Investigación Operativa
Coloreo Particionado de Grafos

7 de diciembre de 2015

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

Resumen: ???
Keywords: ???

Índice

1. Modelo

Dado un grafo $G(V, E)$ con $n = |V|$ vértices y $m = |E|$ aristas, un coloreo de G se define como una asignación de un color o etiqueta a cada $v \in V$ de forma tal que para todo par de vértices adyacentes $(p, q) \in E$ poseen colores distintos. El clásico 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 vértices V que se encuentra particionado en $V_1, ..., V_k$, el problema consiste en asignar un color $c \in C$ a solo un vértice de cada partición de forma tal que dos vértices adyacentes no reciban el mismo color y minimizando la cantidad de colores utilizados.

Este problema se puede modelar con Programación 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. Función objetivo

De esta forma la función 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 vértices adyacentes no comparten color. Recordar que no necesariamente se le asigna un color a todo vértice.

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

Solo se le asigna un color a un único vértice de cada partición $p \in P$. Esto implica que cada vértice tiene a lo sumo solo un color.

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

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

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

Integralidad y positividad de las variables:

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

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

1.3. Eliminación de simetrías

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

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

2. Branch & Bound

La implementación 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 están definida por:

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

Demostración Para esta demostración utilizaremos las desigualdades Chvátal-Gomory sobre las restricciones del LP planteado en la sección 1.2 e inducción. A priori el teorema es bastante intuitivo. Si pinto algún vértice de una clique, no puedo pintar ninguno adyacente del mismo color sin importar la forma en la que particione los vértices del grafo. Sea n el tamaño de la clique maximal.

Casos Base

1. $n = 1$: Si en la clique maximal tengo solo un vértice, no existe arista que contenga este vértice, caso contrario la clique tendría dos elementos. Por lo tanto, este vértice puede estar pintado o no dentro de la partición. Es decir, se cumple la ecuación que queremos probar.
2. $n = 2$: Si la clique maximal tiene dos elementos, por definición son conexos. Por la restricción que indica que los vértices adyacentes no comparten color, aquí hay 2 opciones. La primera opción es que a ningún vértice se le asigna un color j_0. La otra opción 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 vértices, 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 definición 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 hipótesis inductiva, sabemos que:

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

Al agregar un vértice 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 vértice x_{nj_0} se le asigna un color, por las restricciones de las aristas que agregamos al resto de los vértices de la clique no se le puede asignar el color j_0.
- Si al vértice x_{nj_0} no se le asigna un color o se le asigna un color diferente a j_0, por hipótesis inductiva sabemos que lo que queremos probar vale. □

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:

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

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

- Si no se asigna el color j_0 a algun vértice 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 $(|C| - 1)/2$ vértices. 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 separación* buscan acotar el espacio de búsqueda para que se parezca mas a la cápsula convexa. Existen algoritmos de separación exactos y heurísticos. Los algoritmos heurísticos, luego de resolver la relajación del problema entero y encontrar una solución óptima 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 solución óptima de la relajación, se agrega esta nueva restricción y se vuelve a resolver el programa lineal. Este procedimiento se conoce como algoritmo de plano de corte. Si una solución óptima 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

En general, construir las familias de desigualdades enunciadas en las secciones anteriores de forma exhaustiva es un problema NP-Hard. Por esta razón los algoritmos heurísticos son sumamente útiles para buscar una aproximación polinomial al problema. Las heurísticas que enunciaremos a continuación utilizan algunas propiedades de la representación de nuestro grafo, ya sea para su construcción 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 simétrica 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 vértices 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 índice en la lista se puede ver claramente en el código. La idea es bastante simple y se basa principalmente en usar la expresión para la suma de enteros consecutivos.

En segundo lugar, numeramos todos los vértices con enteros comenzando con $id = 1$. Por construcción, luego nuestras heurísticas nos garantizaran que nuestro conjunto de índices que representa a un miembro de una familia esta ordenado. Esto es muy ventajoso en el sentido que podemos saber fácilmente si un nuevo potencial miembro de la familia esta contenido dentro de un miembro existente. Por otro lado, tiene una clara desventaja: la familia dependerá de como los vértices son numerados.

En un principio, la estrategia que seguimos fue generar las diferentes familias una vez, y luego verificar en cada iteración si la solución de la relajación violaba alguna desigualdad. Dado que esta estrategia en general no daba resultados muy satisfactorios, luego decidimos generar las familias en función del resultado de la relajación para cada iteración.

Por otro lado, muchas veces nuestra heurística generaba familias de desigualdades violadas muy grandes, y agregar todas terminaba siendo contraproducente. Por lo tanto, decidimos buscar algún criterio para poder determinar cuales son las mejores desigualdades a agregar y luego definir un *threshold* para decidir cuantas agregamos al LP. El criterio que utilizamos es el modulo de la diferencia entre los miembros de la desigualdad, aunque pueden existir otros en función también de la cantidad de variables en la desigualdad. Muchas veces las desigualdades mas violadas difieren solamente en pocas variables, por lo que esto también podría ser tenido en cuenta.

3.4.1. Heurística de Separación para Clique Maximal

Para esta heurística, lo que hacemos es recorrer los vértices que tienen una solución positiva en la relajación del LP en orden. En primer lugar, tomamos el primer vértice, y luego comenzamos a recorrer la lista hasta que encontramos un vértice 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 vértice, y así sucesivamente. Luego agregamos las mejores *threshold* desigualdades por score. Este procedimiento se puede ilustrar con el siguiente pseudocódigo:

Algorithm 1 Algoritmo para agregar cliques violadas

```
1: procedure generateCliqueFamilly(V, E, sol, threshold, lp)
2:     set < score, set < int >> clique_familly
3:     for id ← 1, |V| do
4:         if sol[id] > 0 + ϵ then
5:             continue
6:         end if
7:         set < int > clique
8:         clique.insert(id)
9:         for id2 ← id + 1, |V| do
10:            if sol[id2] > 0 + ϵ then
11:                continue
12:            end if
13:            if clique.adyacentToAll(id2) then
14:                clique.insert(id2)
15:            end if
16:        end for
17:        if ¬clique_familly.isContained(clique) then
18:            clique_familly.insert(< getScore(clique), clique >)
19:        end if
20:    end for
21:    sortByScore(clique_familly)
22:    addTopCliqueRestrictions(lp, clique_familly, threshold)
23: end procedure
```

Notar que en la practica solo consideramos cliques de tamaño mayor a 2, dado que si no se pisan con las restricciones de adyacencia del LP. A su vez, esta heurística debe ser generalizada para todos los colores, cosa que no mostramos dado que no aportaba nada al momento de mostrar la idea del algoritmo de forma clara.

