CSE 462
Report on Presentations
Clique Cover Problem

Group 10

1505002
1505044
1505057
1505097
1505101

Department of Computer Science and Engineering
Bangladesh University of Engineering and Technology
(BUET)
Dhaka 1000

December 18, 2020

Contents

List of Figures

List of Tables

1 Introduction

1.1 Complete Graph

A complete graph is a simple undirected graph in which every pair of distinct vertices is connected by a unique edge.

1.2 Clique

A clique of a graph $G(V, E)$ is a complete subgraph of G.

1.3 Clique Cover Problem

Input: An undirected graph $G(V, E)$ and an integer K.

Output: True if the vertices of G can be partitioned into K sets S_i ,whenever two vertices in the same sets Si are adjacent. S_i do not need to be disjoint, they can be non disjoint. But we can make them disjoint by putting common vertices in only one set without any problem. Thus we can think S_i are disjoint.

Note: There is also edge clique cover problem but we are only interested in vertex clique cover. So if we say clique cover, we are indicating vertex clique cover.

1.4 Applications

- DNA molecular solution problem, data partitioning problem in embedded processor-based systems (memory chips), image processing problems etc.
- Applications of the vertex clique cover problem arise in network security, scheduling and VLSI design.
- Algorithms for clique cover can also be used to solve the closely related problem of finding a maximum clique, which has a range of applications in biology, such as identifying related protein sequences.

1.5 Hardness Status of Clique Cover Problem

The clique cover problem in computational complexity theory is the algorithmic problem of finding a minimum clique cover, or (rephrased as a decision problem) finding a clique cover whose number of cliques is below a given threshold. Finding a minimum clique cover is **NP-hard**, and its decision version is **NP-complete**. It was one of Richard Karp's original 21 problems shown **NP-complete** in his **1972** paper **"Reducibility Among Combinatorial Problems"**.

The equivalence between clique covers and coloring is a reduction that can be used to prove the **NP-completeness** of the *clique cover problem* from the known **NP-completeness** of *graph coloring*.

2 Proof of NP-completeness

To prove that clique cover is **NP-complete**, first we prove that it is in **NP**.

We are given a graph $G(V, E)$ and the clique cover of the graph of size k that is k subsets of V. We first check whether each such subset form a complete graph that is any two vertices of a set are neighbors. If not the answer is no. Otherwise we again check if union of all the sets is equal to V. If not the answer is no otherwise yes.

Clearly, this takes at most $O(n^2)$ where n is the number of vertices . So, the clique cover problem is clearly in **NP**.

The next thing we do is show that it is **NP-hard**. To do so, we show that the *clique cover problem* is at least as hard as the *k-coloring problem.*

$$\text{K-coloring} \leq_p \text{Clique Cover}$$

If a graph is *k-colorable*, then it can be partitioned into k independent sets (one for each color class).Then we just exploit the normal reduction between *Independent Set* and *Clique Cover* by taking the complement graph (we swap edges for non-edges and vice versa), so any independent set becomes a clique.

$$\text{K-coloring} \leq_p \text{K-Independent Set}$$
$$\text{K-Independent Set} \leq_p \text{Clique Cover}$$

So if $\overline{G(V,E)}$ is *k-colorable*, it can be partitioned into k independent sets. Hence $\overline{G(V,E)}$ (complement of $G(V,E)$ can be partitioned into (i.e. covered by) k cliques. Conversely if $\overline{G(V,E)}$ can be covered by k cliques, $G(V,E)$ has a partition into k *independent sets*, and hence is *k-colorable.* And so the Clique Cover problem is NP-complete.

3 Clique Cover and It's Variations

3.1 Clique Cover Decision Problem

Given a graph $G(V, E)$ and a number K, we need to answer yes or no if we can partition $V(G)$ in K cliques.

3.2 Minimum Clique Cover Problem

Given a graph $G(V, E)$, we need to output the smallest number for which clique cover exists. This number is called the clique cover number.

Both of these problem are equivalent. That means if we solve one of the problems we can easily construct a solution for the other problem in polynomial time from that solution.

4 Algorithms

4.1 Brute Force Algorithm

Below is a brute force algorithm for finding clique covers of a graph G.

Step 1: Find the power set of V where V is the set of vertices of graph G.

Step 2: Take an empty list S.

