Investigacion Operativa
Coloreo Particionado de Grafos

2 de diciembre de 2015

Integrante	LU	Correo electrónico
Martin Baigorria	575/14	martinbaigorria@gmail.com
Andrew Ab	???	???

Reservado para la cátedra

Instancia	Docente	Nota
Primera entrega		
Segunda entrega		

Resumen: ???

Keywords: ???

Índice

1. Modelo

Dado un grafo $G(V, E)$ con $n = |V|$ vertices y $m = |E|$ aristas, un coloreo de G se define como una asignacion de un color o etiqueta a cada $v \in V$ de forma tal que para todo par de vertices adyacentes $(p, q) \in E$ poseen colores distintos. El clasico problema de *coloreo de grafos* consiste en encontrar un coloreo del grafo que utilize la menor cantidad de colores posibles.

En este trabajo resolveremos una variante de este problema, el *coloreo particionado de grafos*. A partir de un conjunto de vertices V que se encuentra particionado en $V_1, ..., V_k$, el problema consiste en asignar un color $c \in C$ a solo un vertice de cada particion de forma tal que dos vertices adyacentes no reciban el mismo color y minimizando la cantidad de colores utilizados.

Este problema se puede modelar con Programacion Lineal Entera. Para ello, definamos las siguientes variables:

$$x_{pj} = \begin{cases} 1 & \text{si el color } j \text{ es asignado al vertice } p \\ 0 & \text{en caso contrario} \end{cases}$$

$$w_j = \begin{cases} 1 & \text{si } x_{pj} = 1 \text{ para algun vertice } p \\ 0 & \text{en caso contrario} \end{cases}$$

1.1. Funcion objetivo

De esta forma la funcion objetivo del LP consiste en minimizar la cantidad de colores utilizados:

$$min \sum_{j \in C} w_j \tag{1}$$

Notar que $|C|$ esta acotado superiormente por la cantidad de particiones k.

1.2. Restricciones

Los vertices adyacentes no comparten color. Recordar que no necesariamente se le asigna un color a todo vertice.

$$x_{ij} + x_{kj} \leq 1 \qquad \forall (i, k) \in E, \;\; \forall j \in C \tag{2}$$

Solo se le asigna un color a un unico vertice de cada particion $p \in P$. Esto implica que cada vertice tiene a lo sumo solo un color.

$$\sum_{i \in V_p} \sum_{j \in C} x_{ij} = 1 \qquad \forall p \in P \tag{3}$$

Si un nodo usa color j, $w_j = 1$:

$$x_{ij} \leq w_j \qquad \forall i \in V, \forall j \in C \tag{4}$$

Integralidad y positividad de las variables:

$$x_{ij} \in \{0, 1\} \qquad \forall i \in V, \forall j \in C \tag{5}$$

$$w_j \in \{0, 1\} \qquad \forall j \in C \tag{6}$$

1.3. Eliminacion de simetrias

Una de nuestras ideas para eliminar las simetrias fue usar la clasica condicion de coloreo que dice que los colores se deben utilizar en orden. Aunque existen otras, notamos que esta condicion mejoro ampliamente la ejecucion del LP. Formalmente, se puede expresar como:

$$w_j \geq w_{j+1} \qquad \forall\, 1 \leq j \leq |C| \tag{7}$$

2. Branch & Bound

La implementacion del modelo y del Branch & Bound se encuentran en el apendice.

3. Desigualdades

3.1. Desigualdad de Clique

Sea $j_0 \in \{1, ..., n\}$ y sea K una clique maximal de G. La desigualdad clique estan definida por:

$$\sum_{p \in K} x_{pj_0} \leq w_{j_0} \tag{8}$$

Demostración Para esta demostracion utilizaremos las desigualdades Chvátal-Gomory sobre las restricciones del LP planteado en la seccion 1.2 e induccion. A priori el teorema es bastante intuitivo. Si pinto algun vertice de una clique, no puedo pintar ninguno adyacente del mismo color sin importar la forma en la que particione los vertices del grafo. Sea n el tamanio de la clique maximal.

Casos Base

1. $n = 1$: Si en la clique maximal tengo solo un vertice, no existe arista que contenga este vertice, caso contrario la clique tendria dos elementos. Por lo tanto, este vertice puede estar pintado o no dentro de la particion. Es decir, se cumple la ecuacion que queremos probar.

2. $n = 2$: Si la clique maximal tiene dos elementos, por definicion son conexos. Por la restriccion que indica que los vertices adyacentes no comparten color, aqui hay 2 opciones. La primera opcion es que a ningun vertice se le asigna un color j_0. La otra opcion es que dada la estructura de particiones, se le asigne solo a uno de ellos el color j_0. Por lo tanto la desigualdad para $n = 2$ vale.

3. $n = 3$: Este es el caso mas interesante en el que utilizamos la desigualdad de Chvátal-Gomory. Si la clique tiene 3 vertices, hay tres desigualdades que se deben cumplir:

   - $x_{1j_0} + x_{2j_0} \leq 1$
   - $x_{2j_0} + x_{3j_0} \leq 1$
   - $x_{1j_0} + x_{3j_0} \leq 1$

   Multiplicando todas estas desigualdades por 1/3 y sumando entonces:
   $1/3(x_{1j_0} + x_{2j_0}) + 1/3(x_{2j_0} + x_{3j_0}) + 1/3(x_{2j_0} + x_{3j_0}) \leq 3/2$
   Como x_{ij} toma valores enteros, entonces: $1/3(x_{1j_0} + x_{2j_0}) + 1/3(x_{2j_0} + x_{3j_0}) + 1/3(x_{2j_0} + x_{3j_0}) \leq 1$
   Simplificando: $x_{1j_0} + x_{2j_0} + x_{3j_0} \leq 1$.
   Utilizando la definicion de w_j entonces: $x_{1j_0} + x_{2j_0} + x_{3j_0} \leq w_{j_0}$
   Por lo tanto la desigualdad vale para $n = 3$.

