Investigación Operativa

Segundo cuatrimestre 2015

Departamento de Computación
Facultad de Ciencias Exactas y Naturales
Universidad de Buenos Aires

Trabajo Práctico

Integrante	LU	Correo electrónico
Marco Vanotti	229/10	mvanotti@dc.uba.ar
Igal Frid	231/07	ipfrid@gmail.com

Abstract

En este trabajo, consideramos el problema de *coloreo particionado de grafos*: dado un grafo G(V, E) y una partición de los nodos $V_1, ..., V_p$, asignarle un color a un nodo de cada partición de forma tal que no haya dos nodos adyacentes pintados del mismo color y a su vez haya solo un nodo coloreado por partición. El objetivo es hallar un coloreo que use la menor cantidad de colores. Proponemos una solución al problema utilizando *programación lineal entera*, y analizamos la performance que tiene utilizar algoritmos del tipo *branch & bound* y *cut & branch*.

1 Introducción

Un coloreo (válido) de los nodos de un grafo $G = (V, E)$ es una asignación $f : V \to C$, tal que $f(u) \neq f(v), \forall (u, v) \in E$.

A su vez, en el problema de coloreo particionado de grafos, sea P el conjunto de particiones de V, cada partición de P debe tener exactamente un único nodo coloreado.

Dado un grafo G y un coloreo C, definimos al número cromático del coloreo como la cantidad de colores distintas que se utilizan dentro del mismo y lo notamos $\chi(C)$.

Como vimos a lo largo de la materia, hay muchos problemas que pueden modelarse utilizando un modelo de programación lineal entera. En este trabajo práctico el objetivo es, dado un grafo G particionado, obtener un coloreo válido para G tal que el número cromático sea mínimo.

Dado que no se conocen algoritmos polinomiales para resolver problemas de programación lineal entera, además de encontrar un modelo que sea correcto para el problema que queremos resolver, debemos poder encontrar un algoritmo que permita encontrar soluciones para instancias relativamente grandes. Para ello vamos a tener que hallar una serie de optimizaciones que sean específicas a nuestro problema e implmenentarlas en el algoritmo.

Para comparar los resultados, vamos a implementar dos tipos de algoritmos, uno **Branch and Bound** y luego uno **Cut and Branch**. Para implementar los planos de corte del algoritmo Cut and Branch vamos a utilizar dos familias de desigualdades: las desigualdades de **Clique Máxima** y las desigualdades de **Agujero Impar**.

Implementamos un programa (**prog**) que permite resolver el problema. Para la utilización del mismo se lo debe ejecutar por consola y pasarle los siguientes parámetros:

- **Archivo de instancia:** Es el nombre del archivo que contiene la instancia del grafo y sus particiones que se quiere procesar.

- **Archivo de salida:** Es el nombre del archivo que va a contener la solución del problema.
- **Cantidad de iteraciones de planos de corte:** Es la cantidad de iteraciones de planos de corte que se van a utilizar en el algoritmo Cut and Branch. Si no se pasa ningún valor para este parámetro por default se utilizará 3.
- **Varsel:** Establece la regla para la selección de la variable de ramificación en el nodo que ha sido seleccionado para ramificaciones a la hora de recorrer el árbol. Por default se utiliza 0.
- **Nodesel:** Utilizado para establecer la regla para seleccionar el próximo nodo a procesar cuando se produce retroceso en el recorrido del árbol. Por default se utiliza 0.

Un ejemplo de como se ejecuta el programa es:
./prog archivoInstancia archivoSalida 10
El resultado de la ejecución va a ser el archivo de salida que tendrá el coloreo válido con la menor cantidad de colores posibles.

En el siguiente trabajo vamos a poder encontrar el modelo de programación lineal entera, la descripción de los algoritmos utilizados, demostraciones de que las desigualdades usadas para generar planos de corte son válidas para todo punto factible del modelo. Junto con toda la explicación del proceso realizado, vamos a poder encontrar y entender cuales fueron los experimentos que se realizaron para corroborar que el algoritmo implementado es correcto y que además está optimizado. Por último vamos a presentar una serie de conclusiones a las que hemos llegado.

2 Modelado del Problema

Modelamos el problema del coloreo particionado de grafos con un modelo de Programación Lineal Entera de la siguiente manera:

Dado $G = (V, E)$, P conjunto de particiones, definimos las variables:

- Para $p = 1, \ldots, n$, $j = 1, \ldots, \#P$

$$x_{pj} = \begin{cases} 1 \text{ si el color } j \text{ es asignado al vértice } p \\ 0 \text{ en caso contrario} \end{cases}$$

- Para $j = 1, \ldots, \#P$

$$w_j = \begin{cases} 1 \text{ si } x_{pj} = 1 \text{ para algún vértice } p \\ 0 \text{ en caso contrario} \end{cases}$$

El modelo de programación lineal entera es:

Minimizar la cantidad de colores utilizados

$$\text{Min} \sum_{j=1}^{\#P} w_j$$

Sujeto A

- Dos nodos adyacentes no estan pintados del mismo color

$$x_{pj} + x_{qj} \leq 1 \ (p, q) \in E, \ j = 1, \ldots, \#P$$

- Si un nodo usa el color $j \Rightarrow w_j = 1$

$$x_{pj} - w_j \leq 0 \ p = 1, \ldots, n, \ j = 1, \ldots, \#P$$

- Cada partición debe tener exactamente un nodo pintado

$$\sum_{p \in K} \sum_{j=1}^{\#P} x_{pj} = 1 \ \forall K \text{ partición de } P$$

- Las variables son binarias

$$x_{pj} \in \{0, 1\} \ p = 1, \ldots, n, \ j = 1, \ldots, \#P$$
$$w_j \in \{0, 1\} \ j = 1, \ldots, \#P$$

La primer restricción es una restricción intrínseca de los problemas de coloreo: dos nodos adyacentes no pueden estar pintados del mismo color.

La segunda restricción relaciona las variables del problema: Si algun nodo es pintado del color j, entonces la variable wj debe valer 1.

La tercer restricción es para este problema en particular: cada partición debe tener exactamente un nodo pintado.

Si bien este modelo es adecuado para el problema, tiene la desventaja de tener soluciones simétricas (tenemos muchas soluciones con el mismo coloreo pero usando distintos colores). Sin embargo, utilizarlo puesto que el objetivo del trabajo es analizar la performance de distintos algoritmos de branch & bound, tener un modelo más complejo sólo entorpecería el análisis.

3 Algoritmos

3.1 Algoritmo Branch and Bound

Un algoritmo *Branch and Bound* es un algoritmo que se utiliza para resolver problemas de programación lineal entera o mixtos.

El algoritmo consiste en tomar la relajación lineal del LP original y luego resolverla. Llamemos x^* a la solución de la relajación lineal.

Si la solución obtenida de la relajación tiene en todas las variables de tipo entero valores enteros ($x_i^* \in \mathbb{Z}, \forall x_i$ variable entera), entonces el algoritmo termina y la solución obtenida de la relajación (x^*) es solución óptima del LP original.

Si existe al menos una variable entera cuyo valor en la solución de la relajación lineal no es entero ($\exists x_i$ variable entera tal que $x_i^* \notin \mathbb{Z}$), debemos partir el problema en dos subproblemas *(Branching)* y resolver cada uno por separado. Para partir el problema, podemos tomar al x_i^* que no es entero y separar en dos dejando por un lado los puntos factibles que tienen $x_i \leq \lfloor x_i^* \rfloor$ y por otro lado las que tienen $x_i \geq \lfloor x_i^* \rfloor + 1$.

