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

TP: Programación Lineal Entera

Investigación Operativa

Segundo Cuatrimestre de 2015

Alumno	LU	E-mail
Germán Romano	786/11	romano.german@live.com.ar
Leandro Vega	698/11	leandrogvega@gmail.com

Índice

1. Introducción

El problema de coloreo de grafos ha sido ampliamente estudiado y aparece en numerosas aplicaciones de la vida real, como por ejemplo en problemas de scheduling, asignación de frecuencias, secuenciamiento, etc. Formalmente, el problema puede ser definido de la siguiente forma: Dado un grafo $G = (V, E)$ con $n = |V|$ vértices y $m = |E|$ aristas, un coloreo de G consiste en una asignación de colores o etiquetas a cada vértice $p \in V$ de forma tal que todo par de vértices $(p, q) \in E$ poseen colores distintos. El problema de coloreo de grafos consiste en encontrar un coloreo que utilice la menor cantidad posible de colores distintos.

A partir de diferentes aplicaciones, surgieron variaciones o generalizaciones de este problema, como el problema de Coloreo Particionado de grafos. En este problema el conjunto V se encuentra dividido en una partición $V_1, ..., V_k$, y el objetivo es asignar un color a sólo un vértice de cada subconjunto de la partición, de manera tal que dos vértices adyacentes no reciban el mismo color, minimizando la cantidad de colores utilizados.

En este trabajo se presentan implementaciones de diversos algoritmos de Programación Lineal Entera para resolver el problema de Coloreo Particionado, haciendo uso del paquete *IBM ILOG CPLEX Optimization Studio*[1]. También, se compara el desempeño de dichos algoritmos aplicados a diversos tipos de instancias del problema.

Todas las mediciones presentadas fueron realizadas en un equipo con procesador Intel Pentium Dual de 1.86 GHz, con 2 Gb de RAM y sistema operativo Ubuntu 12.04 de 64-bit.

2. Modelado

Sea P el conjunto de índices de colores, E el conjunto de ejes del grafo, V su conjunto de vértices y V* una partición de V, modelamos el problema de Coloreo Particionado mediante Programación Lineal Entera de la siguiente manera:

Variables:

$$x_{ij} = \begin{cases} 1, & \text{si el color } j \text{ es asignado al vértice } i \\ 0, & \text{en caso contrario} \end{cases}$$

$$w_j = \begin{cases} 1, & \text{si } x_{ij} = 1 \text{ para algún vértice } i \\ 0, & \text{en caso contrario} \end{cases}$$

Minimizar: $\sum_{i=1}^{|P|} w_j$

Sujeto a:

(1) $x_{ij} + x_{kj} \leq 1 \qquad \forall (i,k) \in E, \forall j \in \{1, ..., |P|\}$

(2) $\sum_{j=1}^{|P|} x_{ij} \leq 1 \qquad \forall i \in V$

(3) $\sum_{j=1}^{|P|} \sum_{i \in V_l} x_{ij} = 1 \qquad \forall V_l \in V^* = \{V_1, ..., V_k\}$

(4) $n * w_j - \sum_{i=1}^{n} x_{ij} \geq 0 \qquad \forall j \in \{1, ..., |P|\}$

(5) $w_j \geq w_{j+1} \qquad \forall j \in \{1, ..., |P| - 1\}$

(6) $x_{ij} \in \{0, 1\} \qquad \forall i \in V, \forall j \in \{1, ..., |P|\}$

$$(7) \quad w_j \in \{0,1\} \qquad \forall j \in \{1,...,|P|\}$$

La función objetivo a minimizar equivale a la cantidad de colores utilizados. Cada restricción modela algún aspecto particular del problema:

(1): Dos nodos adyacentes no pueden ser pintados con el mismo color.

(2): Cada nodo se pinta con a lo sumo un color.

(3): Cada subconjunto de la partición V^* contiene exactamente un nodo pintado.

(4): La variable w_j vale 1 si y sólo si se asigna el color j a algún nodo.

