DEPARTAMENTO
DE COMPUTACION
Facultad de Ciencias Exactas y Naturales - UBA

Trabajo Práctico nº 3

Dibujo incremental de grafos bipartitos

Algoritmos y Estructuras de Datos III (2008)

Grupo 3

Integrante	LU	Correo electrónico
González, Emiliano	426/06	xjesse_jamesx@hotmail.com
Martínez, Federico	17/06	federicoemartinez@gmail.com
Sainz-Trápaga, Gonzalo	454/06	gonzalo@sainztrapaga.com.ar

En el siguiente trabajo se explora el problema NP-completo de dibujo de grafos bipartitos (en su variante tradicional e incremental). Se proponen algoritmos eficientes para las problemáticas adyacentes del conteo de cruces, así como un algoritmo exacto basado en la técnica de *backtracking* y uno heurístico basado en GRASP. Se estudian sus tiempos de ejecución y la calidad de los resultados obtenidos comparado con otras heurísticas y con los resultados exactos.

Facultad de Ciencias Exactas y Naturales
Universidad de Buenos Aires
Ciudad Universitaria - (Pabellón I/Planta Baja)
Intendente Güiraldes 2160 - C1428EGA
Ciudad Autónoma de Buenos Aires - Rep. Argentina
Tel/Fax: (54 11) 4576-3359
http://www.fcen.uba.ar

Índice general

Parte 1

Consideraciones preliminares

1.1. Aplicaciones prácticas

El problema de minimización de cruces en dibujo de grafos bipartitos tiene varias aplicaciones prácticas en campos diversos. La aplicación más típica es sin duda la representación de relaciones bipartitas en esquemas o diagramas para ser leídos por seres humanos. La presencia de cruces en los ejes que unen a las partes del diagrama resulta confusa para la lectura: en este contexto resulta conveniente minimizar la cantidad de cruces en el dibujo para hacer más legible la información. El aspecto incremental del problema se aplica claramente a esta aplicación. Dado un diagrama existente, si se le agregan elementos sigue siendo deseable minimizar la cantidad de cruces en los ejes, pero en muchos casos puede ser más importante mantener el orden relativo de los elementos existentes del diagrama, ya que los usuarios que conocen un diagrama relativamente complejo recuerdan intuitivamente la posición de los elementos ya existentes. Si se invirtieran los órdenes originales para obtener menos cruces de ejes, el efecto indeseable del reordenamiento podría ser más confuso que la existencia de algunos cruces adicionales.

El problema de dibujo bipartito convencional (es decir, no incremental) tiene aplicaciones prácticas más diversas. En VLSI (*Very Large Scale Integration*) se combinan grandes cantidades de componentes electrónicos en un único microchip. La conexión entre estos componentes se hace con *buses* que pueden representarse como ejes de un grafo bipartito. El interés de ordenar las interfases de los componentes para minimizar cruces proviene de que, por un lado, dependiendo de la cantidad de capas del microchip puede o no ser viable que haya cruces entre los conductores y, por otra parte, toda superposición de conductores en un sistema electrónico de alta frecuencia produce interferencia electromagnética que puede afectar el rendimiento del dispositivo. A mayor escala, un problema similar existe en la diagramación de circuitos electrónicos sobre placas de computadoreas.

Finalmente, los grafos bipartitos sirven para modelar diversas estructuras de datos en problemas de *data mining*. Un ejemplo sencillo es la relación entre documentos y palabras clave que los referencian. La minimización de los cruces en un dibujo bipartito de un grafo de estas características puede servir como punto de partida para resolver un problema de particionamiento del conjunto de palabras clave (con el fin de agrupar de forma automatizada las palabras clave que refieren a una misma área de interés). Si los *clusters* de palabras claves se presentan al usuario, una vez más se observa el interés del problema incremental: es importante que al agregar nuevos documentos o palabras claves (nodos del grafo) el particionamiento existente se mantenga coherente.

1.2. Conteo de cruces

1.2.1. Distintos algoritmos

Dentro del problema a resolver, un tema esencial a explorar es el del conteo de cruces, ya que tanto la solución exacta como las heurísticas necesitan contabilizar con frecuencia los cruces que resultan de un dibujo dado para tomar decisiones (como por ejemplo para decidir cual es la posición óptima para insertar un nodo, o si una permutación es mejor o peor que otra). Por esta razón consideramos que un aspecto importante a optimizar para lograr mejorar el desempeño de nuestros algoritmos es precisamente el conteo de cruces.

Lo primero que se puede observar es que los cruces en el dibujo dependen solamente de la posición relativa de los nodos en las particiones. Se produce un cruce si hay dos nodos en una partición v_i, v_j que estén relacionados con un w_k y w_p respectivamente tal que v_i esta "*arriba*" de v_j y w_k esta "*abajo*" de w_p, como se ve en 1.1.

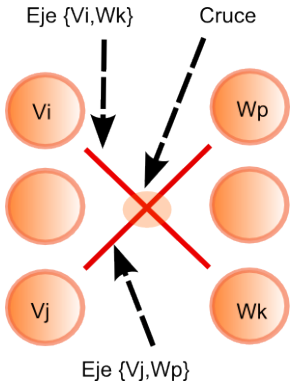


Figura 1.1: Esquema de un cruce

Podemos entonces caracterizar en qué casos se producen cruces:
Dado un orden de los nodos $p_1 =< v_1, v_2, ..., v_n >$ *en una partición y un orden* $p_2 =< w_1, w_2, ..., w_q >$ *en la otra partición, se produce un cruce si existen ejes* (v_i, w_j), (v_k, w_p) *tal que* $i < k \wedge p < j$.

De esta manera algoritmo sencillo para contar los cruces consiste en tomar cada par de ejes y comparar las posiciones relativas de sus nodos, lo cual puede ser computado con un costo temporal en $\theta(m^2)$, siendo m la cantidad de ejes del grafo. La cantidad de posibles pares de ejes es $\binom{m}{2} = \frac{m*(m-1)}{2}$. Este costo resulta grande en grafos densos (con muchos ejes), donde m puede ser del orden de $n_1 * n_2$, siendo n_i la cantidad de nodos en la partición i. Esta complejidad corresponde al modelo uniforme de complejidad, puesto que las operaciones en juego no involucran una dificultad aritmética importante, y por lo tanto podemos despreciar los costos de las operaciones matemáticas convencionales. El modelo uniforme es el elegido para usarse con este problema, y todas las complejidades examinadas en el trabajo corresponden a este modelo.

Veamos entonces como construir un algoritmo que nos permita mejorar el orden de complejidad del conteo de cruces en un dibujo. Introducimos para eso un criterio de orden entre los ejes del grafo: se ordenan los ejes del grafo primero por su posición en la partición p_1 y luego por su posición en p_2. Si tomamos dos ejes $e_i = (i_1, i_2)$ y $e_j = (j_1, j_2)$ con $i_1, j_1 < |p_1|$ y $i_2, j_2 < |p_2|$ las posiciones de los nodos que unen, consideramos que $e_i < e_j \Leftrightarrow i_1 < j_1 \vee (i_1 == j_1 \wedge i_2 < j_2)$. Asumimos por simplicidad que i_k es la posición en la partición 1, y j_k es la posición en la partición 2.

Si observamos luego el orden en el que quedan las segundas componentes de los ejes, podemos ver que cada inversión de orden es un cruce en el dibujo. Una inversión en el orden $\pi =< j_1, j_2, ...j_n >$ es un par (j_k, j_m) tal que $j_k > j_m \wedge k < m$. Esto vale porque como primero se ordena por la posición en la primer partición, y luego por la posición en la segunda, si hay una inversión es porque para los ejes (v_1, j_k) y (v_2, j_m) valía que $v_1 < v_2 \wedge j_k > j_m$, que es la condición para que se produzca un cruce.

Resulta de esto que si ordenamos los ejes de esa manera y después contamos las inversiones podemos obtener el número de cruces. El mecanismo utilizado para contar los ejes es *radix sort* a partir de la definición de los ejes como el par de posiciones que unen (y no como el par de nodos que unen, que es lo convencional). Esta definición de los ejes puede calcularse fácilmente recorriendo los dibujos y almacenando las posiciones de cada nodo en tiempo $O(n_1+n_2+m)$, siendo n_i la cantidad de nodos en la partición i y m la cantidad de ejes entre ellos. Una vez "traducidos" los ejes a esta notación, pueden ordenarse como vimos antes y contar la cantidad de inversiones en tiempo $O(m^2)$, lo cual redunda en la misma complejidad que teníamos previamente. Un procedimiento válido para contar las inversiones es utilizar *insertion sort* y sumar un cruce por cada *swap* realizado para ordenar la secuencia. Con este método, el costo final del conteo es $O(max(n_1, n_2, m) + c)$ con c la cantidad de *swaps* (o cruces) del dibujo.

Para ver un ejemplo de este algoritmo, consideremos el siguiente grafo bipartito:

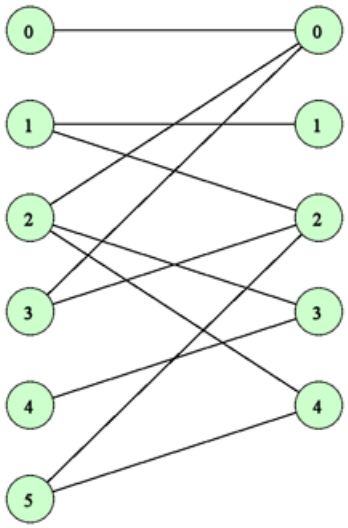


Figura 1.2: Grafo de ejemplo para la aplicación del segundo algoritmo de conteo de cruces

Ahora ordenamos los ejes de acuerdo a su primer componente y luego a su segunda, obteniendo:

$$\langle (0,0),(1,1),(1,2),(2,0),(2,3),(2,4),(3,0),(3,2),(4,3),(5,2),(5,4)\rangle$$

Entonces, las segundas componentes son:

$$\langle 0,1,2,0,3,4,0,2,3,2,4\rangle$$

Entonces, aplicamos *insertion sort*. El 0 (rojo) recorre dos posiciones antes de insertarse en su posición correcta.

$$\langle 0,0,1,2,3,4,0,2,3,2,4\rangle$$

Luego el tercer 0 se swapea cuatro veces:

$$\langle 0,0,0,1,2,3,4,2,3,2,4\rangle$$

El segundo 2 cambia dos veces de posición:

$$\langle 0,0,0,1,2,2,3,4,3,2,4\rangle$$

Ahora el segundo 3 cambia una vez de posición:

$$\langle 0,0,0,1,2,2,3,3,4,2,4\rangle$$

Finalmente el último 2 es swapeado tres posiciones. Entonces la cantidad de cruces del grafo es $2+4+2+1+3=12$.

Si bien en el caso general este algoritmo tiene un rendimiento superior, en peor caso sigue siendo $O(m^2)$. Veamos un tercer acercamiento al problema [1]: sea $|v_2|$ la cantidad de nodos de la partición 2 en un dibujo dado. Consideremos un arbol binario con 2^k hojas donde k es tal que $2^{k-1} < |v_2| <= 2^k$. El árbol se completa de modo de que cada nodo esté en una hoja (aunque podría haber hojas que tengan ningún nodo), dispuestos de izquierda a derecha según su orden en la partición. Este árbol tiene $2^{k+1}-1$ nodos, y además $k = \lceil log_2(|v2|)\rceil$. Todos los nodos del árbol (hojas o no) se inicializan con un contador de valor 0, y se tiene un contador adicional que representa el total de cruces con los ejes agregados hasta el momento, también inicializado en 0.

La idea es ordenar los ejes como lo hicimos para el algoritmo anterior. Luego lo que haremos es "agregar" los ejes en el árbol según dicho orden. Agregar el eje consiste en incrementar en uno el contador de la hoja correspondiente al nodo de la segunda componente del eje, y a continuación se incrementa también el contador del padre de dicha hoja, y sucesivamente se propaga el incremento por todos los contadores del único camino desde la hoja hasta la raíz. A medida que propagamos el incremento hacia arriba, modificamos el contador del total de cruces. Para esto, se chequea si el nodo en que estamos parados es hijo izquierdo de su padre. Si no lo es, se pasa al siguiente nodo, pero de serlo se suma al total de cruces del dibujo el valor del contador almacenado en el hermano del nodo actual (el hijo derecho del padre).

El procedimiento puede resultar confuso en un principio por lo cual mostraremos un ejemplo de su aplicación. Apliquemos el algoritmo para el grafo de 1.2:

Como tenemos $|v_2| = 5$, el árbol va a tener 8 hojas y un total de 15 nodos.

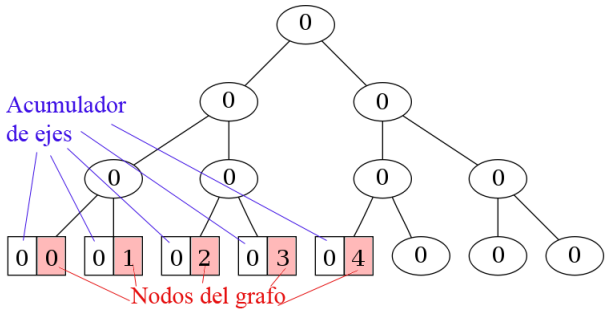


Figura 1.3: Árbol para contar cruces en el grafo de ejemplo

Los nodos cuadrados (del árbol) son las hojas que representan nodos del grafo (cuyo número es el de la derecha de la hoja). Cada nodo tiene un contador, en principio inicializado en 0. El contador de cruces también está inicializado en 0.

Recordemos que luego de ordenar los ejes el vector era:

$$\langle (0,0), (1,1), (1,2), (2,0), (2,3), (2,4), (3,0), (3,2), (4,3), (5,2), (5,4) \rangle$$

Lo primero que hacemos entonces es insertar el eje $(0,0)$, de modo que se incrementa el contador de la hoja 0 y este incremento se propaga hacia arriba. Como 0 está en un nodo izquierdo, lo que se hace es sumar a la cantidad de cruces el valor del contador del hermano de 0 (la hoja 1). La razón detrás de este procedimiento es que el valor de dicho contador indica la cantidad de ejes insertados que terminaban en el nodo 1; además dado el orden que se usa para agregar a los nodos, sabemos que la primer coordenada de estos ejes que terminaban en 1 era menor que la de ejes que estoy insertando ahora, y por lo tanto hay un cruce. En síntesis, agregué un eje $(a, 1)$ antes que uno $(b, 0)$ y por como estaban ordenados los ejes y nodos, sabemos que $a < b$ y como $1 > 0$, hay una inversión.

Al subir de nivel, como también estamos en un nodo izquierdo, agregamos a la cantidad de cruces el valor del contador del hermano correspondiente. En este caso, dicho contador guarda la cantidad de ejes agregados que terminan en 2 o 3. Procedemos de esta manera hasta la raíz del árbol.

Luego de esta inserción obtenemos el siguiente árbol:

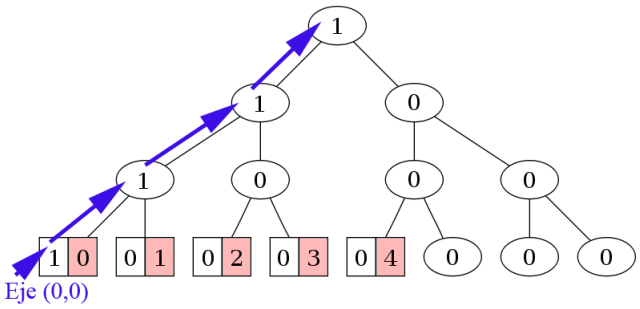


Figura 1.4: Árbol con el eje (0,0) insertado

De manera análoga, se insertan los ejes $(1,1)$ y $(1,2)$ sin que se generen cruces, y el árbol queda de la siguiente manera:

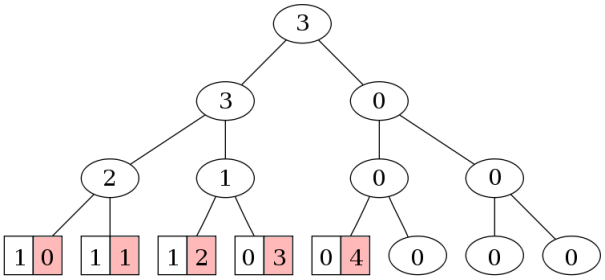


Figura 1.5: Árbol con los ejes (0,0), (1,1) y (1,2) insertados

Luego corresponde insertar el eje $(2, 0)$. Incrementamos en uno el contador de la hoja 0. Como es un nodo izquierdo, sumamos a cantidad de cruces el valor de la hoja 1, que es 1. Este incremento corresponde al cruce del eje $(2, 0)$ con el eje $(1, 1)$. Subimos un nivel e incrementamos en 1 el contador del padre de la hoja 0. Nuevamente, sumamos al contador de cruces el valor del contador del hermano del nodo actual, ya que otra vez estamos en un hijo izquierdo. Este nuevo incremento corresponde al cruce entre el eje $(2, 0)$ y el eje $(1, 2)$. Seguimos subiendo hasta llegar a la raíz, incrementando el valor de los contadores, y en caso de pasar por un nodo izquierdo, sumando el valor de los contadores de los nodos derechos. En este caso particular volvemos a caer en un nodo izquierdo, pero el contador de su hermano es 0. Esto se debe a que todavia no insertamos ningún eje que terminara en 4.

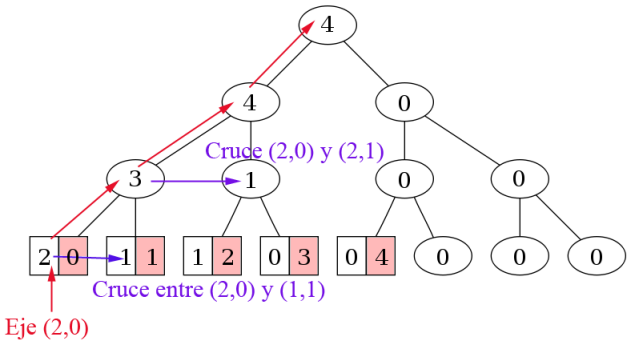


Figura 1.6: Árbol con los ejes (0,0),(1,1),(1,2) y (2,0) insertados

El procedimiento se repite de la misma manera hasta colocar todos los ejes. Como veremos más adelante, este procedimiento tiene como orden $O(max(m, |v_1|, |v_2|) + m * log(min(|v_1|, |v_2|)))$.

Repasando, tenemos 3 algoritmos para calcular los cruces:

- El primero consiste en revisar todo par de ejes posible. Tiene un costo $O(m^2)$.
- El segundo ordena los ejes y luego realiza *insertion sort* para contar inversiones. Su costo es $O(max(m, |v_1|, |v_2|) + c)$.
- El tercero utiliza el arbol binario para contar inversiones. Su orden es $O(max(m, |v_1|, |v_2|) + m * log(min(|v_1|, |v_2|)))$.

Todos estos algoritmos requieren conocer las posiciones de los nodos dentro de cada partición. En caso de no estar disponible, éste puede obtenerse $O(|v_1| + |v_2|)$, costo que se suma al de los algoritmos en caso de que no se tenga dicha información.

Si $m > log(min(|v_i|, |v_2|))$ conviene utilizar el tercer algoritmo, pues tiene una complejidad menor que la de los otros dos. Si en cambio $m < log(min(|v_1|, |v_2|))$ (un grafo con muy pocos ejes) resulta conveniente utilizar el segundo algoritmo ya que provee un mejor orden. Podemos saltear esta diferenciación preprocesando el grafo para evitar los casos donde ocurre el segundo caso. Para ello, construimos un grafo auxiliar eliminando todos los nodos aislados del grafo original (sean fijos o móviles). Esto es lícito ya que la inserción de nodos sin ejes no afecta de ninguna manera el criterio a optimizar (los cruces entre ejes). De esta manera, el algoritmo 3 se muestra como la mejor opción, ya que si $m > |v_i|$ resulta que $O(max(m, |v_1|, |v_2|) + m * log(min(|v_1|, |v_2|))) \subseteq O(m * log(min(|v_1|, |v_2|)))$. Este preprocesamiento se discute con más detalle en la sección 1.3.

Pseudocódigo

Algoritmo 1 Cuenta los cruces en un dibujo usando un arbol acumulador

Parámetros: particiones del dibujo (suponemos para simplificar que $|p_2| <= |p_1|$)
Parámetros: ejes del dibujo, posiciones de los nodos (índices dentro de su partición)

```
1: ejesPorPos ← [ ]
2: Para cada eje(x,y) del dibujo hacer
3:    agregar a ejesPorPos la tupla (posición de x, posición de y)
4: Fin para
5: lista ← RadixSort de ejesPorPos de modo que (x, y) < (z, w) si x < z ∨ x = z ∧ y < w
6: primerIndice ← 2^⌈log2(|p2|)⌉
7: arbol ← [0, ..., 0]   (2 * primerIndice − 1)
8: {árbol es el árbol acumulador}
9: decrementar primerIndice
10: cruces ← 0
11: Para cada eje de la lista hacer
12:    indice ← segunda componente del eje + primerIndice
13:    arbol[indice] ← arbol[indice] + 1 {un eje más en esta hoja}
14:    Mientras indice > 0 hacer
15:       Si indice es impar entonces
16:          {estoy en un nodo izquierdo}
17:          cruces ← cruces + arbol[indice]
18:       Fin si
19:       indice ← (indice − 1)/2 {subo un nivel}
20:       arbol[indice] ← arbol[indice] + 1
21:    Fin mientras
22: Fin para
23: Devolver cruces
```

1.2.2. Cálculo de complejidad

Sea v_i la cantidad de nodos de la particion i, y m la cantidad de ejes del grafo.

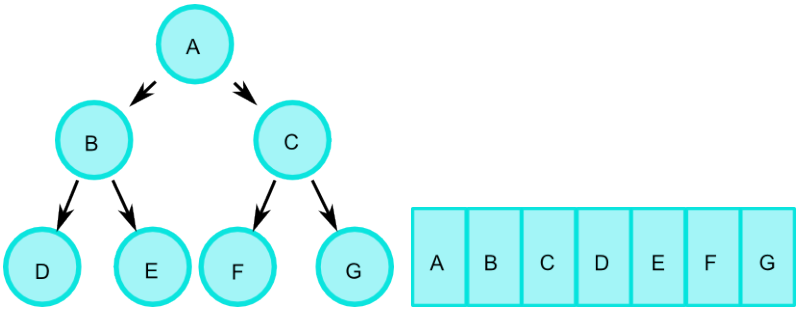
Si bien comentamos que la función podría, en caso de que no esté disponible, calcular los índices de los elementos dentro de su partición, esta funcionalidad no fue utilizada, de modo que el algoritmo recibe los índices ya precalculados. De este modo, el costo de armar y mantener los índices será tenido en cuenta en los respectivos algoritmos que utilicen a esta función.

Lo primero que hacemos es traducir los ejes segun los índices de sus nodos. Para hacer esto, lo que se hace es recorrer todos los ejes (se recorren las listas de adyacencia de los nodos de una de las particiones) y se van guardando en una lista los ejes con sus componentes traducidas (ciclo de la línea 2).

Luego ordenamos los ejes. Para esto utilizamos *radix sort*, cuyo costo es $max(m, v_1, v_2)$, puesto que tengo que ordenar m ejes y para hacerlo voy a necesitar de v_1 *buckets* primero y v_2 *buckets* más adelante.

Una vez ordenado y armado el árbol (tarea que puede llevarse a cabo en $O(v_2)$ como se verá más adelante) vamos a mirar todos los ejes. Para cada eje recorremos el árbol desde una hoja hasta la raíz. Dado que el árbol tiene 2^k hojas, tiene $log_2(2^k)$ de altura, pero $log_2(2^k) = k = \lceil log_2(v_2) \rceil$.

El árbol se puede implementar sobre un arreglo, de modo que la posición 0 sea la raíz, la 1 y 2 sus hijos izquierdo y derecho respectivamente, y así sucesivamente (de manera análoga a como se hace para implementar *heapsort* sobre arreglos). De esta manera, los nodos izquierdos del árbol quedan en las posiciones impares, y aumentar cada contador, así como moverse dentro del árbol de un nodo a su padre se puede hacer en $O(1)$. Por lo tanto, el costo de insertar todos los ejes es $O(max(m, v_1, v_2) + m * log\ v_2)$.



(a) Árbol acumulador (b) Representación sobre arreglo

Ahora bien, en lugar de ordenar primero por la primer componente y luego respecto a la segunda, podríamos hacer el mismo procedimiento pero ordenando primero por la segunda y luego por la primera y armar el árbol para p_1. Por lo tanto el procedimiento se podría realizar utilizando el p_i de menor cardinal, con lo cual el costo de las inserciones es de $O(max(m, v_1, v_2) + m * log(min(v_1, v_2)))$.

Como dijimos, este algoritmo podría no ser eficiente si la cantidad de ejes es muy pequeña, ya que para hacer *radix sort* se van a recorrer todos los nodos de ambas particiones, lo cual podría tener un costo más elevado que mirar todos los ejes. Sin embargo, también comentamos que vamos a filtrar a los nodos de grado 0 antes de trabajar con el dibujo, lo cual evita este caso y hace que el algoritmo sea eficiente para todas las entradas.

1.2.3. Reutilización de los cálculos

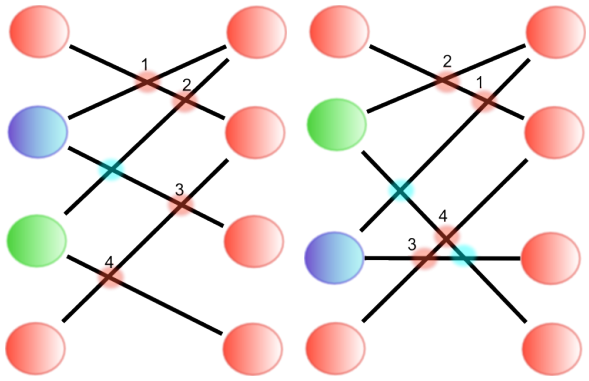
En muchas situaciones tiene sentido calcular completamente la cantidad de cruces de un dibujo. Sin embargo, para la implementación de muchos de los métodos de este trabajo práctico solo se realizan pequeñas modificiones en un dibujo y se desea conocer la variación de cruces que se produce por estos pequeños cambios. Recalcular los cruces del dibujo completo tiene un costo relativamente alto, y además se desperdician todos los cálculos previamente realizados que afectan a las partes del dibujo que no fueron modificadas.

Permutaciones de dos nodos consecutivos

En particular, si tenemos un orden de los nodos $\pi = \langle v_1, v_2, ...v_i, v_{i+1}, ..., v_n \rangle$ y realizamos un *swap* entre dos posiciones consecutivas i, $i+1$, podemos observar que si $\pi_1 = cruces(\langle v_1, v_2, ..., v_{i+1}, v_i, ...v_n \rangle$ y definimos $cruces(\pi, \rho)$ como la cantidad de cruces entre los ejes del grafo, dado que los nodos de la primer partición estan ordenados según π y los de la segunda segunda según ρ, vale que:

$$cruces(\pi_1, \rho) = cruces(\pi, \rho) - CrucesEntre(v_i, v_{i+1}, \rho) + CrucesEntre(v_{i+1}, v_i, \rho)$$

Donde $CrucesEntre(a, b, \rho)$ es la cantidad de cruces entre ejes de a y ejes de b si a está en una posición relativa menor que la de b y dado que los nodos de la otra partición estan en el orden ρ. Esto se debe a que, como dijimos anteriormente, los cruces dependen solo del orden relativo entre los nodos del dibujo. Entonces, si intercambiamos dos posiciones consecutivas, como el orden relativo de los demás nodos se mantiene (es decir, los que estaban "abajo" de ellos siguen estando allí, y los que estaban "arriba" también), solo se modifican los cruces que ocurrían entre los dos nodos afectados por el *swap*.



(c) (d)

Figura 1.7: *Swap* de nodos consecutivos

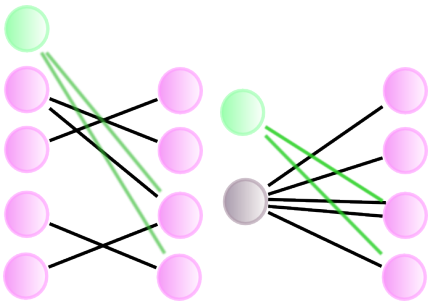
Para calcular $CrucesEntre()$ se puede utilizar el algoritmo antes descripto para el conteo de cruces en general, simulando una partición que sólo contenga a los nodos a y b, y teniendo en cuenta únicamente a los ejes que salen de ellos dos. En este caso, como la partición mas chica tiene 2 nodos (o menos, si una partición era solo un nodo), resulta que el algoritmo puede calcular los cruces en $O(|v_j| + m_a + m_b)$ siendo m_i la cantidad de ejes incidentes al nodo i y v_j es partición opuesta. Básicamente, lo que se hace es mirar solamente los ejes que salen de a y de b, armar un árbol de dos posiciones y aplicar el procedimiento anteriormente descripto.

En el caso en que a y b tengan pocos ejes, de modo tal que $m_a * m_b < |v_j|$, utilizamos la versión simple del algoritmo que toma todos los pares de ejes de a y b. De este modo el orden para contar cruces en un *swap* es $min(max(|v_j|, m_a, m_b), m_a * m_b)$.

Agregado y extracción de nodos

Otro caso donde resulta útil recalcular todos los cruces es cuando se agrega un nodo al dibujo existente, y más particularmente si se agrega al principio o al final de una partición. Esta situación es bastante común también, y, sumada a la de calcular cruces por *swap* nos permite obtener de forma eficiente los cruces que se producen por agregar un nodo en cualquier posición de una partición (ya que primero se lo agrega en un extremo, y luego se lo permuta hasta la posición desdeada).

En este caso, los cruces existentes entre otros nodos se mantienen, y solo se podrían agregar nuevos cruces con los ejes del nodo recién agregado. Por lo tanto, una estrategia válida para calcular los cruces agregados es colapsar los nodos de la partición donde se está agregando el nuevo nodo, dejando solo al nuevo nodo y a un nodo w con todos los ejes del resto de la partición (producto de "compactarlos" como ilustra la figura 1.8), de modo de que el algoritmo sea $O(m + |v_1| + |v_2| + m)$. Esto es así porque el costo de la traducción y del *radix sort* no pueden evitarse, pero al igual que en el caso de los cruces por permutación entre dos nodos consecutivos, el árbol tiene solo 2 elementos: el nodo que se agrega y el nodo producto de la compactación.



(a) (b)

Figura 1.8: Compactación del grafo para inserción de un nodo

Estas situaciones pueden parecer muy específicas pero responden a las necesidades particulares de los algoritmos presentados en el resto del trabajo.

1.3. Preprocesamiento del grafo

Vimos previamente que un algoritmo de preprocesamiento sencillo y ejecutado una única vez nos permite disminuir el costo de algunas operaciones. Este algoritmo debería eliminar los nodos aislados y computar nuevos valores para los nodos que mantengan el invariante del grafo de entrada respecto de la numeración de los nodos. Después de calcular la solución al problema, un segundo algoritmo vuelve a completar el grafo con los nodos quitados insertándolos en posiciones válidas, y restituye los valores originales de los nodos para poder presentar la solución al usuario.

