

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	3
1.1. Funcion objetivo	3
1.2. Restricciones	3
1.3. Eliminacion de simetrias	3
2. Branch & Bound	4
3. Desigualdades	4
3.1. Desigualdad de Clique	4
3.2. Desigualdad de Aujero Impar	5
3.3. Planos de Corte	5
3.4. Heurísticas	5
3.4.1. Heurística de Separacion para Clique Maximal	6
3.4.2. Heurística de Separacion para Aujero Impar	6
4. Cut & Branch	7
5. Experimentacion	8
6. Conclusion	9
7. Apéndice A: Código	9
7.1. coloring.cpp	9

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 utilice 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, \dots, 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:

$$\min \sum_{j \in C} w_j \quad (1)$$

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.

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

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.

$$\sum_{i \in V_p} \sum_{j \in C} x_{ij} = 1 \quad \forall p \in P \quad (3)$$

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

$$x_{ij} \leq w_j \quad \forall i \in V, \forall j \in C \quad (4)$$

Integralidad y positividad de las variables:

$$x_{ij} \in \{0, 1\} \quad \forall i \in V, \forall j \in C \quad (5)$$

$$w_j \in \{0, 1\} \quad \forall j \in C \quad (6)$$

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:

$$w_j \geq w_{j+1} \quad \forall 1 \leq j \leq |C| \quad (7)$$

2. Branch & Bound

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

3. Desigualdades

3.1. Desigualdad de Clique

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

$$\sum_{p \in K} x_{pj_0} \leq w_{j_0} \quad (8)$$

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 tamaño 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:

$$\begin{aligned} \blacksquare x_{1j_0} + x_{2j_0} &\leq 1 \\ \blacksquare x_{2j_0} + x_{3j_0} &\leq 1 \\ \blacksquare x_{1j_0} + x_{3j_0} &\leq 1 \end{aligned}$$

Multiplicando todas estas desigualdades por $1/3$ y sumando entonces:

$$1/3(x_{1j_0} + x_{2j_0}) + 1/3(x_{2j_0} + x_{3j_0}) + 1/3(x_{1j_0} + x_{3j_0}) \leq 1$$

Como x_{ij} toma valores enteros, entonces: $1/3(x_{1j_0} + x_{2j_0}) + 1/3(x_{2j_0} + x_{3j_0}) + 1/3(x_{1j_0} + x_{3j_0}) \leq 1$

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:

$$x_{1j_0} + x_{nj_0} \leq 1, x_{2j_0} + x_{nj_0} \leq 1, \dots, x_{(n-1)j_0} + x_{nj_0} \leq 1$$

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

$$\sum_{p \in C_{2k+1}} x_{pj_0} \leq kw_{j_0} \quad (9)$$

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 agujero estara en una particion diferente. Aqui nuevamente tenemos dos casos:

- Si no se asigna el color j_0 a algun vertice del agujero, 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. \square

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 heurísticos. Los algoritmos heurísticos, 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 heurístico, 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 heurísticas que encuentran desigualdades validas pueden fallar y el algoritmo no puede continuar.

3.4. Heurísticas

Las heurísticas 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 heurísticas 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

```
1: procedure GENERATECLIQUEFAMILY( $V, E$ )
2:    $set \leftarrow set \langle int \rangle \rangle clique\_family$ 
3:   for  $id \leftarrow 1, |V|$  do
4:      $set \leftarrow int \rangle clique$ 
5:     clique.insert(id)
6:     for  $id2 \leftarrow id + 1, |V|$  do
7:       if clique.adjacentToAll(id2) then
8:         clique.insert(id2)
9:       end if
10:    end for
11:    if  $\neg clique\_family.isContained(clique)$  then
12:      clique_family.insert(clique)
13:    end if
14:  end for
15: end procedure
```

Notar que en la practica solo consideramos cliques de tamano 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

```
1: procedure GENERATEODDholeFAMILY( $V, E$ )
2:    $set \leftarrow set \langle int \rangle \rangle oddhole\_family$ 
3:   for  $id \leftarrow 1, |V|$  do
4:      $set \leftarrow int \rangle path$ 
5:     path.insert(id)
6:     for  $id2 \leftarrow id + 1, |V|$  do
7:       if isAdjacent(path.end, id2) then
8:         path.insert(id2)
9:       end if
10:    end for
11:    while path.size()  $\geq 3$  and (path.size() mod 2 == 0 or  $\neg isAdjacent(path.start, path.end)$ ) do
12:      path.erase(path.end)
13:    end while
14:    if path.size()  $\geq 3$  and isAdjacent(path.start, path.end) then
15:      oddhole_family.insert(path)
16:    end if
17:  end for
18: end procedure
```