3.4.2. Heurística de Separación para Aujero Impar

Para esta heurística, seguimos un procedimiento similar al anterior. Recorremos los vértices en orden, y los vamos agregando si son adyacentes. Al final, el conjunto de vértices 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 condición hasta que se satisfaga. Finalmente, agregamos las mejores *threshold* desigualdades por score. Este procedimiento se puede ilustrar con el siguiente pseudocódigo:

Algorithm 2 Algoritmo para agregar aujeros impares violados

```
1: procedure generateOddholeFamilly(V, E, sol, threshold, lp)
2:     set < score, set < int >> oddhole_familly
3:     for id ← 1, |V| do
4:         if sol[id] > 0 + ϵ then
5:             continue
6:         end if
7:         set < int > path
8:         path.insert(id)
9:         for id2 ← id + 1, |V| do
10:            if sol[id2] > 0 + ϵ then
11:                continue
12:            end if
13:            if isAdyacent(path.end, id2) then
14:                path.insert(id2)
15:            end if
16:        end for
17:        while path.size() ≥ 3 and (path.size() mod 2 == 0 or ¬isAdyacent(path.start, path.end)) do
18:            path.erase(path.end)
19:        end while
20:        if path.size() ≥ 3 and isAdyacent(path.start, path.end) then
21:            oddhole_familly.insert(< getScore(path), path >)
22:        end if
23:    end for
24:    sortByScore(oddhole_familly)
25:    addTopPathRestrictions(lp, oddhole_familly, threshold)
26: end procedure
```

Notar que en ambas heurísticas utilizamos la tolerancia ϵ para evitar problemas numéricos.

3.5. Cut & Branch

Dado que las familias de desigualdades anteriormente expuestas no describen de forma exhaustiva la cápsula convexa del problema, los algoritmos de planos de corte no necesariamente convergen. Por esta razón decidimos implementar un algoritmo de Cut & Branch. Los algoritmos Cut & Branch buscan aplicar planos de corte a la raíz del árbol de enumeración de Branch & Bound, lo que *potencialmente* puede mejorar el tiempo de ejecución de los problemas al reducir el espacio de búsqueda y permitiendo mejores podas. Una vez aplicados los cortes, se resuelve el problema resultante mediante Branch & Cut.

En nuestra implementación, los parámetros que deben ser calibrados para este algoritmo son la cantidad de iteraciones y el threshold. Por cada iteración, el algoritmo resuelve la relajación del problema y agrega a lo sumo *threshold* restricciones de cada tipo.

4. Experimentacion

5. Conclusión

6. Apéndice A: Código

6.1. coloring.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

ILOSTLBEGIN // macro to define namespace

// helper functions
int getVertexIndex(int id, int color, int partition_size);
inline int fromMatrixToVector(int from, int to, int vertex_size);
inline bool isAdyacent(int from, int to, int vertex_size, bool* adjacencyList);
bool adyacentToAll(int id, int vertex_size, bool* adjacencyList, const set<int>& clique);
bool cliqueNotContained(const set<int>& clique, int color, const vector<tuple<double, int, set<int> > >& clique_familly);

// 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* adjacencyList);
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* adjacencyList, int iterations, int load_limit, int select_cuts);
int maximalCliqueFamillyHeuristic(CPXENVptr& env, CPXLPptr& lp, int vertex_size, int edge_size, int partition_size, bool* adjacencyList, double* sol, int load_limit);
int loadUnsatisfiedCliqueRestriction(CPXENVptr& env, CPXLPptr& lp, int partition_size, const set<int>& clique, int color);

int oddholeFamillyHeuristic(CPXENVptr& env, CPXLPptr& lp, int vertex_size, int edge_size, int partition_size, bool* adjacencyList, double* sol, int load_limit);
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);
```

