Investigación Operativa
Coloreo Particionado de Grafos

7 de diciembre de 2015

Integrante	LU	Correo electrónico
Martin Baigorria	575/14	martinbaigorria@gmail.com
Andrés Armesto Brosio	512/14	andresarmesto@gmail.com

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 agujero Impar

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

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

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

- Si no se asigna el color j_0 a algun vértice 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 $(|C| - 1)/2$ 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

```
procedure GENERATECLIQUEFAMILLY(V, E, sol, threshold, lp)
    set < score, set < int >> clique_familly
    for id ← 1, |V| do
        if sol[id] > 0 + ϵ then
            continue
        end if
        set < int > clique
        clique.insert(id)
        for id2 ← id + 1, |V| do
            if sol[id2] > 0 + ϵ then
                continue
            end if
            if clique.adyacentToAll(id2) then
                clique.insert(id2)
            end if
        end for
        if ¬clique_familly.isContained(clique) then
            clique_familly.insert(< getScore(clique), clique >)
        end if
    end for
    sortByScore(clique_familly)
    addTopCliqueRestrictions(lp, clique_familly, threshold)
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 agujero 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 agujeros impares violados

```
procedure generateOddholeFamilly(V, E, sol, threshold, lp)
    set < score, set < int >> oddhole_familly
    for id ← 1, |V| do
        if sol[id] > 0 + ϵ then
            continue
        end if
        set < int > path
        path.insert(id)
        for id2 ← id + 1, |V| do
            if sol[id2] > 0 + ϵ then
                continue
            end if
            if isAdyacent(path.end, id2) then
                path.insert(id2)
            end if
        end for
        while path.size() ≥ 3 and (path.size() mod 2 == 0 or ¬isAdyacent(path.start, path.end)) do
            path.erase(path.end)
        end while
        if path.size() ≥ 3 and isAdyacent(path.start, path.end) then
            oddhole_familly.insert(< getScore(path), path >)
        end if
    end for
    sortByScore(oddhole_familly)
    addTopPathRestrictions(lp, oddhole_familly, threshold)
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 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. Experimentación

Dada una cantidad de vértices, los grafos se generan con formato DIMACS [1]. El generador toma como parámetro la densidad del grafo. Dada una clique con esa cantidad de vértices, se elijen vértices al azar hasta que se llega a la densidad deseada. Debido a que estas instancias están diseñadas para coloreo de grafos, asignamos los vértices de forma uniforme en el total de particiones pasado por parámetro a nuestro programa de coloreo particionado.

Por cuestiones de tiempo, para cada una de las experimentaciones CPLEX se ejecuto sin limtar la cantidad de threads con un procesador Intel(R) Core(TM) i7-3610QM CPU @ 2.30GHz y 16GB de memoria RAM.

4.1. Eliminación de simetría

Al igual que el problema de coloreo de grafos, el problema del coloreo particionado de grafos presenta una gran cantidad de soluciones simétricas. De no romper la simetría del problema, los algoritmos de búsqueda normalmente tendrían un espacio mucho mayor de búsqueda, lo que afecta el tiempo de ejecución de forma considerable a medida que crece el tamaño del problema. Para romper la simetría en nuestro problema, en la sección 1.3 mostramos como utilizamos la clásica condicion de coloreo de que los colores se deben utilizar en orden. Esto se puede ver en el siguiente gráfico:

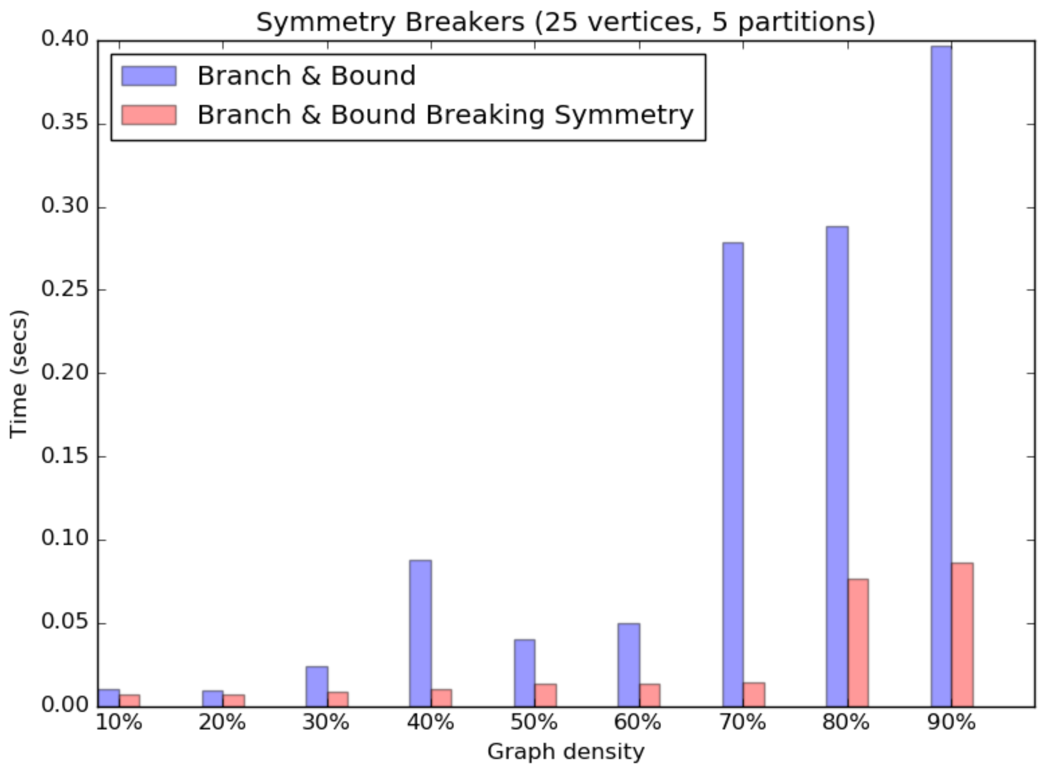

Figura 1: Eliminación de simetría.

Esto nos da una noción sumamente relevante de la importancia y la efectividad de romper la simetría en los LP. Cabe mencionar que existen muchas otras estrategias para romper la simetría de los problemas, y esta no es necesariamente la mejor.

[1] Para ver algunos ejemplos del formato: http://mat.gsia.cmu.edu/COLOR/instances.html

4.2. Efectividad de las familias de desigualdades

La idea de este experimento fue comparar las diferentes estrategias de planos de corte. Para ello se tomo a 40 como el a la cantidad de cortes de cada tipo que se podían agregar, con una sola iteración:

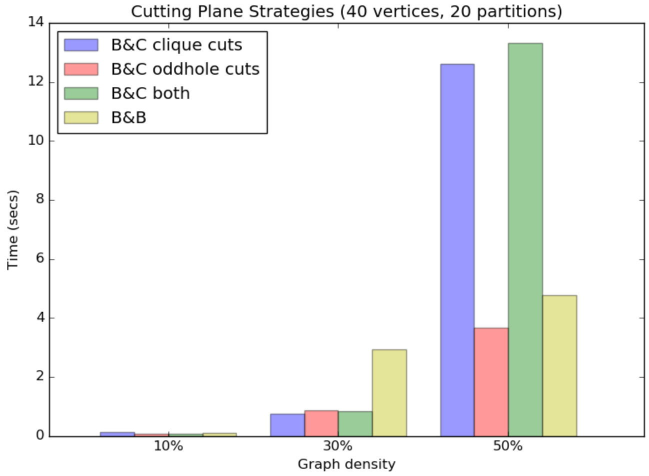

Figura 2: Estrategias de planos de corte (tiempo)

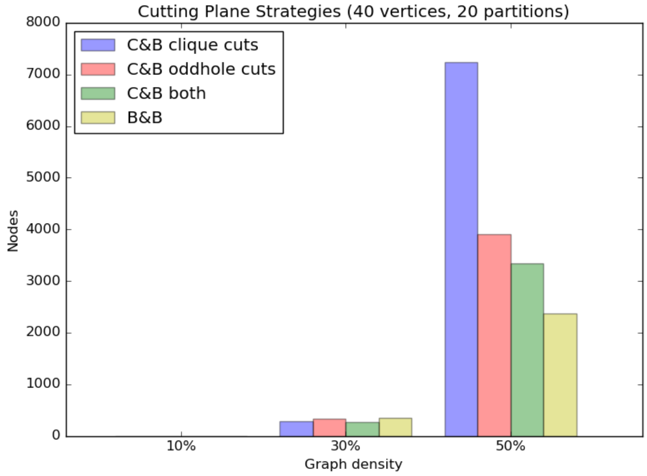

Figura 3: Estrategias de planos de corte (nodos recorridos)

Lo primero que podemos observar que no siempre hay una estrategia ganadora. Claramente la estrategia a seguir depende de la densidad del grafo. Cuanto mas denso, mas cliques nuestra heurística podría encontrar y a priori uno esperaría que los tiempos mejoren. Esto no sucede, y de hecho agregar las restriciones de clique empeora el tiempo de ejecución con respecto al resultado de B&B. También podemos observar que mejor tiempo de ejecución no necesariamente implica que se recorren menos nodos del árbol de enumeración. En contra de lo que esperábamos, las desigualdades de agujero impar parecen funcionar bien, aunque por supuesto se debe llevar a cabo una experimentación mas exhaustiva.