4. Cut & Branch

5. Experimentacion

6. Conclusion

7. Apéndice A: Código

7.1. coloring.cpp

```
1 #include <ilcplex/ilocplex.h>
2 #include <ilcplex/cplex.h>
3
4 #include <stdlib.h>
5 #include <cassert>
6
7 #include <algorithm>
8 #include <string>
9 #include <vector>
10 #include <set>
11
12 #define TOL 1e-05
13 #define CUTTING_PLANE_ITERATIONS 1
14
15 ILOSTLBEGIN // macro to define namespace
16
17 // helper functions
18 int getVertexIndex(int id, int color, int partition_size);
19 inline int fromMatrixToVector(int from, int to, int edge_size);
20 inline bool isAdjacent(int from, int to, int edge_size, bool* adjacencyList);
21 bool adjacentToAll(int id, int edge_size, bool* adjacencyList, const set<int>& clique
22 );
23 bool cliqueNotContained(const set<int>& clique, const set<set<int>>& clique_set);
24
25 // load LP
26 int loadObjectiveFunction(CPXENVptr& env, CPXLPptr& lp, int vertex_size, int
27 partition_size, char vtype);
28 int loadAdjacencyColorRestriction(CPXENVptr& env, CPXLPptr& lp, int vertex_size, int
29 edge_size, int partition_size, bool* adjacencyList);
30 int loadSingleColorInPartitionRestriction(CPXENVptr& env, CPXLPptr& lp, vector<vector
31 <int>>& partitions, int partition_size);
32 int loadAdjacencyColorRestriction(CPXENVptr& env, CPXLPptr& lp, int vertex_size, int
33 partition_size);
34 int loadSymmetryBreaker(CPXENVptr& env, CPXLPptr& lp, int partition_size);
35
36 // cutting planes
37 int loadCuttingPlanes(CPXENVptr& env, CPXLPptr& lp, int vertex_size, int edge_size,
38 int partition_size, bool* adjacencyList);
39
40 int maximalCliqueFamilyHeuristic(set<set<int>>& clique_familly, int vertex_size,
41 int edge_size, int partition_size, bool* adjacencyList);
42 int findUnsatisfiedCliqueRestrictions(CPXENVptr& env, CPXLPptr& lp, set<set<int>>&
43 clique_familly, int vertex_size, int partition_size, int n, double* sol);
44 int loadUnsatisfiedCliqueRestriction(CPXENVptr& env, CPXLPptr& lp, int partition_size
45 , const set<int>& clique, int color);
46
47 int oddholeFamilyHeuristic(set<set<int>>& oddhole_familly, int vertex_size, int
48 edge_size, int partition_size, bool* adjacencyList);
49 int findUnsatisfiedOddholeRestrictions(CPXENVptr& env, CPXLPptr& lp, set<set<int>>&
50 oddhole_familly, int vertex_size, int partition_size, int n, double* sol);
51 int loadUnsatisfiedOddholeRestriction(CPXENVptr& env, CPXLPptr& lp, int
52 partition_size, const set<int>& path, int color);
53
```

```

42 // cplex functions
43 int solveLP(CPXENVptr& env, CPXLPptr& lp, int edge_size, int vertex_size, int
    partition_size);
44 int convertVariableType(CPXENVptr& env, CPXLPptr& lp, int vertex_size, int
    partition_size, char vtype);
45 int setBranchAndBoundConfig(CPXENVptr& env);
46 int checkStatus(CPXENVptr& env, int status);
47
48 // colors array!
49 const char* colors[] = {"Blue", "Red", "Green", "Yellow", "Grey", "Green", "Pink", "
    AliceBlue", "AntiqueWhite", "Aqua", "Aquamarine", "Azure", "Beige",
50 "Bisque", "Black", "BlanchedAlmond", "BlueViolet", "Brown", "BurlyWood", "CadetBlue", "
    Chartreuse", "Chocolate", "Coral", "CornflowerBlue",
51 "Cornsilk", "Crimson", "Cyan", "DarkBlue", "DarkCyan", "DarkGoldenRod", "DarkGray", "
    DarkGrey", "DarkGreen", "DarkKhaki", "DarkMagenta", "DarkOliveGreen",
52 "Darkorange", "DarkOrchid", "DarkRed", "DarkSalmon", "DarkSeaGreen", "DarkSlateBlue", "
    DarkSlateGray", "DarkSlateGrey", "DarkTurquoise",
53 "DarkViolet", "DeepPink", "DeepSkyBlue", "DimGray", "DimGrey", "DodgerBlue", "FireBrick", "
    FloralWhite", "ForestGreen", "Fuchsia",
54 "Gainsboro", "GhostWhite", "Gold", "GoldenRod", "Gray", "GreenYellow", "HoneyDew", "HotPink",
    "IndianRed", "Indigo",
55 "Ivory", "Khaki", "Lavender", "LavenderBlush", "LawnGreen", "LemonChiffon", "LightBlue", "
    LightCoral", "LightCyan", "LightGoldenRodYellow",
56 "LightGray", "LightGrey", "LightGreen", "LightPink", "LightSalmon", "LightSeaGreen", "
    LightSkyBlue", "LightSlateGray", "LightSlateGrey",
57 "LightSteelBlue", "LightYellow", "Lime", "LimeGreen", "Linen", "Magenta", "Maroon", "
    MediumAquaMarine", "MediumBlue", "MediumOrchid",
58 "MediumPurple", "MediumSeaGreen", "MediumSlateBlue", "MediumSpringGreen", "
    MediumTurquoise", "MediumVioletRed", "MidnightBlue",
59 "MintCream", "MistyRose", "Moccasin", "NavajoWhite", "Navy", "OldLace", "Olive", "OliveDrab",
    "Orange", "OrangeRed", "Orchid",
60 "PaleGoldenRod", "PaleGreen", "PaleTurquoise", "PaleVioletRed", "PapayaWhip", "PeachPuff",
    "Peru", "Plum", "PowderBlue",
61 "Purple", "RosyBrown", "RoyalBlue", "SaddleBrown", "Salmon", "SandyBrown", "SeaGreen", "
    SeaShell", "Sienna", "Silver", "SkyBlue",
62 "SlateBlue", "SlateGray", "SlateGrey", "Snow", "SpringGreen", "SteelBlue", "Tan", "Teal", "
    Thistle", "Tomato", "Turquoise", "Violet",
63 "Wheat", "White", "WhiteSmoke", "YellowGreen"};
64
65 int main(int argc, char **argv) {
66
67     if (argc != 3) {
68         printf("Usage: type (1,2) %s inputFile\n", argv[0]);
69         exit(1);
70     }
71
72     int solver = atoi(argv[1]);
73
74     if (solver == 1) {
75         printf("Solver: Branch & Bound\n");
76     } else {
77         printf("Solver: Cut & Branch\n");
78     }
79
80     /* read graph input file
81     * format: http://mat.gsia.cmu.edu/COLOR/instances.html
82     * graph representation chosen in order to load the LP easily.
83     * - vector of edges
84     * - vector of partitions