Luego se aplica recursión a cada subproblema hasta obtener distintas soluciones enteras y nos quedamos con una que sea óptima. Si en algún momento obtenemos un valor de la función objetivo de la relajación lineal que es peor que un valor entero que tengamos, podemos podar esa rama ya que sabemos que las soluciones enteras que se encuentren en esa región no van a tener valores de función objetivo mejores que su relajación.

En resumen, lo que hacen los algoritmos branch and bound es dividir el problema en subproblemas mas chicos y luego resolverlos hasta quedarse con una solución óptima. Para recorrer todas las posibles soluciones lo que se va armando es un árbol y se lo va recorriendo en busca de soluciones.

Las variantes que pueden utilizarse en los algoritmos branch and bound es la forma en la que se recorre el árbol. Puede definirse qué nodo se explora primero y en qué dirección se hace. Dependiendo de la manera en la que se recorre el árbol vamos a ir encontrando distintas soluciones intermedias y los tiempos que se demora en recorrer todo el árbol pueden variar mucho según la estrategia que se utilice.

En el presente trabajo para la implementación del algoritmo de branch and bound se utilizó la librería CPLEX y los algoritmos que la misma tiene para resolver los problemas. Experimentamos variando los parámetros **NODESEL** y **VARSEL** de la librería.
El algoritmo 1 muestra el pseudocódigo del algoritmo usado en el punto 2 del trabajo.

Algoritmo 1 Resolver Problema usando Solo Branch and Bound

Entrada: Archivo que contiene instancia de grafo con particiones $G(V, E, P)$.
Salida: Un archivo que contiene un coloreo válido y óptimo para el grafo y el tiempo de corrida.

1: $G \leftarrow$ leer archivo de entrada
2: $lp \leftarrow$ crear contexto de CPLEX
3: Setear parámetros para correr utilizando branch and bound puro
4: $lp \leftarrow generarLP(grafo)$
5: $restringirLP(lp)$ // Setea como binarias las variables que deben serlo
6: $resultado, tiempoDeCorrida \leftarrow resolverLP(lp)$
7: $archivoDeSalida \leftarrow generarResultados(resuldado, tiempoDeCorrida)$
8: **devolver** $archivoDeSalida$

3.2 Algoritmos de Planos de Corte

Los algoritmos de planos de corte son otra técnica que se utiliza para resolver problemas de programación lineal entera o mixtos al igual que branch and bound.

Estos algoritmos consisten en tomar la relajación lineal del problema original y resolverla al igual que lo hacen los algoritmos branch and bound. Llamemos nuevamente x^* al óptimo obtenido al resolver la relajación lineal del porblema original.

Al igual que la anterior técnica, en el caso de que todas las variables enteras tengan valores enteros, terminamos el algoritmo dado que encontramos un óptimo que cumple con todas las restricciones que debe cumplir.

Si x^* no cumple con todas las restricciones de integralidad, estamos en una solución que no es factible. Lo que hacemos ahora en lugar de dividir el problema y resolver las distintas partes como hace la técnica de branch and bound, es obtener una o varias desigualdades que sean válidas para el conjunto de soluciones factibles S pero que no sea válida para la solución obtenida hasta el momento x^* y las agregamos a nuestro LP. Luego volvemos a resolver la relajación lineal del LP.

De esta forma lo que hacemos es ir reduciendo la región factible sin dejar ninguna solución factible fuera de forma tal que en algún momento la solución de la relajación lineal nos de un resultado factible para el problema original.

Es muy importante que las desigualdades que se agreguen sean válidas para todo punto factible ya que de no serlo podemos dejar afuera alguna solución y el óptimo obtenido podría no ser el verdadero óptimo.

También es muy importante que las desigualdades agregadas dejen afuera a x^* dado que en caso de no hacerlo la próxima iteración al resolver la relajación lineal va a dar el mismo resultado.

En la práctica se suele usar familias de desigualdades para ir agregando en

cada iteración de planos de corte. En el presente trabajo práctico se utilizaron dos familias de desigualdades distintas: la **desigualdad de clique maximal** y la **desigualdad de agujero impar**.

3.2.1 Familia de Desigualdades de Clique Maximal

Sea $j_0 \in 1, ..., \#P$ y sea K una clique maximal de G. La desigualdad *clique* está definida por:

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

Propiedad: La desigualdad *clique* es válida para todo punto de S.

Demostración: Por el absurdo, supongamos que esto no pasa. Entonces existe una clique K y un j_0 tal que $\sum_{p \in K} x_{pj_0} > w_{j_0}$.

Entonces tiene que suceder:

$$w_{j_0} = 0 \wedge \exists p \ / \ x_{pj_0} = 1 \tag{1}$$

$$w_{j_0} = 1 \wedge \exists p, \ q \ / \ x_{pj_0} = 1 \wedge x_{qj_0} = 1 \tag{2}$$

De 1 tenemos que $x_{pj_0} - w_{j_0} = 1$, lo cual es un absurdo, pues es una restricción impuesta en el modelo del LP.

De 2 tenemos que $x_{pj_0} + x_{qj_0} = 2$, que también es absurdo: p y q son adyacentes pues pertenecen a una clique, no pueden tener el mismo color.

El absurdo provino de suponer que

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

no era válida para todo punto de S.

Luego

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

es válido para todo punto de S.

3.2.2 Familia de Desigualdades de Agujero Impar

Sea $j_0 \in 1, ..., \#P$ y sea $C_{2k+1} = v_1, ..., v_{2k+1}, k \geq 2$ un agujero de longitud impar en G. La desigualdad *add-hole* está definida por:

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

Propiedad: La desigualdad *add-hole* es válida para todo punto de S.

Demostración: Probamos por *Chvátal-Gomory*

$$
\begin{gathered}
x_{v_1 j_0} + x_{v_2 j_0} \leq 1 \\
x_{v_2 j_0} + x_{v_3 j_0} \leq 1 \\
\vdots \\
x_{v_{2k} j_0} + x_{v_{2k+1} j_0} \leq 1 \\
x_{v_{2k+1} j_0} + x_{v_1 j_0} \leq 1 \\
\hline
2x_{v_1 j_0} + 2x_{v_2 j_0} + \ldots + 2x_{v_{2k} j_0} + 2x_{v_{2k+1} j_0} \leq 2k+1
\end{gathered}
$$

Si $w_{j_0} = 0$, entonces todos los $x_{pj_0} = 0$, luego la desigualdad vale.
Si $w_{j_0} = 1$, tenemos que:

$$2x_{v_1 j_0} + 2x_{v_2 j_0} + \ldots + 2x_{v_{2k} j_0} + 2x_{v_{2k+1} j_0} \leq (2k+1)w_{j_0}$$

Que esto es igual a:

$$
\begin{aligned}
2 \sum_{p \in C_{2k+1}} x_{pj_0} &\leq 2kw_{j_0} + w_{j_0} \\
\sum_{p \in C_{2k+1}} x_{pj_0} &\leq kw_{j_0} + \frac{1}{2} w_{j_0}
\end{aligned}
$$

Pero como x_{pj_0} y w_{j_0} son variables enteras y en particular son binarias, vale que:

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

Luego, la desigualdad es válida para todo punto de S.

3.3 Algoritmo Cut and Branch

Al igual que las técnicas anteriores, los algoritmos *Cut and Branch* sirven para resolver problemas de programación lineal entera o mixtos.

Proviene de unir las dos técnicas mencionadas anteriormente. Para comenzar se realiza una cantidad de iteraciones de planos de corte y luego se termina de resolver el problema aplicando branch and bound.

Si bien pareciera que esta técnica no aporta nada nuevo, los resultados que se obtienen al combinar ambas técnicas son muy positivos.

Dentro de esta técnica hay muchas variables que podemos modificar para experimentar los comportamientos, por ejemplo la cantidad de iteraciones de planos de corte, tipos de desigualdades que se agregan en cada iteración, cantidad de iteraciones de cada tipo que se agregan y por supuesto las formas de recorrer el árbol al hacer branch and bound.