Para hacerlo, básicamente tomamos los nodos de la entrada, así como sus ejes y construimos un dibujo. A partir de este dibujo, que ya tiene armadas las listas de adyacencia, podemos separar en una lista a los nodos de grado distinto a cero.

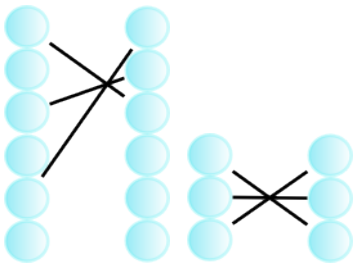


Figura 1.9: Filtrado de nodos aislados

Si nos contentáramos con extraer los nodos, nos encontraríamos frente al problema de que los nodos no tienen como identificadores números consecutivos. Esto viola el invariante del problema original y si bien podría no ser problemático, nos permite hacer optimizaciones varias, en particular respecto del tamaño de los diccionarios que almacenan valores asociados a un cierto nodo.

Por esta razón, no solo filtramos a los nodos de grado nulo, sino que se realiza un nueva numeración de los nodos que quedan. La misma cumple que los nodos fijos tienen númeración consecutiva (que además indica la posición relativa, es decir si $a¡b$ con a y b nodos de la misma partición, entonces el nodo fijo a está antes que el nodo fijo b) y que los números son asignados primero a los nodos fijos de la primer partición, luego a los nodos fijos de la segunda, en tercer lugar a los nodos móviles de la primera y finalmente a los móviles de la segunda.

Para realizar dicha traducción se utilizan diccionarios sobre arreglos a fin de poder realizarla rápidamente. Se asocia a cada identificador nuevo el identificador que tenía el nodo en el grafo original, para poder deshacer la traducción más adelante. Los ejes también son traducidos, por lo cual se necesita momentáneamente poder ir de un identificador viejo a su nuevo valor, por lo cual se necesita provisoriamente otro índice de identificadores.

Una vez que se realizó la traducción, las heurísticas y el algoritmo exacto trabajan con el dibujo nuevo. Cuando terminan, se realiza el proceso inverso para volver de los nodos nuevos a los viejos, y además se inserta a los nodos viejos aislados en posiciones válidas dentro del dibujo devuelto por el algoritmo elegido.

Todo este proceso nos agrega un costo $O(|v_1| + |v_2| + m)$ donde $|v_i|$ es la cantidad de nodos de la partición i original (sin filtrar) y m la cantidad de ejes. En el peor caso, ningún nodo tiene grado cero, por lo que esta mejora representa un *overhead*. Sin embargo, como se verá mas adelante, el orden de los distintos algoritmos es lo suficientemente elevado como para hacer despreciable este costo adicional, mientras que las mejoras obtenidas en los casos favorables hacen muy apreciable su utilización.

Filtrar a los nodos aislados tiene sentido como optimización general más allá del conteo de cruces. En el caso del algoritmo exacto, donde se evalúa un árbol de permutaciones de los nodos cuyo tamaño es de orden factorial en la cantidad de nodos, eliminar al menos unos pocos nodos aislados disminuye sustancialmente el tiempo necesario para computar la solución ya que elimina una cantidad muy grande de permutaciones que solo difieren de otras en la posición de los nodos aislados (que, como dijimos, no afectan al criterio a optimizar).

Parte 2

Algoritmo exacto

2.1. Desarrollo

Un dibujo incremental válido consiste en una permutación de los nodos de cada partición que mantenga el orden relativo de los nodos previamente fijados. Dados v_i nodos originales en la partición i, se agregan IV_i nodos en cada partición. La cantidad posible de soluciones es:

$$IV_1! * \binom{IV_1 + v_1}{v_1} * (IV_2! * \binom{IV_2 + v_2}{v_2})$$

Esto se debe a que la partición 1 tiene $IV_1 + v_1$ nodos, y por lo tanto hay esa cantidad de posiciones. De esas, v_1 estarán destinadas a los nodos existentes, cuyo orden relativo es fijo. Una vez que elegimos sus posiciones, el orden entre ellos es fijo. En las IV_1 posiciones restantes podemos poner cualquier permutación de los nodos nuevos. Luego la cantidad de órdenes válidos para la partición 1 es:

$$IV_1! * \binom{IV_1 + v_1}{v_1}$$

Luego, para cada uno de estos órdenes válidos en la partición 1, tenemos (análogamente) una cantidad equivalente para la partición 2:

$$IV_2! * \binom{IV_2 + v_2}{v_2}$$

permutaciones en la segunda partición. El total de combinaciones es finalmente el producto de las combinaciones de cada partición, que resulta en la fórmula presentada anteriormente.

Dada la naturaleza exponencial del problema a resolver, decidimos utilizar la técnica de *backtracking* para formular un algoritmo exacto. Comenzamos por desarrollar un algoritmo de fuerza bruta que simplemente explora el árbol de combinaciones que va generando progresivamente, y luego lo fuimos refinando agregando optimizaciones y podas.

El algoritmo de backtracking aprovecha la naturaleza recursiva del problema de dibujo incremental, agregando progresivamente cada nodo móvil en todas sus posiciones válidas y produciendo así un nuevo conjunto de nodos fijos que se incrementará con una llamada recursiva. Dado un candidato inicial con una cantidad de cruces dada, esta situación nos permite realizar una poda sencilla del árbol de combinaciones. Ocurre que inevitablemente todo dibujo incremental del grafo parte de un dibujo original cuya cantidad de cruces acota inferiormente la del dibujo incrementado. Por lo tanto, al construir un candidato para una llamada recursiva, si la cantidad de cruces en su parte fija supera a la del mejor candidato hallado hasta el momento, no tiene sentido descender por la rama y puede podarse sin perder soluciones.

Con esta idea, resulta útil proveerse rápidamente de un candidato inicial cuya cantidad de cruces sea baja, ya que *a priori* permitirá descartar mayor cantidad de ramas por pasarse de su valor. Con este fin, tiene sentido utilizar alguna solución heurística de las desarrolladas en este trabajo.

El pseudocódigo del algoritmo resultante es aproximadamente:

Algoritmo 2 Halla la solución exacta al problema de dibujo bipartito incremental

Parámetros: fijo1, fijo2, movil1, movil2, adyacencias

```
construir un candidato abritrario y
Si fijo1 = ∅ y fijo2 = ∅ entonces
  Si el dibujo obtenido tiene menos cruces que el mejor candidato entonces
    reemplazar el mejor candidato por este dibujo
  Fin si
Si no y fijo1 ≠ ∅ entonces
  tomar el primer elemento de movil1
  Para cada posición del elemento en fijo1 hacer
    poner el elemento en esa posición
    Si el dibujo obtenido no tiene más cruces que el mejor candidato entonces
      llamar recursivamente
    Fin si
    sacar el elemento de esa posición
  Fin para
Si no y fijo2 ≠ ∅ entonces
  tomar el primer elemento de movil2
  Para cada posición del elemento en fijo2 hacer
    poner el elemento en esa posición
    Si el dibujo obtenido no tiene más cruces que el mejor candidato entonces
      llamar recursivamente
    Fin si
    sacar el elemento de esa posición
  Fin para
Fin si
Devolver el mejor candidato hallado
```

De esta manera, el árbol de *backtracking* que tenemos es el de la figura 2.1:

El algoritmo lo recorre en orden DFS, cortando aquellas ramas que pueden ser descartadas inmediatamente sin visitarlas.

2.1.1. Implementación eficiente

Dado que el algoritmo recursivo se ejecutará una vez por cada nodo del árbol de combinaciones, es importante que su ejecución sea lo más eficiente posible para disminuir el tiempo total de ejecución.

La primera versión del algoritmo era similar a la de fuerza bruta: recorría el árbol de combinaciones, y cuando obtenía una permutación completa, construía el dibujo y contaba enteramente sus cruces. A continuación agregamos la poda simple descripta anteriormente. Sin embargo, resultaba claro que recalcular los cruces de todo el dibujo para cada hoja del árbol de permutaciones no era eficiente ya que gran parte de los cálculos eran redundantes entre hojas vecinas del árbol, puesto que compartían gran parte de las posiciones de los nodos en el dibujo.

Utilizando los métodos descriptos anteriormente, decidimos efectuar los cálculos mediante una técnica incremental. Constatamos que la iteración que en el pseudocódigo corresponde a agregar un nodo móvil en todas las posiciones posibles dentro del dibujo fijo de su partición puede describirse en términos de 3 operaciones: agregar el nodo al final del dibujo, permutarlo n veces hacia atrás con su vecino inmediato, y finalmente extraerlo del principio del dibujo donde habrá quedado ubicado. Con este procedimiento, un nodo móvil dado pasa por todas las posiciones posibles dentro del dibujo fijo original. La figura 2.2 ilustra este proceso.

Dado un candidato, la cantidad de cruces que se agregan por agregarle un nuevo nodo al final a una partición puede ser calculada mucho más rápidamente que los cruces de todo el dibujo. Además, como se vio anteriormente, calcular la cantidad de cruces que resulta de un *swap* también es eficiente. Esta mejora se incluye en el algoritmo evitando así recalcular todos los cruces para cada permutación, y en cambio llevando una cuenta temporal de cruces que se modifica continuamente para reflejar los cruces del candidato que se está evaluando.

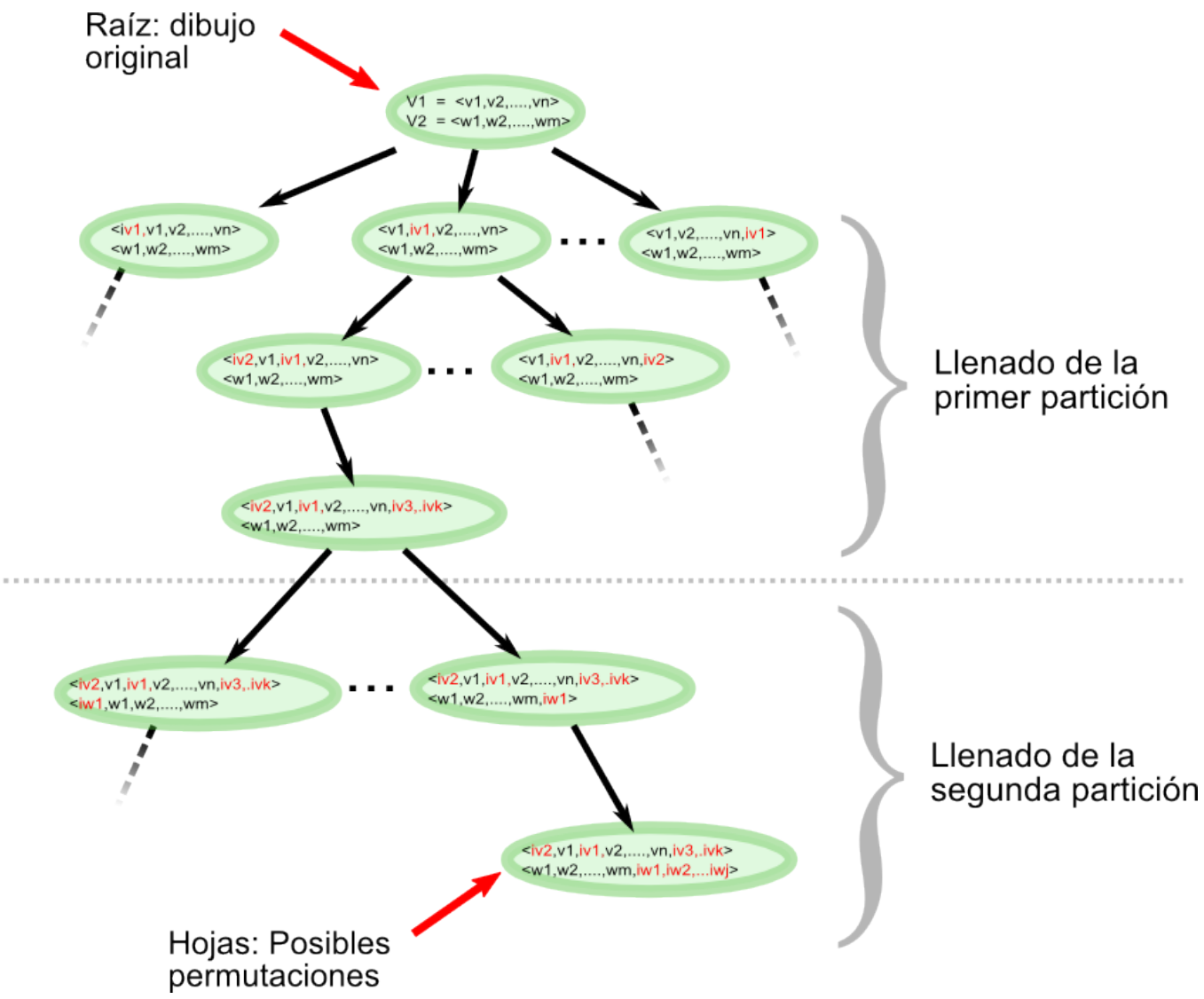


Figura 2.1: Árbol de *backtracking*

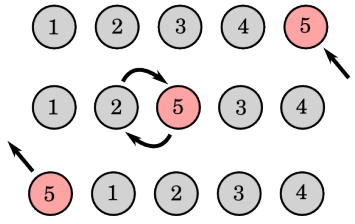


Figura 2.2: Permutaciones mediante swaps

2.1.2. Tabulado de resultados

Aún tras incluir los cálculos incrementales como se describió en el último párrafo, se puede aprovechar de forma más eficiente aún la realización de ciertos cálculos. Consideremos un dibujo con dos permutaciones $V =< v_1, v_2, ..., v_n >$, $W =< w_1, w_2, ..., w_k >$. La cantidad de cruces puede obtenerse como:

$$Cruces(V, W) = \sum_{i=1}^{k-1} \sum_{j=i+1}^{k} crucesEntre(w_i, w_j, V)$$

Esto es, dada una permutación de V, los cruces de todo el dibujo, para cualquier permutación de W, dependen únicamente de los valores de $crucesEntre(w_i, w_j, V)$, función que calcula los cruces entre dos nodos w_i y w_j para la permutación de V elegida y suponiendo que w_i está antes de w_j en el dibujo. Esto puede hacerse ya que la cantidad de cruces que se producen entre los ejes de 2 nodos de una partición depende únicamente de su orden relativo y de las posiciones dentro de la partición restante.

Por lo tanto, una vez que construimos una permutación de una de las 2 particiones (parte superior del árbol en el gráfico explicativo del árbol de *backtracking*), podemos tabular los k^2 valores de $crucesEntre()$ necesarios para el cálculo de los cruces de cualquier permutación de la otra partición: todo el subárbol que pende de una permutación completa de la partición 1 comparte los mismos valores de $crucesEntre()$. Teniendo esta tabla, el cálculo de cruces del dibujo completo puede realizarse mediante la suma de los mismos según la fórmula de arriba, lo cual podría agilizar el conteo de cruces. Cabe destacarse que el llenado de esta tabla tiene un costo no despreciable, y por tanto es importante realizar pruebas para determinar si las ventajas de realizar este tabulado no son superadas por el costo de la creación de la tabla.

Como se verá más adelante, las pruebas mostraron que el uso de esta tabulación mejora significativamente el rendimiento del algoritmo. En función de esta mejora, podemos observar que los cálculos realizados con ayuda de la tabla son mucho más rápidos que los que se realizan sin ella. Por lo tanto decidimos incluir una segunda mejora que consiste en decidir cual de las 2 particiones tiene más permutaciones posibles, y asignar ésta a la parte inferior del árbol cuya exploración está más optimizada gracias al uso de la tabla. Antes de esta decisión, se tomaba de forma arbitraria que lo recibido como "partición 1" se ubicaba en la parte superior del árbol, puesto que la situación era simétrica y solo influía el tamaño total del árbol.

Si bien no es posible realizar la misma tabulación para la partición asignada a la parte superior del árbol (puesto que no disponemos de la ubicación de los nodos en la otra partición aún), es posible realizar una optimización parcial: si bien no conocemos las posiciones de *todos* los nodos en la partición vecina, si conocemos la posición de aquellos que son fijos y por tanto no modifican su orden relativo. Se puede construir una tabla más pequeña para, nuevamente, agilizar el conteo de cruces en las permutaciones de la partición de la parte superior.

2.1.3. Podas

Disponiendo de la tabla de cruces entre pares de nodos, pudimos construir una función de poda más efectiva, mediante una cota inferior más fina para la cantidad mínima de ejes que produce una rama. Nuevamente, para una permutación dada de V, tenemos que:

$$Cruces(V, W) \geq \sum_{i=1}^{k-1} \sum_{j=i+1}^{k} min(crucesEntre(w_i, w_j, V), crucesEntre(w_j, w_i, V))$$

Como tenemos estos valores tabulados, resulta muy sencillo calcular esta cota y podar en función del valor obtenido. Esto utiliza el hecho de que dados dos nodos w_i, w_j de la partición 2, una vez que se los coloque en el dibujo, agregarán una cierta cantidad de cruces (ya sea $crucesEntre(w_i, w_j)$ o $crucesEntre(w_j, w_i)$ dependiendo de su orden relativo). Como disponemos de estos dos valores, podemos usar el mínimo entre ambos como una cota inferior de los cruces producidos por la dupla w_i, w_j dada la permutación de V tomada.

Una vez que tenemos esta cota inferior, podemos descartar de antemano a aquellas ramas donde el valor de dicha cota supere la cantidad de cruces del mejor candidato obtenido hasta el momento. Una vez más, este cálculo redunda en un costo adicional que podría ser contraproducente en caso de que la poda no lograra eliminar una cantidad significativa de ramas. Nuevamente, es necesario realizar pruebas para determinar su efectividad. Las mediciones observaron que este mecanismo de poda es particularmente bueno y elimina partes sustanciales del árbol de permutaciones sin un costo excesivo (producto, en gran parte, de la disponibilidad de la tabla de cruces descripta previamente, que aligera mucho el cómputo de la cota).

2.2. Pseudocódigo

Algoritmo 3 Resuelve de forma exacta el problema de dibujo incremental de grafos bipartitos

```
1: inicializarParametros
2: mejorSolucion ← construir una solución con la heuristica constructiva
3: buscarMejorSolucion()
4: Devolver mejorSolucion
```

Algoritmo 4 Inicializa variables que va a usar el algoritmo exacto

```
1: determinar cual de las particiones tiene menos permutaciones, llamarla 1, a la otra llamarla 2
2: construir índice de posiciones para los nodos que ya están en el dibujo
3: construir listas de adyacencia para los nodos de 1 y los fijos de 2
4: calcular los cruces del candidato original
5: construir la tabla para la partición 1
6: Si no hay móviles en la partición 2 entonces
7:    construir la tabla para la partición 2
8: Fin si
```

Algoritmo 5 Construye todas las particiones que pueden ser solución buscando la mejor (o una de las mejores)

```
1: Si no hay mas móviles por poner entonces
2:    Si los cruces de la solución que armé son menos que los de la mejor hasta ahora entonces
3:       mejorSolucion ← solucionActual
4:    Fin si
5: Si no y no hay mas móviles que poner en la partición 1 entonces
6:    agregar atrás al primer móvil a la partición (sacarlo de móviles y actualizar índices)
7:    cruces ← cruces + los cruces que me agrega el nuevo nodo
8:    Mientras el nuevo no recorrió todas las posiciones en la partición hacer
9:       Si cruces + minimoCruces ¡los cruces del mejor entonces
10:         buscarMejorSolucion()
11:      Fin si
12:      intercambiar al nuevo por el nodo que está adelante de él (actualizar posiciones y cruces)
13:   Fin mientras
14:   sacar al nodo {quedó adelante de todo en la partición}
15:   actualizar posiciones y cruces
16: Si no
17:   agregar atrás al primer móvil a la partición (sacarlo de móviles y actualizar índices)
18:   cruces ← cruces + los cruces que me agrega el nuevo nodo
19:   Mientras el nuevo no recorrió todas las posiciones en la partición hacer
20:      Si cruces + minimoCruces ¡los cruces del mejor entonces
21:         Si no quedan más nodos móviles en esta partición entonces
22:            construir tabla para la partición 2
23:         Fin si
24:         buscarMejorSolucion()
25:      Fin si
26:      intercambiar al nuevo por el nodo que está adelante de él (actualizar posiciones y cruces)
27:   Fin mientras
28:   sacar al nodo {quedó adelante de todo en la partición}
29:   actualizar posiciones y cruces
30: Fin si
```

- Para construir la tabla 1, se toman todos los nodos de la partición 1 de a pares (i, j) y se guardan en una matriz en la posición (i, j) el valor de $CrucesEntre(i, j)$. Para contar los cruces, solo se tienen en cuenta los nodos fijos de la partición 2. Para construir la tabla 2 se procede análogamente pero se toman en cuenta a todos los nodos de la partición 1 (no solo a los fijos) para contar los cruces.

- minimoCruces utiliza las tablas (la correspondiente a la partición que se está llenando) y realiza la cuenta antes descripta para obtener una cota inferior a la cantidad de cruces que se pueden originar agregando los nodos faltantes en cualquier orden.

Las listas de adyacencia parciales, así como los vectores de posiciones parciales se utilizan como datos de entrada para las funciones incrementales de conteo de cruces. Al ir manteniendo actualizados estos datos durante la ejecución, evitamos el costo de recalcular índices y adyacencias de todo el grafo al hacer modificaciones incrementales.

2.3. Detalles de implementación

La implementación del algoritmo fue realizada originalmente en Python por la facilidad que brinda el lenguaje de alto nivel para hacer pruebas y comparaciones. Se comenzó por la versión de fuerza bruta y se fue refinando progresivamente el algoritmo agregando optimizaciones, varias de las cuales fueron descriptas anteriormente. Es importante destacar que las optimizaciones no fueron hechas de forma aleatoria sino que se realizó periódicamente un *profiling* del algoritmo para determinar cuales eran las funciones que insumían más tiempo durante la ejecución, y luego enfocar los esfuerzos de optimización en ellas. Una vez terminado, el algoritmo completo se reimplementó en C++.

Vale la pena comentar algunos detalles y optimizaciones propios de la implementación. En primer lugar, se utilizaron *buffers* únicos para almacenar las secuencias de nodos fijos y móviles que utiliza cada llamada recursiva, que se almacenan en atributos de la clase *SolucionExacta* y por tanto son compartidos por todas las llamadas recursivas. Esto permite ahorrar memoria en el *stack* (que puede tomar un tamaño considerable) reduciéndolo al tamaño mínimo indispensable. Se evitan además operaciones de copias innecesarias de estos parámetros. Esto tiene como efecto secundario la imposibilidad de ejecutar en paralelo dos instancias del algoritmo en una máquina multiprocesador. Si se pasan todos los parámetros por copia (incluyendo posiblemente las tablas de valores) la mejora de tiempo de ejecución del algoritmo en multiprocesadores puede ser esencialmente lineal en la cantidad de CPUs.

Las secuencias en cuestión se implementaron sobre lista doblemente enlazada para permitr las operaciones de insertar atrás, *swap* y extraer de adelante en $O(1)$. Todos los diccionarios se implementaron sobre vectores ya que sus tamaños son lineales o a lo sumo cuadráticos en $V_1 + V_2$ y por lo tanto no tiene sentido ahorrar memoria en estos aspectos a costa de velocidad cuando el consumo de memoria del algoritmo estará determinado por el tamaño del *stack*.

Para construir el candidato inicial se utilizó la heurística constructiva que describimos en 3.

2.4. Cálculo de complejidad

Antes de correr nuestro algoritmo exacto utilizamos el algoritmo de preprocesamiento descripto en 1.3, lo cual conlleva un costo inicial de $O(V_1 + V_2 + m)$ donde V_i es la cantidad de nodos de la partición i sin filtrar y m la cantidad de ejes del grafo.

Definimos además definimos $moviles_i$ como la cantidad de nodos nuevos, ya filtrados en la partición i. Por otro lado $fijos_i$ es la cantidad de nodos viejos ya filtrados.

En primer lugar el algoritmo inicializa ciertas variables que nos serán de utilidad tales como las listas de adyacencia parciales, vectores con posiciones, etc. Todo esto tiene un costo $O(v_1 + v_2 + m)$ donde v_i es la cantidad de nodos de la partición i ya filtrada. Esto lo podemos acotar por $O(v_max + m * log(v_max))$ con $v_max = max(v_1, v_2)$. En esta inicialización se llena la tabla de resultados para la primer partición. Lo que calculamos es $crucesEntre(w_i, w_j) \forall w_i, w_j \in v_1$. Esto tiene un costo $O(v_1^2 * v_{max})$, ya que se efectúan $O(v_1^2)$ cálculos y cada cálculo tiene costo $O(v_{max})$ como se indicó previamente.

Para realizar el estudio de este algoritmo decidimos considerar por separado el llenado de cada partición. Como comentamos antes, primero se llena una de las particiones, y a continuación la otra, pero se efectúan optimizaciones distintas en función del tabulado dependiendo de qué caso se trate. Decidimos además ignorar las podas ya que no es posible prever fácilmente si se realizarán o no, por lo tanto consideraremos (como peor caso) el caso en que se evalúan todas las podas pero éstas no permiten nunca descartar casos.

Primero observemos el costo que tenemos al pasar por cada nodo del arbol de permutaciones de la primer partición. En cada paso lo que hacemos es insertar un elemento al final, y contar cuantos cruces nos agrega, lo cual puede hacerse en $O(v_1 + m)$. Esto último no es una cota fina puesto que únicamente se inserta en cada nivel tantas veces como nodos

padre tenga ese nivel, pero utilizamos esta cota para simplificar el cálculo. Luego intentamos aplicar la poda: este cálculo tiene un costo $O(v_1^2)$ ya que tengo que obtener la suma de los mínimos de los cruces entre dos elementos y dichos elementos ya están en la tabla. Después hacemos otra llamada para ir armando el siguiente nivel (analizada por separado). Una vez que regresamos de esa llamada, hacemos un *swap* del nodo con un adyacente y actualizamos los cruces, acción que puede hacerse en $O(1)$ ya que tenemos los cruces entre ambos tabulados. Una vez que el elemento recorre toda la partición hay que sacarlo, lo cual también tiene un costo $O(v_1 + m)$ porque se actualizan nuevamente los cruces.

Entonces recorrer el árbol de llenado de la primer partición tiene un costo:

$$O(nodos * (v_1^2 + m))$$

Donde *nodos* es la cantidad de nodos del árbol de *backtracking*. Veamos qué valor tiene esta variable. Podemos obtener la cantidad de nodos del árbol en función de la cantidad de nodos móviles y fijos que tiene la partición involucrada de la siguiente manera:

$$tamArbol(moviles, fijos) = \begin{cases} 1 & \text{si } moviles = 0 \\ (fijos + 1) * tamArbol(moviles - 1, fijos + 1) + 1 & \text{si } moviles \neq 0 \end{cases} \quad (2.1)$$

Esta fórmula no es fácilmente manejable, razón por la cual usaremos una cota. Dado que cada nivel tiene más nodos que el nivel anterior (si tengo una permutación de k elementos, al agregar un nuevo elemento obtenemos $k+1$ posibles órdenes), podemos acotar la cantidad de nodos del árbol como:

$$h * moviles_1! * \binom{moviles_1 + fijos_1}{fijos_1}$$

Donde h es la altura del arbol y $moviles_1! * \binom{moviles_1 + fijos_1}{fijos_1}$ es la cantidad de hojas. La altura del árbol es igual a la cantidad de nodos móviles en la partición 1, ya que en cada nivel estamos agregando un nodo, mas 1 de la raíz donde no agregamos nada todavía. Luego:

$$nodos \leq (moviles_1 + 1) * moviles_1! * \binom{moviles_1 + fijos_1}{fijos_1}$$

Luego el orden de completar todas las permutaciones de la primer partición es:

$$O(moviles_1 * moviles_1! * \binom{moviles_1 + fijos_1}{fijos_1}(m + v_1^2))$$

En cada hoja del árbol de *backtracking* de la primer partición, completamos la tabla para la segunda partición. Esto tiene un orden $O(v_2^2 * v_{max})$.

En segunda instancia, completar la segunda partición tiene el mismo costo que completar la primera, con la salvedad de que en las hojas tenemos un costo $O(v_1 + v_2)$ por copiar al mejor dibujo (el grafo no se copia, se usa una referencia) en el atributo *mejorDibujo*. Entonces, completar el árbol de la segunda particion tiene un costo:

$$O(moviles_2 * moviles_2! * \binom{moviles_2 + fijos_2}{fijos_2}(m + v_2^2) + (v_1 + v_2) * moviles_2! * \binom{moviles_2 + fijos_2}{fijos_2}(m + v_2^2))$$

Ahora, para cada hoja del árbol de la primer partición, llenamos todo un árbol de la segunda. Por lo tanto el costo final es el producto de ambas complejidades, lo que redunda en:

$$O(moviles_1! * \binom{moviles_1 + fijos_1}{fijos_1}((m + v_1^2) * moviles_1 + moviles_2! * \binom{moviles_2 + fijos_2}{fijos_2} * ((m + v_2^2) * moviles_2 + (v_1 + v_2)) + v_2^2 * v_{max}) + V_1 + V_2)$$

Y usando que:

$$moviles_1! * \binom{moviles_1 + fijos_1}{fijos_1} = \frac{(moviles_1 + fijos_1)!}{fijos_1!}$$

:

tenemos que:

$$O(\frac{(moviles_1 + fijos_1)!}{fijos_1!}((m + v_1^2) * moviles_1 + \frac{(moviles_2 + fijos_2)!}{fijos_2!} * ((m + v_2^2) * moviles_2 + (v_1 + v_2)) + v_2^2 * v_{max}) + V_1 + V_2)$$

Por otro lado, resulta que $moviles_i + fijos_i = v_i$, entonces tenemos lo siguiente:

$$O(\frac{v_1!}{fijos_1!}((m + v_1^2) * moviles_1 + \frac{(v_2)!}{fijos_2!} * ((m + v_2^2) * moviles_2 + (v_1 + v_2)) + v_2 * v_{max}) + V_1 + V_2)$$

El tamaño de la entrada t, es:

$$t = log(P_1) + \sum_{i=1}^{P_1} log((p_1)_i) + log(P_2) + \sum_{i=1}^{P_2} log((p_2)_i) + log(m_p) + \sum_{i=1}^{m_p} log((e_i)_0) + log((e_i)_1)$$

$$+log(IV_1) + \sum_{i=1}^{IV_1} log((iv_1)_i) + log(IV_2) + \sum_{i=1}^{IV_2} log((iv_2)_i) + log(m_{iv}) + \sum_{i=1}^{m_{iv}} log((e'_i)_0) + log((e'_i)_1)$$

donde P_i es la cantidad de nodos originales de la primera partición, m_p es la cantidad de ejes originales, IV_i es la cantidad de nodos que se agregan a la partición i y m_iv es la cantidad de ejes que se agregan.

dado que $t \geq v_i$, $t \geq m$, $t \geq V_i$, resulta que:

$O(\frac{v_1!}{fijos_1!}((m + v_1^2) * moviles_1 + \frac{(v_2)!}{fijos_2!} * ((m + v_2^2) * moviles_2 + (v_1 + v_2)) + v_2 * v_{max}) + V_1 + V_2) \subseteq O(t!(t^3 * t! + P(t)))$, con P(t) un polinomio en t. Luego el algoritmo es $O(t^3 * t!^2)$

2.5. Análisis experimental

2.5.1. Experiencias realizadas

A continuación se presentan varias experiencias que fueron realizadas con el propósito de examinar el comportamiento de los algoritmos, así como de las optimizaciones que fueron realizadas.