```

```

85     */
86     FILE* fp = fopen(argv[2], "r");
87
88     if (fp == NULL) {
89         printf("Invalid input file.\n");
90         exit(1);
91     }
92
93     char buf[100];
94     int vertex_size, edge_size;
95
96     set<pair<int,int>> edges; // sometimes we have to filter directed graphs
97
98     while (fgets(buf, sizeof(buf), fp) != NULL) {
99         if (buf[0] == 'c') continue;
100        else if (buf[0] == 'p') {
101            sscanf(&buf[7], "%d %d", &vertex_size, &edge_size);
102        }
103        else if (buf[0] == 'e') {
104            int from, to;
105            sscanf(&buf[2], "%d %d", &from, &to);
106            if (from < to) {
107                edges.insert(pair<int,int>(from, to));
108            } else {
109                edges.insert(pair<int,int>(to, from));
110            }
111        }
112    }
113
114    // build adjacency list
115    edge_size = edges.size();
116    int adjacency_size = edge_size*edge_size - ((edge_size+1)*edge_size/2);
117    bool* adjacencyList = new bool[adjacency_size]; // can be optimized even more
118    // with a bitfield.
119    fill_n(adjacencyList, adjacency_size, false);
120    for (set<pair<int,int>>::iterator it = edges.begin(); it != edges.end(); ++it) {
121        adjacencyList[fromMatrixToVector(it->first, it->second, edge_size)] = true;
122    }
123
124    // set random seed
125    // srand(time(NULL));
126
127    // assign every vertex to a partition
128    int partition_size = rand() % vertex_size + 1;
129    vector<vector<int>> partitions(partition_size, vector<int>());
130
131    // warning: this procedure doesn't guarantee every partition will have an element
132
133    for (int i = 1; i <= vertex_size; ++i) {
134        int assign_partition = rand() % partition_size;
135        partitions[assign_partition].push_back(i);
136    }
137
138    // update partition_size
139    for (std::vector<vector<int>>::iterator it = partitions.begin(); it !=
140        partitions.end(); ++it) {
141        if (it->size() == 0) --partition_size;
142    }

```

```

141     printf("Graph: vertex_size: %d, edge_size: %d, partition_size: %d\n", vertex_size
        , edge_size, partition_size);
142
143     // start loading LP using CPLEX
144     int status;
145     CPXENVptr env; // pointer to enviroment
146     CPXLPptr lp;   // pointer to the lp.
147
148     env = CPXopenCPLEX(&status); // create enviroment
149     checkStatus(env, status);
150
151     // create LP
152     lp = CPXcreateprob(env, &status, "Instance of partitioned graph coloring.");
153     checkStatus(env, status);
154
155     setBranchAndBoundConfig(env);
156
157     if (solver == 1) { // pure branch & bound
158         loadObjectiveFunction(env, lp, vertex_size, partition_size, CPX_BINARY);
159     } else {
160         loadObjectiveFunction(env, lp, vertex_size, partition_size, CPX_CONTINUOUS);
161     }
162
163     loadAdyacencyColorRestriction(env, lp, vertex_size, edge_size, partition_size,
        adyacencyList);
164     loadSingleColorInPartitionRestriction(env, lp, partitions, partition_size);
165     loadAdyacencyColorRestriction(env, lp, vertex_size, partition_size);
166     loadSymmetryBreaker(env, lp, partition_size);
167
168     if (solver != 1) loadCuttingPlanes(env, lp, vertex_size, edge_size,
        partition_size, adyacencyList);
169
170     // write LP formulation to file, great to debug.
171     status = CPXwriteprob(env, lp, "graph.lp", NULL);
172     checkStatus(env, status);
173
174     convertVariableType(env, lp, vertex_size, partition_size, CPX_BINARY);
175
176     solveLP(env, lp, edge_size, vertex_size, partition_size);
177
178     delete [] adyacencyList;
179
180     return 0;
181 }
182
183 int getVertexIndex(int id, int color, int partition_size) {
184     return partition_size + ((id-1)*partition_size) + (color-1);
185 }
186
187 /* since the adyacency matrix is symmetric and the diagonal is not needed, we can
    simply
188 * store the upper diagonal and get adyacency from a list. the math is quite simple,
    it
189 * just uses the formula for the sum of integers. ids are numbered starting from 1.
190 */
191 inline int fromMatrixToVector(int from, int to, int edge_size) {
192
193     // for speed, many parts of this code are commented, since by our usage we always
194     // know from < to and are in range.