Paso Inductivo: $P(n-1) \implies P(n)$
Como vale la hipotesis inductiva, sabemos que:

$$\sum_{p \in K-n} x_{pj_0} \leq w_{j_0}$$

Al agregar un vertice a la clique, agregamos $n - 1$ aristas:
$x_{1j_0} + x_{nj_0} \leq 1$, $x_{2j_0} + x_{nj_0} \leq 1$,..., $x_{(n-1)j_0} + x_{nj_0} \leq 1$
Utilizando esto, podemos ver que:

$$x_{nj_0} + \sum_{p \in K-n} x_{pj_0} \leq w_{j_0}$$

Esto es claramente equivalente a lo que queremos demostrar y se puede justificar a partir de dos casos:

- Si al vertice x_{nj_0} se le asigna un color, por las restricciones de las aristas que agregamos al resto de los vertices de la clique no se le puede asignar el color j_0.
- Si al vertice x_{nj_0} no se le asigna un color o se le asigna un color diferente a j_0, por hipotesis inductiva sabemos que lo que queremos probar vale. $\square$

3.2. Desigualdad de Aujero Impar

Sea $j_0 \in \{1, ..., n\}$ y sea $C_{2k+1} = v_1, ..., v_{2k+1}$, $k \geq 2$, un aujero de longitud impar. La desigualdad esta definida por:

$$\sum_{p \in C_{2k+1}} x_{pj_0} \leq kw_{j0} \tag{9}$$

Demostración Por teoremas de coloreo (que se prueban en general por induccion), sabemos que el numero cromatico $\chi(C) = 3$. En el peor de los casos, cada vertice del aujero estara en una particion diferente. Aqui nuevamente tenemos dos casos:

- Si no se asigna el color j_0 a algun vertice del aujero, la desigualdad vale.
- Si se asigna el color j_0, en el peor de los casos el mismo sera utilizado por a lo sumo $(|C| - 1)/2$ vertices. Como $|C| = 2k + 1$, $(2k + 1 - 1)/2 = k$. Por lo tanto vale la desigualdad. □

3.3. Planos de Corte

Luego de relajar el PLEM, los algoritmos de separacion buscan acotar el espacio de busqueda para que se parezca mas a la capsula convexa. Existen algoritmos de separacion exactos y heuristicos. Los algoritmos heuristicos, luego de resolver la relajacion del problema entero y encontrar una solucion optima x^*, retornan una o mas desigualdades de la clase violadas por alguna familia de desigualdades.

Dado que es un algoritmo heuristico, es posible que exista una desigualdad de la clase violada aunque el procedimiento no sea capaz de encontrarla. Si se encuentra una desigualdad que es violada por la solucion optima de la relajacion, se agrega esta nueva restriccion y se vuelve a resolver el programa lineal. Este procedimiento se conoce como algoritmo de plano de corte. Si una solucion optima al problema existe, este tipo de algoritmo no necesariamente la encuentra. Por ejemplo, las heuristicas que encuentran desigualdades validas pueden fallar y el algoritmo no puede continuar.

3.4. Heuristicas

Las heuristicas que enunciaremos a continuacion utilizan algunas propiedades de la representacion de nuestro grafo, ya sea para su construccion o para lograr una mejor complejidad temporal y espacial.

En primer lugar, representamos la estructura del grafo mediante una matriz de adyacencias. Esta matriz se implemento utilizando una lista. Dado que la matriz de adyacencias es simetrica y la diagonal no es necesaria para este problema en particular, guardamos solo la parte triangular superior de la misma. Esto nos da la ventaja de poder saber si dos vertices son adyacentes o no en $\mathcal{O}(1)$ y reduce la complejidad espacial de forma considerable. La formula que utilizamos para generar la biyeccion entre arista e indice en la lista se puede ver claramente en el codigo. La idea es bastante simple y se basa principalmente en usar la expresion para la suma de enteros consecutivos.

En segundo lugar, numeramos todos los vertices con enteros comenzando con $id = 1$. Por construccion, luego nuestras heuristicas nos garantizaran que nuestro conjunto de indices que representa a un miembro de una familia esta ordenado. Esto es muy ventajoso en el sentido que podemos saber si un nuevo potencial miembro de la familia ya ha sido agregado a la misma. Las familias se generan solo una vez al principio, y luego en diferentes iteraciones de los algoritmos de planos de corte se verifica si son violadas para ser agregadas como restricciones.

3.4.1. Heuristica de Separacion para Clique Maximal

Para esta heuristica, lo que hacemos es recorrer los vertices en orden. En primer lugar, tomamos el primer vertice, y luego comenzamos a recorrer la lista hasta que encontramos un vertice adyacente. Lo agregamos al conjunto que representa al miembro de la clique, y seguimos agregando elemento en orden de forma que cumplan que son adyacentes con todos los que ya hemos agregado. Una vez recorrida toda la lista, agregamos este conjunto a la familia. Luego comenzamos a generar una nueva familia a partir del segundo vertice, y asi sucesivamente. Este algoritmo se puede ilustrar con el siguiente pseudocodigo:

Algorithm 1 Algoritmo para generar familia de cliques maximales

procedure GENERATECLIQUEFAMILLY(V, E)
    $set < set < int >> clique\_familly$
    for $id \leftarrow 1, |V|$ **do**
        $set < int > clique$
        clique.insert(id)
        for $id2 \leftarrow id + 1, |V|$ **do**
            if clique.adyacentToAll(id2) **then**
                clique.insert(id2)
            end if
        end for
        if $\neg clique\_familly.isContained(clique)$ **then**
            clique_familly.insert(clique)
        end if
    end for
end procedure

Notar que en la practica solo consideramos cliques de tamanio mayor a 2, dado que si no se pisan con las restricciones de adyacencia del LP.

