# Investigación Operativa
# Coloreo Particionado de Grafos

8 de diciembre de 2015


| Integrante | LU | Correo electrónico |
|---|---|---|
| Martin 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

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 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:

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

# 2. Branch & Bound

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

# 3. Desigualdades

## 3.1. Desigualdad de Clique

Sea $j_0 \in \{1, ..., n\}$ 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 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:
   - $x_{1j_0} + x_{2j_0} \leq 1$
   - $x_{2j_0} + x_{3j_0} \leq 1$
   - $x_{1j_0} + x_{3j_0} \leq 1$

   Duda: Multiplicando todas estas desigualdades por 1/2 y sumando entonces: $1/2(x_{1j_0} + x_{2j_0}) + 1/2(x_{2j_0} + x_{3j_0}) + 1/2(x_{2j_0} + x_{3j_0}) \leq 3/2$ 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 $n = 3$.

   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:

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

$$\sum_{p \in K-(n-1)} x_{pj_0} \leq w_{j_0}$$

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:

$$x_{nj_0} + \sum_{p \in K-(n-1)} x_{pj_0} \leq w_{j_0}$$

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, es decir aquella que relaja las condiciones de integralidad sobre las variables. Luego, el procedimiento termina si se verifica que el problema es infactible, o si se halla una solución que cumple las condicioens 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. A continuación, se vuelve a resolver la relajación lineal, agregando en la formulación la desigualdad encontrada. 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 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, 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 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 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 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

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

Notar que en la 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

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

Notar que en ambas heurísticas utilizamos la tolerancia $\epsilon$ 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 & Bound.

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 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:

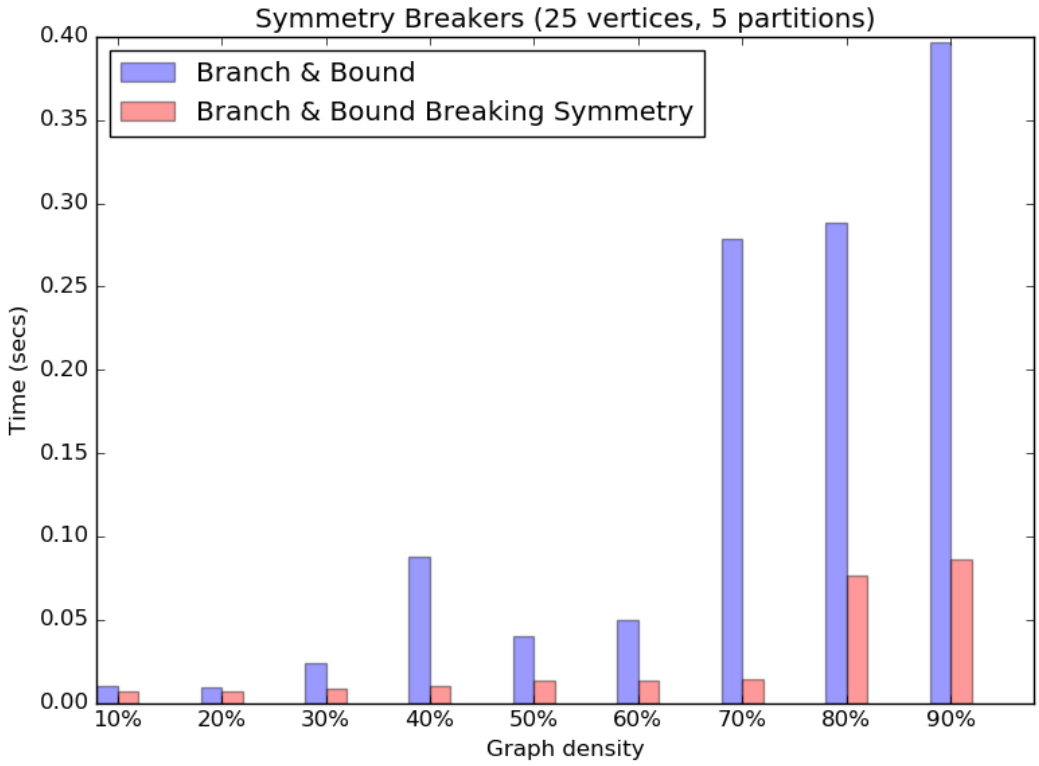



Figura 1: Tiempo de resolución del modelo incluyendo o no eliminación de simetría.

Esto nos brinda la noción sumamente relevante 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 escogida bajo ninguna circunstancia debe ser considerada 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:

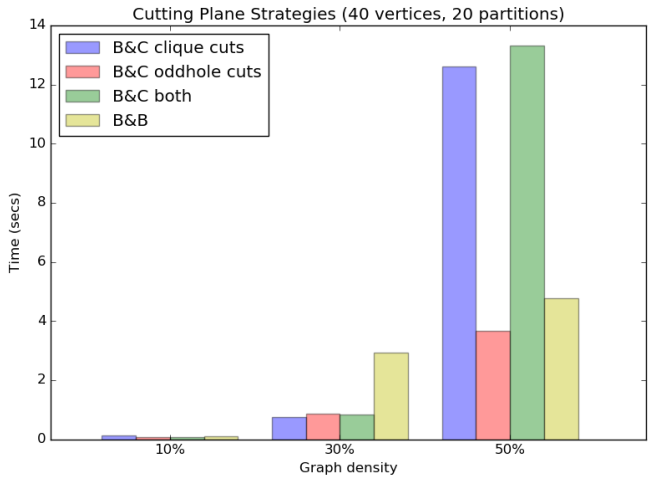



Figura 2: Estrategias de planos de corte (tiempo)

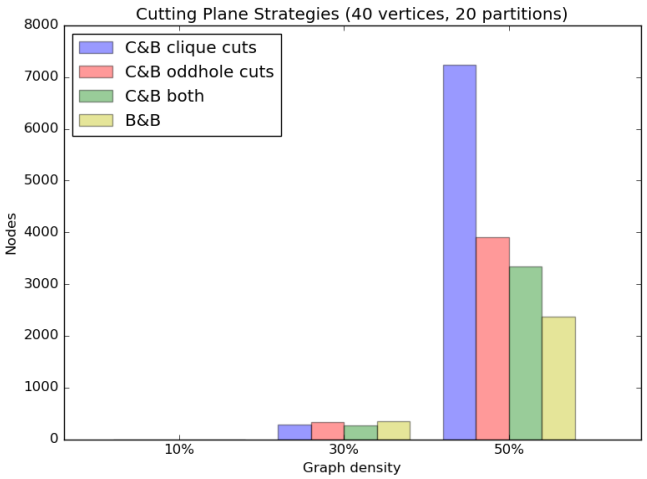



Figura 3: Estrategias de planos de corte (nodos recorridos)

Lo primero que podemos observar es que no siempre hay una estrategia ganadora por sobre las otras. Se observa con claridad una dependencia entre la densidad del grafo y la estrategia que tuvo mejores resultados. Cuanto más denso, más cliques nuestra heurística debería encontrar, y a priori uno esperaría que los tiempos mejoren. Esto no sucede, de hecho agregar las restriciones de clique empeora el tiempo de ejecución con respecto al resultado de utilizar B&B. También podemos observar que un mejor tiempo de ejecución no necesariamente implica que se recorren menos nodos en árbol de enumeración. En contra de lo que esperábamos inicialmente, las desigualdades de agujero impar parecen funcionar bien, aunque por supuesto esto se podría constatar con mayor peso de llevar a cabo una experimentación mas exhaustiva.

## 4.3. Efecto de aumentar 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 [2]. Para Cut & Branch, sólo utilizamos los mejores 40 cortes de clique con una iteración. A medida que aumenta el número de particiones, podemos observar cómo la ganancia del corte es mayor.

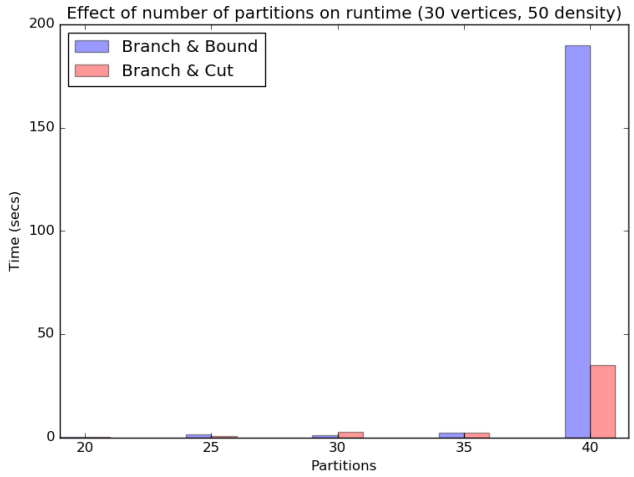



Figura 4: Tiempo de ejecucion a medida que aumenta el numero de particiones.