(5): Los colores se utilizan "en orden". Esto limita considerablemente la cantidad de posibles coloreos y, por lo tanto, el tamaño del árbol de búsqueda.

(6) y (7): Las variables x_{ij} y w_j son binarias.

Las instancias de entrada fueron generadas de manera aleatoria. Dada una cantidad de nodos, un factor de densidad de las aristas del grafo y la cantidad de subconjuntos que posee la partición del conjunto de nodos, se generaban números aleatorios que definían a qué subconjunto de la partición pertenecía cada nodo y si dos nodos iban a ser adyacentes o no. Para facilitar el chequeo de correctitud de las soluciones obtenidas, decidimos implementar los grafos en formato *DOT*, que permite graficar fácilmente las instancias.

3. Branch and Bound

El algoritmo Branch and Bound, para la resolución de un problema de Programación Lineal Entera, procede de la siguiente manera:

1. Resolver la relajación lineal del LP.
2. Si todas las variables resultan enteras, fin.
3. Si no, se toma una de las variables fraccionarias x_i con valor x_i^* y se generan dos sub-LP (ambos copias del LP anterior) con una nueva restricción cada uno: a uno se le agrega la restricción $x_i \leq \lfloor x_i^* \rfloor$, y al otro $x_i \geq \lceil x_i^* \rceil$
4. Aplicar Branch and Bound a los dos sub-LP.

Para mejorar el tiempo de cómputo del algoritmo se aplican diferentes tipos de podas. Por ejemplo, si la relajación lineal de un sub-LP da un resultado peor al de la mejor solución entera hallada, esa rama se elimina y se prosigue con las demás.

Para implementar dicho algoritmo, se configuró el paquete CPLEX de manera que omita la etapa de preprocesamiento y aplique el algoritmo Branch and Cut, pero sin adicionar planos de corte.

4. Planos de Corte

El algoritmo de Planos de Corte consiste en agregar nuevas restricciones (o planos de corte) al LP a partir de desigualdades válidas para el problema en cuestión. El ciclo consiste en resolver la relajación lineal del LP y, mediante los llamados algoritmos de separación, agregar aquellas restricciones válidas para el problema en su versión entera pero violadas por la solución de la relajación, de manera de achicar la región factible de la relajación y acercarla lo más posible a la cápsula convexa del problema entero.

Si bien este algoritmo, muy probablemente, no devuelva una solución entera, permite obtener una solución "más entera" que aquella que pudiese brindarnos el método Simplex por sí solo, por ejemplo. Además, el algoritmo de Planos de Corte puede combinarse con la estrategia Branch and Bound para mejorar los tiempos de ejecución de este último algoritmo, como veremos en la próxima sección.

En nuestra implementación, la cantidad de veces que se ejecutará el ciclo de búsqueda de planos de corte depende del tope máximo de iteraciones que se configure, y de la cantidad de restricciones violadas halladas en el último ciclo. Esto es, se itera hasta que: o bien se superó el límite de iteraciones, o bien en el último ciclo no se hallaron restricciones violadas. Esto último se decidió luego de observar que al no hallarse restricciones violadas en una iteración, la probabilidad de que en futuros ciclos sí se hallen es muy baja.

Nuestro algoritmo de Planos de Corte integra dos familias de desigualdades válidas: Desigualdades Clique y Desigualdades *Odd-hole*.

4.1. Desigualdades Clique

Familia 1: *Sea $j_0 \in \{1, ..., n\}$ y sea K una clique maximal de G. La desigualdad clique está definida por*

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

4.1.1. Demostración de validez

- Por definición, una clique es un subgrafo completo.
- Por definición de coloreo sabemos que dos nodos adyacentes tienen diferente color.

Entonces, sea K una clique maximal de G, se cumple que: $\forall p_1, p_2 \in K$, $color(p_1) \neq color(p_2)$. Esto se debe a que todos los nodos son adyacentes entre sí en una clique.