```
40 int convertVariableType(CPXENVptr& env, CPXLPptr& lp, int vertex_size, int
       partition_size, char vtype);
41 int setTraversalStrategy(CPXENVptr& env, int strategy);
42 int setBranchingVariableStrategy(CPXENVptr& env, int strategy);
43 int setBranchAndBoundConfig(CPXENVptr& env);
44
45 int checkStatus(CPXENVptr& env, int status);
46
47 // colors array!
48 const char* colors[] = {"Blue", "Red", "Green", "Yellow", "Grey", "Green", "Pink", "
       AliceBlue","AntiqueWhite","Aqua","Aquamarine","Azure","Beige",
49 "Bisque","Black","BlanchedAlmond","BlueViolet","Brown","BurlyWood","CadetBlue","
       Chartreuse","Chocolate","Coral","CornflowerBlue",
50 "Cornsilk","Crimson","Cyan","DarkBlue","DarkCyan","DarkGoldenRod","DarkGray","
       DarkGrey","DarkGreen","DarkKhaki","DarkMagenta","DarkOliveGreen",
51 "Darkorange","DarkOrchid","DarkRed","DarkSalmon","DarkSeaGreen","DarkSlateBlue","
       DarkSlateGray","DarkSlateGrey","DarkTurquoise",
52 "DarkViolet","DeepPink","DeepSkyBlue","DimGray","DimGrey","DodgerBlue","FireBrick","
       FloralWhite","ForestGreen","Fuchsia",
53 "Gainsboro","GhostWhite","Gold","GoldenRod","Gray","GreenYellow","HoneyDew","HotPink"
       ,"IndianRed","Indigo",
54 "Ivory","Khaki","Lavender","LavenderBlush","LawnGreen","LemonChiffon","LightBlue","
       LightCoral","LightCyan","LightGoldenRodYellow",
55 "LightGray","LightGrey","LightGreen","LightPink","LightSalmon","LightSeaGreen","
       LightSkyBlue","LightSlateGray","LightSlateGrey",
56 "LightSteelBlue","LightYellow","Lime","LimeGreen","Linen","Magenta","Maroon","
       MediumAquaMarine","MediumBlue","MediumOrchid",
57 "MediumPurple","MediumSeaGreen","MediumSlateBlue","MediumSpringGreen","
       MediumTurquoise","MediumVioletRed","MidnightBlue",
58 "MintCream","MistyRose","Moccasin","NavajoWhite","Navy","OldLace","Olive","OliveDrab"
       ,"Orange","OrangeRed","Orchid",
59 "PaleGoldenRod","PaleGreen","PaleTurquoise","PaleVioletRed","PapayaWhip","PeachPuff",
       "Peru","Plum","PowderBlue",
60 "Purple","RosyBrown","RoyalBlue","SaddleBrown","Salmon","SandyBrown","SeaGreen","
       SeaShell","Sienna","Silver","SkyBlue",
61 "SlateBlue","SlateGray","SlateGrey","Snow","SpringGreen","SteelBlue","Tan","Teal","
       Thistle","Tomato","Turquoise","Violet",
62 "Wheat","White","WhiteSmoke","YellowGreen"};
63
64 int main(int argc, char **argv) {
65
66     if (argc != 11) {
67         printf("Usage: %s inputFile solver partitions symmetry_breaker iterations
               select_cuts load_limit custom_config traversal_strategy branching_strategy
               \n", argv[0]);
68         exit(1);
69     }
70
71     int solver = atoi(argv[2]);
72     int partition_size = atoi(argv[3]);
73     bool symmetry_breaker = (atoi(argv[4]) == 1);
74     int iterations = atoi(argv[5]);
75     int select_cuts = atoi(argv[6]);              // 0: clique only, 1: oddhole only,
           2: both
76     int load_limit = atoi(argv[7]);
77     int custom_config = atoi(argv[8]);           // 0: default, 1: custom
78     int traversal_strategy = atoi(argv[9]);
79     int branching_strategy = atoi(argv[10]);
80
```

```
81      if (solver == 1) {
82          printf("Solver: Branch & Bound\n");
83      } else {
84          printf("Solver: Cut & Branch\n");
85      }
86
87      /* read graph input file
88       * format: http://mat.gsia.cmu.edu/COLOR/instances.html
89       * graph representation chosen in order to load the LP easily.
90       * - vector of edges
91       * - vector of partitions
92       */
93      FILE* fp = fopen(argv[1], "r");
94
95      if (fp == NULL) {
96          printf("Invalid input file.\n");
97          exit(1);
98      }
99
100     char buf[100];
101     int vertex_size, edge_size;
102
103     set<pair<double,int> > edges; // sometimes we have to filter directed graphs
104
105     while (fgets(buf, sizeof(buf), fp) != NULL) {
106         if (buf[0] == 'c') continue;
107         else if (buf[0] == 'p') {
108             sscanf(&buf[7], "%d %d", &vertex_size, &edge_size);
109         }
110         else if (buf[0] == 'e') {
111             int from, to;
112             sscanf(&buf[2], "%d %d", &from, &to);
113             if (from < to) {
114                 edges.insert(pair<double,int>(from, to));
115             } else {
116                 edges.insert(pair<double,int>(to, from));
117             }
118         }
119     }
120
121     // build adyacency list
122     edge_size = edges.size();
123     int adyacency_size = vertex_size*vertex_size - ((vertex_size+1)*vertex_size/2);
124     bool* adjacencyList = new bool[adyacency_size]; // can be optimized even more
            with a bitfield.
125     fill_n(adjacencyList, adyacency_size, false);
126     for (set<pair<double,int> >::iterator it = edges.begin(); it != edges.end(); ++it
        ) {
127         adjacencyList[fromMatrixToVector(it->first, it->second, vertex_size)] = true;
128     }
129
130     // set random seed
131     // srand(time(NULL));
132
133     // asign every vertex to a partition
134     // int partition_size = rand() % vertex_size + 1;
135     vector<vector<int> > partitions(partition_size, vector<int>());
136
137     for (int i = 0; i < vertex_size; ++i) {
```

```
138             partitions[i % partition_size].push_back(i+1);
139         }
140 
141         // warning: this procedure doesn't guarantee every partition will have an element
                .
142         // for (int i = 1; i <= vertex_size; ++i) {
143         //  int assign_partition = rand() % partition_size;
144         //  partitions[assign_partition].push_back(i);
145         // }
146 
147         // // update partition_size
148         // for (std::vector<vector<int> >::iterator it = partitions.begin(); it !=
                partitions.end(); ++it) {
149         //  if (it->size() == 0) --partition_size;
150         // }
151 
152         printf("Graph: vertex_size: %d, edge_size: %d, partition_size: %d\n", vertex_size
                , edge_size, partition_size);
153 
154         // start loading LP using CPLEX
155         int status;
156         CPXENVptr env; // pointer to enviroment
157         CPXLPptr lp;   // pointer to the lp.
158 
159         env = CPXopenCPLEX(&status); // create enviroment
160         checkStatus(env, status);
161 
162         // create LP
163         lp = CPXcreateprob(env, &status, "Instance of partitioned graph coloring.");
164         checkStatus(env, status);
165 
166         if (custom_config == 1) setBranchAndBoundConfig(env);
167         setTraversalStrategy(env, traversal_strategy);
168         setBranchingVariableStrategy(env, branching_strategy);
169 
170         if (solver == 1) { // pure branch & bound
171             loadObjectiveFunction(env, lp, vertex_size, partition_size, CPX_BINARY);
172         } else {
173             loadObjectiveFunction(env, lp, vertex_size, partition_size, CPX_CONTINUOUS);
174         }
175 
176         loadAdyacencyColorRestriction(env, lp, vertex_size, edge_size, partition_size,
                adjacencyList);
177         loadSingleColorInPartitionRestriction(env, lp, partitions, partition_size);
178         loadAdyacencyColorRestriction(env, lp, vertex_size, partition_size);
179 
180         if (symmetry_breaker) loadSymmetryBreaker(env, lp, partition_size);
181 
182         if (solver != 1) loadCuttingPlanes(env, lp, vertex_size, edge_size,
                partition_size, adjacencyList, iterations, load_limit, select_cuts);
183 
184         // write LP formulation to file, great to debug.
185         status = CPXwriteprob(env, lp, "graph.lp", NULL);
186         checkStatus(env, status);
187 
188         convertVariableType(env, lp, vertex_size, partition_size, CPX_BINARY);
189 
190         solveLP(env, lp, edge_size, vertex_size, partition_size);
191 
```