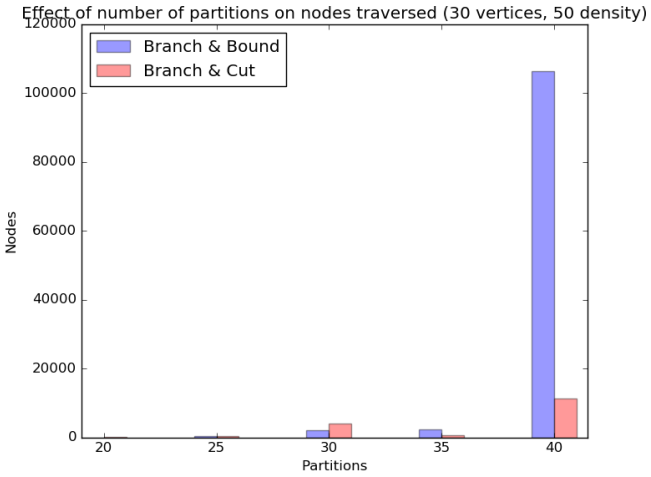



Figura 5: Nodos recorridos a medida que aumenta el numero de particiones.

## 4.4. Efecto de aumentar la densidad del grafo

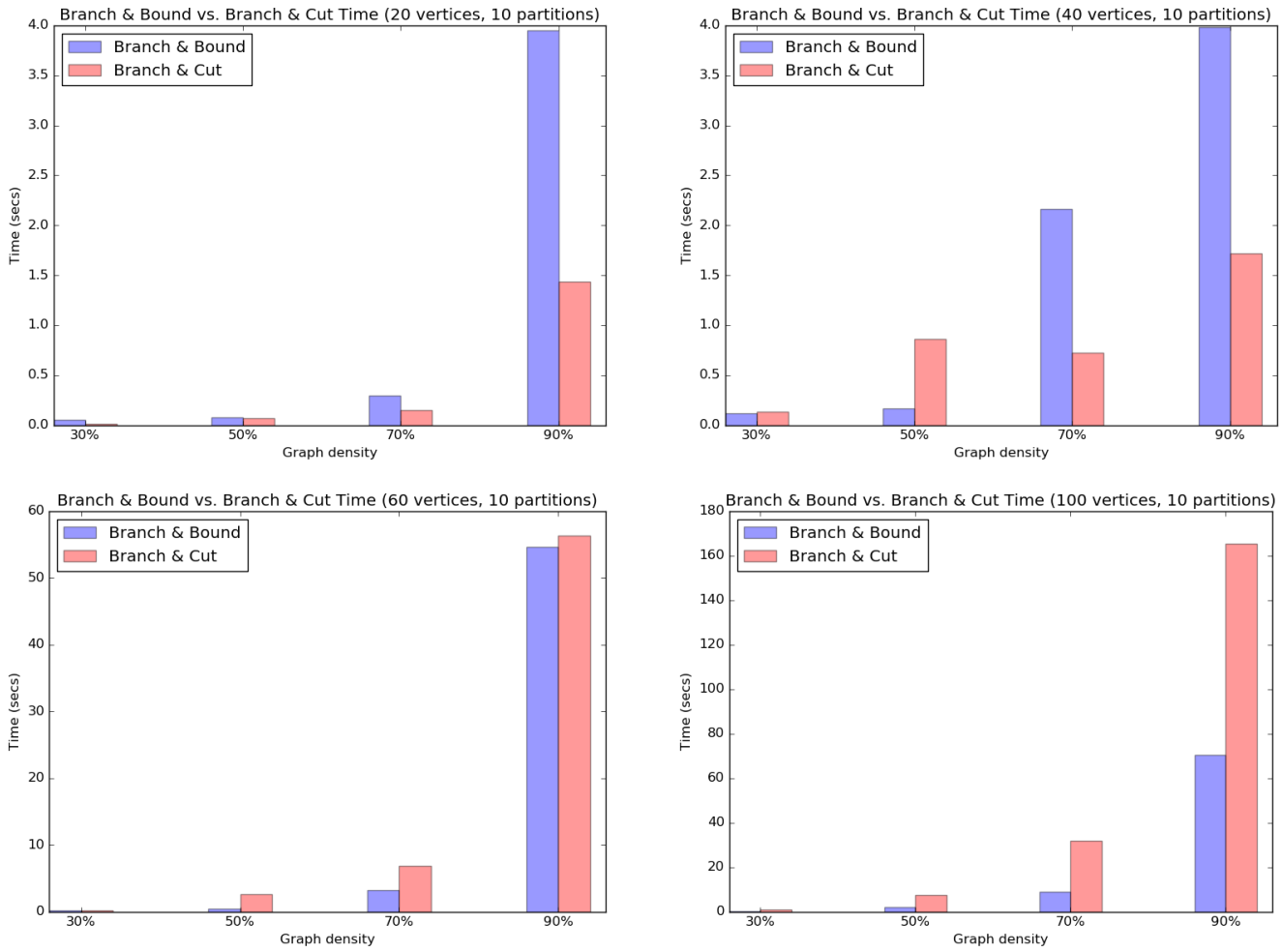



Figura 6: Efecto de aumentar la densidad del grafo.

A medida que aumenta la densidad del grafo, el problema de coloreo se vuelve sin duda más difícil. En los casos donde el número 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 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.

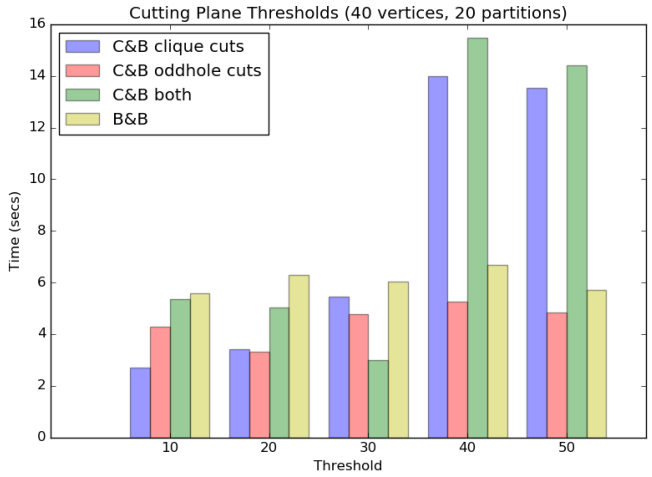



Figura 7: Tiempo de ejecución al incrementar el número de restricciones incorporadas.

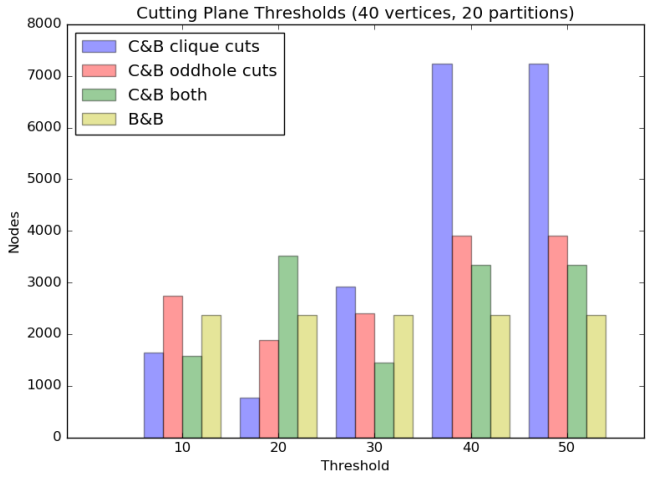



Figura 8: Nodos recorridos al incrementar el número de restricciones incorporadas.

