Texas A&M University

UIN - 928001157

Linear Programming

ISEN 622 Project

The Maximum Clique Problem

Author:
Surya Narayan Santhana Krishnan

Supervisor:
Dr. Sergiy Butenko

December 12, 2018

The Maximum Clique Problem

Surya Narayan Santhana Krishnan
UIN : 928001157
narayan@tamu.edu[1]

[1]Industrial & Systems Engineering, Texas A&M University, College Station, Texas 77840, United States

12/12/2018

Abstract

This article discusses the maximum clique problem and its relationship with the maximum independent set problem, their formulations and an implementation to solve them by Constraint Programming using Linear Relaxation through a Column-Generation approach.

1 Introduction

The maximum clique and maximum independent set problems are classical NP-Hard problems in combinatorial optimization. These problems are also closely associated with graph coloring and minimum clique partitioning problems. Due to their important role in several theoretical fields and applicability in a wide variety of practical settings, these problems have been extensively studied from different perspectives by mathematicians, computer scientists, operations researchers, engineers, biologists, and social scientists.

Clique-like structures frequently arise in many other applications, where one is interested in detecting large groups of elements that are all closely related to each other in some sense. Depending on the application of interest, such structures are often referred to as clusters, modules, complexes or cohesive subgroups. If the elements in the application of interest are represented as vertices (nodes) and the relationships between the elements are represented as edges (links, arcs), then clusters can be naturally modeled as cliques in this graph-theoretic representation.

Throughout this article, $G = (V, E)$ is an arbitrary undirected and unweighted graph. $V = \{1, 2, .., n\}$ is the vertex set of G and $E \subseteq V \times V$. $A_G = (a_{ij})_{n\times n}$ is the adjacency matrix of G, where $a_{ij} = 0$ if $(i,j) \notin E$. A *clique* is a subset of vertices $C \subseteq V$, if $i,j \in E$ for any $i,j \in C$. An *independent* set is a subset of vertices $C \subseteq V$, if $i,j \notin E$ for any $i,j \in C$. A clique (independent set) is called *maximal* if it is not a subset of a larger clique (independent set) in G, and *maximum* if there is no larger clique (independent set) in G. The cardinality of a of a maximum clique in G is denoted $\omega(G)$ and is called the *clique number* of G. It is also known as the fractional clique number.

2 Formulation

Let I^* denote the set of all maximal independent sets in G. Then the maximum clique problem can be formulated as the following integer program:

$$\begin{aligned} & maximize \sum_{j \in V} x_j \\ & subject\ to \sum_{j \in V} x_j \leqslant 1, \quad I \in I^* \\ & x_j \in \{0,1\},\ j \in V. \end{aligned} \tag{1}$$

The variables of the above LP for each vertex j ∈ V take the binary values 1 if vertex is included in the maximum clique, and 0 otherwise. The constraints of this LP correspond to every maximal independent set $I \in I^*$. The constraints represent the fact that every maximal independent set can atmost have one vertex member of the set belonging to the maximum clique. We relax the model by replacing the integrality constraints with non-negativity. We obtain the following linear program yielding an upper bound $\bar{\omega} \geqslant \omega(G)$:

$$\begin{aligned} \bar{\omega}(G) = max & \sum_{j \in V} x_j \\ s.t. & \sum_{j \in V} x_j \leqslant 1, \quad I \in I^* \\ & x_j \geqslant 0,\ j \in V. \end{aligned} \tag{2}$$

To convert the above formulation to a column generation scheme, we consider the dual of (2):

$$\begin{aligned} \bar{\omega}(G) = min & \sum_{I \in I^*} y_I \\ s.t. & \sum_{I \in I_j} y_I \geqslant 1, \quad j \in V \\ & x_j \geqslant 0,\ I \in I^*, \end{aligned} \tag{3}$$

where I_j denote the set of all maximal independent sets containing vertex $j \in V$.