```
192     delete[] adjacencyList;
193
194     return 0;
195 }
196
197 int getVertexIndex(int id, int color, int partition_size) {
198     return partition_size + ((id-1)*partition_size) + (color-1);
199 }
200
201 /* since the adyacency matrix is symmetric and the diagonal is not needed, we can
    simply
202  * store the upper diagonal and get adyacency from a list. the math is quite simple,
     it
203  * just uses the formula for the sum of integers. ids are numbered starting from 1.
204  */
205 inline int fromMatrixToVector(int from, int to, int vertex_size) {
206
207     // for speed, many parts of this code are commented, since by our usage we always
208     // know from < to and are in range.
209
210     // assert(from != to && from <= vertex_size && to <= vertex_size);
211
212     // if (from < to)
213         return from*vertex_size - (from+1)*from/2 - (vertex_size - to) - 1;
214     // else
215     //  return to*vertex_size - (to+1)*to/2 - (vertex_size - from) - 1;
216 }
217
218 inline bool isAdyacent(int from, int to, int vertex_size, bool* adjacencyList) {
219     return adjacencyList[fromMatrixToVector(from, to, vertex_size)];
220 }
221
222 bool adyacentToAll(int id, int vertex_size, bool* adjacencyList, const set<int>&
    clique) {
223     for (set<int>::iterator it = clique.begin(); it != clique.end(); ++it) {
224         if (!isAdyacent(*it, id, vertex_size, adjacencyList)) return false;
225     }
226     return true;
227 }
228
229 bool cliqueNotContained(const set<int>& clique, int color, const vector<tuple<double,
    int, set<int> > >& clique_familly) {
230     for (vector<tuple<double, int, set<int> > >::const_iterator it = clique_familly.
        begin(); it != clique_familly.end(); ++it) {
231         // by construction, sets are already ordered.
232         if (get<1>(*it) == color && includes(get<2>(*it).begin(), get<2>(*it).end(),
            clique.begin(), clique.end())) return false;
233     }
234     return true;
235 }
236
237 int loadObjectiveFunction(CPXENVptr& env, CPXLPptr& lp, int vertex_size, int
    partition_size, char vtype) {
238
239     // load objective function
240     int n = partition_size + (vertex_size*partition_size);
241     double *objfun  = new double[n];
242     double *ub      = new double[n];
243     char     *ctype = new char[n];
```

```
244     char **colnames = new char*[n];
245
246     for (int i = 0; i < partition_size; ++i) {
247         objfun[i] = 1;
248         ub[i] = 1;
249         ctype[i]    = vtype;
250         colnames[i] = new char[10];
251         sprintf(colnames[i], "w_%d", (i+1));
252     }
253
254     for (int id = 1; id <= vertex_size; ++id) {
255         for (int color = 1; color <= partition_size; ++color) {
256             int index = getVertexIndex(id, color, partition_size);
257             objfun[index]   = 0;
258             ub[index] = 1;
259             ctype[index]    = vtype;
260             colnames[index] = new char[10];
261             sprintf(colnames[index], "x%d_%d", id, color);
262         }
263     }
264
265     // CPLEX bug? If you set ctype, it doesn't identify the problem as continous.
266     int status = CPXnewcols(env, lp, n, objfun, NULL, ub, NULL, colnames);
267     checkStatus(env, status);
268
269     // free memory
270     for (int i = 0; i < n; ++i) {
271         delete[] colnames[i];
272     }
273
274     delete[] objfun;
275     delete[] ub;
276     delete[] ctype;
277     delete[] colnames;
278
279     return 0;
280 }
281
282 int loadAdyacencyColorRestriction(CPXENVptr& env, CPXLPptr& lp, int vertex_size, int
        edge_size, int partition_size, bool* adjacencyList) {
283
284     // load first restriction
285     int ccnt = 0;                           // new columns being added.
286     int rcnt = edge_size * partition_size; // new rows being added.
287     int nzcnt = rcnt*2;                     // nonzero constraint coefficients being
            added.
288
289     double *rhs = new double[rcnt];         // independent term in restrictions.
290     char *sense = new char[rcnt];           // sense of restriction inequality.
291
292     int *matbeg = new int[rcnt];            // array position where each restriction
            starts in matind and matval.
293     int *matind = new int[rcnt*2];          // index of variables != 0 in restriction
            (each var has an index defined above)
294     double *matval  = new double[rcnt*2];  // value corresponding to index in
            restriction.
295     char **rownames = new char*[rcnt];      // row labels.
296
297     int i = 0;
```