Como podemos observar, agregar más 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

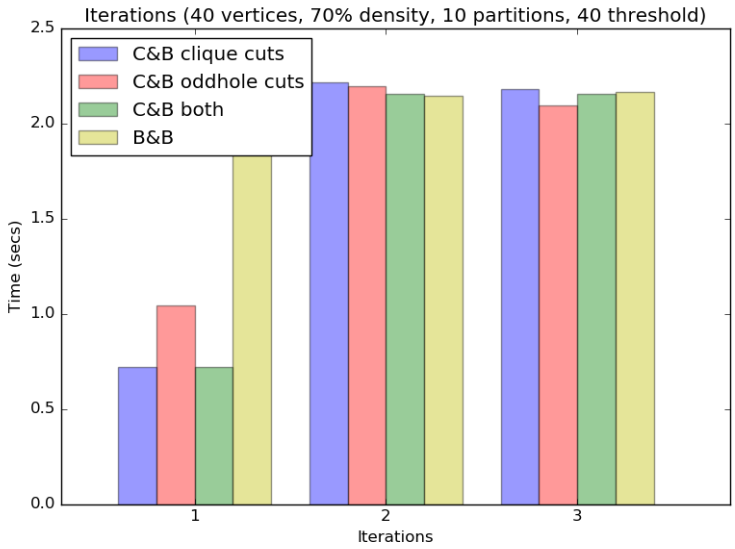



Figura 9: Tiempo de ejecución al aumentar la cantidad de iteraciones de planos de corte.

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

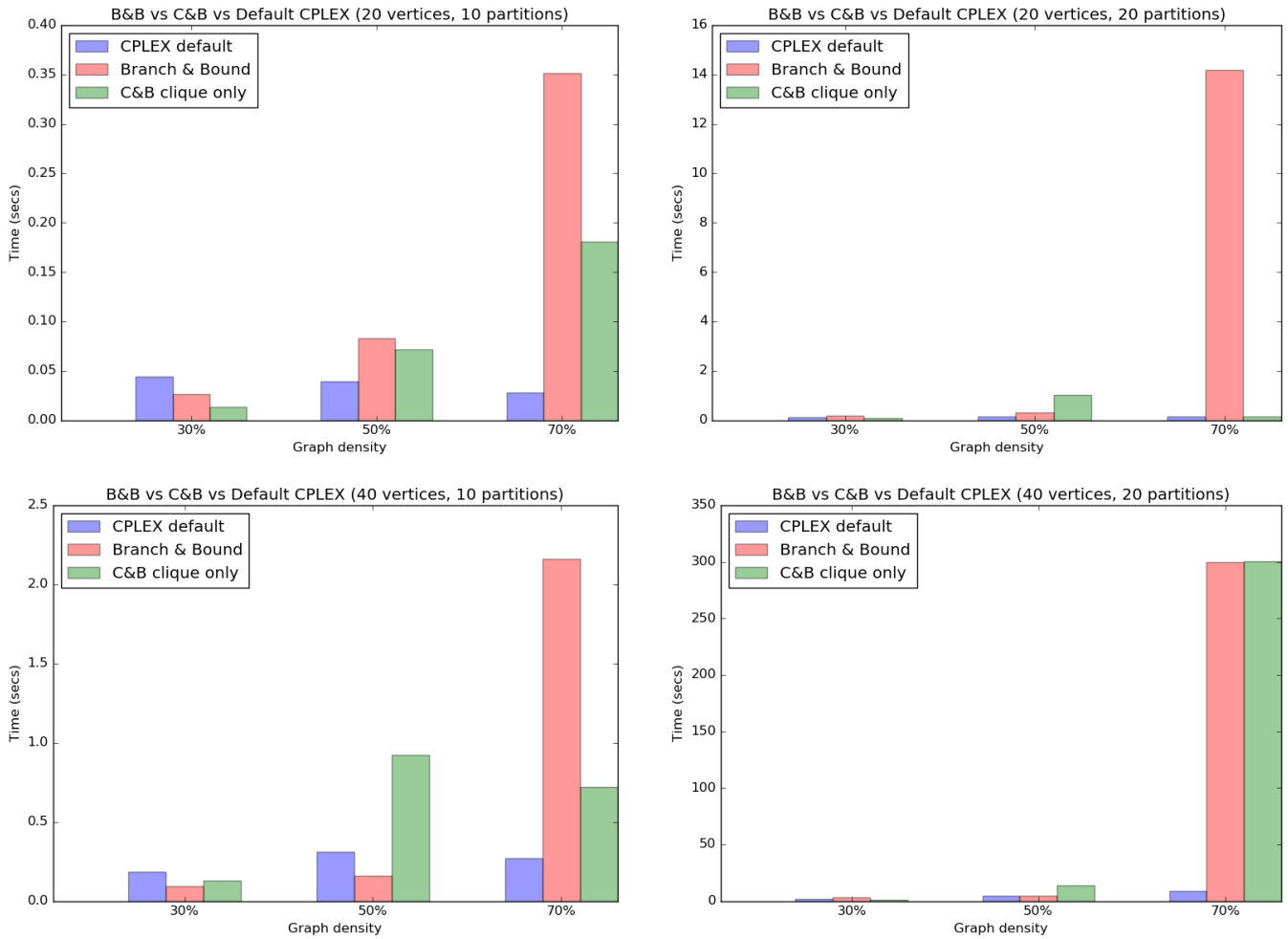



Figura 10: Comparacion B&B, C&B, CPLEX default para diferentes grafos.

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.

## 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 solo analizaremos DFS y BBS. DFS (Depth First Search) recorre el árbol de enumeración de B&B primero en profundidad. Por otro lado, BBS (Best Bound Search) recorre el árbol de enumeración utilizando alguna estrategia para intentar buscar una buena cota lo mas rápido posible. 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 cual es la mejor variable sobre la cual hacer branching. Hay muchas reglas, como por ejemplo *max/min infeasibility*. Mientras que la regla de *minimum infeasibility* busca hacer branching sobre mas cercana al entero, la regla de *maximum infeasibility* busca hacer exactamente lo contrario [3].

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, BESTBOUND + MININFEAS.

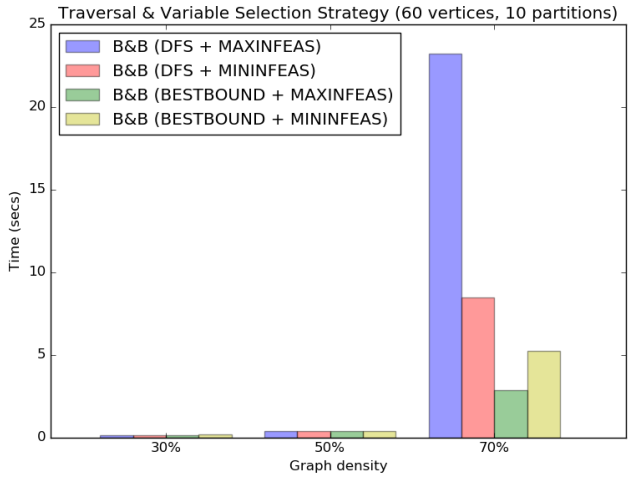



Figura 11: Tiempo de ejecución dependiendo de la estrategias de recorrido y selección de variable.