2.1 Restricted Master Problem

In the above formulation, each column represent a maximal independent set. Thus, the variables of the above LP correspond to each maximal independent set, and the number of maximal independent sets in a graph on n vertices can be as large as $3^{n/3}$. Generating all maximal independent sets is as hard as solving the maximum clique problem, therefore we use a greedy approach to generate a set of few maximal independent sets such that they cover all the vertices. The generated set $I' \subseteq I^*$ is used as the basis (column-wise) to initialize the restricted master problem (RMP).

$$\begin{aligned} \bar{\omega}(G) = min & \sum_{I \in I'} y_I \\ s.t. & \sum_{I \in I'_j} y_I \geqslant 1, \quad j \in V \\ & x_j \geqslant 0,\ I \in I', \end{aligned} \tag{4}$$

2.2 Column Generation Sub-Problem

The CGSP would then be:

$$w = \max_{I \in I^*} \left\{ \sum_{j \in I} d_j - 1 \right\}, \tag{5}$$

where d is the dual prices of (4). Thus, the CGSP is a maximum weight independent set problem, seeking to find an independent set maximizing the sum of vertex weights in G, where the weights are given by d. Also, d is the optimal solution of the following LP, which is dual to the RMP (4):

$$\begin{aligned} & max \sum_{j \in V} x_j \\ & s.t. \sum_{j \in I} x_j \leqslant 1, \quad I \in I' \\ & \quad x_j \geqslant 0, \; j \in V. \end{aligned} \tag{6}$$

The CGSP can be solved using the following IP formulation:

$$\begin{aligned} & max \sum_{j \in V} d_j x_j \\ & s.t.\; x_i + x_j \leqslant 1, \quad \{i, j\} \in E \\ & \quad x_j \in \{0, 1\} \; j \in V. \end{aligned} \tag{7}$$

3 Implementation

The greedy algorithm used to generate the subset I' of maximal independent sets and the column generation algorithm is shown below:

3.1 Greedy generation of maximal independent sets

To find a good starting basis for the RMP (4), we generate a set of maximal independent sets such that all the vertices of G are included. The pseudo-code is given in Algorithm 1.

The algorithm works as follows. For each vertex j, starting with j, all neighbours of j are removed, and then, at each step, a minimum degree vertex is chosen from the residual graph R_G and added to the maximal independent set of vertex j. Subsequently, as vertices are added to the set, their neighbours are removed and the steps are repeated until the residual graph is empty. This algorithm thus leaves us with a set of maximal independent sets for each $j \in V$. This algorithm is implemented in C++. The complete C++ program is in section 5.

3.2 Constraint Programming - RMP & CGSP Implementation

The RMP & CGSP problems are implemented in AMPL mathematical programming language and Gurobi 8.1.0 solver is used. The .mod and .run file code is in section 5. The .dat file is generated as a matrix from the generated set I' by the C++ program. The AMPL implementation column generation approach is given in Algorithm 2.

Algorithm 1 GREEDY MAXIMAL INDEPENDENT SETS

n = number of vertices in G
//Let $\mathbf{R_G}$ *be the* **residual graph**
//Let **set[1..n]** *be the set of maximal independent sets containing vertex* **j**
$for\ j\ =\ 1\ to\ n:$
    $R_G\ =\ G$
    R_G.**remove**(j)
    $set[j]$.**insert**(j)
    $\mathbf{R_G}$.**remove_neighbours**(j)
    $if\ R_G \neq \varnothing:$
        x = **get_min_degree_vertex**(R_G)
    $while\ R_G \neq \varnothing:$
        $set[j]$.**insert**(x)
        R_G.**remove**(x)
        R_G.**remove**_*neighbours*(x)
        $if\ R_G \neq \varnothing:$
            x = **get_min_degree_vertex**(R_G)

Algorithm 2 COLUMN GENERATION IN AMPL

Master Problem $\leftarrow formulation(4)$
Sub-Problem $\leftarrow formulation(7)$
do :
    solve **Master Problem**
    get **Dual prices** *from Master Problem*
    solve **Sub-Problem** *using the dual prices*
    if **CGSP objective** $<=\ 1:$
        break *loop*
    else :
        add *the Optimal solution of the sub − problem as a* **new column** *to the master problem*
continue *loop*

4 Results

19 graph instances were chosen from the second DIMACS implementation challenge and ran. The results are tablulated below.

