Algoritmos y Estructura de Datos III

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

Trabajo Práctico 3

Integrante	LU	Correo electrónico
Candioti, Alejandro	784/13	amcandio@gmail.com
Maldonado, Kevin	018/14	maldonadokevin11@gmail.com
Tripodi, Guido	843/10	guido.tripodi@hotmail.com

Reservado para la cátedra

Instancia	Docente	Nota
Primera entrega		
Segunda entrega		

Contents

1 Ejercicio 1

1.1 Enunciado del problema

Diseñar e implementar un algoritmo exacto para el caso en el que cada materia tiene un máximo de 2 aulas posibles donde se podría dictar. Este problema, donde cada vértice tiene un máximo de 2 colores disponibles se conoce como 2-List Coloring y tiene solución en tiempo polinomial, por lo tanto, el algoritmo que ustedes implementen debe tener solución en tiempo polinomial.

1.2 Explicación de resolución del problema

Para resolver este problema, tuvimos que reducirlo a un problema con conocida solución polinomial llamado 2-SAT. Para pasar de un problema a otro podemos pensar, para cada nodo i, la siguiente preposición p_i: "El nodo está pintado del primer color". Como todo nodo tiene que estar pintado, podemos afirmar que si la preposición anterior es falsa, implica que el nodo está pintado del segundo color.
Ahora bien, tomemos cualquier par de nodos conectados i, j tales que estos comparten algún color entre sus opciones (supongamos el primer color sin pérdida de genearalidad). Entonces, para colorear ambos nodos, necesitamos que se cumpla la siguente preposición: $p_i \Rightarrow \neg p_j \wedge p_j \Rightarrow \neg p_i$.
En conclusión, para generar una coloracion posible de dichos nodos, tenemos que encontrar una asignación de valores lógicos (verdadero o falso) a cada preposición p_i tal que se cumpla, para todo par de nodos conectados y compartiendo por lo menos un color, las preposiciónes indicadas anteriormente. Como cada preposición de la forma $p \Rightarrow q$ se puede expresar como $\neg p \vee q$, el problema anterior se puede reducir a 2-SAT.

Para su resolución, realizamos al algoritmo explicado en la cátedra. Cuya complejida es $O(E)$ para transformar el input al teorema de 2-SAT y $O(E+V)$ para encontrar el coloreo. La complejidad total termina siendo $O(E+V)$ que es polinomial con respecto al tamaño de la entrada.

2 Ejercicio 2

2.1 Enunciado del problema

Diseñar e implementar un algoritmo exacto para List Coloring y desarrollar los siguientes puntos:

1. Explicar detalladamente el algoritmo implementado. Elaborar podas y estrategias que permitan mejorar los tiempos de ejecución. En los casos en los que el backtracking reduzca el problema a 2-List Coloring utilizar el algoritmo del item anterior como caso base.

2. Calcular el orden de complejidad temporal de peor caso del algoritmo.

3. Realizar una experimentación que permita observar los tiempos de ejecución del algoritmo en función del tamaño de entrada y de las podas y/o estrategias implementadas.

2.2 Explicación de resolución del problema

Para este problema realizamos un backtracking que recorre todas las maneras de pintar el grafo hasta que encuentra una coloración. Si se recorren todas las coloraciones posibles y no se encuentra ninguna, el algoritmo indica que no hay coloración.

El algoritmo elegido, al ser un backtracking, pinta de algún color un nodo y luego llama recursivo para el resto del grafo. Cuando sale de esa llamada recursiva, si esta indica que no pudo resolverto prueba otro color en ese nodo y repite la llamada recursiva. Así hasta quedarse sin opciones.

La forma más inocente de realizar este algoritmo es generar todas las coloraciones, y para cada una chequear si no tiene conflictos. Esto claramente no es lo mejor, porque al no estar realizando podas, terminamos evaluando todas las posibilidades.
En nuestra solución, realizamos las siguientes podas:

- Cuando fijamos un color a un nodo, eliminamos ese color como opción a todos sus vecinos. De esta manera, no pasamos por soluciones que generan conflictos antes de terminar de pintar todo el grafo.

- Cuando fijamos un color a un nodo y alguno de sus vecinos se queda sin opciones, volvemos atras. Cabe aclarar que esta poda no es equivalente a la anterior, ya que en la otra recién descartariamos la solución al intentar colorear el nodo sin opciones.

- Llamamos "color facil" al color de un nodo tal que no se encuentra entre los colores de sus vecinos ya pintados ni entre las opciones de color de sus vecinos aún no pintados. Si dicho color existe para algun nodo, lo mejor es pintarlo de ese color, ya que no generará nuevos conflictos cuando intentemos pintar los demás nodos. La poda que añadimos al algoritmo equivale a hacer esto mismo, es decir, si el nodo que estamos intentando pintar tiene algún "color fácil", lo pintamos de ese color sin pasar por las demás posibilidades.

Por otra parte, una estrategia que decidimos aplicar es pintar en cada paso el nodo con más opciones de colores. Si bien lo más natural parece intentar colorear los nodos con menos opciones, esta estrategia nos permite llegar en menos pasos al caso en que todos los nodos tienen a lo sumo dos opciones, que lo resolvemos en tiempo polinomial con la ayuda del algoritmo del ejercicio 1.

En cuanto a la implementación del algoritmo realizamos la manera clásica de implementar un algoritmo de backtracking. Es decir, con llamadas recursivas pasando el estado actual por parámetro. Para la implementación utilizamos las siguientes estructuras:

- *Graph*: Explicada en la sección de estructuras de datos, permite:
  - Colorear un nodo y actualizar los posibles colores de sus vecinos.
  - Devuelve las opciones actuales de un nodo sin pintar. Complejidad $O(1)$.

  Estas dos operaciones son cruciales para recorrer todas las posibilidades en el backtracking y realizar una de las podas antes explicadas, ya que las "opciones actuales" de un nodo se calculan teniendo en cuenta las restricciones que presentan los nodos vecinos ya pintados.
- *ValueSortedMap*: Explicada en la sección de estructuras de datos, permite:
- Asociar valores a claves.
- Darnos la clave con menor valor asociado.
- Dar y eliminar la clave con el mayor valor asociado. Esta estructura nos sirve para muchas podas y estrategias. Nos indica si hay un nodo sin opciones, nos indica si el problema es reducible al problema 2-ListColoring y nos ayuda a elegir en cada paso el nodo con mayor cantidad de opciones.

2.3 Análisis de complejidades

La complejidad del peor caso del algoritmo, al ser un backtracking, depende principalmente de la cantidad de configuraciones posibles, ya que en el antes nombrado peor caso, se recorren todas las configuraciones posibles.
Por lo tanto, si no tenemos encuenta las podas ni las estrategias, la complejidad del algoritmo resulta ser $O(C^V)$ donde V es la cantidad de vertices y C la cantidad de colores en total.

Ahora bien, las podas y estrategias realizadas modifican la complejidad del algoritmo, analicemoslas por separado:

- Al reducirle las opciones de colores a los vecinos del nodo que acabamos de pintar tenemos que realizar tantas operaciones de agregado a un *Set* ($O(1)$) como vecinos tenga el nodo. Como el nodo tiene como mucho V vecinos, el costo de esta poda es $O(V)$.
- La poda que descarta soluciones en las que algun nodo se quedo sin opciones tiene el costo que tiene consultar cual es el nodo con menor cantidad de opciones. Como utilizamos la estructura *ValueSortedMap* (explicada en secciones posteriores), realizamos esta operación en $O(log(V))$.
- Para encontrar un "color facil" debemos mirar todos los colores de todos los vecinos de un nodo, y guardarlos en un set. Luego, calcular la diferencia entre los colores del nodo y el set antes computado. Asumiendo que las operaciones de un set (hash table) son $O(1)$, calcular el color fácil tiene costo $O(V * C)$.
- La estrategia de tomar el el nodo con mas opciones en cada paso tiene varias etapas:
  - Inicializar la estructura antes de empezar el backtracking. Costo $O(V * log(V))$.
  - Sacar el nodo con mas opciones de la estructura para pintar. Costo $O(log(V))$.
  - Actualizar a los vecinos luego de pintar. Costo $O(V * log(V))$.

Asumiendo el peor de los casos, donde las podas no ayudan a achicar el arbol de busqueda, si consideramos a los estados por los que pasa el backtracking como un arbol de C^V hojas, para cada una de estas hojas habremos realizado V operaciones de podas (por qué ya coloreamos los V vértices). Como las operaciones de podas cuestan $O(V + log(V) + V * C + V * log(V)) = O(V * (C + log(V)))$ en total, y las realizamos V veces, por cada hoja habremos realizado $O(V * (C + log(V)))$ operaciones. Finalmente, como en el backtracking pasamos por C^V configuraciones, la complejidad total resulta $O(C^V * V * (C + log(V)))$.

3 Ejercicio 3

3.1 Enunciado del problema

Diseñar e implementar una heurística constructiva golosa para List Coloring y desarrollar los siguientes puntos:

1. Explicar detalladamente el algoritmo implementado.
2. Calcular el orden de complejidad temporal de peor caso del algoritmo.
3. Describir instancias de List Coloring para las cuales la heurística no proporciona una solución óptima. Indicar que tan mala puede ser la solución obtenida respecto de la solución factible.
4. Realizar una experimentación que permita observar la performance del algoritmo en términos de tiempo de ejecución en función del tamaño de entrada.

3.2 Explicación de resolución del problema

El algoritmo desarrollado representa a una heurística golosa constructiva.
Este se basa en la idea intuitiva de que es conveniente comenzar pintando primero los nodos con más restricciones, dejando los que menos restricciones tienen para el final.

Esto se debe a que cuando fijamos un color en un nodo, aumentamos las restricciones de sus vecinos, dejando a estos sin este color como opción. Por lo tanto, si realizamos lo contrario y empezamos por los nodos con menor cantidad de restricciones, en cada paso nos encontraremos con nodos con **"pocas"** opciones, lo que hará más probable que se nos agoten las opciones más rapidamente.

En concreto, lo que hace nuestra heurística es mantener ordenadados a los nodos por su cantidad de opciones. Luego, mientras existan nodos sin colorear, toma el de menor cantidad de opciones y lo pinta del color **"más conveniente"**. Por color "mas conveniente" nos referimos al que genera menos conflictos entre sus vecinos ya pintados (si hay mas de uno alguno de ellos). Esto se puede calcular chequeando los colores de los vecinos ya pintados y tomar el que esta dentro de los colores posibles del nodo que aparece menos veces.

Resumiendo, nuestro algoritmo recorre todos los vértices, tomando siempre el de menor cantidad de opciones y a cada uno lo pinta del "mejor color".