- Sea j_0 un color, supongamos que $\exists p_1 \in K$ tal que $color(p_1) = j_0$, o sea $x_{p_1 j_0} = 1$. Entonces $\forall p_2 \in K$ tal que $p_2 \neq p$, $x_{p_2 j_0} = 0$. Por lo tanto $\sum_{p \in K} x_{pj_0} = 1$.
  Observando la variable w_{j_0}, sabemos que existe un nodo pintado del color j_0 (nodo p). Por lo tanto $w_{j_0} = 1$.
  Conclusión: $\sum_{p \in K} x_{pj_0} = w_{j_0}$.
- Sea j_0 un color, supongamos que $\nexists p_1 \in K$ tal que $color(p_1) = j_0$. Entonces $\sum_{p \in K} x_{pj_0} = 0$.
  - Además, supongamos que $\exists q \in V(G)$ tal que $color(q) = j_0$. Como el color j_0 es usado en un nodo, $w_{j_0} = 1$.
    Conclusión: $\sum_{p \in K} x_{pj_0} \leq w_{j_0}$.
  - Por último, supongamos que $\nexists q \in V(G)$ tal que $color(q) = j_0$. En este caso, como no hay nodo pintado por j_0, $w_{j_0} = 0$.
    Conclusión: $\sum_{p \in K} x_{pj_0} = w_{j_0}$.

Al haber visto y observado todos los casos, hemos demostrado que efectivamente la desigualdad $\sum_{p \in K} x_{pj_0} \leq w_{j_0}$ se cumple. □

4.1.2. Implementación de heurística de separación

El algoritmo de separación para restricciones de la familia 1 hace uso de una heurística[2] para descomponer al grafo en cliques, con algunas modificaciones para adaptarlo a nuestro problema. La heurística original se muestra en la siguiente figura, donde $\Gamma(v_j, c)$ representa la cantidad de nodos adyacentes al nodo v_j pertenecientes a la clique c, y $sizes(c)$ la cantidad de nodos de la clique c.

Algorithm 1: Greedy Clique Covering (GCC)

Input: graph $G = [V, E]$
permutation $P = [P_1, P_2, ..., P_n]$ of vertices in V
Output: clique covering S of G

```
1  for c = 1..n
2      sizes(c) = 0
3  for i = 1..n
4      j = P_i
5      c = find_equal(Γ(v_j, c), sizes(c))
6      V_c = V_c ∪ {v_j}
7  return S = {V_1, V_2, ..., V_k}
```

Dado que, a la hora de armar las cliques, contamos con la solución de la relajación lineal del LP, se puede aprovechar esta información para hallar la mayor cantidad posible de restricciones violadas por iteración, así como también para chequear la menor cantidad posible de restricciones. Para esto, se tomaron algunas medidas a la hora de armar las cliques y chequear si se cumplen o no las restricciones de la familia 1:

1. Asignar cada nodo pintado a una clique distinta. Dado que la sumatoria de la desigualdad puede valer a lo sumo 1, un nodo pintado dentro de la clique suele bastar para violarla.

2. Ignorar aquellas cliques que no contengan algún nodo pintado, puesto que la sumatoria de las desigualdades correspondientes a dichas cliques vale 0 y, por lo tanto, estas desigualdades nunca se violan.

3. No chequear las restricciones correspondientes a colores que no fueron utilizados en la última solución. Esto evita chequear aquellas restricciones con lado derecho e izquierdo nulo.

4.2. Desigualdades *Odd-hole*

Familia 2: *Sea* $j_0 \in \{1, ..., n\}$ *y* $C_{2k+1} = v_1, ..., v_{2k+1}, k \geq 2$ *un agujero de longitud impar. La desigualdad odd-hole está definida por*

