contrast

A Comparison of all three Maximal Clique Enumeration Algorithms.

1. Chiba & Nishizeki(1985)
    - They introduced an edge-searching strategy that leverages the graph's arboricity.
    - For graphs with low arboricity(planar graphs) a single clique can be enumerated in linear time.
    - There is no explicit pivoting strategy, the algorithm chooses to "pick" and "not pick" a vertex based on some tests.
    - While conceptually simple, we in our computational experiments have found that the maximality test and lexico-test to ensure maximality and uniqueness of the cliques is a naive approach and increases the time taken to converge.

2. Tomita, Tanaka and Takahashi(2006)
    - This algorithm is a refined version of the classic Bron–Kerbosch procedure that uses pivoting to prune the search tree.
    - It achieves worst-case optimal performance with a time complexity of O(3^(n/3)) for an n-vertex graph, matching the known upper bound on the number of maximal cliques.
    - Its strength lies in simplicity and proof rigour to demonstrate worst-case time complexity.

3. Eppstein, Löffler & Strash (2010)
    - This version of the Bron-Kerbosh algorithm is parametrized by the graph's degeneracy(a measure of sparsity).
    - It runs in O(d . n . 3^(d/3)) time, where d is the degeneracy of the graph, n is the number of vertices.
    - This algorithm is particularly useful in sparse graphs(like in social networks).
    - However, if the graph is dense (high degeneracy), the benefits diminish, and performance can degrade toward the worst-case exponential behavior.

In summary, while Chiba & Nishizeki approach laid important groundwork for subgraph enumeration using arboricity, Tomita et al. 's method pushes the envelope with worst-case optimality through effective pruning. Meanwhile, Eppstein et al.'s algorithm adapts well to the practical realities of sparse graphs by using degeneracy as a key parameter, often resulting in faster performance on real-world networks.

Key observations

A note on our key observations in all the algorithms.

- A complete graph has only one maximal clique.
- Thus, if a graph has n complete connected components, it will have n maximal cliques.
- A singly connected graph can only have multiple cliques if it is not complete because then we would have proper subsets where we cannot expand the subset anymore due to lack of connecting edges.
- The time complexity to enumerate maximal cliques has been found to be upper-bounded by O(3^(n/3)) as mentioned and proved in the Tomita-Takahashi-Tanaka paper.
- The problem of listing maximal cliques is an NP-hard problem.
- The clique decision problem - “Does the graph contain a clique of size k?” is known to be an NP-complete problem.
- Now, if we enumerate all possible maximal cliques, we can also answer the clique decision problem. This shows that the MCE problem is at least as hard as the clique decision problem.
- Solving MCE would also solve the clique decision problem and hence MCE is in fact an NP-hard problem.
- A class of graphs is said to have few cliques if every member of the class has a polynomial number of maximal cliques(in the order of number of vertices). Naturally, such graphs would be of interest in network theory, etc.
- There is also a class of graphs that has exponential number of maximal cliques.
- One such example is Turan graph or Moon-Moser graph as mentioned in Tomita-Tanaka-Takahashi paper. These graphs have exactly 3^(n/3) maximal cliques when n mod 3 = 0 which in fact is the upper bound to maximal clique enumeration too.

Use-cases of Maximal Clique Enumeration

Our suggestions for possible applications of Maximal Clique Enumeration in the real world.

- Finding number of friend circles in a sparse graph
  - Our graph would be represented by G = (V,E) where V represents all the people in the network and every e = (u,v) in E represents “friendship” between two people.
  - Our goal is to find the number of “friend circles” in this network.
  - Friend circle - A friend circle will be defined as a group of people where everyone is a friend to everyone and this group should be as large as possible.
  - This actually translates to :
    Group of people - subgraph of vertices
    Everyone is a friend to everyone - complete subgraph
    Group should be as large as possible - maximal complete subgraph
    Find number of such friend groups - maximal complete subgraph enumeration also known as maximal clique enumeration

- Effective team formation in organisations
  - In an organisation, everyone is specialised in something or the other and has a tandem with a group of people.
  - The goal of an organisation would be to find the largest group of people with the property that everyone has a working rapport with everyone else in the group i.e., a maximal clique.
  - By forming such groups, an organisation can reduce the overhead of managing multiple redundant teams (non-maximal cliques) and still efficiently perform the deliverables.

- Protein complexes in biological networks
  - Vertices: Proteins, Edges: Represent physical interactions between proteins.
  - A clique here represents a group of proteins that interact with each other.
  - A maximal clique here represents a protein complex with the property that no other protein can be added to this complex that can interact with every protein in this complex.
  - It is of great biological significance to find the number of protein complexes in a biological network.

THANK YOU