En cuanto a implementación, se precisó de una estructura de datos que permitiera tener a los nodos ordenados por su cantidad de opciones, ademas de ir actualizando esa cantidad. La estructura *ValueSortedMap* explicada en su sección correspondiente cumple con estos requisitos.
Por otra parte, se necesitó de la estructura *Graph*, que indicaba las opciones posibles de un nodo (las que no generan conflictos).
Una vez teniendo estas estructuras, el algoritmo se reduce a un ciclo por los nodos, en el cual se pinta del "mejor color" al nodo de la iteración y, como esto modifica la cantidad de colores posibles de los vecinos, se actualiza esta cantidad en la estructura *ValueSortedMap*.

3.3 Análisis de complejidades

Esta información la podemos obtener en O(1), y pintar un nodo del **"mejor color"** que a su vez, modifica las restricciones de sus vecinos. Se realiza en complejidad O(#vecinos del nodo) asumiendo

que el agregar/eliminar/modificar es O(1) en tablas de hash.

El algoritmo recorre todos los vértices, tomando siempre el menor de la estructura de datos antes mencionada y a cada uno de ellos lo pinta del **"mejor color"**. Despues, actualiza a los vecinos en la estructura de datos.
Por ende, la complejidad del algoritmo esta atada a la complejidad de la estructura de datos utilizada.

Nuestra implementación realiza tanto la operación de eliminación del mínimo como la operación de modificación de clave en O(log(tamaño)).
Entonces, en cada iteración realizamos los siguientes pasos:

- Se remueve el mínimo de la estructura con un Costo O(log(V)).
- Se lo pinta del **"mejor color posible"** insumiendo O(#vecinos), como en el peor caso podría estar conectado con todos sus vecinos, equivale a O(V).
- Para cada vecino se actualiza su nueva cantidad de colores posibles o **"grado de libertad"**. Esto tiene costo logarítmico para cada vecino, lo que equivale a O(V * log(V)).

Si sumamos todos estos items, obtenemos un total de O(E * log(V)) por iteración, como realizamos V iteraciones, el algoritmo resulta **O(V * V * log(V))** en el peor caso.

3.4 Instancias desfavorables

Como mirar casos manualmente era una tarea lenta y dificil de realizar, para encontrar casos en los que el algoritmo no funciona, implementamos un generador de grafos aleatorio, y nos quedamos con los que el algoritmo del ejercicio 2 indica que se puede colorear todo el grafo.

A continuación mostraremos algunos de estos casos encontrados con el metodo antes descripto. En estos el algoritmo no resuelve el problema.
Con el siguiente grafo:

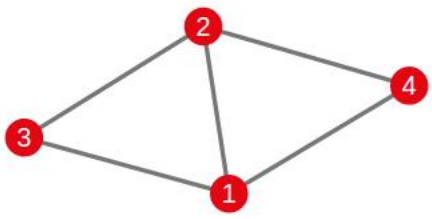


Gráfico 3.3.1 - Grafo con número en nodos sin coloreo

En este grafo las opciones indicadas para coloreo fueron:

- Nodo 1 = Color 0 y Color 1
- Nodo 2 = Color 0 y Color 2
- Nodo 3 = Color 0 y Color 1
- Nodo 4 = Color 0 y Color 2

Obteniendo [0, 2, 1, 0] como resultado para la Heurística que se planteo y [1, 2, 0, 0] para el backtracking que enunciamos del ejercicio 2.

A continuación mostraremos como quedaría coloreado el grafo y como tendría que ser:

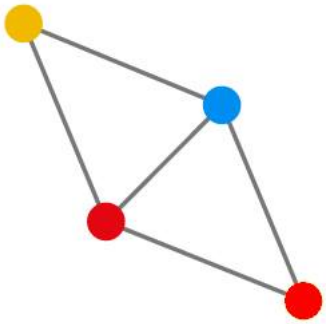


Gráfico 3.3.2 - Grafo con coloreo - Error de Heurística

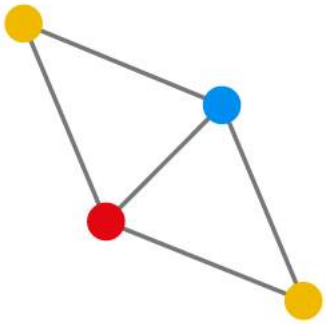


Gráfico 3.3.3 - Grafo con coloreo - Solución Correcta

Luego, con el grafo:

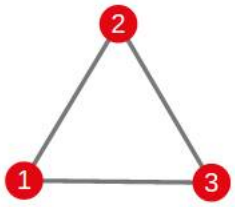


Gráfico 3.3.4 - Grafo triangular con número en nodos sin coloreo

En este grafo las opciones indicadas para coloreo fueron:

- Nodo 1 = Color 1 y Color 2
- Nodo 2 = Color 0 y Color 1
- Nodo 3 = Color 0 y Color 1

Obteniendo [1, 0, 0] como valor final para la Heurística que se planteo y [2, 0, 1] para el backtracking que enunciamos del ejercicio 2.

A continuación mostraremos un ejemplo de como quedó y como tendría que haber quedado coloreado el grafo:

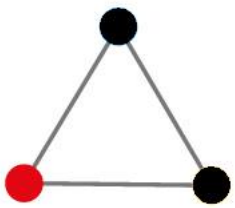


Gráfico 3.3.5 - Grafo con coloreo - Error de Heurística

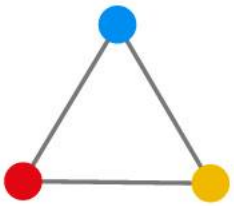


Gráfico 3.3.6 - Grafo con coloreo - Solución Correcta

Por último, con el grafo:

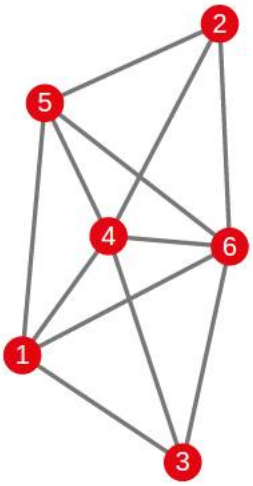


Gráfico 3.3.7 - Grafo con número en nodos sin coloreo

En este grafo las opciones indicadas para coloreo fueron:

- Nodo 1 = Color 0 y Color 1
- Nodo 2 = Color 0, Color 1 y Color 3
- Nodo 3 = Color 2 y Color 3
- Nodo 4 = Color 0, Color 2 y Color 3
- Nodo 5 = Color 0, Color 1 y Color 3
- Nodo 6 = Color 0, Color 2 y Color 3

Obteniendo [0, 1, 2, 3, 1, 0] como valor final para la Heurística que se planteo y [1, 1, 3, 0, 3, 2] para el backtracking que enunciamos del ejercicio 2.

A continuación mostraremos un ejemplo de como terminó quedando coloreado el grafo y como debería quedar:

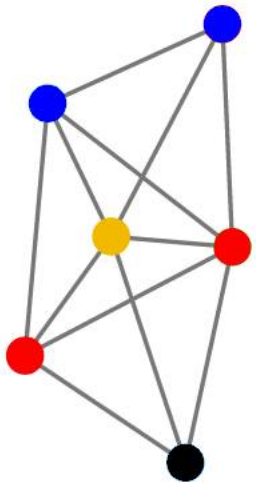


Gráfico 3.3.8 - Grafo con coloreo - Error de Heurística

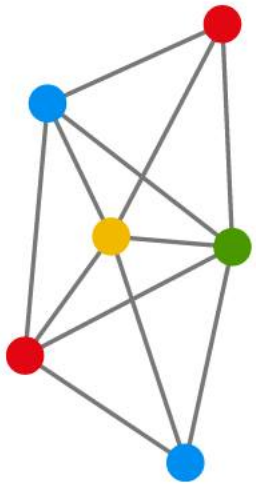


Gráfico 3.3.9 - Grafo con coloreo - Solución Correcta

3.4.1 Performance De Algoritmo y Gráfico

Para corroborar la performance de nuestro algoritmo, trabajamos con distintos tipos de grafos.

Como la complejidad calculada fue O(E * V * LOG(V)) decidimos fijar E y tener V movil. Trabajando con grafos circulares variando el tamaño de entrada entre 0 y 10000 y corriendo la misma

medición varias veces sacamos un promedio del mismo para obtener datos más relevantes y obtuvimos el siguiente resultado:

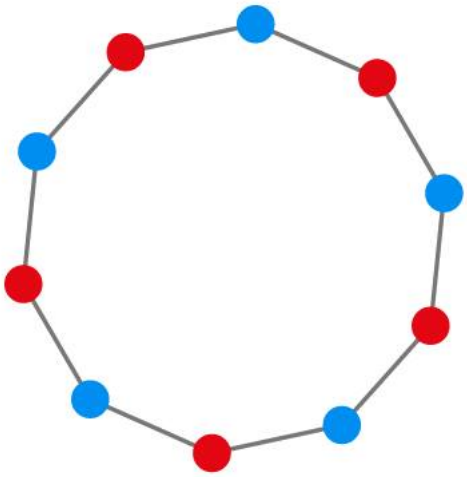


Gráfico 3.4.1 - Grafo circular

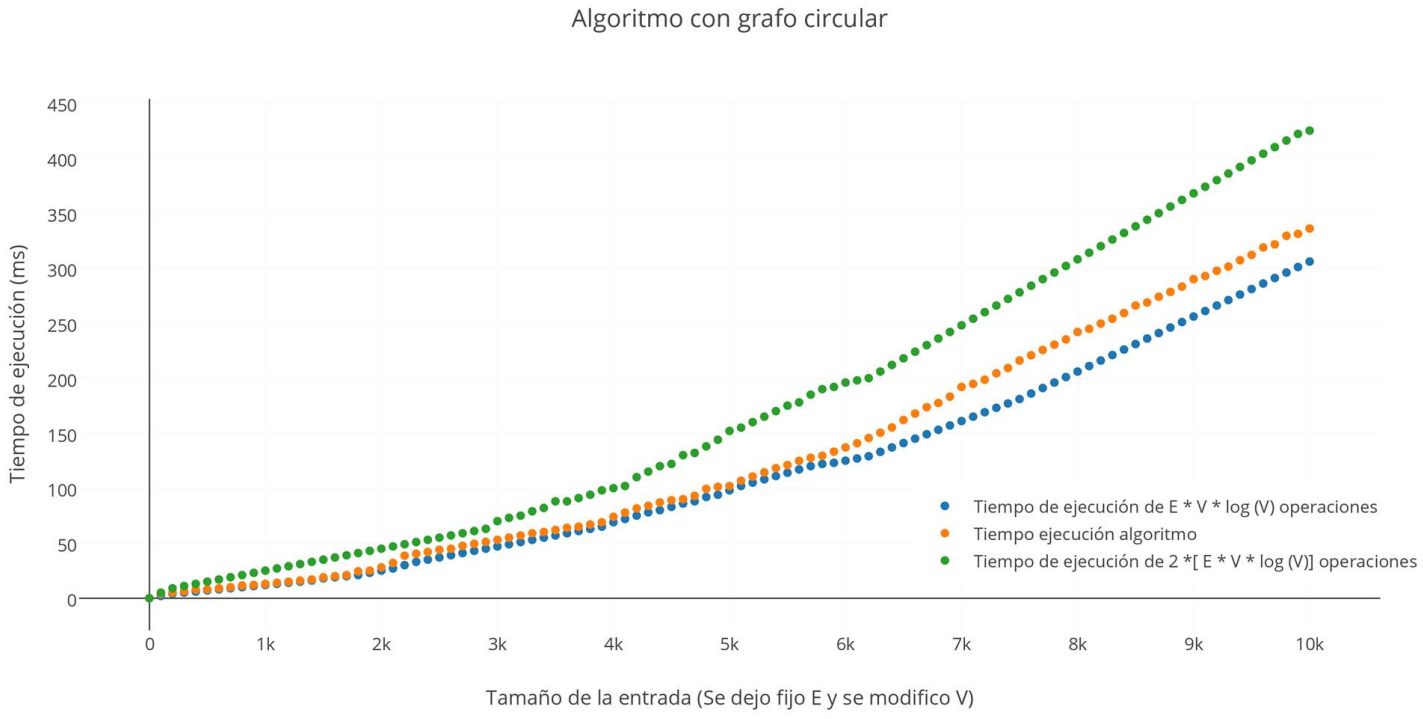


Gráfico 3.4.2 - Medición grafo circular

En la figura 3.4.1 se puede observar el tipo de grafo que utilizamos. En la figura 3.4.2, se puede ver como el tiempo de ejecución de nuestro algoritmo se encuentra entre los tiempos de realizar E * LOG(V) operaciones y 2*[E * LOG(V)] demostrando como en este tipo de grafo la complejidad

calculada de nuestro algoritmo fue correcta.

Trabajando con grafos completos variando el tamaño de entrada entre 0 y 10000 y realizando la misma medición varias veces sacamos un promedio del mismo para obtener datos más consistentes y obtuvimos el siguiente resultado:

Gráfico 3.4.3 - Grafo completo

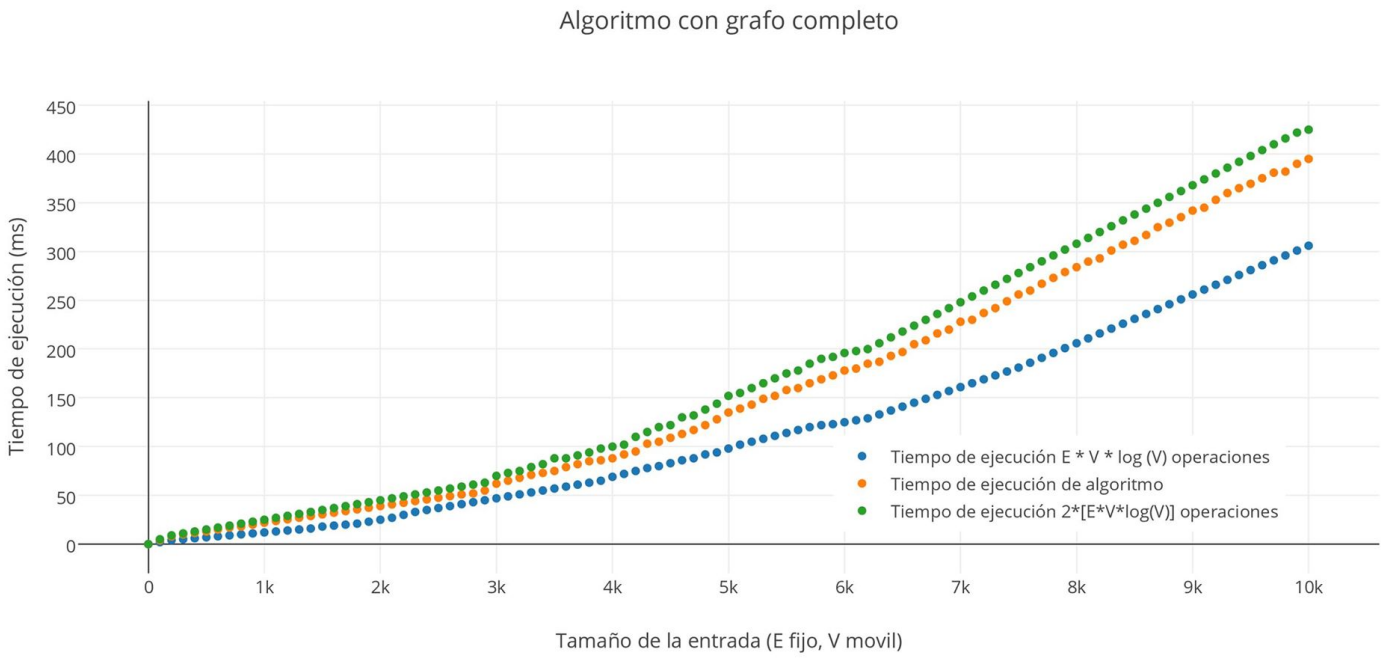


Gráfico 3.4.4 - Medición con grafo completo

En la figura 3.4.3 se puede observar el tipo de grafo que utilizamos. En la figura 3.4.4, se puede ver como el tiempo de ejecución de nuestro algoritmo es un poco mayor que un grafo circular, ya

que tenemos más nodos interconectados y a pesar de esta situación nos mantenemos dentro de los tiempos estipulados corroborando la complejidad calculada.

Trabajando con grafos en el cual todos los nodos estan conectados unicamente con el del centro variando el tamaño de entrada entre 0 y 10000 y corriendo la misma medición varias veces sacamos un promedio del mismo para obtener datos más relevantes y obtuvimos el siguiente resultado:

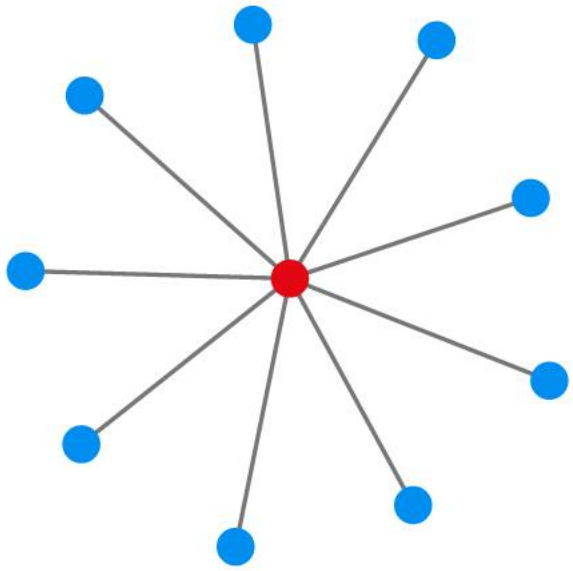


Gráfico 3.4.5 - Grafo único nodo conectado con todos

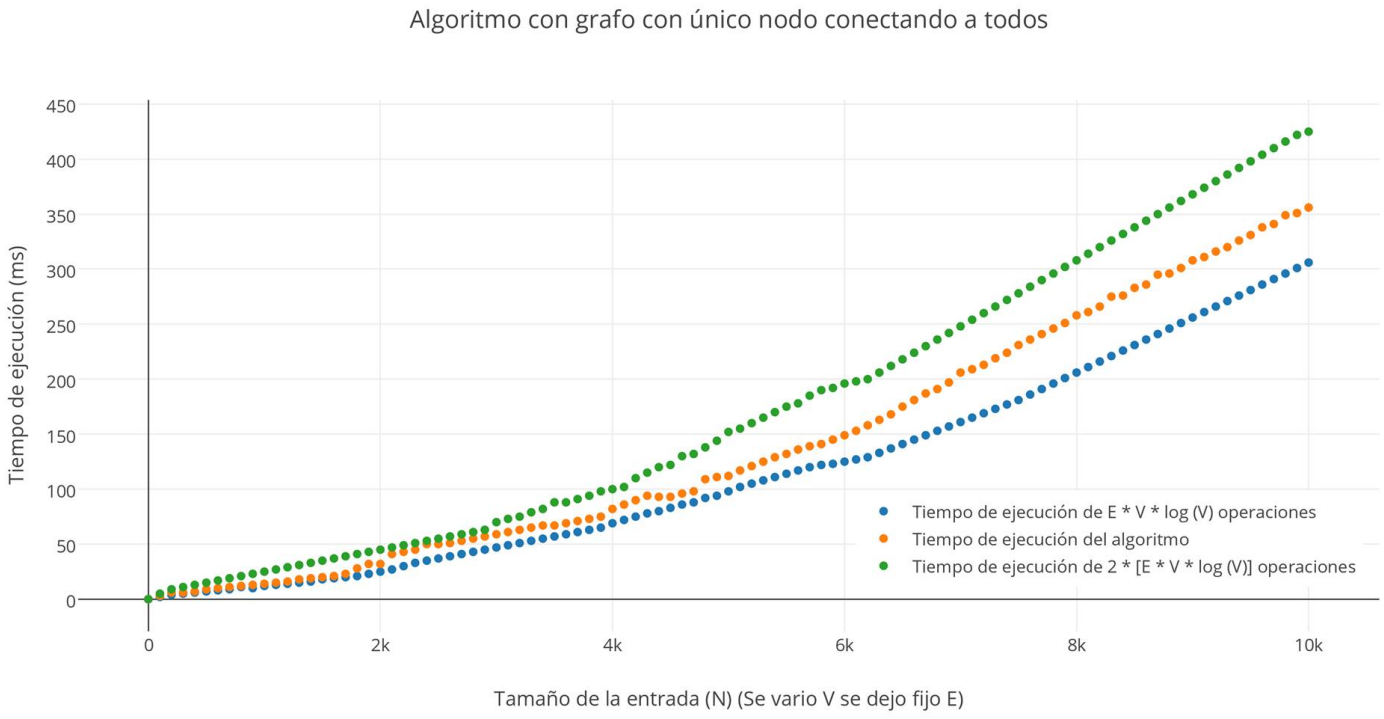


Gráfico 3.4.6 - Medición Grafo único nodo conectado con todos

En la figura 3.4.5 se puede observar el tipo de grafo que utilizamos. En la figura 3.4.6, se puede ver como el tiempo de ejecución de nuestro algoritmo es similar e inferior al grafo circular ya que un único nodo se encuentra conectados con todos

4 Ejercicio 4

4.1 Enunciado del problema

Diseñar e implementar una heurística de búsqueda local para List Coloring y desarrollar los siguientes puntos:

1. Explicar detalladamente el algoritmo implementado. Plantear al menos dos vecindades distintas para la búsqueda.

2. Calcular el orden de complejidad temporal de peor caso de una iteración del algoritmo de búsqueda local (para las vecindades planteadas). Si es posible, dar una cota superior para la cantidad de iteraciones de la heurística.