3.4.2. Heuristica de Separacion para Aujero Impar

Para esta heuristica, seguimos un procedimiento similar al anterior. Recorremos los vertices en orden, y los vamos agregando si son adyacentes. Al final, el conjunto de vertices resultante es un camino. Luego, vemos si el ultimo elemento del camino es adyacente al primero y si el camino tiene longitud impar. Si esto sucede, agregamos el conjunto a la familia. Si no sucede, quitamos el ultimo elemento y verificamos nuevamente la condicion hasta que se satisfaga. Este procedimiento se puede ilustrar con el siguiente pseudocodigo:

Algorithm 2 Algoritmo para generar familia de aujeros impares

procedure GENERATEODDHOLEFAMILLY(V, E)
    $set < set < int >> oddhole\_familly$
    for $id \leftarrow 1, |V|$ **do**
        $set < int > path$
        path.insert(id)
        for $id2 \leftarrow id + 1, |V|$ **do**
            if isAdyacent(path.end, id2) **then**
                path.insert(id2)
            end if
        end for
        while path.size() ≥ 3 **and** (path.size() mod 2 == 0 **or** ¬isAdyacent(path.start, path.end)) **do**
            path.erase(path.end)
        end while
        if path.size() ≥ 3 **and** isAdyacent(path.start, path.end) **then**
            oddhole_familly.insert(path)
        end if
    end for
end procedure

4. Cut & Branch

5. Experimentacion

6. Conclusion

7. Apéndice A: Código

7.1. coloring.cpp

```cpp
#include <ilcplex/ilocplex.h>
#include <ilcplex/cplex.h>

#include <stdlib.h>
#include <cassert>

#include <algorithm>
#include <string>
#include <vector>
#include <set>

#define TOL 1e-05
#define CUTTING_PLANE_ITERATIONS 1

ILOSTLBEGIN // macro to define namespace

// helper functions
int getVertexIndex(int id, int color, int partition_size);
inline int fromMatrixToVector(int from, int to, int edge_size);
inline bool isAdyacent(int from, int to, int edge_size, bool* adyacencyList);
bool adyacentToAll(int id, int edge_size, bool* adyacencyList, const set<int>& clique);
bool cliqueNotContained(const set<int>& clique, const set<set<int> >& clique_set);

// load LP
int loadObjectiveFunction(CPXENVptr& env, CPXLPptr& lp, int vertex_size, int partition_size, char vtype);
int loadAdyacencyColorRestriction(CPXENVptr& env, CPXLPptr& lp, int vertex_size, int edge_size, int partition_size, bool* adyacencyList);
int loadSingleColorInPartitionRestriction(CPXENVptr& env, CPXLPptr& lp, vector<vector<int> >& partitions, int partition_size);
int loadAdyacencyColorRestriction(CPXENVptr& env, CPXLPptr& lp, int vertex_size, int partition_size);
int loadSymmetryBreaker(CPXENVptr& env, CPXLPptr& lp, int partition_size);

// cutting planes
int loadCuttingPlanes(CPXENVptr& env, CPXLPptr& lp, int vertex_size, int edge_size, int partition_size, bool* adyacencyList);

int maximalCliqueFamillyHeuristic(set<set<int> >& clique_familly, int vertex_size, int edge_size, int partition_size, bool* adjacencyList);
int findUnsatisfiedCliqueRestrictions(CPXENVptr& env, CPXLPptr& lp, set<set<int> >& clique_familly, int vertex_size, int partition_size, int n, double* sol);
int loadUnsatisfiedCliqueRestriction(CPXENVptr& env, CPXLPptr& lp, int partition_size, const set<int>& clique, int color);

int oddholeFamillyHeuristic(set<set<int> >& oddhole_familly, int vertex_size, int edge_size, int partition_size, bool* adyacencyList);
int findUnsatisfiedOddholeRestrictions(CPXENVptr& env, CPXLPptr& lp, set<set<int> >& oddhole_familly, int vertex_size, int partition_size, int n, double* sol);
int loadUnsatisfiedOddholeRestriction(CPXENVptr& env, CPXLPptr& lp, int partition_size, const set<int>& path, int color);