$Instance$	$\|V\|$	$\|E\|$	$\bar{\omega}$	$DIMACS\ \bar{\omega}$	$no.\ of\ columns\ generated$
$brock$200_1	200	14834	38.0161	21	362
$brock$200_2	200	9876	21.127	12	572
$brock$200_3	200	12048	27.2307	15	472
$brock$200_4	200	13089	30.6283	17	450
c125.9	125	6963	43.0567	34	128
c250.9	250	27984	71.3746	44	333
c500.9	500	112332	126.5030	57	328
c-fat200-5	200	8473	66.6667	58	175
$hamming$6-2	64	1824	32	32	21
$hamming$6-4	64	704	5.33333	4	108
$hamming$8-2	256	31616	128	128	108
$hamming$8-4	256	20864	16	16	500
san200_0.7_1	200	13930	30	30	422
san200_0.7_2	200	13930	18	18	4019
san200_0.9_1	200	17910	70	70	100
san200_0.9_2	200	17910	60	60	187
san200_0.9_3	200	17910	44	44	690
$sanr$200_0.7	200	13868	33.4807	18	313
$sanr$200_0.9	200	17863	59.8245	42	242

5 C++ program and AMPL code

The C++ program contains the following subroutines:

1. Adjacency List represenation of Graph
2. Adjacency Matrix represenation of Graph
3. MAIN routine
4. greedy_RMP subroutine
5. remove_neighbours subroutine
6. get_min_degree_vertex subroutine
7. Maximality check 1 subroutine
8. Maximality check 2 subroutine

5.1 COLUMN GENERATION IMPLEMENTATION IN AMPL

.MOD FILE:

```
# ---------------------------------------------
# MASTER PROBLEM
# ---------------------------------------------
set SETS_J;
param nSETS integer >= 0;
set MI_SETS;

let nSETS := 0;
set M_SETS:= 1..nSETS;
set EDGES;
set NODES;

param a{SETS_J, MI_SETS} integer >= 0;
param z{SETS_J, M_SETS} default 0;
param e{EDGES, NODES};

var y{M_SETS} integer >= 0;

minimize Independence:
      sum {j in M_SETS} y[j];

subj to Sets_Containing_j{i in SETS_J}:
      sum {j in M_SETS} z[i,j] * y[j] >= 1;
```

```
# ---------------------------------------------

# CGSP - MAXIMUM WEIGHT INDEPENDENT SET PROBLEM

# ---------------------------------------------


param d{SETS_J} default 0.0;

var x{SETS_J} binary;


maximize I_Set_Weight:

      sum {i in SETS_J} d[i] * x[i];                  #d is the vector of
dual prices of the MP


subj to Edge_Constraints{i in EDGES}:

      #{j in NODES : j = 'node1'} x[e[i,j]] + {k in NODES : k =
'node2'}x[e[i,k]] <= 1;

      x[e[i,'node1']] + x[e[i,'node2']] <= 1;
```

.RUN FILE:

```
# ------------------------

# RUN FILE

# ------------------------


reset;


option solver gurobi;

option solution_round 6;


model clique.mod;

data clique.dat;
```

```
problem Master_Pro: y, Independence, Sets_Containing_j;
      option relax_integrality 1;
      option presolve 0;

problem Sub_Pro: x, I_Set_Weight, Edge_Constraints;
      option relax_integrality 0;
      option presolve 0;

param sub_counter integer >=0;

param breaker integer >=0;
let sub_counter := 0;
let breaker := 0;

for {q in SETS_J}{
      let nSETS := nSETS + 1;
      let {j in SETS_J} z[j, nSETS] := a[j, nSETS];
      };

repeat {
      solve Master_Pro;
      let {i in SETS_J} d[i] := Sets_Containing_j[i].dual;

      solve Sub_Pro;
      #display x;
      display I_Set_Weight;

      if I_Set_Weight > 1.001 then {
            let sub_counter := sub_counter + 1;
```

```
            let nSETS := nSETS + 1;
            display nSETS;
            let {k in SETS_J} z[k,nSETS] := x[k];
            }
      else {
            let breaker := -1;
            break;
            }
      };

      display Independence;
      display sub_counter;
      display breaker;
      display nSETS;
```