Cada iteración de planos de corte agrega desigualdades de las familas que se detallaron anteriormente. La cantidad de desigualdades por familia depende del algoritmo que se implementó para cada una, el cual se va a explicar a continuación. Y por último, la cantidad de iteraciones de planos de corte a utilizar fue variando durante la experimentación para observar el comportamiento, los

detalles se encuentran en la sección de Experimentación.

Para implementar un algoritmo *cut and branch* lo que hicimos es leer la instancia del archivo de entrada y luego generar el modelo de programación lineal asociado al problema. En esta instancia nos quedamos con la relajación lineal del problema original ya que lo que vamos a hacer es agregar las desigualdades provenientes de los planos de corte y para eso necesitamos que el problema sea de programación lineal y no programación lineal entera.

Una vez que tenemos la relajación lineal agregamos todos los planos de corte que sean necesarios. El funcionamiento de este algoritmo se explicará más adelante.

Cuando los planos de corte ya fueron agregados y la región factible fue reducida, transformamos la relajación lineal del problema en un problema de programación lineal entera modificando el tipo de variable de todas las variables de continua a binaria.

Finalmente resolvemos el problema utilizando *Branch & Bound*.

Algoritmo 2 Algoritmo Cut and Branch

Entrada: Archivo que contiene instancia de grafo con particiones $G(V, E, P)$, cantidad de iteraciones de planos de corte $cantIteracionesPlanosDeCorte$.
Salida: Un archivo que contiene un coloreo válido y óptimo para el grafo y el tiempo de corrida.

1: $G \leftarrow$ leer archivo de entrada
2: $lp \leftarrow$ crear contexto de CPLEX
3: Setear parámetros para correr utilizando branch and bound puro
4: $lp \leftarrow generarLP(grafo)$
5: $agregarPlanosDeCorte(lp, G, cantIteracionesPlanosDeCorte)$
6: $restringirLP(lp)$ *//Setea como binarias las variables que deben serlo*
7: $resultado, tiempoDeCorrida \leftarrow resolverLP(lp)$
8: $archivoDeSalida \leftarrow generarResultados(resuldado, tiempoDeCorrida)$
9: **devolver** $archivoDeSalida$

El algoritmo 2 muestra el pseudocódigo del algoritmo implementado para el punto 4 del trabajo.

Dado que los planos de corte mencionados anteriormente son independientes para cada color, decidimos tratar cada color por separado.

Tomamos el conjunto de variables x^*_{ic} y w^*_c para cada color c. Al momento de armar un agujero impar o una clique, analizamos los nodos en orden descendente según el valor de x^*_{ic}, es decir, miramos primero los nodos con mayor valor en la solución.

Cada familia de desigualdades tiene su propio algoritmo para encontrar desigualdades que sean violadas por x^* y las agrega al lp.

Algoritmo 3 Agregar Planos de Corte

Entrada: El lp, el grafo $G(V, E, P)$ y cantidad de iteraciones de planos de corte $cantIteracionesPlanosDeCorte$.

Salida: Agrega las desigualdades dadas por los planos de corte a lp.

```
1:  cantidadDeColores ← G.cantidadDeParticiones
2:  para i = 1; i < cantIteracionesPlanosDeCorte; i++ hacer
3:    G ← leer archivo de entrada
4:    solucion, tiempoDeCorrida ← resolverLP(lp)
5:    para j = 1; j < cantidadDeColores; j++ hacer
6:      valorWj ← solucion_{w_j}
7:      variables ← solucion_{x_{1j},...,x_{nj}}
8:      ordenarDescendentemente(variables)
9:      agregarCliquesQueViolenDesigualdad(lp, variables, color, valorWj, grafo)
10:     agregarAgujerosImparesQueViolenDesigualdad(lp, variables, color, valorWj, grafo)
11:   fin para
12: fin para
```

El algoritmo 3 muestra como se agregan las desigualdades de planos de corte de ambas familias en cada iteración.

A continuación se describen los algoritmos particulares usados para cada familia de desigualdades.

3.3.1 Agregar cliques máximas cuyas desigualdades violen la solución

Para encontrar desigualdades clique violadas por la solución de la relajación lineal, lo que hacemos es recorrer los nodos a medida que tienen mayor valor en la solución de la relajación. Por cada nodo recorremos el resto y vamos agregando los nodos que son adyacentes a todos los que forman parte de la clique para obtener una clique maximal. Una vez que agregamos todos los nodos que podíamos, verificamos que la clique no esté contenida en otra clique previamente usada. Este último paso es para sólo agregar cliques maximales.

Repetimos este proceso comenzando a armar la clique por cada nodo del grafo. Realizamos esto para poder agregar varias cliques en la misma iteración para el mismo color. En una primera versión agregábamos solo una clique por cada color en cada iteración. Luego probamos agregar todas las que podíamos y comprobamos que de esta manera el algoritmo era mas performante y por eso dejamos ese cambio.

El algoritmo 4 muestra como se generan y agregan al LP las desigualdades de la familia de clique máxima al LP.

Algoritmo 4 Agregar cliques que violen desigualdad

Entrada: El lp, los valores de la solucion para el color $solucion$, el color $color$, el valor de la solución para la variable correspondiente al color $valorWj$ y el grafo G.

Salida: Agrega desigualdades de la familia clique al lp que sean violadas por $soluion$. *//solucion es un array donde en cada posicion tenemos el nodo y el valor que tiene la variable* $x_{nodo,j}$

1: $coeficienteWj \leftarrow -1$
2: $cliquesUsadas \leftarrow \{\}$
3: **para** i = 1; i < G.cantidadDeNodos; i++ **hacer**
4: $clique \leftarrow \{solucion_i.nodo\}$
5: $sumaClique \leftarrow solucion_i.valor$
6: **para** j = i+1; j < G.cantidadDeNodos; j++ **hacer**
7: $nodo \leftarrow solucion_j.nodo$
8: **si** G.esAdyacenteATodos(clique, nodo) **entonces**
9: $clique \leftarrow clique \cup \{nodo\}$
10: $sumaClique \leftarrow sumaClique + solucion_j.valor$
11: **fin si**
12: **fin para**
13: **si** $clique \in cliquesUsadas$ **entonces**
14: continue
15: **fin si**
16: **si** $sumaClique > valorWj$ **entonces**
17: $agregarDesigualdadALP(lp, clique, color, cowficienteWj)$
18: $cliquesUsadas \leftarrow cliquesUsadas \cup \{clique\}$
19: **fin si**
20: **fin para**

3.3.2 Agregar agujeros impares cuyas desigualdades violen la solución

Para encontrar desigualdades de esta familia, lo que necesitamos es encontrar circuitos de longitud impar pertenecientes al grafo cuya desigualdad correspondiente sea violada por la solución obtenida.

Vamos a modelar el circuito como una lista ordenada de nodos donde el primero y el último sean el mismo, luego lo que vamos a hacer es eliminar el último y nos quedamos con el resto de los nodos que ordenados van a formar un circuito en el grafo.

La manera con la que vamos a encontrar esos circuitos es partir de un circuito inicial formado por una única arista que pertenezca al grafo.

A partir del circuito que tenemos, vamos recorrer los nodos que lo forman en orden y entre cada par de nodos (llamémoslos $nodo_1$ y $nodo_2$) vamos a buscar un nodo que no pertenezca al circuito y que sea adyacente a $nodo_1$ y $nodo_2$. Cuando encontramos un nodo que cumple con esa condición, lo que hacemos es insertarlo en el circuito entre $nodo1$ y $nodo_2$ como se puede apreciar en las imagenes 1, 2 y 3.