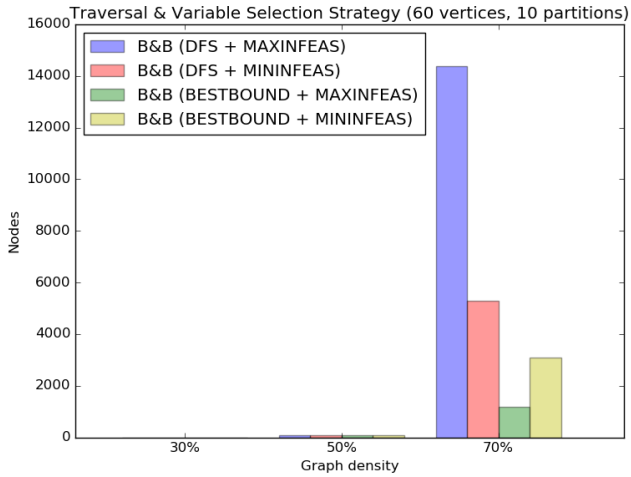
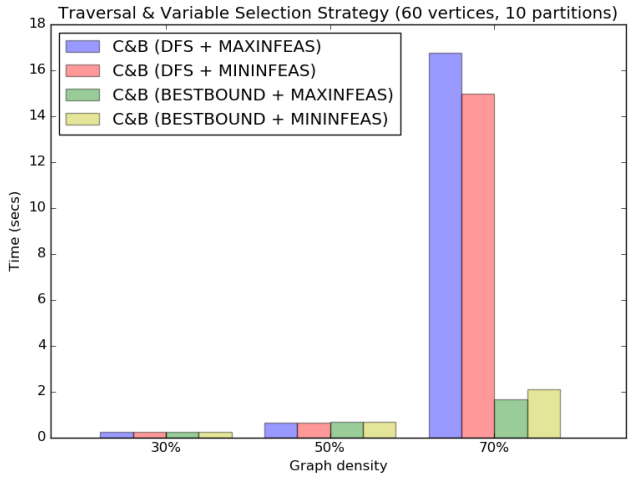
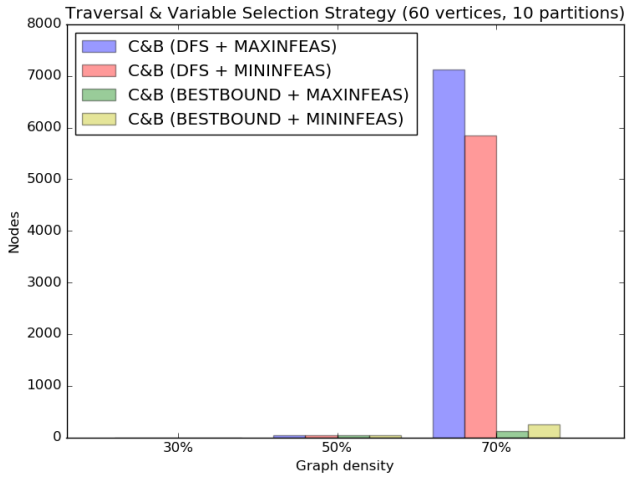



Figura 12: Nodos recorridos dependiendo de la estrategias de recorrido y selección de variable.




Como podemos observar, en general C&B tiene tiempos de ejecución menores y a su vez recorre menos nodos. La mejor estrategia para este problema parece ser BESTBOUND + MAXINFEAS. CPLEX utiliza por default BESTBOUND, aunque utiliza una heurística para elegir la variable de branching.

[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

## 4.9. Instancias DIMACS

Las instancias DIMACS son comúnmente utilizadas en la literatura como instancias de benchmarking. A continuación mostramos nuestros tiempos de ejecución con B&B y B&C utilizando 1 iteración y $threshold = 30$ con solo desigualdades de clique. A su vez, ambas utilizan el recorrido del árbol de enumeración dado por default en CPLEX.

Cada tabla utiliza un numero de particiones diferente. Dado que las instancias DIMACS no tienen un numero de partición debido a que se utilizan normalmente para coloreo, asignamos uno nosotros y luego dividimos los vértices en orden en las diferentes particiones de forma uniforme.

Tomamos como tiempo de ejecucion limite 10 minutos, y reportamos el numero de colores encontrado por B&B hasta ese momento. En todos los casos, B&B y B&C coincidieron con el numero de colores utilizados, por lo que los reportamos solo en una columna.

Cuadro 1: Benchmark con 10 particiones

| Problem | n | m | Tiempo tomado por B&B (secs) | Tiempo tomado por 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 | 10 min | 10 min | 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 |
| myciel2 | 32766 | 0 | 4.37 | 110.76 | 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 |

# 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 iba a ser al menos tan difícil como el problema de coloreo, que ya en si es un problema sumamente complicado.

Las desigualdades de planos de cortes que se han implementado en este trabajo son desigualdades utilizadas normalmente para coloreo. Por esta razón, notamos a lo largo de todo el trabajo que en general los algoritmos de Cut & Branch funcionan bien a medida que aumenta el número de particiones. La intuición nos sugiere que deben existir mejores familias que exploten el hecho de que el grafo esté dividido en particiones, aunque encontrarlas escape del alcance de este trabajo [2].

Uno de los primeros problemas que encontramos al programar el algoritmo de Cut & Branch fue tener buenas heurísticas para las familias de desigualdades que probamos válidas. Aquí es donde interviene sin duda la creatividad del investigador para diseñar estas heurísticas, principalmente porque se sabe que generarlas de manera exhaustiva es un problema NP-Hard. 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 general notamos que el tiempo de cómputo no está dominado por la generación de estas familias, sino por la resolución del programa lineal. Probablemente esto no siempre sea cierto, y el investigador deba procurar un balance entre el tiempo de ejecución de la heurística y el tiempo de ejecución necesario para resolver el programa lineal. Por ejemplo, si consideramos el caso extremo donde generamos la familia entera, este problema es no polinomial, y seguramente dominará el tiempo de ejecución.

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 muy 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 obtenido. 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 mucha dificultad, y mejora los tiempos de ejecución de forma considerable.

Una posible mejora para resolver este problema sería implementar un algoritmo Branch & Cut, buscando buenas heurísticas iniciales y primales para el cálculo de cotas del óptimo, y haciendo cortes en cada nodo del árbol utilizando los *callbacks* de CPLEX [1]. 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

// 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);
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);
```

```
45
46 int checkStatus(CPXENVptr& env, int status);
47
48 // colors array!
49 const char* colors[] = {"Blue", "Red", "Green", "Yellow", "Grey", "Green", "Pink", "
       AliceBlue","AntiqueWhite","Aqua","Aquamarine","Azure","Beige",
50 "Bisque","Black","BlanchedAlmond","BlueViolet","Brown","BurlyWood","CadetBlue","
       Chartreuse","Chocolate","Coral","CornflowerBlue",
51 "Cornsilk","Crimson","Cyan","DarkBlue","DarkCyan","DarkGoldenRod","DarkGray","
       DarkGrey","DarkGreen","DarkKhaki","DarkMagenta","DarkOliveGreen",
52 "Darkorange","DarkOrchid","DarkRed","DarkSalmon","DarkSeaGreen","DarkSlateBlue","
       DarkSlateGray","DarkSlateGrey","DarkTurquoise",
53 "DarkViolet","DeepPink","DeepSkyBlue","DimGray","DimGrey","DodgerBlue","FireBrick","
       FloralWhite","ForestGreen","Fuchsia",
54 "Gainsboro","GhostWhite","Gold","GoldenRod","Gray","GreenYellow","HoneyDew","HotPink"
       ,"IndianRed","Indigo",
55 "Ivory","Khaki","Lavender","LavenderBlush","LawnGreen","LemonChiffon","LightBlue","
       LightCoral","LightCyan","LightGoldenRodYellow",
56 "LightGray","LightGrey","LightGreen","LightPink","LightSalmon","LightSeaGreen","
       LightSkyBlue","LightSlateGray","LightSlateGrey",
57 "LightSteelBlue","LightYellow","Lime","LimeGreen","Linen","Magenta","Maroon","
       MediumAquaMarine","MediumBlue","MediumOrchid",
58 "MediumPurple","MediumSeaGreen","MediumSlateBlue","MediumSpringGreen","
       MediumTurquoise","MediumVioletRed","MidnightBlue",
59 "MintCream","MistyRose","Moccasin","NavajoWhite","Navy","OldLace","Olive","OliveDrab"
       ,"Orange","OrangeRed","Orchid",
60 "PaleGoldenRod","PaleGreen","PaleTurquoise","PaleVioletRed","PapayaWhip","PeachPuff",
       "Peru","Plum","PowderBlue",
61 "Purple","RosyBrown","RoyalBlue","SaddleBrown","Salmon","SandyBrown","SeaGreen","
       SeaShell","Sienna","Silver","SkyBlue",
62 "SlateBlue","SlateGray","SlateGrey","Snow","SpringGreen","SteelBlue","Tan","Teal","
       Thistle","Tomato","Turquoise","Violet",
63 "Wheat","White","WhiteSmoke","YellowGreen"};
64
65 int main(int argc, char **argv) {
66
67     if (argc != 11) {
68         printf("Usage: %s inputFile solver partitions symmetry_breaker iterations
               select_cuts load_limit custom_config traversal_strategy branching_strategy
               \n", argv[0]);
69         exit(1);
70     }
71
72     int solver = atoi(argv[2]);
73     int partition_size = atoi(argv[3]);
74     bool symmetry_breaker = (atoi(argv[4]) == 1);
75     int iterations = atoi(argv[5]);
76     int select_cuts = atoi(argv[6]);              // 0: clique only, 1: oddhole only,
           2: both
77     int load_limit = atoi(argv[7]);
78     int custom_config = atoi(argv[8]);           // 0: default, 1: custom
79     int traversal_strategy = atoi(argv[9]);
80     int branching_strategy = atoi(argv[10]);
81
82     if (solver == 1) {
83         printf("Solver: Branch & Bound\n");
84     } else {
85         printf("Solver: Cut & Branch\n");
86     }
87
88     /* read graph input file
89      * format: http://mat.gsia.cmu.edu/COLOR/instances.html
```