```

```

195
196 // assert(from != to && from <= edge_size && to <= edge_size);
197
198 // if (from < to)
199     return from*edge_size - (from+1)*from/2 - (edge_size - to) - 1;
200 // else
201 //     return to*edge_size - (to+1)*to/2 - (edge_size - from) - 1;
202 }
203
204 inline bool isAdyacent(int from, int to, int edge_size, bool* adjacencyList) {
205     return adjacencyList[fromMatrixToVector(from, to, edge_size)];
206 }
207
208 bool adjacentToAll(int id, int edge_size, bool* adjacencyList, const set<int>& clique
209 ) {
210     for (set<int>::iterator it = clique.begin(); it != clique.end(); ++it) {
211         if (!isAdyacent(id, *it, edge_size, adjacencyList)) return false;
212     }
213     return true;
214 }
215
216 bool cliqueNotContained(const set<int>& clique, const set<set<int>>& clique_set) {
217     for (set<set<int>>::iterator it = clique_set.begin(); it != clique_set.end(); ++
218         it) {
219         // by construction, sets are already ordered.
220         if (includes(it->begin(), it->end(), clique.begin(), clique.end())) return
221             false;
222     }
223     return true;
224 }
225
226 int loadObjectiveFunction(CPXENVptr& env, CPXLPtr& lp, int vertex_size, int
227 partition_size, char vtype) {
228
229     // load objective function
230     int n = partition_size + (vertex_size*partition_size);
231     double *objfun = new double[n];
232     double *ub = new double[n];
233     char *ctype = new char[n];
234     char **colnames = new char*[n];
235
236     for (int i = 0; i < partition_size; ++i) {
237         objfun[i] = 1;
238         ub[i] = 1;
239         ctype[i] = vtype;
240         colnames[i] = new char[10];
241         sprintf(colnames[i], "w-%d", (i+1));
242     }
243
244     for (int id = 1; id <= vertex_size; ++id) {
245         for (int color = 1; color <= partition_size; ++color) {
246             int index = getVertexIndex(id, color, partition_size);
247             objfun[index] = 0;
248             ub[index] = 1;
249             ctype[index] = vtype;
250             colnames[index] = new char[10];
251             sprintf(colnames[index], "x-%d%d", id, color);
252         }
253     }
254 }

```

```

250
251 // CPLEX bug? If you set ctype, it doesn't identify the problem as continuous.
252 int status = CPXnewcols(env, lp, n, objfun, NULL, ub, NULL, colnames);
253 checkStatus(env, status);
254
255 // free memory
256 for (int i = 0; i < n; ++i) {
257     delete[] colnames[i];
258 }
259
260 delete[] objfun;
261 delete[] ub;
262 delete[] ctype;
263 delete[] colnames;
264
265 return 0;
266 }
267
268 int loadAdjacencyColorRestriction(CPXENVptr& env, CPXLPptr& lp, int vertex_size, int
edge_size, int partition_size, bool* adjacencyList) {
269
270 // load first restriction
271 int ccnt = 0; // new columns being added.
272 int rcnt = edge_size * partition_size; // new rows being added.
273 int nzcnt = rcnt*2; // nonzero constraint coefficients being
added.
274
275 double *rhs = new double[rcnt]; // independent term in restrictions.
276 char *sense = new char[rcnt]; // sense of restriction inequality.
277
278 int *matbeg = new int[rcnt]; // array position where each restriction
starts in matind and matval.
279 int *matind = new int[rcnt*2]; // index of variables != 0 in restriction
(each var has an index defined above)
280 double *matval = new double[rcnt*2]; // value corresponding to index in
restriction.
281 char **rownames = new char*[rcnt]; // row labels.
282
283 int i = 0;
284 for (int from = 1; from <= vertex_size; ++from) {
285     for (int to = from + 1; to <= vertex_size; ++to) {
286
287         if (!isAdjacent(from, to, edge_size, adjacencyList)) continue;
288
289         for (int color = 1; color <= partition_size; ++color) {
290             matbeg[i] = i*2;
291
292             matind[i*2] = getVertexIndex(from, color, partition_size);
293             matind[i*2+1] = getVertexIndex(to, color, partition_size);
294
295             matval[i*2] = 1;
296             matval[i*2+1] = 1;
297
298             rhs[i] = 1;
299             sense[i] = 'L';
300             rownames[i] = new char[40];
301             sprintf(rownames[i], "%s", colors[color-1]);
302
303             ++i;