El nodo agregado pasa a ser el nuevo $nodo_2$ y se repite el proceso. Si entre dos nodos no se encuentra ningún posible candidato para ser agregado, se pasa al siguiente par de nodos (siendo $nodo_2$ el nuevo $nodo_1$).

Cada vez que podemos agregar un nodo al circuito, verificamos si la longitud del mismo es impar. En caso de serlo, sea $2K + 1$ la longitud del circuito, verificamos si la suma de todos los nodos que forman el circuito es mayor que $K * valorWj$ (es decir, si viola la desigualdad). Si lo es quiere decir que encontramos un agujero impar cuya desigualdad asociada es violada por la solución actual. Guardamos esta solución temporalmente.

Una vez que no se puede agrandar más el agujero, se toma el agujero impar más grande que violó la desigualdad (notar que este puede no ser el agujero impar más grande que se generó).

El motivo por el cual modelamos al circuito como una lista donde el primer y el último nodo son el mismo es para no perder la posibilidad de insertar nodos entre el último nodo del agujero y el primero (que en realidad al ser un circuito no es importante por donde se comienza a recorrerlo).

Figure 1: Arista actual

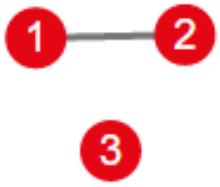

Figure 2: Arista actual y nodo adyacente a ambos nodos

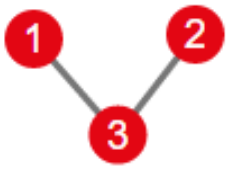

Figure 3: Nodo agregado entre los nodos que forman la arista

El algoritmo termina cuando se recorre el último par de nodos del circuito y no hay mas nodos para agregar.

En cada iteración de planos de corte ejecutamos el proceso mencionado anteriormente tantas veces como cantidad de nodos tenga el grafo por color para no agregar una única desigualdad por color por iteración, sino que podemos aprovechar y agregar varias. Pusimos la cota de cantidad de nodos para no agregar demasiadas ya que el proceso podría demorarse mucho y afectar la performance del algoritmo.

Sin embargo, cada vez que agregamos un agujero impar, comenzamos por dos nodos que no hayan sido usados previamente en otro agujero impar. (el resto de los nodos del agujero sí pueden repetirse, solo tratamos de garantizar que los agujeros sean un tanto distintos entre sí).

El algoritmo 5 muestra como se generan y agregan al LP las desigualdades de la familia de agujero impar.

Algoritmo 5 Agregar Agujeros impares que violen desigualdad

Entrada: El lp, los valores de la solucion para el color *solucion*, el color *color*, el valor de la solución para la variable correspondiente al color $valorWj$ y el grafo G.

Salida: Agrega desigualdades de la familia agujero impar al lp que sean violadas por *soluion*. {//*solucion* es un array donde en cada posicion tenemos el *nodo* y el *valor* que tiene la variable $x_{nodo,j}$}

1: $nodosUsados \leftarrow \{\}$
2: **para** $rep = 1; rep \leq G.cantidadDeNodos; rep++$ **hacer**
3: $sumaAgujero \leftarrow 0$
4: $agujero \leftarrow []$
5: **para** cada par de nodos $(p, q) \in solucion$ **hacer**
6: **si** $(p, q) \notin E(G) || p \in nodosUsados || q \in nodosUsados$ **entonces**
7: continue
8: **fin si**
9: $agujero \leftarrow [p, q, p]$
10: $sumaAgujero \leftarrow p.valor + q.valor$
11: Break For
12: **fin para**
13: **si** $agujero.size == 0$ **entonces**
14: Return
15: **fin si**
16: $indice \leftarrow 1$
17: $agujeroMasGrande \leftarrow []$
18: **mientras** $indice \leq agujero.size$ **hacer**
19: $(p, q) \leftarrow agujero_{i-1}, agujero_i$
20: **si** Existe nodo v ady. a p y q que no esté en el agujero **entonces**
21: Agregar a v entre p y q en $agujero$
22: $sumaAgujero \leftarrow sumaAgujero + v.valor$
23: $coefWj \leftarrow (agujero.size - 1)/2$
24: **si** $agujero.size$ es impar $\wedge sumaAgujero > coefWj * valorWj$ **entonces**
25: $agujeroMasGrande \leftarrow agujero$
26: **fin si**
27: Continue
28: **fin si**
29: $indice \leftarrow indice + 1$
30: **fin mientras**
31: **si** $agujeroMasGrande.size > 0$ **entonces**
32: $agregarDesigualdadALP(lp, agujero, color, cowficienteWj)$
33: $nodosUsados \leftarrow nodosUsados \cup agujeroMasGrande$
34: **fin si**
35: **fin para**

4 Experimentación

4.1 Metodología

Para realizar la experimentación y que las mediciones sean confiables, decidimos correr todas las instancias de prueba en una misma computadora sin estar corriendo nada más para que las pruebas no sean alteradas.

Al correr las instancias de prueba elegimos setear un timeout de 10 minutos para cada una ya que creemos que para hacer una prueba lo menos sesgada posible necesitábamos correr muchas instancias y para eso es necesario setear un timeout porque sinó podrían no terminar nunca. Decidimos que el mismo sea 10 minutos ya que nos parece que es un período de tiempo bastante amplio que nos va a permitir observar resultados muy interesantes. Si bien creemos que si extendemos el timeot obtendremos resultados mas precisos, lamentáblemente el tiempo que tomarían las pruebas sería demasiado y no podemos tomarnos tanto tiempo ya que no lo tenemos.

Dividimos la experimentación en dos partes: recorrido del árbol al resolver branch and bound y cantidad de iteraciones de planos de corte para resolver cut and branch.

La primera parte consiste en explorar las distintas alternativas de recorrer el árbol. Esto lo hicimos a través de configuración de parámetros de la libraría CPLEX. Los parámetros que fuimos modificando fueron:

- **CPX_PARAM_VARSEL**: Establece la regla para la selección de la variable de ramificación en el nodo que ha sido seleccionado para ramificaciones.
- **CPX_PARAM_NODESEL**: Utilizado para establecer la regla para seleccionar el próximo nodo a procesar cuando se produce reversión de rastreo.

Consideramos que haciendo variar ambos parámetros ibamos a poder saber cómo se comporta el algoritmo en cada caso y poder decidir qué valor de qué parámetro era mas conveniente para resolver nuestro problema. Para las pruebas hicimos todas las combinaciones posibles de estos parámetros para así poder observar bien cual combinación era la mejor.

La segunda parte consiste en ver el comportamiento del algoritmo con la distinta cantidad de planos de corte, para realizar estas pruebas dejamos fijos los valores de los parámetros CPX_PARAM_VARSEL y CPX_PARAM_NODESEL en el valor por default para que CPLEX eligiera la mejor alternativa ya que en las pruebas que hicimos fueron los mejores resultados obtenidos. Además en esta instancia queríamos probar cómo se comportaban los planos de corte que fuimos agregando, con lo cual fuimos variando para una misma instancia de prueba la cantidad de planos de corte que utilizamos.

Para ambos casos usamos instancias de prueba con distinta cantidad de nodos, densidades de aristas y cantidad de particiones.

4.1.1 Instancias de Prueba

Utilizamos dos tipos de instancias de prueba:

- **Instancias generadas aleatoriamente**: Para generar las instancias de manera aleatoria, creamos dos programas, uno para generar instancias de grafos al cual le pasábamos la cantidad de nodos y la densidad de aristas y generaba un archivo con la instancia. Luego para particionar el grafo creamos otro programa, el cual lee el archivo de instancia de un grafo y se le indica la cantidad de particiones que queremos obtener del mismo y genera un archivo con el grafo particionado.
- **Instancias del problema de coloreo de grafos conocidas**: Son instancias que tomamos de la página de internet *http://mat.gsia.cmu.edu/color/instances.html*. Sobre estas instancias elegimos algunas que tengan una cantidad de nodos que no sean demasiados para que las pruebas puedan ejecutarse. Luego para todas esas instancias generamos las particiones con el programa anteriormente mencionado.