```

```
// cplex functions
int solveLP(CPXENVptr& env, CPXLPptr& lp, int edge_size, int vertex_size, int partition_size);
int convertVariableType(CPXENVptr& env, CPXLPptr& lp, int vertex_size, int partition_size, char vtype);
int setBranchAndBoundConfig(CPXENVptr& env);
int checkStatus(CPXENVptr& env, int status);

// colors array!
const char* colors[] = {"Blue", "Red", "Green", "Yellow", "Grey", "Green", "Pink", "AliceBlue","AntiqueWhite","Aqua","Aquamarine","Azure","Beige",
"Bisque","Black","BlanchedAlmond","BlueViolet","Brown","BurlyWood","CadetBlue","Chartreuse","Chocolate","Coral","CornflowerBlue",
"Cornsilk","Crimson","Cyan","DarkBlue","DarkCyan","DarkGoldenRod","DarkGray","DarkGrey","DarkGreen","DarkKhaki","DarkMagenta","DarkOliveGreen",
"Darkorange","DarkOrchid","DarkRed","DarkSalmon","DarkSeaGreen","DarkSlateBlue","DarkSlateGray","DarkSlateGrey","DarkTurquoise",
"DarkViolet","DeepPink","DeepSkyBlue","DimGray","DimGrey","DodgerBlue","FireBrick","FloralWhite","ForestGreen","Fuchsia",
"Gainsboro","GhostWhite","Gold","GoldenRod","Gray","GreenYellow","HoneyDew","HotPink","IndianRed","Indigo",
"Ivory","Khaki","Lavender","LavenderBlush","LawnGreen","LemonChiffon","LightBlue","LightCoral","LightCyan","LightGoldenRodYellow",
"LightGray","LightGrey","LightGreen","LightPink","LightSalmon","LightSeaGreen","LightSkyBlue","LightSlateGray","LightSlateGrey",
"LightSteelBlue","LightYellow","Lime","LimeGreen","Linen","Magenta","Maroon","MediumAquaMarine","MediumBlue","MediumOrchid",
"MediumPurple","MediumSeaGreen","MediumSlateBlue","MediumSpringGreen","MediumTurquoise","MediumVioletRed","MidnightBlue",
"MintCream","MistyRose","Moccasin","NavajoWhite","Navy","OldLace","Olive","OliveDrab","Orange","OrangeRed","Orchid",
"PaleGoldenRod","PaleGreen","PaleTurquoise","PaleVioletRed","PapayaWhip","PeachPuff","Peru","Plum","PowderBlue",
"Purple","RosyBrown","RoyalBlue","SaddleBrown","Salmon","SandyBrown","SeaGreen","SeaShell","Sienna","Silver","SkyBlue",
"SlateBlue","SlateGray","SlateGrey","Snow","SpringGreen","SteelBlue","Tan","Teal","Thistle","Tomato","Turquoise","Violet",
"Wheat","White","WhiteSmoke","YellowGreen"};

int main(int argc, char **argv) {

    if (argc != 3) {
        printf("Usage: type (1,2) %s inputFile\n", argv[0]);
        exit(1);
    }

    int solver = atoi(argv[1]);

    if (solver == 1) {
        printf("Solver: Branch & Bound\n");
    } else {
        printf("Solver: Cut & Branch\n");
    }

    /* read graph input file
     * format: http://mat.gsia.cmu.edu/COLOR/instances.html
     * graph representation chosen in order to load the LP easily.
     * - vector of edges
     * - vector of partitions
```

```
     */
    FILE* fp = fopen(argv[2], "r");

    if (fp == NULL) {
        printf("Invalid input file.\n");
        exit(1);
    }

    char buf[100];
    int vertex_size, edge_size;

    set<pair<int,int> > edges; // sometimes we have to filter directed graphs

    while (fgets(buf, sizeof(buf), fp) != NULL) {
        if (buf[0] == 'c') continue;
        else if (buf[0] == 'p') {
            sscanf(&buf[7], "%d %d", &vertex_size, &edge_size);
        }
        else if (buf[0] == 'e') {
            int from, to;
            sscanf(&buf[2], "%d %d", &from, &to);
            if (from < to) {
                edges.insert(pair<int,int>(from, to));
            } else {
                edges.insert(pair<int,int>(to, from));
            }
        }
    }

    // build adyacency list
    edge_size = edges.size();
    int adyacency_size = edge_size*edge_size - ((edge_size+1)*edge_size/2);
    bool* adyacencyList = new bool[adyacency_size]; // can be optimized even more with a bitfield.
    fill_n(adyacencyList, adyacency_size, false);
    for (set<pair<int,int> >::iterator it = edges.begin(); it != edges.end(); ++it) {
        adyacencyList[fromMatrixToVector(it->first, it->second, edge_size)] = true;
    }

    // set random seed
    // srand(time(NULL));

    // asign every vertex to a partition
    int partition_size = rand() % vertex_size + 1;
    vector<vector<int> > partitions(partition_size, vector<int>());

    // warning: this procedure doesn't guarantee every partition will have an element.
    for (int i = 1; i <= vertex_size; ++i) {
        int assign_partition = rand() % partition_size;
        partitions[assign_partition].push_back(i);
    }

    // update partition_size
    for (std::vector<vector<int> >::iterator it = partitions.begin(); it != partitions.end(); ++it) {
        if (it->size() == 0) --partition_size;
    }