```

```

304     }
305 }
306 }
307
308 // debug flag
309 // status = CPXsetintparam(env, CPXPARAMDATACHECK, CPX_ON);
310
311 // add restriction
312 int status = CPXaddrows(env, lp, ccnt, rcnt, nzcnt, rhs, sense, matbeg, matind,
313     matval, NULL, rownames);
314 checkStatus(env, status);
315
316 // free memory
317 for (int i = 0; i < rcnt; ++i) {
318     delete [] rownames[i];
319 }
320
321 delete [] rhs;
322 delete [] sense;
323 delete [] matbeg;
324 delete [] matind;
325 delete [] matval;
326 delete [] rownames;
327
328 return 0;
329 }
330
331 int loadSingleColorInPartitionRestriction(CPXENVptr& env, CPXLPtr& lp, vector<vector
<int>>& partitions, int partition_size) {
332
333     // load second restriction
334     int p = 1;
335     for (std::vector<vector<int>> >::iterator it = partitions.begin(); it !=
partitions.end(); ++it) {
336
337         int size = it->size(); // current partition size.
338         if (size == 0) continue; // skip empty partitions.
339
340         int ccnt = 0; // new columns being added.
341         int rcnt = 1; // new rows being added.
342         int nzcnt = size*partition_size; // nonzero constraint coefficients
being added.
343
344         double *rhs = new double[rcnt]; // independent term in restrictions.
345         char *sense = new char[rcnt]; // sense of restriction inequality.
346
347         int *matbeg = new int[rcnt]; // array position where each
restriction starts in matind and matval.
348         int *matind = new int[nzcnt]; // index of variables != 0 in
restriction (each var has an index defined above)
349         double *matval = new double[nzcnt]; // value corresponding to index in
restriction.
350         char **rownames = new char*[rcnt]; // row labels.
351
352         matbeg[0] = 0;
353         sense[0] = 'E';
354         rhs[0] = 1;
355         rownames[0] = new char[40];

```

```

356     sprintf(rownames[0], "partition_ %d", p);
357
358     int i = 0;
359     for (std::vector<int>::iterator it2 = it->begin(); it2 != it->end(); ++it2) {
360         for (int color = 1; color <= partition_size; ++color) {
361             matind[i] = getVertexIndex(*it2, color, partition_size);
362             matval[i] = 1;
363             ++i;
364         }
365     }
366
367     // add restriction
368     int status = CPXaddrows(env, lp, cnt, rcnt, nzcnt, rhs, sense, matbeg,
369                             matind, matval, NULL, rownames);
370     checkStatus(env, status);
371
372     // free memory
373     delete[] rownames[0];
374     delete[] rhs;
375     delete[] sense;
376     delete[] matbeg;
377     delete[] matind;
378     delete[] matval;
379     delete[] rownames;
380
381     ++p;
382 }
383
384 return 0;
385 }
386
387 int loadSymmetryBreaker(CPXENVptr& env, CPXLPptr& lp, int partition_size) {
388     int cnt = 0; // new columns being added.
389     int rcnt = partition_size - 1; // new rows being added.
390     int nzcnt = 2*rcnt; // nonzero constraint coefficients being
391                         // added.
392     double* rhs = new double[rcnt]; // independent term in restrictions.
393     char *sense = new char[rcnt]; // sense of restriction inequality.
394
395     int *matbeg = new int[rcnt]; // array position where each restriction
396                                 // starts in matind and matval.
397     int *matind = new int[rcnt*2]; // index of variables != 0 in restriction
398                                 // (each var has an index defined above)
399     double *matval = new double[rcnt*2]; // value corresponding to index in
400                                 // restriction.
401     char **rownames = new char*[rcnt]; // row labels.
402
403     int i = 0;
404     for (int color = 0; color < partition_size - 1; ++color) {
405         matbeg[i] = i*2;
406         matind[i*2] = color;
407         matind[i*2+1] = color + 1;
408         matval[i*2] = -1;
409         matval[i*2+1] = 1;
410
411         rhs[i] = 0;
412         sense[i] = 'L';
413     }

```



```

410         rownames[i] = new char[40];
411         sprintf(rownames[i], "%s", "symmetry_breaker");
412
413         ++i;
414     }
415
416
417     // add restriction
418     int status = CPXaddrows(env, lp, ccnt, rcnt, nzcnt, rhs, sense, matbeg, matind,
419                             matval, NULL, rownames);
420     checkStatus(env, status);
421
422     // free memory
423     for (int i = 0; i < rcnt; ++i) {
424         delete [] rownames[i];
425     }
426
427     delete [] rhs;
428     delete [] sense;
429     delete [] matbeg;
430     delete [] matind;
431     delete [] matval;
432     delete [] rownames;
433
434     return 0;
435 }
436
437 int loadCuttingPlanes(CPXENVptr& env, CPXLPptr& lp, int vertex_size, int edge_size,
438                      int partition_size, bool* adjacencyList) {
439
440     printf("Finding Cutting Planes.\n");
441
442     // calculate runtime
443     double inittime, endtime;
444     int status = CPXgettime(env, &inittime);
445
446     int n = partition_size + (vertex_size*partition_size);
447
448     set<set<int>> oddhole_familly;
449     oddholeFamillyHeuristic(oddhole_familly, vertex_size, edge_size, partition_size,
450                             adjacencyList);
451
452     set<set<int>> clique_familly;
453     maximalCliqueFamillyHeuristic(clique_familly, vertex_size, edge_size,
454                                   partition_size, adjacencyList);
455
456     double *sol = new double[n];
457     int iteration = 1;
458     int unsatisfied_restrictions = 0;
459     while (iteration <= CUTTING_PLANE_ITERATIONS) {
460
461         printf("Iteration %d\n", iteration);
462
463         // solve LP
464         status = CPXlpopt(env, lp);
465         checkStatus(env, status);
466
467         status = CPXgetx(env, lp, sol, 0, n - 1);
468         checkStatus(env, status);