4.3. Efecto de aumentar el numero de particiones

A medida que aumentamos el numero de particiones, el problema comienza a ser mas difícil y a parecerse mas a uno de coloreo. Esto lo podemos ver en el siguiente gráfico. Para Cut & Branch, solo utilizamos los mejores 40 cortes de clique con una iteración. A medida que el problema se hace mas difícil, podemos observar como la ganancia del corte es mayor.

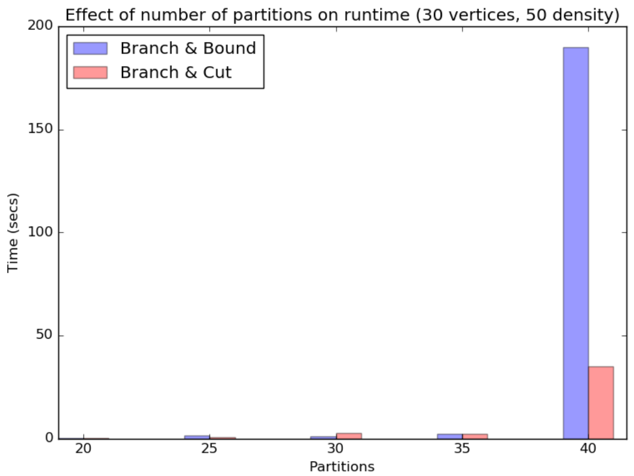

Figura 4: Tiempo de ejecucion a medida que aumenta el numero de particiones.

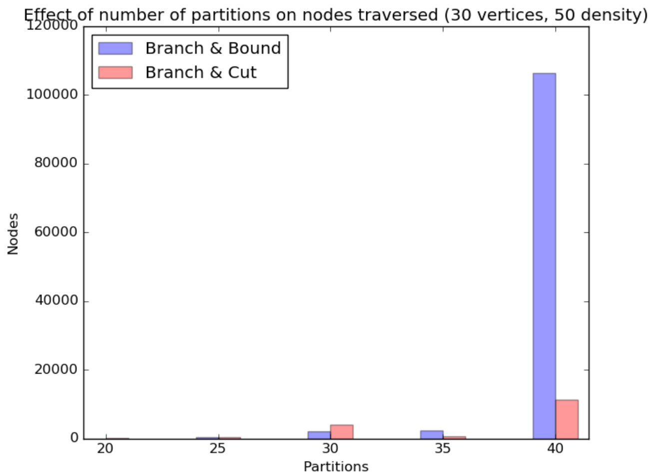

Figura 5: Nodos recorridos a medida que aumenta el numero de particiones.