```
90        * graph representation chosen in order to load the LP easily.
91        * - vector of edges
92        * - vector of partitions
93        */
94       FILE* fp = fopen(argv[1], "r");
95
96       if (fp == NULL) {
97           printf("Invalid input file.\n");
98           exit(1);
99       }
100
101      char buf[100];
102      int vertex_size, edge_size;
103
104      set<pair<double,int> > edges; // sometimes we have to filter directed graphs
105
106      while (fgets(buf, sizeof(buf), fp) != NULL) {
107          if (buf[0] == 'c') continue;
108          else if (buf[0] == 'p') {
109              sscanf(&buf[7], "%d %d", &vertex_size, &edge_size);
110          }
111          else if (buf[0] == 'e') {
112              int from, to;
113              sscanf(&buf[2], "%d %d", &from, &to);
114              if (from < to) {
115                  edges.insert(pair<double,int>(from, to));
116              } else {
117                  edges.insert(pair<double,int>(to, from));
118              }
119          }
120      }
121
122      // build adyacency list
123      edge_size = edges.size();
124      int adyacency_size = vertex_size*vertex_size - ((vertex_size+1)*vertex_size/2);
125      bool* adjacencyList = new bool[adyacency_size]; // can be optimized even more
             with a bitfield.
126      fill_n(adjacencyList, adyacency_size, false);
127      for (set<pair<double,int> >::iterator it = edges.begin(); it != edges.end(); ++it
             ) {
128          adjacencyList[fromMatrixToVector(it->first, it->second, vertex_size)] = true;
129      }
130
131      // set random seed
132      // srand(time(NULL));
133
134      // asign every vertex to a partition
135      // int partition_size = rand() % vertex_size + 1;
136      vector<vector<int> > partitions(partition_size, vector<int>());
137
138      for (int i = 0; i < vertex_size; ++i) {
139          partitions[i % partition_size].push_back(i+1);
140      }
141
142      // warning: this procedure doesn't guarantee every partition will have an element
             .
143      // for (int i = 1; i <= vertex_size; ++i) {
144      //  int assign_partition = rand() % partition_size;
145      //  partitions[assign_partition].push_back(i);
146      // }
147
148      // // update partition_size
```

```
149     // for (std::vector<vector<int> >::iterator it = partitions.begin(); it !=
            partitions.end(); ++it) {
150     //  if (it->size() == 0) --partition_size;
151     // }
152
153     printf("Graph: vertex_size: %d, edge_size: %d, partition_size: %d\n", vertex_size
            , edge_size, partition_size);
154
155     // start loading LP using CPLEX
156     int status;
157     CPXENVptr env; // pointer to enviroment
158     CPXLPptr lp;   // pointer to the lp.
159
160     env = CPXopenCPLEX(&status); // create enviroment
161     checkStatus(env, status);
162
163     // create LP
164     lp = CPXcreateprob(env, &status, "Instance of partitioned graph coloring.");
165     checkStatus(env, status);
166
167     setCPLEXConfig(env);
168     if (custom_config == 1) setBranchAndBoundConfig(env);
169     setTraversalStrategy(env, traversal_strategy);
170     setBranchingVariableStrategy(env, branching_strategy);
171
172     if (solver == 1) { // pure branch & bound
173         loadObjectiveFunction(env, lp, vertex_size, partition_size, CPX_BINARY);
174     } else {
175         loadObjectiveFunction(env, lp, vertex_size, partition_size, CPX_CONTINUOUS);
176     }
177
178     loadAdyacencyColorRestriction(env, lp, vertex_size, edge_size, partition_size,
            adjacencyList);
179     loadSingleColorInPartitionRestriction(env, lp, partitions, partition_size);
180     loadAdyacencyColorRestriction(env, lp, vertex_size, partition_size);
181
182     if (symmetry_breaker) loadSymmetryBreaker(env, lp, partition_size);
183
184     if (solver != 1) loadCuttingPlanes(env, lp, vertex_size, edge_size,
            partition_size, adjacencyList, iterations, load_limit, select_cuts);
185
186     // write LP formulation to file, great to debug.
187     status = CPXwriteprob(env, lp, "graph.lp", NULL);
188     checkStatus(env, status);
189
190     convertVariableType(env, lp, vertex_size, partition_size, CPX_BINARY);
191
192     solveLP(env, lp, edge_size, vertex_size, partition_size);
193
194     delete[] adjacencyList;
195
196     return 0;
197 }
198
199 int getVertexIndex(int id, int color, int partition_size) {
200     return partition_size + ((id-1)*partition_size) + (color-1);
201 }
202
203 /* since the adyacency matrix is symmetric and the diagonal is not needed, we can
        simply
204  * store the upper diagonal and get adyacency from a list. the math is quite simple,
        it
```

```
205  * just uses the formula for the sum of integers. ids are numbered starting from 1.
206  */
207 inline int fromMatrixToVector(int from, int to, int vertex_size) {
208
209     // for speed, many parts of this code are commented, since by our usage we always
210     // know from < to and are in range.
211
212     // assert(from != to && from <= vertex_size && to <= vertex_size);
213
214     // if (from < to)
215         return from*vertex_size - (from+1)*from/2 - (vertex_size - to) - 1;
216     // else
217     //  return to*vertex_size - (to+1)*to/2 - (vertex_size - from) - 1;
218 }
219
220 inline bool isAdyacent(int from, int to, int vertex_size, bool* adjacencyList) {
221     return adjacencyList[fromMatrixToVector(from, to, vertex_size)];
222 }
223
224 bool adyacentToAll(int id, int vertex_size, bool* adjacencyList, const set<int>&
        clique) {
225     for (set<int>::iterator it = clique.begin(); it != clique.end(); ++it) {
226         if (!isAdyacent(*it, id, vertex_size, adjacencyList)) return false;
227     }
228     return true;
229 }
230
231 bool cliqueNotContained(const set<int>& clique, int color, const vector<tuple<double,
        int, set<int> > >& clique_familly) {
232     for (vector<tuple<double, int, set<int> > >::const_iterator it = clique_familly.
            begin(); it != clique_familly.end(); ++it) {
233         // by construction, sets are already ordered.
234         if (get<1>(*it) == color && includes(get<2>(*it).begin(), get<2>(*it).end(),
                clique.begin(), clique.end())) return false;
235     }
236     return true;
237 }
238
239 int loadObjectiveFunction(CPXENVptr& env, CPXLPptr& lp, int vertex_size, int
        partition_size, char vtype) {
240
241     // load objective function
242     int n = partition_size + (vertex_size*partition_size);
243     double *objfun  = new double[n];
244     double *ub      = new double[n];
245     char     *ctype = new char[n];
246     char **colnames = new char*[n];
247
248     for (int i = 0; i < partition_size; ++i) {
249         objfun[i] = 1;
250         ub[i] = 1;
251         ctype[i]    = vtype;
252         colnames[i] = new char[10];
253         sprintf(colnames[i], "w_%d", (i+1));
254     }
255
256     for (int id = 1; id <= vertex_size; ++id) {
257         for (int color = 1; color <= partition_size; ++color) {
258             int index = getVertexIndex(id, color, partition_size);
259             objfun[index]   = 0;
260             ub[index] = 1;
261             ctype[index]    = vtype;
```