```

```
    printf("Graph: vertex_size: %d, edge_size: %d, partition_size: %d\n", vertex_size, edge_size, partition_size);

    // start loading LP using CPLEX
    int status;
    CPXENVptr env; // pointer to enviroment
    CPXLPptr lp;   // pointer to the lp.

    env = CPXopenCPLEX(&status); // create enviroment
    checkStatus(env, status);

    // create LP
    lp = CPXcreateprob(env, &status, "Instance of partitioned graph coloring.");
    checkStatus(env, status);

    setBranchAndBoundConfig(env);

    if (solver == 1) { // pure branch & bound
        loadObjectiveFunction(env, lp, vertex_size, partition_size, CPX_BINARY);
    } else {
        loadObjectiveFunction(env, lp, vertex_size, partition_size, CPX_CONTINUOUS);
    }

    loadAdyacencyColorRestriction(env, lp, vertex_size, edge_size, partition_size, adyacencyList);
    loadSingleColorInPartitionRestriction(env, lp, partitions, partition_size);
    loadAdyacencyColorRestriction(env, lp, vertex_size, partition_size);
    loadSymmetryBreaker(env, lp, partition_size);

    if (solver != 1) loadCuttingPlanes(env, lp, vertex_size, edge_size, partition_size, adyacencyList);

    // write LP formulation to file, great to debug.
    status = CPXwriteprob(env, lp, "graph.lp", NULL);
    checkStatus(env, status);

    convertVariableType(env, lp, vertex_size, partition_size, CPX_BINARY);

    solveLP(env, lp, edge_size, vertex_size, partition_size);

    delete[] adyacencyList;

    return 0;
}

int getVertexIndex(int id, int color, int partition_size) {
    return partition_size + ((id-1)*partition_size) + (color-1);
}

/* since the adyacency matrix is symmetric and the diagonal is not needed, we can simply
 * store the upper diagonal and get adyacency from a list. the math is quite simple, it
 * just uses the formula for the sum of integers. ids are numbered starting from 1.
 */
inline int fromMatrixToVector(int from, int to, int edge_size) {

    // for speed, many parts of this code are commented, since by our usage we always
    // know from < to and are in range.
```

```
    // assert(from != to && from <= edge_size && to <= edge_size);

    // if (from < to)
        return from*edge_size - (from+1)*from/2 - (edge_size - to) - 1;
    // else
    //  return to*edge_size - (to+1)*to/2 - (edge_size - from) - 1;
}

inline bool isAdyacent(int from, int to, int edge_size, bool* adyacencyList) {
    return adyacencyList[fromMatrixToVector(from, to, edge_size)];
}

bool adyacentToAll(int id, int edge_size, bool* adyacencyList, const set<int>& clique) {
    for (set<int>::iterator it = clique.begin(); it != clique.end(); ++it) {
        if (!isAdyacent(id, *it, edge_size, adyacencyList)) return false;
    }
    return true;
}

bool cliqueNotContained(const set<int>& clique, const set<set<int> >& clique_set) {
    for (set<set<int> >::iterator it = clique_set.begin(); it != clique_set.end(); ++it) {
        // by construction, sets are already ordered.
        if (includes(it->begin(), it->end(), clique.begin(), clique.end())) return false;
    }
    return true;
}

int loadObjectiveFunction(CPXENVptr& env, CPXLPptr& lp, int vertex_size, int partition_size, char vtype) {

    // load objective function
    int n = partition_size + (vertex_size*partition_size);
    double *objfun  = new double[n];
    double *ub      = new double[n];
    char     *ctype = new char[n];
    char **colnames = new char*[n];

    for (int i = 0; i < partition_size; ++i) {
        objfun[i] = 1;
        ub[i] = 1;
        ctype[i]    = vtype;
        colnames[i] = new char[10];
        sprintf(colnames[i], "w_%d", (i+1));
    }

    for (int id = 1; id <= vertex_size; ++id) {
        for (int color = 1; color <= partition_size; ++color) {
            int index = getVertexIndex(id, color, partition_size);
            objfun[index]   = 0;
            ub[index] = 1;
            ctype[index]    = vtype;
            colnames[index] = new char[10];
            sprintf(colnames[index], "x_%d%d", id, color);
        }
    }
```

```cpp

    // CPLEX bug? If you set ctype, it doesn't identify the problem as continous.
    int status = CPXnewcols(env, lp, n, objfun, NULL, ub, NULL, colnames);
    checkStatus(env, status);

    // free memory
    for (int i = 0; i < n; ++i) {
        delete[] colnames[i];
    }

    delete[] objfun;
    delete[] ub;
    delete[] ctype;
    delete[] colnames;

    return 0;
}

int loadAdyacencyColorRestriction(CPXENVptr& env, CPXLPptr& lp, int vertex_size, int edge_size, int partition_size, bool* adyacencyList) {

    // load first restriction
    int ccnt = 0;                               // new columns being added.
    int rcnt = edge_size * partition_size;      // new rows being added.
    int nzcnt = rcnt*2;                         // nonzero constraint coefficients being added.

    double *rhs = new double[rcnt];             // independent term in restrictions.
    char *sense = new char[rcnt];               // sense of restriction inequality.

    int *matbeg = new int[rcnt];                // array position where each restriction starts in matind and matval.
    int *matind = new int[rcnt*2];              // index of variables != 0 in restriction (each var has an index defined above)
    double *matval  = new double[rcnt*2];       // value corresponding to index in restriction.
    char **rownames = new char*[rcnt];          // row labels.

    int i = 0;
    for (int from = 1; from <= vertex_size; ++from) {
        for (int to = from + 1; to <= vertex_size; ++to) {

            if (!isAdyacent(from, to, edge_size, adyacencyList)) continue;