5.2 C++ PROGRAM TO GENERATE MAXIMAL INDEPENDENT SETS & CREATE .DAT FILE

```
#include <iostream>

#include <fstream>
#include <string>
#include <set>
#include <vector>

using namespace std;

int vertex_count;
int edge_count;

struct node
{
      int value = 0;
      struct node *next = NULL;
};
typedef struct node NODE;

struct LList
```

```
{
      NODE *head = new NODE();
      struct node *tail;
};
typedef struct LList LLIST;

//int v_count;
LLIST *Adj = NULL;

class Graph {

private:

      bool** adjacencyMatrix;
      int vertexCount;
      bool init;

public:

      Graph(int vertexCount, bool init) {

            this->vertexCount = vertexCount;
            this->init = init;

            adjacencyMatrix = new bool*[vertexCount];

            for (int i = 0; i < vertexCount; i++) {

                  adjacencyMatrix[i] = new bool[vertexCount];

                  for (int j = 0; j < vertexCount; j++)

                        adjacencyMatrix[i][j] = init;

            }

      }



      void addEdge(int i, int j) {

            if (i >= 0 && i < vertexCount && j >= 0 && j < vertexCount) {

                  adjacencyMatrix[i][j] = true;

                  adjacencyMatrix[j][i] = true;

            }
```

```
	}

	void removeEdge(int i, int j) {

		if (i >= 0 && i < vertexCount && j >= 0 && j < vertexCount) {

			adjacencyMatrix[i][j] = false;

			adjacencyMatrix[j][i] = false;

		}

	}

	bool isEdge(int i, int j) {

		return adjacencyMatrix[i][j];

	}

	void printG(void) {
		for (int i = 0; i < vertexCount; i++) {
			for (int j = 0; j < vertexCount; j++) {
				cout << adjacencyMatrix[i][j] << " ";
			}
			cout << endl;
		}
	}

	~Graph() {

		for (int i = 0; i < vertexCount; i++)

			delete[] adjacencyMatrix[i];

		delete[] adjacencyMatrix;

	}

};

vector<set<int>> greedy_RMP(LLIST *Adj, const int& vertex_count, Graph& G);
void remove_neighbours(set<int>& mset, int i);
int get_min_degree_vertex(set<int>& mset, Graph& G);
void Check_Maximality(vector<set<int>>& mxl_set, Graph& G);
```

```
void Check_Maximality2(vector<set<int>>& mxl_set, Graph& G);

void addEdge(const int& v1, const int& v2)
{
	NODE *ptr = new NODE();
	ptr->value = v2;

	if (Adj[v1 - 1].head->value == 0) //value 0 means no neighbours yet
	{
		Adj[v1 - 1].head = ptr;
		Adj[v1 - 1].tail = Adj[v1 - 1].head;
	}
	else
	{
		Adj[v1 - 1].tail->next = ptr;
		Adj[v1 - 1].tail = ptr;
	}
}
//int v_count;
void printAdj(int v_count)
{
	//if zero is printed, it means no neighbours
	for (int i = 0; i < v_count; i++)
	{
		cout << "Adjacency list of " << i + 1 << " :" << endl;
		NODE *ptr = Adj[i].head;

		bool flag = 1;
		while (flag)
		{
			cout << ptr->value;
			if (ptr->next == NULL)
				flag = 0;
			else
			{
				ptr = ptr->next;
				cout << ",";
			}
		}
		cout << endl;
	}
}

struct Graph_L {
	int *V;
	LLIST *Nei;
};

set<int> all_V;
void greedy_RMP(LLIST *Adj, const int& vertex_count);
```

```
int *edge_vec;

int main(void)
{
	ifstream file;
	string filepath, filename, name, keyword, word, a, b, e;
	int node_1, node_2;

	filepath = "F:\\Research\\Problem sets\\Max
Clique\\DIMACS_all_ascii\\";
	name = "p_hat300-1.clq";