Step 3: Take every element(which is actually a subset of V) of $P(V)$ and check whether it's a complete subgraph or clique then add this to list S otherwise skip the element.

Step 4: Find all possible subsets of S of size k.

Step 5: Take every subsets and check whether it covers all vertices of V or not.

Step 1,2 and 3 runs in $O(n^2 2^n)$
Step 4 runs in $O(2^{kn})$
Step 5 runs in $O(n 2^{kn})$
So the overall time complexity of brute force algorithm is $O(n 2^{kn})$

4.2 Exponential Time Exact Algorithm

- For the *edge clique cover* problem there exist an exact exponential algorithm of complexity $O^*(2^n)$.
- However for *vertex clique cover*, **"Set Partitioning via Inclusion-Exclusion"** paper is more generalization of the problem of Vertex Clique Cover. So, we can use their idea to find an exact exponential algorithm of complexity $O^*(2^n)$.
- The dual problem of *vertex clique cover* is *K-Coloring Problem*. When it is parameterized by number of colors, it is **para-NP-hard**. But when parameterized by vertex cover or treewidth it is in **FPT**. So same applies for *vertex clique cover*.

4.3 Heuristic - Metaheuristic Algorithm

4.3.1 Greedy Clique Covering (GCC)

Input: graph $G = [V, E]$
permutation $P = [P_1, P_2, ..., P_n]$ of vertices in V
Output: clique covering S of G
Algorithm:

for $c \leftarrow 1...n$ **do**
  $sizes(c) \leftarrow 0$
end for
for $c \leftarrow 1...n$ **do**
  $j \leftarrow P_i$
  $c \leftarrow find\_equal(\Gamma(v_j, c), sizes(c))$

$V_c \leftarrow V_c \bigcup \{v_j\}$
end for
return $S \leftarrow V_1, V_2, ..., V_k$

4.3.2 Iterative Greedy Clique Covering (IG)

Input: graph $G = [V, E]$
Output: clique covering S of G
Algorithm:
$P \leftarrow random\_permutation(1, 2, ..., n)$
while stopping criterion not met **do**
$V_1, V_2, ..., V_k \leftarrow GCC(G, P)$
if p_{prev} **then**
$P \leftarrow [V_k, V_{k-1}, ..., V_1]$
else
$P \leftarrow random\_permutation(V_1, V_2, ..., V_k)$
end if
if $\nu(G)$ is known & $k = \nu(G)$ **then**
return $S \leftarrow V_1, V_2, ..., V_k$
end if
end while
return $S \leftarrow V_1, V_2, ..., V_k$

