DEPARTAMENTO DE COMPUTACION

Facultad de Ciencias Exactas y Naturales - UBA

Trabajo Práctico Número 3

16 de Mayo de 2016 — Algoritmos y Estructuras de Datos III

Grupo 8

Integrante	LU	Correo electrónico
Ciruelos Rodríguez, Gonzalo	063/14	gonzalo.ciruelos@gmail.com
Costa, Manuel José Joaquín	035/14	manucos94@gmail.com
Gatti, Mathias Nicolás	477/14	mathigatti@gmail.com
Maddonni, Axel Ezequiel	200/14	axel.maddonni@gmail.com

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

0. Introducción

En este trabajo desarrollaremos varias soluciones para el problema de el subgrafo común máximo entre dos grafos G_1 y G_2, con respecto a los vértices.

Más precisamente, dados $G_1 = (V_1, E_1)$ y $G_2 = (V_2, E_2)$ dos grafos simples, el problema de máximo subgrafo común consiste en encontrar un grafo $H = (V_H, E_H)$ isomorfo tanto a un subgrafo de G_1 como a un subgrafo de G_2 que maximice $|E_H|$.

Nuestros acercamientos al problema van a ser dos. Primero, vamos a desarrollar una solución exacta, es decir, una solución que encuentra el subgrafo común que maximiza la cantidad de aristas. Sin embargo, como veremos, esta forma de resolverlo no es razonable dado que se desconoce una solución en tiempo polinomial, por lo que encontrar la mejor solución no es viable para entradas grandes.

Luego, veremos varios algoritmos aproximados para el problema del subgrafo común máximo. Estos consisten en sacrificar exactitud a cambio de tiempo de ejecución. Su complejidad será polinomial, pero como dijimos, las soluciones que generen serán *aproximadas*, o sea, no exactas.

0.1. Experimentación

La experimentación en general sigue los pasos sugeridos por las consignas del trabajo. Los métodos de generación de casos estarán explicados al final de cada sección de experimentación, o en su defecto, en el apéndice.

Sobre la experimentación de tiempos, como las complejidades en general dependen de muchos parámetros, los resultados se vuelven difíciles de representar. Es por eso que seguiremos el mismo método de representación que utilizamos en los TPs anteriores. Supongamos que la complejidad del algoritmo es $O(f(n, m))$, entonces nuestro gráfico tendrá $f(n, m)$ en el eje x y $T(n, m) =$ "El tiempo que tarda el algoritmo para una entrada de tamaño (n,m)" en el eje y, de esta manera, nos interesará ver que el gráfico es el de una constante.

Por supuesto, también haremos experimentos en los que fijamos parámetros y movemos otros, para corroborar que las performances se comportan como deben. En caso de que las complejidades dependan de un parámetro, haremos un gráfico clásico, en caso contrario, haremos lo mismo que explicamos anteriormente.

1. Ejercicio 1

2. Ejercicio 2: Algoritmo exacto

2.1. Explicación detallada del algoritmo

El problema del sugrafo común máximo (con respecto a aristas) es un problema perteneciente a la clase NP, por lo que hasta el momento no se conocen algoritmos que lo resuelvan de manera exacta en tiempo polimonial.

Por esa razón, el algoritmo que lo resuelve de manera exacta debe ser de la clase de algoritmos que exploran todo el espacio de soluciones y se quedan con la mejor. De entre esos algoritmos, elegiremos un algoritmo de backtracking, dado que proponiendo buenas podas, se pueden mejorar los tiempos de ejecución del algoritmo, evitando mirar el espacio de soluciones en su enteridad.

Nuestras soluciones, es decir, nuestos isomorfismos, van a estar representados como un vector de pares. Cada par (v_{1i}, v_{2i}) es un par de vértices que cumplen que $v_{1i} \in V(G_1)$ y $v_{2i} \in V(G_2)$ y nuestro isomorfismo los mapea.

Por ejemplo, si nuestro isomorfismo es $f : V(G_1) \to V(G_2)$ tal que

$f(0) = f(1)$
$f(1) = f(2)$
$f(3) = f(5)$
$f(6) = f(0)$

Lo vamos a almacenar de la siguiente manera: $(0, 1), (1, 2), (3, 5), (6, 0)$. Nótese que no importa que el "isomorfismo" sea parcial, dado que en realidad estamos describiendo subrafos isomorfos.

Por lo tanto, nuestro algoritmo de backtracking se va a basar en probar todas las posibles combinaciones de listas de pares, y buscar cual representa el isomorfismo con la mayor cantidad de aristas.

Una primera poda que proponemos (bastante fácil de explicar y muy poderosa) es la que sigue: podemos notar que si ciegamente consideramos todas las combinaciones de pares, estaremos considerando todos los isomorfismos varias veces. Por ejemplo, el vector $(0, 1), (1, 2)$ y el vector $(1, 2), (0, 1)$ representan el mismo isomorfismo, pero nuestro algoritmo naif los analizará 2 veces.

Por esta razón, diseñamos la siguiente poda: solo considerar vectores cuyo vector de primera coordenadas esté ordenado ascendentemente. En el ejemplo anterior, cuando le llegue al turno a $(1, 2), (0, 1)$, no lo analizaremos ni a él, ni a ninguno de sus descendientes, dado que todos ellos serán analizados como descendientes de $(0, 1), (1, 2)$.

Otra poda que realizamos es que, si encontramos la solución máxima posible teóricamente (es decir, la solución cuya cantidad de aristas coincide con el mínimo de las aristas de G_1 y G_2), terminar con el algoritmo inmediatamente.

Una última poda que realizamos es considerar solamente aquellos vectores cuyo largo sea exactamente el del mínimo de los vértices de G_1 y G_2. Esto se debe a que, si un vector es más chico, vamos a poder considerar a un vector extendido, cuyo isomorfismo potencialmente tendrá más aristas.

El algoritmo utiliza una variable global llamada *solucin*, que tiene 2 campos, *aristas* e *isomorfismo*. En esta variable global irá guardando la mejor solución encontrada hasta el momendo. *solucin.aristas* debe inicializarse en 0.

Sin más que analizar, pasemos a ver el pseudocódigo del algoritmo. Vale la pena aclarar que asumimos como preconodición que G_1 tiene menos nodos que G_2 (en tal caso de que asi no fuere, el llamador debe ocuparse de dar vuelta los parámetros).