	filename = filepath + name;
	if (name[0] == 'D' | name[0] =='M' | name[0] == 'b' | name[0] == 's' |
name[0] == 'c' | name[0] == 'h' | name[0] == 'p')
		keyword = "edge";
	else
		keyword = "col";
	file.open(filename.c_str());

	while (file >> word && word != keyword);
	file >> word;
	vertex_count = stoi(word);
	file >> word;
	edge_count = stoi(word);
	cout << "No. of nodes: " << vertex_count << endl;
	cout << "No. of edges: " << edge_count << endl;

	bool graph_inverted = false;
	Graph Gm(vertex_count, graph_inverted);
	//Graph_L G;

	int *Vertices = new int[vertex_count];
	Adj = new LLIST[vertex_count];

	edge_vec = new int[edge_count * 2];

	int j = 0;
	for (int i = 1; i <= edge_count; i++)
	{
		file >> e >> a >> b;
		node_1 = stoi(a) - 1;
		node_2 = stoi(b) - 1;
		edge_vec[j] = node_1 + 1;
		edge_vec[j + 1] = node_2 + 1;
		j = j + 2;
		Gm.addEdge(node_1, node_2);
		addEdge(node_1 + 1, node_2 + 1);
		addEdge(node_2 + 1, node_1 + 1);

	}
```

```
	for (int i = 0; i < vertex_count; i++)
	{
		all_V.insert(i);
	}

	//generate maximal Independent sets for each vertex (Greedy)
	vector<set<int>> mxl_set;
	mxl_set = greedy_RMP(Adj, vertex_count, Gm);

	//printAdj(vertex_count);
	//Gm.printG();
	//cout << "ex: " << Adj[0].head->next->next->next->next->next->next-
>next->next->next->next->value;
}


vector<set<int>> greedy_RMP(LLIST *Adj, const int& vertex_count, Graph& G)
{
	vector < set<int> > mxl_set(vertex_count);
	node *ptr;
	set<int> set_all;
	int x;

	for (int i = 0; i < vertex_count; i++)
	{
		//initialise set with all vertices
		set_all = all_V;

		//remove the vertex for which the set is defined (first residual
graph)

		set_all.erase(i);
		mxl_set[i].insert(i);
		//remove all neighbours for the vertex for which the set is
defined (trivial step)
		remove_neighbours(set_all, i);
		//get minimum degree vertex from the residual graph
		if (!set_all.empty())
			x = get_min_degree_vertex(set_all, G);
		while (!set_all.empty())
		{
			mxl_set[i].insert(x);
			set_all.erase(x);
			remove_neighbours(set_all, x);
			if (!set_all.empty())
				x = get_min_degree_vertex(set_all, G);
		}
```

```
		cout << endl;
		//int i = 0;
		cout << "Size of independent set containing vertex " << i + 1 <<
" : " << mxl_set[i].size() << endl;
		for (auto it : mxl_set[i])
			cout << it + 1 << "--";
		cout << endl;

	}
	//Check_Maximality(mxl_set, G);
	//Check_Maximality2(mxl_set, G);

	ofstream fout;
	fout.open("F:\\Research\\Problem sets\\Max Clique\\Output\\out.txt");
	for (int i = 0; i < vertex_count; i++)
	{
		set<int>::iterator it = mxl_set[i].begin();
		fout << *it+1;
		it++;
		for (it; it!=mxl_set[i].end(); it++)
		{
			fout << " " << *it+1;
		}
		fout << endl;
	}
	fout.close();

	//struct

	//vector<vector<int> vect(vertex_count, 0)> Tech_matrix(vertex_count);
	int *Tech_matrix = new int[vertex_count*vertex_count]();
	//cout << endl << "Tech[last element] : " <<
Tech_matrix[(vertex_count*vertex_count)-1];
	//delete [] Tech_matrix;

	int row_id = 0, col_id = 0;
	for (int i = 0; i < vertex_count; i++)
	{
		for (auto it : mxl_set[i])
		{
			row_id = i * vertex_count;
			col_id = it;
			Tech_matrix[row_id + col_id] = 1;