5 Implementation

5.1 Exact Implementation

```
#include<vector>
#include<algorithm>
#include<iostream>
#include<fstream>
#include<cstring>
#include<bitset>
#include <chrono>

using namespace std;

//0 - based
const int MAX_VERTEX = 30, MOD1 = 1e9+7, MOD2 = 1e9+9;

void find_clique( int vertex, int cliques[], int *edge_mat )
{
    int tot = 1<<vertex;

    for( int flag = 1; flag < tot; flag++ )
```

```
    {
        cliques[flag] = 1;
        for( int i = 0; i < vertex and cliques[flag] == 1; i++ )
        {
            for( int j = 0; j < vertex; j++ )
            {
                if( i != j and (flag&(1<<i)) != 0  and (flag&(1<<j)) != 0 and ed
                {
                    cliques[flag] = 0;
                    break;
                }
            }
        }
    }
    return;
}

void sos_dp(int vertex, int cliques[])
{
    int tot = 1<<vertex;

    for( int i = 0; i < vertex; i++ )
    {
        for( int mask = 0; mask < tot; mask++ )
        {
            if( mask&(1<<i) )
            {
                cliques[mask] += cliques[mask^(1<<i)];
            }
        }
    }

    return;
}

//a^b
int expo( int a, int b, int mod )
{
    int val = 1;
    while(b > 0)
    {
        if( b&1 )
            val = (1LL*val*a)%mod;
        a = (1LL*a*a)%mod;
        b /= 2;
    }
```

```
    return val;
}

bool KCliqueSolvable(int vertex, int zetaClique[], int k)
{
    long long ans1 = 0, ans2 = 0, tot = (1<<vertex)-1;

    for( int i = 1; i < tot; i++ )
    {
        if( zetaClique[i] == 0 ) continue;

        int bit = vertex - __builtin_popcount(i);

        if( bit&1 )
        {
            ans1 += expo( zetaClique[i], k, MOD1 );
            ans2 += expo( zetaClique[i], k, MOD2 );
        }
        else
        {
            ans1 -= expo( zetaClique[i], k, MOD1 );
            ans2 -= expo( zetaClique[i], k, MOD2 );
        }
        ans1 %= MOD1;
        ans2 %= MOD2;
    }

    int sum = zetaClique[tot];
    long long rem1 = (expo(sum, k, MOD1) - ans1)%MOD1, rem2 = (expo(sum, k, MOD2

    return rem1 != 0 and rem2 != 0;

}

int cliqueCoverNumber(int vertex, int zetaClique[])
{
    int lo = 0, hi = vertex;
    while(lo < hi)
    {
        int m = (lo+hi)/2;
        int f = KCliqueSolvable(vertex, zetaClique, m);
        if(f)
        {
            hi = m;
        }
        else
```

```
        {
            lo = m+1;
        }
    }
    return hi;
}

int main()
{
    ifstream fin;
    //input file: vertex number start from 0
    //vertex edge
    //each edge in one line
    fin.open("in0.txt");

    ofstream fout;
    fout.open("out_exact0.txt");


    auto start = chrono::high_resolution_clock::now();

    int vertex, edge;

    fin >> vertex >> edge;

    int tot = (1<<vertex);

    int *cliques = new int[tot];
    int edge_mat[vertex][vertex];
    memset( edge_mat, 0, sizeof edge_mat );
    memset( cliques, 0, sizeof cliques );

    for( int i = 0; i < edge; i++ )
    {
        int a, b;
        fin >> a >> b;
        edge_mat[a][b] = 1;
        edge_mat[b][a] = 1;
    }

    find_clique(vertex, cliques, edge_mat[0]);

    cout << "All clique find done!" << endl;

    sos_dp(vertex, cliques);
```

```
    cout << "SOS Dp done!" << endl;

    int ans = cliqueCoverNumber(vertex, cliques);

    auto stop = chrono::high_resolution_clock::now();
    int duration = chrono::duration_cast<chrono::milliseconds>(stop - start).cou


    cout << "Clique Cover Number is: " << ans << endl;
    cout << "Time For Execution: " << duration << " ms" << endl;

    fout << ans << " " << duration << endl;
    return 0;
}
```

5.2 Iterative Greedy Implementation

```
#include<vector>
#include<algorithm>
#include<iostream>
#include<fstream>
#include<cstring>
#include<bitset>
#include<chrono>
#include<cstdlib>
#include<ctime>

using namespace std;

const int MAX_VERTEX = 30, SECONDS_TO_RUN = 20;

vector< vector<int> > solve( vector<int> perm, int **edge_mat, int vertex )
{
    vector< vector<int> > sol;
    for( int i = 0; i < vertex; i++ )
    {
        int ind = -1;
        for( int j = 0; j < sol.size(); j++ )
        {
            int f = 1;
            for( int k = 0; k < sol[j].size(); k++ )
            {
                if( edge_mat[i][ sol[j][k] ] == 0 )
                {
                    f = 0;
                    break;
```

```
                }
            }
            if(f)
            {
                ind = j;
                break;
            }
        }
        if(ind != -1)
        {
            sol[ind].push_back(i);
        }
        else
        {
            vector<int> vec = {i};
            sol.push_back(vec);
        }
    }
    return sol;
}

int main()
{
    srand(time(0));

    ifstream fin;
    //input file: vertex number start from 0
    //vertex edge
    //each edge in one line
    fin.open("in0.txt");

    ofstream fout;
    fout.open("out_ig0.txt");

    int vertex, edge;

    fin >> vertex >> edge;

    int **edge_mat;
    edge_mat = new int*[vertex];
    vector<int> perm;


    for( int i = 0; i < vertex; i++ )
    {
        perm.push_back(i);
```

```
        edge_mat[i] = new int[vertex];
    }

    for( int i = 0; i < vertex; i++ )
    {
        for( int j = 0; j < vertex; j++ )
        {
            edge_mat[i][j] = 0;
        }
    }

    for( int i = 0; i < edge; i++ )
    {
        int a, b;
        fin >> a >> b;
        edge_mat[a][b] = 1;
        edge_mat[b][a] = 1;
    }

    random_shuffle(perm.begin(), perm.end());

    auto start = chrono::high_resolution_clock::now();

    int ans = 1e9, duration = 0, it = 0;

    while(duration <= SECONDS_TO_RUN*1000)
    {
        vector< vector<int> > sol = solve(perm, edge_mat, vertex);

        int cur_sol = sol.size();
        ans = min( ans, cur_sol );


        int ind = rand()%cur_sol;
        swap( sol[0], sol[ind] );


        perm.clear();
        for( int i = 0; i < cur_sol; i++ )
        {
            for( int j = 0; j < sol[i].size(); j++ )
            {
                perm.push_back( sol[i][j] );
            }
        }
```

```
        it++;

        auto stop = chrono::high_resolution_clock::now();
        duration = chrono::duration_cast<chrono::milliseconds>(stop - start).cou

        if(it%100000==0)
        {
            cout << "Iteration No: " << it << endl;
        }
    }

    cout << "Clique Cover Number is: " << ans << endl;
    cout << "Time For Execution: " << duration << " ms" << endl;

    fout << ans << " " << duration << endl;


    return 0;
}
```