Algorithm 1 Pseudocódigo del procedimiento Backtracking

```
1: procedure BT(Grafo g1, Grafo g2, vector<int> vertices1, vector<int> vertices2, Isomorfismo iso)
2:     if solucion.aristas == MIN(g1.m, g2.m) then                                   ▷ O(1)
3:         return
4:     if !ordenado_asc(iso) then                                                    ▷ O(n_1)
5:         return
6:     if iso.size == g1.n then                                                      ▷ O(1)
7:         aristas ← contar_aristas_isomorfismo(g1, g2, iso)                         ▷ O(n_1^2)
8:         if aristas > solucion.aristas then                                        ▷ O(1)
9:             solucion.isomorfismo ← iso                                            ▷ O(n_1)
10:            solucion.aristas ← aristas                                            ▷ O(1)
11:        return
12:    for u ∈ vertices1 do                                                          ▷ v_1 veces O(1)
13:        for v ∈ vertices2 do                                                      ▷ v_1v_2 veces O(1)
14:            nuevo_iso = iso                                                       ▷ v_1v_2 veces O(n_1)
15:            nuevo_iso.push_back(make_pair(u, v))                                  ▷ v_1v_2 veces O(1)
16:            bt(g1, g2, copiar_sin(vertices1, u), copiar_sin(vertices2, v), nuevo_iso)   ▷ v_1v_2 veces T(n_1, n_2, v_1 − 1, v_2 − 1)
```

Algorithm 2 Pseudocódigo del procedimiento contar aristas isomorfismo

```
1: procedure CONTAR_ARISTAS_ISOMORFISMO(Grafo g1, Grafo g2, Isomorfismo iso) → Int
2:     aristas ← 0                                                                   ▷ O(1)
3:     for p ∈ iso do                                                                ▷ n_1 veces O(1)
4:         vg1 = p.first                                                             ▷ n_1 veces O(1)
5:         vg2 = p.second                                                            ▷ n_1 veces O(1)
6:         for q ∈ iso do                                                            ▷ n_1 veces O(1)
7:             ug1 = q.first                                                         ▷ n_1^2 veces O(1)
8:             ug2 = q.second                                                        ▷ n_1^2 veces O(1)
9:             if g1.adj_matrix[vg1][ug1] ∧ g2.adj_matrix[vg2][ug2] then             ▷ n_1^2 veces O(1)
10:                aristas + +                                                       ▷ n_1^2 veces O(1)
       return aristas
```

Donde $n_i = |V(G_i)|$ y $m_i = |E(G_i)|$, para $i = 1, 2$. Nótese que el largo del vector isomorfismo está acotado superiormente por n_1, pues $n_1 < n_2$ y el isomorfismo mapea a lo sumo a todos los vértices de n_1 y no puede mapear más cosas.

Además, v_i para $i = 1, 2$ es el tamaño del vector *verticesi*.

El algoritmo debe ser llamado de la siguiente manera:

$$bt(grafo1, \{1, ..., |V(grafo1)|\}, grafo2, \{1, ..., |V(grafo2)|\}, \{\})$$

.

Dado que inicialmente tódos los vértices están sin ser utilizados, y el isomorfismo es el isomorfismo vacío.

2.1.1. Complejidad del algoritmo

Calculemos la complejidad del algoritmo. Primero, como vimos, la complejidad del algoritmo contar_aristas_isomorfismo es $O(n_1^2)$ y es bastante fácil de calcular.

Ahora, calculemos la complejidad del algoritmo de backtracking. Sea $T(n_1, n_2, v_1, v_2)$ el tiempo que el algoritmo tarda para una entrada de ese tamaño. El tamaño del vector *iso* puede acotarse por n_1, como vimos antes.

Como se ve en en análisis de complejidad, $T(n_1, n_2, v_1, v_2) = n_1 + n_1^2 + v_1v_2n_1 + v_1v_2 + v_1v_2T(n_1, n_2, v_1 - 1, v_2 - 1)$.

Además, notemos que siempre pasa que $v_i < n_i$, luego, la complejidad puede acotarse por $T(n_1, n_2, v_1, v_2) < n_1 + n_1^2 + v_1v_2n_1 + v_1v_2 + v_1v_2T(n_1, n_2, v_1 - 1, v_2 - 1)$. Nos tomamos la libertad de escribir $T(v_1, v_2)$, dado que son los únicos parámetros variables (n_1 y n_2 están fijos).

Como $n_i > v_i$, $T(v_1, v_2) < n_1 + n_1^2 + n_1n_2n_1 + n_1n_2 + v_1v_2T(v_1 - 1, v_2 - 1)$.

Luego, $T(v_1, v_2) < 4n_1^2n_2 + v_1v_2T(v_1 - 1, v_2 - 1)$.

Además, $T(n_1, n_2, 0, v_2) = n_1 + n_1^2$ (recordemos que siempre va a pasar que $n_1 < n_2$, por lo tanto $v_1 < v_2$). Por lo que la profundidad de la fórmula recursiva depende solo de v_1. Luego,

$$\begin{aligned}
T(v_1, v_2) &< 4n_1^2n_2 + v_1v_2T(v_1 - 1, v_2 - 1) \\
&< 4n_1^2n_2 + v_1v_2(4n_1^2n_2 + (v_1 - 1)(v_2 - 1)T(v_1 - 2, v_2 - 2)) \\
&= 4n_1^2n_2 + v_1v_2 4n_1^2n_2 + v_1v_2(v_1 - 1)(v_2 - 1)T(v_1 - 2, v_2 - 2) \\
&< 4n_1^2n_2 + 4n_1^3n_2^2 + v_1v_2(v_1 - 1)(v_2 - 1)T(v_1 - 2, v_2 - 2) \\
&< ... \\
&< 4n_1^2n_2 + ... + 4n_1^{v_1}n_2^{v_1-1} + v_1(v_1 - 1)...(v_1 - v_1 + 1)v_2(v_2 - 1)...(v_2 - v_1 + 1)T(0, v_2 - v_1) \\
&< 4n_1^2n_2 + 4n_1^3n_2^2 + ... + 4n_1^{v_1}n_2^{v_1-1} + 4n_1^{v_1+1}n_2^{v_1} \\
&< 4n_1^2n_2 + 4n_1^3n_2^2 + ... + 4n_1^{v_1+1}n_2^{v_1} \\
&< 4^{v_1+1}n_1^{v_1+1}n_2^{v_1+1} \\
&= (4n_1n_2)^{v_1+1}
\end{aligned}$$

Luego, $T(n1, n2) \in O((4n_1n_2)^{n_1+1})$.