		}
	}

	fout.open("F:\\Research\\Problem sets\\Max
Clique\\Output\\out_matrix.txt");
	for (int i = 0; i < vertex_count; i++)
	{
```

```
		for (int j = 0; j < vertex_count; j++)
		{
			fout << Tech_matrix[(i*vertex_count) + j] << " ";
		}
		fout << endl;
	}
	fout.close();

	fout.open("F:\\Research\\Problem sets\\Max Clique\\Test\\clique.dat");
	//param e
	string sEDGES, sMI_SETS, sSETS_J;
	for (int i = 0; i < edge_count; i++)
	{
		sEDGES = sEDGES + " e" + to_string(i + 1);
	}
	for (int i = 0; i < vertex_count; i++)
	{
		sMI_SETS = sMI_SETS + " " + to_string(i + 1);
		sSETS_J = sSETS_J + " " + to_string(i + 1);
	}
	fout << "set EDGES :=" << sEDGES << ";" << endl;
	fout << "set NODES :=" << " node1 node2;" << endl << endl;
	fout << "set MI_SETS :=" << sMI_SETS << ";" << endl;
	fout << "set SETS_J :=" << sSETS_J << ";" << endl << endl;

	fout << "param e: node1 node2 :=" << endl;

	int j = 0;
	for (int i = 0; i < edge_count; i++)
	{
		fout << "e" << i + 1 << " ";
		fout << edge_vec[j] << " " << edge_vec[j + 1] << endl;
		j = j + 2;
	}
	fout << ";" << endl;

	fout << "param a: ";
	fout << sMI_SETS;
	fout << " :=" << endl;

	for (int i = 0; i < vertex_count; i++)
	{
		//fout << "sets_j";
		fout.width(3); fout << left << i+1;
		for (int j = 0; j < vertex_count; j++)
		{
			fout << "  " << Tech_matrix[(j*vertex_count) + i];
		}
		fout << endl;
	}
```

```
	fout << ";";
	fout.close();
	delete[] Tech_matrix;



	return mxl_set;
}

void remove_neighbours(set<int>& mset, int i)
{
	node *ptr;
	ptr = Adj[i].head;
	while (ptr != NULL)
	{
		mset.erase(ptr->value-1);
		ptr = ptr->next;
	}
}

int get_min_degree_vertex(set<int>& mset, Graph& G)
{
	int deg = 0;
	int prev_deg = mset.size();
	int x;

	for (auto it_1 : mset)
	{
		deg = 0;
		for (auto it_2 : mset)
		{
			deg += G.isEdge(it_1, it_2);
		}
		if (deg <= prev_deg)
		{
			x = it_1;
			prev_deg = deg;
		}
	}
	if (!(x > -1) && !(x < vertex_count))
		cout << "Uninitialised error";
	return x;
}

void Check_Maximality(vector<set<int>>& mxl_set, Graph& G)
{
	cout << endl;
	cout << "vec size " << mxl_set.size() << endl;
	bool b = true;
	set<int> s_all, s1_all;
```

```
	for (int i = 0; i < mxl_set.size(); i++)
	{
		s_all = all_V;
		s1_all = s_all;
		for (auto it : mxl_set[i])
		{
			for (auto it2 : s_all)
			{
				if (G.isEdge(it, it2))
					s1_all.erase(it2);
			}
			s_all = s1_all;
		}
		if (s_all!=mxl_set[i])
		{
			b = false;
			break;
		}
	}

	cout << "Maximality check: " << b;
}

void Check_Maximality2(vector<set<int>>& mxl_set, Graph& G)
{
	cout << endl;
	cout << "vec size " << mxl_set.size() << endl;
	bool b = false;
	set<int> s_all, s1_all;

	for (int i = 0; i < mxl_set.size(); i++)
	{
		s_all = all_V;
		s1_all = s_all;

		for (auto it : mxl_set[i])
		{
			s_all.erase(it);
		}

		for (auto it : s_all)
		{
			b = false;
			//cout << b;
			for (auto it2 : mxl_set[i])
			{
				b = b || G.isEdge(it, it2);
			}
			//cout << b;
```

```
                        //cout << endl;
                        if (b == true)
                            break;
                    }
            }

            cout << "Maximality check 2: " << b;
    }
```