```
262             colnames[index] = new char[10];
263             sprintf(colnames[index], "x%d_%d", id, color);
264         }
265     }
266
267     // CPLEX bug? If you set ctype, it doesn't identify the problem as continous.
268     int status = CPXnewcols(env, lp, n, objfun, NULL, ub, NULL, colnames);
269     checkStatus(env, status);
270
271     // free memory
272     for (int i = 0; i < n; ++i) {
273         delete[] colnames[i];
274     }
275
276     delete[] objfun;
277     delete[] ub;
278     delete[] ctype;
279     delete[] colnames;
280
281     return 0;
282 }
283
284 int loadAdyacencyColorRestriction(CPXENVptr& env, CPXLPptr& lp, int vertex_size, int
    edge_size, int partition_size, bool* adjacencyList) {
285
286     // load first restriction
287     int ccnt = 0;                           // new columns being added.
288     int rcnt = edge_size * partition_size; // new rows being added.
289     int nzcnt = rcnt*2;                     // nonzero constraint coefficients being
        added.
290
291     double *rhs = new double[rcnt];         // independent term in restrictions.
292     char *sense = new char[rcnt];           // sense of restriction inequality.
293
294     int *matbeg = new int[rcnt];            // array position where each restriction
        starts in matind and matval.
295     int *matind = new int[rcnt*2];          // index of variables != 0 in restriction
        (each var has an index defined above)
296     double *matval  = new double[rcnt*2]; // value corresponding to index in
        restriction.
297     char **rownames = new char*[rcnt];      // row labels.
298
299     int i = 0;
300     for (int from = 1; from <= vertex_size; ++from) {
301         for (int to = from + 1; to <= vertex_size; ++to) {
302
303             if (!isAdyacent(from, to, vertex_size, adjacencyList)) continue;
304
305             for (int color = 1; color <= partition_size; ++color) {
306                 matbeg[i] = i*2;
307
308                 matind[i*2]   = getVertexIndex(from, color, partition_size);
309                 matind[i*2+1] = getVertexIndex(to  , color, partition_size);
310
311                 matval[i*2]   = 1;
312                 matval[i*2+1] = 1;
313
314                 rhs[i] = 1;
315                 sense[i] = 'L';
316                 rownames[i] = new char[40];
317                 sprintf(rownames[i], "%s", colors[color-1]);
318
```

```
319                 ++i;
320             }
321         }
322     }
323
324     // debug flag
325     // status = CPXsetintparam(env, CPX_PARAM_DATACHECK, CPX_ON);
326
327     // add restriction
328     int status = CPXaddrows(env, lp, ccnt, rcnt, nzcnt, rhs, sense, matbeg, matind,
        matval, NULL, rownames);
329     checkStatus(env, status);
330
331     // free memory
332     for (int i = 0; i < rcnt; ++i) {
333         delete[] rownames[i];
334     }
335
336     delete[] rhs;
337     delete[] sense;
338     delete[] matbeg;
339     delete[] matind;
340     delete[] matval;
341     delete[] rownames;
342
343     return 0;
344 }
345
346
347 int loadSingleColorInPartitionRestriction(CPXENVptr& env, CPXLPptr& lp, vector<vector
    <int>>& partitions, int partition_size) {
348
349     // load second restriction
350     int p = 1;
351     for (std::vector<vector<int> >::iterator it = partitions.begin(); it !=
        partitions.end(); ++it) {
352
353         int size = it->size();                  // current partition size.
354         if (size == 0) continue;                // skip empty partitions.
355
356         int ccnt = 0;                           // new columns being added.
357         int rcnt = 1;                           // new rows being added.
358         int nzcnt = size*partition_size;        // nonzero constraint coefficients
            being added.
359
360         double *rhs = new double[rcnt];         // independent term in restrictions.
361         char *sense = new char[rcnt];           // sense of restriction inequality.
362
363         int *matbeg = new int[rcnt];            // array position where each
            restriction starts in matind and matval.
364         int *matind = new int[nzcnt];           // index of variables != 0 in
            restriction (each var has an index defined above)
365         double *matval  = new double[nzcnt];    // value corresponding to index in
            restriction.
366         char **rownames = new char*[rcnt];      // row labels.
367
368         matbeg[0] = 0;
369         sense[0]  = 'E';
370         rhs[0]    = 1;
371         rownames[0] = new char[40];
372         sprintf(rownames[0], "partition_%d", p);
373
```

```
374         int i = 0;
375         for (std::vector<int>::iterator it2 = it->begin(); it2 != it->end(); ++it2) {
376             for (int color = 1; color <= partition_size; ++color) {
377                 matind[i] = getVertexIndex(*it2, color, partition_size);
378                 matval[i] = 1;
379                 ++i;
380             }
381         }
382 
383         // add restriction
384         int status = CPXaddrows(env, lp, ccnt, rcnt, nzcnt, rhs, sense, matbeg,
                matind, matval, NULL, rownames);
385         checkStatus(env, status);
386 
387         // free memory
388         delete[] rownames[0];
389         delete[] rhs;
390         delete[] sense;
391         delete[] matbeg;
392         delete[] matind;
393         delete[] matval;
394         delete[] rownames;
395 
396         ++p;
397     }
398 
399     return 0;
400 }
401 
402 int loadSymmetryBreaker(CPXENVptr& env, CPXLPptr& lp, int partition_size) {
403 
404     int ccnt = 0;                           // new columns being added.
405     int rcnt = partition_size - 1;          // new rows being added.
406     int nzcnt = 2*rcnt;                     // nonzero constraint coefficients being
            added.
407 
408     double* rhs = new double[rcnt];         // independent term in restrictions.
409     char *sense = new char[rcnt];           // sense of restriction inequality.
410 
411     int *matbeg = new int[rcnt];            // array position where each restriction
            starts in matind and matval.
412     int *matind = new int[rcnt*2];          // index of variables != 0 in restriction
            (each var has an index defined above)
413     double *matval  = new double[rcnt*2];   // value corresponding to index in
            restriction.
414     char **rownames = new char*[rcnt];      // row labels.
415 
416     int i = 0;
417     for (int color = 0; color < partition_size - 1; ++color) {
418         matbeg[i] = i*2;
419         matind[i*2]   = color;
420         matind[i*2+1] = color + 1;
421         matval[i*2]   = -1;
422         matval[i*2+1] = 1;
423 
424         rhs[i] = 0;
425         sense[i] = 'L';
426         rownames[i] = new char[40];
427         sprintf(rownames[i], "%s", "symmetry_breaker");
428 
429         ++i;
430     }
```

```
431
432
433     // add restriction
434     int status = CPXaddrows(env, lp, ccnt, rcnt, nzcnt, rhs, sense, matbeg, matind,
            matval, NULL, rownames);
435     checkStatus(env, status);
436
437     // free memory
438     for (int i = 0; i < rcnt; ++i) {
439         delete[] rownames[i];
440     }
441
442     delete[] rhs;
443     delete[] sense;
444     delete[] matbeg;
445     delete[] matind;
446     delete[] matval;
447     delete[] rownames;
448
449     return 0;
450 }
451
452 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) {
453
454     printf("Finding Cutting Planes.\n");
455
456     // calculate runtime
457     double inittime, endtime;
458     int status = CPXgettime(env, &inittime);
459
460     int n = partition_size + (vertex_size*partition_size);
461
462     double *sol = new double[n];
463     int i = 1;
464     int unsatisfied_restrictions = 0;
465     while (i <= iterations) {
466
467         printf("Iteration %d\n", i);
468
469         // solve LP
470         status = CPXlpopt(env, lp);
471         checkStatus(env, status);
472
473         status = CPXgetx(env, lp, sol, 0, n - 1);
474         checkStatus(env, status);
475
476         // print relaxation result
477         // for (int id = 1; id <= vertex_size; ++id) {
478         //  for (int color = 1; color <= partition_size; ++color) {
479         //      int index = getVertexIndex(id, color, partition_size);
480         //      if (sol[index] == 0) continue;
481         //      cout << "x" << id << "_" << color << " = " << sol[index] << endl;
482         //  }
483         // }
484
485         if (select_cuts == 0 || select_cuts == 2) unsatisfied_restrictions +=
            maximalCliqueFamillyHeuristic(env, lp, vertex_size, edge_size,
            partition_size, adjacencyList, sol, load_limit);
486         if (select_cuts == 1 || select_cuts == 2) unsatisfied_restrictions +=
            oddholeFamillyHeuristic(env, lp, vertex_size, edge_size, partition_size,
```