Nótese que esta es una cota bastante poco ajustada, pero es suficientemente exacta para el análisis que queremos hacer (una cota más ajustada, como se ve en los cálculos anteriores, involucra fórmulas con factoriales y combinatorios).

2.2. Performance del algoritmo

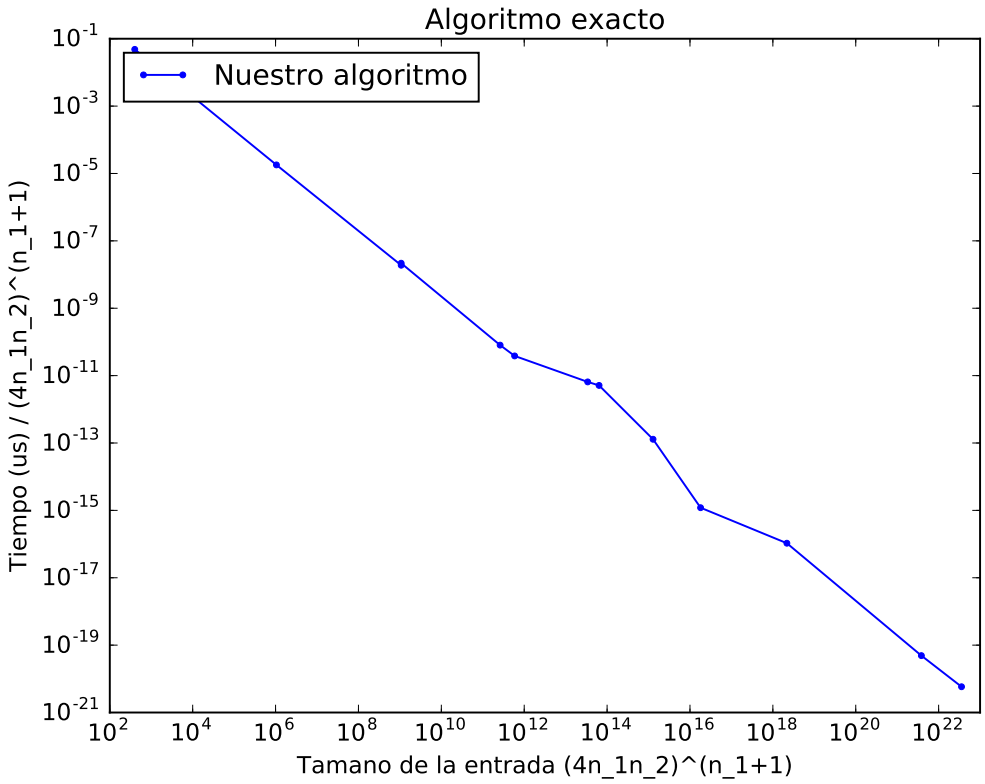


Figura 1

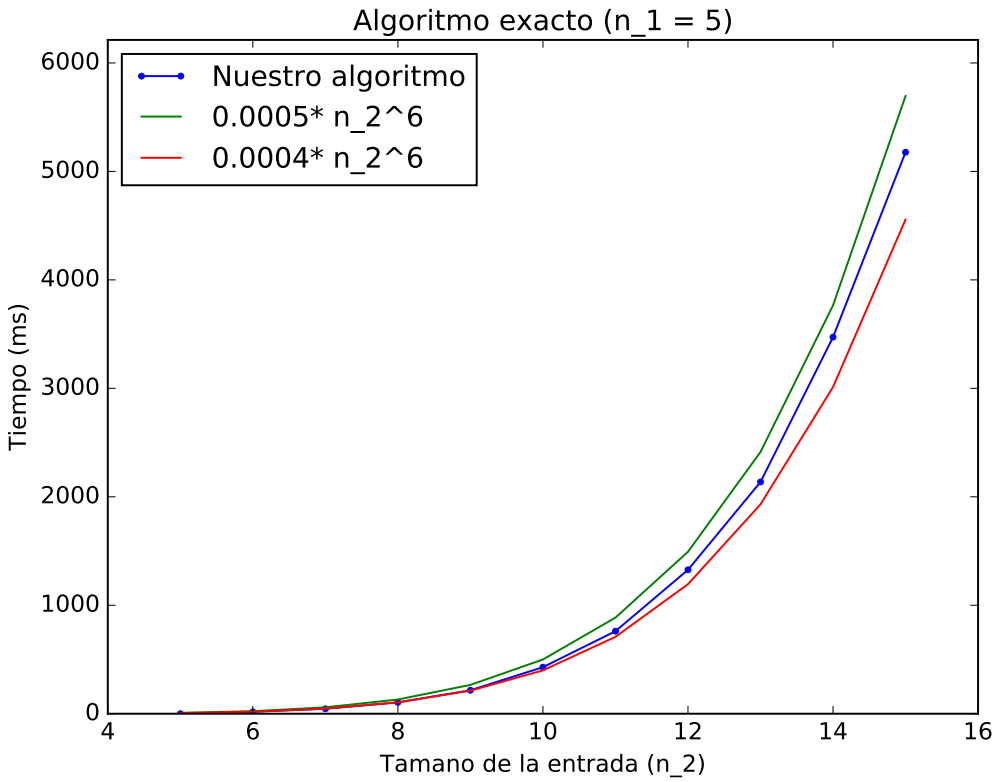

Algoritmo exacto (n_1 = 5)
Nuestro algoritmo
0.0005* n_2^6
0.0004* n_2^6
Tiempo (ms)
Tamano de la entrada (n_2)
0
1000
2000
3000
4000
5000
6000
4
6
8
10
12
14
16

Figura 2

3. Ejercicio 3

4. Ejercicio 4: Algoritmo Goloso

4.1. Explicación detallada del algoritmo

Como ya mencionamos en el ejercicio 2 este problema no parece poder ser resuelto en tiempo polinomial, para poder acercarnos a una solución en tiempos razonables sacrificaremos la seguridad de conseguir siempre la opción óptima a cambio de mejorar la complejidad temporal. Esto se hará a partir de la implementación de una heuristica golosa, un programa que a partir de ciertas suposiciones, no necesariamente validas siempre pero muchas veces útiles, nos permitirá tomar desiciones rapidamente. Será golosa porque tomará desiciones buscando mejorar su estado actual sin pensar en la solución optima final, o sea, sin pensar a largo plazo.