Observamos primero la efectividad de las optimizaciones más importantes que realizamos sobre el algoritmo de *backtracking* trivial:

- En primer lugar buscamos mostrar la ventaja de tabular los valores de *crucesEntre* en memoria y utilizar los métodos incrementales de conteo de cruces respecto de la implementación más trivial que consiste en volver a contar completamente los cruces de todo el dibujo formado en cada hoja del árbol de *backtracking*. Esta prueba fue realizada con la implementación en Python de los algoritmos, utilizado CPython 2.5 para 3 casos grandes del problema, promediando 10 ejecuciones sobre grafos del mismo tamaño pero con ejes aleatorios para cada punto. Se consideró un grafo "ralo" si tiene el 15 % de los ejes del grafo bipartito completo, y "denso" cuando tiene el 85 %.
- En segundo lugar examinamos el comportamiento del algoritmo con tabulado de valores según si se utiliza o no la inversión de las particiones (para garantizar que la mitad del árbol más grande sea la segunda que se completa). Una vez más las comparaciones se hicieron en Python promediando 10 ejecuciones sobre instancias grandes del problema.
- Por último comparamos la eficiencia de los dos criterios de poda: el más sencillo que consiste únicamente en eliminar las ramas que se "pasan"del mejor candidato obtenido hasta el momento, y la más avanzada que consiste en utilizar la cota inferior que se desprende de los valores tabulados previamente. Se contabiliza la cantidad de nodos visitados del árbol de *backtracking* como porcentaje del tamaño total del árbol. Se utilizó una instancia simétrica de tamaño medio y se promediaron 20 ejecuciones para cada punto.

A continuación nos enfocamos en la performance del algoritmo completo:

- Graficamos primero el tiempo insumido por el algoritmo para resolver un caso de tamaño medio variando la cantidad de ejes (que no afecta el tamaño del árbol de *backtracking* que depende únicamente de los nodos) pero sí interviene en algoritmos auxiliares.
- Finalmente examinamos el tiempo que insume el algoritmo para resolver instancias de diferentes tamaños. Tomamos como variable el tamaño del árbol de *backtracking* característico de la instancia en cuestión ya que nos pareció mucho más representativo de la dificultad de la instancia que simplemente la cantidad de nodos del grafo. Los casos fueron elegidos de forma conveniente para obtener puntos a intervalos regulares, pero dado que el tamaño del árbol crece de forma importante cuando se aumentan las cantidades de nodos, no fue fácil encontrar valores apropiados que permitan regularidad perfecta. Se graficaron tiempos para distintas cantidades de ejes en cada tamaño de grafo. Se utilizó una escala logarítmica en ambos ejes del gráfico para permitir su lectura dadas las magnitudes involucradas, y para ofrecer un punto de referencia se acotaron los tiempos por una función conveniente.

2.5.2. Resultados

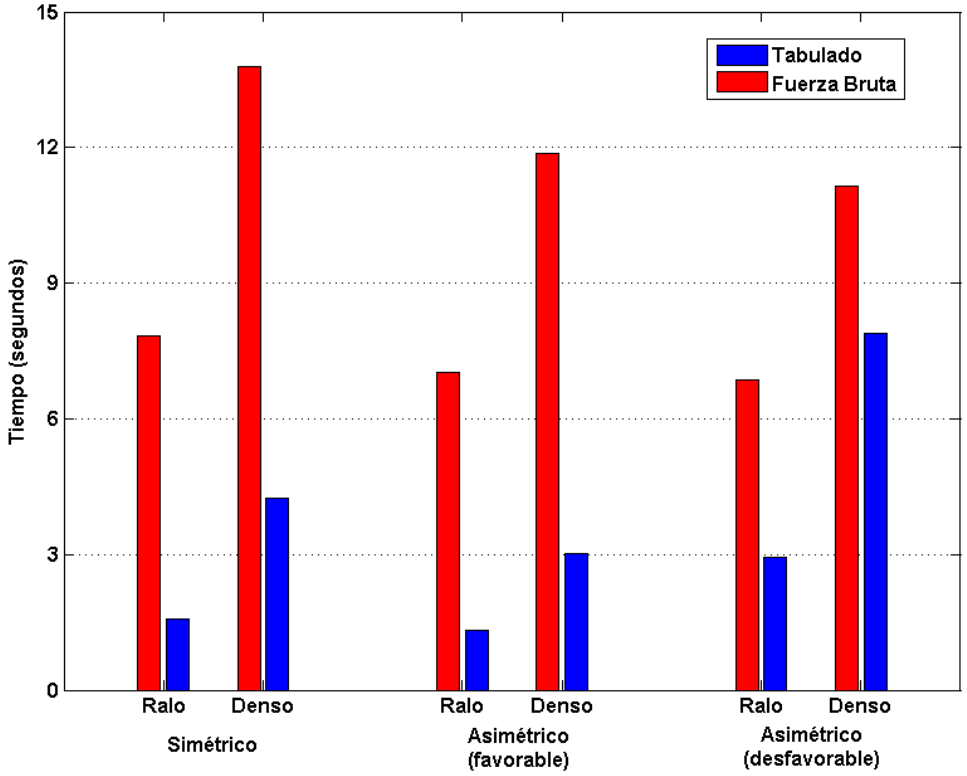


Figura 2.3: Tiempos con y sin tabulado de valores

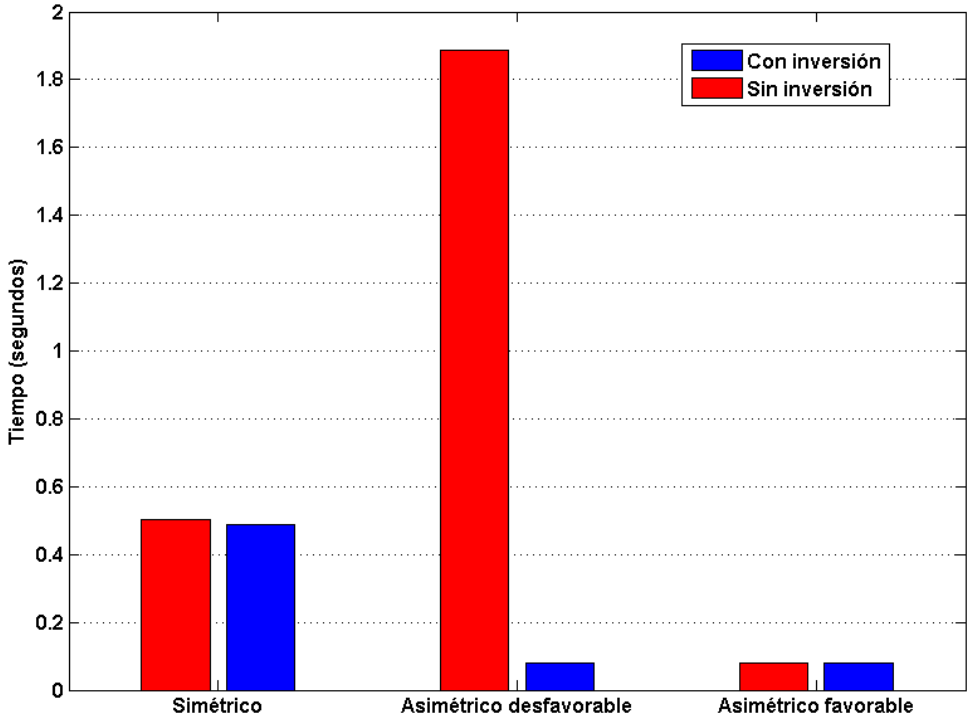


Figura 2.4: Tiempos con y sin inversión de particiones

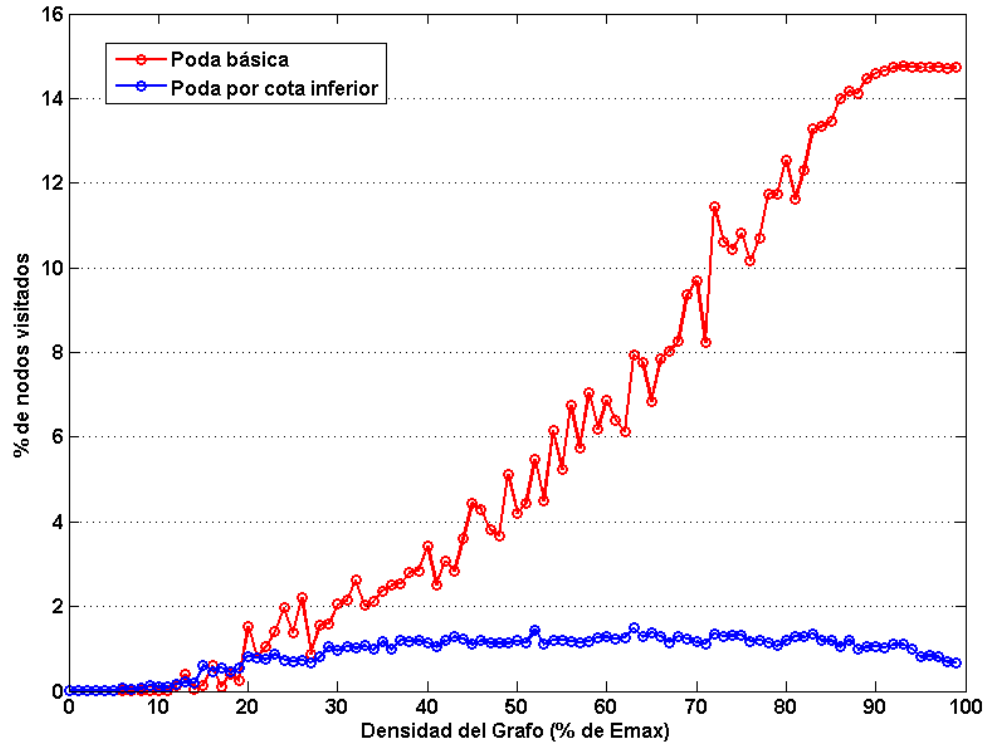


Figura 2.5: Comparación de podas

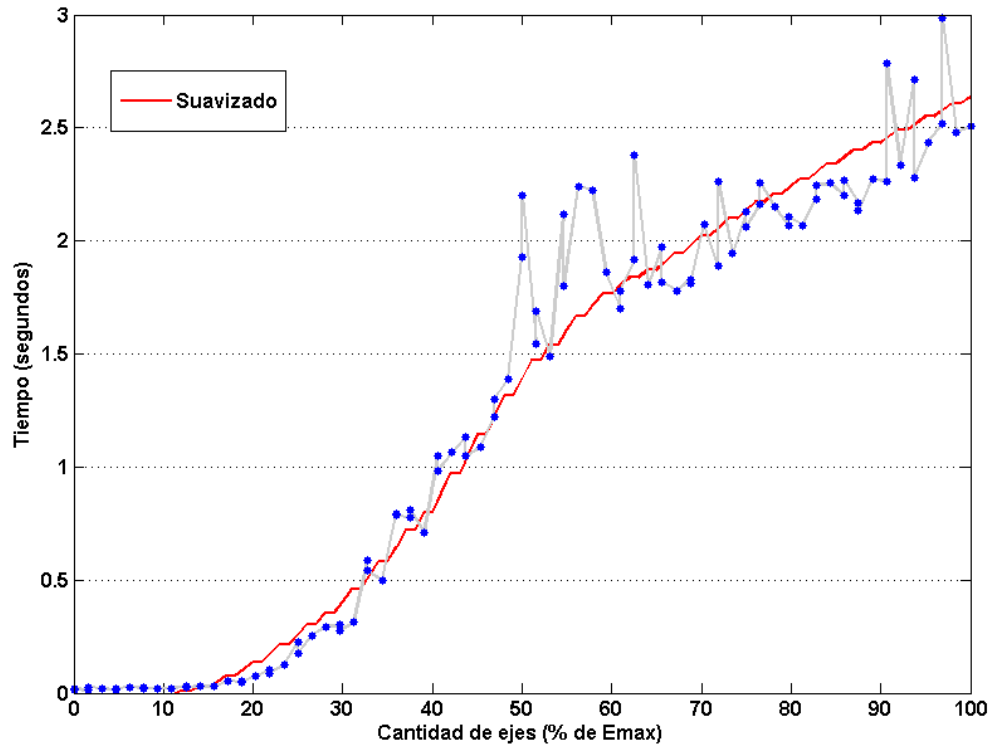


Figura 2.6: Tiempos en función de la densidad del grafo

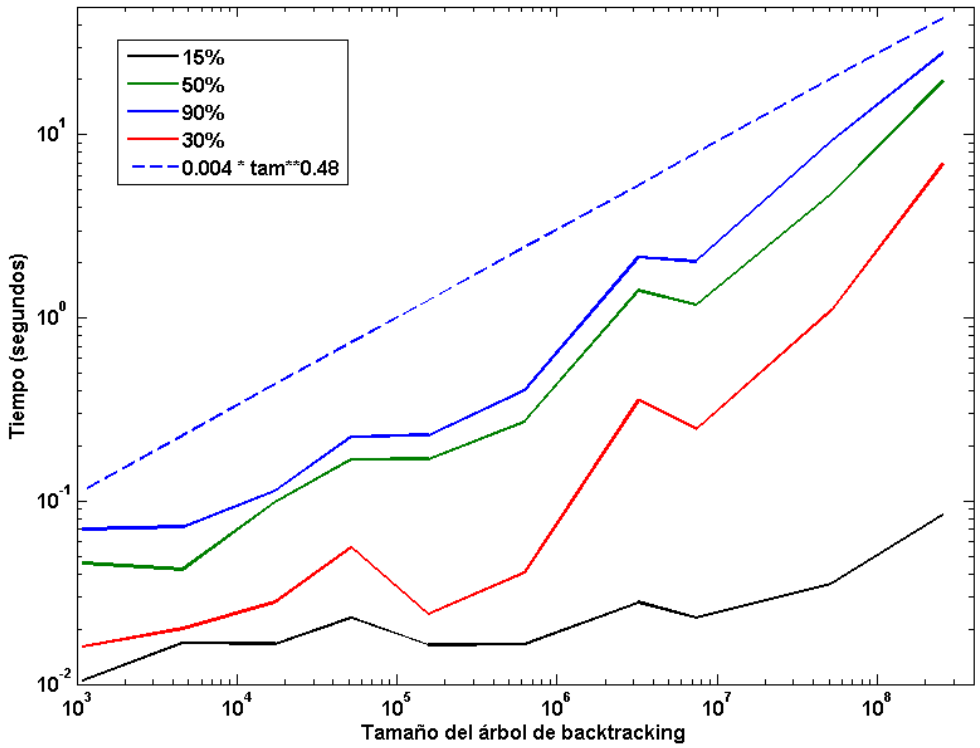


Figura 2.7: Tiempos en función del tamaño del árbol de *backtracking*

2.6. Discusión

Los gráficos iniciales son bastante ilustrativos de las mejoras realizadas al algoritmo. Resulta claro que tabular los valores de la función *crucesEntre* y utilizar conteos incrementales de los cruces redunda en un rendimiento superior del algoritmo, que insume entre 5 y 10 veces menos tiempo para resolver los casos que el algoritmo de fuerza bruta o backtracking trivial. Observamos también la diferencia que existe cuando el grafo es asimétrico en caso favorable y desfavorable (que corresponden a que la partición que se ubica primero en el árbol de *backtracking* es la que tiene más permutaciones o no). De todas maneras, en ambos casos el rendimiento fue mejor que el del algoritmo de fuerza bruta mostrando que el *overhead* de mantenimiento de tablas no niega los beneficios de tener los valores precalculados.

A continuación vemos la diferencia de tiempo sustancial que ocurre al invertir las particiones para que la que tiene menos permutaciones siempre se complete primero. El gráfico anterior sugería esto al comparar los casos favorable y desfavorable de instancias asimétrica. Esta optimización sencilla reduce a todos los casos asimétricos a la situación favorable y de ello se desprenden mejoras significativas de performance. Si bien no se aprecia en este gráfico, estas mejoras son mayores mientras más asimétrico es el grafo (ya que en general el algoritmo de la primera mitad del árbol es bastante más lento que el de la segunda mitad, como se discutió antes).

En el tercer gráfico se observa la eficiencia de las podas. Puede observarse que ambas podas tienen un rendimiento muy razonable, logrando en cualquier caso podas de al menos 85 % del árbol completo de *backtracking*. Sin embargo, la poda más sofisticada no solo tiene un comportamiento más constante (la sencilla poda cada vez menos según se incrementa la densidad del grafo), sino que además sus valores son mucho mejores, no podando nunca menos del 98 % del árbol. Esta poda por sí sola mejora el rendimiento del algoritmo en 2 órdenes de magnitud, lo cual representa una mejora importante.

Los últimos dos gráficos refieren al rendimiento de la implementación en C++. Rápidamente se observa que la densidad del grafo es un factor no despreciable en el tiempo de ejecución del algoritmo. Esto se debe a que las funciones de conteo de cruces insumen un tiempo que depende esencialmente de la cantidad de ejes del grafo, y el mayor tiempo insumido por las invocaciones a estas funciones se aprecia claramente en el rendimiento global del algoritmo.

Por último observamos el tiempo de ejecución global del algoritmo. Este gráfico se aproxima bastante a lo que uno podría esperar: el tiempo de ejecución del algoritmo está en relación directa con el tamaño del árbol de *backtracking*. Las irregularidades en la curva se deben a que, a fines de elegir valores uniformemente espaciados del tamaño del árbol, debieron usarse intermitentemente instancias simétricas y asimétricas que como vimos anteriormente tienen tiempos de ejecución característicos diferentes. Proponemos una función que acota el tiempo de ejecución del algoritmo: $0{,}004 * \sqrt{(tam)}$ donde *tam* es el tamaño del árbol a examinar. La constante puede variar dependiendo de la computadora que se use y las optimizaciones del compilador - el valor propuesto corresponde a G++ 4.3 con símbolos de *debugging* en un Pentium IV 2.8. El uso del flag -O3 de G++ produce un reducción por un factor de aproximadamente tres en los tiempos insumidos.

Si bien la tendencia de las curvas nos permite intuir que la cota puede no ser suficiente para instancias muy grandes, si puede dar una idea razonable del tiempo que será necesario darle al algoritmo para terminar, y en todo caso es de esperarse que la relación entre el tamaño del árbol y el tiempo insumido sea a lo sumo lineal.

Parte 3

Heurísticas Constructivas

3.1. Introducción

Presentamos a continuación tres diferentes ideas de heurística constructiva para el problema de dibujo de grafo bipartito. Comentaremos las ideas detrás de cada una así como un ejemplo de su aplicación y el pseudocódigo correspondiente, para luego examinar empíricamente su comportamiento a partir de prototipos en Python. Con esta información, elegiremos la de mejor comportamiento para su implementación definitiva en C++, que será la que utilicemos posteriormente en el algoritmo GRASP. Las heurísticas utilizan los métodos incrementales para conteo de cruces descriptos en 1.2.3.

En todos los casos se trata de heurísticas *greedy* que parten del dibujo a incrementar y agregan progresivamente los elementos nuevos (nodos, ejes o ambos) en la posición que optimiza algún criterio *greedy*.

3.2. Descripción de las heurísticas

3.2.1. Heurística de inserción de nodos

El primer enfoque que evaluamos consiste en partir del dibujo original (aquel cuyos nodos están en un orden que no puede ser alterado) y se completa progresivamente agregando los nodos nuevos en la posición óptima si se asume que el dibujo previo estaba fijo. Esto reduce el problema de obtener las mejores posiciones para cada nodo (en conjunto) a una sucesión de problemas en que hay que obtener la mejor posición para insertar únicamente un nodo, lo cual es mucho más simple desde un punto de vista de complejidad.

Esencialmente, se elige uno de los nodos móviles y se examina cuantos cruces genera insertarlo en cada una de sus posiciones posibles. A continuación se toma la posición que menos cruces produce y se lo inserta en ella, pasando el nodo a formar parte de los nodos fijos hasta el momento, y recomenzando el procedimiento hasta que no queden nodos por insertar.

La elección se hace alternativamente para una y otra partición (siempre que queden nodos móviles por insertar en cada partición, se elige e inserta uno de cada una, en lugar de insertar primero todos los que van a una partición, y a continuación los que van a la otra).

En el caso en que en el grafo incremental aparezcan ejes nuevos entre nodos que cuya posición ya estaba determinada, estos ejes se agregan de antemano al grafo ya que agregan información sobre los cruces que se producirán al agregar los demás nodos.

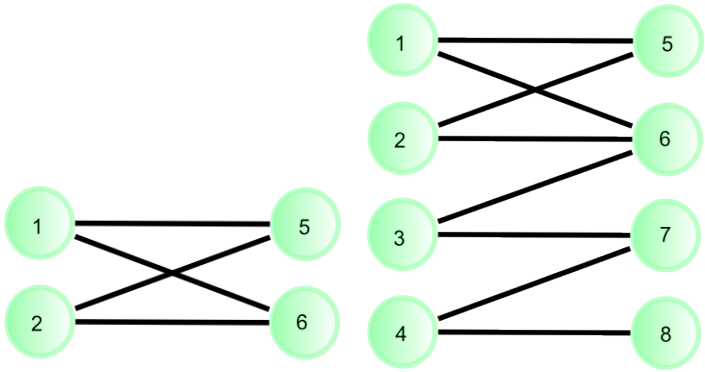
Resta eliminar la ambiguedad del orden en que se eligen los nodos móviles que se van a insertar. Consideramos tres variantes:

1. Escogerlo al azar
2. Escoger el nodo que tenga más adyacencias ya colocadas

3. Escoger el nodo que tenga menos adyacencias ya colocadas

Entendemos por cantidad de “adyacencias ya colocadas” a la cantidad de ejes que unen a un nodo móvil con nodos ya fijados del grafo (ya sea porque eran parte del dibujo original, o porque su posición ya fue establecida previamente por la heurística).

A modo de ejemplo, vamos a aplicar la heurística en sus distintas variantes al siguiente grafo:

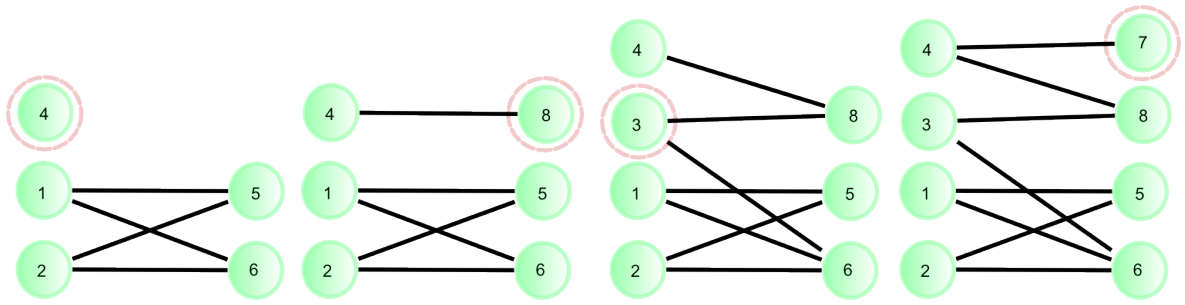


(a) (b)

Figura 3.1: Dibujo de partida y dibujo óptimo

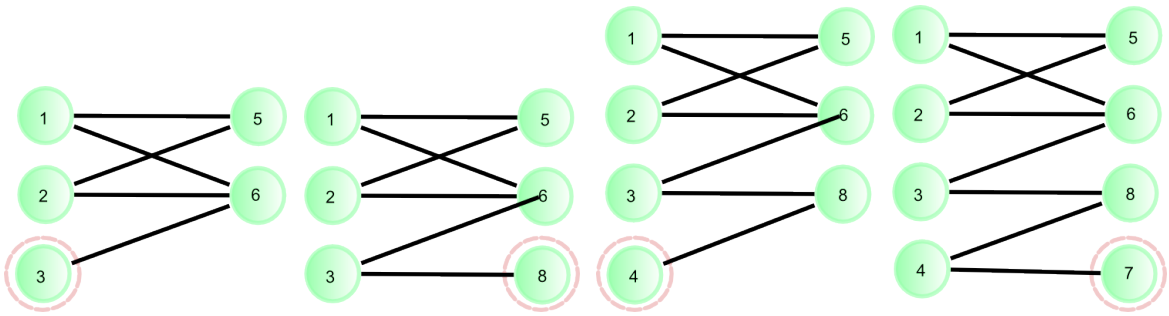
A continuación mostramos la ejecución de la heurística. Los nodos marcados con rojo son los que el algoritmo agrega en cada iteración.

- Inserción con selección aleatoria



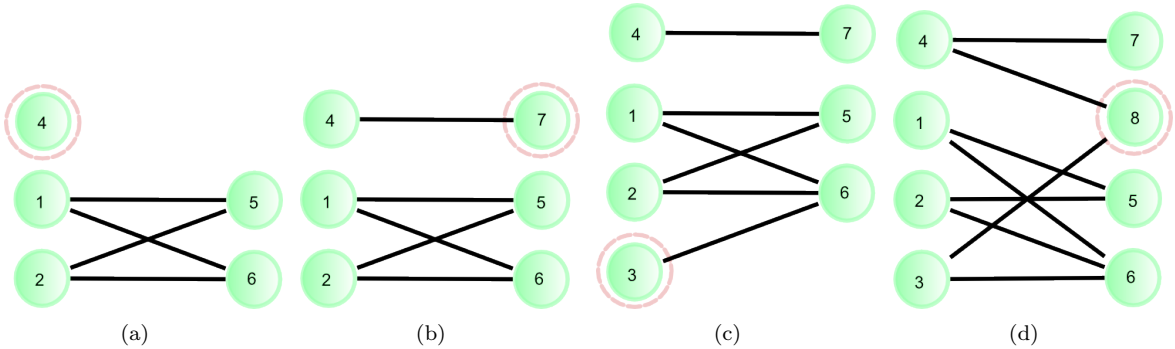
(a) (b) (c) (d)

- Inserción con selección por mayor grado



(a) (b) (c) (d)

- Inserción con selección por menor grado



Pseudocódigo

Algoritmo 6 Propone un dibujo mediante la inserción golosa de nodos

Parámetros: dibujo original a incrementar

```
1: cruces ← cruces del dibujo original
2: Mientras queden nodos por poner hacer
3:   Para cada partición donde queden nodos móviles por colocar hacer
4:     nodo ← elegir uno entre los nodos móviles
5:     sacar al nodo de entre los nodos a poner
6:     colocar al nodo en la ultima posición en su partición
7:     crucesNuevo ← cruces + cruces que se producen al agregar el nodo atrás de la particion
8:     mejorCruces ← crucesNuevo
9:     mejorPos = ultima posición
10:    Mientras no se revisaron todas las posiciones dentro de la partición hacer
11:      mover al nodo a la proxima posición {"swapear" al nodo con el que está en la posición anterior}
12:      crucesPreSwap ← cruces entre el nodo a insertar y el nodo anterior antes del swap
13:      crucesPostSwap ← cruces entre el nodo a insertar y el nodo anterior después del swap
14:      cruces ← cruces - crucesPreSwap + crucesPostSwap
15:      Si cruces < mejorCruces entonces
16:        mejorCruces ← cruces
17:        mejorPos ← la posición que estoy verificando
18:      Fin si
19:    Fin mientras
20:    poner al nodo en mejorPos
21:    cruces ← mejorCruces
22:  Fin para
23: Fin mientras
```

3.2.2. Heuristica de inserción de ejes

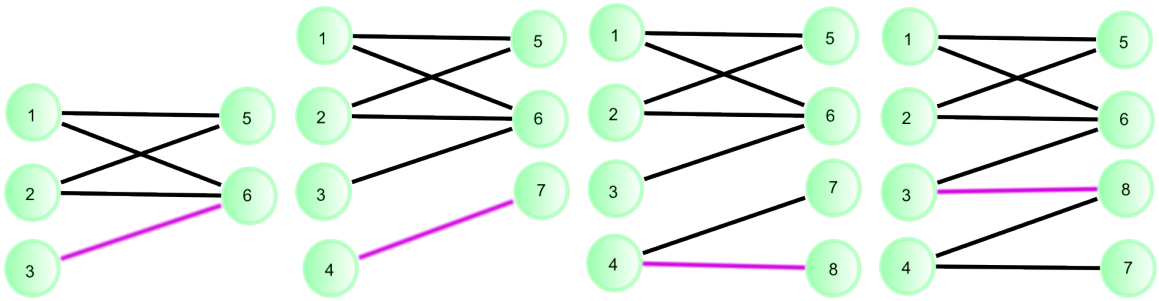
Una vez más partimos del dibujo original, pero en este caso procedemos agregando ejes: tomamos un eje "nuevo" y insertamos sus extremos en el par de posiciones válidas que minimiza el total de cruces en el dibujo obtenido. Entendemos por eje "nuevo" a aquellos ejes que tienen al menos uno de sus extremos no fijado de antemano. En el caso en que uno de los extremos del eje pertenezca al dibujo original (su posición relativa a los otros nodos de ese dibujo es fija), este nodo podría de todos modos tener más de una posición válida en el dibujo incrementado y por lo tanto se prueban todas las posiciones que mantienen el orden relativo original.

En cada iteración, si uno de los nodos del eje que se está insertando ya fue fijado por una pasada anterior del algoritmo, se lo extrae e inserta nuevamente. Si bien podría parecer que el algoritmo repite cálculos de forma redundante, esto no es necesariamente cierto puesto que se reinserta un nodo cuya posición ya se había establecido, el nuevo dibujo contiene más ejes que el que se había usado para tomar la decisión anterior, y por tanto la nueva decisión será más "informada" que la primera desde un punto de vista heurístico (lo cual no implica que sea mejor). Este tipo de decisión permite que la inserción

de un nuevo eje pueda incluso disminuir la cantidad de cruces respecto del dibujo anterior, cosa que era imposible en la heurística anterior (donde el número de cruces a lo sumo se mantiene igual en cada iteración, pero nunca mejora).

Por estas diferencias podríamos presuponer que esta heurística, al ser más sofisticada que la anterior, podría obtener mejores resultados. Sin embargo, el aumento de costo no es despreciable: para cada eje habrá que recorrer todos los pares de posiciones posibles para sus dos extremos observando cuantos ejes produce cada uno. Esto deberá ser tenido en cuenta cuando realicemos el análisis experimental.

