Investigación Operativa
Coloreo Particionado de Grafos

11 de diciembre de 2015

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

Resumen: El presente trabajo práctico tiene como objetivo resolver el problema del coloreo particionado de grafos utilizando programación lineal. Para ello, en un primer momento modelamos el problema y lo implementamos utilizando CPLEX. Se experimenta con diferentes configuraciones de los métodos Branch & Bound y Branch & Cut, evaluando diferentes heurísticas, configuraciones y variando los tipos de grafos a resolver.

Keywords: Linear Programming, Partitioned Graph Coloring, Branch & Bound, Cut & Branch, CPLEX.

Índice

1. Modelo **3**
- 1.1. Función objetivo . . . 3
- 1.2. Restricciones . . . 3
- 1.3. Eliminación de simetrías . . . 3

2. Branch & Bound **4**

3. Desigualdades **4**
- 3.1. Desigualdad de Clique . . . 4
- 3.2. Desigualdad de Agujero Impar . . . 5
- 3.3. Planos de Corte . . . 5
- 3.4. Heurísticas . . . 5
  - 3.4.1. Heurística de Separación para Clique Maximal . . . 6
  - 3.4.2. Heurística de Separación para Agujero Impar . . . 7
- 3.5. Cut & Branch . . . 7

4. Experimentación **8**
- 4.1. Eliminación de simetría . . . 8
- 4.2. Efectividad de las familias de desigualdades . . . 9
- 4.3. Efecto de aumentar el número de particiones . . . 9
- 4.4. Efecto de aumentar la densidad del grafo . . . 10
- 4.5. Efecto de aumentar la cantidad de restricciones incorporadas por iteración . . . 11
- 4.6. Efecto de aumentar la cantidad de iteraciones de planos de corte . . . 11
- 4.7. Comparación B&B, C&B, CPLEX default . . . 12
- 4.8. Estrategias de recorrido del árbol de enumeración y selección de variable de branching . . . 12
- 4.9. Instancias DIMACS . . . 13

5. Conclusión **17**

6. Apéndice A: Código **18**
- 6.1. coloring.cpp . . . 18

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 generalización 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 sólo 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.

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

Sólo se le asigna un color a un único vértice de cada partición $p \in P$. Esto implica que cada vértice tiene a lo sumo sólo 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 simetría fue usar la clásica condición de coloreo que establece que los colores se deben utilizar en orden. Aunque existen otras, notamos que esta condición mejoró ampliamente la ejecución del LP. Formalmente, se puede expresar como:

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

2. Branch & Bound

Los algoritmos de tipo Branch & Bound se utilizan para resolver problemas de programación lineal entera (PLE) o programación lineal entera mixta (PLEM, PEM, o MIP en inglés). El algoritmo consiste en tomar el problema de PEM y resolver en primera instancia la relajación lineal, es decir aquella que relaja las condiciones de integralidad sobre las variables. Con esto se obtiene un x^*, es decir una solución óptima de la relajación lineal. Si la solución obtenida cumple con las condiciones de integralidad, se ha encontrado un óptimo y el algoritmo termina. Si existe al menos una variable que no la cumple, se parte el problema original en 2 o más subproblemas. El proceso de partición se llama *branching*, y existe diversidad de criterios para realizarlo. Sin embargo, todas las formas de branching cumplen que todos los puntos factibles del problema original deben estar en alguna partición, y que el x^* hallado no pertenece a ninguna de ellas, de forma de no caer nuevamente en él. El proceso se repite en cada subproblema que nace, y termina cuando no quedan nodos por explorar. A su vez, otra parte importante del algoritmo consiste en cortar aquellas ramas cuyo valor óptimo de la relajación lineal es peor que el valor óptimo obtenido hasta ese momento. A este fenómeno se lo llama poda, o *bounding* en inglés.

Los criterios más importantes a determinar en un algoritmo de Branch & Bound son cómo realizar el branching (qué variables) y qué nodos a explorar, o cómo recorrer el árbol de enumeración (verticalmente, horizontalmente, etc.).

La implementación del modelo y del Branch & Bound se encuentran en el apéndice.

3. Desigualdades

3.1. Desigualdad de Clique

Sea $j_0 \in \{1, ..., |C|\}$ 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 sólo un vértice, no existe arista que contenga este vértice, caso contrario la clique tendría al menos 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 asigne el color j_0. La otra opción es que, dada la estructura de particiones, se le asigne sólo a uno de ellos el color j_0. Por lo tanto, vale la desigualdad para $n = 2$.

3. $n = 3$: Este es el caso más 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 1/2 y sumando entonces:

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

   Desarrollando: $x_{1j_0} + x_{2j_0} + x_{3j_0} \leq 3/2$.

   Como x_{ij} toma valores enteros, se implica: $x_{1j_0} + x_{2j_0} + x_{3j_0} \leq 1$

   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, por lo que deben cumplir:
$x_{1j_0} + x_{nj_0} \leq 1$, $x_{2j_0} + x_{nj_0} \leq 1$,..., $x_{(n-1)j_0} + x_{nj_0} \leq 1$
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 el color j_0, las restricciones de las aristas agregadas agregamos hacen que 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 color, o se le asigna un color diferente a j_0, se obtiene la expresión de la hipótesis inductiva, y 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:

$$\sum_{p \in C_{2k+1}} x_{pj_0} \leq k w_{j_0} \tag{9}$$

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

- Si no se asigna el color j_0 a algún vértice del agujero, la desigualdad vale.
- Si se asigna el color j_0, en el peor de los casos el coloreo particionado coincide con el coloreo tradicional. En ese caso, se asignará el color j_0 a lo sumo a $(|C|-1)/2$ vértices. Dado $|C| = 2k+1$, $(2k+1-1)/2 = k$. Por lo tanto vale la desigualdad. □

3.3. Planos de Corte

Los algoritmos de planos de corte comenzaron a estudiarse en los años 60's. Fueron introducidos por Ralph E. Gomory, a quien luego se le sumó Václav Schvátal. En sus términos básicos, en un primer paso se resuelve la relajación lineal del problema. Luego, el procedimiento termina si se verifica que el problema es infactible, o si se halla una solución que cumple las condiciones de integralidad. En caso de que no ocurra esto, los *algoritmos de separación* buscan identificar desigualdades lineales que permitan separar el óptimo fraccionario hallado de los puntos enteros factibles. De este modo, se intenta que el poliedro se parezca más a la cápsula convexa. Se dice que la solución fraccionaria *viola* la desigualdad hallada, y al buscarla se debe garantizar que no queden puntos enteros factibles fuera del nuevo poliedro. Una vez encontradas una o más desigualdades válidas, se agregan a la formulación y se resuelve nuevamente la relajación lineal. Se repite el proceso hasta que se encuentre una solución que cumpla con la integralidad de las variables, el problema resulte infactible, o no se pueda obtener más desigualdades válidas.