Para las pruebas realizadas con instancias aleatorias utilizamos instancias de 30, 40 y 50 nodos, con densidades 0.10, 0.25, 0.50, 0.75 y 0.90 de forma tal de poder experimentar con grafos de distintos tamaños y todas las densidades posibles.

Para las cantidades de particiones utilizamos 5, 10, 15 y 20. Más particiones hicieron que los algoritmos demoren mucho tiempo y las dejamos afuera de la experimentación.

A su vez para cada combinación de cantidad de nodos y densidades generamos 5 instancias diferentes de grafos y luego para cada una de ellas y cada cantidad de particiones generamos 5 particiones distintas. De esta forma nos aseguramos tener una alta cantidad de casos de pruebas de instancias muy variadas y al tomar los promedios creemos que los resultados son bastante fiables. En resumen, para cada combinación de cantidad de nodos, densidad de aristas y cantidad de particiones tenemos 25 instancias diferentes para luego promediar los resultados.

Para las instancias tomadas de la página de internet seguimos la misma idea. Tratamos de experimentar con instancias de distintos tamaños y distintas densidades. Utilizamos las instancias myciel3, myciel4, myciel5, myciel6, queen5_5, queen6_6, queen7_7, queen8_8, queen8_12, queen9_9. Para cada una utilizamos las cantidades de particiones 5, 10, 15, 20 con excepción de las primeras instancias que tienen menos nodos. Para cada combinación de instancia y cantidad de particiones generamos 5 particiones distintas para poder tomar promedios.

En todos los casos para poder experimentar el comportamiento al variar la cantidad de iteraciones de planos de corte corrimos cada instancia con 0, 3, 8, 11 y 19 iteraciones de planos de corte y luego comparamos los resultados.

En total fueron mas de 8000 casos de prueba (varios dias de corridas) de instancias realmente variadas, con lo cual consideramos que los reusltados realmente son bastante fieles.

4.2 Resultados

Dadas la cantidad de pruebas que realizamos, decidimos agrupar las instancias en base a la cantidad de nodos, densidad de aristas y cantidad de particiones y tomar promedios de tiempos para poder presentar los resultados y que sean fáciles de leer.

Además, al tomar promedios de los tiempos de corridas y no solo los valores absolutos de cada instancia, logramos minimizar el sesgo que puede producir alguna instancia que sea un outlayer.

Todos los tiempos de corrida que se pueden ver en las tablas están medidos en segundos.

Dado que no todas las instancias pudieron terminar exitosamente antes del timeout, en todas las tablas aclaramos cuantas instancias si lo hicieron para poder dar una idea de que las que faltan es porque tardaron mas de 10 minutos en ejecutarse y no las tomamos en cuenta a la hora de promediar los resultados ya que de haberlo hecho ibamos a obtener resultados alterados por estas instancias.

4.2.1 Resultados de instancias generadas aleatoriamente para Branch and Bound

Para realizar la experimentación sobre las maneras que tenemos de recorrer el árbol al utilizar branch and bound decidimos tomar instancias generadas aleatoriamente con 20 nodos y hacer variar la densidad de aristas del grafo, cantidad de particiones y los parámetros de CPLEX sobre cómo se recorre el árbol.

Usamos una cantidad fija de nodos ya que para hacer variar el tamaño de la instancia era suficiente con aumentar la cantidad de aristas y la cantidad de particiones de cada instancia. Además tomar instancias mucho mas grandes iba a hacer que las pruebas tomen muchísimo mas tiempo ya que en esta instancia de la experimentación, todavía no hay nada optimizado y lo que se ejecuta es Branch and Bound puro.

Para cada densidad y cada cantidad de particiones se tomaron 25 instancias diferentes, 5 instancias de grafos distintas y a su vez a cada una se le realizó 5 particiones distintas para luego al tomar los promedios poder lograr resultados mas fieles.

En la tabla 1 se pueden ver los resultados obtenidos al correr exta parte de la experimentación. Como bien se puede ver tomamos todas las combinaciones posibles de recorridos del árbol, esto lo hicimos variando los valores de los parámetros CPX_PARAM_VARSEL y CPX_PARAM_NODESEL.

Luego comparamos los resultados y pudimos concluir que para las instancias mas pequeñas, los tiempos al variar los valores en los parámetros no varían demasiado. En cambio en las instancias mas grandes, podemos ver que para ciertas combinaciones de parámetros (es decir, ciertas formas de recorrer el árbol) algunas instancias no pudieron terminar antes del timeout mientras que otras lo

hicieron sin problemas.

Concluimos que la mejor opción es dejar ambos parámetros con valor 0 ya que fue la opción que mejores resultados nos brindó. En los siguientes experimentos dejamos fijo esos valores en 0.

			Cantidad de Particiones					
			5		8		10	
Densidad	Varsel	Nodesel	#	Tiempo	#	Tiempo	#	Tiempo
0.25	-1	0	25	0.0082	25	0.0466	24	0.0470
	0	0	25	0.0088	25	0.0545	25	0.0635
	0	1	25	0.0082	25	0.0541	25	0.0678
	0	2	25	0.0087	25	0.0554	25	0.0690
	0	3	25	0.0079	25	0.0552	25	0.0688
	1	0	25	0.0078	25	0.0482	25	0.2430
	2	0	25	0.0078	25	0.0533	25	0.0649
	3	0	25	0.0083	25	0.0528	25	0.1227
	4	0	25	0.0087	25	0.0309	25	0.0414
0.50	-1	0	25	0.0364	25	1.2467	24	3.5322
	0	0	25	0.0437	25	8.1819	25	0.8758
	0	1	25	0.0430	25	0.2400	25	0.7347
	0	2	25	0.0436	25	0.2158	25	0.7899
	0	3	25	0.0437	25	0.2187	25	0.7821
	1	0	25	0.0349	25	1.2910	25	3.6988
	2	0	25	0.0432	25	0.2301	25	0.8731
	3	0	25	0.0335	25	0.6747	25	2.0184
	4	0	25	0.0248	25	0.1100	25	0.3302
0.75	-1	0	25	0.0537	25	7.7119	23	112.69
	0	0	25	0.0648	25	0.9984	25	10.657
	0	1	25	0.0632	25	1.1411	25	11.9340
	0	2	25	0.0620	25	1.1974	25	11.8280
	0	3	25	0.0636	25	1.1898	25	11.573
	1	0	25	0.0489	25	6.2028	23	86.872
	2	0	25	0.0630	25	1.0042	25	10.658
	3	0	25	0.0823	25	3.1502	25	57.133
	4	0	25	0.0315	25	0.3810	25	6.2590

Table 1: Tabla de resultados para instancias de Branch and Bound puro.

4.2.2 Resultados de instancias generadas aleatoriamente para Cut and Branch

Como bien dijimos anteriormente, para las intancias generadas aleatoriamente tomamos grafos con distintas cantidades de nodos, distintas densidades y cantidad de particiones. Para cada combinación probamos correr 25 casos y tomamos el promedio de los tiempos y la cantidad de casos que terminaron antes del time-out. Al igual que en el caso anterior, los 25 casos de cada tamaño son debido a que se toman 5 instancias de grafos distintas y luego para cada una se realizan 5 particiones.

En las tablas 2, 3 y 4 podemos ver los resultados obtenidos al correr los algoritmos para las instancias generadas aleatoriamente.