Si aplicamos la heurística para el grafo de 3.2.1, obtenemos lo siguiente:



(a) (b) (c) (d)

Notemos como en el último paso, al agregar el eje $(3, 8)$ se mueve al nodo 8 de la posición que le había asignado antes, de modo de reducir la cantidad de cruces. Por otro lado observamos que si bien el algoritmo logró la solución óptima, esto se debió a que cuando existía ambiguedad entre varias posiciones donde poner a los nodos (debido a que todas ellas producían la misma cantidad de cruces), los puso abajo. Si hubiese elegido ponerlos arriba (opción válida, dado que genera la misma cantidad de cruces: 0) el resultado hubiera sido distinto.

Pseudocódigo

Algoritmo 7 Propone un dibujo mediante la inserción golosa de ejes

```
1: ejesPuestos ← los ejes del dibujo original
2: puesto[v_i] ← ¿v_i estaba en el dibujo original?
3: Para cada eje (x, y) a agregar hacer
4:   Si ya puse a x entonces
5:     sacarlo
6:   Si no
7:     puesto[x] = True
8:   Fin si
9:   Si ya puse a y entonces
10:    sacarlo
11:  Si no
12:    puesto[y] = True
13:  Fin si
14:  agregar el eje a ejesPuestos
15:  agregar x a la lista de adyacencia de y
16:  agregar y a la lista de adyacencia de x
17:  calcular los rangos en los cuales puedo mover x y y {si alguno estaba en el dibujo original, hay que respetar
     el orden relativo}
18:  insertar x en su primer posición válida
19:  insertar y en su primer posición válida
20:  mejoresCruces ← los cruces por ponerlos en esta posición
21:  mejorPosición ← posición actual
22:  Para cada posición válida para x hacer
23:    Para cada posición válida para y hacer
24:      contar los cruces por dejarlos en esa posición
25:      Si generan menos cruces que mejoresCruces entonces
26:        mejoresCruces ← cruces por tenerlos en esta posición
27:        mejorPosicion ← posición actual
28:      Fin si
29:      mover y a su proxima posición
30:    Fin para
31:    mover x a su próxima posición válida
32:    mover y a su primer posición válida
33:  Fin para
34:  mover x y a y a la mejorPosicion
35: Fin para
```

3.2.3. Heurística de inserción de nodos por mediana

La idea general de esta heurística es buscar que ningún nodo este demasiado "lejos" de sus adyacentes. Como criterio heurístico para lograr esto, utilizamos como posición de un nodo la mediana de las posiciones de sus adyacentes.

El procedimiento es el siguiente: en un principio se comienza con solamente los nodos que ya estaban en el dibujo original acompañados de sus ejes. Tomamos entonces al nodo de mayor grado (con respecto a los nodos que ya están puestos, análogamente a la primera heurística), calculamos la mediana de las posiciones de sus adyacentes y una vez que la obtenemos, probamos insertar al nodo en la posición correspondiente a la mediana obtenida, o en las posiciones inmediata anterior o posterior a ésta, eligiendo de las tres a la que genere menos cruces en el dibujo. En el caso en que la mediana no sea un índice válido (porque una partición tiene menos nodos que la otra, y el valor de la mediana se calcula a partir de la posición de los nodos de la más grande, y por tanto la mediana podría ser mayor que el tamaño de la partición más chica) la truncamos. Repetimos el procedimiento iterativamente hasta que están puestos todos los nodos.

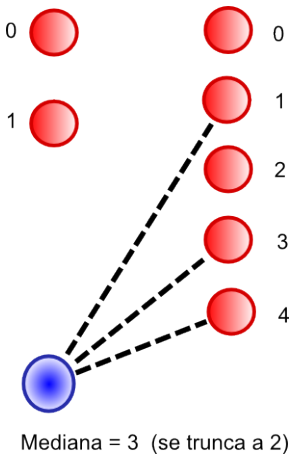


Figura 3.2: ejemplo de inserción por mediana con truncamiento

Al igual que en la heurística de inserción de nodos, si había ejes a agregar entre los nodos que ya estaban, estos se agregan al inicio del algoritmo para dar más información a las posteriores inserciones.

Si aplicamos la heurística a 3.2.1 lo que se obtiene es, en este caso, la misma secuencia que en la heurística de inserción *greedy* de nodos. Esto se debe a que siempre se agregan nodos que tienen un único adyacente, y éste está ubicado al final de la partición vecina.

Esta heurística utiliza un criterio *greedy* que podríamos calificar de indirecto: las otras heurísticas son *greedy* en el sentido que minimizan el número de cruces de cada inserción (minimizan localmente el mismo criterio que debe minimizarse globalmente), mientras que la de la mediana utiliza como criterio local la distancia a la que queda cada nodo de sus adyacentes. Es una heurística similar a la del baricentro, que en el caso del problema de dibujo de grafos bipartitos sin la característica de ser incrementales se comporta de forma muy favorable.

Como mejora adicional, Luego de ubicar a todos los nodos, se hace una pasada en cada partición intercambiando nodos en posiciones consecutivas si esta permutación disminuye el número total de cruces del dibujo. Esto se hace con el objetivo de paliar los problemas originados por el truncamiento de los valores de las medianas descripto anteriormente.

Pseudocódigo

Algoritmo 8 Propone un dibujo mediante inserción por la mediana de los adyacentes

```
1:  Mientras queden nodos sin poner hacer
2:    elegir un nodo de grado máximo con respecto a los que ya están puestos
3:    calcular la mediana de las posiciones de sus adyacentes
4:    Si mediana > tamaño actual de la partición entonces
5:      mediana ← tamaño de la partición
6:    Fin si
7:    Para  cada i = mediana-1, mediana, mediana+1 hacer
8:      Si es una posición válida entonces
9:        contar los cruces por ponerlo en esa posición
10:       Si lo inserté por primera vez o me genera menos cruces que la mejorPosicion entonces
11:         mejorPosicion ← posicionActual
12:       Fin si
13:     Fin si
14:   Fin para
15:   poner al nodo en mejorPosicion
16: Fin mientras
```

3.3. Comparación de las heurísticas constructivas

A fin de decidir cual o cuales de estas heurísticas se comporta mejor, y al igual que con el algoritmo exacto, decidimos hacer primero una implementación en Python por las facilidades que da el lenguaje en cuanto a velocidad de desarrollo. Utilizando estas implementaciones, aplicamos las heurísticas a numerosos casos de prueba y comparamos los resultados, teniendo en cuenta no solo la calidad de los resultados obtenidos sino también el costo temporal.

A priori lo que esperamos es que la heurística de inserción de ejes de mejores resultados por el hecho de que reinserta nodos, lo cual le da varias oportunidades para fijar la posición de un nudo dado, por lo cual podría corregir errores cometidos por insertar cuando todavía había pocos nodos en el dibujo.

Sin embargo, es de esperarse que este método sea considerablemente más lento que los demás: en primer lugar, porque itera tantas veces como ejes se agreguen, lo cual podría ser $O(n^2)$ siendo n la cantidad de nodos, y en segundo lugar porque cada una de estas iteraciones requiere de $O(n^2)$ conteos de cruces. Si a esto le sumamos el costo de contar los cruces vemos que el costo de esta heurística es bastante elevado.

Respecto de la heurística de la mediana, resulta difícil prever como se comportará el algoritmo. Si bien es de esperarse que tenga un costo temporal reducido (ya que solo "prueba" tres posiciones para cada nodo y por lo tanto realiza pocos conteos de cruces), su performance no es fácil de predecir.

Finalmente, respecto a la inserción *greedy* de los nodos, creemos que su costo será menor que el de la inserción de ejes, pero sus resultados podrían no ser tan buenos en razón de su simpleza inherente.

Ejecutamos los siguientes tests:

1. Comparación de Heuristicas de inserción *greedy* de nodos: Primero comparamos a las diferentes formas de elegir al nodo candidato, para observar si alguna de las formas de hacerlo se desempeña mejor.
2. Comparación entre heurísticas:
   a) n nodos en cada partición, n creciente, $\frac{n^2}{2}$ ejes, 60 % de nodos nuevos
   b) n nodos en cada partición, n creciente, $\frac{n^2}{2}$ ejes, 40 % de nodos nuevos
   c) n nodos en cada partición, n creciente, $3n$ ejes, 60 % de nodos nuevos
   d) n nodos en cada partición, n creciente, $3n$ ejes, 40 % de nodos nuevos
   e) $n = 30$, cantidad de ejes creciente, 40 % de nodos nuevos

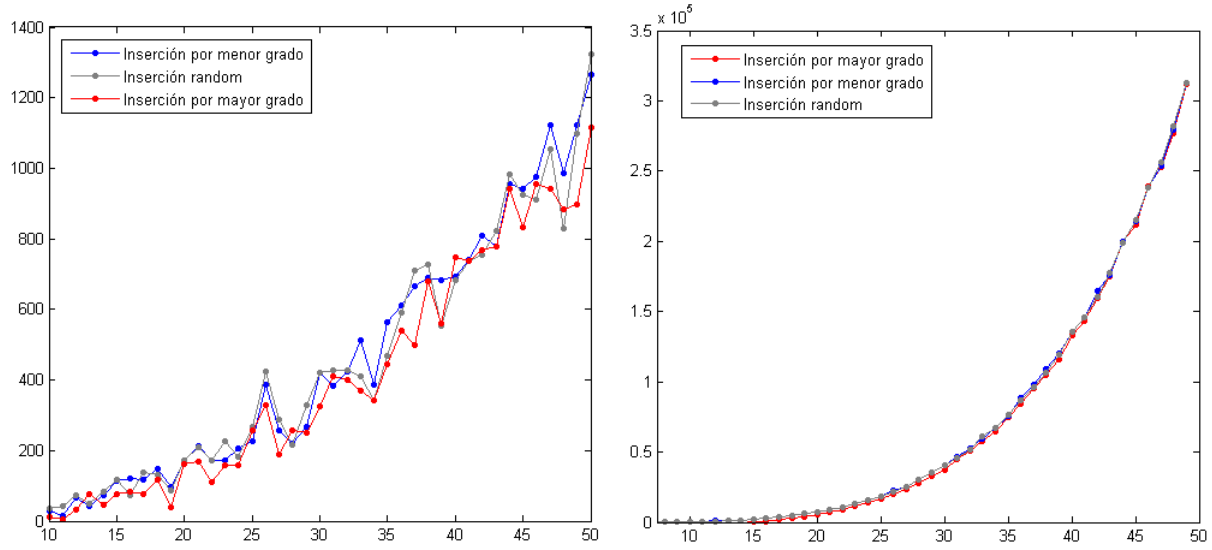
En cada uno de ellos se midió sobre grafos aleatorios de los tamaños mencionados la cantidad de cruces lograda y el tiempo empleado para lograr el dibujo.

Si bien los tiempos de ejecución en un lenguaje interpretado como Python, son por lo general bastante mayores que los tiempos de ejecución en C++, consideramos que son igualmente válidos como herramienta de comparación para observar una tendencia general en el comportamiento de las heurísticas. Por otro lado, dado que implementar en este lenguaje nos resulta mucho más sencillo, consideramos que vale la pena probar a tres heurísticas experimentalmente en lugar de proponer una única para implementar en C++.

3.3.1. Criterios de selección de nodos para la heurística de inserción de nodos

Las pruebas que realizamos consistieron en aplicar la heurística de inserción de nodos a grafos aleatorios variando la cantidad de nodos en cada partición.

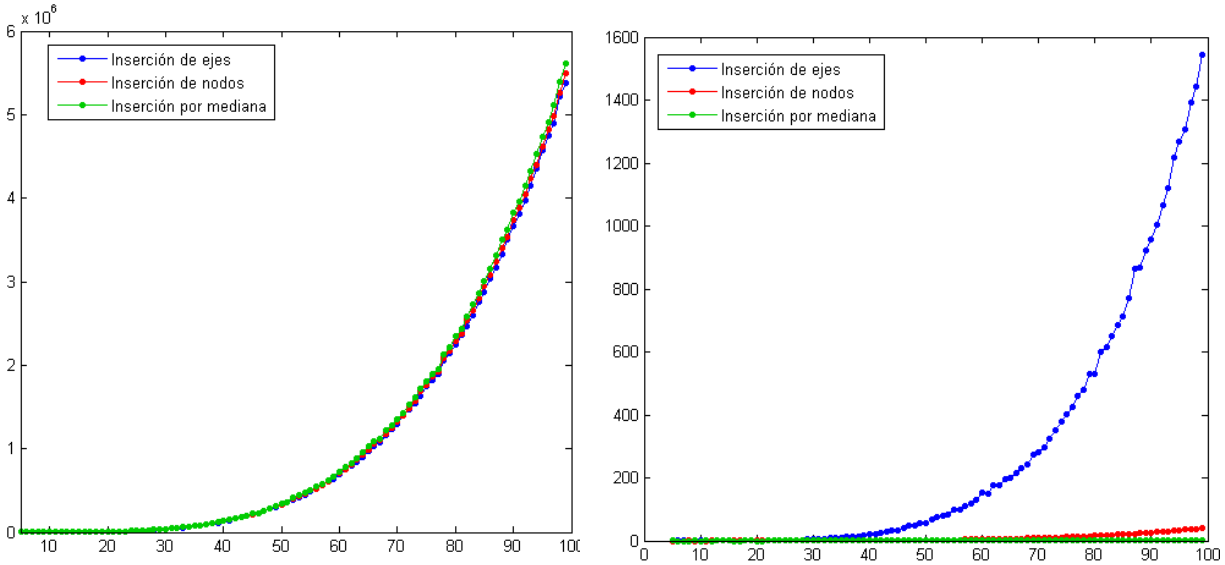
En la primer experiencia utilizamos grafos con $m = 2 * n$ y un 40 % de nodos fijos (en adelante n es la cantidad de nodos de cada partición). En la segunda experiencia, la cantidad de ejes fue $m = \frac{n^2}{2}$ y el porcentaje de nodos fijos fue también del 40 %.



(a) (b)

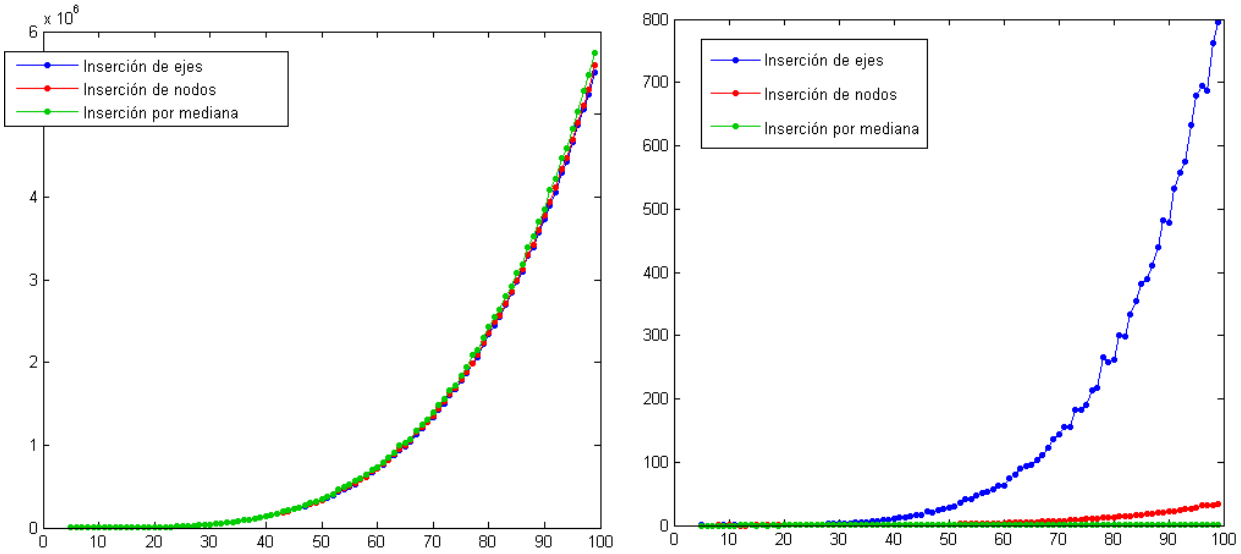
Observamos en estas experiencias que la cantidad de cruces encontrada por los tres métodos es relativamente similar. Sin embargo el criterio de mayor grado parecería comportarse ligeramente mejor que los otros dos. Para nosotros tiene sentido que esto sea así, ya que si se utiliza el nodo de mayor grado con respecto a lo que ya está puesto, ese nodo tiene más información (mas adyacentes puestos) por lo que sería razonable que pueda ubicarse mejor. Está claro que podría fallar, sin embargo a partir de esta idea, más lo que se observa en las pruebas, decidimos utilizar al nodo de mayor grado para la heurística de inserción de nodos.

3.3.2. Comparación de heurísticas constructivas



(a) Cruces producidos en función de n (b) Tiempo en segundos en función de n

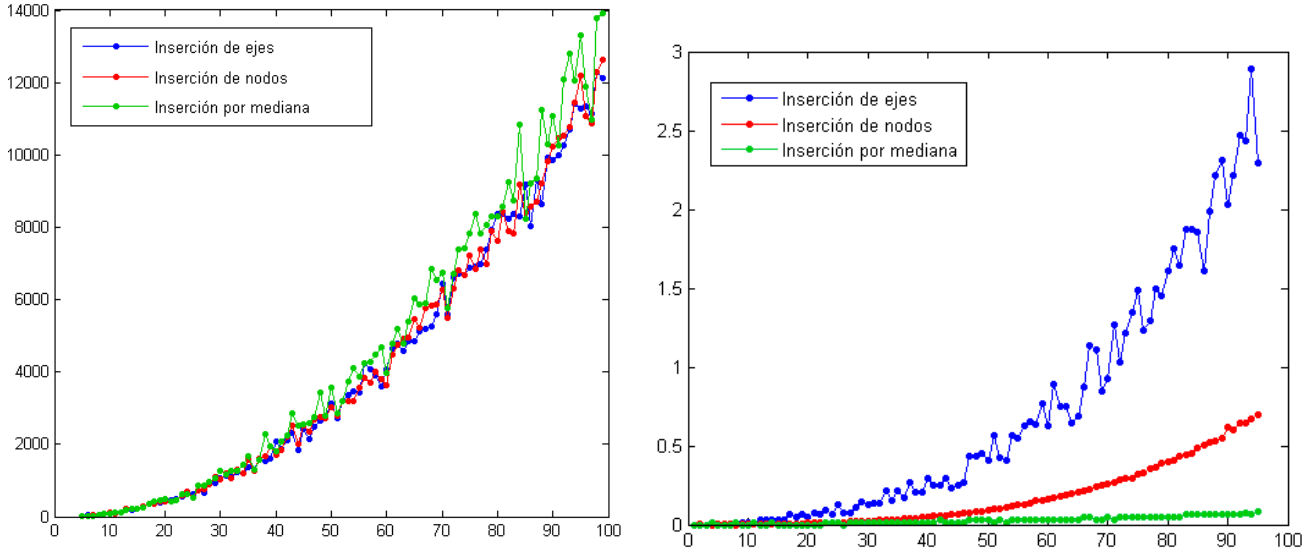
Figura 3.3: n nodos en cada partición, n creciente, $\frac{n^2}{2}$ ejes, 60 % de nodos nuevos



(a) Cruces producidos en función de n

(b) Tiempo en segundos en función de n

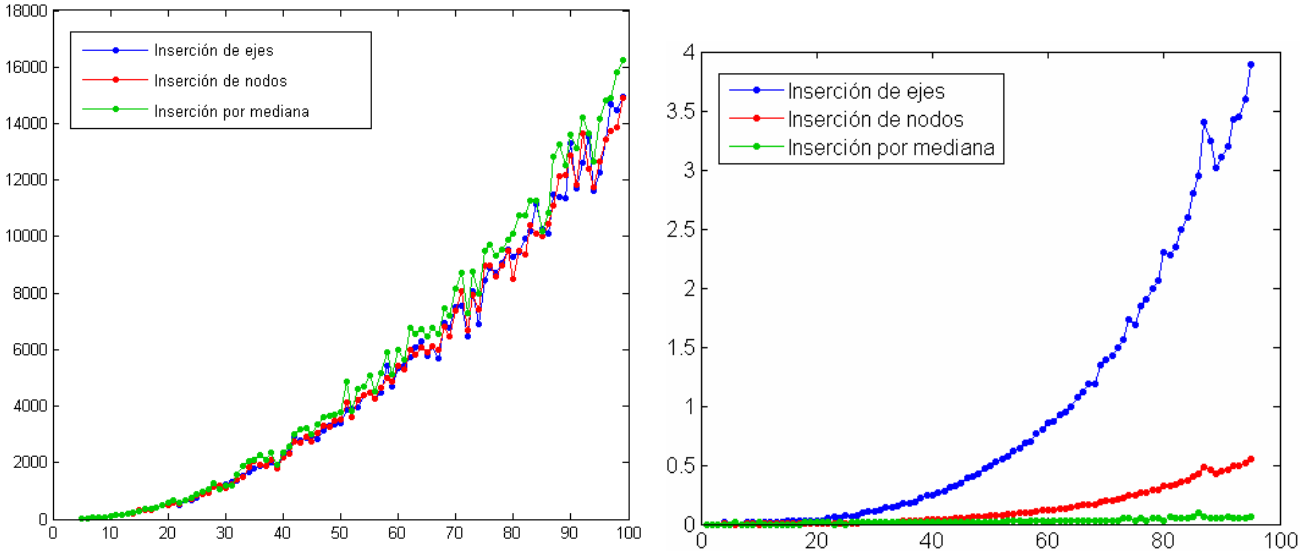
Figura 3.4: n nodos en cada partición, n creciente, $\frac{n^2}{2}$ ejes, 40 % de nodos nuevos



(a) Cruces producidos en función de n

(b) Tiempo en segundos en función de n

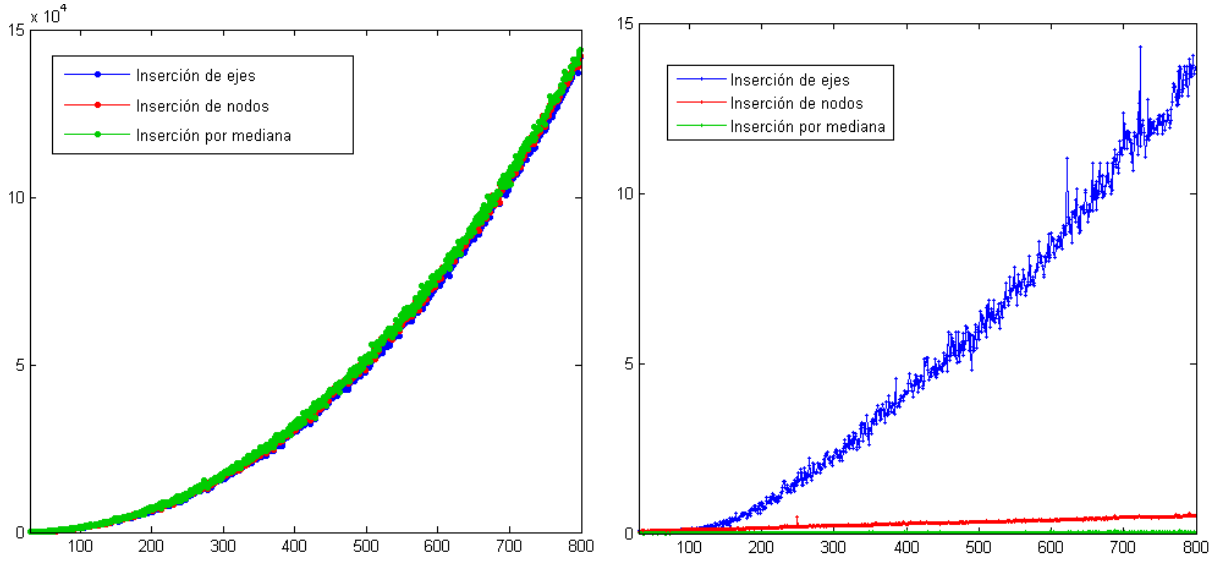
Figura 3.5: n nodos en cada partición, n creciente, $3n$ ejes, 60 % de nodos nuevos



(a) Cruces producidos en función de n

(b) Tiempo en segundos en función de n

Figura 3.6: n nodos en cada partición, n creciente, $3n$ ejes, 40 % de nodos nuevos



(a) Cruces producidos en función de m

(b) Tiempo en segundos en función de m

Figura 3.7: $n = 30$, m creciente, 40 % de nodos nuevos

3.4. Análisis de los resultados

Al observar los gráficos de las experiencias lo primero que salta a la vista es que el tiempo de ejecución de la heurística de inserción de ejes es mucho más grande que el de las demás. Esta situación, que se hace más notoria en grafos densos, hace que su uso no sea recomendable, más aún si tenemos en cuenta como muestran las demás experiencias que los resultados que obtiene no son significativamente mejores que los de las otras. En la experiencia 5 vemos como para un grafo con 30 nodos y 799 ejes la diferencia entre la inserción de nodos y la inserción de ejes de 442 cruces a favor de la inserciόon de ejes (142131 contra 141689), lo cual es sólo un 0.3 %, pero los tiempos fueron de 0.5160 y 13.6870 segundos respectivamente, lo cual es aproximadamente 26 veces más. En función de esta ineficiencia temporal, si bien los resultados que ofrece son buenos, decidimos descartar la heurística de inserción de ejes.

Por otro lado, la heurística de la mediana se muestra como la más rápida, como habíamos estimado previamente. Sin embargo en cuanto a la cantidad de cruces suele dar peores resultados que las otras dos. Es por esto, que decimos entonces

descartar la heurística de la mediana por obtener resultados de peor calidad sin un ahorro sustancial de tiempo de ejecución.

Finalmente elegimos implementar en C++ la heurística de inserción de nodos, ya que consideramos que de las tres alternativas planteadas obtiene un buen compromiso entre calidad de los resultados y tiempos cortos de ejecución.

3.5. Detalles de implementación

La implementación al igual que en el caso del algoritmo exacto, utiliza diversos contenedores de la STL. Si se observa el código en C++ hay algunas modificaciones respecto del algoritmo presentado hasta el momento que responden a los cambios realizados para agregar factores aleatorios a las soluciones propuestas por la heurística. Estas modificaciones son necesarias para la implementación de GRASP y son detalladas en 5.2.

Como particularidad de implementación se puede mencionar el uso de `std :: sort` para ordenar la secuencia de nodos por insertar según su grado de adyacencias (si solo se debe tomar el máximo esto es ineficiente, pero en el contexto del GRASP donde deben tomarse los k primeros elementos es lo más conveniente). Para este fin se utiliza un objeto adaptador que es el responsable de las comparaciones entre elementos según su grado.

3.6. Cálculo de complejidad

Antes de comenzar, realizamos una limpieza del grafo, para no considerar a los nodos que tienen grado nulo. Como comentamos en el apartado 1.3, esto tiene un costo $O(V_1 + V_2 + m)$ donde V_i es la cantidad de nodos de partición i sin filtrar y m la cantidad de ejes. Este costo se deberá sumar a la complejidad total.

La heurística de inserción de nodos comienza asignando variables y atributos útiles para la reutilización de cálculos. Inicializamos dos listas de nodos fijos y dos de nodos a agregar (un par para cada partición), y los diccionarios de adyacencias parcial, grado parcial, posiciones y nodos móviles. Todo esto tiene un costo de $O(v_{max} + m)$. Además, en esta parte inicializamos la variable *cruces*, contando los cruces del dibujo original (fijo) sin agregar ningún nodo, lo que nos cuesta $O(m*log(fijos_{max})+fijos_{max}))$. Finalmente la parte de inicialización nos cuesta $O(n+m+m*log(fijos_{max})+fijos_{max})$, con m su cantidad de ejes, y $fijos_{max}$ la cantidad de nodos fijos de la partición que tiene más nodos fijos.

Veamos ahora cuanto nos cuesta elegir un nodo entre los nodos a agregar. Primero ordenamos de forma creciente (con sort de STL) los nodos a agregar por su grado parcial (considerando solamente los ejes que van del nodo hacia un nodo fijo o del nodo hacia un nodo que ya fue agregado al dibujo en iteraciones previas). Este ordenamiento tiene complejidad $O(cantMovilesPi * log(cantMovilesPi))$. Si bien para tomar el máximo valor no es necesario ordenar, las extensiones futuras vinculadas con la aleatorización del algoritmo si requieren de disponer de la secuencia ordenada. Por esta razón, consideramos este costo como una cota de peor caso de todas las variantes del algoritmo.

Luego de ordenarlos, tomamos el primero de la secuencia. Luego estos pasos tienen un costo $O(cantMovilesPi * log(cantMovilesPi))$, siendo Pi la partición sobre la cual estamos agregando el nodo.

Ahora buscamos una de las mejores posiciones en la partición para insertar este nodo. Primero actualizamos la lista de adyacencias parciales para incorporar las adyacencias del nodo que voy a agregar (esto es necesario pues las funciones de conteo de cruces se usan dentro del subgrafo fijo y por tanto para que tengan en cuenta al nodo a agregar, es necesario completarlas con sus ejes) y luego insertamos el nodo al final de la partición.

Actualizar la lista de adyacencias parcial tiene un costo $O(m_a)$, con m_a los ejes del nodo a agregar. Sin embargo, una vez que se agregan todos los nodos, en el peor de los casos se agregan todos los ejes también, por lo que esto tiene un costo $O(m)$. Luego contamos los cruces por agregar atrás, lo cual tiene un costo $O(v_{max} + m)$. Ahora *swapeamos* el nodo con su nodo anterior de la partición y recalculamos los cruces hasta que el nodo a agregar queda primero en la partición. Aquí podemos utilizar lo mostrado en el apartado que explica el conteo de cruces, y lo hacemos con costo $O(min(max(vi, m_a, m_b), m_a * m_b))$. Repetir este proceso por cada posición en la partición que estamos agregando, tiene un costo de $O(cantModosParticion * min(max(vi, m_a, m_b), m_a * m_b))$. Además, $max(v_i, m_a, m_b) \leq v_{max}$, puesto que m_a y m_b son a lo sumo tan grandes como la partición mas grande (en el caso en el que los nodos estén relacionados con todos los de la partición de enfrente) y también vale $min(max(vi, m_a, m_b), m_a * m_b) \leq max(vi, m_a, m_b)$. De esto resulta que la complejidad de este ciclo nos queda $O(v_{max}^2)$.

Finalmente, insertamos el nodo y actualizamos los grados parciales y el vector de posiciones, con un costo de $O(m+v_{max})$.