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

4.2.1. Demostración de validez

- Por definición, un ciclo es un camino cerrado en el que no se repite ningún vértice a excepción del primero que aparece dos veces como principio y fin del camino.
- Por definición de coloreo sabemos que dos nodos adyacentes tienen diferente color.
- Conjunto independiente: Dado un grafo G, un subconjunto de vértices $H \subset V(G)$ se llama conjunto independiente si no hay dos vértices en H que sean adyacentes.
- Un ciclo de longitud impar puede pintar del mismo color, como máximo, $(longitudCiclo - 1)/2$ nodos. Esto se debe a que al armarnos un conjunto independiente máximo del ciclo, vamos a tener ese tamaño. Vamos a comprobar esto, para eso tengo que tratar de meter la mayor cantidad de nodos que no tengan aristas adyacentes:
  - Tenemos un ciclo que va desde el nodo 1 hasta el nodo *longCiclo*, siendo *longCiclo* la longitud del ciclo. Llamemos a la longitud del ciclo como $2k + 1$, dado que el tamaño es impar.
  - Selecciono el nodo 1. Es lo mismo iniciar con 1 o con cualquier número, dado que es un ciclo y la enumeración la podes hacer como más te convenga.
  - Luego, el nodo 3 puede integrar también mi conjunto independiente, dado que sólo *longCiclo* y 2 eran adyacentes al nodo 1 inicial.

- Continúo saltando cada dos nodos hasta el final. El final será cuando el siguiente nodo al que deba saltar es un nodo ya agregado en el conjunto ó sea adyacente a uno ya agregado.
- Como salto de dos en dos, voy a llegar hasta el nodo $1 + 2p$, siendo p la cantidad de saltos dados. Si p llegara a ser k, entonces habremos llegado al último nodo posible y su sucesor será el primero, o sea 1. Este nodo, $1+2k$ será imposible agregarlo porque rompería la definición de conjunto independiente, entonces tendremos que parar en el salto anterior, o sea, $1+2(k-1)$. Esto quiere decir que hemos agregado, en nuestro conjunto independiente, k nodos (uno por el inicial y $k-1$ de los saltos realizados).
- Si despejamos $2k + 1 = longitudCiclo$, entonces obtendremos $k = (longitudCiclo - 1)/2$. Demostrando efectivamente lo mencionado en el inciso.

Sean: j_0 un color, $C_{2k+1} = p_1, p_2, ..., p_{2k+1}$ un ciclo de longitud impar, $k \geq 2$. Supongamos que $\sum_{p \in C_{2k+1}} x_{pj_0} > kw_{j_0}$.

- Si $\nexists q \in C_{2k+1}$ tal que $color(q) = j_0$, podemos concluir que $\sum_{p \in C_{2k+1}} x_{pj_0} = 0$. Esto es absurdo, dado que $w_{j_0} \geq 0$ por definición y $k \geq 2$. Por lo tanto $kw_{j_0} \geq 0$ y contradice a nuestra suposición.
- Si $\exists q \in C_{2k+1}$ tal que $color(q) = j_0$, entonces $x_{qj_0} = 1$ y $w_{j_0} = 1$. Esto nos dice que $\sum_{p \in C_{2k+1}} x_{pj_0} > k$. Con lo cual, para comprobar dicha afirmación, necesitamos tener pintados con color j_0 una cantidad de $k+1$ nodos dentro del ciclo. Esto es absurdo, dado que como mucho, podemos pintar del mismo color $(longitudCiclo - 1)/2 \iff (2k + 1 - 1)/2 \iff 2k/2 \iff k$, y $k < k + 1$. □

4.2.2. Implementación de heurística de separación

El algoritmo de separación para restricciones de la familia 2 hace uso de una heurística golosa para encontrar ciclos impares en el grafo. De la misma forma que en la familia 1, en esta familia también hacemos modificaciones para adaptarlo a nuestro problema.

La heurística original consiste en:

- Tomar un nodo dentro del grafo.
- Ir recorriendo un camino de nodos, pasando de uno a otro por el vecino que tenga mayor cantidad de vecinos.
- Agregar cada nodo que visitamos al arreglo del ciclo que vamos a devolver.
- Setear cada nodo del ciclo como 'visto', para evitar insertar nuevamente dichos nodos.
- Finalizar cuando un nodo, dentro del conjunto de sus vecinos, tenga al nodo inicial.
- Si no puede seguir avanzando por ningún camino, esto significa que todos sus vecinos ya fueron 'vistos' y no tiene entre sus vecinos al nodo inicial, y entonces se informará que no se ha encontrado ciclo.
- Esto se repite para todos los nodos del grafo.