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 más desigualdades de la clase violadas la solución x^*.

Debido a la naturaleza heurística del algoritmo, es posible que exista una desigualdad de la clase violada, aunque éste sea incapaz 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 válidas 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, se recurre a algoritmos heurísticos 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 sólo 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 asimismo reduce la complejidad espacial de forma considerable. La fórmula que utilizamos para generar la biyección entre arista e índice en la lista puede verse 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 garantizarán que nuestro conjunto de índices que representa a un miembro de una familia está ordenado. Esto es muy ventajoso en el sentido que podemos saber fácilmente si un nuevo potencial miembro de la familia está 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 cuáles son las mejores desigualdades a agregar, y luego definir un *threshold* para decidir cuántas agregamos al LP. El criterio que utilizamos es el módulo 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 más 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 en orden los vértices que tienen una solución positiva en la relajación del LP. 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 elementos en orden de forma que cumplan la 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 GENERATECLIQUEFAMILY(V, E, sol, threshold, lp)
    set < score, set < int >> clique_family
    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_family.isContained(clique) then
            clique_family.insert(< getScore(clique), clique >)
        end if
    end for
    sortByScore(clique_family)
    addTopCliqueRestrictions(lp, clique_family, threshold)
end procedure
```

Notar que en la práctica sólo 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, lo que no fue mostrado para facilitar la visualización del algoritmo.

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 GENERATEODDHOLEFAMILY(V, E, sol, threshold, lp)
    set < score, set < int >> oddhole_family
    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_family.insert(< getScore(path), path >)
        end if
    end for
    sortByScore(oddhole_family)
    addTopPathRestrictions(lp, oddhole_family, threshold)
end procedure
```

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

3.5. Cut & Branch

Dado que los algoritmos de planos de corte no necesariamente convergen, decidimos implementar un algoritmo Cut & Branch. Los algoritmos Cut & Branch buscan aplicar planos de corte siguiendo alguna estrategia en los nodos 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 & Bound.

En nuestra implementación, aplicaremos planos de corte solo en la raíz del árbol de enumeración. A su vez, debemos tener en cuenta que 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 la cantidad de vértices, los grafos se generan en el formato estándar 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, cada uno de los experimientos CPLEX fue ejecutado sin límite de 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 tendrían un espacio de búsqueda mucho mayor, moviéndose por soluciones que, siendo computacionalmente distintas, en la práctica se trata de la misma. Esto 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 cómo utilizamos la clásica condicion de coloreo de que los colores se deben utilizar en orden. Este fenómeno se puede ver en el siguiente gráfico:

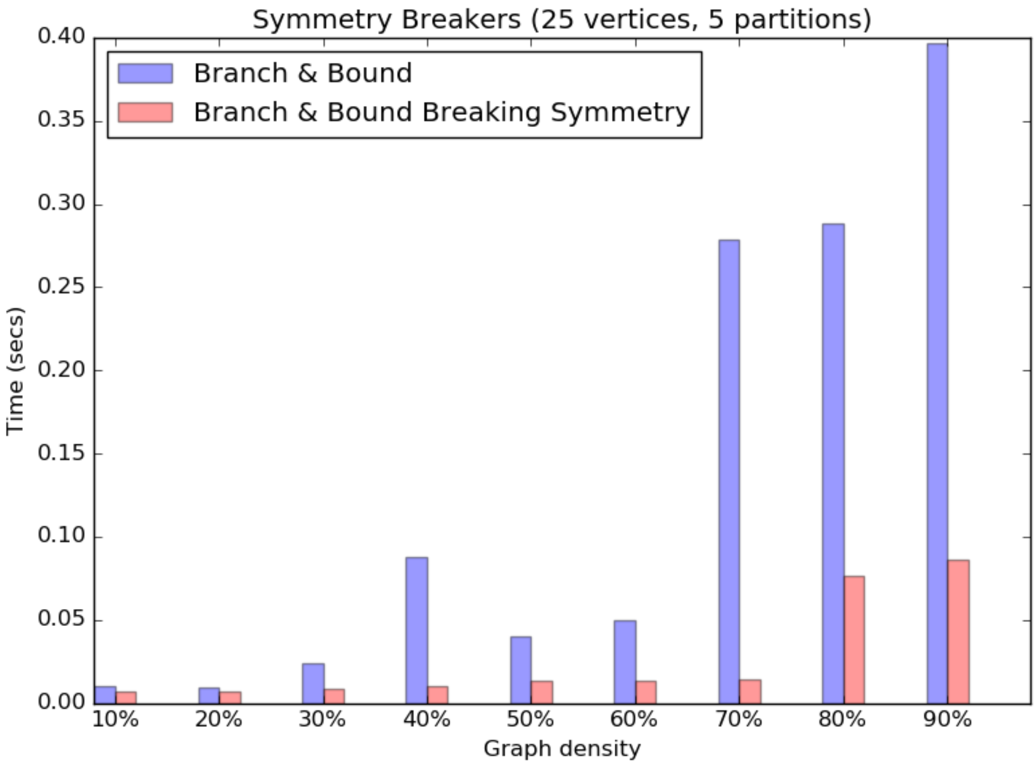

Figura 1: Tiempo de resolución del modelo incluyendo o no eliminación de simetría.

Esto nos brinda la noción de la importancia y efectividad de romper simetría al realizar la formulación de un LP. Cabe mencionar que existen muchas otras estrategias o expresiones para disminuir aun más el grado de simetría de la formulación. La formulación elegida no debe ser considerada necesarimante la mejor posible.

[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 es comparar las diferentes estrategias de planos de corte. Para ello, se eligió a 40 como la cantidad de cortes de cada tipo que se podían agregar, con una sola iteración:

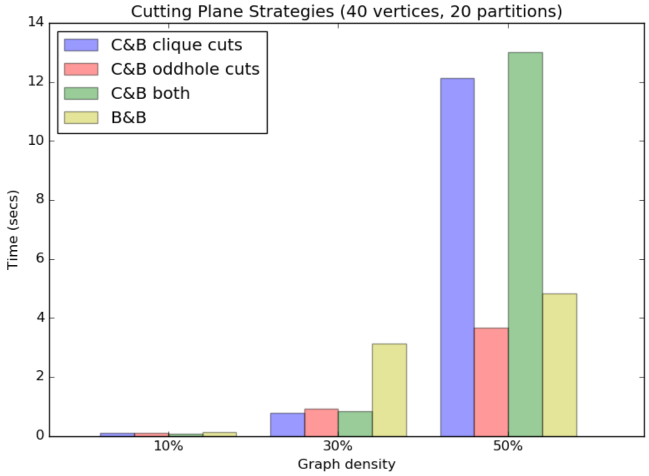

Figura 2: Estrategias de planos de corte (tiempo).