            for (int color = 1; color <= partition_size; ++color) {
                matbeg[i] = i*2;

                matind[i*2]   = getVertexIndex(from, color, partition_size);
                matind[i*2+1] = getVertexIndex(to  , color, partition_size);

                matval[i*2]   = 1;
                matval[i*2+1] = 1;

                rhs[i] = 1;
                sense[i] = 'L';
                rownames[i] = new char[40];
                sprintf(rownames[i], "%s", colors[color-1]);

                ++i;
```

```
            }
        }
    }

    // debug flag
    // status = CPXsetintparam(env, CPX_PARAM_DATACHECK, CPX_ON);

    // add restriction
    int status = CPXaddrows(env, lp, ccnt, rcnt, nzcnt, rhs, sense, matbeg, matind, matval, NULL, rownames);
    checkStatus(env, status);

    // free memory
    for (int i = 0; i < rcnt; ++i) {
        delete[] rownames[i];
    }

    delete[] rhs;
    delete[] sense;
    delete[] matbeg;
    delete[] matind;
    delete[] matval;
    delete[] rownames;

    return 0;
}


int loadSingleColorInPartitionRestriction(CPXENVptr& env, CPXLPptr& lp, vector<vector<int> >& partitions, int partition_size) {

    // load second restriction
    int p = 1;
    for (std::vector<vector<int> >::iterator it = partitions.begin(); it != partitions.end(); ++it) {

        int size = it->size();                  // current partition size.
        if (size == 0) continue;                // skip empty partitions.

        int ccnt = 0;                           // new columns being added.
        int rcnt = 1;                           // new rows being added.
        int nzcnt = size*partition_size;        // nonzero constraint coefficients being added.

        double *rhs = new double[rcnt];         // independent term in restrictions.
        char *sense = new char[rcnt];           // sense of restriction inequality.

        int *matbeg = new int[rcnt];            // array position where each restriction starts in matind and matval.
        int *matind = new int[nzcnt];           // index of variables != 0 in restriction (each var has an index defined above)
        double *matval = new double[nzcnt];     // value corresponding to index in restriction.
        char **rownames = new char*[rcnt];      // row labels.

        matbeg[0] = 0;
        sense[0]  = 'E';
        rhs[0]    = 1;
        rownames[0] = new char[40];
```

```
        sprintf(rownames[0], "partition_%d", p);

        int i = 0;
        for (std::vector<int>::iterator it2 = it->begin(); it2 != it->end(); ++it2) {
            for (int color = 1; color <= partition_size; ++color) {
                matind[i] = getVertexIndex(*it2, color, partition_size);
                matval[i] = 1;
                ++i;
            }
        }

        // add restriction
        int status = CPXaddrows(env, lp, ccnt, rcnt, nzcnt, rhs, sense, matbeg, matind, matval, NULL, rownames);
        checkStatus(env, status);

        // free memory
        delete[] rownames[0];
        delete[] rhs;
        delete[] sense;
        delete[] matbeg;
        delete[] matind;
        delete[] matval;
        delete[] rownames;

        ++p;
    }

    return 0;
}

int loadSymmetryBreaker(CPXENVptr& env, CPXLPptr& lp, int partition_size) {

    int ccnt = 0;                          // new columns being added.
    int rcnt = partition_size - 1;         // new rows being added.
    int nzcnt = 2*rcnt;                    // nonzero constraint coefficients being added.

    double* rhs = new double[rcnt];        // independent term in restrictions.
    char *sense = new char[rcnt];          // sense of restriction inequality.

    int *matbeg = new int[rcnt];           // array position where each restriction starts in matind and matval.
    int *matind = new int[rcnt*2];         // index of variables != 0 in restriction (each var has an index defined above)
    double *matval  = new double[rcnt*2];  // value corresponding to index in restriction.
    char **rownames = new char*[rcnt];     // row labels.

    int i = 0;
    for (int color = 0; color < partition_size - 1; ++color) {
        matbeg[i] = i*2;
        matind[i*2]   = color;
        matind[i*2+1] = color + 1;
        matval[i*2]   = -1;
        matval[i*2+1] = 1;

        rhs[i] = 0;
        sense[i] = 'L';
```

```
        rownames[i] = new char[40];
        sprintf(rownames[i], "%s", "symmetry_breaker");

        ++i;
    }


    // add restriction
    int status = CPXaddrows(env, lp, ccnt, rcnt, nzcnt, rhs, sense, matbeg, matind, matval, NULL, rownames);
    checkStatus(env, status);

    // free memory
    for (int i = 0; i < rcnt; ++i) {
        delete[] rownames[i];
    }

    delete[] rhs;
    delete[] sense;
    delete[] matbeg;
    delete[] matind;
    delete[] matval;
    delete[] rownames;

    return 0;
}

int loadCuttingPlanes(CPXENVptr& env, CPXLPptr& lp, int vertex_size, int edge_size, int partition_size, bool* adyacencyList) {

    printf("Finding Cutting Planes.\n");

    // calculate runtime
    double inittime, endtime;
    int status = CPXgettime(env, &inittime);

    int n = partition_size + (vertex_size*partition_size);

    set<set<int> > oddhole_familly;
    oddholeFamillyHeuristic(oddhole_familly, vertex_size, edge_size, partition_size, adyacencyList);

    set<set<int> > clique_familly;
    maximalCliqueFamillyHeuristic(clique_familly, vertex_size, edge_size, partition_size, adyacencyList);

    double *sol = new double[n];
    int iteration = 1;
    int unsatisfied_restrictions = 0;
    while (iteration <= CUTTING_PLANE_ITERATIONS) {

        printf("Iteration %d\n", iteration);