```
            adjacencyList, sol, load_limit);
487
488         if (unsatisfied_restrictions == 0) break;
489
490         unsatisfied_restrictions = 0;
491         ++i;
492     }
493
494     status = CPXgettime(env, &endtime);
495     double elapsed_time = endtime-inittime;
496     cout << "Time taken to add cutting planes: " << elapsed_time << endl;
497
498     return 0;
499 }
500
501 int maximalCliqueFamillyHeuristic(CPXENVptr& env, CPXLPptr& lp, int vertex_size, int
    edge_size, int partition_size, bool* adjacencyList, double* sol, int load_limit) {
502
503     printf("Generating clique familly.\n");
504
505     int loaded = 0;
506
507     vector<tuple<double, int, set<int> > > clique_familly;
508
509     for (int color = 1; color <= partition_size; ++color) {
510
511         for (int id = 1; id <= vertex_size; id++) {
512
513             if (sol[getVertexIndex(id, color, partition_size)] == 0) continue;
514
515             double sum = sol[getVertexIndex(id, color, partition_size)];
516             set<int> clique;
517             clique.insert(id);
518             for (int id2 = id + 1; id2 <= vertex_size; ++id2) {
519                 if (sol[getVertexIndex(id2, color, partition_size)] == 0) continue;
520
521                 if (adyacentToAll(id2, vertex_size, adjacencyList, clique)) {
522                     clique.insert(id2);
523                     sum += sol[getVertexIndex(id2, color, partition_size)];
524                 }
525             }
526             if (clique.size() > 2 && sum > sol[color-1] + TOL) {
527                 if (cliqueNotContained(clique, color, clique_familly)) {
528                     double score = sum - sol[color-1];
529                     clique_familly.push_back(tuple<double, int, set<int> >(score,
                        color,clique));
530                 }
531             }
532         }
533     }
534
535     sort(clique_familly.begin(), clique_familly.end(), greater<tuple<double, int, set
        <int> > >());
536
537     //print the familly
538     for (vector<tuple<double, int, set<int> > >::const_iterator it = clique_familly.
        begin();
539          it != clique_familly.end() && loaded < load_limit; ++loaded, ++it) {
540
541         loadUnsatisfiedCliqueRestriction(env, lp, partition_size, get<2>(*it), get
            <1>(*it));
542         cout << "Score: " << get<0>(*it) << " - ";
```

```
543         for (set<int>::iterator it2 = get<2>(*it).begin(); it2 != get<2>(*it).end();
                ++it2) {
544             cout << *it2 << " ";
545         }
546         cout << endl;
547     }
548 
549     printf("Loaded %d/%d unsatisfied clique restrictions! (all colors)\n", loaded, (
            int) clique_familly.size());
550 
551     return loaded;
552 }
553 
554 int loadUnsatisfiedCliqueRestriction(CPXENVptr& env, CPXLPptr& lp, int partition_size
        , const set<int>& clique, int color) {
555 
556     int ccnt = 0;
557     int rcnt = 1;
558     int nzcnt = clique.size() + 1;
559 
560     double rhs = 0;
561     char sense = 'L';
562 
563     int matbeg = 0;
564     int* matind    = new int[clique.size() + 1];
565     double* matval = new double[clique.size() +1];
566     char **rowname = new char*[rcnt];
567     rowname[0] = new char[40];
568     sprintf(rowname[0], "unsatisfied_clique");
569 
570     matind[0] = color - 1;
571     matval[0] = -1;
572 
573     int i = 1;
574     for (set<int>::iterator it = clique.begin(); it != clique.end(); ++it) {
575         matind[i] = getVertexIndex(*it, color, partition_size);
576         matval[i] = 1;
577         ++i;
578     }
579 
580     // add restriction
581     int status = CPXaddrows(env, lp, ccnt, rcnt, nzcnt, &rhs, &sense, &matbeg, matind
            , matval, NULL, rowname);
582     checkStatus(env, status);
583 
584     // free memory
585     delete[] matind;
586     delete[] matval;
587     delete rowname[0];
588     delete rowname;
589 
590     return 0;
591 }
592 
593 int oddholeFamillyHeuristic(CPXENVptr& env, CPXLPptr& lp, int vertex_size, int
        edge_size, int partition_size, bool* adjacencyList, double* sol, int load_limit) {
594 
595     printf("Generating oddhole familly.\n");
596 
597     int loaded = 0;
598 
599     vector<tuple<double, int, set<int>>> path_familly; // dif, color, path
```

```
600
601     for (int color = 1; color <= partition_size; ++color) {
602
603         for (int id = 1; id <= vertex_size; id++) {
604
605             if (sol[getVertexIndex(id, color, partition_size)] == 0) continue;
606
607             double sum = 0;
608             set<int> path;
609             path.insert(id);
610             for (int id2 = id + 1; id2 <= vertex_size; ++id2) {
611                 if (sol[getVertexIndex(id2, color, partition_size)] == 0) continue;
612
613                 if (isAdyacent(*(--path.end()), id2, vertex_size, adjacencyList)) {
614                     path.insert(id2);
615                 }
616
617             }
618
619             while (path.size() >= 3 && (path.size() %2 == 0 ||
620                 !isAdyacent(*path.begin(), *(--path.end()), vertex_size,
                        adjacencyList))) {
621                 path.erase(--path.end());
622             }
623
624             for (set<int>::iterator it = path.begin(); it != path.end(); ++it) {
625                 sum += sol[getVertexIndex(*it, color, partition_size)];
626             }
627
628             int k = (path.size() - 1) / 2;
629             if (path.size() > 2 && sum > k*sol[color-1] + TOL) {
630                 double score = sum - k*sol[color-1];
631                 path_familly.push_back(tuple<double, int, set<int> >(score,color,path
                        ));
632             }
633         }
634     }
635
636     sort(path_familly.begin(), path_familly.end(), greater<tuple<double, int, set<int
        > > >());
637
638     //print the familly
639     for (vector<tuple<double, int, set<int> > >::const_iterator it = path_familly.
        begin();
640          it != path_familly.end() && loaded < load_limit; ++loaded, ++it) {
641          loadUnsatisfiedOddholeRestriction(env, lp, partition_size, get<2>(*it), get
             <1>(*it));
642          cout << "Score: " << get<0>(*it) << " - ";
643          for (set<int>::iterator it2 = get<2>(*it).begin(); it2 != get<2>(*it).end();
             ++it2) {
644              cout << *it2 << " ";
645          }
646          cout << endl;
647     }
648
649     printf("Loaded %d/%d unsatisfied oddhole restrictions! (all colors)\n", loaded, (
        int) path_familly.size());
650
651     return loaded;
652 }
653
```

```
654 int loadUnsatisfiedOddholeRestriction(CPXENVptr& env, CPXLPptr& lp, int
        partition_size, const set<int>& path, int color) {
655
656     int ccnt = 0;
657     int rcnt = 1;
658     int nzcnt = path.size() + 1;
659
660     double rhs = 0;
661     char sense = 'L';
662
663     int matbeg = 0;
664     int* matind    = new int[path.size() + 1];
665     double* matval = new double[path.size() +1];
666     char **rowname = new char*[rcnt];
667     rowname[0] = new char[40];
668     sprintf(rowname[0], "unsatisfied_oddhole");
669
670     int k = (path.size() - 1) / 2;
671
672     matind[0] = color - 1;
673     matval[0] = -k;
674
675     int i = 1;
676     for (set<int>::iterator it = path.begin(); it != path.end(); ++it) {
677         matind[i] = getVertexIndex(*it, color, partition_size);
678         matval[i] = 1;
679         ++i;
680     }
681
682     // add restriction
683     int status = CPXaddrows(env, lp, ccnt, rcnt, nzcnt, &rhs, &sense, &matbeg, matind
            , matval, NULL, rowname);
684     checkStatus(env, status);
685
686     // free memory
687     delete[] matind;
688     delete[] matval;
689     delete rowname[0];
690     delete rowname;
691
692     return 0;
693 }
694
695 int loadAdyacencyColorRestriction(CPXENVptr& env, CPXLPptr& lp, int vertex_size, int
        partition_size)  {
696
697     // load third restriction
698     int ccnt = 0;                              // new columns being added.
699     int rcnt = vertex_size * partition_size; // new rows being added.
700     int nzcnt = rcnt*2;                        // nonzero constraint coefficients being
            added.
701
702     double *rhs = new double[rcnt];            // independent term in restrictions.
703     char *sense = new char[rcnt];              // sense of restriction inequality.
704
705     int *matbeg = new int[rcnt];               // array position where each restriction
            starts in matind and matval.
706     int *matind = new int[rcnt*2];             // index of variables != 0 in
           restriction (each var has an index defined above)
707     double *matval  = new double[rcnt*2];      // value corresponding to index in
           restriction.
708     char **rownames = new char*[rcnt];         // row labels.
```