5.3 Datasets

5.3.1 Randomly Generated

10 randomly generated small graphs with nodes numbering 10-24 and random edges numbering 11-52.

Graph Names	Nodes	Edges
Gnp10_0.2.clq	10	11
Gnp11_0.2.clq	11	11
Gnp13_0.2.clq	13	11
Gnp14_0.2.clq	14	22
Gnp16_0.2.clq	16	27
Gnp17_0.2.clq	17	26
Gnp19_0.2.clq	19	30
Gnp21_0.2.clq	21	38
Gnp22_0.2.clq	2	51
Gnp24_0.2.clq	24	52

Table 1: Randomly Generated Graphs

5.3.2 DIMACS

A dataset provided by **Center for Discrete Mathematics and Theoretical Computer Science** which was used in 1993 for testing NP-Hard problems. There are a total of 21 graphs with different edge density,nodes and edges.

Graph Name	Nodes	Edges	Edge Density	Graph Type
C125.9.clq	125	6963	89.8 %	Dense
gen200_p0.9_44.b	200	17910	90 %	Dense
gen200_p0.9_55.b	200	17910	90 %	Dense
brock200_2.b	200	9876	50 %	Normal
brock200_4.b	200	13089	65 %	Normal
C250.9.clq	250	27984	89.9 %	Dense
p_hat300_1.clq	300	10933	24.4 %	Sparse
p_hat300-2.clq	300	21928	48.9 %	Normal
gen400_p0.9_55.b	400	71820	90 %	Dense
gen400_p0.9_75.b	400	71820	90 %	Dense
gen400_p0.9_65.b	400	71820	90 %	Dense
brock400_2.b	400	59786	75 %	Dense
brock400_4.b	400	59765	75 %	Dense
C500.9.clq	500	112332	90 %	Dense
p_hat700-1.clq	700	60999	24.9 %	Sparse
p_hat700-2.clq	700	121728	49.8 %	Normal
brock800_2.b	800	208166	65 %	Normal
brock800_4.b	800	207643	65 %	Normal
C1000.9.clq	1000	450079	90.1 %	Dense
hamming10-4.clq	1024	434176	82.9 %	Dense
p_hat1500-1.clq	1500	284923	25.3 %	Sparse

Figure 1: DIMACS Dataset

5.4 Runtime Analysis

Graph Name	Nodes	Edges	Clique Cover	Brute Force	Backtracking
Gnp10_0.2.clq	10	11	5	26.873	0.002
Gnp11_0.2.clq	11	11	6	324.117	0.007
Gnp13_0.2.clq	13	11	9	728.866	0.002
Gnp14_0.2.clq	14	22	6	578.319	0.131
Gnp16_0.2.clq	16	27	8	8047.176	11.05
Gnp17_0.2.clq	17	26	9	2093.475	1.977
Gnp19_0.2.clq	19	30	10	3851.746	2.356
Gnp21_0.2.clq	21	38	10	18745.945	7533.604
Gnp22_0.2.clq	22	51	9	3895.389	366.368
Gnp24_0.2.clq	24	52	11	2847.249	20.143