Dado que este procedimiento lo hacemos para todos los nodos a agregar, la complejidad resultante es:

$$O(cantMoviles*(\underbrace{m+v_{max}}_{a}+\underbrace{v_{max}^2}_{b}+\underbrace{Moviles+MovilesP1*log(MovilesP1)+MovilesP2*log(MovilesP2))}_{c}+\underbrace{m*log(p_{max})+p_{max}}_{d})(*$$

- a es por agregar al nodo atrás de la partición
- b es por moverlo por toda la partición mediante *swaps*
- c es por obtener cada vez uno de grado máximo
- d es por obtener por primera vez los cruces

Pero $* \subseteq O(Moviles * v_{max}^2 + m * log(fijos_{max}) + fijos_{max})$, siendo Moviles la cantidad total de nodos a agregar, v_{max} la cantidad de nodos de la partición más grande del dibujo resultante, m la cantidad de ejes del dibujo resultante y p_{max} la cantidad de nodos fijos de la partición que más nodos fijos tiene.

Esto es así porque:

$O(m + v_{max}) \subseteq O(Moviles * v_{max}^2)$ dado que $m \leq v_{max} * v_{max}$

$O(MovilesPi + MovilesPi * log(MovilesPi)) \subseteq O(MovilesPi * log(MovilesPi))$, y como $MovilesPi * log(MovilesPi) \leq v_{max} * log(v_{max}) \leq v_{max} * v_{max}$, vale que $O(MovilesPi + MovilesPi * log(MovilesPi)) \subseteq O(v_{max}^2)$

Finalmente, a esta complejidad obtenida debemos agregarle el costo de la "limpieza" del grafo. Por lo tanto, la complejidad total es:

$$O(Moviles * v_{max}^2 + m * log(fijos_{max}) + fijos_{max} + V_1 + V_2 + m)$$

El tamaño de la entrada t, lo podemos definir de la siguiente manera:

$$t = log(P_1) + \sum_{i=1}^{P_1} log((p_1)_i) + log(P_2) + \sum_{i=1}^{P_2} log((p_2)_i) + log(m_p) + \sum_{i=1}^{m_p} log((e_i)_0) + log((e_i)_1)$$
$$+log(IV_1) + \sum_{i=1}^{IV_1} log((iv_1)_i) + log(IV_2) + \sum_{i=1}^{IV_2} log((iv_2)_i) + log(m_{iv}) + \sum_{i=1}^{m_{iv}} log((e'_i)_0) + log((e'_i)_1)$$

donde P_i es la cantidad de nodos originales de la primera partición, m_p es la cantidad de ejes originales, IV_i es la cantidad de nodos que se agregan a la partición i y m_iv es la cantidad de ejes que se agregan.

Luego vale que:

$$t \geq log(P_1) + v_1 + log(P_2) + V_2 + log(m_p) + m + log(IV_1) + log(IV_2) + log(m_{iv})$$

A partir de esto, vemos que:

$$t \geq V_i \geq v_i$$
$$t \geq V_{max} \geq v_{max}$$
$$t \geq Moviles$$
$$t \geq fijos_{max}$$
$$t \geq m$$

Y por lo tanto resulta que el orden de toda la heurística es:

$$O(Moviles * v_{max}^2 + m * log(fijos_{max}) + fijos_{max} + V_1 + V_2 + m) \subseteq O(t^3 + t * log(t) + t) \subseteq O(t^3)$$

3.7. Análisis de la heurística

3.7.1. Casos patológicos

Para ver qué tan malo podría ser el comportamiento de nuestra heurística, intentamos buscar casos donde el resultado que proponga el algoritmo diste arbitrariamente de la solución óptima.

Un caso donde la heurística se equivoca es el siguiente:

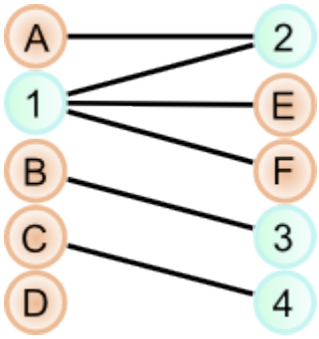
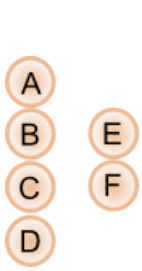


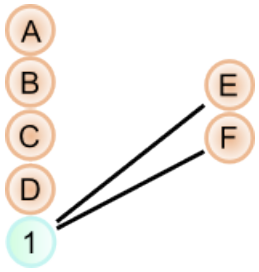
Figura 3.8: Caso patológico para la heurística constructiva

En este ejemplo los nodos numerados son los nodos que se agregan, mientras que los que tienen letras son los del dibujo original.

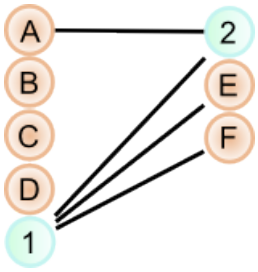
Veamos qué hace la heurística constructiva frente a este caso:



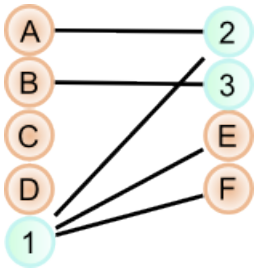
(a) Partimos del dibujo original



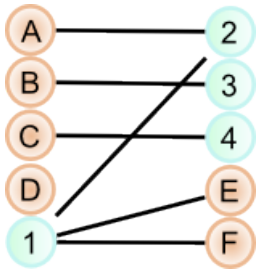
(b) El 1 tiene mayor grado hacia lo que ya está. Como sirve cualquier posición, lo insertamos en la primera que probamos (recordar que vamos desde atrás hacia adelante)



(c) La mejor posición para el nodo 2 es arriba de todo ya que no genera cruces



(d) El nodo 3 está obligado a colocarse en una posición que genera un cruce



(e) Ocurre lo mismo para el nodo 4

Como vemos, en este caso, frente a un grafo para el cual existe un dibujo sin ningún cruce, nuestra heurística obtiene un dibujo con 2 cruces. Ahora bien, si tuviéramos un nodo nuevo más que estuviera relacionado con D, el dibujo óptimo seguiría teniendo 0 cruces, y sin embargo nuestra heurística daría 3 cruces. Esto puede generalizarse: en general si a este grafo le agregamos nodos fijos debajo de D y nodos nuevos unidos a éstos (con grado 1), el número de cruces óptimo sigue siendo 0, pero nuestra heurística va a proponer un dibujo con una cruz más por cada par de nodos que se agregue.

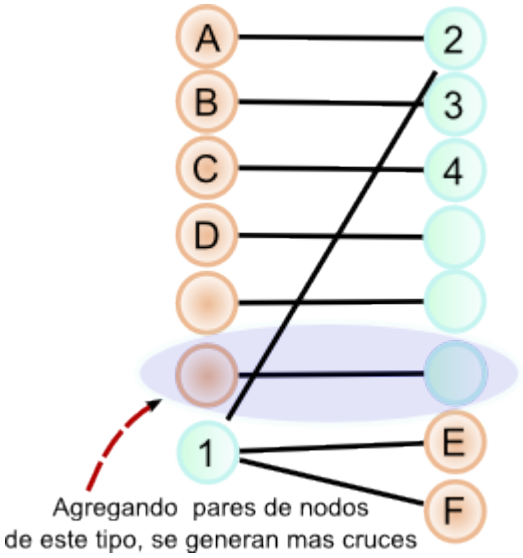


Figura 3.9: Familia de casos patológicos

Luego, la cantidad de cruces del dibujo propuesto por la heurística constructiva es k, donde k es la cantidad de pares de nodos de la forma nodo viejo - nodo nuevo que se agregan al dibujo original.

Veamos gráficamente la cantidad de cruces que encuentra la heurística en función de la cantidad de nodos:

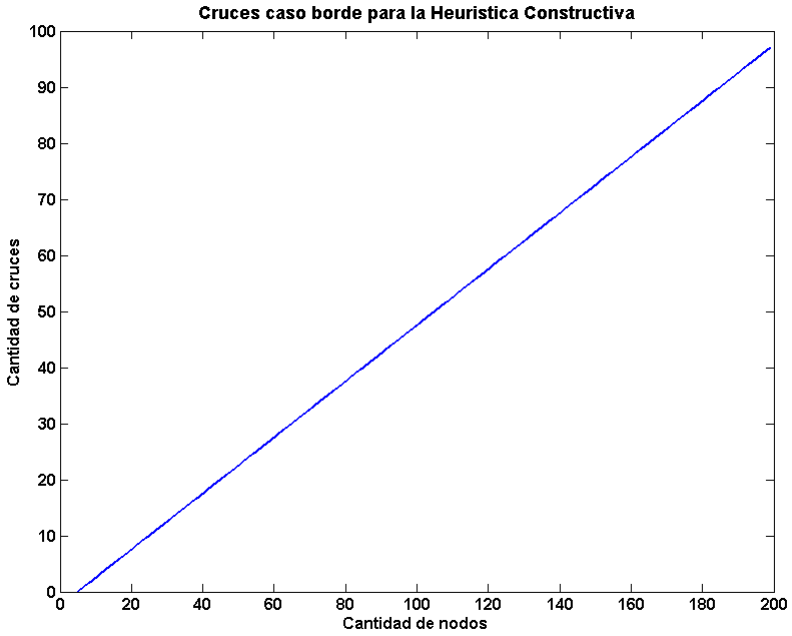
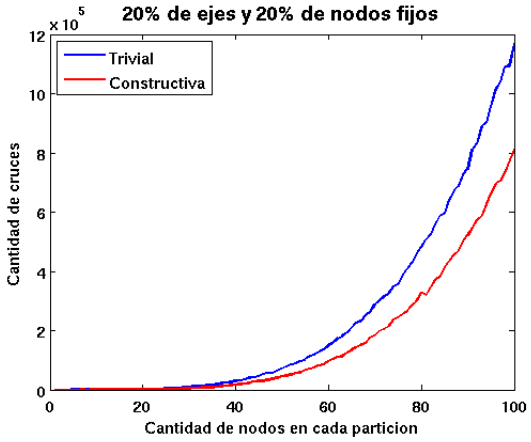


Figura 3.10: Cantidad de cruces al aplicar la heurística a grafos de la familia presentada

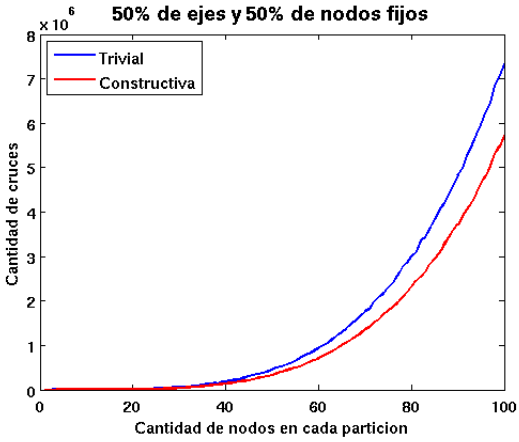
3.7.2. Comparación con la heurística trivial

La heurística que elegimos se comporta bien en cuanto a tiempo y cruces si la comparamos con las otras heurísticas planteadas. A modo de referencia nos pareció interesante comparar su comportamiento con el de la heurística optimista, consistente en devolver el dibujo agregando los nodos nuevos al al final dl dibujo en el orden en que fueron recibidos, asumiendo que se trata de la solución óptima.

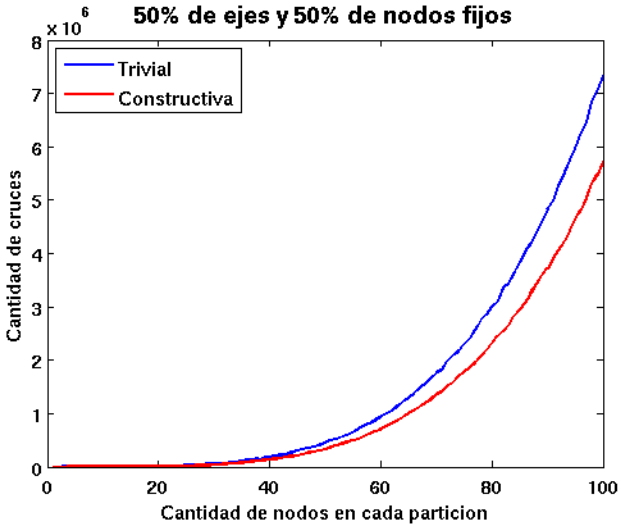
Esta experiencia nos permitirá ver si por lo menos vale la pena el tiempo gastado en construir la solución. A continuación se muestran los resultados:



(a)



(b)



(c)

Figura 3.11: Comparación entre inserción *greedy* de nodos y heurística optimista

Como vemos, nuestra heurística se muestra considerablemente mejor que el acercamiento optimista, y esta mejora se hace más clara en grafos densos y con un número alto de nodos.

3.7.3. Tiempo de ejecución

Para observar el comportamiento decidimos medir los tiempos variando distintos parámetros en los grafos que utilizamos como entrada para la heurística, con la idea de ver qué influencia tienen en el tiempo de ejecución del algoritmo las variaciones de la cantidad de nodos, ejes y nodos nuevos.

Nuestra primera experiencia consistió en utilizar grafos aleatorios con n nodos en cada partición. Variamos este n con distintos porcentajes de nodos móviles y ejes. Los resultados son los siguientes:

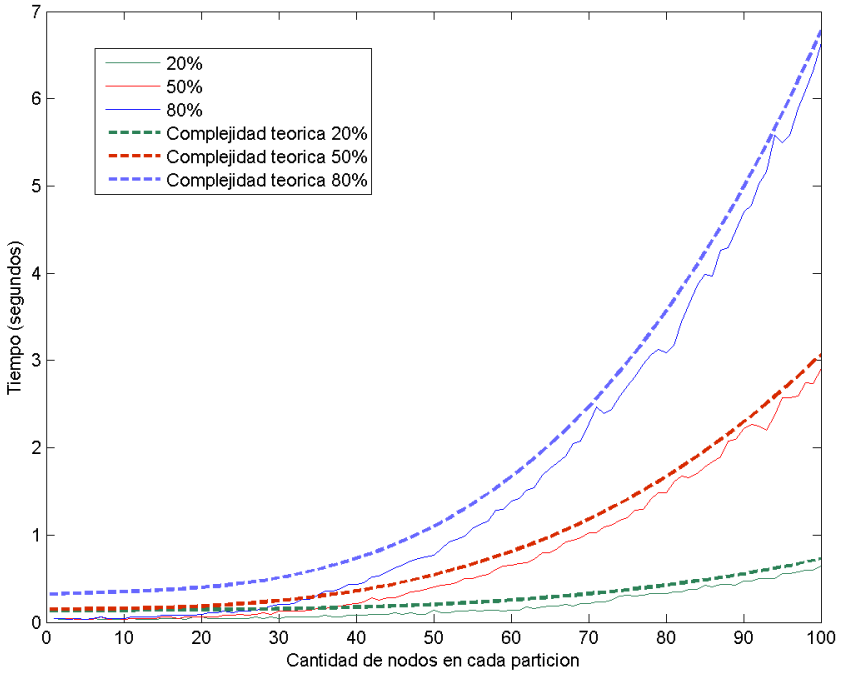


Figura 3.12: Tiempo en función de la cantidad de nodos en cada partición

Los porcentajes indican la proporción de ejes con respecto al grafo bipartito completo correspondiente, y la cantidad de nodos móviles respecto del total de los nodos de cada partición.

Lo primero que observamos es que la complejidad teórica aproxima muy bien los resultados empíricos. Por otro lado, se puede notar también a simple vista que no solo la cantidad de nodos tiene una fuerte influencia en el tiempo de ejecución, sino que también la tiene alguno de los dos parámetros estudiados. Esto es de esperarse como se vio en el análisis de complejidad de la heurística (sección 3.6). Esto tiene sentido, ya que más ejes hacen más costoso el cálculo de cruces, y más nodos móviles aumentan el número de pasos hasta construir una solución. A continuación estudiamos por separado los dos parámetros.

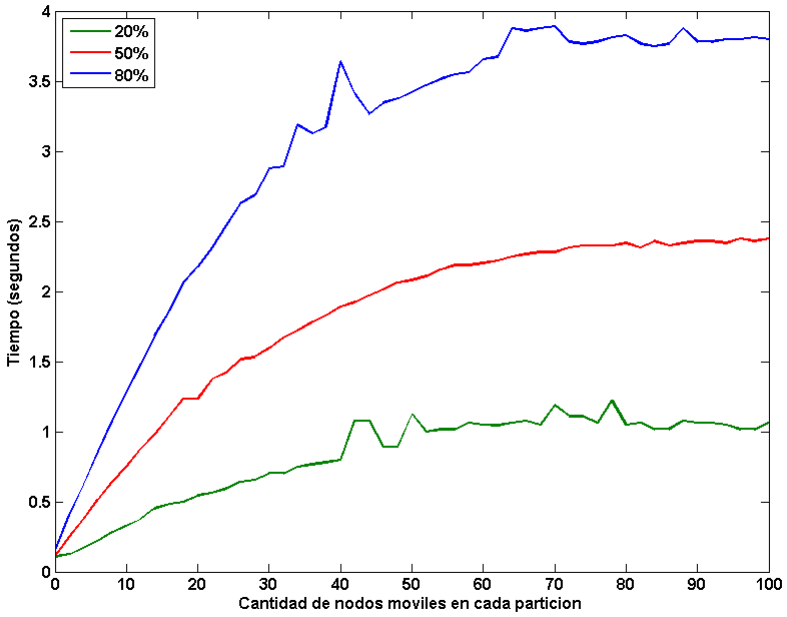


Figura 3.13: Tiempo en función de la cantidad de nodos móviles en cada partición

Los porcentajes indican la proporción de ejes en el grafo con respecto al completo.

Como vemos, hay una influencia muy grande de la cantidad de nodos móviles. A medida que aumenta la cantidad de nodos móviles, aumenta el tiempo, pero observamos que a partir de un 50 % de nodos móviles, el crecimiento parece desacelerarse paulatinamente. Esto concuerda con el hecho de que cada decisión de inserción de un nuevo nodo se basa en una cantidad de chequeos de cruces en tantas ubicaciones como existan en el dibujo. Como una ubicación está determinada por los espacios entre nodos ya fijados, si el dibujo original es más chico, se deberán hacer menos intentos para colocar el primer nodo móvil que si hubiera más nodos ya fijados.

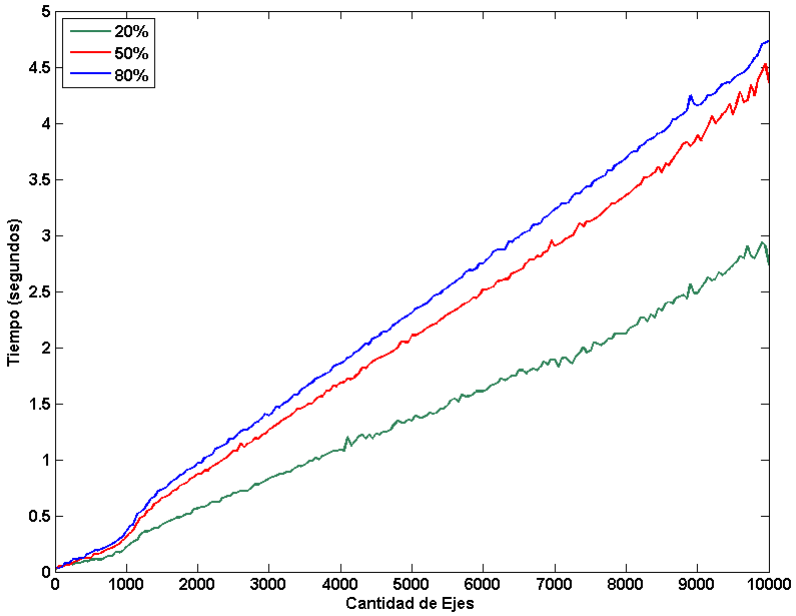


Figura 3.14: Tiempo en función de la cantidad de ejes en cada partición

En este caso, los porcentajes hacen referencia a la cantidad de nodos libres.

Lo que observamos es que una mayor cantidad de ejes conduce a un mayor tiempo de ejecución. El tiempo de ejecución tiene un aspecto lineal, lo cual se condice con el análisis teórico, ya que la complejidad quedaba como una función lineal de m.

A modo de conclusión, podemos decir que los tiempos de ejecución obtenidos empíricamente reflejaron a los obtenidos en el análisis teórico de la heurística.

Parte 4

Búsqueda Local

4.1. Introducción

De manera análoga a lo que hicimos para las heurísticas constructivas, plantearemos a continuación diferentes heurísticas de búsqueda local. A continuación comentaremos como proceden dichas heurísticas, y posteriormente realizaremos diversas experiencias para poder decidir a partir de éstas cual utilizaremos en el algoritmo GRASP. Se implementaron prototipos en Python al igual que con los demás algoritmos.

Los algoritmos de búsqueda local parten de un dibujo original y buscan un óptimo local en la vecindad del dibujo propuesto. El criterio a optimizar, una vez más, es heurístico y varía de un algoritmo a otro. El rasgo común reside en que todos ellos deben ser deterministas - dado un mismo dibujo a mejorar, el algoritmo debe producir siempre el mismo dibujo mejorado.

4.2. Descripción de las heurísticas

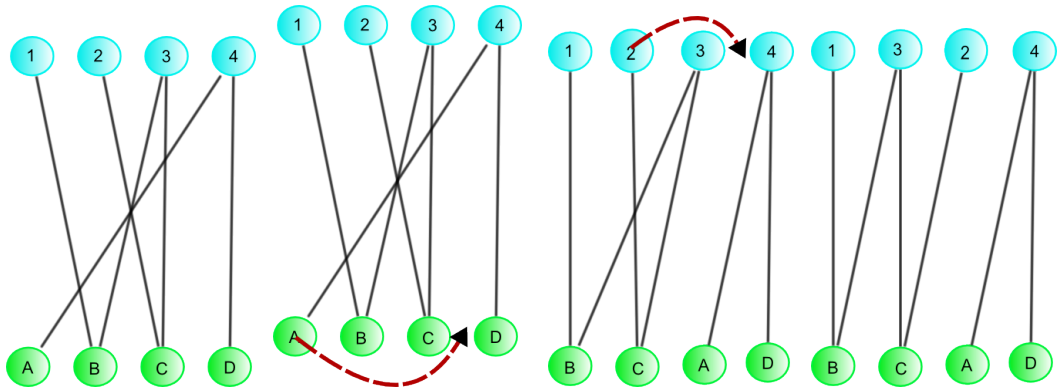
4.2.1. Búsqueda local por reinserción de nodos

Este método de búsqueda local procede tomando cada nodo del dibujo propuesto, sacándolo y reubicándolo de forma análoga a las inserciones de la heurística constructiva por inserción de nodos descripta anteriormente. La idea es tomar un nodo y reinsertarlo en la posición que genera menos cruces. El procedimiento se repite iterativamente una vez para cada nodo del dibujo.

Esa secuencia se repite sucesivamente, reinsertando de a n nodos hasta que no se logre obtener una mejora. Cuando la cantidad de cruces no es mejorada en una iteración, termina la búsqueda y se considera que se alcanzó un mínimo local.

En el caso de los nodos “fijos”, cuyo orden relativo debe ser mantenido, se intenta la reinserción en las posiciones que no violan dicho invariante.

Veamos el siguiente ejemplo de aplicación de la búsqueda local por reinserción (para simplificar no se consideraron nodos fijos):



(a) Dibujo a mejorar (4 cruces)

(b) Buscamos donde reinsertar al nodo A. Delante de D logramos minimizar los cruces

(c) Movemos al nodo A. No podemos mover a nadie más de esta partición de modo de bajar el número de cruces, por lo cual pasamos a la siguiente partición. Moviendo a 4 no logramos nada, por lo que buscamos mover a 3 (1 cruce)

(d) Movimos a 3, y ya no queda ninguna mejora por hacer (0 cruces)

Pseudocódigo

A continuación presentamos el pseudocódigo de una iteración de la búsqueda local. Este algoritmo se aplica iterativamente hasta no lograr obtener mejora alguna.

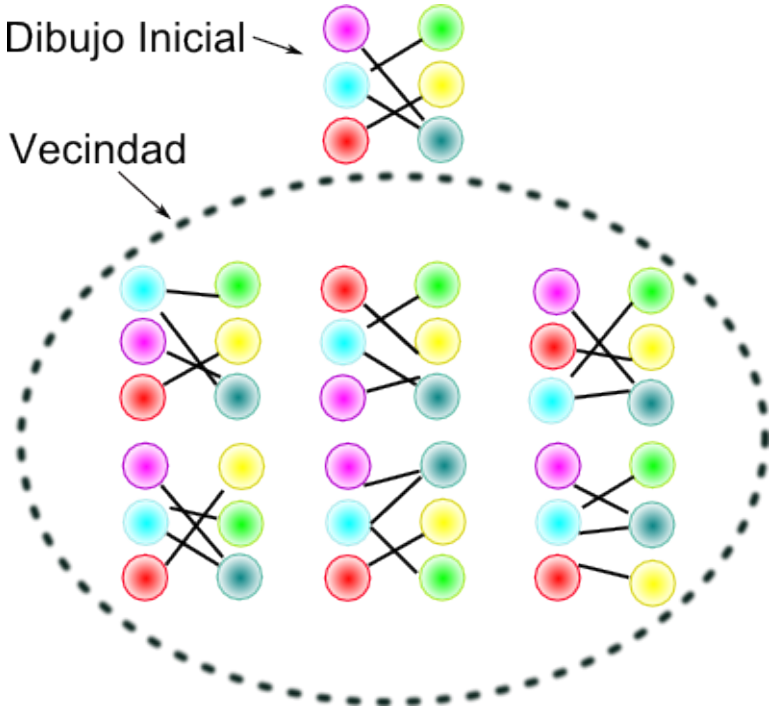
Algoritmo 9 Intenta mejorar un dibujo mediante la reinserción golosa de nodos

Parámetros: lista de nodos del dibujo, índice con la posición de cada nodo en la lista

```
1: Para cada nodo del dibujo hacer
2:    sacar al nodo del mismo
3:    calcular el rango en el cual se puede mover al nodo
4:    insertar al nodo en la primer posición del rango
5:    recalcular los índices
6:    mejoresCruces ← los cruces por ponerlos en esta posición
7:    mejorPosición ← posición actual
8:    crucesAhora ← mejoresCruces
9:    Para cada posición donde se puede poner al nodo hacer
10:      contar los cruces entre el nodo y su vecino siguiente {crucesEntre(nodo,vecino)}
11:      contar los cruces entre el vecino siguiente y el nodo {crucesEntre(vecino,nodo)}
12:      crucesAhora ← crucesAhora - crucesEntre(nodo,vecino) + crucesEntre(vecino,nodo) {cambio de cruces
         por hacer swap}
13:      hacer el swap entre el nodo y su vecino
14:      actualizar los índices
15:      Si crucesAhora < mejoresCruces entonces
16:         mejoresCruces ← crucesAhora
17:         mejorPosición ← posicion actual
18:      Fin si
19:   Fin para
20:   poner al nodo en mejorPosicion
21: Fin para
```

4.2.2. Búsqueda local por intercambio de nodos

Esta heurística contempla como soluciones vecinas de un dibujo a aquellas que se pueden obtener por un intercambio válido entre dos nodos del dibujo.



Primero se considera la vecindad del candidato propuesto, consistente en todo posible intercambio de dos nodos (siempre que dicho intercambio no viole el orden relativo de los nodos originales), y luego prueba cual de todos esos intercambios reporta mayor beneficio en cuanto al número de cruces. Una vez hallado dicho par, nos movemos a la solución vecina realizando el intercambio de dichos nodos. Al hacerlo terminamos un paso de la búsqueda local.

El procedimiento se repite hasta que ningún intercambio genere una reducción en el número de cruces, en cuyo caso decimos que alcanzamos un mínimo local.

Pseudocódigo

Algoritmo 10 Intenta mejorar un dibujo mediante intercambio goloso de nodos

```
1: vecindad = {(x, y) por cada x en alguna partición e y de la misma partición, si es válido intercambiar x por y}
2: mejorIntercambio ← ninguno
3: crucesPorIntercambio ← cantidad de cruces del dibujo
4: Para (x, y) en la vecindad hacer
5:     crucesVecino ← cantidad de cruces al intercambiar x e y
6:     Si crucesVecino < crucesPorIntercambio entonces
7:         mejorIntercambio ← (x, y)
8:         crucesPorIntercambio ← cruces al intercambiar x e y
9:     Fin si
10: Fin para
11: Si mejorIntercambio ≠ ninguno entonces
12:     realizar el intercambio
13: Fin si
```

4.2.3. Búsqueda local con inserción por mediana

Una de las heurísticas constructivas que planteamos es la inserción de nodos por mediana. Esta heurística no funcionó tan bien como esperábamos a partir de la lectura de otras fuentes, ya que si bien era rápida, generaba más cruces que las otras heurísticas golosas. Nuestra idea entonces es aplicar el concepto de la heurística constructiva de la mediana, pero como búsqueda local.

En este contexto, como todos los nodos están ya puestos, cada nodo tiene ahora la información de todos sus adyacentes. Es por esta razón que creemos que podría haber una mejora importante en el rendimiento del algoritmo.

La idea es entonces muy similar a la de inserción por mediana: tomamos cada nodo de una partición y tratamos de moverlo a la posición correspondiente a la mediana de las posiciones de sus nodos adyacentes (o sus vecinos inmediatos). Si con esta reubicación se disminuyen los cruces totales del dibujo, se realiza la modificación, y en caso contrario se pasa al nodo siguiente. Una vez hecho esto para todos los nodos, lo que hacemos es tratar de intercambiar adyacentes, con el objetivo de reducir el número de cruces.

Nuevamente, la búsqueda termina cuando no es posible reducir el número de cruces ya sea por medio de la reubicación de un nodo por mediana o intercambiando pares de nodos.

Pseudocódigo

Algoritmo 11 Intenta mejorar un dibujo con inserción por mediana

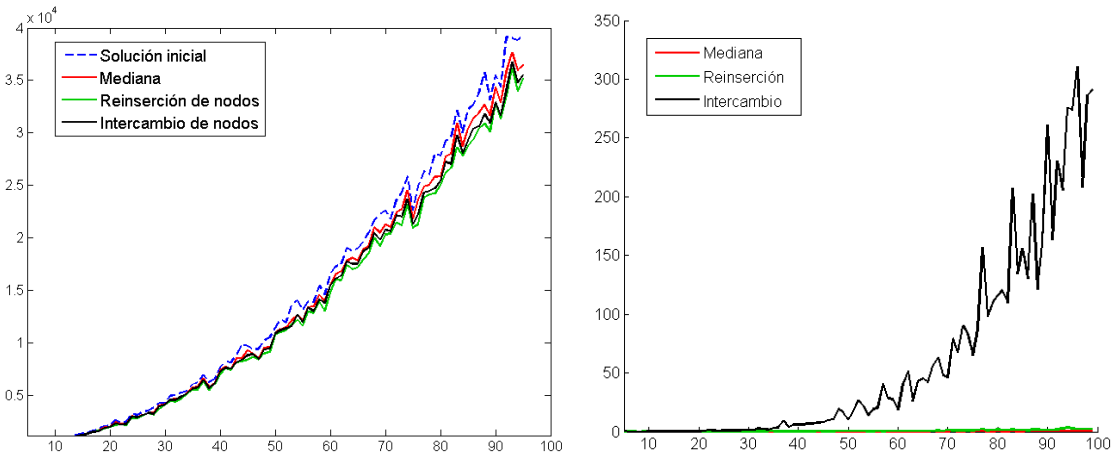
```
1:  Para cada nodo del dibujo hacer
2:    calcular la mediana de las posiciones de los adyacentes al nodo
3:    mejorPos ← posicionActual
4:    mejoreCruces ← cruces en el dibujo
5:    Para  posicion = mediana -1, mediana, mediana + 1 hacer
6:      Si  se puede insertar en esa posición y baja el número de cruces en el dibujo entonces
7:        mejorPos ← posicion
8:        mejoresCruces ← cruces en el dibujo al poner al nodo en posicion
9:      Fin si
10:   Fin para
11:   poner al nodo en mejorPos
12: Fin para
```

4.3. Comparación de las heurísticas de búsqueda local

Con el fin de determinar que heurística implementaríamos en C++ para luego ser usada en el GRASP, decidimos realizar pruebas sobre los prototipos de la misma manera que hicimos con los algoritmos anteriores. Una vez más consideramos la mejora lograda por los algoritmos y el tiempo insumido por su ejecución.