Nuestra modificación consiste en:

- Encontrar un $w_j \geq 0$, y luego conseguirnos un k tal que $kw_j \leq 1 \wedge k \geq 2$. De esta manera, si luego comenzamos a armar un ciclo con un nodo p tal que $x_{pj} \geq 0$ (que va a existir, ya que sino $w_j = 0$), seguramente tengamos altas chances de romper dicha desigualdad, que es el objetivo de nuestro plano de corte.
- Con p, ya tenemos el nodo inicial con el cual comenzar la heurística original.

- Luego marcamos límite mínimo y máximo del ciclo. El límite mínimo es $2 * k + 1$ cuando k es mínimo, el k mínimo es 2, por lo tanto el limite mínimo será 5. El máximo estará representado por $2 * k + 1$, siendo k el valor conseguido anteriormente. Además debemos corroborar que el tamaño del ciclo sea impar.
- Esto lo repetiremos para cada color j tal que $w_j \geq 0$.

Pseudocódigo de la heurística implementada:

En este pseudocódigo, las siguiente variables están definidas globalmente: Colores, w_j, x_{pj}.

```
1: function encuentra_ciclos_impares(G(V, E))
2:     vector < vector < int >> ciclos_impares
3:     for j ∈ Colores do
4:         if w_j > 0 then
5:             Conseguir k tal que k * w_j < 1
6:             if k >= 2 then
7:                 Conseguir p tal que x_pj > 0 ∧ p ∈ V(G)
8:                 vector < Nodo > ciclo
9:                 encontro_ciclo_impar = true
10:                vector < bool > visto(cantidadNodos, false) // Ningun nodo ha sido visto
11:                buscar_ciclo(p, p, G(V, E), 5, 2 * k + 1, ciclo, encontro_ciclo_impar, visto)
12:                if encontro_ciclo_impar then
13:                    ciclos_impares.push_back(ciclo)
14:                end if
15:            end if
16:        end if
17:    end for
18:    return ciclos_impares
19: end function
```

```
1: function buscar_ciclo(inicia, actual, G(V, E), limite_bajo, limite_alto, ciclo, encontro_ciclo, visto)
2:     visto[actual] = true
3:     ciclo.push_back(actual)
4:     if limite_bajo ≤ 0 ∧ (actual, inicial) ∈ E(G) ∧ esImpar(|ciclo|) then
5:         Termine
6:     else if limite_alto = 0 then
7:         encontro_ciclo = false
8:     else
9:         nuevo_actual = vecino_con_mas_vecinos(actual, G(V, E), visto)
10:        if nuevo_actual ≥ 0 then
11:            //A este If entramos si encontro un vecino al cual pasar
12:            buscar_ciclo(inicial, nuevo_actual, G(V, E), limite_bajo−1, limite_alto−1, ciclo, encuentro_ciclo, visto)
13:        else
14:            encontro_ciclo = false
15:        end if
16:    end if
17: end function
```

La función *vecino_con_mas_vecinos* busca el vecino del nodo *actual* que tenga mayor cantidad de vecinos, pero que no haya sido *visto* durante el recorrido de la búsqueda del ciclo.

4.3. Resultados

A continuación, presentamos las mejoras obtenidas en la relajación lineal del LP sobre diferentes tipos de instancias con el algoritmo de Planos de Corte presentado, estableciendo un tope máximo de 50 iteraciones de los algoritmos de separación. Los datos representan la mejora promedio entre los resultados de tres instancias distintas por cada combinación posible de cantidad de subconjuntos de la partición (k) y densidad (d). Con mejora nos referimos al incremento del valor óptimo de la función objetivo al aplicar los planos de corte hallados.

Todas las instancias de evaluación constan de 50 nodos y tres niveles de densidad y tamaño de la partición diferentes (bajo, medio y alto).