```

```

465
466 // for (int id = 1; id <= vertex_size; ++id) {
467 //     for (int color = 1; color <= partition_size; ++color) {
468 //         int index = getVertexIndex(id, color, partition_size);
469 //         if (sol[index] == 0) continue;
470 //         cout << "x_" << id << " " << color << " = " << sol[index] << endl;
471 //     }
472 // }
473
474 // check which elements in the family do not satisfy the inequality
475 if (clique_family.size() > 0) {
476     unsatisfied_restrictions += findUnsatisfiedCliqueRestrictions(env, lp,
477         clique_family, vertex_size, partition_size, n, sol);
478 }
479
480 if (odddhole_family.size() > 0) {
481     unsatisfied_restrictions += findUnsatisfiedOddholeRestrictions(env, lp,
482         oddhole_family, vertex_size, partition_size, n, sol);
483 }
484
485 if (unsatisfied_restrictions == 0) break;
486
487 unsatisfied_restrictions = 0;
488 iteration++;
489 }
490
491 status = CPXgettime(env, &endtime);
492 double elapsed_time = endtime - inittime;
493 cout << "Time taken to add cutting planes: " << elapsed_time << endl;
494
495 return 0;
496 }
497
498 int oddholeFamilyHeuristic(set<set<int>> & oddhole_family, int vertex_size, int
499 edge_size, int partition_size, bool* adjacencyList) {
500
501     printf("Generating oddhole family.\n");
502
503     for (int id = 1; id <= vertex_size; ++id) {
504         set<int> path;
505         path.insert(id);
506         for (int id2 = id + 1; id2 <= vertex_size; ++id2) {
507             if (isAdjacent(*(--path.end()), id2, edge_size, adjacencyList)) {
508                 path.insert(id2);
509             }
510         }
511
512         while (path.size() >= 3 && (path.size() % 2 == 0 ||
513             !isAdjacent(*path.begin(), *(--path.end()), edge_size, adjacencyList))) {
514             path.erase(--path.end());
515         }
516
517         if (path.size() >= 3 && isAdjacent(*path.begin(), *(--path.end()), edge_size,
518             adjacencyList)) {
519             oddhole_family.insert(path);
520         }
521     }
522
523     // print the family

```

```

520 // for (set<set<int> >::iterator it = oddhole_familly.begin(); it !=
521 //      oddhole_familly.end(); ++it) {
522 //     cout << "Path: ";
523 //     for (set<int>::iterator it2 = it->begin(); it2 != it->end(); ++it2) {
524 //         cout << *it2 << " ";
525 //     }
526 //     cout << endl;
527 // }
528 int familly_size = oddhole_familly.size() * partition_size;
529
530 printf("Familly generated (size: %d)\n", familly_size);
531
532 return familly_size;
533 }
534
535 int findUnsatisfiedOddholeRestrictions(CPXENVptr& env, CPXLPptr& lp, set<set<int> >&
536 oddhole_familly, int vertex_size, int partition_size, int n, double* sol) {
537
538     int counter = 0;
539     for (set<set<int> >::iterator it = oddhole_familly.begin(); it != oddhole_familly
540         .end(); ++it) {
541
542         for (int color = 1; color <= partition_size; ++color) {
543             double sum = 0;
544             for (set<int>::iterator it2 = it->begin(); it2 != it->end(); ++it2) {
545                 double coef = sol[getVertexIndex(*it2, color, partition_size)];
546                 sum += sol[getVertexIndex(*it2, color, partition_size)];
547             }
548             int k = (it->size() - 1) / 2;
549             if (sum > k*sol[color-1]) {
550                 loadUnsatisfiedOddholeRestriction(env, lp, partition_size, *it, color
551                     );
552                 ++counter;
553             }
554         }
555     }
556
557     printf("%d unsatisfied oddhole restrictions found!\n", counter);
558
559     return counter;
560 }
561
562 int loadUnsatisfiedOddholeRestriction(CPXENVptr& env, CPXLPptr& lp, int
563 partition_size, const set<int>& path, int color) {
564
565     int ccnt = 0;
566     int rcnt = 1;
567     int nzcnt = path.size() + 1;
568
569     double rhs = 0;
570     char sense = 'L';
571
572     int matbeg = 0;
573     int* matind = new int[path.size() + 1];
574     double* matval = new double[path.size() + 1];
575     char **rowname = new char*[rcnt];
576     rowname[0] = new char[40];
577     sprintf(rowname[0], "unsatisfied_oddhole");