3. Realizar una experimentación que permita observar la performance del algoritmo comparando los tiempos de ejecución y la calidad de las soluciones obtenidas, en función de las vecindades utilizadas y elegir, si es posible, la configuración que mejores resultados provea para el grupo de instancias utilizado.

4.2 Explicación de resolución del problema

En este algorítmo, se desarrollaron heurísticas que buscan mejorar en cada paso una solución obtenida previamente.

4.3 Vecindades

Se plantearon dos vecindades distintas, donde la segunda se implementó en función de la primera.
La primera (*Exercise4B*) que se nos ocurrió, consiste en pasar de una configuración a otra cambiando un nodo de color, siempre y cuando esto reduzca la cantidad de conflictos de la configuración.
La segunda (*Exercise4*), inspirada en la anterior, modifica el color de un nodo por el que genere la menor cantidad de conflictos con sus vecinos pintados. Es decir, re-utiliza la definicion antes nombrada de "mejor color".

4.4 Inicialización

Para la primera vecindad, decidimos colorear el grafo tomando algun color posible de cada nodo. Para la segunda, en cambio, inicializamos el grafo con la misma lógica con la que nos movemos entre soluciones, es decir, tomamos el color que genere la menor cantidad de conflictos entre los coloreados.

4.5 Algoritmos e Implementación

Los algoritmos, al ser de busqueda local, realizan varias pasadas moviendose en cada una a una solución mejor (dependiendo de la vecindad elegida), es decir, van registrando si en la pasada lograron alguna mejora. El algoritmo termina luego de que en alguna pasada no se haya logrado alguna mejora.

Al ser los algoritmos muy similares, sus implementaciones lo son. Ambas consisten de un ciclo sobre una variable que indica si se puede seguir mejorando mediante más pasadas.

Siendo mas detallistas, se realizan pasadas. Para cada pasada, por cada nodo, se pinta del color que indique alguna de las vecindades elegidas, registrandose a su vez, si en algún momento de la pasada se produjo una mejora en la cantidad de aristas denominadas **"buenas"**. Si luego de la

pasada se registra alguna mejora, se realiza una nueva pasada, de lo contrario, el algoritmo finaliza. (Cabe aclarar que con aristas "buenas" nos referimos a aristas con sus dos nodos de los bordes pintados y con colores distintos).

Para la implementación de estos utilizamos la estructura *Graph* que permite setearle el color a un nodo y actualizar los colores posibles para cada nodo. También, soporta una operación que setea a un nodo del "mejor color" necesaria para la implementación del algoritmo de la segunda vecindad.

Es posible demostrar que ambos algoritmos terminan. Supongamos que el algoritmo no termina, luego, en cada iteración aumenta la cantidad de **"aristas buenas"**, lo que implica que la cantidad posible de éstas no esta acotada. Luego, como la cantidad de aristas buenas es menor a la cantidad total de aristas, que esta acotada, llegamos a un absurdo, que vino de suponer que el algoritmo no finalizaba.

4.6 Análisis de complejidades

Con la idea utilizada en la demostración de finalización de nuestro algoritmo podemos generar una cota para la complejidad de los algoritmos planteados en esta sección.
Si en nuestro algoritmo realizamos K pasadas, quiere decir que mejoramos la solución en $K-1$ pasadas, es decir, aumentamos la cantidad de aristas buenas en $K-1$ pasadas. Como la cantidad de aristas buenas es como mínimo 0 y como mucho E y en cada pasada aumentamos en al menos 1 la cantidad de aristas buenas, realizamos como mucho $O(E)$ pasadas.
Como en cada pasada, iteramos los V nodos, esto nos da que la complejidad de los algoritmos es $O(E*V*X)$ donde X es lo que cuesta procesar un nodo en particular para intentar mejorar la solución.

Para la primer vecindad, en el peor de los casos probamos con todos los colores. Probar con un color hace tantas operaciones como vecinos tenga un nodo, por lo que es $O(E)$, como lo hace para cada color, la complejidad termina siendo $O(E^2 * V * C)$, con C la cantidad total de colores. La complejidad de una iteración resulta $O(V * E * C)$

Para la segunda vecindad, calcular el mejor color y setearlo tiene costo $O(E)$ (hace tantas operaciones como vecinos tenga), por lo que la complejidad termina siendo $O(E^2 * V)$. La complejidad de una iteración resulta $O(V * E)$.

4.7 Experimientos y conclusiones

4.7.1 Test

Para corroborar la performance de nuestro algoritmo del inciso A, trabajamos con distintos tipos de grafos.

Como la complejidad calculada fue O(E * E * V) decidimos fijar V y tener E movil.
Trabajando con grafos circulares variando el tamaño de entrada entre 0 y 10000 y corriendo la misma medición varias veces sacamos un promedio del mismo para obtener datos más relevantes y obtuvimos el siguiente resultado:

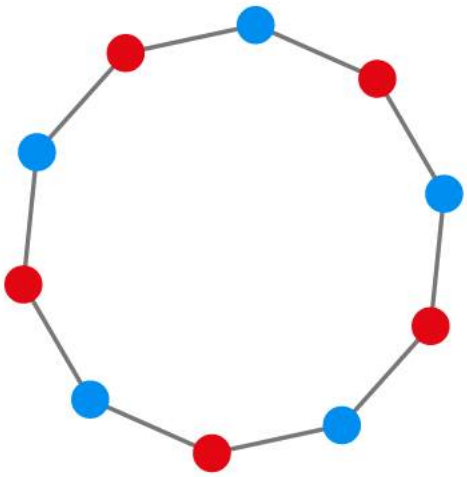


Gráfico 4.4.1 - Grafo circular

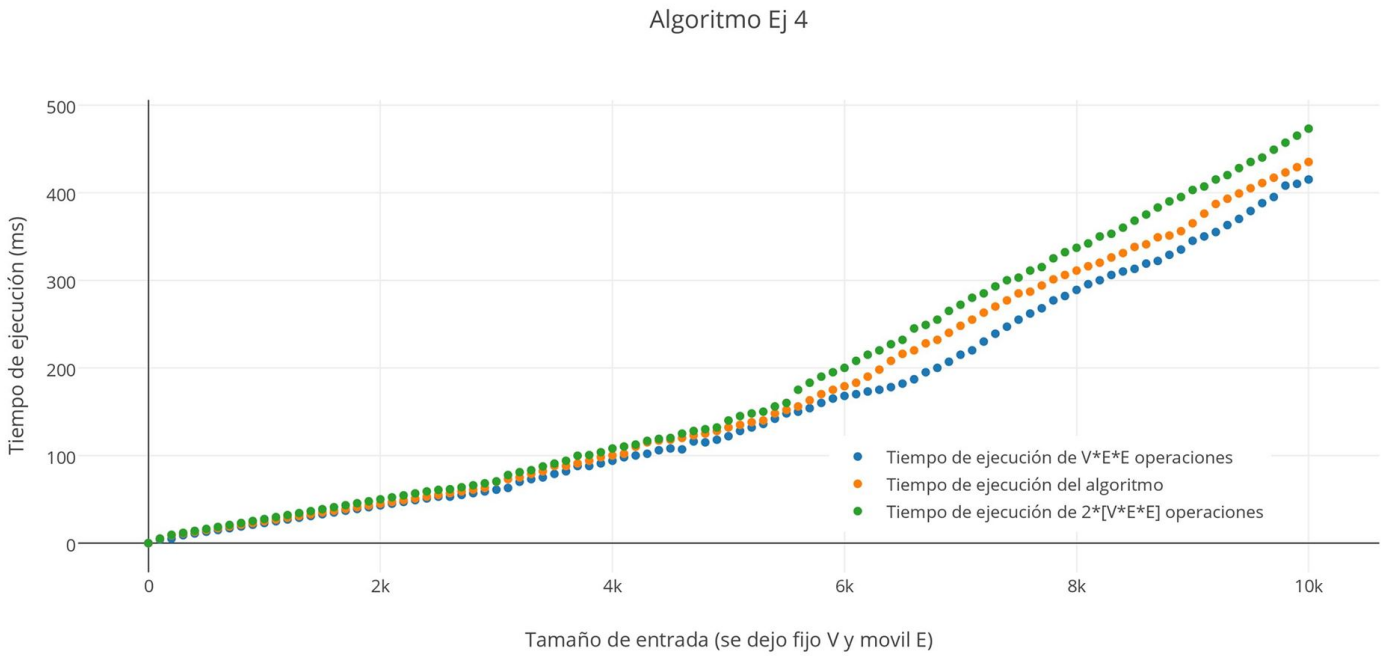


Gráfico 4.4.2 - Medición grafo circular

En la figura 4.4.1 se puede observar el tipo de grafo que utilizamos. En la figura 4.4.2, se puede ver como el tiempo de ejecución de nuestro algoritmo se encuentra entre los tiempos de realizar (E * E * V) operaciones y 2 (E * E * V) demostrando como en este tipo de grafo la complejidad calculada de nuestro algoritmo fue correcta.