4.4. Efecto de aumentar la densidad del grafo

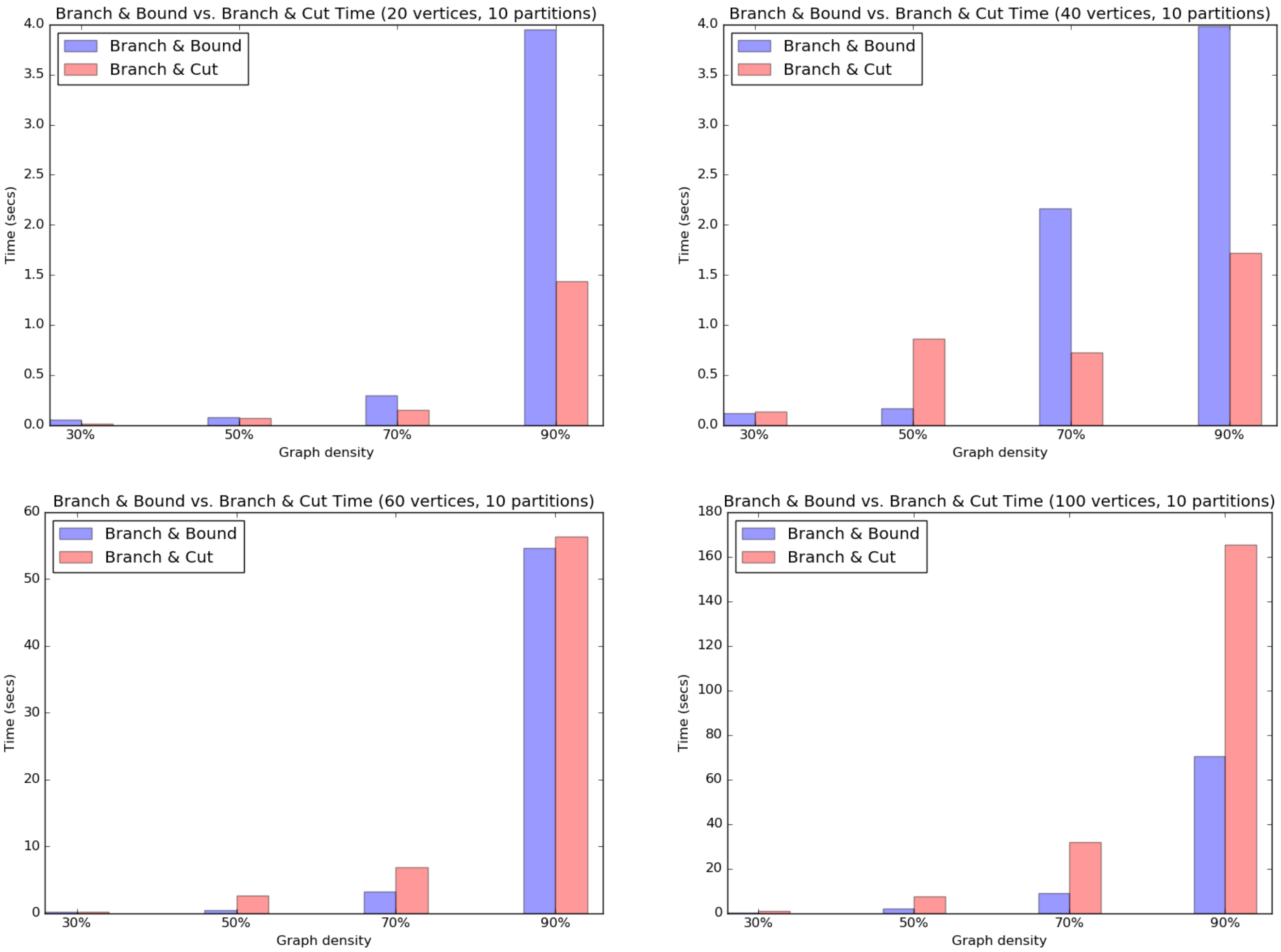

Figura 6: Efecto de aumentar la densidad del grafo.

A medida que aumenta la densidad del grafo, la dificultad del problema es claramente mayor. En los casos donde el numero de particiones es mayor en relación al numero de vértices, Branch & Cut con 1 iteración y 40 desigualdades violadas parece funcionar mejor. Esto no sucede en grafos esparsos, donde Branch & Bound puro tiene un menor tiempo de ejecución.

4.5. Efecto de aumentar la cantidad de restricciones incorporadas por iteración

Para todos nuestros experimentos en general utilizamos solo 1 iteración con un limite de 40 desigualdades por familia. La idea de este experimento fue evaluar esta configuración. Para ello utilizamos un grafo con 40 vértices y 20 particiones.

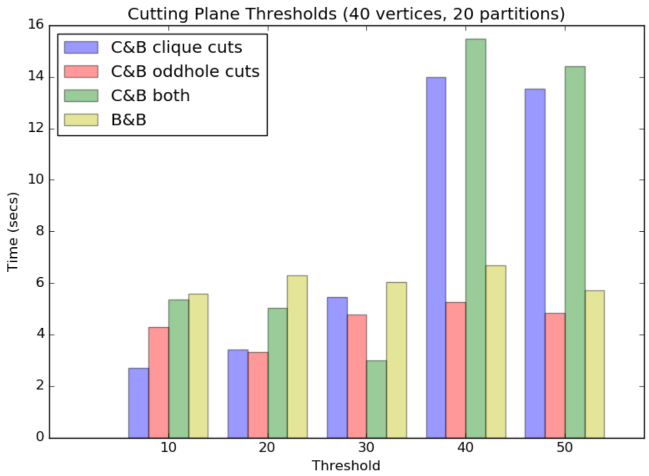

Figura 7: Tiempo de ejecución al incrementar el numero de restricciones incorporadas.

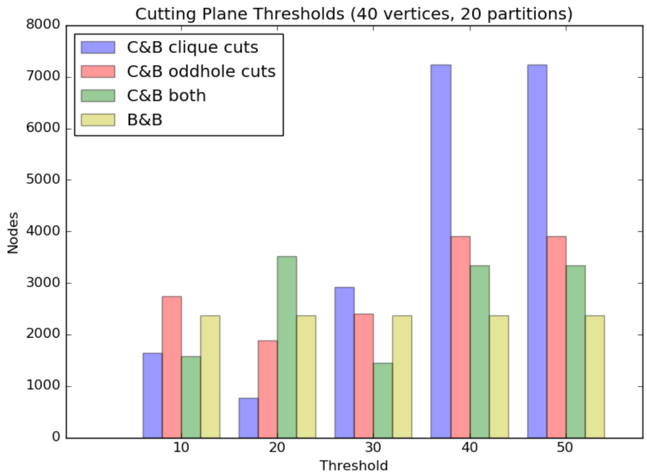

Figura 8: Nodos recorridos al incrementar el numero de restricciones incorporadas.