        // solve LP
        status = CPXlpopt(env, lp);
        checkStatus(env, status);

        status = CPXgetx(env, lp, sol, 0, n - 1);
        checkStatus(env, status);
```

```

        // for (int id = 1; id <= vertex_size; ++id) {
        //  for (int color = 1; color <= partition_size; ++color) {
        //      int index = getVertexIndex(id, color, partition_size);
        //      if (sol[index] == 0) continue;
        //      cout << "x_" << id << " " << color << " = " << sol[index] << endl;
        //  }
        // }

        // check which elements in the familly do not satisfy the inequality
        if (clique_familly.size() > 0) {
            unsatisfied_restrictions += findUnsatisfiedCliqueRestrictions(env, lp, clique_familly, vertex_size, partition_size, n, sol);
        }

        if (oddhole_familly.size() > 0) {
            unsatisfied_restrictions += findUnsatisfiedOddholeRestrictions(env, lp, oddhole_familly, vertex_size, partition_size, n, sol);
        }

        if (unsatisfied_restrictions == 0) break;

        unsatisfied_restrictions = 0;
        iteration++;
    }

    status = CPXgettime(env, &endtime);
    double elapsed_time = endtime-inittime;
    cout << "Time taken to add cutting planes: " << elapsed_time << endl;

    return 0;
}

int oddholeFamillyHeuristic(set<set<int> >& oddhole_familly, int vertex_size, int edge_size, int partition_size, bool* adyacencyList) {

    printf("Generating oddhole familly.\n");

    for (int id = 1; id <= vertex_size; ++id) {
        set<int> path;
        path.insert(id);
        for (int id2 = id + 1; id2 <= vertex_size; ++id2) {
            if (isAdyacent(*(--path.end()), id2, edge_size, adyacencyList)) {
                path.insert(id2);
            }
        }

        while (path.size() >= 3 && (path.size() % 2 == 0 ||
            !isAdyacent(*path.begin(), *(--path.end()), edge_size, adyacencyList))) {
            path.erase(--path.end());
        }

        if (path.size() >= 3 && isAdyacent(*path.begin(), *(--path.end()), edge_size, adyacencyList)) {
            oddhole_familly.insert(path);
        }
    }

    // print the familly
```

```
    // for (set<set<int> >::iterator it = oddhole_familly.begin(); it != oddhole_familly.end(); ++it) {
    //  cout << "Path: ";
    //  for (set<int>::iterator it2 = it->begin(); it2 != it->end(); ++it2) {
    //      cout << *it2 << " ";
    //  }
    //  cout << endl;
    // }

    int familly_size = oddhole_familly.size() * partition_size;

    printf("Familly generated (size: %d)\n", familly_size);

    return familly_size;
}

int findUnsatisfiedOddholeRestrictions(CPXENVptr& env, CPXLPptr& lp, set<set<int> >& oddhole_familly, int vertex_size, int partition_size, int n, double* sol) {

    int counter = 0;
    for (set<set<int> >::iterator it = oddhole_familly.begin(); it != oddhole_familly.end(); ++it) {

        for (int color = 1; color <= partition_size; ++color) {
            double sum = 0;
            for (set<int>::iterator it2 = it->begin(); it2 != it->end(); ++it2) {
                double coef = sol[getVertexIndex(*it2, color, partition_size)];
                sum += sol[getVertexIndex(*it2, color, partition_size)];
            }
            int k = (it->size() - 1) / 2;
            if (sum > k*sol[color-1]) {
                loadUnsatisfiedOddholeRestriction(env, lp, partition_size, *it, color);
                ++counter;
            }
        }
    }

    printf("%d unsatisfied oddhole restrictions found!\n", counter);

    return counter;
}

int loadUnsatisfiedOddholeRestriction(CPXENVptr& env, CPXLPptr& lp, int partition_size, const set<int>& path, int color) {

    int ccnt = 0;
    int rcnt = 1;
    int nzcnt = path.size() + 1;

    double rhs = 0;
    char sense = 'L';

    int matbeg = 0;
    int* matind    = new int[path.size() + 1];
    double* matval = new double[path.size() +1];
    char **rowname = new char*[rcnt];
    rowname[0] = new char[40];
    sprintf(rowname[0], "unsatisfied_oddhole");
```

```
    int k = (path.size() - 1) / 2;

    matind[0] = color - 1;
    matval[0] = -k;

    int i = 1;
    for (set<int>::iterator it = path.begin(); it != path.end(); ++it) {
        matind[i] = getVertexIndex(*it, color, partition_size);
        matval[i] = 1;
        ++i;
    }

    // add restriction
    int status = CPXaddrows(env, lp, ccnt, rcnt, nzcnt, &rhs, &sense, &matbeg, matind
        , matval, NULL, rowname);
    checkStatus(env, status);

    // free memory
    delete[] matind;
    delete[] matval;
    delete rowname[0];
    delete rowname;

    return 0;
}

int maximalCliqueFamillyHeuristic(set<set<int> >& clique_familly, int vertex_size,
    int edge_size, int partition_size, bool* adjacencyList) {

    printf("Generating clique familly.\n");

    for (int id = 1; id <= vertex_size; id++) {
        set<int> clique;
        clique.insert(id);
        for (int id2 = id + 1; id2 <= vertex_size; ++id2) {
            if (adyacentToAll(id2, edge_size, adjacencyList, clique)) {
                clique.insert(id2);
            }
        }
        if (clique.size() > 2) {
            if (cliqueNotContained(clique, clique_familly)) {
                clique_familly.insert(clique);
            }
        }
    }

    // print the familly
    // for (set<set<int> >::iterator it = clique_familly.begin(); it !=
        clique_familly.end(); ++it) {
    //  cout << "Clique: ";
    //  for (set<int>::iterator it2 = it->begin(); it2 != it->end(); ++it2) {
    //      cout << *it2 << " ";
    //  }
    //  cout << endl;
    // }

    int familly_size = clique_familly.size() * partition_size;