Trabajando con grafos completos variando el tamaño de entrada entre 0 y 10000 y realizando la misma medición varias veces sacamos un promedio del mismo para obtener datos más consistentes y obtuvimos el siguiente resultado:

Gráfico 4.4.3 - Grafo completo

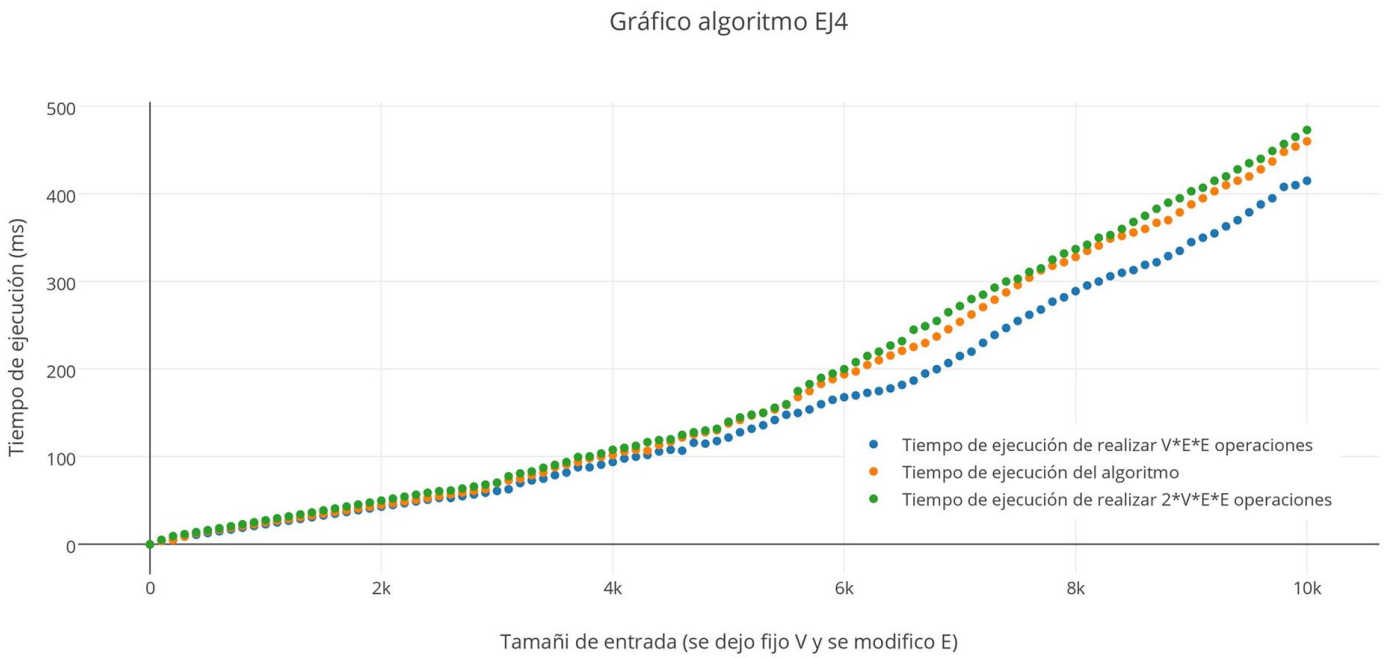


Gráfico 4.4.4 - Medición con grafo completo

En la figura 4.4.3 se puede observar el tipo de grafo que utilizamos. En la figura 4.4.4, se puede ver como el tiempo de ejecución de nuestro algoritmo es un poco mayor que un grafo circular, ya que tenemos más nodos interconectados y a pesar de esta situación nos mantenemos dentro de los tiempos estipulados corroborando la complejidad calculada.

Para el algoritmo del inciso B decidimos utilizar los mismos tipos de gráficos para realizar las respectivas mediciones y luego realizamos un gráfico comparativo de los mismos para chequear que tan "buenas" resultaban las soluciones.

Se detalla a continuación lo enunciado:

En este algoritmo la complejidad teórica resultante fue O(E * E * V * C), y por lo tanto decidimos fijar V y C y tener E movil.

Trabajando con grafos circulares variando el tamaño de entrada entre 0 y 10000 y corriendo la misma medición varias veces sacamos un promedio del mismo para obtener datos más consisos y obtuvimos el siguiente resultado:

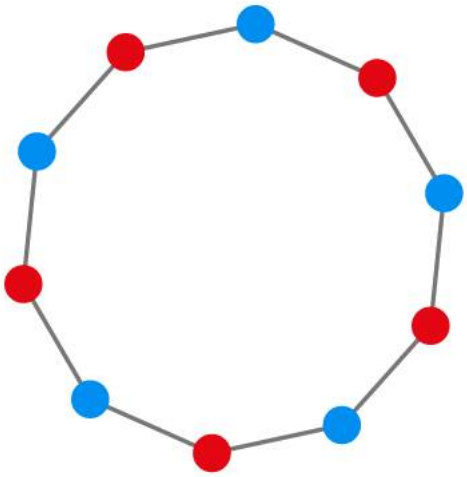


Gráfico 4.4.4 - Grafo circular

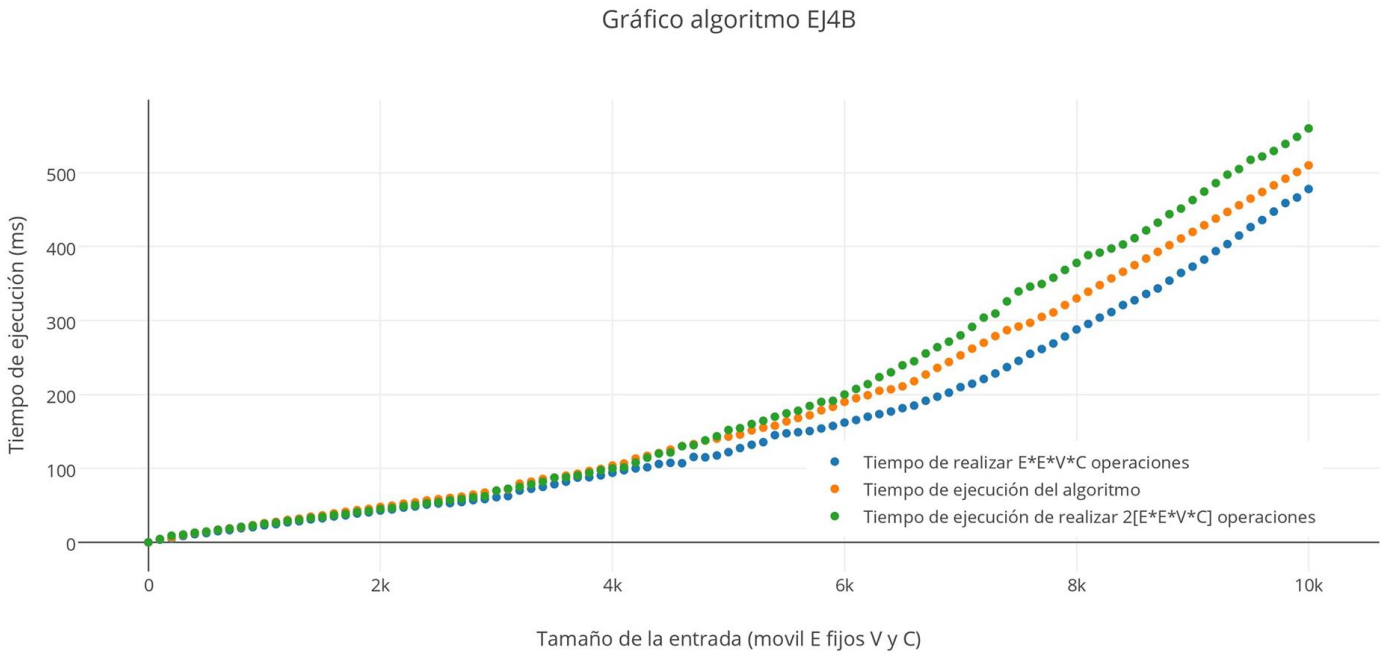


Gráfico 4.4.5 - Medición grafo circular

En la figura 4.4.4 se puede observar el tipo de grafo que utilizamos. En la figura 4.4.5, se puede ver como el tiempo de ejecución de nuestro algoritmo se encuentra acotado por los tiempos de realizar (E * E * V * C) operaciones y 2 (E * E * V * C) demostrando como en este tipo de grafo la complejidad calculada de nuestro algoritmo fue correcta.

Trabajando con grafos completos variando el tamaño de entrada entre 0 y 10000 y realizando la misma medición varias veces sacamos un promedio del mismo para obtener datos más consistentes y obtuvimos el siguiente resultado:

Gráfico 4.4.6 - Grafo completo

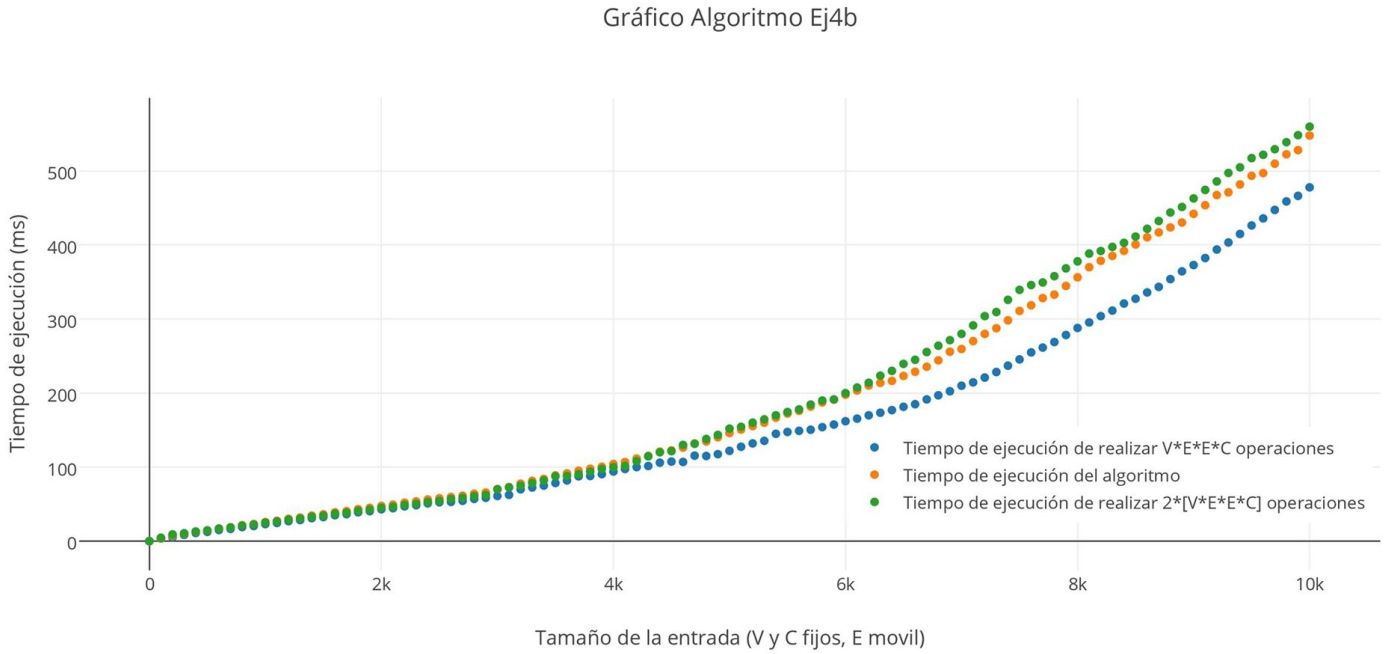


Gráfico 4.4.7 - Medición con grafo completo

En la figura 4.4.6 se puede observar el tipo de grafo que utilizamos. En la figura 4.4.7, se puede ver como el tiempo de ejecución de nuestro algoritmo es un poco mayor que un grafo circular, ya que tenemos más nodos interconectados y a pesar de esta situación nos mantenemos dentro de los tiempos estipulados corroborando la complejidad calculada.