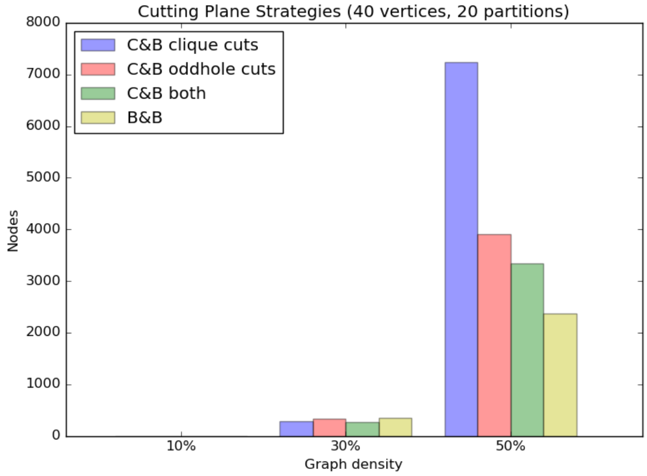

Figura 3: Estrategias de planos de corte (nodos).

Lo primero que podemos observar es que no siempre hay una estrategia ganadora por sobre las otras. Al aumentar la densidad del grafo, pueden más distinguidamente los distintos métodos.

En el caso de los grafos más densos, uno esperaría a priori que nuestra heurística encuentre más cliques, y que los tiempos sean mejores. Esto no sucede, y de hecho, agregar las restriciones de clique empeora el tiempo de ejecución con respecto a utilizar simplemente B&B. En los grafos menos densos, puede notarse un efecto positivo de los cortes, es decir preferencia del C&B sobre el B&B.

En contra de lo que esperábamos inicialmente, las desigualdades de agujero impar parecen funcionar bien, realizando mejores tiempos que el B&B. Sin embargo, esta hipótesis podría constatarse con mayor peso de llevar a cabo una experimentación más exhaustiva.

Por último, también podemos observar que un mejor tiempo de ejecución no necesariamente implica que se recorren menos nodos del árbol. Esto puede ser interesante tenerlo en cuenta en el caso de que uno compare en qué lugar se consume el tiempo de ejecución, o para evaluar distintas formas de poda del árbol.

4.3. Efecto de aumentar el número de particiones

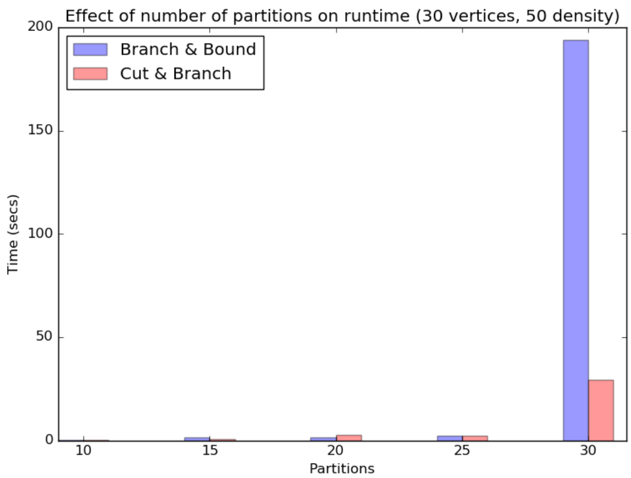

Figura 4: Tiempo de ejecucion a medida que aumenta el número de particiones.

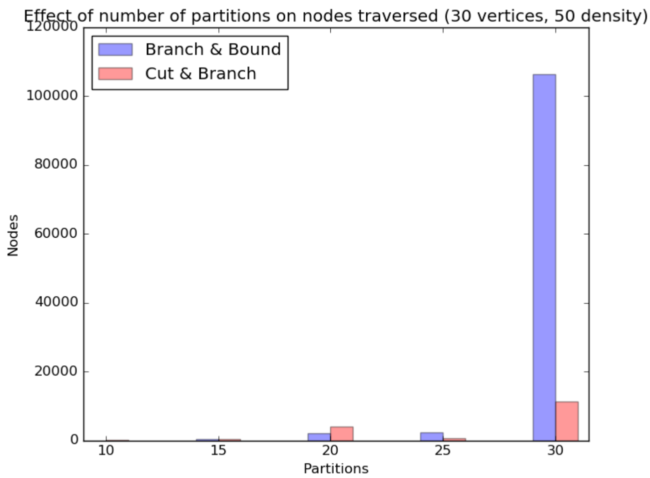

Figura 5: Nodos recorridos a medida que aumenta el número de particiones.

A medida que aumentamos el número de particiones, el problema comienza a parecerse más a uno de coloreo. Dado que las desigualdades que implementamos son clásicas de coloreo, es de esperar que la performance mejore a medida que aumenta el número de particiones. Para Cut & Branch, sólo utilizamos los mejores 40 cortes de clique con una iteración. A medida que aumenta el número de particiones, podemos observar que el efecto positivo de incluir los cortes es considerable.

En cuanto a los nodos recorridos, puede verse que en el caso del C&B la cantidad es notablemente menor. Se atribuye esto a la virtud de los cortes de generar una buena solución al inicio, permitiendo descartar ramas del árbol.