```
298     for (int from = 1; from <= vertex_size; ++from) {
299         for (int to = from + 1; to <= vertex_size; ++to) {
300 
301             if (!isAdyacent(from, to, vertex_size, adjacencyList)) continue;
302 
303             for (int color = 1; color <= partition_size; ++color) {
304                 matbeg[i] = i*2;
305 
306                 matind[i*2]   = getVertexIndex(from, color, partition_size);
307                 matind[i*2+1] = getVertexIndex(to  , color, partition_size);
308 
309                 matval[i*2]   = 1;
310                 matval[i*2+1] = 1;
311 
312                 rhs[i] = 1;
313                 sense[i] = 'L';
314                 rownames[i] = new char[40];
315                 sprintf(rownames[i], "%s", colors[color-1]);
316 
317                 ++i;
318             }
319         }
320     }
321 
322     // debug flag
323     // status = CPXsetintparam(env, CPX_PARAM_DATACHECK, CPX_ON);
324 
325     // add restriction
326     int status = CPXaddrows(env, lp, ccnt, rcnt, nzcnt, rhs, sense, matbeg, matind,
        matval, NULL, rownames);
327     checkStatus(env, status);
328 
329     // free memory
330     for (int i = 0; i < rcnt; ++i) {
331         delete[] rownames[i];
332     }
333 
334     delete[] rhs;
335     delete[] sense;
336     delete[] matbeg;
337     delete[] matind;
338     delete[] matval;
339     delete[] rownames;
340 
341     return 0;
342 }
343 
344 
345 int loadSingleColorInPartitionRestriction(CPXENVptr& env, CPXLPptr& lp, vector<vector
    <int>>& partitions, int partition_size) {
346 
347     // load second restriction
348     int p = 1;
349     for (std::vector<vector<int> >::iterator it = partitions.begin(); it !=
        partitions.end(); ++it) {
350 
351         int size = it->size();                  // current partition size.
352         if (size == 0) continue;                // skip empty partitions.
353 
```

```
354         int ccnt = 0;                           // new columns being added.
355         int rcnt = 1;                           // new rows being added.
356         int nzcnt = size*partition_size;        // nonzero constraint coefficients
                being added.
357
358         double *rhs = new double[rcnt];         // independent term in restrictions.
359         char *sense = new char[rcnt];           // sense of restriction inequality.
360
361         int *matbeg = new int[rcnt];            // array position where each
                restriction starts in matind and matval.
362         int *matind = new int[nzcnt];           // index of variables != 0 in
                restriction (each var has an index defined above)
363         double *matval  = new double[nzcnt];    // value corresponding to index in
                restriction.
364         char **rownames = new char*[rcnt];      // row labels.
365
366         matbeg[0] = 0;
367         sense[0]  = 'E';
368         rhs[0]    = 1;
369         rownames[0] = new char[40];
370         sprintf(rownames[0], "partition_%d", p);
371
372         int i = 0;
373         for (std::vector<int>::iterator it2 = it->begin(); it2 != it->end(); ++it2) {
374             for (int color = 1; color <= partition_size; ++color) {
375                 matind[i] = getVertexIndex(*it2, color, partition_size);
376                 matval[i] = 1;
377                 ++i;
378             }
379         }
380
381         // add restriction
382         int status = CPXaddrows(env, lp, ccnt, rcnt, nzcnt, rhs, sense, matbeg,
                matind, matval, NULL, rownames);
383         checkStatus(env, status);
384
385         // free memory
386         delete[] rownames[0];
387         delete[] rhs;
388         delete[] sense;
389         delete[] matbeg;
390         delete[] matind;
391         delete[] matval;
392         delete[] rownames;
393
394         ++p;
395     }
396
397     return 0;
398 }
399
400 int loadSymmetryBreaker(CPXENVptr& env, CPXLPptr& lp, int partition_size) {
401
402     int ccnt = 0;                           // new columns being added.
403     int rcnt = partition_size - 1;          // new rows being added.
404     int nzcnt = 2*rcnt;                     // nonzero constraint coefficients being
            added.
405
406     double* rhs = new double[rcnt];         // independent term in restrictions.
```

```
407     char *sense = new char[rcnt];           // sense of restriction inequality.
408
409     int *matbeg = new int[rcnt];            // array position where each restriction
            starts in matind and matval.
410     int *matind = new int[rcnt*2];          // index of variables != 0 in restriction
            (each var has an index defined above)
411     double *matval  = new double[rcnt*2];   // value corresponding to index in
            restriction.
412     char **rownames = new char*[rcnt];      // row labels.
413
414     int i = 0;
415     for (int color = 0; color < partition_size - 1; ++color) {
416         matbeg[i] = i*2;
417         matind[i*2]   = color;
418         matind[i*2+1] = color + 1;
419         matval[i*2]   = -1;
420         matval[i*2+1] = 1;
421
422         rhs[i] = 0;
423         sense[i] = 'L';
424         rownames[i] = new char[40];
425         sprintf(rownames[i], "%s", "symmetry_breaker");
426
427         ++i;
428     }
429
430
431     // add restriction
432     int status = CPXaddrows(env, lp, ccnt, rcnt, nzcnt, rhs, sense, matbeg, matind,
            matval, NULL, rownames);
433     checkStatus(env, status);
434
435     // free memory
436     for (int i = 0; i < rcnt; ++i) {
437         delete[] rownames[i];
438     }
439
440     delete[] rhs;
441     delete[] sense;
442     delete[] matbeg;
443     delete[] matind;
444     delete[] matval;
445     delete[] rownames;
446
447     return 0;
448 }
449
450 int loadCuttingPlanes(CPXENVptr& env, CPXLPptr& lp, int vertex_size, int edge_size,
        int partition_size, bool* adjacencyList, int iterations, int load_limit, int
        select_cuts) {
451
452     printf("Finding Cutting Planes.\n");
453
454     // calculate runtime
455     double inittime, endtime;
456     int status = CPXgettime(env, &inittime);
457
458     int n = partition_size + (vertex_size*partition_size);
459
```