Hicimos dos versiones distintas. La primer versión consiste en mapear los nodos de mayor grado entre si hasta agotar todos los del grafo mas chico. Esto puede funcionar en algunos grafos pero claramente no siempre será la mejor opción.

Algorithm 3 Pseudocódigo de la primer heurística golosa

```
1:  procedure GOLOSO1(Grafo g1, Grafo g2, set<int> vertices1, set<int> vertices2) → MCS
2:      ordenar_por_grado(vertices1, g1)                                   ▷ O(n1^2)
3:      ordenar_por_grado(vertices2, g2)                                   ▷ O(n2^2)
4:      MCS solucion                                                       ▷ O(1)
5:      for int i = 0, i < vertices1.tamanio(), i++ do
6:          solucion.isomorfismo.insertar_atras(< vertices1[i], vertices2[i] >)   ▷ n1 veces O(1)
7:      int aristas
8:      aristas = contar_aristas_isomorfismo(g1, g2, u, v, solucion.isomorfismo)   ▷ O(n1^2)
9:      solucion.aristas = aristas                                         ▷ O(1)
10:     return solucion
```

Como se puede observar en el pseudo-código, el programa inicia ordenando por grado los vértices con $ordenar_por_grado()$ ayudandose con las matrices de adyacencia de g_1 y g_2. Luego crea el isomorfismo escogiendo mapear el nodo de mayor grado de g_1 con el de g_2, luego el segundo y asi sucesivamente hasta que se agoten. Una vez hecho esto se calculan las aristas del isomorfismo con $contar_aristas_isomorfismo()$, verificando que aristas se comparten en ambos grafos. (Para mas detalles sobre las funciones nombradas recurrir al apéndice.)

La segunda heurística tiene una mayor complejidad temporal pero da soluciones de calidad superior, esto será expuesto en sección de experimentación. El algoritmo inicia haciendo un mapeo entre el nodo de mayor grado de G_1 y G_2 luego expande este isomorfismo buscando en cada iteración agregar el mapeo de nodos que maximice la cantidad de aristas (del isomorfismo). Si hay empates se queda con la primer opción.

El pseudocódigo es el siguiente

Algorithm 4 Pseudocódigo de la heurística golosa

```
1: procedure GOLOSO(Grafo g1, Grafo g2, vector<int> vertices1, vector<int> vertices2) → MCS
2:     MCS solucion                                                          ▷ O(1)
3:     solucion.aristas = 0
4:     int vertice1 = mayor_adj(vertices1, g1)                               ▷ O(n_1)
5:     int vertice2 = mayor_adj(vertices2, g2)                               ▷ O(n_2)
6:     solucion.isomorfismo.insertar_atras(< vertice1, vertice2 >)           ▷ O(1)
7:     vertices1.borrar(vertice1)                                            ▷ O(log(n_1))
8:     vertices2.borrar(vertice2)                                            ▷ O(log(n_2))
9:     while vertices1.tamanio() ≠ 0 do
10:        par<int,int> par_mayor_deg =< vertices1.primero(), vertices2.primero() >   ▷ n_1 veces O(1)
11:        for u ∈ vertices1 do
12:            for v ∈ vertices2 do
13:                int aristas                                               ▷ n_1^2 n_2 veces O(1)
14:                aristas = contar_aristas_isomorfismo(g1, g2, u, v, solucion.isomorfismo)   ▷ n_1^2 n_2 veces O(n_1^2)
15:                if aristas > solucion.aristas then                        ▷ n_1^2 n_2 veces O(1)
16:                    solucion.aristas = aristas                            ▷ n_1^2 n_2 veces O(1)
17:                    par_mayor_deg =< u, v >                               ▷ n_1^2 n_2 veces O(1)
18:        solucion.isomorfismo.insertar_atras(par_mayor_deg)                ▷ n_1 veces O(1)
19:        vertices1.borrar(par_mayor_deg.primero)                           ▷ n_1 veces O(log(n_1))
20:        vertices1.borrar(par_mayor_deg.segundo)                           ▷ n_1 veces O(log(n_2))
21:    return solucion
```

4.2. Complejidad temporal de peor caso

El primer algoritmo tiene complejidad $O(n_2^2+n_1^2)$ lo cual es acotable superiormente por $O(n_2^2)$ ya que n_2 siempre es mayor que n_1, es la precondición del programa.

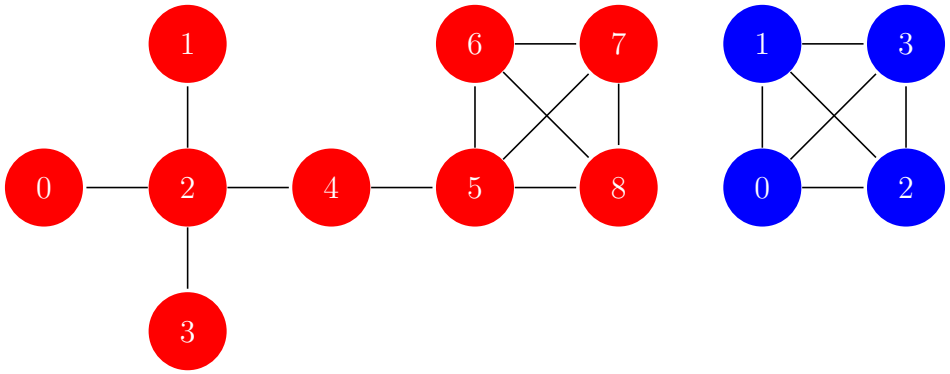
En la segunda heuristica la complejidad es precisamente $O(n_1^4 n_2)$. Esto resulta de los 3 ciclos anidados que hay mas una operación de costo cuadratico respecto de n_1. Si sumamos todo lo que se detalló en el pseudocódigo se puede ver que queda $O(n_1 + n_2 + \log(n_1) + \log(n_2) + n_1^2.n_2 + n_1^2.n_2.n_1^2 + n_1.\log(n_1) + n_1.\log(n_2)) = O(n_1^4 n_2)$