4.4. Efecto de aumentar la densidad del grafo

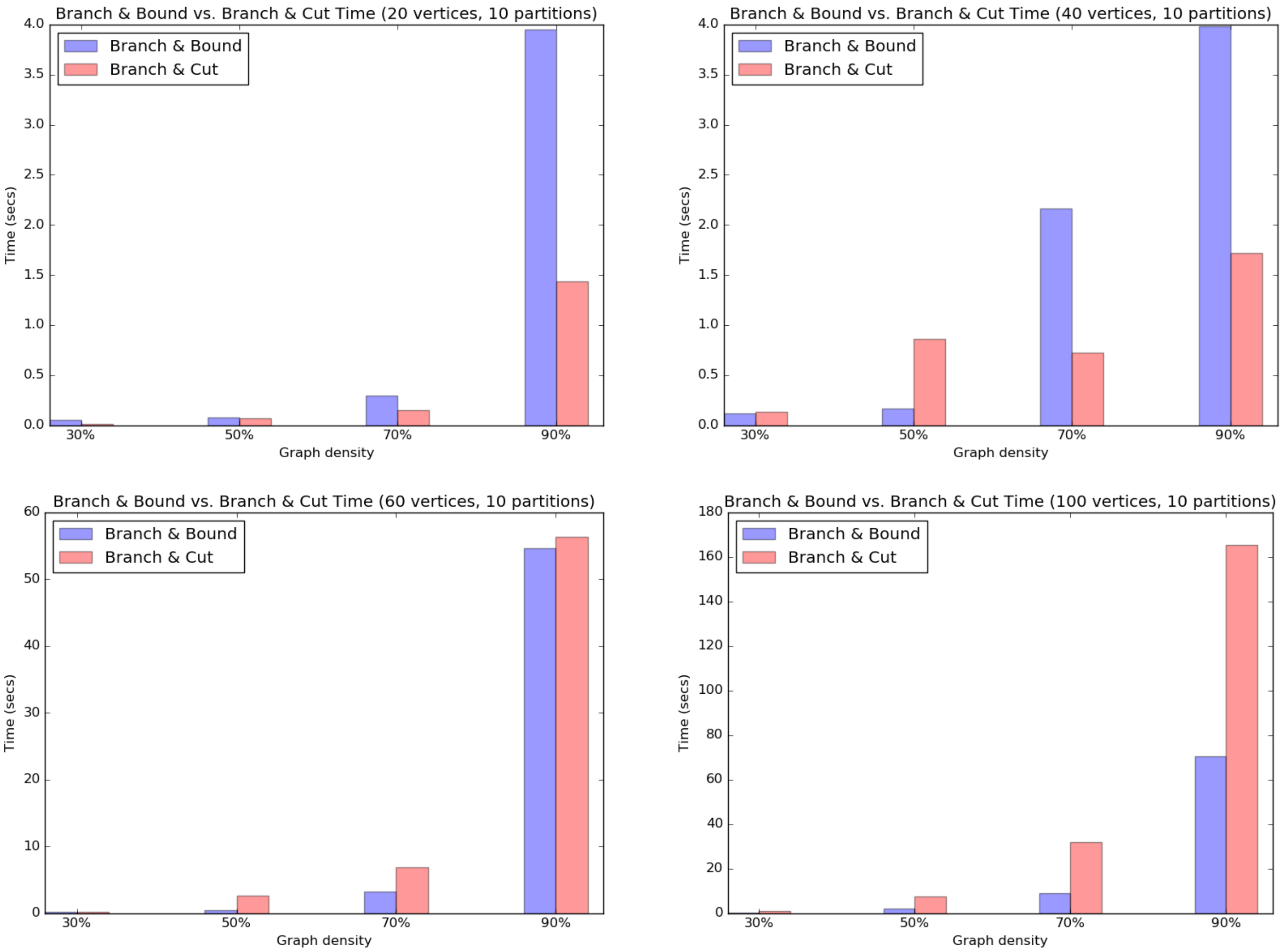

Figura 6: Efecto de aumentar la densidad del grafo.

A medida que aumenta la densidad del grafo, el problema de coloreo se vuelve sin duda más difícil. En gráficos más densos y cuya relación de número de particiones sobre número de vértices es mayor se puede apreciar una tendencia a funcionar mejor del Branch & Cut (nuevamente se utiliza 1 iteración y 40 desigualdades violadas). Sin embargo, en grafos esparsos, podemos notar una mayor efectividad del Branch & Bound puro, en cuanto a tiempos de ejecución. De cualquier modo, no es posible obtener resultados muy generales, sino más bien mejores o peores algoritmos para cierta densidad y cantidad de particiones.

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

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

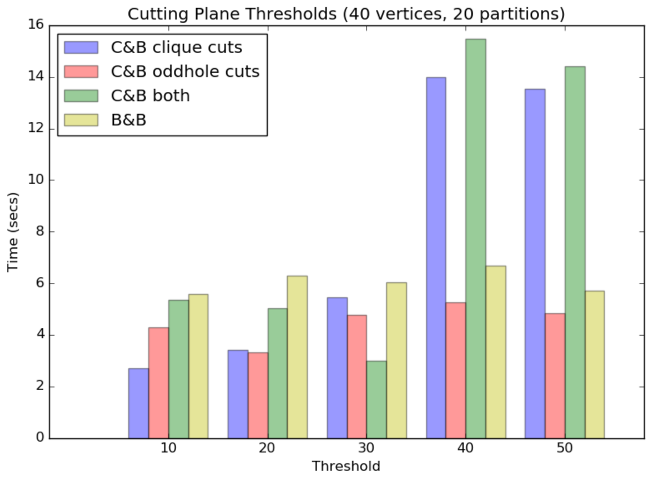

Figura 7: Tiempo de ejecución al incrementar el número de restricciones incorporadas.

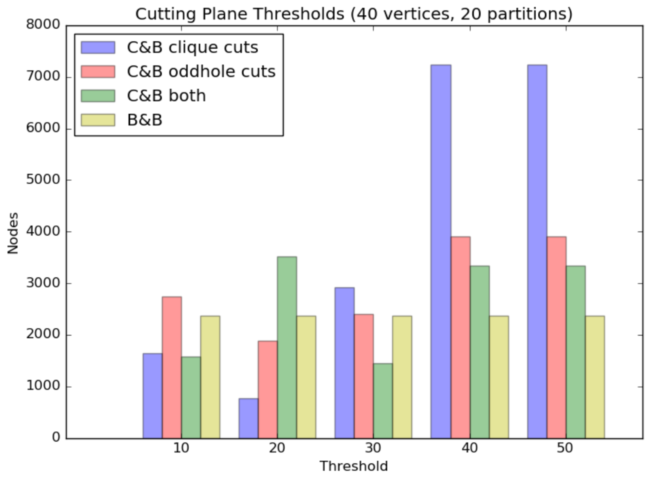

Figura 8: Nodos recorridos al incrementar el número de restricciones incorporadas.

Como podemos concluirse de los gráficos, agregar más restricciones no es siempre ventajoso. En un principio, sumar restricciones parece mejorar la ejecución del C&B (para el caso de ambas familias), pero ya a partir de las 40 el tiempo de ejecución empeora de forma abrupta para las cliques. Esto no sucede para las restricciones de agujero impar, cuyo tiempo de ejecución no posee gran variabilidad con respecto al *threshold*.

Claro está, que este incremento en el tiempo puede estar relacionado con la heurística de clique utilizada, que puede no se lo suficientemente buena. Debido al saldo importante que ocurre con los cortes por cliques, el *threshold* de 30 se considera empíricamente aceptable.

Por otro lado, es visible en el primer gráfico una dependencia del tiempo de ejecución del B&B con respecto al *threshold* analizado. Sabemos que esta variabilidad no corresponde, y se atribuye a ruido del CPU.