```
460     double *sol = new double[n];
461     int i = 1;
462     int unsatisfied_restrictions = 0;
463     while (i <= iterations) {
464 
465         printf("Iteration %d\n", i);
466 
467         // solve LP
468         status = CPXlpopt(env, lp);
469         checkStatus(env, status);
470 
471         status = CPXgetx(env, lp, sol, 0, n - 1);
472         checkStatus(env, status);
473 
474         // print relaxation result
475         // for (int id = 1; id <= vertex_size; ++id) {
476         //  for (int color = 1; color <= partition_size; ++color) {
477         //      int index = getVertexIndex(id, color, partition_size);
478         //      if (sol[index] == 0) continue;
479         //      cout << "x" << id << "_" << color << " = " << sol[index] << endl;
480         //  }
481         // }
482 
483         if (select_cuts == 0 || select_cuts == 2) unsatisfied_restrictions +=
                maximalCliqueFamillyHeuristic(env, lp, vertex_size, edge_size,
                partition_size, adjacencyList, sol, load_limit);
484         if (select_cuts == 1 || select_cuts == 2) unsatisfied_restrictions +=
                oddholeFamillyHeuristic(env, lp, vertex_size, edge_size, partition_size,
                adjacencyList, sol, load_limit);
485 
486         if (unsatisfied_restrictions == 0) break;
487 
488         unsatisfied_restrictions = 0;
489         ++i;
490     }
491 
492     status = CPXgettime(env, &endtime);
493     double elapsed_time = endtime-inittime;
494     cout << "Time taken to add cutting planes: " << elapsed_time << endl;
495 
496     return 0;
497 }
498 
499 int maximalCliqueFamillyHeuristic(CPXENVptr& env, CPXLPptr& lp, int vertex_size, int
        edge_size, int partition_size, bool* adjacencyList, double* sol, int load_limit) {
500 
501     printf("Generating clique familly.\n");
502 
503     int loaded = 0;
504 
505     vector<tuple<double, int, set<int> > > clique_familly;
506 
507     for (int color = 1; color <= partition_size; ++color) {
508 
509         for (int id = 1; id <= vertex_size; id++) {
510 
511             if (sol[getVertexIndex(id, color, partition_size)] == 0) continue;
512 
513             double sum = sol[getVertexIndex(id, color, partition_size)];
```

```
514             set<int> clique;
515             clique.insert(id);
516             for (int id2 = id + 1; id2 <= vertex_size; ++id2) {
517                 if (sol[getVertexIndex(id2, color, partition_size)] == 0) continue;
518
519                 if (adyacentToAll(id2, vertex_size, adjacencyList, clique)) {
520                     clique.insert(id2);
521                     sum += sol[getVertexIndex(id2, color, partition_size)];
522                 }
523             }
524             if (clique.size() > 2 && sum > sol[color-1] + TOL) {
525                 if (cliqueNotContained(clique, color, clique_familly)) {
526                     double score = sum - sol[color-1];
527                     clique_familly.push_back(tuple<double, int, set<int> >(score,
                        color,clique));
528                 }
529             }
530         }
531     }
532
533     sort(clique_familly.begin(), clique_familly.end(), greater<tuple<double, int, set
        <int> > >());
534
535     //print the familly
536     for (vector<tuple<double, int, set<int> > >::const_iterator it = clique_familly.
        begin();
537          it != clique_familly.end() && loaded < load_limit; ++loaded, ++it) {
538
539          loadUnsatisfiedCliqueRestriction(env, lp, partition_size, get<2>(*it), get
             <1>(*it));
540          cout << "Score: " << get<0>(*it) << " - ";
541          for (set<int>::iterator it2 = get<2>(*it).begin(); it2 != get<2>(*it).end();
             ++it2) {
542              cout << *it2 << " ";
543          }
544          cout << endl;
545     }
546
547     printf("Loaded %d/%d unsatisfied clique restrictions! (all colors)\n", loaded, (
        int) clique_familly.size());
548
549     return loaded;
550 }
551
552 int loadUnsatisfiedCliqueRestriction(CPXENVptr& env, CPXLPptr& lp, int partition_size
    , const set<int>& clique, int color) {
553
554     int ccnt = 0;
555     int rcnt = 1;
556     int nzcnt = clique.size() + 1;
557
558     double rhs = 0;
559     char sense = 'L';
560
561     int matbeg = 0;
562     int* matind    = new int[clique.size() + 1];
563     double* matval = new double[clique.size() +1];
564     char **rowname = new char*[rcnt];
565     rowname[0] = new char[40];
```

```
566     sprintf(rowname[0], "unsatisfied_clique");
567
568     matind[0] = color - 1;
569     matval[0] = -1;
570
571     int i = 1;
572     for (set<int>::iterator it = clique.begin(); it != clique.end(); ++it) {
573         matind[i] = getVertexIndex(*it, color, partition_size);
574         matval[i] = 1;
575         ++i;
576     }
577
578     // add restriction
579     int status = CPXaddrows(env, lp, ccnt, rcnt, nzcnt, &rhs, &sense, &matbeg, matind
        , matval, NULL, rowname);
580     checkStatus(env, status);
581
582     // free memory
583     delete[] matind;
584     delete[] matval;
585     delete rowname[0];
586     delete rowname;
587
588     return 0;
589 }
590
591 int oddholeFamillyHeuristic(CPXENVptr& env, CPXLPptr& lp, int vertex_size, int
    edge_size, int partition_size, bool* adjacencyList, double* sol, int load_limit) {
592
593     printf("Generating oddhole familly.\n");
594
595     int loaded = 0;
596
597     vector<tuple<double, int, set<int>>> path_familly; // dif, color, path
598
599     for (int color = 1; color <= partition_size; ++color) {
600
601         for (int id = 1; id <= vertex_size; id++) {
602
603             if (sol[getVertexIndex(id, color, partition_size)] == 0) continue;
604
605             double sum = 0;
606             set<int> path;
607             path.insert(id);
608             for (int id2 = id + 1; id2 <= vertex_size; ++id2) {
609                 if (sol[getVertexIndex(id2, color, partition_size)] == 0) continue;
610
611                 if (isAdyacent(*(--path.end()), id2, vertex_size, adjacencyList)) {
612                     path.insert(id2);
613                 }
614
615             }
616
617             while (path.size() >= 3 && (path.size() %2 == 0 ||
618                 !isAdyacent(*path.begin(), *(--path.end()), vertex_size,
                        adjacencyList))) {
619                 path.erase(--path.end());
620             }
621
```