4.3. Instancias no óptimas

La primer heurística tiene varios casos donde fallará rotundamente. Un ejemplo puede ser el caso en que G_1 es el grafo completo G_n y G_2 la unión de n grafos estrella S_n. En ese caso, por el funcionamiento de nuestro algoritmo, se mapeará a cada nodo del grafo completo con el centro de cada una de las n estrellas, ya que los centros de las estrellas son los nodos de mayor grado, esto formará un isomorfismo sin aristas, ya que en G_2 todos los nodos escogidos son disconexos entre si. Esta solución será mucho peor que la óptima que

se dá al tomar el centro de una estrella y $n-1$ nodos conexos a este y mapearlos a los del grafo completo, la solución óptima tiene n aristas.

Un ejemplo donde la segunda heuristica golosa puede ser tan mala como uno quiera es cuando en su entrada recibe al grafo completo G_n y a un grafo que resulta de la unión de G_n con S_n, el cual es ilustrado mas abajo.



Si recordamos como funciona nuestra segunda heurística golosa, se puede ver que esta arrancará mapeando el centro de la estrella (el nodo de mayor grado) con algun nodo del grafo completo. Luego seguirá con un nodo que maximice la cantidad de aristas del isomorfismo, el primero que encontrará que cumpla esto será el nodo 0, seguirá de esta manera mapeando los extremos de la estrella con los nodos del grafo completo. En vez de elegir la opción óptima, mapear el grafo completo con el grafo completo contenido en G_1.

Para estos casos la salida de la heuristica, $H(n)$, siempre va a devolver un isomorfismo con n aristas, ya que mapeara los nodos de la estrella con los del grafo completo. En vez de esto la solución óptima, $Opt(n)$, sería G_n, o sea tendría $\frac{n.(n-1)}{2}$ aristas, por lo cual este no es un algoritmo aproximado, para n suficientemente grande no existe ningun número real positivo, ρ, que acote inferiormente a $\frac{H(n)}{Opt(n)} = \frac{n}{n.(n-1)/2} = \frac{2}{n-1}$.

4.4. Performance del algoritmo

4.5. Experimentación

En esta parte del informe nos dedicaremos a corroborar empíricamente ciertas hipótesis sobre nuestra herística golosa.

Primero intentamos ver que la complejidad del peor caso, $(n_1^4.n_2)$ sea una cota superior correcta. Esto es corroborado en el siguiente gráfico.

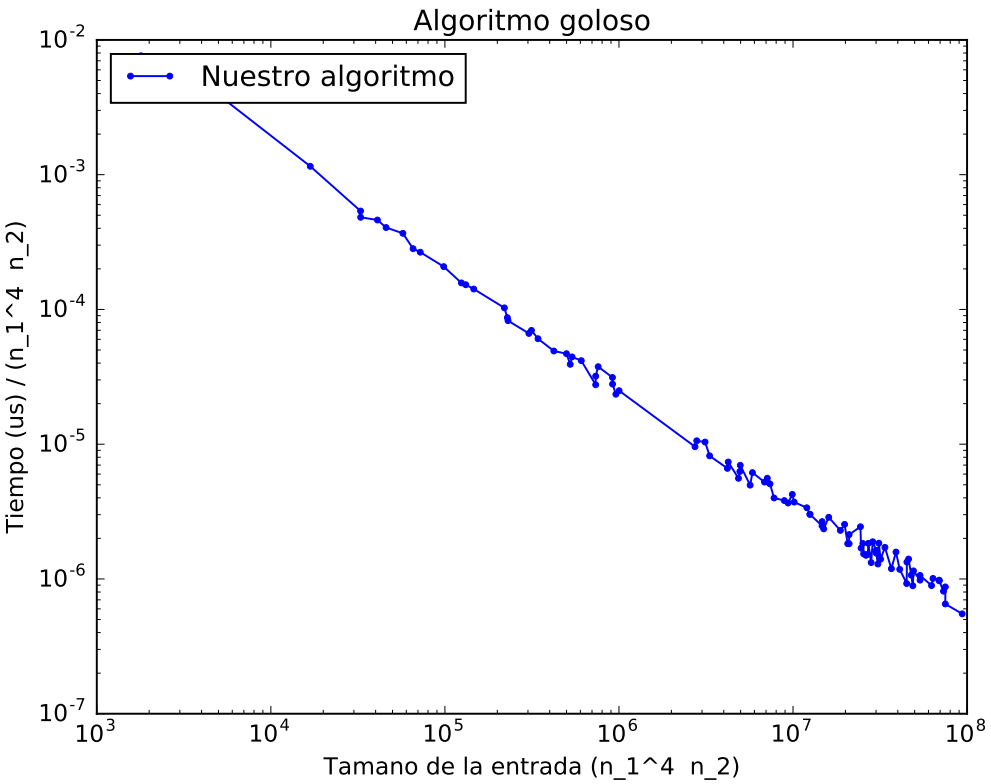


Figura 3

Se puede observar como, para instancias cada vez mas grandes, el tiempo en relación a la cota esperada decrece tendiendo a cero, de esto se puede deducir que nuestra cota no es la mas ajustada posible, o sea no serviría como cota inferior.

Como se puede observar, la complejidad que calculamos previamente depende de dos variables, n_1 y n_2, los siguientes gráficos analizan por separado que pasa cuando se varía cada uno de estos valores y se intenta corroborar que la complejidad es correcta.