```

```
    printf("Familly generated (size: %d)\n", familly_size);

    return familly_size;
}

int findUnsatisfiedCliqueRestrictions(CPXENVptr& env, CPXLPptr& lp, set<set<int> >& clique_familly, int vertex_size, int partition_size, int n, double* sol) {

    int counter = 0;
    for (set<set<int> >::iterator it = clique_familly.begin(); it != clique_familly.end(); ++it) {

        for (int color = 1; color <= partition_size; ++color) {
            double sum = 0;
            for (set<int>::iterator it2 = it->begin(); it2 != it->end(); ++it2) {
                double coef = sol[getVertexIndex(*it2, color, partition_size)];
                sum += sol[getVertexIndex(*it2, color, partition_size)];
            }
            if (sum > sol[color-1]) {
                loadUnsatisfiedCliqueRestriction(env, lp, partition_size, *it, color);
                ++counter;
            }
        }
    }

    printf("%d unsatisfied clique restrictions found!\n", counter);

    return counter;
}

int loadUnsatisfiedCliqueRestriction(CPXENVptr& env, CPXLPptr& lp, int partition_size, const set<int>& clique, int color) {

    int ccnt = 0;
    int rcnt = 1;
    int nzcnt = clique.size() + 1;

    double rhs = 0;
    char sense = 'L';

    int matbeg = 0;
    int* matind    = new int[clique.size() + 1];
    double* matval = new double[clique.size() +1];
    char **rowname = new char*[rcnt];
    rowname[0] = new char[40];
    sprintf(rowname[0], "unsatisfied_clique");

    matind[0] = color - 1;
    matval[0] = -1;

    int i = 1;
    for (set<int>::iterator it = clique.begin(); it != clique.end(); ++it) {
        matind[i] = getVertexIndex(*it, color, partition_size);
        matval[i] = 1;
        ++i;
    }

    // add restriction
```

```
    int status = CPXaddrows(env, lp, ccnt, rcnt, nzcnt, &rhs, &sense, &matbeg, matind
        , matval, NULL, rowname);
    checkStatus(env, status);

    // free memory
    delete[] matind;
    delete[] matval;
    delete rowname[0];
    delete rowname;

    return 0;
}

int loadAdyacencyColorRestriction(CPXENVptr& env, CPXLPptr& lp, int vertex_size, int
    partition_size) {

    // load third restriction
    int ccnt = 0;                              // new columns being added.
    int rcnt = vertex_size * partition_size; // new rows being added.
    int nzcnt = rcnt*2;                        // nonzero constraint coefficients being
         added.

    double *rhs = new double[rcnt];           // independent term in restrictions.
    char *sense = new char[rcnt];             // sense of restriction inequality.

    int *matbeg = new int[rcnt];              // array position where each restriction
         starts in matind and matval.
    int *matind = new int[rcnt*2];            // index of variables != 0 in
        restriction (each var has an index defined above)
    double *matval  = new double[rcnt*2];     // value corresponding to index in
        restriction.
    char **rownames = new char*[rcnt];        // row labels.

    int i = 0;
    for (int v = 1; v <= vertex_size; ++v) {
        for (int color = 1; color <= partition_size; ++color) {
            matbeg[i] = i*2;

            matind[i*2]   = getVertexIndex(v, color, partition_size);
            matind[i*2+1] = color-1;

            matval[i*2]   = 1;
            matval[i*2+1] = -1;

            rhs[i] = 0;
            sense[i] = 'L';
            rownames[i] = new char[40];
            sprintf(rownames[i], "color_res");

            ++i;
        }
    }

    // add restriction
    int status = CPXaddrows(env, lp, ccnt, rcnt, nzcnt, rhs, sense, matbeg, matind,
        matval, NULL, rownames);
    checkStatus(env, status);

    // free memory
```

```
    for (int i = 0; i < rcnt; ++i) {
        delete[] rownames[i];
    }

    delete[] rhs;
    delete[] sense;
    delete[] matbeg;
    delete[] matind;
    delete[] matval;
    delete[] rownames;

    return 0;
}

int solveLP(CPXENVptr& env, CPXLPptr& lp, int edge_size, int vertex_size, int
    partition_size) {

    printf("\nSolving MIP.\n");

    int n = partition_size + (vertex_size*partition_size); // amount of total
        variables

    // calculate runtime
    double inittime, endtime;
    int status = CPXgettime(env, &inittime);
    checkStatus(env, status);

    // solve LP
    status = CPXmipopt(env, lp);
    checkStatus(env, status);

    status = CPXgettime(env, &endtime);
    checkStatus(env, status);

    // check solution state
    int solstat;
    char statstring[510];
    CPXCHARptr p;
    solstat = CPXgetstat(env, lp);
    p = CPXgetstatstring(env, solstat, statstring);
    string statstr(statstring);
    if (solstat != CPXMIP_OPTIMAL && solstat != CPXMIP_OPTIMAL_TOL &&
        solstat != CPXMIP_NODE_LIM_FEAS && solstat != CPXMIP_TIME_LIM_FEAS) {
        // printf("Optimization failed.\n");
        cout << "Optimization failed: " << solstat << endl;
        exit(1);
    }

    double objval;
    status = CPXgetobjval(env, lp, &objval);
    checkStatus(env, status);

    // get values of all solutions
    double *sol = new double[n];
    status = CPXgetx(env, lp, sol, 0, n - 1);
    checkStatus(env, status);

    // write solutions to current window
    cout << "Optimization result: " << statstring << endl;
```

```cpp
    cout << "Time taken to solve final LP: " << (endtime - inittime) << endl;
    cout << "Colors used: " << objval << endl;
    for (int color = 1; color <= partition_size; ++color) {
        if (sol[color-1] == 1) {
            cout << "w_" << color << " = " << sol[color-1] << " (" << colors[color-1] << ")" << endl;
        }
    }

    for (int id = 1; id <= vertex_size; ++id) {
        for (int color = 1; color <= partition_size; ++color) {
            int index = getVertexIndex(id, color, partition_size);
            if (sol[index] == 1) {
                cout << "x_" << id << " = " << colors[color-1] << endl;
            }
        }
    }

    delete[] sol;

    return 0;
}

int convertVariableType(CPXENVptr& env, CPXLPptr& lp, int vertex_size, int partition_size, char vtype) {

    int n = partition_size + (vertex_size*partition_size);
    int* indices = new int[n];
    char* xctype = new char[n];

    for (int i = 0; i < n; i++) {
        indices[i] = i;
        xctype[i]  = vtype;
    }
    CPXchgctype(env, lp, n, indices, xctype);

    delete[] indices;
    delete[] xctype;

    return 0;
}

int setBranchAndBoundConfig(CPXENVptr& env) {

    // CPLEX config
    // http://www-01.ibm.com/support/knowledgecenter/SSSA5P_12.2.0/ilog.odms.cplex.help/Content/Optimization/Documentation/CPLEX/_pubskel/CPLEX916.html

    // deactivate pre-processing
    CPXsetintparam(env, CPX_PARAM_PRESLVND, -1);
    CPXsetintparam(env, CPX_PARAM_REPEATPRESOLVE, 0);
    CPXsetintparam(env, CPX_PARAM_RELAXPREIND, 0);
    CPXsetintparam(env, CPX_PARAM_REDUCE, 0);
    CPXsetintparam(env, CPX_PARAM_LANDPCUTS, -1);

    // maximize objective function
    // CPXchgobjsen(env, lp, CPX_MAX);

    // enable/disable screen output
```

```
    CPXsetintparam(env, CPX_PARAM_SCRIND, CPX_OFF);

    // set excecution limit
    CPXsetdblparam(env, CPX_PARAM_TILIM, 3600);

    // disable presolve
    // CPXsetintparam(env, CPX_PARAM_PREIND, CPX_OFF);

    // enable traditional branch and bound
    CPXsetintparam(env, CPX_PARAM_MIPSEARCH, CPX_MIPSEARCH_TRADITIONAL);

    // use only one thread for experimentation
    CPXsetintparam(env, CPX_PARAM_THREADS, 1);

    // do not add cutting planes
    CPXsetintparam(env, CPX_PARAM_EACHCUTLIM, CPX_OFF);

    // disable gomory fractional cuts
    CPXsetintparam(env, CPX_PARAM_FRACCUTS, -1);

    // measure time in CPU time
    // CPXsetintparam(env, CPX_PARAM_CLOCKTYPE, CPX_ON);

    return 0;
}


int checkStatus(CPXENVptr& env, int status) {
    if (status) {
        char buffer[100];
        CPXgeterrorstring(env, status, buffer);
        printf("%s\n", buffer);
        exit(1);
    }
    return 0;
}
```