Como podemos observar, agregar mas restricciones no es siempre ventajoso. En un principio, agregar restricciones parece mejorar la ejecución del C&B, pero ya a partir de 40 el tiempo de ejecución empeora de forma abrupta para las cliques. Esto no sucede para las restricciones de agujero impar. Nuevamente, esto se puede deber a que nuestra heurística de clique no es lo suficientemente buena.

4.6. Efecto de aumentar la cantidad de iteraciones de planos de corte

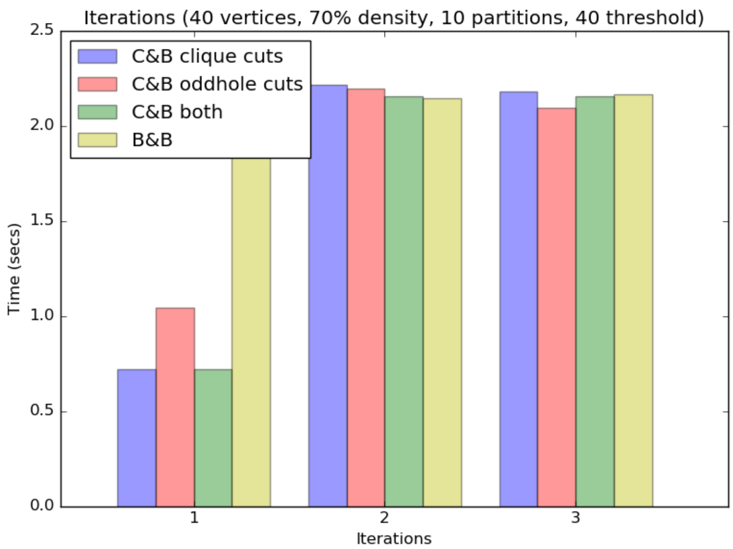

Figura 9: Tiempo de ejecución al aumentar la cantidad de iteraciones de planos de corte.

Como podemos ver, aumentar el numero de iteraciones de planos de corte no necesariamente mejora el tiempo de ejecución. En cada iteración lo que hacíamos era generar una familia en función de la solución de la relajación del problema, y luego agregar las *mejores* restricciones. En relación a la sección anterior, esto también esta relacionado con el *threshold* que elegimos para hacer la experimentación.