		30 nodos									
		Iteraciones de Planos de Corte									
		0		3		8		11		19	
Den	Part	#	Tiempo	#	Tiempo	#	Tiempo	#	Tiempo	#	Tiempo
0.10	5	25	0.0064	25	0.0101	25	0.0135	25	0.0158	25	0.0197
	10	25	0.0230	25	0.0289	25	0.0348	25	0.0416	25	0.0429
	15	25	0.2427	25	0.2613	25	0.2027	25	0.1398	25	0.0991
	20	25	0.6275	25	0.5911	25	0.1507	25	0.1239	25	0.1397
0.25	5	25	0.5296	25	0.0229	25	0.0420	25	0.0505	25	0.0625
	10	25	0.1838	25	0.2414	25	0.2338	25	0.2540	25	0.1405
	15	25	2.5368	25	2.6084	25	1.7262	25	1.4833	25	1.0098
	20	25	103.85	25	50.464	25	34.515	25	40.746	25	19.884
0.50	5	25	0.0552	25	0.0925	25	0.1472	25	0.1550	25	0.2593
	10	25	2.4784	25	2.1307	25	2.3663	25	2.2929	25	2.5794
	15	24	115.89	24	105.86	25	140.48	25	90.351	25	26.283
	20	0	-	0	-	0	-	0	-	1	6.0065
0.75	5	25	0.2348	25	0.2057	25	0.1003	25	0.1136	25	0.1714
	10	25	105.21	25	74.634	25	35.598	25	15.244	25	5.5406
	15	0	-	2	148,57	10	53.059	14	77.361	22	64.754
	20	0	-	0	-	0	-	0	-	0	-
0.90	5	25	1.3286	25	0.4516	25	0.2235	25	0.2612	25	0.1838
	10	9	179.89	24	151.39	25	15.120	25	5.9877	25	1.8320
	15	0	-	0	-	0	-	0	-	0	-
	20	0	-	0	-	0	-	0	-	0	-

Table 2: Tabla de resultados para instancias aleatorias con 30 nodos.

Se puede apreciar en la tabla 4 que para 50 nodos no tenemos densidades mayores a 0.50, esto es debido a que los casos con densidades mayores solo terminaban para cantidades de particiones muy pequeñas y para datos interesantes no, con lo cual no eran datos que aporten casi nada en la experimentación y por eso los excluimos.

Es fácil notar que para las instancias mas pequeñas (menos densidades de aristas y menor cantidad de particiones) las mejoras que se observan al aumentar la cantidad de iteraciones de planos de corte son mínimas. En cambio para las instancias mas grandes se puede observar como los tiempos se reducen drásticamente a medida que aumentan las iteraciones. De hecho hay casos en los que sin una cierta cantidad de iteraciones de planos de corte hay instancias que no terminaron antes de que pasen los 10 minutos de timeout, mientras que luego de agregar planos de corte pudieron terminar a los pocos segundos.

Estos comportamientos se notan con todas las cantidades de nodos que probamos (30, 40 y 50), con lo cual podemos concluir de que no es un comportamiento casual y que el hecho de agregar iteraciones de planos de corte antes

		40 nodos									
		Iteraciones de Planos de Corte									
		0		3		8		11		19	
Den	Part	#	Tiempo	#	Tiempo	#	Tiempo	#	Tiempo	#	Tiempo
0.10	5	25	0.0087	25	0.0164	25	0.0251	25	0.0316	25	0.0399
	10	25	0.0176	25	0.0421	25	0.0627	25	0.0749	25	0.1003
	15	25	0.5807	25	0.6812	25	0.7078	25	0.6365	25	0.4907
	20	25	1.8693	25	1.5887	25	0.5798	25	0.4633	25	0.3786
0.25	5	25	0.0132	25	0.0266	25	0.0568	25	0.0794	25	0.1020
	10	25	0.7895	25	1.1689	25	1.6132	25	1.9308	25	2.0827
	15	25	0.8396	25	0.9792	25	0.7872	25	0.6269	25	0.8479
	20	25	58.8595	25	44.880	25	29.779	25	18.848	25	13.136
0.50	5	25	0.0630	25	0.1994	25	0.2646	25	0.2990	25	0.4723
	10	25	1.2583	25	1.6976	25	1.5428	25	1.3464	25	2.4238
	15	23	67.340	23	71.610	23	55.373	22	24.712	23	16.786
	20	0	-	0	-	0	-	0	-	0	-
0.75	5	25	0.5614	25	0.5840	25	0.7647	25	0.7532	25	0.4540
	10	23	62.629	24	55.774	25	46.723	25	48.585	25	20.600
	15	0	-	0	-	1	321,73	1	283,47	7	157.91
	20	0	-	0	-	0	-	0	-	0	-
0.90	5	25	0.7135	25	0.4766	25	0.2916	25	0.3629	25	0.3764
	10	6	339.57	19	137.15	21	21.496	23	36.236	25	34.132
	15	0	-	0	-	0	-	0	-	0	-
	20	0	-	0	-	0	-	0	-	0	-

Table 3: Tabla de resultados para instancias aleatorias con 40 nodos.

de recorrer el árbol con branch and bound es muy importante para instancias medianas y grandes.

		50 nodos									
		Iteraciones de Planos de Corte									
		0		3		8		11		19	
Den	Part	#	Tiempo	#	Tiempo	#	Tiempo	#	Tiempo	#	Tiempo
0.10	5	25	0.0115	25	0.0217	25	0.0421	25	0.0553	25	0.0793
	10	25	0.0223	25	0.0629	25	0.1077	25	0.1398	25	0.2035
	15	25	0.5922	25	0.7590	25	0.9196	25	0.9089	25	1.2199
	20	25	1.5077	25	1.5326	25	0.9179	25	0.4972	25	0.5229
0.25	5	25	0.0169	25	0.0446	25	0.1138	25	0.1704	25	0.2463
	10	25	0.9762	25	1.8522	25	2.4573	25	2.9876	25	3.2520
	15	25	1.8343	25	2.4156	25	3.5631	25	3.1739	25	2.1408
	20	24	85.358	25	115.26	24	93.239	25	101.90	24	48.926
0.50	5	25	0.0474	25	0.1068	25	0.2787	25	0.3434	25	0.6615
	10	25	2.9237	25	4.2842	25	4.4376	25	3.3602	25	3.9686
	15	23	186.20	18	163.20	17	122.89	20	161.91	24	89.419
	20	0	-	0	-	0	-	0	-	0	-

Table 4: Tabla de resultados para instancias aleatorias con 50 nodos.

4.2.3 Resultados de instancias de coloreo de grafos conocidas para Cut and Branch

Para probar con instancias de prueba de grafos para coloreo decidimos tomar algunas de la página y variar la cantidad de particiones que les realizamos a las mismas. Tomamos 5 particiones distintas de cada cantidad para poder tomar un promedio y que las mediciones no estén sesgadas.

Utilizamos dos familias de instancias, las myciel y las queen. En las tablas 5 y 6 se pueden observar los resultados obtenidos.