Por último, mostraremos los gráficos comparativos tanto en lo que respecta a porcentajes de soluciones como de tiempos de ejecución:

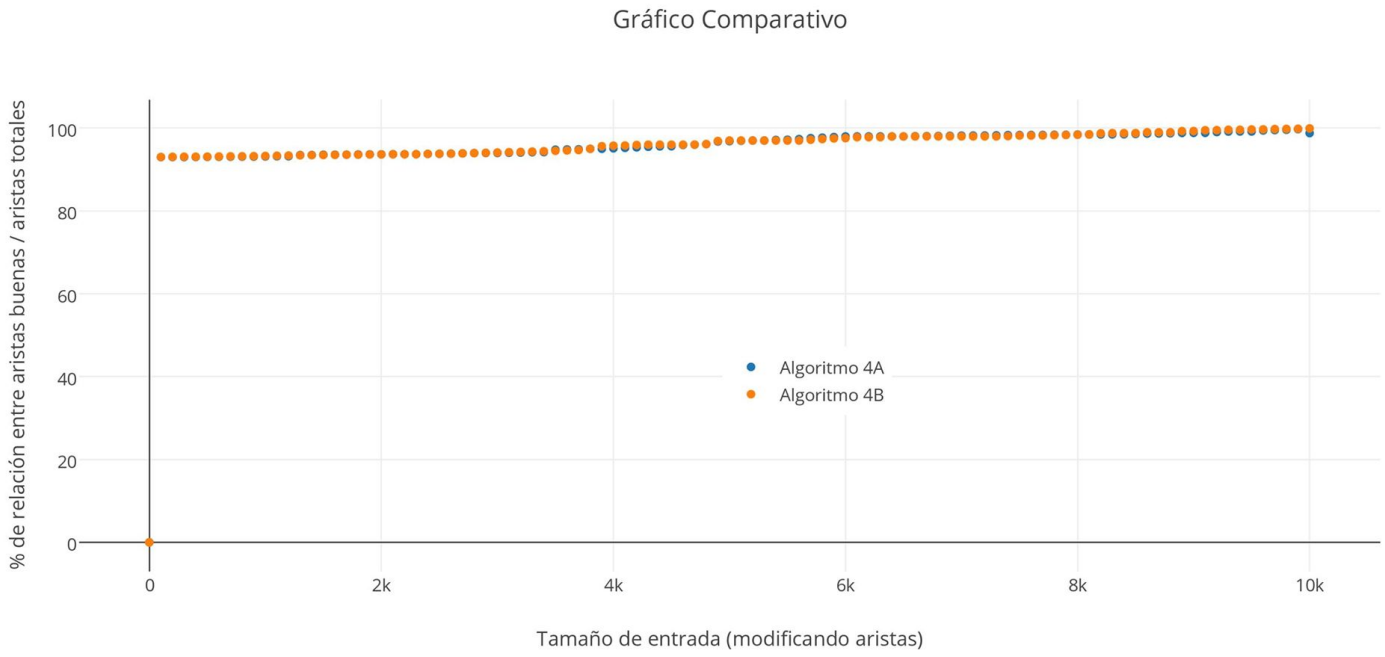


Gráfico 4.4.9 - Gráfico comparativo algoritmos con grafo completo

En la figura 4.4.9 se observan nuevamente ambas funciones, las cuales simbolizan el porcentaje de realizar la relación entre la cantidad de aristas denominadas buenas sobre la totalidad en un grafo completo. Se puede observar como el algoritmo del inciso b es sensiblemente superior y al incrementarse la cantidad de aristas como el porcentaje de relación aumenta gradualmente dando un alto grado de aristas "buenas" ya que existe menor cantidad de restricciones.

Y por último, el gráfico comparativo en función del tiempo de ejecución de los algoritmos para dicho grafo completo.

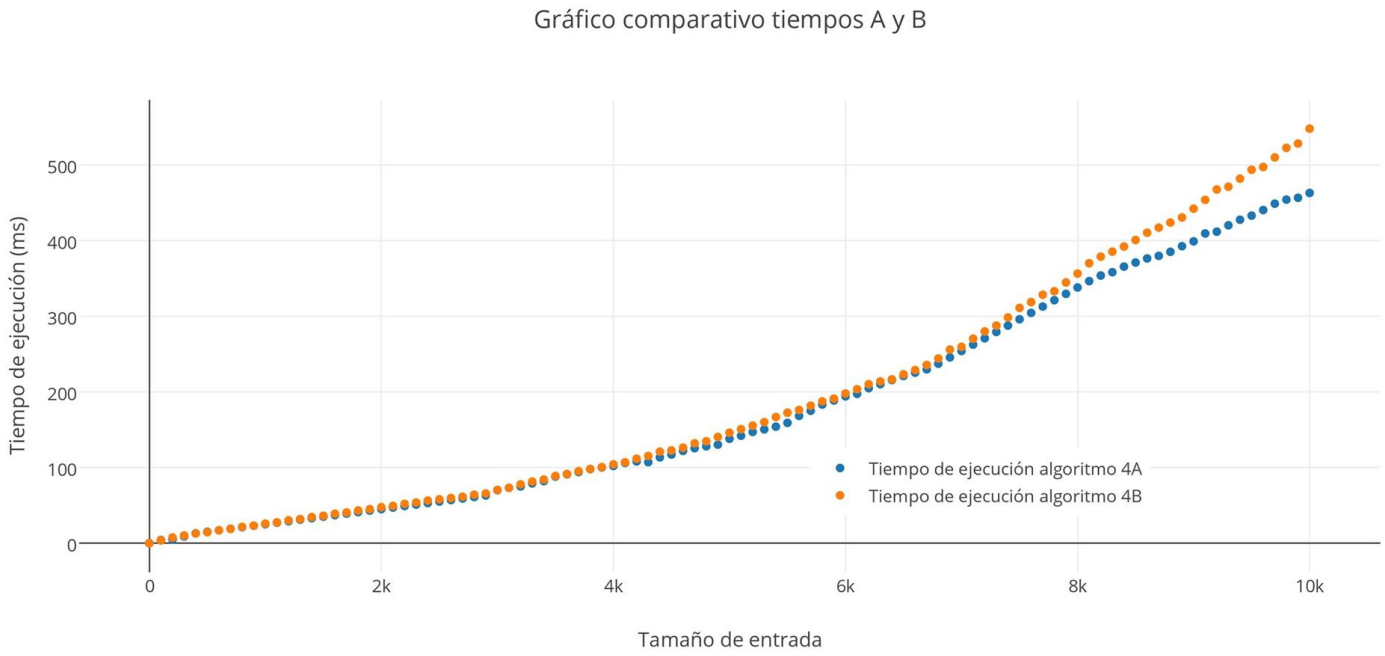


Gráfico 4.4.10 - Gráfico comparativo algoritmos con grafo completo

Se puede observar en las figuras 4.4.10 el tiempo de ejecución de cada algoritmo y como el B insume un poco más de tiempo que el A.

4.8 Conclusión comparación entre Vecindades

Podemos notar que ambas vecindades presentan calidades de solucion similares, mientras que la primera presenta mejor performance de tiempo. Por lo tanto, elegimos esta como vecindad definitiva.

5 Ejercicio 5

5.1 Enunciado del problema

Una vez elegidos los mejores valores de configuración para cada heurística implementada (si fue posible), realizar una experimentación sobre un conjunto nuevo de instancias para observar la performance de los métodos comparando nuevamente la calidad de las soluciones obtenidas y los tiempos de ejecución en función del tamaño de entrada. Para los casos que sea posible, comparar también los resultados del algoritmo exacto implementado. Presentar todos los resultados obtenidos mediante gráficos adecuados y discutir al respecto de los mismos.

5.2 Experimientos y conclusiones

5.2.1 Test

En esta sección se compararon las distintas heuristicas implementadas en el trabajo. Por un lado, la heuristica golosa y por otro la heuristica de busqueda local.
Dada la naturaleza del problema, no existe un único factor a analizar como en trabajos anteriores en los que solo interesaba el tiempo de corrida. Sino que también, al ser un problema de optimización, entra en juego la calidad de las soluciones.

5.3 Calidad de solución

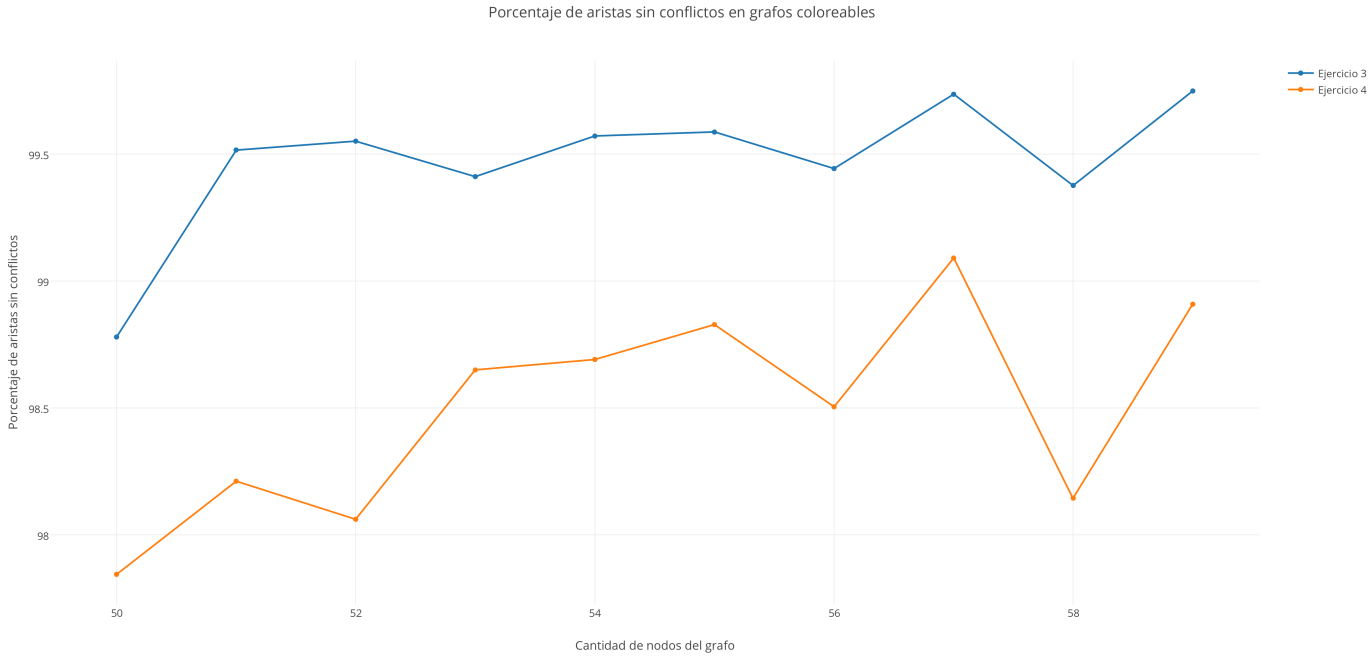
Primero analizaremos la calidad de las soluciones obtenidas por las heuristicas. Para realizar este analisis se implementó un generador de grafos aleatorios y en base a estos se corrieron los distintos algoritmos.
Estos experimentos se dividieron en dos etapas, uno que intenta medir la calidad de soluciones en grafos donde resulta posible su coloreo total y otro que toma en cuenta cualquier tipo de grafo.