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

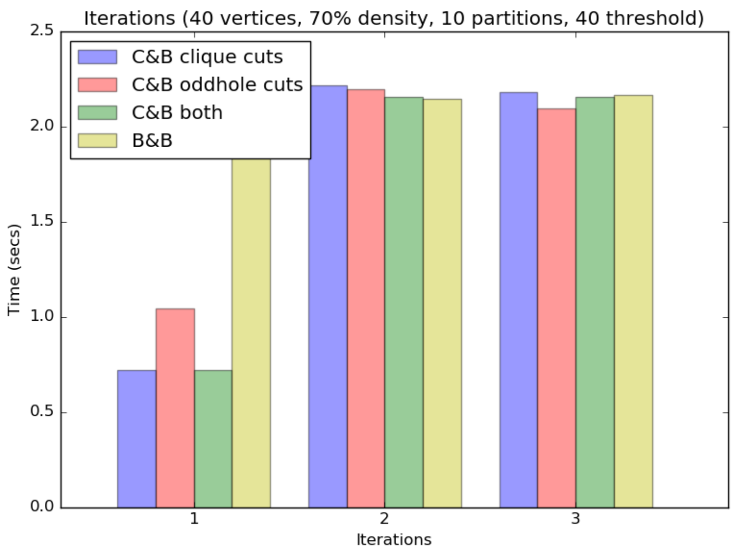

Figura 9: Tiempo de ejecución al aumentar la cantidad de iteraciones de planos de corte.

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

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

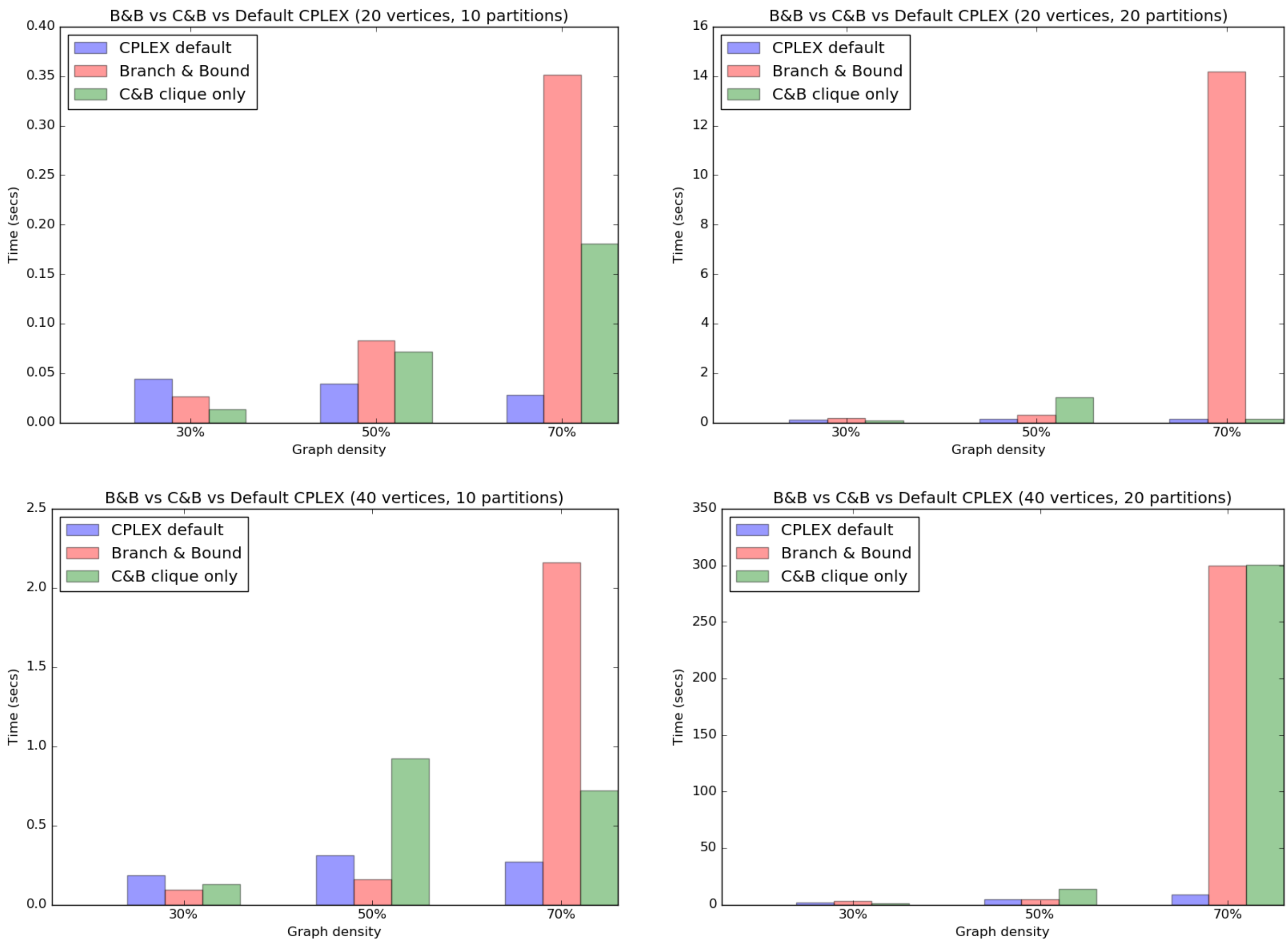

Figura 10: Comparacion B&B, C&B, CPLEX default para diferentes grafos.

Dado que el CPLEX utiliza por default cortes de Gomory y preprocesamiento de variables, no nos sorprende que en general para grafos densos sea notablemente superior. Una propuesta interesante podría ser repetir esta experimentación permitiendo los cortes y el preprocesamiento para todas nuestras estrategias.

Al mismo tiempo, es importante observar el gráfico superior derecho, que corresponde al coloreo de grafos estándar (cada vértice pertenece a una partición diferente). En él se puede apreciar la gran utilidad de las desigualdades de clique para disminuir tiempos de ejecución, en el caso de grafos densos.

4.8. Estrategias de recorrido del árbol de enumeración y selección de variable de branching

Existen muchas estrategias de recorrido del árbol de enumeración. En este trabajo analizaremos únicamente Depth First Search (DFS) y Best Bound Search (BBS). DFS recorre el árbol de enumeración de B&B primero en profundidad, mientras que BBS recorre el árbol de enumeración buscando una buena cota lo más rápido posible, de modo de realizar una buena poda. En general, se utilizan estrategias heurísticas. En el caso de CPLEX, dado un nodo padre se calcula la solución a la relajación de todos sus hijos y luego se continua recorriendo el nodo con el mayor resultado de la función objetivo. [2].