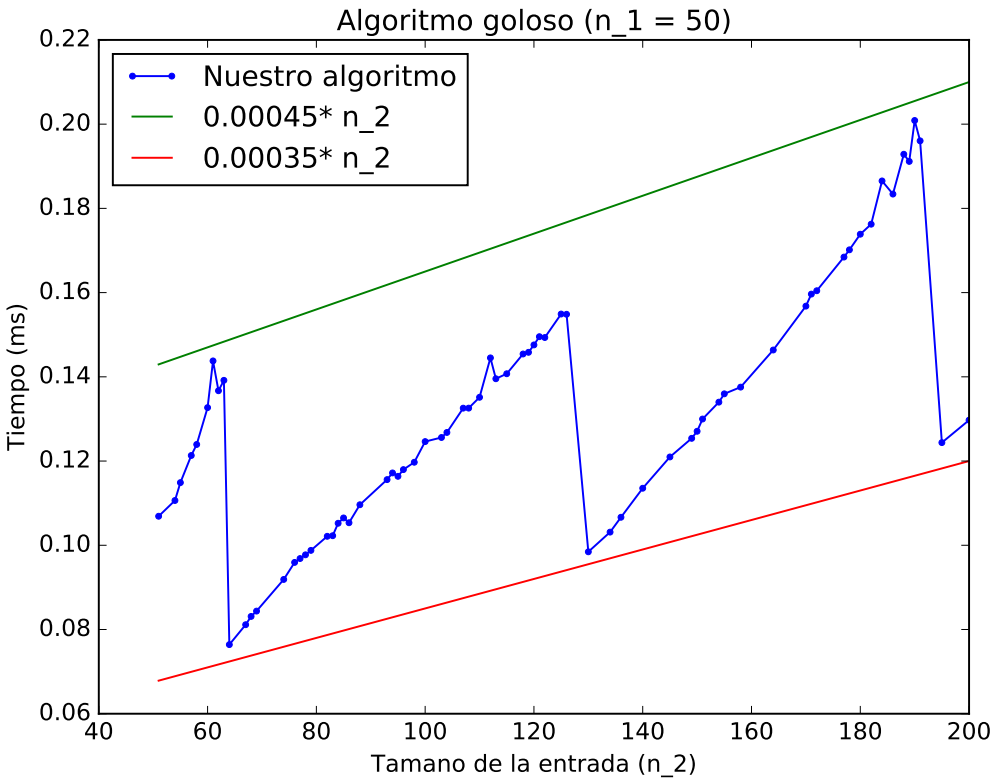


Figura 4

Este gráfico tiene dos particularidades que valen la pena comentar. Primero que púdimos ajustar el problema inferior y superiormente para esta variable. Segundo que hay picos, esto se debe a la manera en que maneja c++ a los vectores, estos se copian y aumentan de tamaño al pasar un tamaño multiplo de 64 elementos.

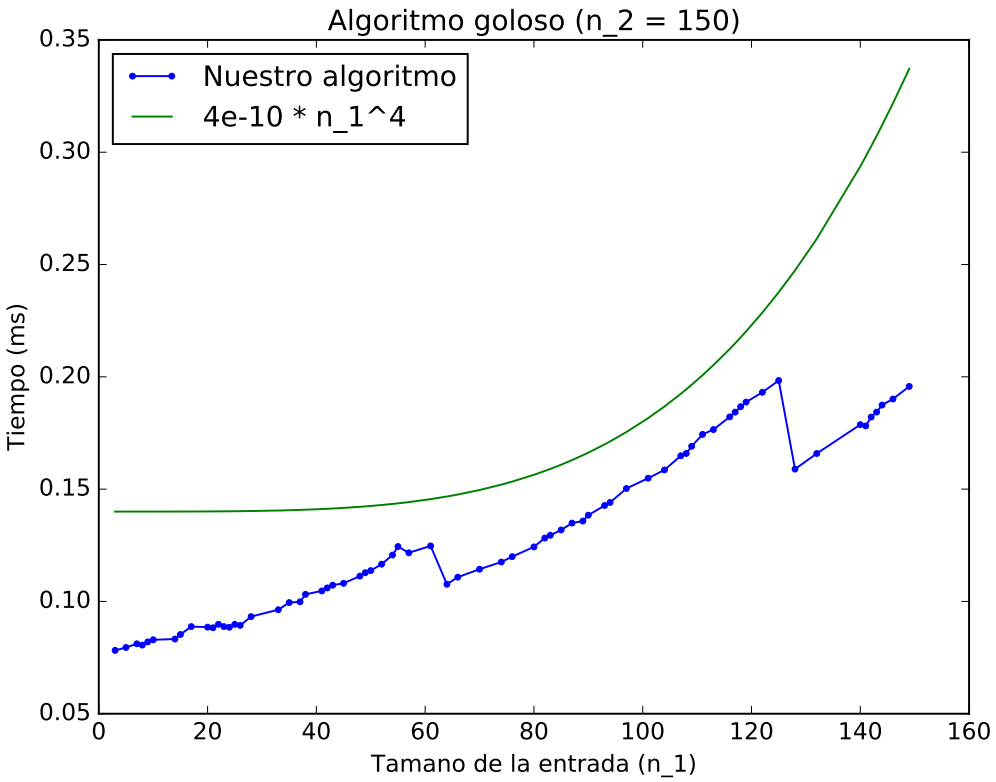


Figura 5

En este gráfico se ve como acotamos superiormente las instancias generadas con la complejidad que habiamos calculado y variando solo n_1.

La siguiente fígura expone la cantidad de aristas que tenian los isomorfismos calculados por nuestras dos heurísticas golosas, la mala (la primera) y la segunda, la que escogimos como nuestro algoritmo predilecto.