		Familia de instancias MyCiel									
		Iteraciones de Planos de Corte									
		0		3		8		11		19	
Nombre	Part	#	Tiempo	#	Tiempo	#	Tiempo	#	Tiempo	#	Tiempo
myciel3 (11N y 20A)	5	5	0.0130	5	0.0149	5	0.0129	5	0.0150	5	0.0130
	10	5	0.1265	5	0.1121	5	0.0662	5	0.0648	5	0.0647
myciel4 (23N y 71A)	5	5	0.0116	5	0.0129	5	0.0160	5	0.0209	5	0.0265
	10	5	0.0622	5	0.0678	5	0.0797	5	0.0554	5	0.0551
	15	5	1.4414	5	1.3819	5	0.9947	5	1.0278	5	0.8797
	20	5	303.25	2	334.25	5	336.73	5	299.62	5	281.95
myciel5 (47N y 236A)	5	5	0.0151	5	0.0246	5	0.0402	5	0.0613	5	0.0811
	10	5	0.0399	5	0.0713	5	0.1165	5	0.1412	5	0.2202
	15	5	1.2562	5	1.4595	5	1.5421	5	1.0859	5	0.5997
	20	5	2.9598	5	3.6555	5	1.9061	5	1.6664	5	1.2531
myciel6 (95N y 755A)	5	5	0.0433	5	0.0817	5	0.1728	5	0.2124	5	0.3934
	10	5	0.1933	5	0.3266	5	0.7448	5	1.0807	5	1.6782
	15	5	0.1164	5	0.5528	5	1.4218	5	1.4612	5	2.1135
	20	5	0.4626	5	1.0378	5	2.0079	5	2.5290	5	4.4451

Table 5: Tabla de resultados para instancias aleatorias con 50 nodos.

		Familia de instancias Queen									
		Iteraciones de Planos de Corte									
		0		3		8		11		19	
Nombre	Part	#	Tiempo	#	Tiempo	#	Tiempo	#	Tiempo	#	Tiempo
queen5_5 (25N y 320A)	5	5	0.0845	5	0.0947	5	0.1065	5	0.1327	5	0.1385
	10	5	2.7837	5	2.0669	5	0.9861	5	0.6290	5	0.2752
	15	5	26.689	5	15.231	5	8.8021	5	1.3639	5	0.7866
	20	0	-	0	-	4	79.793	5	20.146	5	1.3215
queen6_6 (36N y 580A)	5	5	0.0583	5	0.0486	5	0.1583	5	0.1080	5	0.1355
	10	5	0.6183	5	0.4460	5	0.7493	5	0.9488	5	0.5942
	15	4	15.212	5	21.666	5	7.4377	5	16.135	5	7.1219
	20	1	537,43	3	327.49	3	420.92	3	210.61	4	7.7279
queen7_7 (49N y 952A)	5	5	0.0319	5	0.0885	5	0.1319	5	0.1605	5	0.2008
	10	5	2.5356	5	2.2884	5	0.7994	5	1.0215	5	1.2021
	15	4	246.95	4	297.52	5	272.08	5	63.373	5	4.6372
	20	0	-	0	-	0	-	0	-	1	356.99
queen8_12 (96N y 2736A)	5	5	0.0751	5	0.2324	5	0.3239	5	0.3556	5	0.4161
	10	5	180,91	5	68.544	5	1.6550	5	2.0988	5	2.5147
	15	5	54.671	5	23.749	5	8.8454	5	6.6523	5	9.5259
	20	0	-	0	-	0	-	5	39.444	5	19.743
queen8_8 (64N y 1456A)	5	5	0.0435	5	0.1125	5	0.1919	5	0.2120	5	0.3310
	10	5	10.855	5	8.3786	5	1.1150	5	1.1162	5	2.2282
	15	3	102.84	3	235.69	3	72.624	3	445.39	3	141.44
	20	0	-	0	-	0	-	5	346.38	4	82.400
queen9_9 (81N y 2112A)	5	5	0.0601	5	0.1930	5	0.2950	5	0.3511	5	0.4547
	10	5	34.136	5	24.543	5	1.8306	5	2.5212	5	3.9248
	15	5	25.604	5	23.408	5	10.268	5	12.716	5	17.142
	20	0	-	0	-	0	-	2	268.90	5	28.642

Table 6: Tabla de resultados para instancias aleatorias con 50 nodos.

Luego de analizar los datos descriptos anteriormente en las tablas de valores, notamos algunos resultados que nos gustarían resaltar para que se puedan apreciar con mayor detalle.

Uno de los resultados mas llamativos que hemos visto lo obtuvimos a partir de la tabla 6. En ella podemos observar que las mejoras introducidas por los planos de corte son realmente muy notables. En especial para las instancias que surgen a partir de *queen5_5* con 15 particiones. El gráfico 4 muestra el comportamiento de agregar planos de corte a dichas instancias. Podemos ver claramente que a medida que se agregan los planos de corte la performance mejora notablemente, a tal punto que logramos reducir los tiempos al 3% al realizar 19 iteraciones de planos de corte. Incluso para la misma instancia si observamos lo que sucede al utilizar 20 particiones, podemos ver que si no realizamos ninguna iteración de planos de corte, el algoritmo no logra terminar antes de los 10 minutos para ninguna instancia. Al realizar las 19 iteraciones podemos ver que todas las instancias terminan y que en promedio demoran apenas 1.32 segundos.

En general para las instancias aleatorias relativamente grandes los resultados obtenidos son muy buenos. Esto puede observarse en el gráfico 5. En este gráfico se puede observar que en casi todos los casos los tiempos de procesamiento se reducen bastante a medida que se van a gregando iteraciones de planos de corte. Las instancias que están graficadas son las siguientes: 30 nodos con el 25% de aristas y 20 particiones, 30 nodos con el 75% de aristas y 10 particiones, 40 nodos con el 25% de aristas y 20 particiones y 40 nodos con 75% de aristas y 10 particiones. Los resultados fueron extraídos de la tabla 2 y la tabla 3.

En el gráfico 5 podemos ver que para algunas instancias, en algún momento aumenta el tiempo de procesamiento al agregar planos de corte pero luego al seguir agregando mas, el tiempo de ejecución se reduce bastante. Esto nos hace pensar que quizá si aumentábamos aún mas la cantidad de iteraciones podríamos obtener mejores resultados. Esto no lo realizamos ya que decidimos ponerle una cota a la cantidad de pruebas porque sino no terminaríamos nunca ya que hay infinidad de casos para probar.

También podemos observar en el gráfico 6 que para las instancias generadas de manera aleatoria mas pequeñas (las que se ejecutan mas rápido), las mejoras no son demasiado significativas. Hay casos en los que la mejora es mínima y otros casos en los que en realidad no hay mejora. Esto puede ser debido a que como el algoritmo demora tan poco tiempo en correr, la mejora puede ser que se pierda con el overhead de agregar los planos de corte. Las instancias que están graficadas son las siguientes: 40 nodos con el 10% de aristas y 10 particiones, 40 nodos con el 10% de aristas y 15 particiones, 40 nodos con el 25% de aristas y 15 particiones, 50 nodos con 25% de aristas y 5 particiones Y 50 nodos con 10% de aristas y 20 particiones. Los resultados fueron extraídos de la tabla 2 y la tabla 3.

La realidad es que así como estos resultados que acabamos de mostrar, podríamos mostrar un montón de otros resultados interesantes ya que la cantidad de pruebas que realizamos es realmente grande. Consideramos que todos esos resultados pueden observarse en las tablas de valores que mostramos al comienzo de la sección. Dichas tablas tienen todos los valores reales provenientes