Para hacer las pruebas generamos distintos tipos de grafos, les aplicamos la heurística constructiva y luego aplicamos las distintas heurísticas de búsqueda local.

La primera prueba se realizó en grafos con una cantidad de nodos creciente, un 40 % de nodos fijos y $m = 5 * n$.

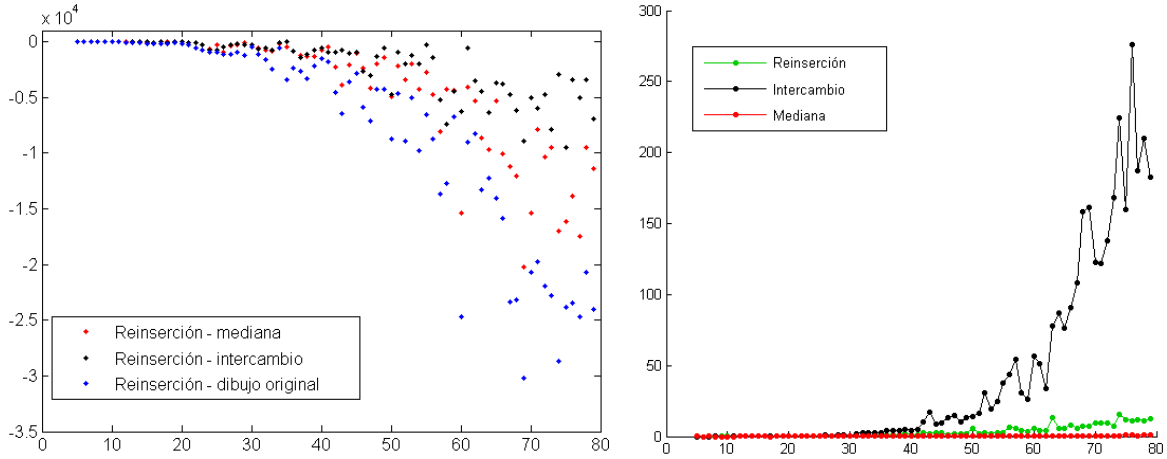


(a) Cantidad de cruces obtenidos en función de n

(b) Tiempo en segundos en función de n

Figura 4.1: n nodos en cada partición, n creciente, $m = 5 * n$, 60 % de nodos nuevos

La siguiente prueba se realizó en grafos más densos con un 40 % de nodos nuevos. Dado que los resultados obtenidos eran relativamente similares, decidimos graficar la diferencia en el número de cruces en el dibujo producido por la reinserción y el producido por las otras heurísticas, a modo de hacer más visibles los resultados.

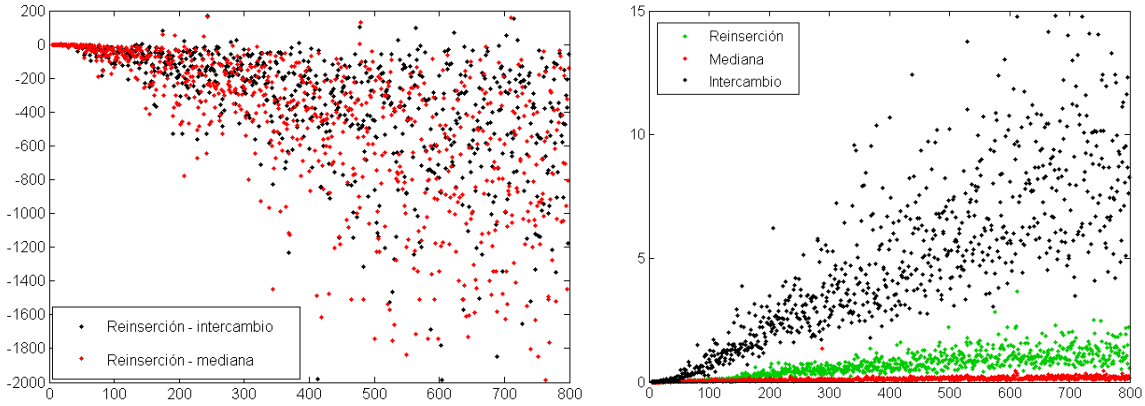


(a) Diferencia en la cantidad de cruces obtenidos en función de n

(b) Tiempo en segundos en función de n

Figura 4.2: n nodos en cada partición, n creciente, $m = n * n/2$, 60 % de nodos nuevos

La tercera y última experiencia se realizó para grafos de 30 nodos, variando la cantidad de ejes. Nuevamente, el método de reinserción fue el que presentó en general un mejor compartamiento, por eso decidimos como en la experiencia anterior graficar la diferencia entre la cantidad de cruces obtenida por cada implementación.



(a) Diferencia en la cantidad de cruces obtenidos en función de m

(b) Tiempo en segundos en función de m

Figura 4.3: n nodos en cada partición, n creciente, $m = n * n/2$, 60 % de nodos nuevos

4.4. Análisis de los resultados

A partir de los gráficos, se puede ver que la heurística de reinserción es la que reduce en mayor medida la cantidad de cruces. Esto se evidencia en las tres experiencias. En la primera, el número de cruces en el dibujo dado por esta heurística está por debajo del número generado por las demás, y en la segunda y tercera al graficar *cruces de la heurística de reinserción - cruces del dibujo de otra heurística* se observa que la mayoría de los puntos se encuentran debajo del 0, lo cual indica que la heurística de reinserción encontró un dibujo con menos cruces. En cuanto al tiempo, si bien fue mayor que el tiempo de la heurística de la mediana, fue considerablemente menor que la de intercambio goloso. Además, consideramos que siempre se mantuvo en rangos razonables, ya que nunca necesitó de un tiempo extremadamente largo para alcanzar un mínimo local.

La heurística de mejor intercambio se ubica en general en segundo lugar en cuanto a resultados; sin embargo, el tiempo que requiere para mejorar un dibujo es demasiado alto. Esto se debe a varias razones: en primer lugar, este método requiere explorar toda la vecindad, que tiene un tamaño $O(n^2)$, y en segundo lugar cada "pasada" solo hace un intercambio (el mejor entre los posibles) por lo que parece razonable pensar que harán falta muchas más iteraciones de la búsqueda para alcanzar un mínimo local.

Finalmente, la heurística de la mediana, al igual que la versión constructiva, fue la mejor en lo que a tiempo se refiere. No obstante, no obtuvo buenos resultados en cuanto a la reducción de cruces.

Por estas razones decidimos descartar a las heurísticas de intercambio de nodos y reinserción por mediana, conservando la reinserción *greedy* de nodos.

4.5. Detalles de implementación

Los objetos de la clase búsqueda local se construyen tomando un dibujo original (fijo) y recibe un dibujo incremental que es el que se debe mejorar. Cuando terminar de procesar, devuelve un nuevo dibujo.

Se utilizaron listas enlazadas para guardar las particiones que se van modificando en cada paso de la búsqueda local. No se utilizaron vectores porque es necesario eliminar elementos de posiciones arbitrarias, lo cual puede hacerse de manera más eficiente sobre esta estructura de datos.

Una vez insertado un nodo, el rango en el cual el mismo puede ubicarse se recorre mediante *swaps*, de modo de poder calcular más eficientemente los cruces por cambiarlo de posición.

Por otro lado, se mantienen durante cada iteración los índices de cada partición en un diccionario sobre vector. Estos índices se pueden actualizar fácilmente cuando se *swapea* un nodo, pero sin embargo el sacar un nodo y reinsertarlo hace

que en algunos casos volver a computar las posiciones pueda ser de orden lineal.

Para poder iterar sobre los elementos de cada partición, llevamos otra copia de la lista que se va a modificar a fin de poder recorrer los elementos en el orden en que vienen dados.

El dibujo que recibe la heurística no posee nodos de grado 0 ya que el grafo es preprocesado al inicio para evitar precisamente ese caso. Además como resultado de este filtrado los nodos que deben guardar una posición relativa dada cumplen que su *id* respeta ese orden (es decir, si el nodo a tiene que estar antes que b entonces $a < b$).

4.6. Cálculo de complejidad

Para realizar el cálculo, definiremos v_i como la cantidad de nodos de la partición i, y m como la cantidad de ejes del dibujo. Dado que el dibujo no posee nodos de grado 0, sabemos que $m > v_i$. Además definimos v_{max} como la cantidad de nodos de la partición mas grande. Utilizaremos el modelo uniforme, ya que consideramos que lo importante es la cantidad de nodos y ejes en el dibujo, más que las operaciones aritmeticas que se realizan (que son tan simples como escasas).

A la complejidad del algoritmo que vamos a describir a continuación, debe sumarse la complejidad de "limpiar" al grafo y de correr la heurística constructiva.

Una vez que hicimos esto, veamos el costo de cada paso.

La heurística de búsqueda local va a iterar para cada nodo del dibujo. Dado un nodo, primero se lo retira de su partición. Esto tiene un costo $O(v_i)$. Una vez que lo retiramos, determinamos el rango donde insertarlo: si no es un nodo de los que deben guardar un orden relativo dado (en adelante nodos fijos), el rango es toda la partición; si es un nodo fijo, el rango es delante del fijo anterior a él (o la primer posición si no existe tal nodo) y detrás del fijo siguiente (o la última posición si no existe tal nodo). Determinar dicho rango es $O(1)$, pues dado un nodo podemos saber fácilmente si es fijo o no por su identificador. Dado un nodo fijo, podemos saber también por su identificador cual es el nodo fijo anterior o siguiente, y al estar los índices actualizados como dijimos anteriormente, conocer la posición de estos nodos también es $O(1)$.

Luego insertamos el nodo en su primer posición válida. Hacerlo es $O(v_i)$ ya que si es un nodo fijo, podríamos tener que insertarlo en posiciones arbitrarias. Este borrado y reinserción del nodo requiere que se actualice el índice de la partición, también en $O(v_{max})$, y por otro lado requiere que se recalculen los cruces. Hacer esto último nos cuesta $O(m * log(v_{min}))$ como vimos en 1.2.

A partir de este momento, el nodo es *swapeado* hacia atrás para recorrer todo su rango. El rango tiene a lo sumo v_{max} posiciones. Para calcular los cruces se calculan entonces los cruces entre el nodo y el nodo inmediatamente siguiente (análogamente a lo realizado con el algoritmo exacto) y se aplica la fórmula dada en 1.2. Calcular los cruces nos cuesta entonces $O(min(max(v_i, m_a, m_b), m_a * m_b))$. Esto sale de que vimos que el orden de calcular cruces entre dos posiciones adyacentes era $O(v_i, m_a, m_b)$, y si teníamos pocos ejes ($m_a * m_b < v_i$) usábamos el algoritmo de conteo de cruces más sencillo cuya complejidad es $O(m_a * m_b)$. Podemos suponer sin equivocarnos que $min(max(v_i, m_a, m_b), m_a * m_b) \subseteq O(v_{max})$.

Al *swapear* nodos se pueden actualizar los índices en $O(1)$ ya que solo cambian dos posiciones. Por otro lado, comparar la cantidad de cruces para decidir si el nodo está en una mejor posición y en caso afirmativo guardar dicha posición también se realiza en $O(1)$.

Una vez que recorrimos todo el rango, sacamos de nuevo al nodo para ponerlo en la mejor posición (y se actualizan los índices), lo cual se hace en $O(v_{max})$.

En conclusión, iteramos para $v_1 + v_2$ nodos, o sea que hay $O(v_{max})$ iteraciones. Cada iteración tiene un costo de $O(v_{max} * (v_{max} + m * log(v_{max}))) \subseteq O(v_{max} * m * log(v_{max}))$. Luego cada paso tiene un costo $O(v_{max}^2 * m * log(v_{max}))$.

Ahora bien, este costo es el de cada paso. El orden de toda la búsqueda local es $O((Pasos) * v_{max}^2 * m * log(v_{max}))$. Necesitaríamos saber cuantos pasos puede realizar la búsqueda. A priori no sabemos cuantos puede realizar, ya que tampoco tenemos una cota que nos diga qué tan lejos puede estar la solución inicial del mínimo local. Por esta razón, lo mejor que podemos hacer es observar que la cantidad de iteraciones no es de orden exponencial.

En el peor de los casos, en la solución inicial cada eje se cruzaba con todos los demás, dando una cantidad de cruces $O(m^2)$, y el mínimo local es 0 (esa es la mayor cantidad de valores distintos de cruces que puede haber). También en peor caso, la mínima disminución por iteración es de 1 cruce (ya que si disminuye menos el algoritmo para). Por lo tanto, la cantidad de iteraciones está acotada por $O(m^2)$. Resulta de esto que la búsqueda local tiene un orden de complejidad

$O(v_{max}^2 * m * log(v_{max}) * m^2)$.

Sin embargo, nos permitimos suponer que en general la cantidad de iteraciones hasta parar será en la práctica mucho menor.

Además de esto hay que tener en cuenta el costo de la limpieza del grafo y el costo de la heurística constructiva. Finalmente, el orden es $O(v_{max}^2 * m * log(v_{max}) * m^2 + Moviles * v_{max}^2 + m * log(fijos_{max}) + fijos_{max} + (V_1 + V_2 + m))$

Con respecto a la complejidad en función del tamaño de la entrada, podemos ver que la entrada es:

$$t = log(P_1) + \sum_{i=1}^{P_1} log((p_1)_i) + log(P_2) + \sum_{i=1}^{P_2} log((p_2)_i) + log(m_p) + \sum_{i=1}^{m_p} log((e_i)_0) + log((e_i)_1)$$

$$+log(IV_1) + \sum_{i=1}^{IV_1} log((iv_1)_i) + log(IV_2) + \sum_{i=1}^{IV_2} log((iv_2)_i) + log(m_{iv}) + \sum_{i=1}^{m_{iv}} log((e'_i)_0) + log((e'_i)_1)$$

donde P_i es la cantidad de nodos originales de la primera partición, m_p es la cantidad de ejes originales, IV_i es la cantidad de nodos que se agregan a la partición i y m_iv es la cantidad de ejes que se agregan al grafo.

Entonces, vale que:

$$t \geq log(P_1) + P_1 + log(P_2) + P_2 + log(m_p) + m_p + log(IV_1) + IV_1 + log(IV_2) + IV_2 + log(m_{iv}) + m_{iv}$$

Pero $P_i + IV_i = V_i$ y $m_p + m_iv = m$, luego:

$$t \geq log(P_1) + v_1 + log(P_2) + V_2 + log(m_p) + m + log(IV_1) + log(IV_2) + log(m_{iv})$$

A partir de esto, vemos que:

$$t \geq V_i \geq v_i$$
$$t \geq V_{max} \geq v_{max}$$
$$t \geq m$$

Luego obtenemos $O(v_{max}^2 * m * log(v_{max}) * m^2 + Moviles * v_{max}^2 + m * log(fijos_{max}) + fijos_{max} + (V_1 + V_2 + m)) \subseteq O(t^5 * log(t))$

4.7. Análisis de la heurística

4.7.1. Casos patológicos

Para observar que tan mala podría ser nuestra heurística de búsqueda local, buscamos alguna familia de instancias del problema donde el algoritmo encuentre una solución arbitrariamente mala.

Consideramos entonces el siguiente grafo:

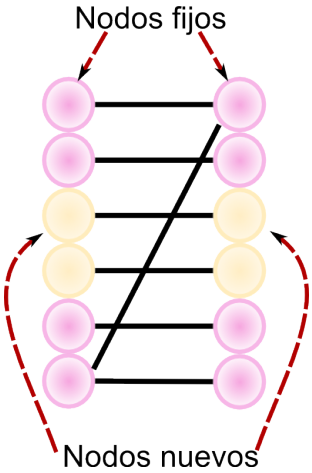


Figura 4.4: Caso patológico para la búsqueda local

Frente a este grafo, la heurística de búsqueda local no podrá hacer ninguna mejora, ya que para cualquier nodo que mueva, siempre obtiene una cantidad de cruces mayor o igual a la que ya tenía. Por ende en este caso el dibujo que provee la heurística frente a esta entrada tiene 4 cruces, 2 mas que la solución óptima:

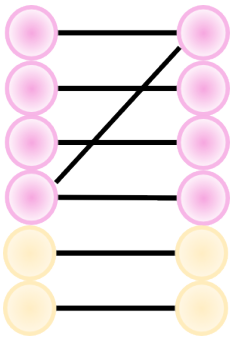


Figura 4.5: Solución óptima para el grafo de ejemplo

En general, podemos considerar como familia de casos problemáticos para la heurística a todos los grafos que tengan la forma anterior, pero que en vez de dos nodos nuevos, incorporen k nodos nuevos:

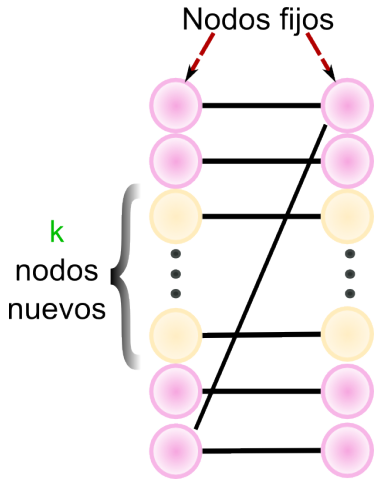


Figura 4.6: Familia de casos patológicos para la búsqueda local

Frente a toda esta familia de grafos, nuestra búsqueda local no puede hacer ninguna mejora. Por esta razón, frente a un caso con una solución óptima de 2 cruces (que como en el ejemplo, se obtiene bajando a los nodos nuevos a la parte inferior del dibujo), la heurística de búsqueda no es capaz de producir ninguna mejora. Así, tomando k apropiados, se pueden producir casos que hacen la heurística de búsqueda cometa un error arbitrariamente grande.

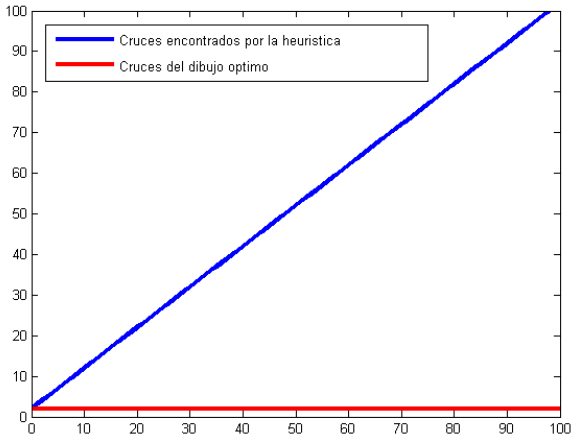
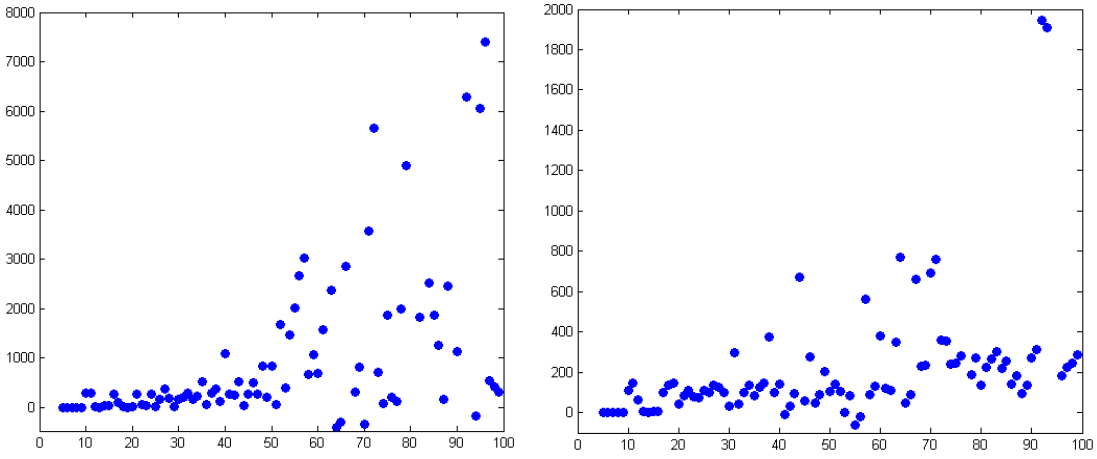


Figura 4.7: Cantidad de cruces en función de la cantidad de nodos nuevos agregados para grafos de la familia 4.6

Este ejemplo, aunque sencillo, nos alcanza para ver que nuestra búsqueda local no es un algoritmo aproximado, ya existen instancias donde la solución que propone el algoritmo es arbitrariamente mala.

4.7.2. Relación con la heurística constructiva

Si bien vimos que de todas las variantes propuestas la que obtuvo un mejor desempeño fue la heurística de búsqueda local por reinserción de nodos, como la finalidad detrás de la heurística es ser utilizada como mejora en un algoritmo GRASP, consideramos que debíamos observar si esta búsqueda local se beneficiaba de partir de un dibujo construído mediante la heurística constructiva. Para observar esto, ejecutamos la heurística de búsqueda local para una misma instancia del problema, a partir de un dibujo generado al azar y de un dibujo generado por la heurística constructiva. Luego medimos la diferencia en el número de cruces obtenido en cada caso. Los resultados fueron los siguientes:



(a) 60 % de nodos nuevos, $m = n * n/5$ (b) 40 % de nodos nuevos, $m = 5 * n$

Figura 4.8: Diferencia en la cantidad de cruces obtenidos en función de n

En los gráficos, lo que observamos es que la mayor parte de los puntos son mayores que cero. Dado que lo que graficamos fue la diferencia entre los cruces obtenidos al comenzar por un dibujo al azar y por el dibujo obtenido por la heurística

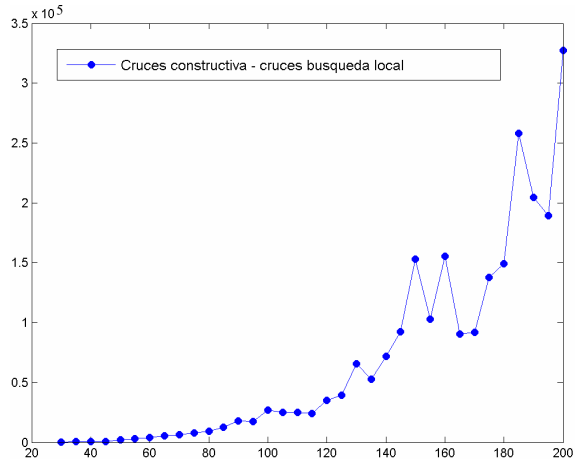
constructiva, resulta que un punto positivo nos dice que $Cruces_{Azar} - Cruces_{Constructiva} > 0$, o sea que $Cruces_{Azar} > Cruces_{Constructiva}$.

Luego, lo que observamos es que en general la construcción de un dibujo con la heurística de inserción greedy de nodos permite que obtengamos, luego de la búsqueda local, una mejor solución que usando la búsqueda local por sí sola, justificando así el uso de la heurística constructiva.

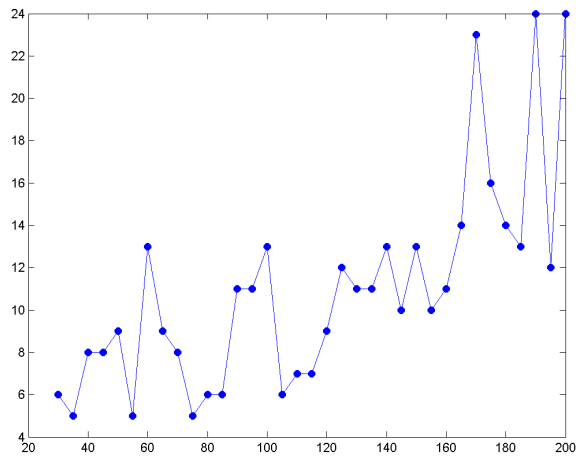
4.7.3. Aplicación a distintas instancias

Para probar el desempeño de nuestra implementación, realizamos varias pruebas sobre diferentes tipos de instancias.

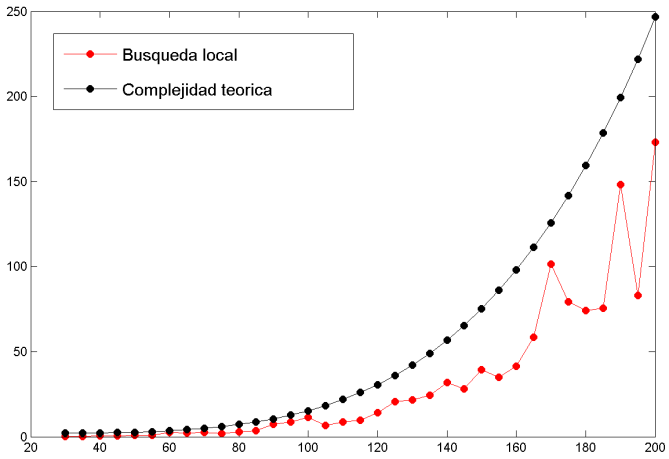
Como primera experiencia, aplicamos la heurística constructiva a grafos con n nodos de cada lado, n creciente, y una cantidad de ejes igual a $n^2/5$. La cantidad de nodos móviles fue del 40 %.



(a) Diferencia en la cantidad de cruces con respecto a la heurística constructiva



(b) Cantidad de iteraciones hasta encontrar un mínimo local



(c) Tiempo (en segundos) y complejidad teórica

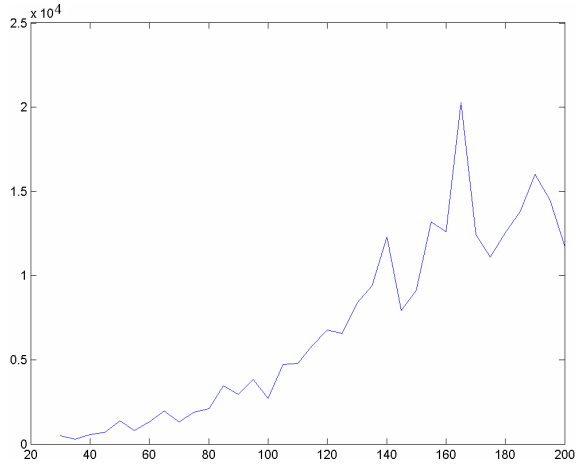
Figura 4.9: Aplicación a grafos con n nodos en cada partición, n creciente, $m = n^2/5$, 40 % de nodos nuevos

En el primer gráfico podemos observar como la búsqueda local logra una mejora sustancial en la cantidad de cruces. Por ejemplo, cuando tenemos 200 nodos de cada lado, la diferencia entre la cantidad de cruces de la heurística constructiva y la de búsqueda local es mayor que $3 * 10^6$. Por otro lado si miramos la cantidad de iteraciones que necesita la búsqueda local hasta llegar a un mínimo, vemos que es en general bastante pequeña, y así para un grafo con 200 nodos, no necesita más de 24 pasos hasta lograr el mínimo local.

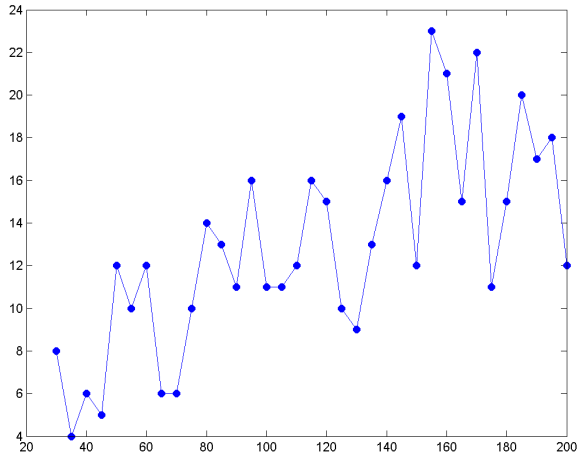
En 4.9(c) vemos el tiempo de ejecución, y como la complejidad teórica acota de forma adecuada los valores obtenidos.

Esta complejidad teórica se calculó utilizando el número real de iteraciones, el cual -recordemos- en el cálculo de complejidad fue acotado por m^2. Sin embargo, como vemos en esta experiencia y en las siguientas, dicha cota es considerablemente gruesa.

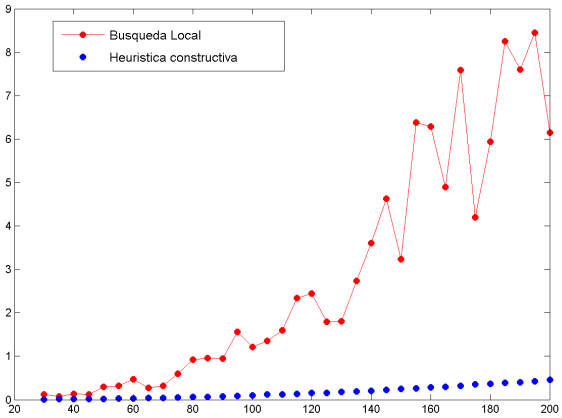
La siguiente experiencia fue similar a la anterior, con la diferencia de que se modificó la cantidad de ejes usando esta vez $m = 5 * n$, así como la cantidad de nodos libres.