Para medir la calidad de soluciones, consideramos el porcentaje de aristas sin conflictos (con ambos extremos pintados y de distinto color) sobre el total de aristas del grafo.

5.3.1 Grafos coloreables

Para realizar este experimento, se utilizó el generador de grafos aleatorios y se filtro a cada uno por los que el algoritmo del ejercicio 2 indicaba que eran coloreables.

Debido a la lentitud de este metodo, no se obtuvo un conjunto de grafos muy grande.

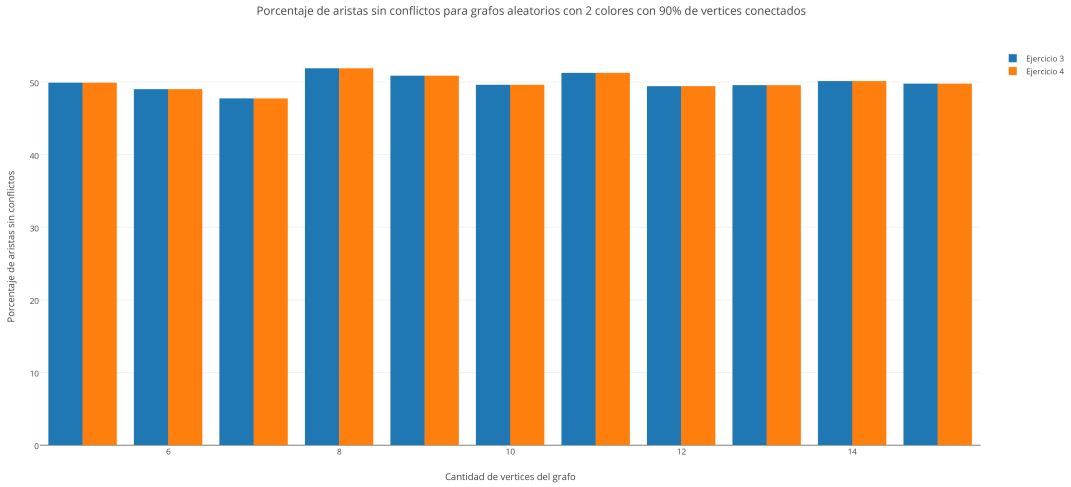


En el gráfico anterior se puede notar, que si bien la heuristica del ejercicio 3 presenta soluciones mas cercanas para este conjunto de grafos, ambas están muy cercanas a la solución optima.

5.3.2 Grafos no necesariamente coloreables

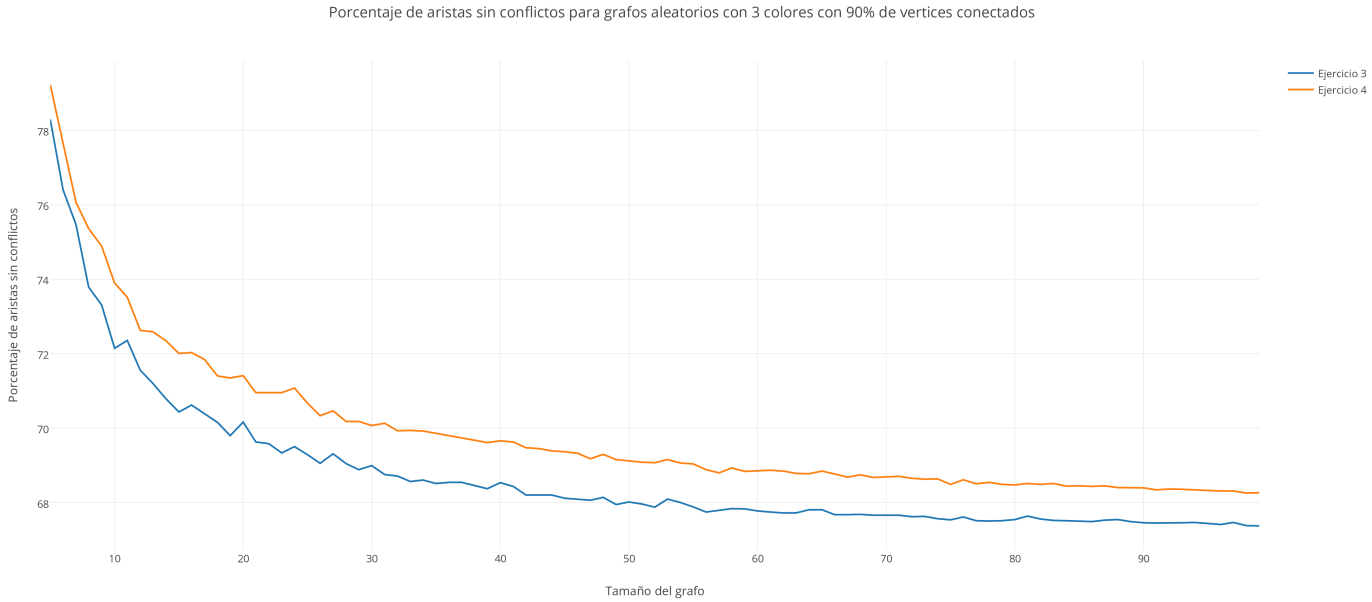
En estos experimentos, tomamos grafos aleatorios variando tamaño y cantidad de colores. Se comparó la calidad de las soluciones provistas por los dos algoritmos.

En el siguiente experimento se corrió a los dos algoritmos con grafos de distinto tamaño, pero con



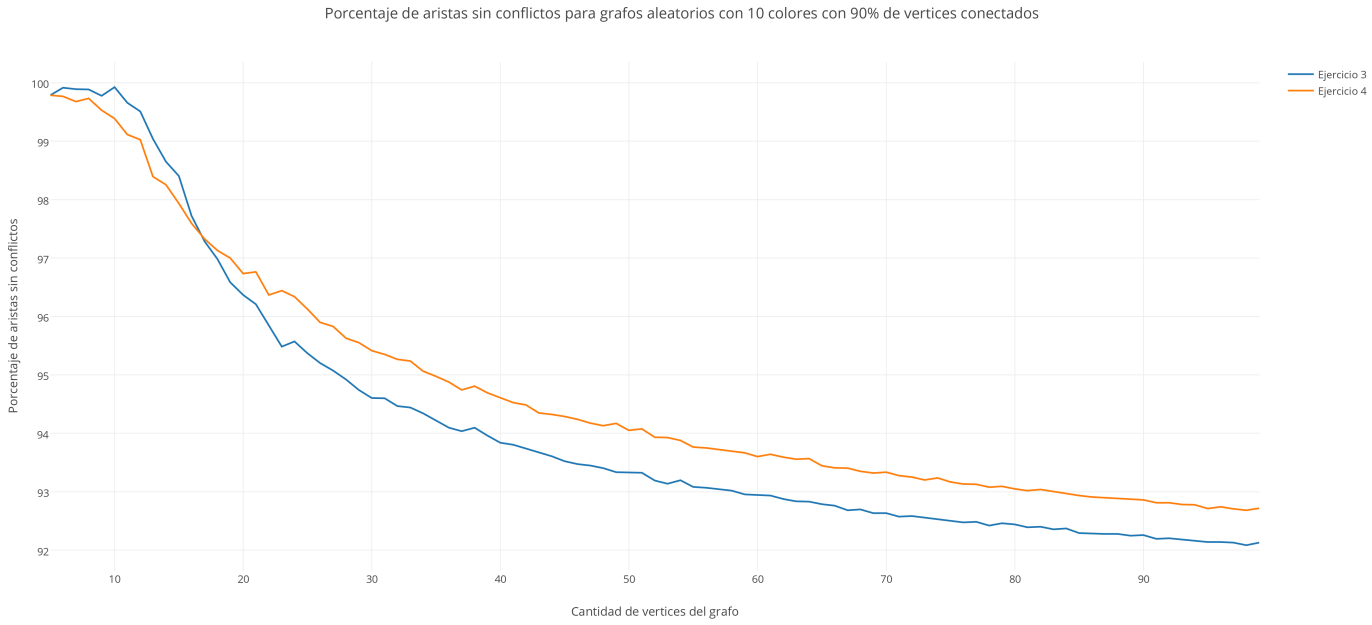
2 colores en total.

Curiosamente, las calidades de solución son las mismas para los dos algoritmos. Lo que nos muestra que los algoritmos se comportan de manera muy similar para este caso. (Queda pendiente la demostración de este interesante hecho)
Esto nos lleva a la pregunta si se mantendrá esta similitud para los demás casos. Por lo tanto, aumentamos la cantidad de colores a 3, y aumentamos el rango de tamaño en los experimentos.



Como podemos notar, las calidades ya empiezan a distinguirse entre los distintos algoritmos. Este caso da como ganador a la heuristica de busqueda local.