Figure 2: Runtime Comparison(in Seconds)

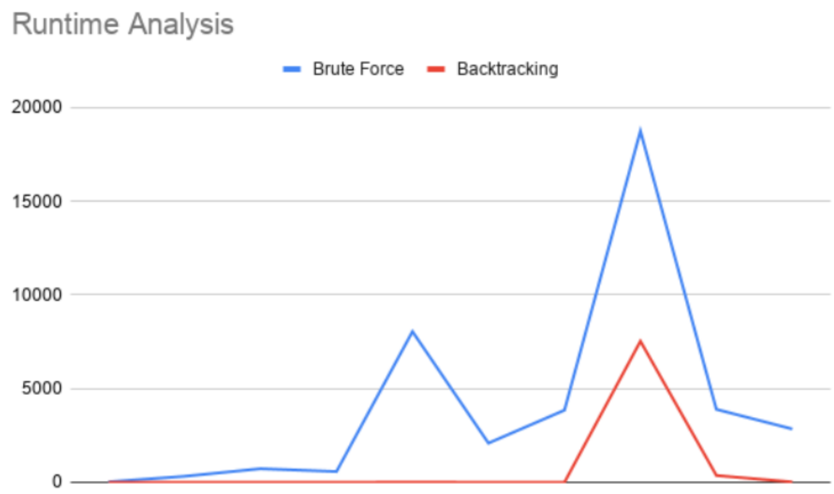

Figure 3: Runtime Analysis

5.5 Clique Cover Size Comparison

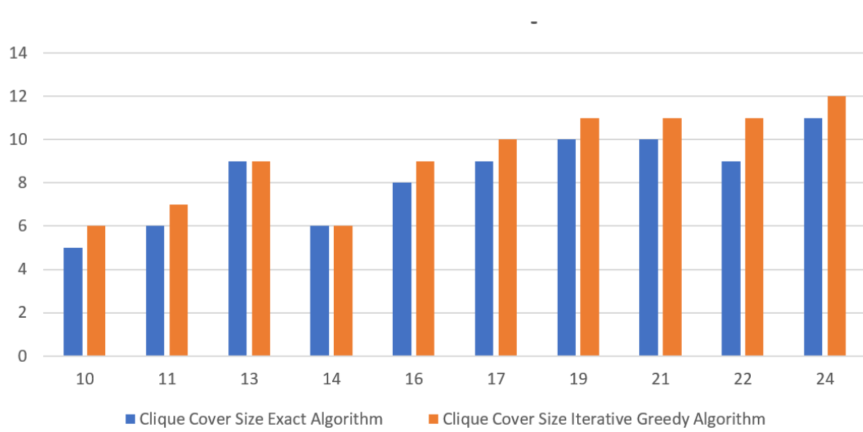

Figure 4: Exact vs IG Clique Cover Size Comparison

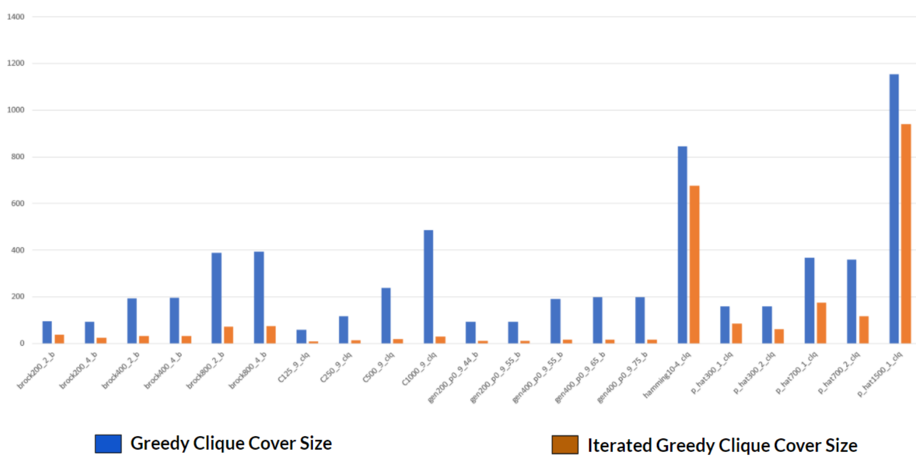

Figure 5: GCC vs IG Clique Cover Size Comparison