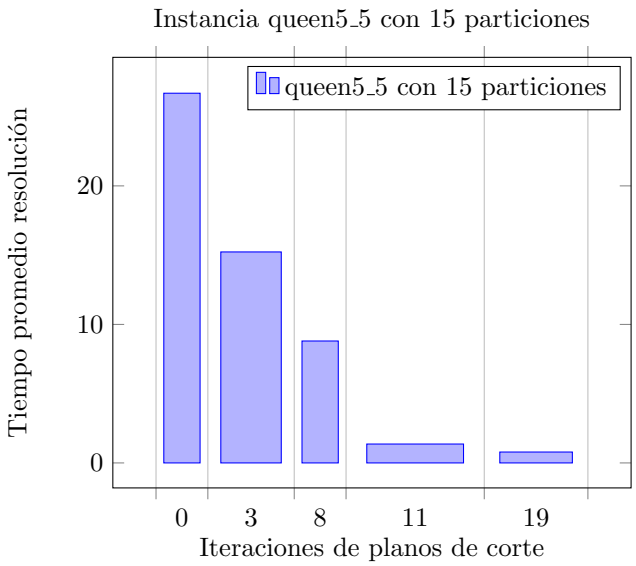

Figure 4: Instancia queen5_5 con 15 particiones

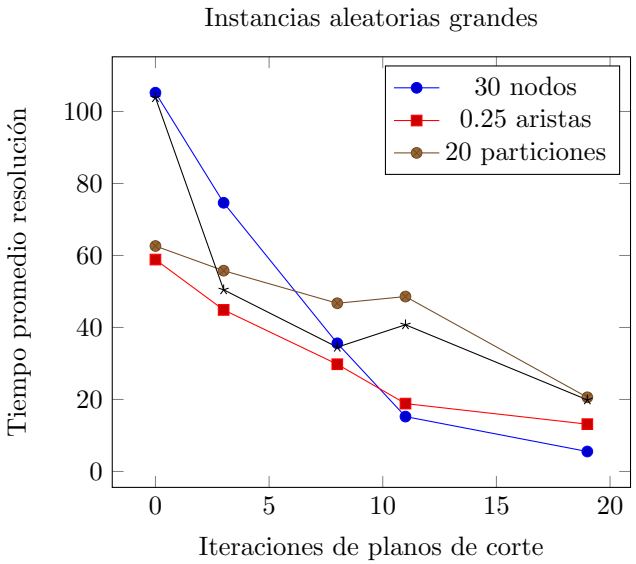

Figure 5: Instancia queen5_5 con 15 particiones

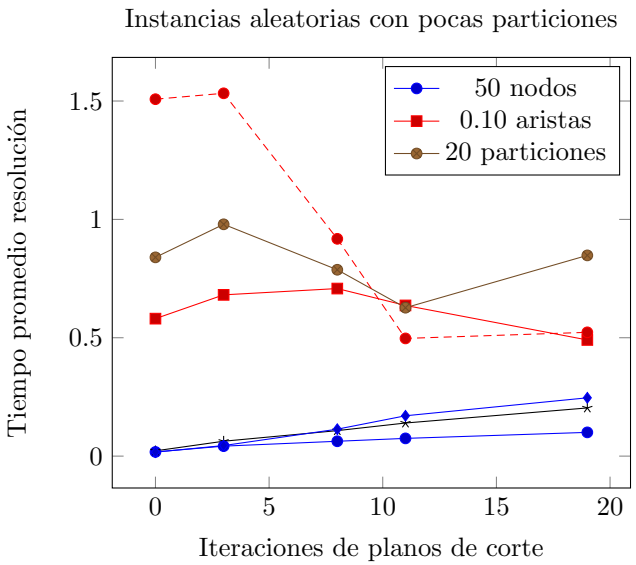

Figure 6: Instancias aleatorias con pocas particiones

de la eperimentación. Los otros gráficos particulares que hicimos para mostrar algunos resultados concretos, son simplemente extractos de la información que brindan las tablas en los cuales queríamos hacer incapié y no queríamos que pasen desapercibidos. Esto quiere decir que la información mas rica se encuentra en las tablas ya que contienen la totalidad de las mediciones.

5 Conclusiones

Luego de haber realizado una gran cantidad de experimentos lo suficientemente variados, logramos realizar un análisis del comportamiento de los algormitmos implementados en el presente trabajo práccito.

Como se puede apreciar en la totalidad de las tablas de resultados, podemos observar que para instancias de grafos con una cantidad de particiones pequeña (5 o 10 particiones), el hecho de agregar iteraciones de planos de corte no genera un aumento considerable en la performance. En la mayoría de los casos a medida que agregamos mas iteraciones de planos de corte, en un principio la performance del algoritmo comienza a deteriorarse y luego una vez que pasamos las 11 o 19 iteraciones de planos de corte logramos obtener una mejora con respecto a no haber realizado ninguna. Esto es debido a que al ser instancias tan pequeñas, el hecho de agregar los planos de corte termina penalizando la ejecución del algoritmo ya que si bien logramos una mejora a la hora de recorrer el árbol en la etapa del algoritmo branch and bound, como el tiempo que se demora es muy poco, el agregado de los planos de corte no resulta despreciable en comparación con la ejecución total del algoritmo. Es por ello que podemos concluir que para instancias pequeñas, no es demasiado bueno agregar planos de corte.

En contraposición a lo anterior, para las instancias mas grandes, podemos ver que agregar planos de corte marca la diferencia. Hemos visto que los tiempos se reducen considerablemente a medida que van aumentando las iteraciones de planos de corte que vamos realizando. En algunos casos los tiempos de procesamiento se reducen a un 20% del total, como por ejemplo para instancias generadas aleatoriamente de 30 nodos con una densidad de aristas del 25% y 20 particiones. Pasamos de demorar poco mas de 100 segundos sin agregar planos de corte a demorar menos de 20 realizando 19 iteraciones. En otros casos las mejoras son mucho mas notables, hemos visto ejemplos donde sin planos de corte los algoritmos no terminaban al cabo del timeout (10 minutos) y al agregar los planos de corte no solo que terminaron, sino que lo han hecho en poco mas de 30 segundos. Un ejemplo de lo mencionado recientemente puede apreciarse en las pruebas realizadas sobre algormitmos de 40 nodos con 90% de densidad de aristas y apenas 10 particiones.

Ejecutando los algoritmos sobre las instancias de prueba de coloreo de grafos si bien notamos mejoras, las mismas no fueron de igual magnitud en todos los casos.

Para la familia de instancias *MyCiel*, al ser instancias relativamente pequeñas, las mejoras realizadas fueron mínimas en algunos casos y hemos logrado recudir hasta un 50% los tiempos en otros. Mientras que en la familia de instancias *Queen* las mejoras son realmente impresionantes. Un ejemplo notable de esto es en la instancia *queen5_5* que con 20 particiones. Al intentar correrla sin el agregado de planos de corte no logramos terminar antes de los 10 minutos en ningun caso. Luego de agregar los planos de corte hemos logrado resolver todos los casos con un tiempo promedio de 1 segundo y medio.

A la hora de buscar una optimización para cierto problema, nos parece que es fundamental realizar una experimentación lo suficientemente amplia y variada como para llegar a conclusiones con un cierto grado de respaldo empírico. Esto quiere decir que si la experimentación cuenta con algunos ejemplos no muy variados, no podemos estar seguros que la mejora realmente es significativa, sino que pudo haber sido casual para ciertos casos particulares. Es decir, que los resultados pueden llegar a estar sesgados. Por ejemplo, si solo nos hubieramos quedado con los resultados obtenidos al procesar las instancias de la familia *Myciel* no habríamos llegado a la misma conclusión que llegamos luego de probar con mas de 8000 instancias de prueba.

Experimentar sobre instancias generadas de forma aleatoria puede llegar a ser una buena muestra ya que no estamos asumiendo nada sobre las instancias a las que se les aplica el proceso. Si probamos sobre muchas instancias aleatorias y en promedio los resultados son los esperados, podemos estar lo suficientemente tranquilos de que lo que se está haciendo en principio si funciona ya que los resultados no estarían sesgados a un simple set de pruebas.

Por último, creemos que en la práctica el hecho de lograr una optimización a un problema de programación lineal entera, puede llegar a significar un cambio muy importante. Podría creerse hasta ese momento que el problema en cuestión no tenía solución en un tiempo aceptable, y a partir de un trabajo similar al que hicimos (dejando de lado las diferencias) podría concluirse que dicho problema, tendría solución, y a raíz de eso lograr un cambio a nivel tecnológico realmente muy grande, y en algunos casos, pasar de lo imposible a lo posible.