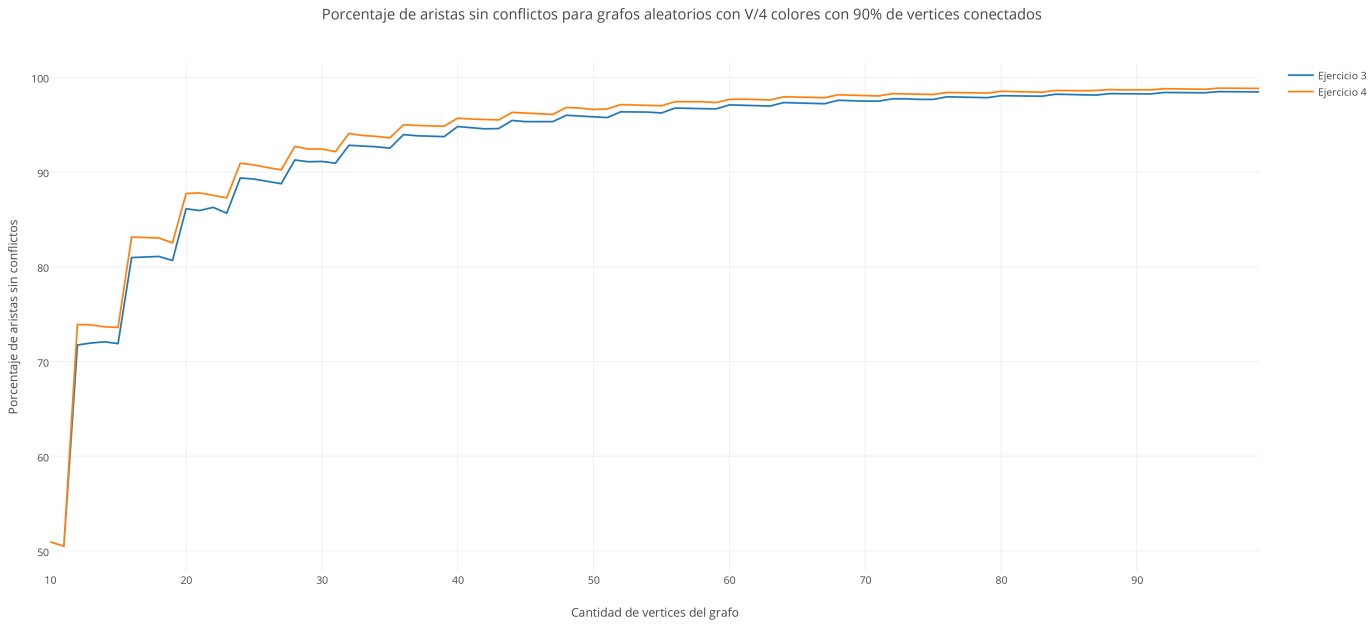
Realizamos un experimento más variando la cantidad de colores, pero siempre manteniendolos constante. La cantidad de colores elegidos fue de 10.



Si bien con tamaños de grafo mas pequeños la heuristica golosa supera en calidad a la de busqueda local en calidad, luego de un aumento en el tamaño del grafo esta diferencia se revierte en favor del algoritmo del ejercicio 4.

Un dato interesante que no nos queda claro su por qué es el hecho de que en todos los experimentos con cantidad de colores fija C pero con tamaño de grafo variable, la calidad se termina acercando al $100 - \frac{100}{C}$ porciento. Por ejemplo, con $C = 2$ estuvo alrededor del 50%, con $C = 3$ alrededor del 66%, con $C = 10$ alrededor del 90%.

Finalmente, decidimos probar con casos en los que la cantidad de colores aumenta linealmente con la cantidad de nodos del grafo. (Cada 4 vertices hay un nuevo color)



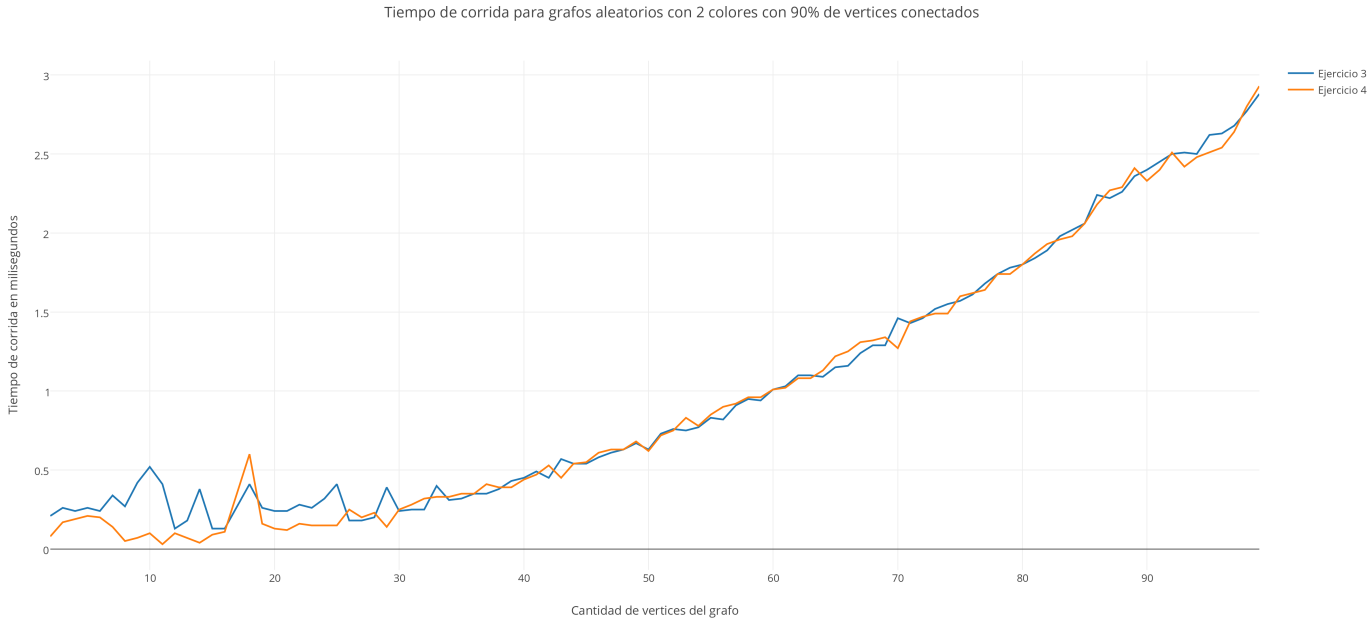
Podemos notar al gráfico con saltos en los multiplos de cuatro, y esto es debido de que en esos casos los ejemplos pasan a tener un color más entre las opciones, por lo que el problema pasa a ser "más fácil" en general.
Esta comparación también da por ganadora a la heuristica de busqueda local.

Resumiendo, pudimos notar en estos experimentos, que si bien en el analisis de calidad con grafos coloreables la heuristica del ejericicio 3 superó en calidad a la del 4, en los demás, esta última resultó ganadora. Creemos que los resultados del primer experimento se debieron a la poca cantidad de grafos analizados para analisis.

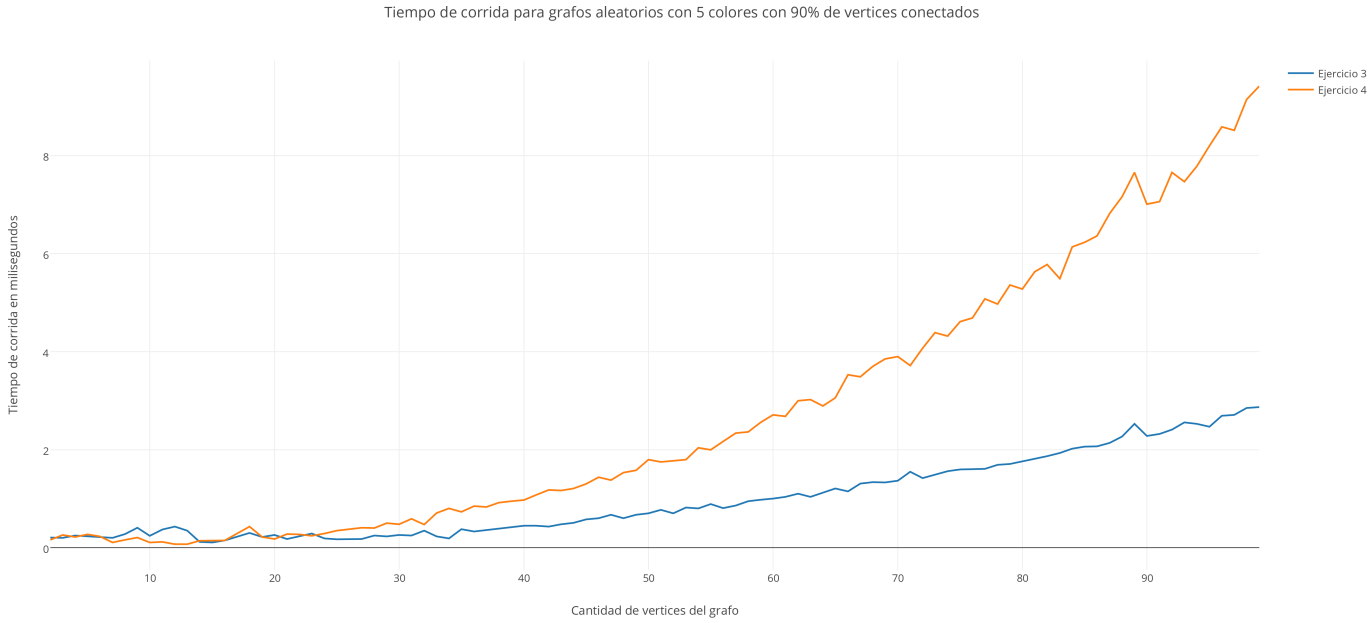
5.4 Performance de solución

En secciones anteriores las complejidades de cada algoritmo ya fueron estudiadas, y por lo tanto, se espera que los tiempos del algoritmo del ejercicio 3 sean notablemente menores que los del ejercicio 4.

En el siguiente gráfico se pueden comparar los tiempos de corrida para grafos de 2 colores en total.



Los tiempos resultan muy parecidos, ya que el hecho de que haya pocos colores en total hace que no sea notorio el factor C que tiene la complejidad del algoritmo del ejercicio 4. Por lo tanto, decidimos aumentar el la cantidad de colores, esta vez a 5.



Y ahora la diferencia de tiempos resulta notoria, como habiamos conjeturado al principio de esta sección.

5.5 Conclusión de comparaciónes

Conluímos con que no hay un claro ganador entre los algoritmos, ya que ambos presentan un intercambio entre performance y calidad.
Por lo tanto, que algoritmo es mejor dependerá de nuestras necesidades, es decir, de nuestro contexto de uso. Por ejemplo, si nos interesan soluciones con la mayor calidad posible podemos optar por la implementación del ejercicio 4. Por el contrario, si lo que nos interesa es la velocidad, podremos optar por la implementacion del ejercicio 3.

Para este tipo de comparaciones, siempre se debe responder preguntas como ¿Qué tanta calidad se necesita? ¿Cuánta performance se puede perder? ¿Los casos que me interesan analisar resultan muy beneficiosos para el primer algoritmo o no hay mucha diferencia?