```

```

574
575     int k = (path.size() - 1) / 2;
576
577     matind[0] = color - 1;
578     matval[0] = -k;
579
580     int i = 1;
581     for (set<int>::iterator it = path.begin(); it != path.end(); ++it) {
582         matind[i] = getVertexIndex(*it, color, partition_size);
583         matval[i] = 1;
584         ++i;
585     }
586
587     // add restriction
588     int status = CPXaddrows(env, lp, cnt, rcnt, nzcnt, &rhs, &sense, &matbeg, matind
589         , matval, NULL, rowname);
589     checkStatus(env, status);
590
591     // free memory
592     delete[] matind;
593     delete[] matval;
594     delete rowname[0];
595     delete rowname;
596
597     return 0;
598 }
599
600 int maximalCliqueFamilyHeuristic(set<set<int>>& clique_familly, int vertex_size,
601     int edge_size, int partition_size, bool* adjacencyList) {
602
603     printf("Generating clique familly.\n");
604
605     for (int id = 1; id <= vertex_size; id++) {
606         set<int> clique;
607         clique.insert(id);
608         for (int id2 = id + 1; id2 <= vertex_size; ++id2) {
609             if (adjacentToAll(id2, edge_size, adjacencyList, clique)) {
610                 clique.insert(id2);
611             }
612         }
613         if (clique.size() > 2) {
614             if (cliqueNotContained(clique, clique_familly)) {
615                 clique_familly.insert(clique);
616             }
617         }
618     }
619
620     // print the familly
621     // for (set<set<int>>::iterator it = clique_familly.begin(); it !=
622     //     clique_familly.end(); ++it) {
623     //     cout << "Clique: ";
624     //     for (set<int>::iterator it2 = it->begin(); it2 != it->end(); ++it2) {
625     //         cout << *it2 << " ";
626     //     }
627     //     cout << endl;
628     // }
629
630     int familly_size = clique_familly.size() * partition_size;

```

```

630     printf("Family generated (size: %d)\n", family_size);
631
632     return family_size;
633 }
634
635 int findUnsatisfiedCliqueRestrictions(CPXENVptr& env, CPXLPptr& lp, set<set<int>>&
    clique_family, int vertex_size, int partition_size, int n, double* sol) {
636
637     int counter = 0;
638     for (set<set<int>>::iterator it = clique_family.begin(); it != clique_family.
        end(); ++it) {
639
640         for (int color = 1; color <= partition_size; ++color) {
641             double sum = 0;
642             for (set<int>::iterator it2 = it->begin(); it2 != it->end(); ++it2) {
643                 double coef = sol[getVertexIndex(*it2, color, partition_size)];
644                 sum += sol[getVertexIndex(*it2, color, partition_size)];
645             }
646             if (sum > sol[color - 1]) {
647                 loadUnsatisfiedCliqueRestriction(env, lp, partition_size, *it, color)
648                     ;
649                 ++counter;
650             }
651         }
652     }
653     printf("%d unsatisfied clique restrictions found!\n", counter);
654
655     return counter;
656 }
657
658 int loadUnsatisfiedCliqueRestriction(CPXENVptr& env, CPXLPptr& lp, int partition_size
    , const set<int>& clique, int color) {
659
660     int ccnt = 0;
661     int rcnt = 1;
662     int nzcnt = clique.size() + 1;
663
664     double rhs = 0;
665     char sense = 'L';
666
667     int matbeg = 0;
668     int* matind = new int[clique.size() + 1];
669     double* matval = new double[clique.size() + 1];
670     char **rowname = new char*[rcnt];
671     rowname[0] = new char[40];
672     sprintf(rowname[0], "unsatisfied-clique");
673
674     matind[0] = color - 1;
675     matval[0] = -1;
676
677     int i = 1;
678     for (set<int>::iterator it = clique.begin(); it != clique.end(); ++it) {
679         matind[i] = getVertexIndex(*it, color, partition_size);
680         matval[i] = 1;
681         ++i;
682     }
683
684     // add restriction

```

```

685     int status = CPXaddrows(env, lp, ccnt, rcnt, nzcnt, &rhs, &sense, &matbeg, matind
        , matval, NULL, rowname);
686     checkStatus(env, status);
687
688     // free memory
689     delete [] matind;
690     delete [] matval;
691     delete rowname[0];
692     delete rowname;
693
694     return 0;
695 }
696
697 int loadAdjacencyColorRestriction(CPXENVptr& env, CPXLPtr& lp, int vertex_size, int
partition_size) {
698
699     // load third restriction
700     int ccnt = 0; // new columns being added.
701     int rcnt = vertex_size * partition_size; // new rows being added.
702     int nzcnt = rcnt*2; // nonzero constraint coefficients being
        added.
703
704     double *rhs = new double[rcnt]; // independent term in restrictions.
705     char *sense = new char[rcnt]; // sense of restriction inequality.
706
707     int *matbeg = new int[rcnt]; // array position where each restriction
        starts in matind and matval.
708     int *matind = new int[rcnt*2]; // index of variables != 0 in
        restriction (each var has an index defined above)
709     double *matval = new double[rcnt*2]; // value corresponding to index in
        restriction.
710     char **rownames = new char*[rcnt]; // row labels.
711
712     int i = 0;
713     for (int v = 1; v <= vertex_size; ++v) {
714         for (int color = 1; color <= partition_size; ++color) {
715             matbeg[i] = i*2;
716
717             matind[i*2] = getVertexIndex(v, color, partition_size);
718             matind[i*2+1] = color-1;
719
720             matval[i*2] = 1;
721             matval[i*2+1] = -1;
722
723             rhs[i] = 0;
724             sense[i] = 'L';
725             rownames[i] = new char[40];
726             sprintf(rownames[i], "color_res");
727
728             ++i;
729         }
730     }
731
732     // add restriction
733     int status = CPXaddrows(env, lp, ccnt, rcnt, nzcnt, rhs, sense, matbeg, matind,
        matval, NULL, rownames);
734     checkStatus(env, status);
735
736     // free memory