4.7. Comparación B&B, C&B, CPLEX default

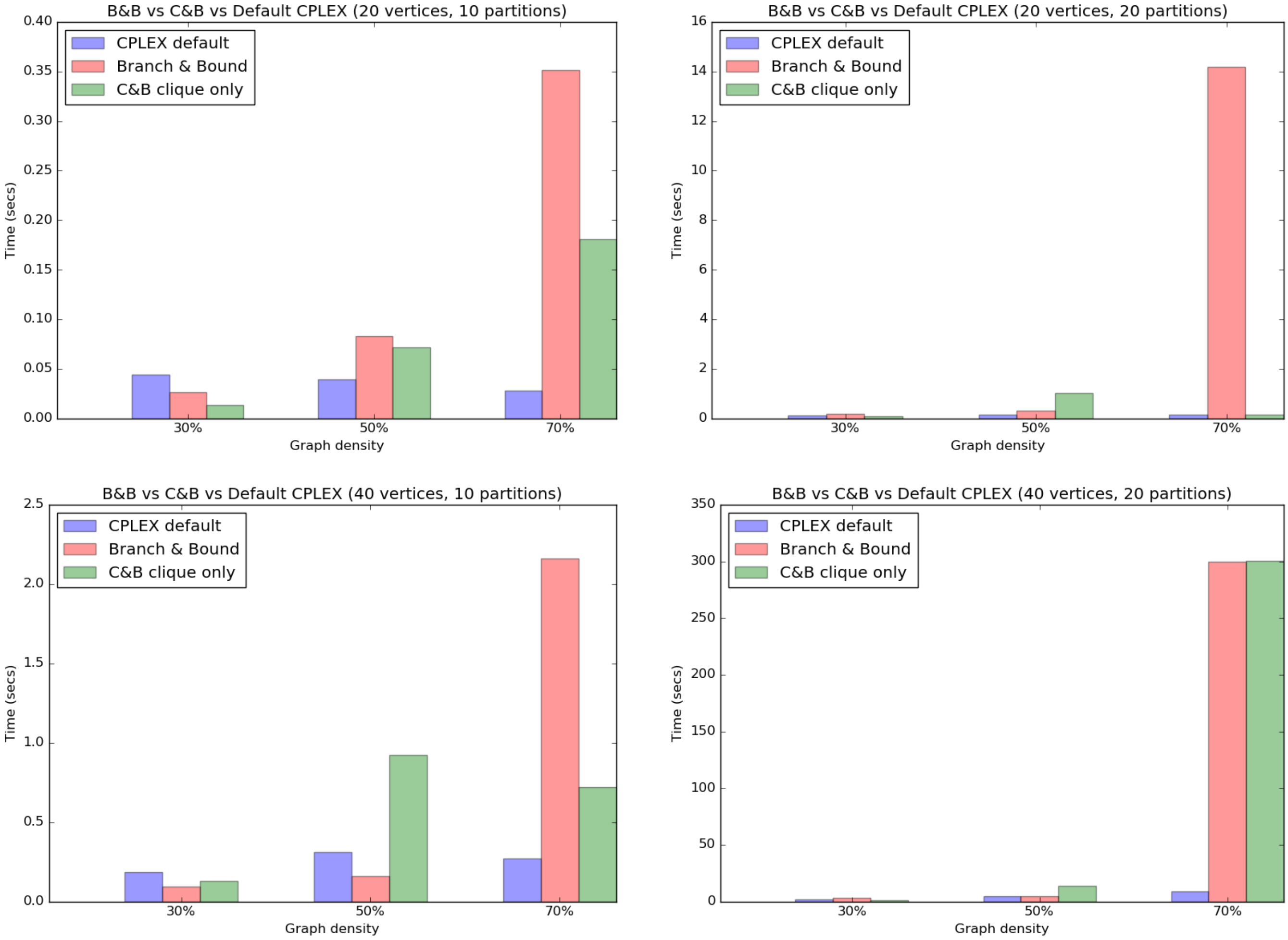

Figura 10: Comparacion B&B, C&B, CPLEX default para diferentes grafos.

Dado que el CPLEX por default utiliza cortes de Gomory y preprocesamiento de variables, no nos sorprende que en general sea superior a nuestras otras estrategias para grafos densos. Una propuesta interesante podría ser repetir esta experimentación permitiendo los cortes y el preprocesamiento para todas nuestras estrategias. Otra observación, el gráfico superior derecho es el caso de coloreo de grafos, dado que cada vértice pertenece a una partición diferente. Aquí podemos ver que las desigualdades de clique son sumamente útiles.

5. Conclusión

6. Apéndice A: Código

6.1. coloring.cpp

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

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

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

#define TOL 1e-05

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);
```

```cpp
int convertVariableType(CPXENVptr& env, CPXLPptr& lp, int vertex_size, int partition_size, char vtype);
int setCPLEXConfig(CPXENVptr& env);
int setTraversalStrategy(CPXENVptr& env, int strategy);
int setBranchingVariableStrategy(CPXENVptr& env, int strategy);
int setBranchAndBoundConfig(CPXENVptr& env);

int checkStatus(CPXENVptr& env, int status);

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

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

    if (argc != 11) {
        printf("Usage: %s inputFile solver partitions symmetry_breaker iterations select_cuts load_limit custom_config traversal_strategy branching_strategy \n", argv[0]);
        exit(1);
    }

    int solver = atoi(argv[2]);
    int partition_size = atoi(argv[3]);
    bool symmetry_breaker = (atoi(argv[4]) == 1);
    int iterations = atoi(argv[5]);
    int select_cuts = atoi(argv[6]);             // 0: clique only, 1: oddhole only, 2: both
    int load_limit = atoi(argv[7]);
    int custom_config = atoi(argv[8]);           // 0: default, 1: custom
    int traversal_strategy = atoi(argv[9]);
    int branching_strategy = atoi(argv[10]);
```

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

    /* read graph input file
     * format: http://mat.gsia.cmu.edu/COLOR/instances.html
     * graph representation chosen in order to load the LP easily.
     * - vector of edges
     * - vector of partitions
     */
    FILE* fp = fopen(argv[1], "r");

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

    char buf[100];
    int vertex_size, edge_size;

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

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

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

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

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

```

```
    for (int i = 0; i < vertex_size; ++i) {
        partitions[i % partition_size].push_back(i+1);
    }

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

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

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

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

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

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

    setCPLEXConfig(env);
    if (custom_config == 1) setBranchAndBoundConfig(env);
    setTraversalStrategy(env, traversal_strategy);
    setBranchingVariableStrategy(env, branching_strategy);

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

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

    if (symmetry_breaker) loadSymmetryBreaker(env, lp, partition_size);

    if (solver != 1) loadCuttingPlanes(env, lp, vertex_size, edge_size, partition_size, adjacencyList, iterations, load_limit, select_cuts);

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

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

```

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

    delete[] adjacencyList;

    return 0;
}

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

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

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

    // assert(from != to && from <= vertex_size && to <= vertex_size);