(a) Diferencia en la cantidad de cruces respecto de la heurística constructiva



(b) Cantidad de iteraciones hasta encontrar un mínimo local



(c) Tiempo (en segundos) de la búsqueda local y de la heurística constructiva

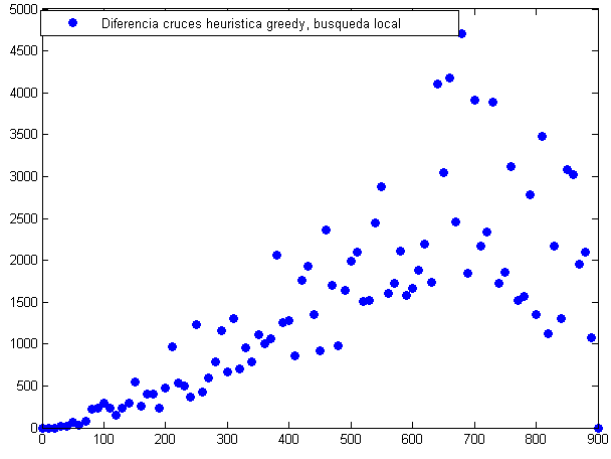
Figura 4.10: Aplicación a grafos con n nodos en cada partición, n creciente, $m = 5 * n$, 60 % de nodos nuevos

Nuevamente notamos como la búsqueda local hace una mejora significativa sobre la solución propuesta por la constructiva, ya que como vemos en 4.10(a) la diferencia en el número de cruces llega al orden de $2 * 10^4$, diferencia que es menor que en la experiencia anterior (esto es esperable ya que al haber menos ejes, es probable que la cantidad de cruces baje, y que por lo tanto la diferencia que se pueda lograr sea menor). Como vemos en 4.10(b), nuevamente la cantidad de iteraciones es considerablemente menor que la cota que propusimos en el análisis teórico. También se observa que la cantidad de iteraciones tiende a aumentar a medida que aumenta la cantidad de nodos del grafo.

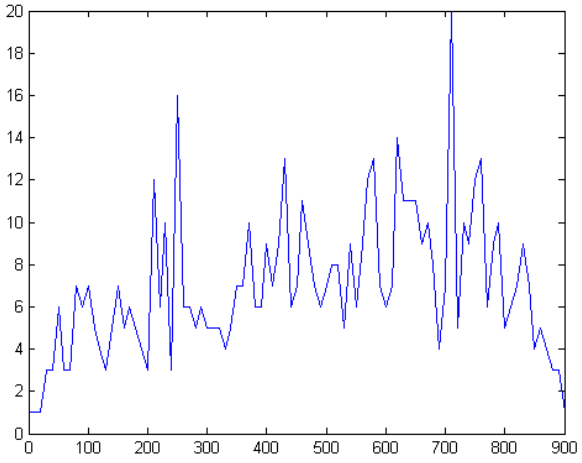
En 4.10(c) vemos la diferencia entre el tiempo de la búsqueda y el de la construcción de una solución mediante la heurística constructiva. Como era de esperarse, la búsqueda local tiene un tiempo mucho mayor esta última (tiene que ser mayor inevitablemente, ya que se parte de una solución que se construye con el método constructivo). Sin embargo, hay un detalle a notar en este gráfico: al compararlo con 4.9(c), notamos que el tiempo es mucho mayor, lo cual refleja el hecho de que el conteo de cruces se hace más caro, en particular en el primer caso: el factor m se convierte en v_i^2, por lo cual es de esperar esta gran diferencia entre los tiempos. También podemos extraer de este gráfico que la construcción de la solución base tiene un costo muy bajo comparado con el costo de mejorarla. Esto refuerza la idea de que es conveniente comenzar con una solución generada por la heurística constructiva, ya que antes vimos que en general el algoritmo completo obtiene menos cruces que comenzando por un ordenamiento al azar, y como vemos en 4.10(c) el tiempo de construcción del

candidato inicial es prácticamente despreciable con respecto al tiempo total de la búsqueda.

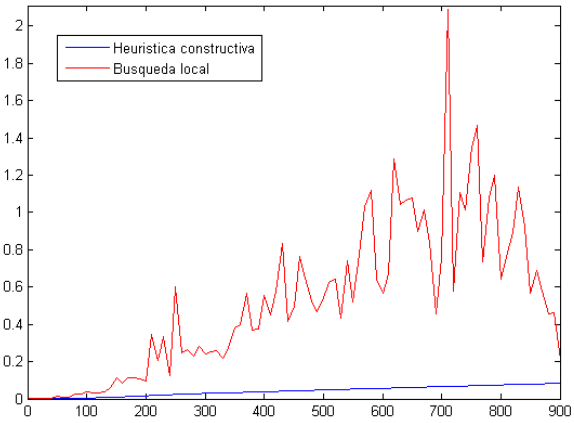
Para ver como influía la cantidad de ejes en el desempeño de la heurística, decidimos hacer una prueba dejando la cantidad de nodos fija (en 30) y variar la cantidad de ejes del grafo. Los resultados son los siguientes:



(a) Diferencia en la cantidad de cruces con respecto a la heurística constructiva



(b) Cantidad de iteraciones hasta encontrar un mínimo local



(c) Tiempo (en segundos) de la búsqueda local y de la heurística constructiva

Figura 4.11: Diferencia entre cruces, cantidad de iteraciones y tiempo de ejecución en función de m

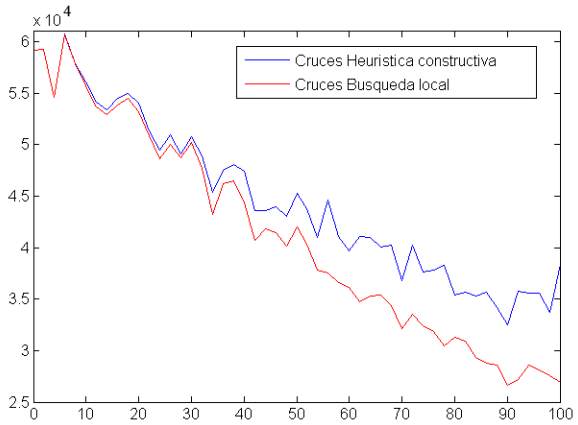
Con respecto a 4.11(a), vemos que la diferencia aumenta y luego parecería comenzar a disminuir. Creemos que esto se debe a que cuando hay pocos ejes es más fácil lograr pocos cruces. A medida que aumenta m es más posible mejorar la solución inicial, pero cuando se va saturando el grafo, es menos lo que se puede hacer para reducir el número de cruces (en el extremo, dado un grafo completo cualquier ordenamiento logra la misma cantidad de cruces).

En 4.11(b) podemos observar como al igual que en las experiencias anteriores, la cantidad de iteraciones es mucho menor que la cota dada en el análisis teorico. Además, en este caso se observa que la cantidad de iteraciones no fue considerablemente más chica que la cantidad de nodos que había en el grafo.

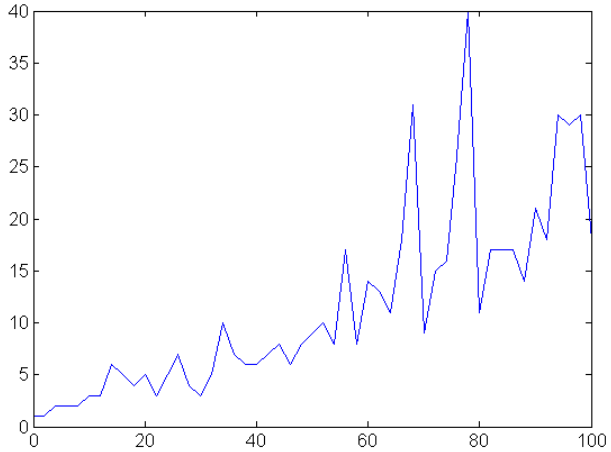
Al mirar 4.11(c), se nota claramente la relación entre el tiempo de ejecución y la cantidad de iteraciones, ya que al observar la gráfica de tiempo así como también la de iteraciones, vemos un patrón muy similar en ambas. A grandes rasgos incluso parecería verificarse que el tiempo de ejecución es de cantidad de iteraciones sobre 10. Es de notar también que cuando hay pocos ejes, el mismo número de iteraciones requiere un tiempo menor, lo cual es muy razonable ya que contar cruces se hace más caro, y esta operación es básicamente el núcleo de cada paso de la búsqueda local en lo que a costo se refiere.

Finalmente, buscamos observar que influencia tiene la cantidad de nodos nuevos en el grafo. Por esta razón creamos

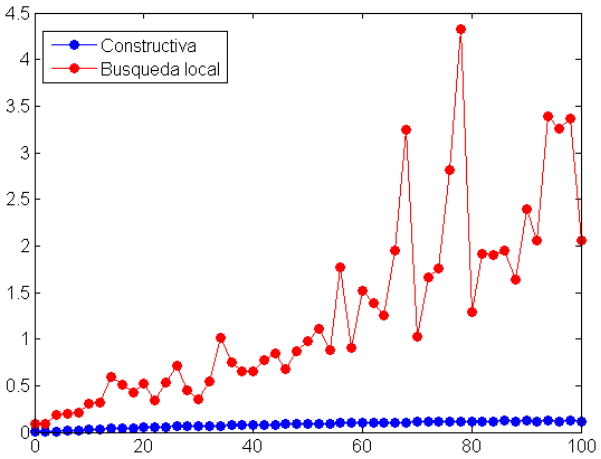
distintos grafos aleatorios, variando esa característica. Los resultados son los que se presentan a continuación:



(a) Cantidad de cruces con respecto a la heurística constructiva



(b) Cantidad de iteraciones hasta encontrar un mínimo local



(c) Tiempo (en segundos) de la búsqueda local y de la heurística constructiva

Figura 4.12: Diferencia entre cruces, cantidad de iteraciones y tiempo de ejecución en función de la cantidad de nodos nuevos

Lo que notamos es que a mayor cantidad de nodos móviles, mayor es la mejora lograda por la heurística al resutado obtenido previamente por la heurística constructiva. También observamos que esta mejora mayor necesita de más iteraciones. La relación que se observa entre el aumento de la cantidad de iteraciones y el del tiempo, consideramos que pone de manifiesto como nuestra heurística de búsqueda local no es tan sensible al aumento de la cantidad de nodos libres, es decir, creemos que demora más porque necesita más iteraciones, pero el costo de cada iteración es similar. Esto se debe a que a los nodos fijos también se los intenta mover, y si bien estos nodos tienen un rango de desplazamiento menor, sacarlos del dibujo y volverlos a insertar ya de por si trae un costo lineal aparejado.

4.7.4. Conclusiones empíricas

A partir de las experiencias realizadas podemos concluir lo siguiente:

- Existen casos donde la solución propuesta por la búsqueda local dista tanto como se desee de la solución óptima.
- Es conveniente comenzar de una solución construida por la heurística constructiva, ya que en general se obtienen mejores resultados y el costo de la construcción suele ser despreciable con respecto al costo de la búsqueda local.
- La búsqueda local es bastante sensible a la cantidad de ejes que posee el grafo, no solo por cuanto puede mejorar sino por lo que aumenta el costo de cada paso.

- No obstante lo anterior, la cota para la cantidad de iteraciones ($O(m^2)$) fue observada en la practica, aunque exagerada. Esto se condice con lo que esperábamos, pero sin embargo no pudimos construir un acotamiento mejor.
- Las mejoras obtenidas con la búsqueda local son sensiblemente mayores cuando la cantidad de nodos a agregar es mayor. Esto se logra a costa de una mayor cantidad de iteraciones del algoritmo de búsqueda.

Parte 5

GRASP

5.1. Introducción

La idea de la metaheurística GRASP es construir una solución mediante una heurística constructiva (en la que interviene algún factor de aleatoriedad) y luego refinarla mediante una segunda heurística de búsqueda local, que halla el óptimo en la vecindad del caso construido inicialmente.

Este procedimiento se repite una cierta cantidad de veces. La idea de fondo es que la heurística constructiva construye candidatos que son apropiados para empezar, y el factor aleatorio introduce variantes sobre estos candidatos para explorar al menos varios óptimos locales distintos. Si no existiera este factor, como la búsqueda local es determinista, se encontraría una única solución óptima localmente (la que está en la vecindad del único candidato construíble).

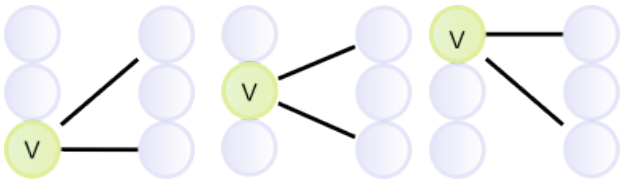
Para construir el algoritmo GRASP es necesario por lo tanto introducir un elemento aleatorio a la heurística constructiva y determinar un criterio de parada apropiado para terminar la ejecución del proceso "construcción - refinamiento" que se lleva a cabo en cada iteración.

5.2. Modificaciones a la heuristica constructiva

Para poder aplicar nuestra heurística constructiva a un procedimiento GRASP, fue necesario introducir algún factor de aleatoriedad a la misma.

Consideramos dos formas de hacerlo:

1. Modificar el criterio de elección del nodo candidato en la inserción: Se considera un valor $\alpha \in [0, 1]$, de modo que en cada paso no se selecciona el de grado máximo, sino que se selecciona un v tal que $d(v) \geq \alpha * d_{max}$. Si $\alpha = 1$, la elección no es aleatoria, en cambio si $\alpha = 0$, se escoge un candidato totalmente al azar. En general, en (0,1), un α más grande implica una lista restringida de candidatos más pequeña.
2. Modificar el criterio de elección de la posición: Frente a un "empate" de posiciones (cuando para un nodo dado a insertar hay dos o más posiciones que generan la misma cantidad de cruces), el algoritmo original se queda con la primera visitada.



(a) (b) (c)

Figura 5.1: Cualquiera de las 3 posiciones para v es a priori tan buena como las otras

Podemos entonces modificar este factor para que de haber "empate" se elija al azar alguna de las posiciones posibles.

Posteriormente, realizaremos experiencias con el fin de determinar si estas modificaciones son útiles, y encontrar qué valor de α es conveniente utilizar.

5.3. Determinación de los parametros

Para nuestro algoritmo basado en GRASP debemos fijar tres parámetros:

1. criterio de parada
2. α (que determina el tamaño de la lista restringida de candidatos)
3. posición aleatoria (que determina si frente a un empate de posiciones nos quedamos con la primera encontrada o con alguna al azar)

El problema de la parametrización de algoritmos basados en metaheurísticas con factores aleatorios es muy delicado. No disponemos de las herramientas como para establecer un criterio de parada óptimo, puesto que la definición de dicho óptimo depende del contexto de uso del algoritmo. Además, el problema tiene parámetros infinitos: cualquier función propuesta a partir de los resultados de las iteraciones de las heurísticas constructivas y de búsqueda puede ser un parámetro válido.

Esta dificultad es tal que da lugar a situaciones curiosas en el desarrollo de algoritmos basados en metaheurísticas. En el caso de los algoritmos genéticos, donde los parámetros suelen ser pocos, no es inusual recurrir a un segundo algoritmo genético para optimizar los parámetros del primero.

Dada esta situación no podemos más que razonar de forma heurística para asignar valores a los parámetros. Al menos será necesario hallar un valor apropiado para el parámetro α de la heurística constructiva, y determinar un criterio de parada cuyo desempeño sea al menos razonable. A partir de ideas heurísticas, intentaremos validar algunas de nuestras predicciones mediante experimentación.

5.3.1. Criterio de parada

Como dijimos anteriormente, el criterio de parada está muchas veces determinado por el contexto de uso del algoritmo. En casos de presentación de datos para su observación por personas, puede ser necesario que dicha presentación tenga un tiempo de respuesta rápido para evitar la percepción de lentitud que puede tener el usuario si el algoritmo es lento. En aplicaciones para integración de componentes electrónicos que se producirán en masa, puede ser aceptable esperar un tiempo sustancial puesto que ese tiempo resulta pequeño comparado con los tiempos y valores involucrados en la producción posterior. En el primer caso se utilizaría un criterio de parada por cota temporal constante (¿ya pasó el tiempo suficiente para impacientar al usuario?), mientras que en el segundo es probable que el algoritmo corra por tanto tiempo como lo permita el plan del proyecto de producción (lo cual podría ser, potencialmente, hasta que la persona responsable le indique al programa que terminó su tiempo de cálculo).

En otros casos se busca simplemente una solución de compromiso - el algoritmo debería tomar un tiempo relativamente corto y obtener una solución en lo posible óptima, o en su defecto bastante buena.

Razonando heurísticamente, consideramos que el criterio de parada debe tener en cuenta la cantidad de nodos que posee el grafo, ya que esta cantidad influye en la cantidad posible de configuraciones y por ende en la cantidad de mejoras que se pueden hacer. Como vimos por ejemplo en la búsqueda local, en general se necesitan más iteraciones para mejorar una

solución de tamaño mayor. Por esta razón el primer criterio que proponemos es el de hacer tantas iteraciones como nodos haya en la partición más grande del grafo.

Un segundo criterio que planteamos varía su cantidad de iteraciones de manera adaptativa. Se toma como valor máximo en el número de iteraciones la cantidad de nodos del grafo. Si en un iteración no se produce una mejora, se disminuye en 1 la cantidad de iteraciones restantes. Si en cambio se produce una mejora, la cantidad máxima de iteraciones se divide por 2. Este criterio utiliza, como el anterior, la idea de que más nodos implica mas trabajo para mejorar, pero por otro lado agrega la idea de que no es posible mejorar indefinidamete y por tanto la ocurrencia de mejoras disminuye la probabilidad de hallar más en el futuro.

5.3.2. Tamaño de la lista de candidatos

Para determinar el tamaño de la lista de candidatos de la heurística constructiva proponemos también dos opciones:

- Tomar un α fijo = 0.75: La idea es que un valor demasiado bajo de α equivale a elegir un nodo cualquiera en lugar de uno de grado máximo. Como observamos empíricamente que elegir uno de grado máximo es apropiado, no es bueno alejarse demasiado de este criterio. Por otro lado, un α demasiado grande hace que la heurística sea esencialmente determinista, y por lo tanto elimina las ventajas de la aleatoriedad en su implementación. Proponemos $\alpha = 0.75$ como un valor intermedio razonable.
- Tomar un α adaptativo: En este caso, se parte de un α alto, 0.95, y en cada iteración, si no se produce mejora, se disminuye su valor. De esta manera, la lista de candidatos comienza siendo pequeña, con la esperanza de lograr buenos resultados rápidamente, y en la medida en que no se observarn mejoras, se da lugar a soluciones más variadas.

5.3.3. Posición aleatoria

En este caso, se consideraron las dos alternativas: tomar una posición aleatoria entre las mejores, o tomar siempre la primer posición.

5.3.4. Experimentos

Con el fin de observar si alguna configuración de los parametros se comportaba mejor que las demás, decidimos aplicar cada posible configuración a distintas instancias del problema. Decidimos identificar a cada combinación mediante un número, lo cual hicimos de la siguiente manera:

1. α 0.75, primera posición, parada por maximo de partición
2. α 0.75, posición aleatoria, parada adaptativa
3. α 0.75, posición aleatoria, parada por maximo de partición
4. α 0.75, primera posición, parada adaptativa
5. α adaptativa, primera posición, parada por maximo de partición
6. α adaptativa, posición aleatoria, parada adaptativa
7. α adaptativa, posición aleatoria, parada por maximo de partición
8. α adaptativa, primera posición, parada adaptativa

Esto evita la consideración heurística de que los parámetros son independientes. Asumir esto introduce un nuevo factor desconocido a la elección de los parámetros, aunque también disminuye la cantidad de combinaciones a examinar. Tomamos la decisión de limitar los parámetros pero sí evaluar todas sus combinaciones.

Para efectuar las comparaciones decidimos medir el tiempo que requiere cada heurística y además considerar la mejora que se logra a partir de la primera iteración del algoritmos, es decir, dada la solución inicial con la que comienza el GRASP, cual es la mejora que se obtiene mediante las iteraciones subsiguientes.

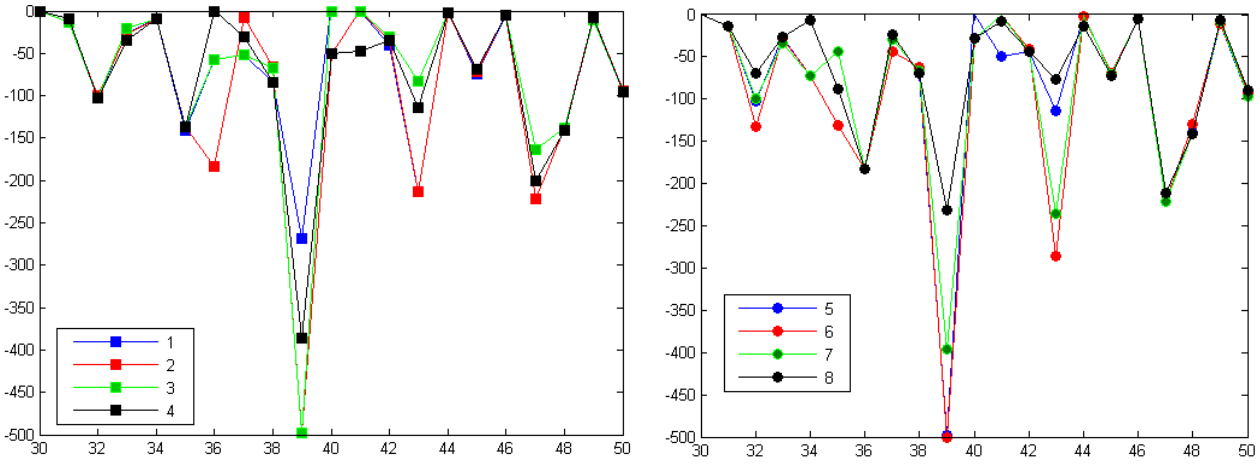
Realizamos las siguientes experiencias:

- Aplicar la heuristica a grafos densos con entre 30 y 50 nodos en cada partición

- Aplicar la heuristica a grafos ralos con entre 50 y 70 nodos en cada partición

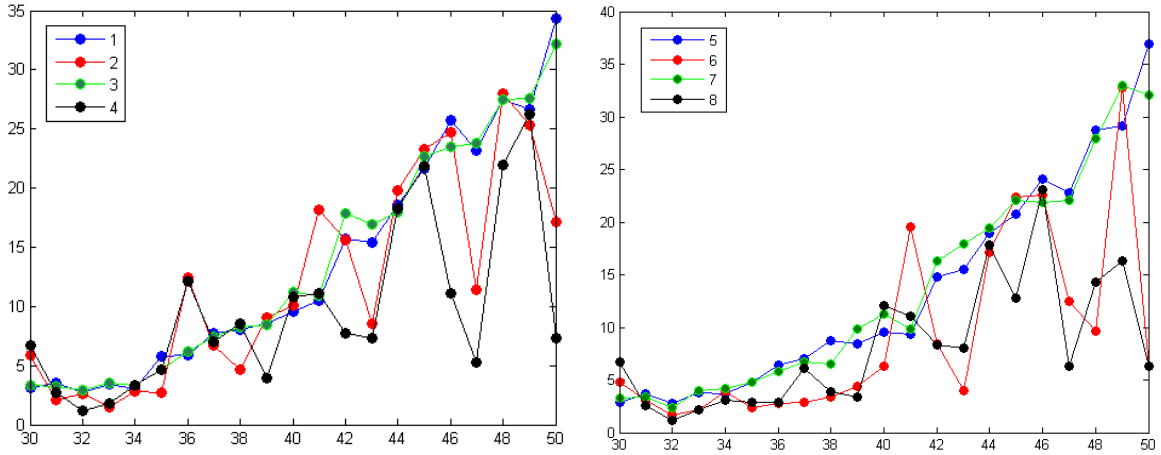
Como en cada experiencia aplicamos las 8 combinaciones, decidimos dividir los gráficos, dejando en uno a los que tienen α fijo (combinaciones 1,2,3,4) y por otro a los que usan un α adaptativo (5,6,7,8) ya que de no hacer esto se hace muy difícil visualizar los resultados.

Los resultados de la primer experiencia son los siguientes:



(a) (b)

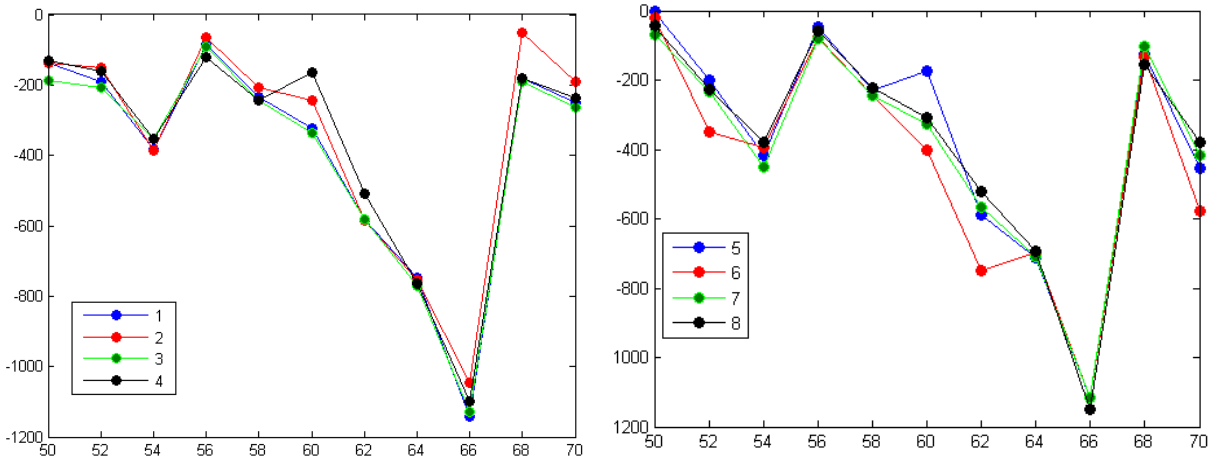
Figura 5.2: Mejora con respecto a la solución propuesta por la búsqueda local



(a) (b)

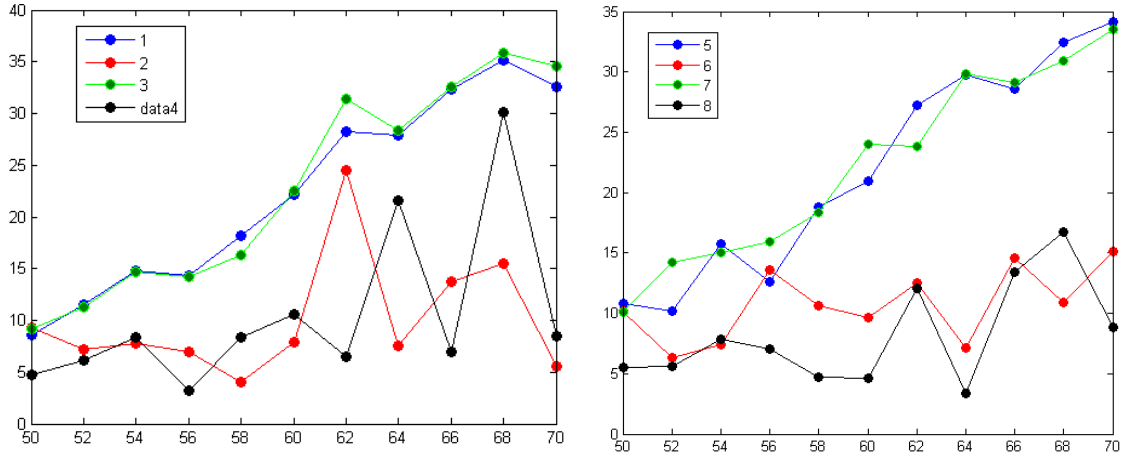
Figura 5.3: Tiempo de ejecución (en segundos)

Lo que podemos observar es que si bien no existe uno que se destaque por sobre el resto, en general 2 y 6 obtienen buenos resultados. Esto es interesante si tenemos en cuenta que son métodos que utilizan el criterio de parada adapatativo. Con respecto a los tiempos de ejecución, el criterio adaptativo suele tener tiempos más bajos. Sin embargo, en los casos donde mejora poco (por ejemplo, para n=41) el método 4 casi no logró mejoras y como podemos observar su tiempo fue más alto en ese caso que el de los métodos no adaptativos.



(a) (b)

Figura 5.4: Mejora con respecto a la solución propuesta por la busqueda local



(a) (b)

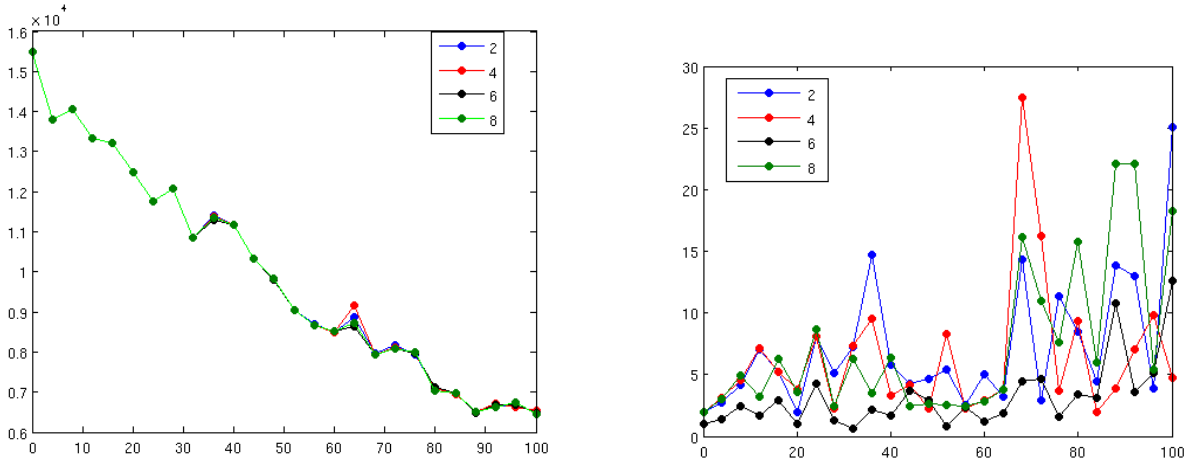
Figura 5.5: Tiempo de ejecución (en segundos)

En esta experiencia se nota claramente la diferencia de tiempo entre los métodos adaptativos y el resto. Con respecto a la mejora en la cantidad de cruces, en este experimento sí se observa un método que se desempeñó mejor: el método número 6. Por otro lado, el método 2, que en la experiencia anterior se había comportado bastante bien, no logró destacarse.

Dado que la cantidad de cruces que encuentra cada versión del GRASP se mantuvo muy similar, decidimos descartar a las heurísticas que no utilizaban un criterio de parada adaptativo, ya que tardaban un tiempo superior sin lograr mejoras importantes.

Decidimos también continuar con la experimentación con las versiones que usan un criterio de parada adaptativo. De esta manera tratamos de observar como se comportaban en función de la densidad del grafo y de la cantidad de nodos móviles.

Los resultados que obtuvimos son los siguientes:



(a) Cantidad de cruces

(b) Tiempo (en segundos)

Figura 5.6: Cruces y tiempo en función de la cantidad de nodos móviles

Nuevamente se mantiene la tendencia a disminuir la cantidad de cruces que encuentra la heurística cuando se aumenta el número de nodos móviles. La misma tendencia se observaba ya en la búsqueda local así como también en la constructiva, haciendo muy probable su aparición tras la combinación de ambas. Observamos también que los tiempos son similares, con la excepción de la versión 6, que necesitó de tiempos mucho menores para lograr una cantidad de cruces menor o igual que los demás.

Nuestra última experiencia en este apartado consistió en dejar fija la cantidad de nodos e ir aumentando la densidad. Dado que nuevamente la cantidad de cruces encontrada fue muy similar, y teniendo en cuenta la tendencia que vemos en las experiencias anteriores en las cuales la versión 6 parecía ser mejor que las demás, decidimos graficar esta vez la diferencia entre la cantidad de cruces encontrada por la versión 6 y el resto de las versiones.