		Cantidad máxima de iteraciones		
k	d	3 iteraciones	10 iteraciones	50 iteraciones
5	0.3	20 %	83.59 %	82.6 %
	0.6	52.61 %	111.46 %	213.69 %
	0.8	14.66 %	126.81 %	192.01 %
10	0.3	31.16 %	100.96 %	102.11 %
	0.6	50 %	192.56 %	211.72 %
	0.8	0 %	72.72 %	74.82 %
15	0.3	161,9 %	233.33 %	233.33 %
	0.6	99.9 %	233.33 %	233.33 %
	0.8	77.7 %	177.82 %	193.44 %

Cuadro 1: Mejoras obtenidas en la relajación lineal, según el tope de iteraciones establecido y el tipo de instancia.

En la figura 1 pueden apreciarse mejor las diferencias al configurar diferentes límites de iteración al algoritmo, agrupando las variantes de partición del grafo.

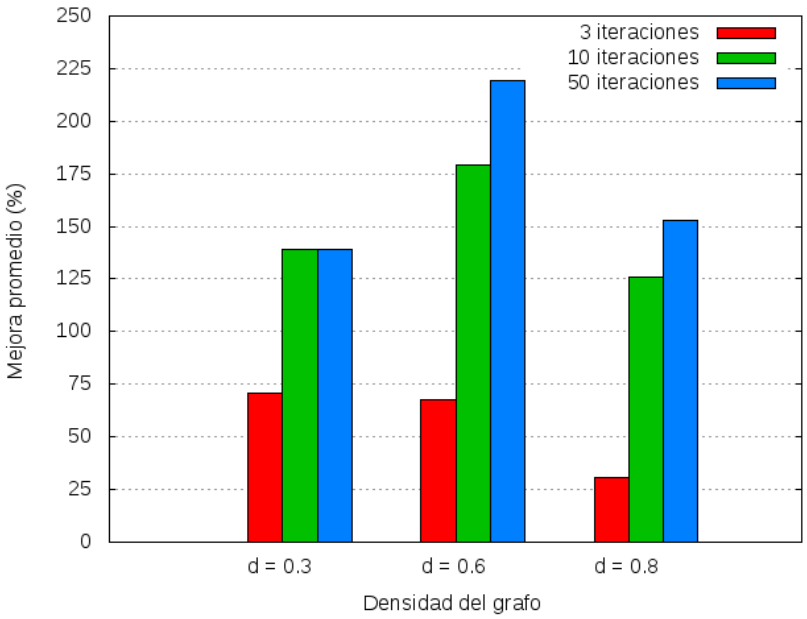

Figura 1: Promedio de las mejoras observadas en base a la densidad del grafo.

Puede observarse que cuanto mayor es la cantidad de subconjuntos de la partición (k), mayores son los porcentajes de mejora. Una explicación que encontramos a esto, es que posiblemente se deba a la

forma de armar nuestras heurísticas de separación. En ellas, se buscan ciclos y cliques a partir de nodos pintados. Al aumentar la cantidad de particiones, muy posiblemente aumentará la cantidad de colores distintos utilizados en el coloreo y, por lo tanto, es factible que se hallen más cliques y ciclos que achiquen la región factible del LP.

También puede notarse que la mejora máxima suele hallarse cuando el grafo tiene una densidad de nivel medio ($d = 0{,}6$).

Puede estimarse que cuando $d = 0{,}3$ (baja cantidad de aristas), va a haber pocos colores usados, debido a que posiblemente podremos pintar muchas particiones con un mismo color. Debido a esto, la cantidad de restricciones nuevas (de ambas familias) va a ser menor, y no podremos aspirar a tener una buena mejora.

Como era de esperarse, las mejoras se incrementan a medida que se eleva el tope máximo de iteraciones. Sin embargo, no se observa una mejora sustancial al pasar de 10 a 50 iteraciones. Evaluando el *trade-off* tiempo de cómputo vs. mejora observada, en algunos casos podría resultar suficiente setear un tope máximo de 10 iteraciones; mientras que en otros casos, donde se aprecie más la mejora en la solución de la relajación (grafos muy grandes, por ejemplo), convendría setear un límite alrededor de las 50 iteraciones, permitiendo al algoritmo hallar la mayor cantidad de planos de corte posibles.