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

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

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

bool cliqueNotContained(const set<int>& clique, int color, const vector<tuple<double, int, set<int> > >& clique_familly) {
    for (vector<tuple<double, int, set<int> > >::const_iterator it = clique_familly.begin(); it != clique_familly.end(); ++it) {
        // by construction, sets are already ordered.
        if (get<1>(*it) == color && includes(get<2>(*it).begin(), get<2>(*it).end(), clique.begin(), clique.end())) return false;
    }
    return true;
}

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

    // load objective function
    int n = partition_size + (vertex_size*partition_size);
    double *objfun = new double[n];
```

```
    double *ub       = new double[n];
    char      *ctype = new char[n];
    char **colnames = new char*[n];

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

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

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

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

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

    return 0;
}

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

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

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

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

```

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

            if (!isAdyacent(from, to, vertex_size, adjacencyList)) continue;

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

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

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

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

                ++i;
            }
        }
    }

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

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

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

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

    return 0;
}


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

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

        int size = it->size();                    // current partition size.
```

```
        if (size == 0) continue;                // skip empty partitions.

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

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

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

        matbeg[0] = 0;
        sense[0]  = 'E';
        rhs[0]    = 1;
        rownames[0] = new char[40];
        sprintf(rownames[0], "partition_%d", p);

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

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

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

        ++p;
    }

    return 0;
}

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

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

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

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

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

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

        ++i;
    }


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

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

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

    return 0;
}

int loadCuttingPlanes(CPXENVptr& env, CPXLPptr& lp, int vertex_size, int edge_size, int partition_size, bool* adjacencyList, int iterations, int load_limit, int select_cuts) {

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

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

```

```
    int n = partition_size + (vertex_size*partition_size);

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

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

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

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

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

        if (select_cuts == 0 || select_cuts == 2) unsatisfied_restrictions += maximalCliqueFamillyHeuristic(env, lp, vertex_size, edge_size, partition_size, adjacencyList, sol, load_limit);
        if (select_cuts == 1 || select_cuts == 2) unsatisfied_restrictions += oddholeFamillyHeuristic(env, lp, vertex_size, edge_size, partition_size, adjacencyList, sol, load_limit);

        if (unsatisfied_restrictions == 0) break;

        unsatisfied_restrictions = 0;
        ++i;
    }

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

    return 0;
}

int maximalCliqueFamillyHeuristic(CPXENVptr& env, CPXLPptr& lp, int vertex_size, int edge_size, int partition_size, bool* adjacencyList, double* sol, int load_limit) {

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

    int loaded = 0;

    vector<tuple<double, int, set<int> > > clique_familly;

    for (int color = 1; color <= partition_size; ++color) {

        for (int id = 1; id <= vertex_size; id++) {

            if (sol[getVertexIndex(id, color, partition_size)] == 0) continue;
```

```

            double sum = sol[getVertexIndex(id, color, partition_size)];
            set<int> clique;
            clique.insert(id);
            for (int id2 = id + 1; id2 <= vertex_size; ++id2) {
                if (sol[getVertexIndex(id2, color, partition_size)] == 0) continue;

                if (adyacentToAll(id2, vertex_size, adjacencyList, clique)) {
                    clique.insert(id2);
                    sum += sol[getVertexIndex(id2, color, partition_size)];
                }
            }
            if (clique.size() > 2 && sum > sol[color-1] + TOL) {
                if (cliqueNotContained(clique, color, clique_familly)) {
                    double score = sum - sol[color-1];
                    clique_familly.push_back(tuple<double, int, set<int> >(score,
                        color,clique));
                }
            }
        }
    }

    sort(clique_familly.begin(), clique_familly.end(), greater<tuple<double, int, set
        <int> > >());

    //print the familly
    for (vector<tuple<double, int, set<int> > >::const_iterator it = clique_familly.
        begin();
        it != clique_familly.end() && loaded < load_limit; ++loaded, ++it) {

        loadUnsatisfiedCliqueRestriction(env, lp, partition_size, get<2>(*it), get
            <1>(*it));
        cout << "Score: " << get<0>(*it) << " - ";
        for (set<int>::iterator it2 = get<2>(*it).begin(); it2 != get<2>(*it).end();
            ++it2) {
            cout << *it2 << " ";
        }
        cout << endl;
    }

    printf("Loaded %d/%d unsatisfied clique restrictions! (all colors)\n", loaded, (
        int) clique_familly.size());

    return loaded;
}

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

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

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

    int matbeg = 0;
    int* matind    = new int[clique.size() + 1];
    double* matval = new double[clique.size() +1];
```

```
    char **rowname = new char*[rcnt];
    rowname[0] = new char[40];
    sprintf(rowname[0], "unsatisfied_clique");

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

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

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

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

    return 0;
}

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

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

    int loaded = 0;

    vector<tuple<double, int, set<int> > > path_familly; // dif, color, path

    for (int color = 1; color <= partition_size; ++color) {

        for (int id = 1; id <= vertex_size; id++) {

            if (sol[getVertexIndex(id, color, partition_size)] == 0) continue;

            double sum = 0;
            set<int> path;
            path.insert(id);
            for (int id2 = id + 1; id2 <= vertex_size; ++id2) {
                if (sol[getVertexIndex(id2, color, partition_size)] == 0) continue;

                if (isAdyacent(*(--path.end()), id2, vertex_size, adjacencyList)) {
                    path.insert(id2);
                }

            }

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