```
622             for (set<int>::iterator it = path.begin(); it != path.end(); ++it) {
623                 sum += sol[getVertexIndex(*it, color, partition_size)];
624             }
625
626             int k = (path.size() - 1) / 2;
627             if (path.size() > 2 && sum > k*sol[color-1] + TOL) {
628                 double score = sum - k*sol[color-1];
629                 path_familly.push_back(tuple<double, int, set<int> >(score,color,path
                    ));
630             }
631         }
632     }
633
634     sort(path_familly.begin(), path_familly.end(), greater<tuple<double, int, set<int
        > > >());
635
636     //print the familly
637     for (vector<tuple<double, int, set<int> > >::const_iterator it = path_familly.
        begin();
638          it != path_familly.end() && loaded < load_limit; ++loaded, ++it) {
639          loadUnsatisfiedOddholeRestriction(env, lp, partition_size, get<2>(*it), get
             <1>(*it));
640          cout << "Score: " << get<0>(*it) << " - ";
641          for (set<int>::iterator it2 = get<2>(*it).begin(); it2 != get<2>(*it).end();
             ++it2) {
642              cout << *it2 << " ";
643          }
644          cout << endl;
645     }
646
647     printf("Loaded %d/%d unsatisfied oddhole restrictions! (all colors)\n", loaded, (
        int) path_familly.size());
648
649     return loaded;
650 }
651
652 int loadUnsatisfiedOddholeRestriction(CPXENVptr& env, CPXLPptr& lp, int
    partition_size, const set<int>& path, int color) {
653
654     int ccnt = 0;
655     int rcnt = 1;
656     int nzcnt = path.size() + 1;
657
658     double rhs = 0;
659     char sense = 'L';
660
661     int matbeg = 0;
662     int* matind    = new int[path.size() + 1];
663     double* matval = new double[path.size() +1];
664     char **rowname = new char*[rcnt];
665     rowname[0] = new char[40];
666     sprintf(rowname[0], "unsatisfied_oddhole");
667
668     int k = (path.size() - 1) / 2;
669
670     matind[0] = color - 1;
671     matval[0] = -k;
672
673     int i = 1;
```

```
674     for (set<int>::iterator it = path.begin(); it != path.end(); ++it) {
675         matind[i] = getVertexIndex(*it, color, partition_size);
676         matval[i] = 1;
677         ++i;
678     }
679 
680     // add restriction
681     int status = CPXaddrows(env, lp, ccnt, rcnt, nzcnt, &rhs, &sense, &matbeg, matind
        , matval, NULL, rowname);
682     checkStatus(env, status);
683 
684     // free memory
685     delete[] matind;
686     delete[] matval;
687     delete rowname[0];
688     delete rowname;
689 
690     return 0;
691 }
692 
693 int loadAdyacencyColorRestriction(CPXENVptr& env, CPXLPptr& lp, int vertex_size, int
    partition_size) {
694 
695     // load third restriction
696     int ccnt = 0;                               // new columns being added.
697     int rcnt = vertex_size * partition_size; // new rows being added.
698     int nzcnt = rcnt*2;                         // nonzero constraint coefficients being
         added.
699 
700     double *rhs = new double[rcnt];             // independent term in restrictions.
701     char *sense = new char[rcnt];               // sense of restriction inequality.
702 
703     int *matbeg = new int[rcnt];                // array position where each restriction
         starts in matind and matval.
704     int *matind = new int[rcnt*2];              // index of variables != 0 in
        restriction (each var has an index defined above)
705     double *matval  = new double[rcnt*2];    // value corresponding to index in
        restriction.
706     char **rownames = new char*[rcnt];          // row labels.
707 
708     int i = 0;
709     for (int v = 1; v <= vertex_size; ++v) {
710         for (int color = 1; color <= partition_size; ++color) {
711             matbeg[i] = i*2;
712 
713             matind[i*2]   = getVertexIndex(v, color, partition_size);
714             matind[i*2+1] = color-1;
715 
716             matval[i*2]   = 1;
717             matval[i*2+1] = -1;
718 
719             rhs[i] = 0;
720             sense[i] = 'L';
721             rownames[i] = new char[40];
722             sprintf(rownames[i], "color_res");
723 
724             ++i;
725         }
726     }
```

```
727
728     // add restriction
729     int status = CPXaddrows(env, lp, ccnt, rcnt, nzcnt, rhs, sense, matbeg, matind,
            matval, NULL, rownames);
730     checkStatus(env, status);
731
732     // free memory
733     for (int i = 0; i < rcnt; ++i) {
734         delete[] rownames[i];
735     }
736
737     delete[] rhs;
738     delete[] sense;
739     delete[] matbeg;
740     delete[] matind;
741     delete[] matval;
742     delete[] rownames;
743
744     return 0;
745 }
746
747 int solveLP(CPXENVptr& env, CPXLPptr& lp, int edge_size, int vertex_size, int
        partition_size) {
748
749     printf("\nSolving MIP.\n");
750
751     int n = partition_size + (vertex_size*partition_size); // amount of total
            variables
752
753     // calculate runtime
754     double inittime, endtime;
755     int status = CPXgettime(env, &inittime);
756     checkStatus(env, status);
757
758     // solve LP
759     status = CPXmipopt(env, lp);
760     checkStatus(env, status);
761
762     status = CPXgettime(env, &endtime);
763     checkStatus(env, status);
764
765     // check solution state
766     int solstat;
767     char statstring[510];
768     CPXCHARptr p;
769     solstat = CPXgetstat(env, lp);
770     p = CPXgetstatstring(env, solstat, statstring);
771     string statstr(statstring);
772     if (solstat != CPXMIP_OPTIMAL && solstat != CPXMIP_OPTIMAL_TOL &&
773         solstat != CPXMIP_NODE_LIM_FEAS && solstat != CPXMIP_TIME_LIM_FEAS) {
774         // printf("Optimization failed.\n");
775         cout << "Optimization failed: " << solstat << endl;
776         exit(1);
777     }
778
779     double objval;
780     status = CPXgetobjval(env, lp, &objval);
781     checkStatus(env, status);
782
```