Cabe destacar que hemos observado, en la mayor parte de los casos evaluados, que nunca se llegaron a ejecutar las 50 iteraciones, sino que se dejaron de hallar planos de corte en una cantidad menor de ejecuciones de los algoritmos de separación.

5. Cut and Branch

El algoritmo Cut and Branch aprovecha las ventajas de la estrategia de resolución Branch and Bound y de los algoritmos de Planos de Corte. El mismo actúa de la siguiente manera:

1. Resolver la relajación lineal del LP.
2. Buscar desigualdades válidas violadas mediante el algoritmo de separación. Si existen, agregar dichas restricciones al LP. Si no, ir al paso 4.
3. Resolver la relajación lineal del nuevo LP y volver al paso 2.
4. Aplicar Branch and Bound al LP resultante.

La implementación de este algoritmo consistió en agregarle a nuestro algoritmo de planos de corte, presentado en la sección anterior, la resolución del LP resultante mediante la estrategia Branch and Bound.

6. Comparación de algoritmos

A continuación, presentamos los tiempos de respuesta y la cantidad de nodos recorridos de los algoritmos Branch and Bound y Cut and Branch, sobre instancias de diferentes características y variando estrategias de recorrido y selección de variables. Todas las instancias de entrada son grafos de 50 nodos, con diferentes densidades (d) y cantidades de subconjuntos de la partición (k).

Se crearon tres instancias distintas de cada tipo (combinando densidades y tamaño de partición en niveles: alto, medio y bajo), y se promediaron los tiempos de ejecución y cantidad de nodos recorridos en cada una de ellas.

En el caso del algoritmo Cut and Branch, se tomo un límite de 50 iteraciones, para permitir que se encuentren todas los planos de corte posibles y observar su impacto en los tiempos de ejecución.

Se probaron dos alternativas de recorrido del árbol: *Best-Bound* (estrategia por default de *CPLEX*) y *Best-Estimate*, y dos estrategias de selección de variable de *branching*: *Auto* (estrategia por default, donde *CPLEX* toma la decisión de qué estrategia es más conveniente) y *Strong-Branching*.

Para cada par de estrategias utilizadas en ambos algoritmos, se muestra sobre la izquierda la cantidad promedio de nodos recorridos del árbol y el tiempo promedio de ejecución (en segundos).

		Branch and Bound							
k	d	BBound-Auto		BBound-Strong		BEstimate-Auto		BEstimate-Strong	
5	0.3	0	0.11 s	0	0.03 s	0	0.03 s	0	0.03 s
	0.6	0	0.06 s	0	0.04 s	0	0.04 s	0	0.05 s
	0.8	1.66	0.17 s	1.66	0.19 s	1.66	0,16 s	3.33	0.29 s
10	0.3	0	0.07 s	0	0.07 s	0	0.07 s	0	0.07 s
	0.6	73.3	1.44 s	16	1.55 s	50.6	1.32 s	9.66	1.19 s
	0.8	356	6.94 s	154	18,97 s	359.3	6.83 s	193	22 s
15	0.3	0	0.16 s	0	0.16 s	0	0.14 s	0	0.16 s
	0.6	2017	22.64 s	708.3	80.11 s	1789	20.15 s	1902	72.5 s
	0.8	1508	378 s	13741	361.6 s	13741	373.1 s	5060	1062.7 s