```
709
710     int i = 0;
711     for (int v = 1; v <= vertex_size; ++v) {
712         for (int color = 1; color <= partition_size; ++color) {
713             matbeg[i] = i*2;
714
715             matind[i*2]   = getVertexIndex(v, color, partition_size);
716             matind[i*2+1] = color-1;
717
718             matval[i*2]   = 1;
719             matval[i*2+1] = -1;
720
721             rhs[i] = 0;
722             sense[i] = 'L';
723             rownames[i] = new char[40];
724             sprintf(rownames[i], "color_res");
725
726             ++i;
727         }
728     }
729
730     // add restriction
731     int status = CPXaddrows(env, lp, ccnt, rcnt, nzcnt, rhs, sense, matbeg, matind,
            matval, NULL, rownames);
732     checkStatus(env, status);
733
734     // free memory
735     for (int i = 0; i < rcnt; ++i) {
736         delete[] rownames[i];
737     }
738
739     delete[] rhs;
740     delete[] sense;
741     delete[] matbeg;
742     delete[] matind;
743     delete[] matval;
744     delete[] rownames;
745
746     return 0;
747 }
748
749 int solveLP(CPXENVptr& env, CPXLPptr& lp, int edge_size, int vertex_size, int
        partition_size) {
750
751     printf("\nSolving MIP.\n");
752
753     int n = partition_size + (vertex_size*partition_size); // amount of total
            variables
754
755     // calculate runtime
756     double inittime, endtime;
757     int status = CPXgettime(env, &inittime);
758     checkStatus(env, status);
759
760     // solve LP
761     status = CPXmipopt(env, lp);
762     checkStatus(env, status);
763
764     status = CPXgettime(env, &endtime);
765     checkStatus(env, status);
766
767     // check solution state
```

```
768     int solstat;
769     char statstring[510];
770     CPXCHARptr p;
771     solstat = CPXgetstat(env, lp);
772     p = CPXgetstatstring(env, solstat, statstring);
773     string statstr(statstring);
774     if (solstat != CPXMIP_OPTIMAL && solstat != CPXMIP_OPTIMAL_TOL &&
775         solstat != CPXMIP_NODE_LIM_FEAS && solstat != CPXMIP_TIME_LIM_FEAS) {
776         // printf("Optimization failed.\n");
777         cout << "Optimization failed: " << solstat << endl;
778         exit(1);
779     }
780
781     double objval;
782     status = CPXgetobjval(env, lp, &objval);
783     checkStatus(env, status);
784
785     // get values of all solutions
786     double *sol = new double[n];
787     status = CPXgetx(env, lp, sol, 0, n - 1);
788     checkStatus(env, status);
789
790     int nodes_traversed = CPXgetnodecnt(env, lp);
791
792     // write solutions to current window
793     cout << "Optimization result: " << statstring << endl;
794     cout << "Time taken to solve final LP: " << (endtime - inittime) << endl;
795     cout << "Colors used: " << objval << endl;
796     cout << "Nodes traversed: " << nodes_traversed << endl;
797     for (int color = 1; color <= partition_size; ++color) {
798         if (sol[color-1] == 1) {
799             cout << "w_" << color << " = " << sol[color-1] << " (" << colors[color-1]
                    << ")" << endl;
800         }
801     }
802
803     for (int id = 1; id <= vertex_size; ++id) {
804         for (int color = 1; color <= partition_size; ++color) {
805             int index = getVertexIndex(id, color, partition_size);
806             if (sol[index] == 1) {
807                 cout << "x_" << id << " = " << colors[color-1] << endl;
808             }
809         }
810     }
811
812     delete[] sol;
813
814     return 0;
815 }
816
817 int convertVariableType(CPXENVptr& env, CPXLPptr& lp, int vertex_size, int
        partition_size, char vtype) {
818
819     int n = partition_size + (vertex_size*partition_size);
820     int* indices = new int[n];
821     char* xctype = new char[n];
822
823     for (int i = 0; i < n; i++) {
824         indices[i] = i;
825         xctype[i]  = vtype;
826     }
827     CPXchgctype(env, lp, n, indices, xctype);
```

```
828
829     delete[] indices;
830     delete[] xctype;
831
832     return 0;
833 }
834
835 int setTraversalStrategy(CPXENVptr& env, int strategy) {
836
837     // MIP node selection strategy
838     // http://www-01.ibm.com/support/knowledgecenter/SSSA5P_12.3.0/ilog.odms.cplex.
        help/Content/Optimization/Documentation/Optimization_Studio/_pubskel/
        ps_refparameterscplex2299.html
839
840     // 0 CPX_NODESEL_DFS           Depth-first search
841     // 1 CPX_NODESEL_BESTBOUND     Best-bound search; default
842     // 2 CPX_NODESEL_BESTEST       Best-estimate search
843     // 3 CPX_NODESEL_BESTEST_ALT   Alternative best-estimate search
844
845     CPXsetintparam(env, CPX_PARAM_NODESEL, strategy);
846
847     return 0;
848 }
849
850 int setBranchingVariableStrategy(CPXENVptr& env, int strategy) {
851
852     // MIP variable selection strategy
853     // http://www-01.ibm.com/support/knowledgecenter/SS9UKU_12.4.0/com.ibm.cplex.zos.
        help/Parameters/topics/VarSel.html
854
855     // -1   CPX_VARSEL_MININFEAS       Branch on variable with minimum infeasibility
856     // 0    CPX_VARSEL_DEFAULT         Automatic: let CPLEX choose variable to branch
          on; default
857     // 1    CPX_VARSEL_MAXINFEAS       Branch on variable with maximum infeasibility
858     // 2    CPX_VARSEL_PSEUDO          Branch based on pseudo costs
859     // 3    CPX_VARSEL_STRONG          Strong branching
860     // 4    CPX_VARSEL_PSEUDOREDUCED   Branch based on pseudo reduced costs
861
862     CPXsetintparam(env, CPX_PARAM_VARSEL, strategy);
863
864     return 0;
865 }
866
867 int setCPLEXConfig(CPXENVptr& env) {
868     // maximize objective function
869     // CPXchgobjsen(env, lp, CPX_MAX);
870
871     // enable/disable screen output
872     CPXsetintparam(env, CPX_PARAM_SCRIND, CPX_OFF);
873
874     // set excecution limit
875     CPXsetdblparam(env, CPX_PARAM_TILIM, 300);
876
877     // measure time in CPU time
878     // CPXsetintparam(env, CPX_PARAM_CLOCKTYPE, CPX_ON);
879
880     return 0;
881 }
882
883 int setBranchAndBoundConfig(CPXENVptr& env) {
884
885     // CPLEX config
```

```
886     // http://www-01.ibm.com/support/knowledgecenter/SSSA5P_12.2.0/ilog.odms.cplex.
            help/Content/Optimization/Documentation/CPLEX/_pubskel/CPLEX916.html
887
888     // deactivate pre-processing
889     CPXsetintparam(env, CPX_PARAM_PRESLVND, -1);
890     CPXsetintparam(env, CPX_PARAM_REPEATPRESOLVE, 0);
891     CPXsetintparam(env, CPX_PARAM_RELAXPREIND, 0);
892     CPXsetintparam(env, CPX_PARAM_REDUCE, 0);
893     CPXsetintparam(env, CPX_PARAM_LANDPCUTS, -1);
894
895     // disable presolve
896     // CPXsetintparam(env, CPX_PARAM_PREIND, CPX_OFF);
897
898     // enable traditional branch and bound
899     CPXsetintparam(env, CPX_PARAM_MIPSEARCH, CPX_MIPSEARCH_TRADITIONAL);
900
901     // use only one thread for experimentation
902     // CPXsetintparam(env, CPX_PARAM_THREADS, 1);
903
904     // do not add cutting planes
905     CPXsetintparam(env, CPX_PARAM_EACHCUTLIM, CPX_OFF);
906
907     // disable gomory fractional cuts
908     CPXsetintparam(env, CPX_PARAM_FRACCUTS, -1);
909
910     return 0;
911 }
912
913
914 int checkStatus(CPXENVptr& env, int status) {
915     if (status) {
916         char buffer[100];
917         CPXgeterrorstring(env, status, buffer);
918         printf("%s\n", buffer);
919         exit(1);
920     }
921     return 0;
922 }
```