```
783     // get values of all solutions
784     double *sol = new double[n];
785     status = CPXgetx(env, lp, sol, 0, n - 1);
786     checkStatus(env, status);
787 
788     int nodes_traversed = CPXgetnodecnt(env, lp);
789 
790     // write solutions to current window
791     cout << "Optimization result: " << statstring << endl;
792     cout << "Time taken to solve final LP: " << (endtime - inittime) << endl;
793     cout << "Colors used: " << objval << endl;
794     cout << "Nodes traversed: " << nodes_traversed << endl;
795     for (int color = 1; color <= partition_size; ++color) {
796         if (sol[color-1] == 1) {
797             cout << "w_" << color << " = " << sol[color-1] << " (" << colors[color-1]
                    << ")" << endl;
798         }
799     }
800 
801     for (int id = 1; id <= vertex_size; ++id) {
802         for (int color = 1; color <= partition_size; ++color) {
803             int index = getVertexIndex(id, color, partition_size);
804             if (sol[index] == 1) {
805                 cout << "x_" << id << " = " << colors[color-1] << endl;
806             }
807         }
808     }
809 
810     delete[] sol;
811 
812     return 0;
813 }
814 
815 int convertVariableType(CPXENVptr& env, CPXLPptr& lp, int vertex_size, int
        partition_size, char vtype) {
816 
817     int n = partition_size + (vertex_size*partition_size);
818     int* indices = new int[n];
819     char* xctype = new char[n];
820 
821     for (int i = 0; i < n; i++) {
822         indices[i] = i;
823         xctype[i]  = vtype;
824     }
825     CPXchgctype(env, lp, n, indices, xctype);
826 
827     delete[] indices;
828     delete[] xctype;
829 
830     return 0;
831 }
832 
833 int setTraversalStrategy(CPXENVptr& env, int strategy) {
834 
835     // MIP node selection strategy
836     // http://www-01.ibm.com/support/knowledgecenter/SSSA5P_12.3.0/ilog.odms.cplex.
           help/Content/Optimization/Documentation/Optimization_Studio/_pubskel/
           ps_refparameterscplex2299.html
837 
```

```
838     // 0 CPX_NODESEL_DFS           Depth-first search
839     // 1 CPX_NODESEL_BESTBOUND     Best-bound search; default
840     // 2 CPX_NODESEL_BESTEST       Best-estimate search
841     // 3 CPX_NODESEL_BESTEST_ALT   Alternative best-estimate search
842
843     CPXsetintparam(env, CPX_PARAM_NODESEL, strategy);
844
845     return 0;
846 }
847
848 int setBranchingVariableStrategy(CPXENVptr& env, int strategy) {
849
850     // MIP variable selection strategy
851     // http://www-01.ibm.com/support/knowledgecenter/SS9UKU_12.4.0/com.ibm.cplex.zos.
        help/Parameters/topics/VarSel.html
852
853     // -1   CPX_VARSEL_MININFEAS       Branch on variable with minimum infeasibility
854     // 0    CPX_VARSEL_DEFAULT         Automatic: let CPLEX choose variable to branch
          on; default
855     // 1    CPX_VARSEL_MAXINFEAS       Branch on variable with maximum infeasibility
856     // 2    CPX_VARSEL_PSEUDO          Branch based on pseudo costs
857     // 3    CPX_VARSEL_STRONG          Strong branching
858     // 4    CPX_VARSEL_PSEUDOREDUCED   Branch based on pseudo reduced costs
859
860     CPXsetintparam(env, CPX_PARAM_VARSEL, strategy);
861
862     return 0;
863 }
864
865 int setBranchAndBoundConfig(CPXENVptr& env) {
866
867     // CPLEX config
868     // http://www-01.ibm.com/support/knowledgecenter/SSSA5P_12.2.0/ilog.odms.cplex.
        help/Content/Optimization/Documentation/CPLEX/_pubskel/CPLEX916.html
869
870     // deactivate pre-processing
871     CPXsetintparam(env, CPX_PARAM_PRESLVND, -1);
872     CPXsetintparam(env, CPX_PARAM_REPEATPRESOLVE, 0);
873     CPXsetintparam(env, CPX_PARAM_RELAXPREIND, 0);
874     CPXsetintparam(env, CPX_PARAM_REDUCE, 0);
875     CPXsetintparam(env, CPX_PARAM_LANDPCUTS, -1);
876
877     // maximize objective function
878     // CPXchgobjsen(env, lp, CPX_MAX);
879
880     // enable/disable screen output
881     CPXsetintparam(env, CPX_PARAM_SCRIND, CPX_OFF);
882
883     // set excecution limit
884     CPXsetdblparam(env, CPX_PARAM_TILIM, 3600);
885
886     // disable presolve
887     // CPXsetintparam(env, CPX_PARAM_PREIND, CPX_OFF);
888
889     // enable traditional branch and bound
890     CPXsetintparam(env, CPX_PARAM_MIPSEARCH, CPX_MIPSEARCH_TRADITIONAL);
891
892     // use only one thread for experimentation
893     // CPXsetintparam(env, CPX_PARAM_THREADS, 1);
```

```
894
895     // do not add cutting planes
896     CPXsetintparam(env, CPX_PARAM_EACHCUTLIM, CPX_OFF);
897
898     // disable gomory fractional cuts
899     CPXsetintparam(env, CPX_PARAM_FRACCUTS, -1);
900
901     // measure time in CPU time
902     // CPXsetintparam(env, CPX_PARAM_CLOCKTYPE, CPX_ON);
903
904     return 0;
905 }
906
907
908 int checkStatus(CPXENVptr& env, int status) {
909     if (status) {
910         char buffer[100];
911         CPXgeterrorstring(env, status, buffer);
912         printf("%s\n", buffer);
913         exit(1);
914     }
915     return 0;
916 }
```