Ambas estrategias son sumamente ventajosas, ya que permiten obtener una cota superior a la solución final para utilizar de poda al hacer backtracking sobre el árbol de enumeración. Dado que no utilizamos heurísticas iniciales, esta estrategia parece razonable.

Por otro lado, las estrategias de selección de variable buscan encontrar cuál es la mejor variable sobre la que hacer branching. Hay muchas reglas, como por ejemplo *max/min infeasibility*. Mientras que la regla de *minimum infeasibility* hace branching sobre aquella variable más cercana al valor entero, la regla de *maximum infeasibility* hace exactamente lo contrario [3].

[2] http://www-01.ibm.com/support/knowledgecenter/SSSA5P_12.6.1/ilog.odms.cplex.help/CPLEX/Parameters/topics/NodeSel.html
[3] http://www-01.ibm.com/support/knowledgecenter/SS9UKU_12.4.0/com.ibm.cplex.zos.help/Parameters/topics/VarSel.html

En esta sección analizaremos 4 combinaciones de estrategias de recorrido del árbol de enumeración y selección de variable de branching para B&B puro y C&B con cortes de clique, 1 iteración y *threshold* 30. Las combinaciones que analizaremos son DFS + MAXINFEAS, DFS + MININFEAS, BESTBOUND + MAXINFEAS y BESTBOUND + MININFEAS.

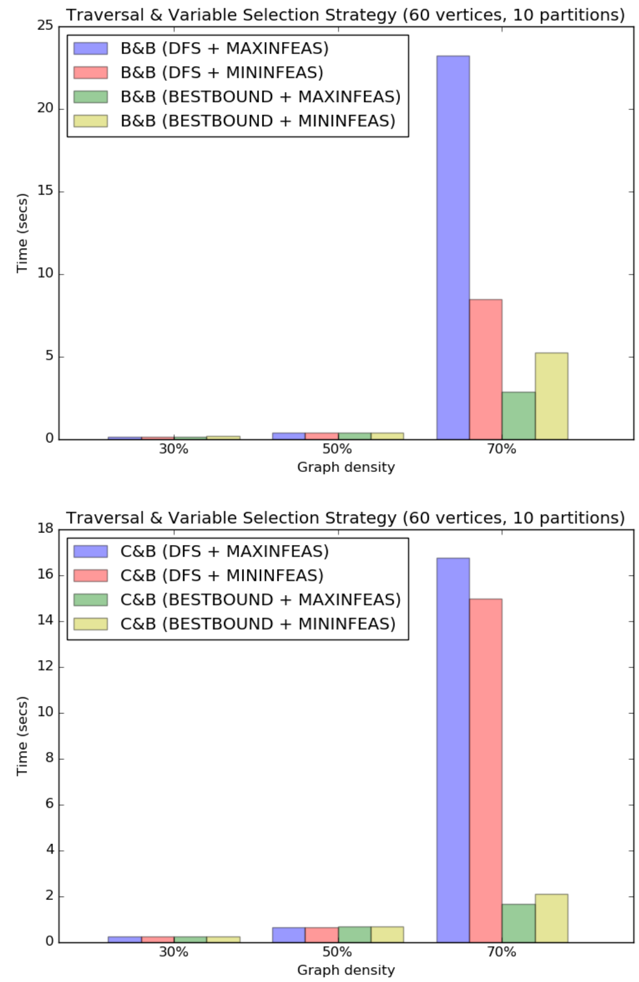

Figura 11: Tiempo de ejecución dependiendo de la estrategias de recorrido y selección de variable.

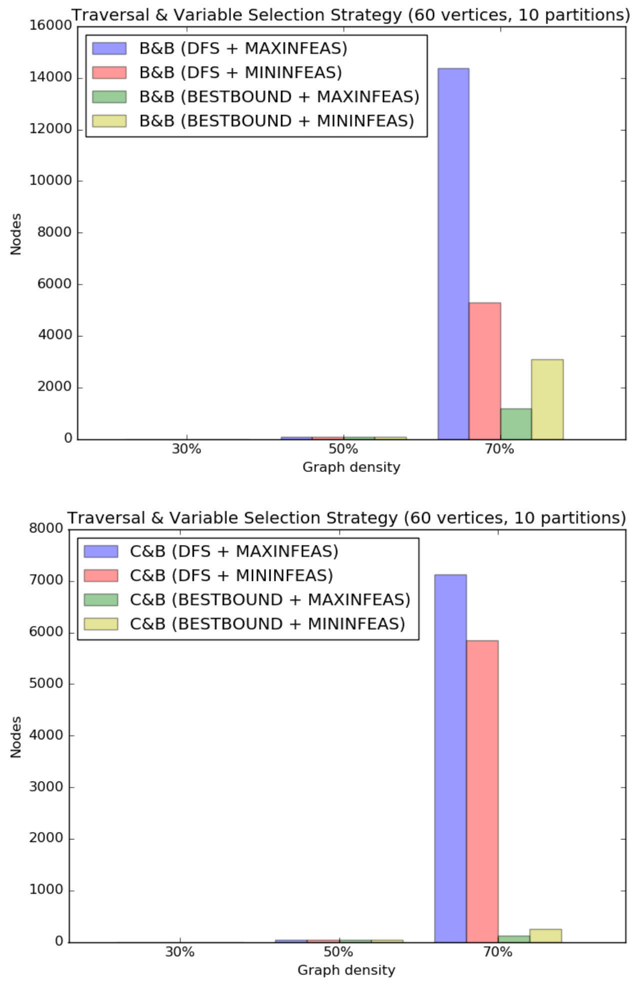

Figura 12: Nodos recorridos dependiendo de la estrategias de recorrido y selección de variable.

Como podemos observar, en general C&B tiene menores tiempos de ejecución, y recorre menos nodos. Al mismo tiempo, de las 4 combinaciones testeadas, la mejor estrategia para este problema parece ser BESTBOUND + MAXINFEAS. CPLEX utiliza por default BESTBOUND, aunque recurre a una heurística para elegir la variable de branching.

4.9. Instancias DIMACS

Las instancias DIMACS son comúnmente utilizadas en la literatura como instancias de benchmarking. A continuación se muestran nuestros tiempos de ejecución con B&B y B&C utilizando 1 iteración y *threshold* 30, utilizando sólo desigualdades de clique. La regla utilizada para recorrer el árbol de enumeración es la que utiliza el CPLEX por default.