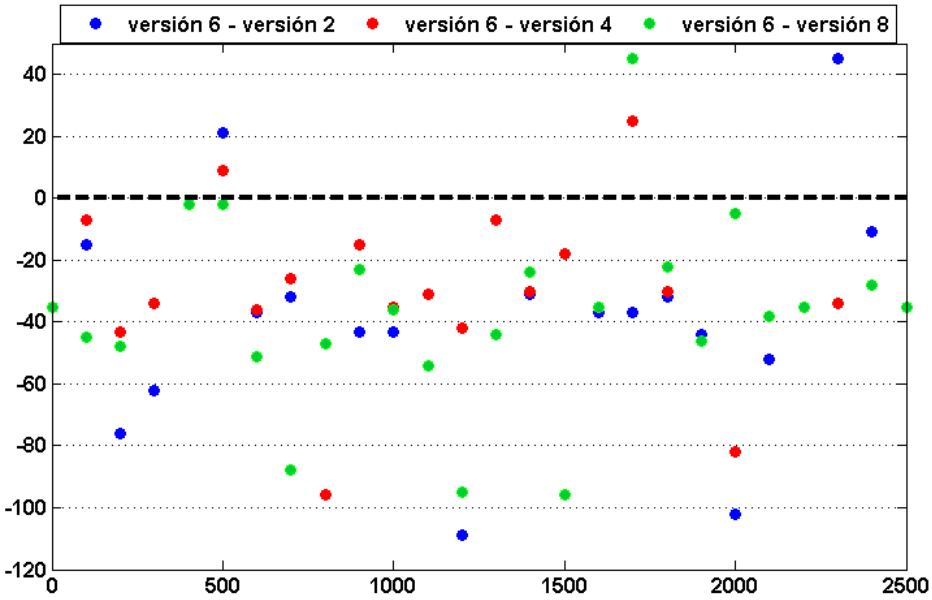


Figura 5.7: Diferencia en la cantidad de cruces encontrada por la versión 6 y el resto (m creciente, 50 nodos en cada partición)

Este gráfico muestra de manera clara, en primer lugar, que la cantidad de cruces es muy similar entre todas las versiones. En segundo lugar vemos como la versión 6 efectivamente tiende a comportarse mejor. Esto se refleja en que al graficar la diferencia entre la cantidad de cruces encontradas por esta versión y la cantidad encontrada por las demás, la mayor parte de los puntos son negativos. Ahora, si bien la mayor parte de los puntos están por debajo del 0, las diferencias son notablemente

pequeñas, lo cual venimos observando desde la primer experiencia.

Medimos por último los tiempos de ejecución:

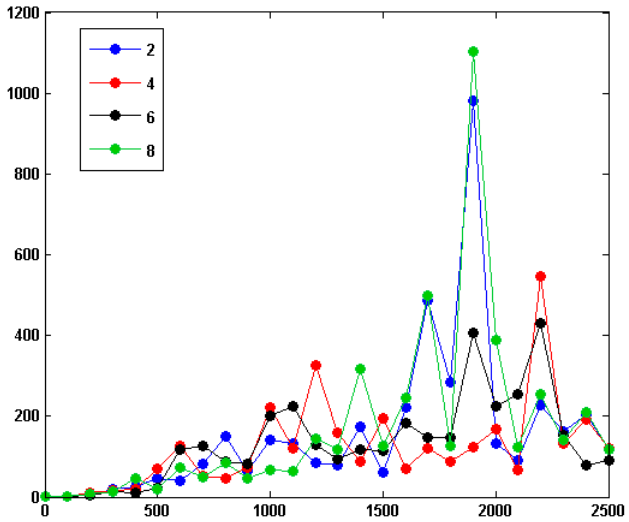


Figura 5.8: Tiempo de ejecución (m creciente, 50 nodos en cada partición)

En este gráfico volvemos a observar un buen desempeño de la versión 6, y además vemos como el método 8 y 2 se encontraron con un caso que no pudieron mejorar rápidamente y obtuvieron un tiempo muy alto.

5.3.5. Conclusiones

Las experiencias son la viva imagen de la complejidad del problema a atacar. Si bien en teoría los diferentes criterios y parámetros parecen afectar de forma sustancial los resultados, no se observa ningún patrón claro en los gráficos, a excepción de la mejora en tiempo que acarrean los criterios de parada adaptativos por sobre los constantes.

Esencialmente, lo que observamos es que si bien hay diferencias entre las distintas combinaciones de parámetros, en general no existe un ganador contundente. No obstante, en ambas experiencias la combinación α *adaptativa, posición aleatoria y parada adaptativa* se mostró como una buena opción, tanto a nivel de mejora en la cantidad de cruces, como a nivel de tiempo de ejecución.

Creemos que esto tiene bastante sentido, porque el criterio adapatativo, como explicamos antes, se basa en que si bajó mucho el número de cruces, es de esperarse que sea menos probable seguir mejorando. De esta manera, suele dar tiempos mas bajos que un criterio estático.

Por otro lado un α adaptativo trata de usar las mejores soluciones que genera una lista restrictiva, ampliándola a medida que esta se va agotando.

Finalmente, elegir la posición de inserción al azar nos brinda en general una familia más amplia para un mismo α, y de esta manera permite encontrar soluciones que con el criterio de primer posición no se encontrarían por quedar fuera del universo alcanzable.

Concluyendo, decidimos tomar estos parámetros para construir nuestro GRASP.

5.4. Pseudocódigo

Algoritmo 12 Propone un dibujo mediante la metaheurística GRASP

```
1: solActual ← construir solución mediante la heurística constructiva y mejorarla mediante la búsqueda local.
2: crucesActual ← cantidad de cruces de la solución propuesta
3: iteraciones ← 0
4: maxIteraciones = cantidad de nodos
5: α ← 0.95
6: Mientras iteraciones ¡maxIteraciones hacer
7:    nuevoDibujo ← construir un dibujo con la heurística constructiva randomizada con α (seleccionando cada
      vez una posición al azar de entre las mejores), y aplicar búsqueda local
8:    nuevosCruces ← cantidad de cruces de nuevoDibujo
9:    Si nuevosCruces < crucesActual entonces
10:      solActual ← nuevoDibujo
11:      crucesActual ← nuevosCruces
12:      maxIteraciones ← maxIteraciones / 2
13:   Si no
14:      iteraciones ← iteraciones + 1
15:      α ← minimo(α - 0.02,0)
16:   Fin si
17: Fin mientras
18: Devolver solActual
```

5.5. Cálculo de complejidad

Examinemos paso por paso el algoritmo. Lo primero que hacemos es crear una primer solución mediante la heurística constructiva y mejorarla con nuestra heurística de búsqueda local. El orden de hacer esto es $O(v_{max}^2 * m * log(v_{max}) * m^2 + Moviles * v_{max}^2 + m * log(fijos_{max}) + fijos_{max} + (V_1 + V_2 + m)))$.

Contar los cruces de esta solución tiene un costo $O(m * log(v_max))$, pero este costo es absorbido por la construcción de la solución inicial.

Luego comenzamos a iterar: cada iteración tiene el costo de las heurísticas, más el del conteo de cruces, que por lo que vimos hace instantes es en total $O(v_{max}^2 * m * log(v_{max}) * m^2 + Moviles * v_{max}^2 + m * log(fijos_{max}) + fijos_{max} + (V_1 + V_2 + m))$. Esto lo hacemos cada vez que iteramos. En el peor de los casos, nunca logramos hacer ninguna mejora y por lo tanto iteramos tantas veces como nodos hay, o sea un total de $O(v_max)$ iteraciones. Luego el costo total de la heurística GRASP es:

$$O(v_{max} * (v_{max}^2 * m * log(v_{max}) * m^2 + Moviles * v_{max}^2 + m * log(fijos_{max}) + fijos_{max} + (V_1 + V_2 + m)))$$

Hay que notar que en un caso más favorable, se lograrán muchas mejoras haciendo que la cantidad de iteraciones no sea lineal en v_max sino de orden logarítmico.

En función del tamaño de la entrada, sabemos que:

$$t = log(P_1) + \sum_{i=1}^{P_1} log((p_1)_i) + log(P_2) + \sum_{i=1}^{P_2} log((p_2)_i) + log(m_p) + \sum_{i=1}^{m_p} log((e_i)_0) + log((e_i)_1)$$

$$+log(IV_1) + \sum_{i=1}^{IV_1} log((iv_1)_i) + log(IV_2) + \sum_{i=1}^{IV_2} log((iv_2)_i) + log(m_{iv}) + \sum_{i=1}^{m_{iv}} log((e'_i)_0) + log((e'_i)_1)$$

Usando esto, más el cálculo hecho para la complejidad de la búsqueda local en función de la entrada, podemos ver que el orden es $O(t^6 * log(t))$.

5.6. Análisis experimental

5.6.1. Casos patológicos

Para determinar un caso malo para nuestra heurística GRASP, lo que tenemos que buscar es algún mal caso de la heurística constructiva que la heurística de búsqueda local no pueda resolver correctamente. Con un caso alcanzará puesto que si bien hay un factor de aleatoriedad en la elección de las posiciones de inserción de los nodos, para un análisis de peor caso es válido suponer que el azar conllevará siempre la elección de la peor opción posible.

Consideremos entonces el ejemplo de caso malo para la constructiva. Recordemos como era:

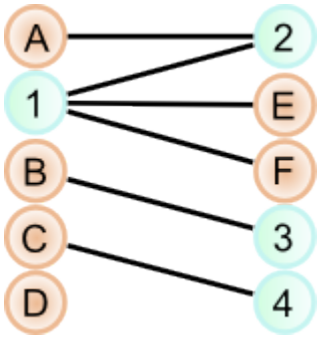
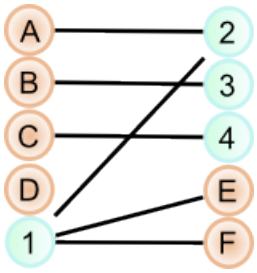
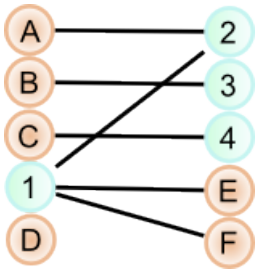


Figura 5.9: Caso patológico para la heurística constructiva

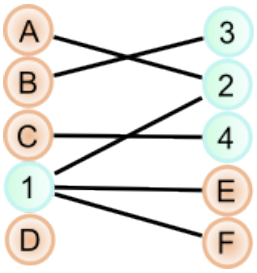
Ahora apliquemos la búsqueda local para ver que resultado obtenemos:



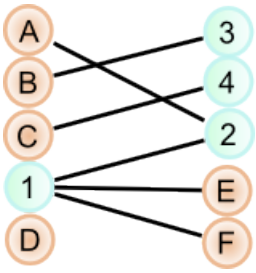
(a) Partimos del dibujo que produce la heurística constructiva



(b) Para los nodos fijos, no se puede hacer nada. Al nodo 1 lo cambiamos de posición pero no porque baja la cantidad de cruces, sino porque en caso de empate, la búsqueda local elige la primer posición visitada



(c) Al nodo 2 no se lo mueve porque no cambia el número de cruces. El nodo 3 tampoco cambia el número de cruces, pero se lo mueve por lo dicho antes del criterio de elección de posiciones

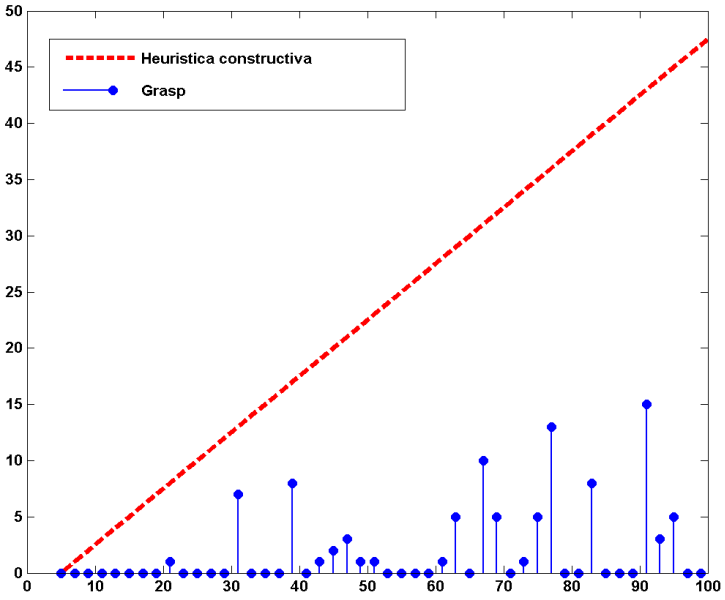


(d) Finalmente se mueve al nodo 4 pero tampoco baja el número de cruces

Como el número de cruces no cambió, consideramos que la búsqueda local llegó a un mínimo local y no se vuelve a intentar mejorar al dibujo. Sin embargo, como vimos en 3.7.1 el dibujo se podía lograr con 0 cruces. Entonces, si consideramos la misma familia que hacía fallar a la heurística constructiva, observamos que la búsqueda local no logra mejorar los dibujos que aquella genera, de modo que el GRASP fallaría de la misma manera que la heurística constructiva frente a estos casos.

Si bien es cierto que en el peor caso siempre se elige de la misma manera a los nodos a insertar, hay que considerar que en la práctica, con un α suficientemente bajo como para dar una lista de candidatos adecuadamente grande, es muy poco probable que se repita siempre la peor elección, y mas aún si se inserta en una posición aleatoria cada vez. Es por esto que en experimentos prácticos el comportamiento de las metaheurísticas con componentes aleatorios suele ser favorable.

Para estudiar el comportamiento del GRASP frente a casos que las otras heurísticas resuelven muy mal, decidimos aplicarlo a casos de esta familia. Los resultados son los siguientes:



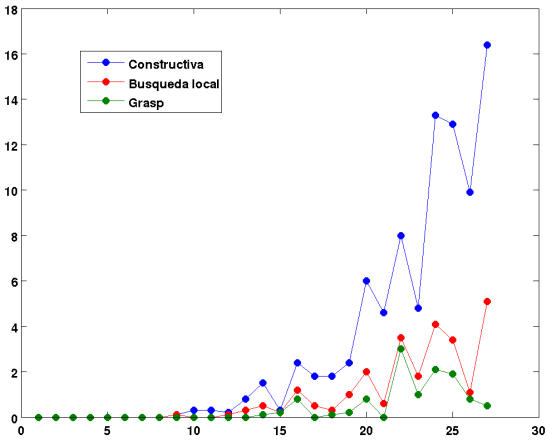
Como vemos, si bien hay casos donde no se logra el óptimo, en general se obtiene un resultado considerablemente mejor que con las heurísticas originales.

5.6.2. Comparativa de heurísticas

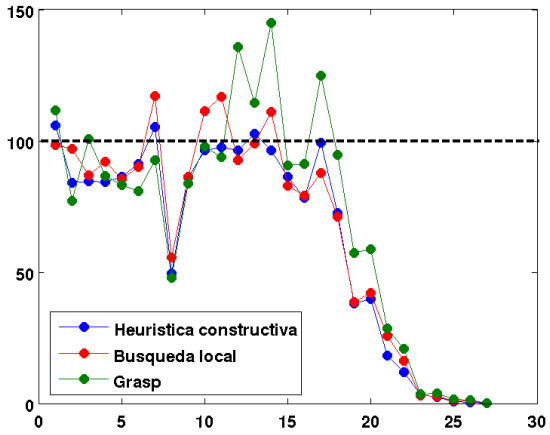
En este apartado realizamos, a modo de experimento final, una comparación entre las distintas heurísticas desarrolladas a lo largo del trabajo.

Primero realizamos experiencias en casos pequeños, para comparar a las heurísticas contra el algoritmo exacto. Estudiamos qué impacto tiene en su performance el aumento del número de nodos, de la densidad del grafo y del porcentaje de nodos móviles.

En la primera experiencia medimos tiempos y cruces en función de la cantidad de nodos totales del grafo.



(a) Diferencia en la cantidad de cruces de las heurísticas y el algoritmo exacto



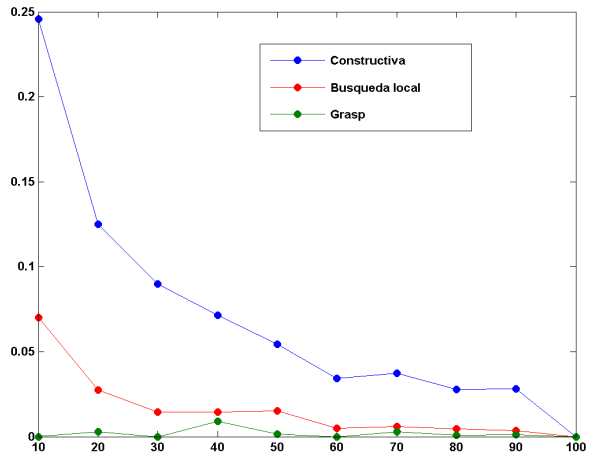
(b) Porcentaje del tiempo del exacto utilizado por las heurísticas

En la primera gráfica vemos como el GRASP logra obtener una gran cantidad de dibujos óptimos, y como en general está siempre muy cerca del valor óptimo. Por otro lado, vemos como efectivamente representa una mejora con respecto a la búsqueda local y a la heurística constructiva aplicadas individualmente.

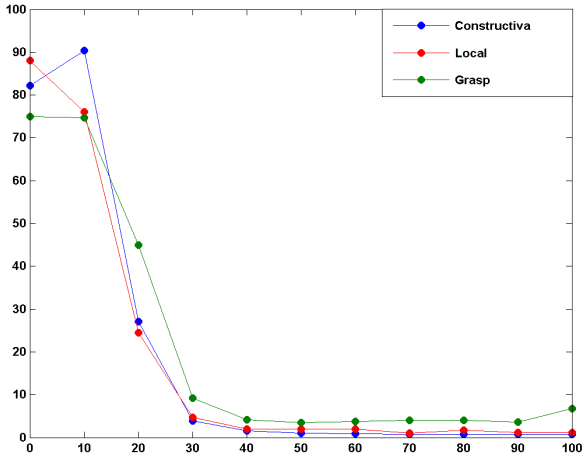
Con respecto a los tiempos, lo llamativo es que en casos pequeños los algoritmos heurísticos tardan más que el algoritmo exacto, lo cual nos indica que los mismos deberían ser aplicados a grafos grandes, donde la diferencia de tiempos se incline a favor de las heurísticas.

Queremos hacer notar, que en algunos casos se observa que la búsqueda local tarda más que la constructiva, a pesar de que una ejecución de búsqueda local conlleva una ejecución de la heurística constructiva para generar el primer candidato. Esto se explica porque las corridas fueron realizadas por separado y al ser los tiempos de esos casos muy pequeños, las anomalías de medición producen esas aparentes inconstistencias.

Las siguientes pruebas se realizaron en grafos de 10 nodos en cada partición, aumentando la cantidad de ejes.



(a) Error relativo de la cantidad de cruces encontrada por las heurísticas

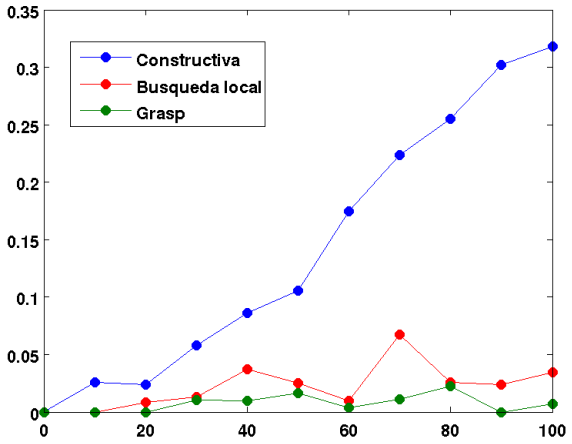


(b) Porcentaje de tiempo del algoritmo exacto empleado por las heurísticas

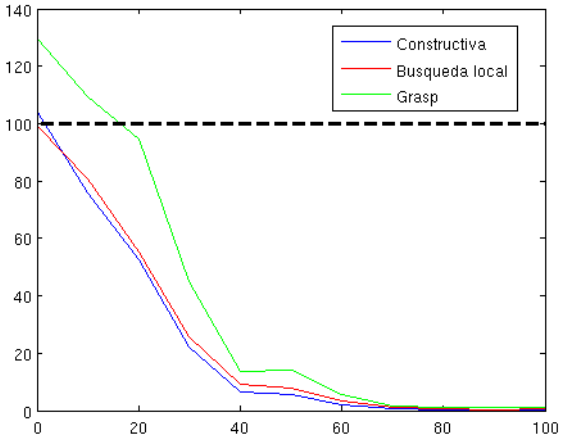
El gráfico del error relativo nos deja ver nuevamente como, si bien las tres heurísticas logran un buen desempeño, el GRASP se comporta destacablemente bien, a tal punto que en todas las mediciones logró siempre un error relativo menor a 0.025.

Por otro lado, vemos como aún en el caso trivial de 0 ejes, las heurísticas logran mejores tiempos. También vemos como a medida que aumenta la densidad del grafo se hace más conveniente utilizar las heurísticas, ya que el *backtracking* se muestra muy sensible a un aumento en la cantidad de ejes.

A continuación corrimos los algoritmos en grafos de 8 nodos en cada partición, y observamos el tiempo necesario así como el desempeño.



(a) Error relativo de la cantidad de cruces encontrados por las heurísticas

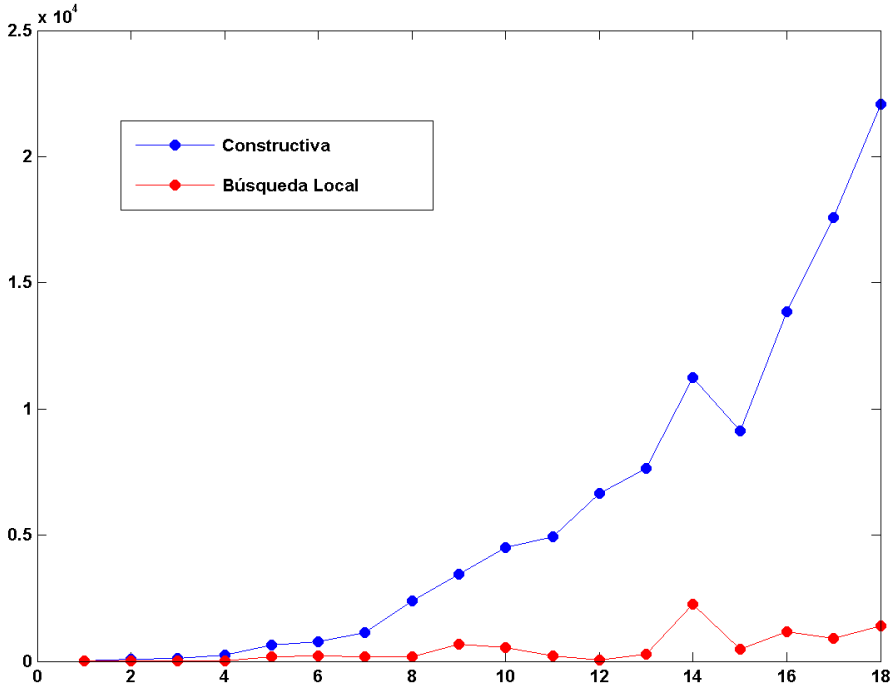


(b) Porcentaje de tiempo del algoritmo exacto usado por las heuristicas

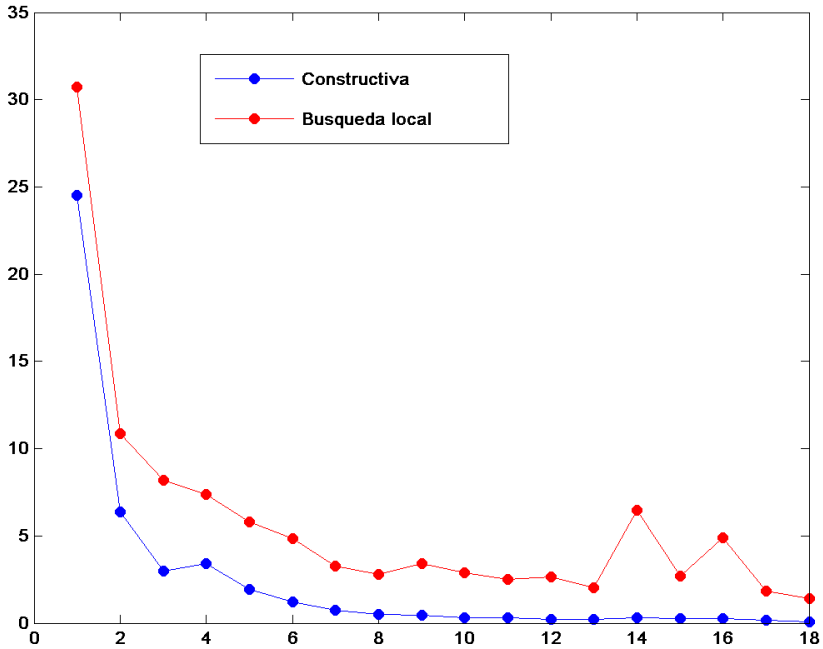
Lo que vemos es que a medida que aumenta el número de nodos móviles, peor se torna la solución que logra la heurística constructiva por sí sola. Creemos que eso se debe a que mientras más nodos tienen que ser insertados, más posibilidades hay de que la heurística tome una decisión incorrecta. Sin embargo, es de destacar como la búsqueda local y el GRASP mejoran notablemente el desempeño de la constructiva.

En lo que a tiempos se refiere, se ve que el *backtracking* es mucho más sensible que las heurísticas a un aumento de la proporción de nodos móviles. También vemos que nuevamente se hace preferible usar los algoritmos heurísticos si la proporción es muy alta, aún en casos relativamente chicos como los de esta experiencia.

Finalmente, decidimos comparar a las tres heurísticas en casos mas grandes, en los que el exacto no puede encontrar la solución correcta en tiempos razonables. Generamos entonces grafos aleatorios con una cantidad de nodos en cada partición que va de 30 a 200 y medimos tiempos y cruces.



(a) Diferencia entre la cantidad de cruces de las heurísticas simples y del GRASP



(b) Porcentaje de tiempo del GRASP usado por las otras heurísticas

En esta experiencia se marca claramente el compromiso entre calidad de la solución y cantidad de cruces. Si bien las heurísticas de búsqueda local y constructiva son mucho más rapidas que el GRASP (lo cual es más que lógico ya que son parte de cada iteración de este último), también es cierto que dan una solución considerablemente peor. Entonces, si bien el GRASP es más lento que ellas, es una alternativa viable para buscar soluciones buenas (sus soluciones son siempre mejores que las de las otras heurísticas) en un tiempo razonable. Una vez más, qué método es preferible utilizar dependerá del contexto de uso, ya que dependiendo de los casos podría ser conveniente conformarse con la precisión de las heurísticas simples para otener un mejor rendimiento temporal.

Parte 6

Conclusión

6.1. Posibles extensiones

Por razones de tiempo, quedaron diversas experiencias y optimizaciones a los algoritmos presentados que no se pudieron realizar. Entre estas contamos:

- Desarrollar otros algoritmos que nos permitan contar cruces mas rápidamente
- Buscar criterios más sofisticados de poda para el algoritmo de *backtracking*
- Intentar disminuir la complejidad espacial y temporal del *backtracking*, principalmente en lo que se refiere al cálculo de la cota inferior
- Proponer nuevas heurísticas constructivas y de búsqueda local que pudieran ser combinadas con las presentadas en este informe para evitar los casos patológicos.
- Realizar un estudio más extensivo sobre los diversos parámetros que hacen al funcionamiento del GRASP, principalmente en lo relacionado con el criterio de parada, a fin de optimizar su rendimiento para distintos escenarios (mayor necesidad de velocidad, de precisión, o alguna relación particular entre ambas).
- Proponer alguna alternativa basada en otra metaheurística, como ser un algoritmo genético para enfrentar este mismo problema
- Diseñar una estructura más eficiente que nos permita mejorar los órdenes de complejidad de los algoritmos heurísticos.

6.2. Conclusiones globales

Durante este trabajo pudimos desarrollar un algoritmo exacto para la resolución del problema del dibujo incremental de grafos bipartitos. Sin embargo, si bien consideramos que este algoritmo se comporta bien dada su naturaleza factorial, no es capaz de resolver instancias medianamente complejas tiempos aceptables, por lo cual propusimos distintas heurísticas.

Pudimos estudiar estas heuristicas, considerando la relación del costo temporal contra la optimalidad de los resultados. Construímos además un algoritmo basado en la metaheurística GRASP, el cual mostró dar buenos resultados en un tiempo razonable, aún para instancias muy grandes del problema.

En general tuvimos que afrontar problemas de organización que se hicieron presentes dada la dificultad del problema. Con problemas de estas características, puede resultar impredecible el tiempo necesario para implementar un algoritmo heurístico y obtener resultados correctos. Desde este punto de vista, fue muy positivo realizar primero prototipos en un lenguaje de alto nivel, lo cual nos dio la libertad de hacer pruebas con mayor rapidez.

Teniendo todo esto en cuenta, consideramos que el trabajo fue interesante porque nos permitió abordar una solución a un problema complejo mediante algoritmos heurísticos, y familiarizarnos con las dificultades relacionadas con este tipo de soluciones.

Bibliografía

[1] Simple and Efficient Bilayer Cross Counting, Wilhelm Barth, Petra Mutzel, Michael Junger, Journal of Graph Algorithms and Applications, vol. 8, no. 2, pp. 179-194 (2004).

[2] A New Lower Bound for the Bipartite Crossing Number with Applications, Farhad Shahrokhi, Ondrej Sýkora, László Székely, Imrich Vrt'o

[3] An Efficient Implementation of Sugiyama's Algorithm for Layered Graph Drawing, Markus Eiglsperger, Martin Siebenhaller, Michael Kaufmann. Journal of Graph Algorithms and Applications http://jgaa.info/ vol. 9, no. 3, pp. 305-325 (2005)

[4] 2-Layer Straightline Crossing Minimization: Performance of Exact and Heuristic Algorithms, Michael Jünger, Petra Mutzel. Journal of Graph Algorithms and Applications http://www.cs.brown.edu/publications/jgaa/ vol. 1, no. 1, pp. 1-25 (1997)

[5] Heuristics, Experimental Subjects, and Treatment Evaluation in Bigraph Crossing Minimization. Matthias Stallmann, Franc Brglez, and Debabrata Ghosh. North Carolina State University

[6] cHawk: An Efficient Biclustering Algorithm based on Bipartite Graph Crossing Minimization. Waseem Ahmad, Ashfaq Khokhar.

[7] Fixed Linear Crossing Minimization by Reduction to the Maximum Cut Problem. Christoph Buchheim and Lanbo Zheng.