```

```

737     for (int i = 0; i < rcnt; ++i) {
738         delete [] rownames[i];
739     }
740
741     delete [] rhs;
742     delete [] sense;
743     delete [] matbeg;
744     delete [] matind;
745     delete [] matval;
746     delete [] rownames;
747
748     return 0;
749 }
750
751 int solveLP(CPXENVptr& env, CPXLPptr& lp, int edge_size, int vertex_size, int
partition_size) {
752
753     printf("\nSolving MIP.\n");
754
755     int n = partition_size + (vertex_size*partition_size); // amount of total
variables
756
757     // calculate runtime
758     double inittime, endtime;
759     int status = CPXgettime(env, &inittime);
760     checkStatus(env, status);
761
762     // solve LP
763     status = CPXmipopt(env, lp);
764     checkStatus(env, status);
765
766     status = CPXgettime(env, &endtime);
767     checkStatus(env, status);
768
769     // check solution state
770     int solstat;
771     char statstring[510];
772     CPXCHARptr p;
773     solstat = CPXgetstat(env, lp);
774     p = CPXgetstatstring(env, solstat, statstring);
775     string statstr(statstring);
776     if (solstat != CPXMIP_OPTIMAL && solstat != CPXMIP_OPTIMALTOL &&
777         solstat != CPXMIP_NODELIM_FEAS && solstat != CPXMIP_TIME_LIM_FEAS) {
778         // printf("Optimization failed.\n");
779         cout << "Optimization failed: " << solstat << endl;
780         exit(1);
781     }
782
783     double objval;
784     status = CPXgetobjval(env, lp, &objval);
785     checkStatus(env, status);
786
787     // get values of all solutions
788     double *sol = new double[n];
789     status = CPXgetx(env, lp, sol, 0, n - 1);
790     checkStatus(env, status);
791
792     // write solutions to current window
793     cout << "Optimization result: " << statstring << endl;

```

```

794     cout << "Time taken to solve final LP: " << (endtime - inittime) << endl;
795     cout << "Colors used: " << objval << endl;
796     for (int color = 1; color <= partition_size; ++color) {
797         if (sol[color-1] == 1) {
798             cout << "w_" << color << " = " << sol[color-1] << " (" << colors[color-1]
              << ")" << endl;
799         }
800     }
801
802     for (int id = 1; id <= vertex_size; ++id) {
803         for (int color = 1; color <= partition_size; ++color) {
804             int index = getVertexIndex(id, color, partition_size);
805             if (sol[index] == 1) {
806                 cout << "x_" << id << " = " << colors[color-1] << endl;
807             }
808         }
809     }
810
811     delete [] sol;
812
813     return 0;
814 }
815
816 int convertVariableType(CPXENVptr& env, CPXLPptr& lp, int vertex_size, int
partition_size, char vtype) {
817
818     int n = partition_size + (vertex_size*partition_size);
819     int* indices = new int[n];
820     char* xtype = new char[n];
821
822     for (int i = 0; i < n; i++) {
823         indices[i] = i;
824         xtype[i] = vtype;
825     }
826     CPXchgctype(env, lp, n, indices, xtype);
827
828     delete [] indices;
829     delete [] xtype;
830
831     return 0;
832 }
833
834 int setBranchAndBoundConfig(CPXENVptr& env) {
835
836     // CPLEX config
837     // http://www-01.ibm.com/support/knowledgecenter/SSSA5P\_12.2.0/ilog.odms.cplex.
      help/Content/Optimization/Documentation/CPLEX/\_pubskel/CPLEX916.html
838
839     // deactivate pre-processing
840     CPXsetintparam(env, CPX_PARAMPRESLVND, -1);
841     CPXsetintparam(env, CPX_PARAMREPEATPRESOLVE, 0);
842     CPXsetintparam(env, CPX_PARAMRELAXPREIND, 0);
843     CPXsetintparam(env, CPX_PARAMREDUCE, 0);
844     CPXsetintparam(env, CPX_PARAMLLANDPCUTS, -1);
845
846     // maximize objective function
847     // CPXchgobjsen(env, lp, CPX_MAX);
848
849     // enable/disable screen output

```



```

850     CPXsetintparam(env, CPX_PARAM_SCRIND, CPX_OFF);
851
852     // set execution limit
853     CPXsetdblparam(env, CPX_PARAM_TILIM, 3600);
854
855     // disable presolve
856     // CPXsetintparam(env, CPX_PARAM_PREIND, CPX_OFF);
857
858     // enable traditional branch and bound
859     CPXsetintparam(env, CPX_PARAM_MIPSEARCH, CPX_MIPSEARCH_TRADITIONAL);
860
861     // use only one thread for experimentation
862     CPXsetintparam(env, CPX_PARAM_THREADS, 1);
863
864     // do not add cutting planes
865     CPXsetintparam(env, CPX_PARAM_EACHCUTLIM, CPX_OFF);
866
867     // disable gomory fractional cuts
868     CPXsetintparam(env, CPX_PARAM_FRACCUTS, -1);
869
870     // measure time in CPU time
871     // CPXsetintparam(env, CPX_PARAM_CLOCKTYPE, CPX_ON);
872
873     return 0;
874 }
875
876
877 int checkStatus(CPXENVptr& env, int status) {
878     if (status) {
879         char buffer[100];
880         CPXgeterrorstring(env, status, buffer);
881         printf("%s\n", buffer);
882         exit(1);
883     }
884     return 0;
885 }

```