```
            }

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

            int k = (path.size() - 1) / 2;
            if (path.size() > 2 && sum > k*sol[color-1] + TOL) {
                double score = sum - k*sol[color-1];
                path_familly.push_back(tuple<double, int, set<int> >(score,color,path
                    ));
            }
        }
    }

    sort(path_familly.begin(), path_familly.end(), greater<tuple<double, int, set<int
        > > >());

    //print the familly
    for (vector<tuple<double, int, set<int> > >::const_iterator it = path_familly.
        begin();
        it != path_familly.end() && loaded < load_limit; ++loaded, ++it) {
        loadUnsatisfiedOddholeRestriction(env, lp, partition_size, get<2>(*it), get
            <1>(*it));
        cout << "Score: " << get<0>(*it) << " - ";
        for (set<int>::iterator it2 = get<2>(*it).begin(); it2 != get<2>(*it).end();
            ++it2) {
            cout << *it2 << " ";
        }
        cout << endl;
    }

    printf("Loaded %d/%d unsatisfied oddhole restrictions! (all colors)\n", loaded, (
        int) path_familly.size());

    return loaded;
}

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

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

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

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

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

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

```

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

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

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

    return 0;
}

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

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

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

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

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

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

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

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

            ++i;
```

```
        }
    }

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

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

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

    return 0;
}

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

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

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

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

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

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

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

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

```
    checkStatus(env, status);

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

    int nodes_traversed = CPXgetnodecnt(env, lp);

    // write solutions to current window
    cout << "Optimization result: " << statstring << endl;
    cout << "Time taken to solve final LP: " << (endtime - inittime) << endl;
    cout << "Colors used: " << objval << endl;
    cout << "Nodes traversed: " << nodes_traversed << endl;
    for (int color = 1; color <= partition_size; ++color) {
        if (sol[color-1] == 1) {
            cout << "w_" << color << " = " << sol[color-1] << " (" << colors[color-1]
                << ")" << endl;
        }
    }

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

    delete[] sol;

    return 0;
}

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

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

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

    delete[] indices;
    delete[] xctype;

    return 0;
}

int setTraversalStrategy(CPXENVptr& env, int strategy) {

    // MIP node selection strategy
    // http://www-01.ibm.com/support/knowledgecenter/SSSA5P_12.3.0/ilog.odms.cplex.
        help/Content/Optimization/Documentation/Optimization_Studio/_pubskel/
```

```
ps_refparameterscplex2299.html

    // 0 CPX_NODESEL_DFS            Depth-first search
    // 1 CPX_NODESEL_BESTBOUND      Best-bound search; default
    // 2 CPX_NODESEL_BESTEST        Best-estimate search
    // 3 CPX_NODESEL_BESTEST_ALT    Alternative best-estimate search

    CPXsetintparam(env, CPX_PARAM_NODESEL, strategy);

    return 0;
}

int setBranchingVariableStrategy(CPXENVptr& env, int strategy) {

    // MIP variable selection strategy
    // http://www-01.ibm.com/support/knowledgecenter/SS9UKU_12.4.0/com.ibm.cplex.zos.help/Parameters/topics/VarSel.html

    // -1   CPX_VARSEL_MININFEAS        Branch on variable with minimum infeasibility
    // 0    CPX_VARSEL_DEFAULT          Automatic: let CPLEX choose variable to branch on; default
    // 1    CPX_VARSEL_MAXINFEAS        Branch on variable with maximum infeasibility
    // 2    CPX_VARSEL_PSEUDO           Branch based on pseudo costs
    // 3    CPX_VARSEL_STRONG           Strong branching
    // 4    CPX_VARSEL_PSEUDOREDUCED    Branch based on pseudo reduced costs

    CPXsetintparam(env, CPX_PARAM_VARSEL, strategy);

    return 0;
}

int setCPLEXConfig(CPXENVptr& env) {
    // maximize objective function
    // CPXchgobjsen(env, lp, CPX_MAX);

    // enable/disable screen output
    CPXsetintparam(env, CPX_PARAM_SCRIND, CPX_OFF);

    // set excecution limit
    CPXsetdblparam(env, CPX_PARAM_TILIM, 300);

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

    return 0;
}

int setBranchAndBoundConfig(CPXENVptr& env) {

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

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

```

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

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

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

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

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

    return 0;
}


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