Cada tabla utiliza fue elaborada utilizando un número de particiones diferente. Dado que las instancias DIMACS no tienen asociadas un número de partición (éstas se utilizan comúnmente para coloreo estándar), asignamos uno nosotros, y luego dividimos los vértices en orden en las diferentes particiones de forma uniforme.

Tomamos como tiempo límite de ejecución 10 minutos, y reportamos el número de colores encontrado por B&B hasta ese momento. En todos los casos, B&B y B&C coincidieron con el número de colores, por lo que los reportamos en una única columna.

En este punto, quizás sería interesante probar con otras configuraciones de C&B para los problemas más difíciles. Para esos problemas, C&B parece mostrar una mejor performance.

Cuadro 1: Benchmark con 10 particiones.

Problema	n	m	Tiempo B&B (secs)	Tiempo B&C (secs)	Colores utilizados
anna	138	493	0.04	0.50	1
david	87	406	0.02	0.29	1
fpsol2.i.1	496	11654	0.51	8.71	1
fpsol2.i.2	451	8691	0.44	7.66	1
fpsol2.i.3	425	8688	0.45	8.15	1
games120	120	638	0.04	0.32	1
homer	561	1629	0.11	0.72	1
huck	74	301	0.02	0.10	1
inithx.i.1	864	18707	0.79	9.66	1
inithx.i.2	645	13979	0.59	6.53	1
inithx.i.3	621	13969	0.59	6.16	1
jean	80	254	0.02	0.10	1
le450_15a	450	8168	0.31	14.68	1
le450_15b	450	8169	0.35	15.68	1
le450_15c	450	16680	0.64	36.05	1
le450_15d	450	16750	0.83	38.00	1
le450_25a	450	8260	0.34	24.95	1
le450_25b	450	8263	0.33	15.41	1
le450_25c	450	17343	0.70	42.17	1
le450_25d	450	17425	0.70	39.60	1
le450_5a	450	5714	0.27	8.19	1
le450_5b	450	5734	0.30	13.78	1
le450_5c	450	9803	0.46	29.12	1
le450_5d	450	9757	0.47	33.13	1
miles1000	128	3216	8.81	7.77	2
miles1500	128	5198	600.00	600.00	3
miles250	128	387	0.03	0.24	1
miles500	128	1170	0.06	0.54	1
miles750	128	2113	0.32	2.73	1
mulsol.i.1	197	3925	0.15	2.20	1
mulsol.i.2	188	3885	0.14	2.01	1
mulsol.i.3	184	3916	0.16	3.02	1
mulsol.i.4	185	3946	0.15	4.37	1
mulsol.i.5	186	3973	0.15	3.42	1
myciel3	11	20	0.01	0.01	3
myciel4	23	71	0.01	0.02	1
myciel5	47	236	0.01	0.04	1
myciel6	95	755	0.04	0.29	1
myciel7	191	2360	0.09	1.75	1
queen10_10	100	1470	0.12	0.61	1
queen11_11	121	1980	0.18	1.03	1
queen12_12	144	2596	0.36	2.29	1
queen13_13	169	3328	0.44	2.73	1
queen14_14	196	4186	0.18	5.05	1
queen15_15	225	5180	0.23	7.02	1
queen16_16	256	6320	0.23	8.62	1
queen5_5	25	160	0.06	0.14	3
queen6_6	36	290	0.12	0.54	2
queen7_7	49	476	0.12	0.41	2
queen8_12	96	1368	3.54	3.96	2
queen8_8	64	728	0.26	0.64	2
queen9_9	81	1056	1.76	2.54	2
school1	385	19095	1.17	90.04	1
school1_nsh	352	14612	0.83	46.13	1
zeroin.i.1	211	4100	0.16	4.87	1
zeroin.i.2	211	3541	0.17	3.82	1
zeroin.i.3	206	3540	0.25	1.91	1

Cuadro 2: Benchmark con 20 particiones.

Problema	n	m	Tiempo B&B (secs)	Tiempo B&C (secs)	Colores utilizados
anna	138	493	0.07	0.49	1
david	87	406	0.04	0.42	1
fpsol2.i.1	496	11654	1.10	34.07	1
fpsol2.i.2	451	8691	0.75	17.08	1
fpsol2.i.3	425	8688	0.83	17.88	1
games120	120	638	1.85	2.00	1
homer	561	1629	0.39	2.35	1
huck	74	301	0.08	0.18	1
inithx.i.1	864	18707	1.95	36.07	1
inithx.i.2	645	13979	1.42	16.36	1
inithx.i.3	621	13969	1.38	17.08	1
jean	80	254	0.04	0.22	1
le450_15a	450	8168	0.93	66.87	1
le450_15b	450	8169	0.87	58.17	1
le450_15c	450	16680	2.06	120.52	1
le450_15d	450	16750	4.21	197.03	1
le450_25a	450	8260	0.97	96.22	1
le450_25b	450	8263	1.14	52.73	1
le450_25c	450	17343	2.94	155.22	1
le450_25d	450	17425	2.49	122.38	1
le450_5a	450	5714	1.25	65.50	1
le450_5b	450	5734	0.68	59.74	1
le450_5c	450	9803	2.04	110.64	1
le450_5d	450	9757	1.31	103.62	1
miles1000	128	3216	600.00	600.00	3
miles1500	128	5198	600.00	600.00	6
miles250	128	387	0.06	0.60	1
miles500	128	1170	16.50	17.50	2
miles750	128	2113	142.62	29.64	2
mulsol.i.1	197	3925	0.67	11.13	1
mulsol.i.2	188	3885	0.65	10.69	1
mulsol.i.3	184	3916	0.79	15.27	1
mulsol.i.4	185	3946	0.77	16.79	1
mulsol.i.5	186	3973	0.71	25.82	1
myciel3	11	20	0.14	0.49	4
myciel4	23	71	0.44	0.54	3
myciel5	47	236	0.072	0.24	1
myciel6	95	755	0.15	1.21	1
myciel7	191	2360	0.39	11.53	1
queen10_10	100	1470	11.04	23.04	2
queen11_11	121	1980	19.26	56.27	2
queen12_12	144	2596	27.91	32.60	2
queen13_13	169	3328	51.07	19.02	2
queen14_14	196	4186	600.00	153.14	2
queen15_15	225	5180	600.00	39.00	2
queen16_16	256	6320	600.00	600.00	2
queen5_5	25	160	1.00	1.00	5
queen6_6	36	290	2.80	4.29	4
queen7_7	49	476	13.84	25.16	4
queen8_12	96	1368	300.04	301.40	3
queen8_8	64	728	23.12	42.98	3
queen9_9	81	1056	600.00	99.08	3
school1	385	19095	8.57	119.25	1
school1_nsh	352	14612	4.31	86.76	1
zeroin.i.1	211	4100	0.40	6.32	1
zeroin.i.2	211	3541	0.32	4.08	1
zeroin.i.3	206	3540	0.32	6.75	1

Aclaración: en la segunda tabla la cantidad de particiones en la instancia *myciel3* coincidió con la cantidad de vértices.