		Cut and Branch (máx. 50 iteraciones)							
k	d	BBound-Auto		BBound-Strong		BEstimate-Auto		BEstimate-Strong	
5	0.3	0	0.28 s	0	0.29 s	0	0.27 s	0	0.29 s
	0.6	0	0.53 s	0	0.52 s	0	0.54 s	0	0.62 s
	0.8	2.66	0.81 s	5	1.09 s	8.3	1.12 s	3.3	0.97 s
10	0.3	2	1.86 s	3.3	1.63 s	0	1.66 s	1	1.45 s
	0.6	232	8.87 s	21	8.46 s	35	6.2 s	19	6.41 s
	0.8	237	18.42 s	192	51.24 s	492	23.47 s	214	58.4 s
15	0.3	0	1.89 s	0	1.98 s	0	1.53 s	0	1.9 s
	0.6	1856	101.7 s	1712	377.2 s	1761	77.95 s	2894	758 s
	0.8	2035	168.2 s	3169	994.5 s	6299	563.51 s	2193	650.5 s

Cuadro 2: Resultados obtenidos por diversas combinaciones de parámetros de los algoritmos, sobre diferentes tipos de instancias.

De los resultados presentados en la tabla anterior, pueden sacarse algunas conclusiones:

- Al contrario de lo que esperábamos, los tiempos de ejecución del algoritmo Cut and Branch son mayores a los de Branch and Bound. Esto puede deberse a diversos motivos. Entre ellos, a que las restricciones agregadas en los ciclos de corte pueden volver más "pesado" al LP resultante (en términos de memoria, por ejemplo), lo que enlentece la etapa de Branch and Bound. Una manera de comprobar esto sería experimentar con la cantidad de restricciones agregadas en la primer etapa y evaluar los tiempos de ejecución obtenidos.

  Por otro lado, la etapa de Planos de Corte demanda un tiempo extra, si bien representa una pequeña fracción del tiempo total demorado. Aparentemente, este tiempo extra no es compensado por las mejoras obtenidas en la relajación al finalizar dicha etapa.

- Los tiempos de cómputo se incrementan a medida que aumenta la densidad del grafo. Esto podría deberse a que, cuanto más aristas posee el grafo, mayor es la cantidad de restricciones que tiene el LP.

- Lo mismo sucede con la cantidad de subconjuntos de la partición. Al aumentar dicho número, crece considerablemente la cantidad de variables del LP, así como también aumenta la cantidad de restricciones y de colores (y con esto, la cantidad de coloreos posibles).

- En los casos más "pesados" ($k = 15$, $d = 0{,}8$), Cut and Branch suele recorrer una menor cantidad de nodos. Podría ocurrir que las restricciones agregadas en los ciclos de Planos de Corte surtan efecto (en términos de tamaño del árbol, vs. Branch and Bound) recién a partir de niveles altos de densidad y tamaño de la partición.

En cuanto a las diferentes estrategias evaluadas:

- La estrategia que más se lució fue la de setear los default de recorrido y selección (*Best-Bound* y *Auto*), porque otorgó los mejores tiempos sobre la muestra que experimentamos, sacando una gran ventaja con los demás. Esta ventaja se puede apreciar principalmente en el Cut and Branch con el grafo de 15 particiones y 0.8 de densidad.
- También pudimos observar que seteando la estrategia de *Strong-Branching* junto a *Best-Bound*, la cantidad de nodos recorridos es menor con respecto a las demás estrategias. Sin embargo, esto no fue suficiente para ganar en tiempo con las demás.
- La estrategia *Best-Estimate* y *Auto*, en varios casos, llega a mejorar los tiempos de *Best-Bound* y *Auto*, aunque en general es por poco margen, y recorre mucho más nodos que la estrategia *Strong-Branching* y *Best-Bound*.
- La estrategia que se vio más desfavorecida fue la combinación de *Best-Estimate* y *Strong-Branching*. Los tiempos de demoras fueron excesivamente altos para los grafos más grandes, aunque por otra parte, a comparación de sus tiempos, el recorrido de nodos que utilizó no fueron muy altos. Incluso, en algunos casos, el recorrido de nodos ha otorgado una cantidad semejante al de *Best-Bound* y *Auto*, aunque este le haya ganado en tiempo.

Referencias

[1] `http://www-01.ibm.com/software/commerce/optimization/cplex-optimizer/`

[2] David Chalupa. Partitioning Networks into Cliques: A Randomized Heuristic Approach. Slovak University of Technology in Bratislava, 2014.