Al ver ambas tablas, puede observarse que los algoritmos concluyeron para la mayoría de las instancias testeadas. También, puede verse que en muchos casos fue necesario con un sólo color para realizar el coloreo, debido a la baja cantidad de particiones en relación con la cantidad de vértices. Este mismo fenómeno se aprecia en los grafos más chicos, donde al duplicar la cantidad de particiones, instancias de menos de 40 o 50 vértices incrementaron la cantidad de colores utilizados, al estar más próximo a un coloreo estándar.

Si miramos las instancias *miles*, puede verse como aumentan los tiempos al incrementar la densidad del grafo. Por otro lado, instancias bien densas como las *school* muestran una marcada diferencia entre el tiempo por B&B y C&B. Esto puede atribuirse a un largo tiempo de ejecución de las heurísticas de separación, debido a la gran cantidad de aristas.

Mirando la tabla de 20 particiones, puede notarse que existen casos en que el B&B no pudo terminar y el C&B sí lo hizo. Existen también otros casos aislados en los que los tiempos de C&B son menores que los de B&B. En términos generales, el desempeño de B&B fue rotundamente mejor, por lo que se entiende que la complejización del algoritmo y todo su desarrollo para incluir cortes debe ser hecho en la medida en que sea necesario, y no antes.

5. Conclusión

El famoso problema de coloreo de grafos, que ha sido estudiado ampliamente en la literatura, es un caso particular del coloreo particionado de grafos, donde cada vértice pertenece a una partición diferente. Por esta razón, en primer lugar notamos que el problema del coloreo particionado sería al menos tan difícil como el problema de coloreo estándar, que de por sí representa un problema complicado.

Las desigualdades de planos de cortes que se han implementado en este trabajo son desigualdades utilizadas normalmente para coloreo. Por esta razón, a lo largo de todo el trabajo hemos notado que los algoritmos de Cut & Branch tienden a funcionar mejor en los casos en que la relación cantidad de particiones sobre cantidad de nodos es mayor. La intuición nos sugiere que deben existir mejores familias, que exploten el hecho de que el grafo esté dividido en particiones, si bien encontrarlas y desarrollarlas escape del alcance de este trabajo.

Uno de los primeros problemas que encontramos al programar el algoritmo de Cut & Branch fue encontrar buenas heurísticas para las familias de desigualdades que probamos válidas. A priori sabíamos, por ejemplo, que generar todas las cliques de manera exhaustiva es un problema NP-Hard. Aquí es donde interviene sin duda la creatividad del investigador para el diseño de los algoritmos, y obtener una heurística que genere un buen resultado. A lo largo de este trabajo probamos varias estrategias, y finalmente nos quedamos con una que depende de la solución de la relajación en cada iteración. Sin embargo, no tenemos ninguna duda de que existen heurísticas mucho más efectivas. A su vez, una vez encontrado este conjunto de desigualdades violadas, es sumamente importante establecer un criterio para decidir cuáles deben ser agregadas al programa lineal. Se pudo comprobar, en términos generales, que agregarlas todas hace que la optimización sea más lenta.

Por otro lado, en nuestro caso hemos notado que el tiempo de cómputo no está dominado por la generación de estas familias, sino por la resolución del programa lineal. La explicación más probable de esto es que el algoritmo de Cut & Branch implementado realiza cortes únicamente en el nodo raíz (una o más iteraciones), por lo que la cantidad de llamados a las heurísticas no crece al recorrer el árbol. En el caso más general, esto no se cumple, y es tarea del investigador procurar un balance entre el tiempo de ejecución de la heurística y el tiempo de ejecución necesario para resolver el programa lineal. Por supuesto, también es importante descubrir en qué tipos de problemas (tamaño, características particulares, etc) conviene utilizar determinadas estrategias como cortes específicos, o diferentes modos de recorrer el árbol de enumeración. En definitiva, el diseño del algoritmo requiere conocer las distintas variables intervinientes, y cómo éstas afectan el resultado final. Es por ello que las horas experimentación con diversas instancias son una etapa clave para conocer el proceso y poder refinarlo. Posteriormente, se podrá acomodarlo o refinarlo, para obtener buenos resultados ya sea en una instancia en particular, o en términos más generales de aplicación.

A lo largo de este trabajo, la performance de CPLEX nos sorprendió notablemente. Por default, CPLEX en sí funciona bastante bien. Conseguir una mejor descripción de la cápsula convexa muchas veces es difícil, y *en términos prácticos* puede llegar a no valer la pena. Debe balancearse el esfuerzo y la dificultad con el tiempo de cómputo y la calidad de la solución obtenida. Por supuesto, también influye el tamaño de instancia que uno necesite resolver. Aquí es donde entra CPLEX, que con una formulación simple del PPL logra optimizar el problema relativamente bien para instancias razonablemente chicas. Como comentario, comprobamos la suma importancia de romper con la simetría de los problemas. Esto en general no representa gran dificultad adicional, y mejora los tiempos de ejecución de forma considerable.

En cuanto a oportunidades de mejora, se podría generalizar el algoritmo Branch & Cut, agregando heurísticas iniciales y primales (qué mejoran el cálculo de cotas, la inicialización y la poda de ramas), o encontrando mejores estrategias de cortes en diferentes partes del árbol, utilizando los *callbacks* de CPLEX. Asimismo, podrían incluirse estrategias de preprocesamiento.

Existe gran cantidad de parámetros a calibrar para lograr una buena performance, y su elección en general se basa en una experimentación que logre emular los casos más comunes en la práctica. Durante este trabajo, no se experimentó en profundidad con instancias y problemas sumamente difíciles debido al tiempo acotado para realizar el mismo. Sería interesante, sin embargo, ver hasta qué punto pueden llegar los algoritmos con una buena configuración.

Referencias

[1] IBM. Ilog cplex optimization studio 12.6.1. `http://www-01.ibm.com/support/knowledgecenter/SSSA5P_12.6.1/ilog.odms.cplex.help/refcppcplex/html/overview.htm`.

[2] Isabel Méndez-Díaz and Paula Zabala. A branch-and-cut algorithm for graph coloring. *Discrete Applied Mathematics*, 154(5):826–847, 2006.

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

// 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 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;
            }
        }
    }
```

```cpp

    // 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);
```

```cpp

    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;
}

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);

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

    delete[] adjacencyList;

    return 0;
}
```