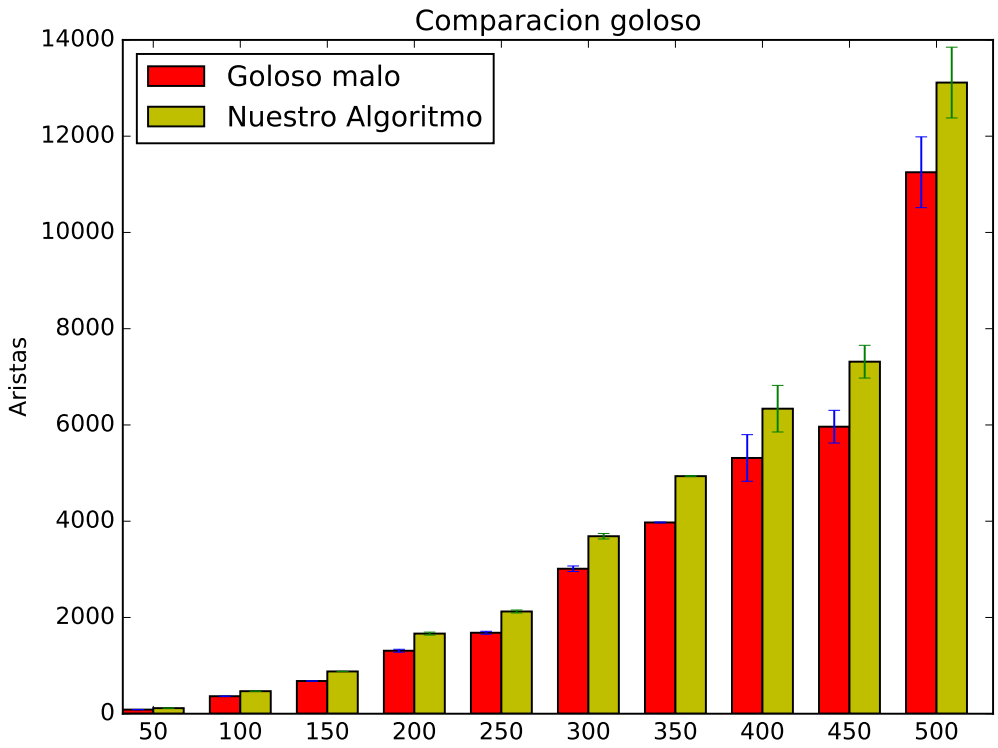


Figura 6

Se puede ver como nuestro algoritmo siempre es en promedio superior. Aunque la ventaja no es tampoco inmensa, lo cual deja a criterio de la aplicación deseada si es preferible usar una u otra, ya que si recordamos lo dicho antes, la complejidad del Goloso Malo es bastante mejor, $O(n_2^2)$.

4.6. Método de experimentación

5. Ejercicio 5

Algorithm 5 Pseudocódigo de INTERCAMBIAR

```
1: procedure INT(Grafo G_1, set<int> vertices_1, Grafo G_2, set<int> vertices_2)→ MCS
2:     MCS source ← goloso(G_1, vertices_1, G_2, vertices_2)                    ▷ O()
3:     bool mejore ← true                                                       ▷ O(1)
4:     while mejore do                                                          ▷ O(min{m_1, m_2})
5:         mejore ← false                                                       ▷ O(1)
6:         for i ← 0 ... |source.isomorfismo| do                                ▷ n_1 veces
7:             for j ← 0 ... |source.isomorfismo|, i ≠ j do                     ▷ n_1 veces
8:                 swap(source.isomorfismo[i].first, source.isomorfismo[j].first)   ▷ O(1)
9:                 int aristas ← contar_aristas_isomorfismo(G_1, G_2, source.isomorfismo)   ▷
   O(n_1^2)
10:                if aristas > source.aristas then                             ▷ O(1)
11:                    source.aristas ← aristas                                 ▷ O(1)
12:                    mejore ← true                                            ▷ O(1)
```

Algorithm 6 Pseudocódigo de REMPLAZAR

```
1: procedure REMP(Grafo G_1, set<int> vertices_1, Grafo G_2, set<int> vertices_2)→ MCS
2:     MCS source ← goloso(G_1, vertices_1, G_2, vertices_2)
3:     bool mejore ← true                                                       ▷ O(1)
4:     vector<int> vertices ← set_to_vector(vertices_2)                         ▷ O(n_2 − n_1)
5:     while mejore do                                                          ▷ O(min{m_1, m_2})
6:         mejore ← false                                                       ▷ O(1)
7:         for i ← 0 ... |vertices| do                                          ▷ n_2 − n_1 veces
8:             for j ← 0 ... |source.isomorfismo| do                            ▷ n_1 veces
9:                 swap(vertices[i], source.isomorfismo[j].second)              ▷ O(1)
10:                int aristas ← contar_aristas_isomorfismo(G_1, G_2, source.isomorfismo)   ▷
   O(n_1^2)
11:                if aristas > source.aristas then                             ▷ O(1)
12:                    source.aristas ← aristas                                 ▷ O(1)
13:                    mejore ← true                                            ▷ O(1)
14:                else
15:                    swap(vertices[i], source.isomorfismo[j].second)          ▷ O(1)
```

Algorithm 7 Pseudocódigo de 3-ROTACION

```
1: procedure 3-ROT(Grafo G_1, set<int> vertices_1, Grafo G_2, set<int> vertices_2)→ MCS
2:     MCS source ← goloso(G_1, vertices_1, G_2, vertices_2)
3:     bool mejore ← true                                                    ▷ O(1)
4:     while mejore do                                                       ▷ O(min{m_1, m_2})
5:         mejore ← false                                                    ▷ O(1)
6:         for i ← 0 ... |source.isomorfismo| do                             ▷ O(n_1) veces
7:             for j ← 0 ... |source.isomorfismo|, i ≠ j do                   ▷ n_1 veces
8:                 for k ← 0 ... |source.isomorfismo| do                      ▷ n_1 veces
9:                     swap(source.isomorfismo[i].first, source.isomorfismo[k].first)   ▷ O(1)
10:                    swap(source.isomorfismo[k].first, source.isomorfismo[j].first)   ▷ O(1)
11:                    int aristas ← contar_aristas_isomorfismo(G_1, G_2, source.isomorfismo)
   ▷ O(n_1^2)
12:                    if aristas > source.aristas then                      ▷ O(1)
13:                        source.aristas ← aristas                          ▷ O(1)
14:                        source.isomorfismo ← source.isomorfismo           ▷ O(n_1)
15:                        mejore ← true                                     ▷ O(1)
```

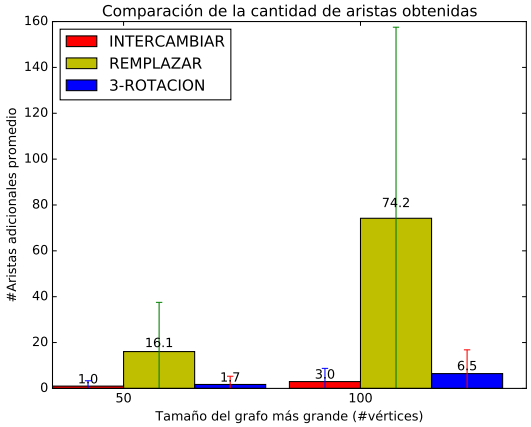


Figura 7:

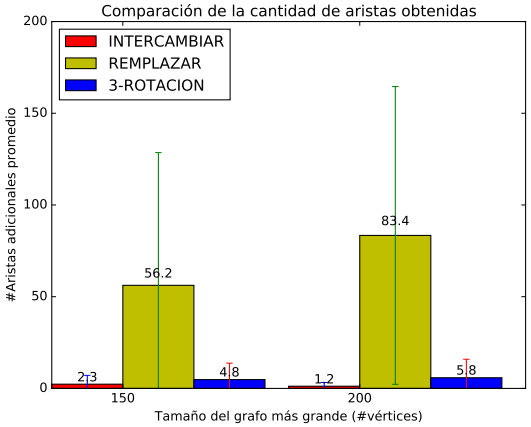


Figura 8:

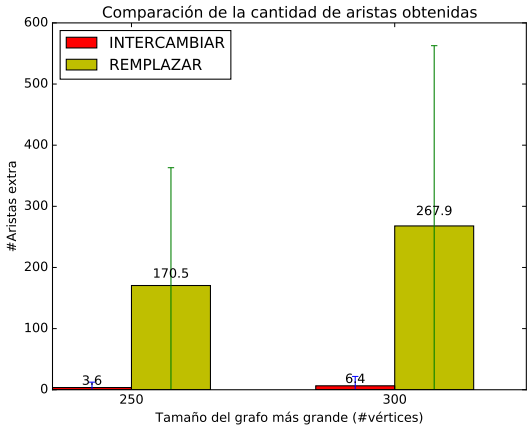


Figura 9:

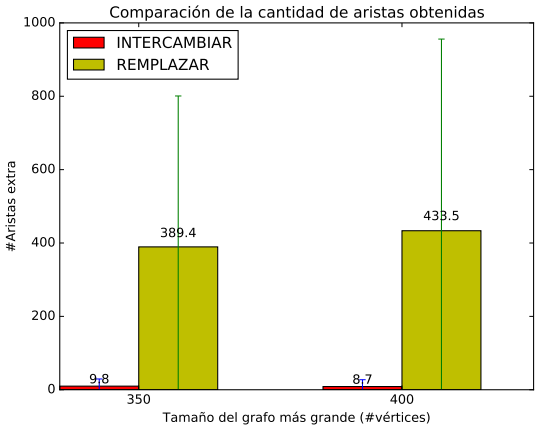


Figura 10:

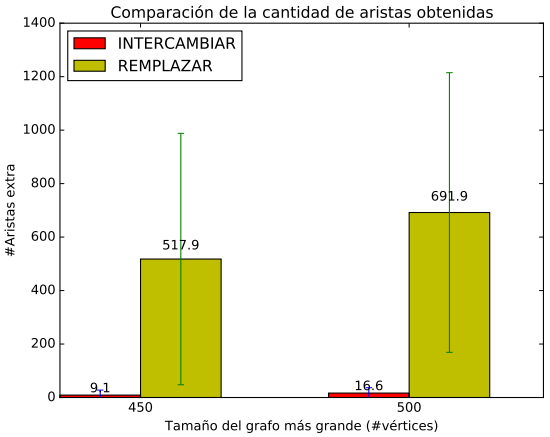


Figura 11:

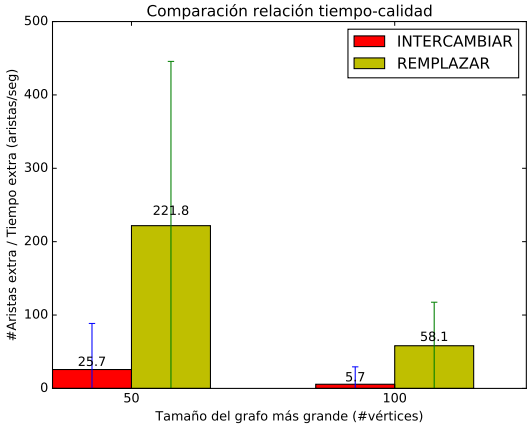


Figura 12:

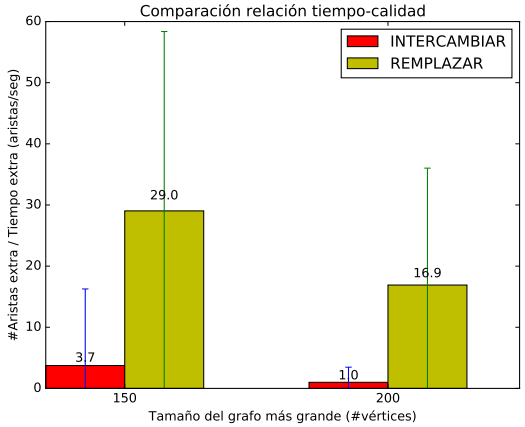


Figura 13:

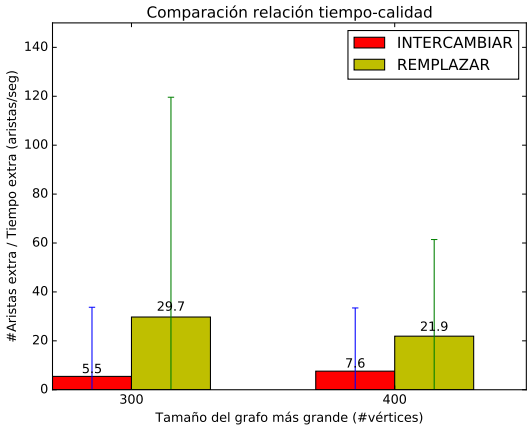


Figura 14:

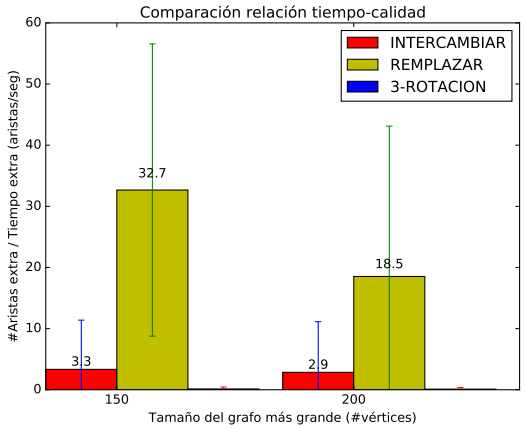


Figura 15:

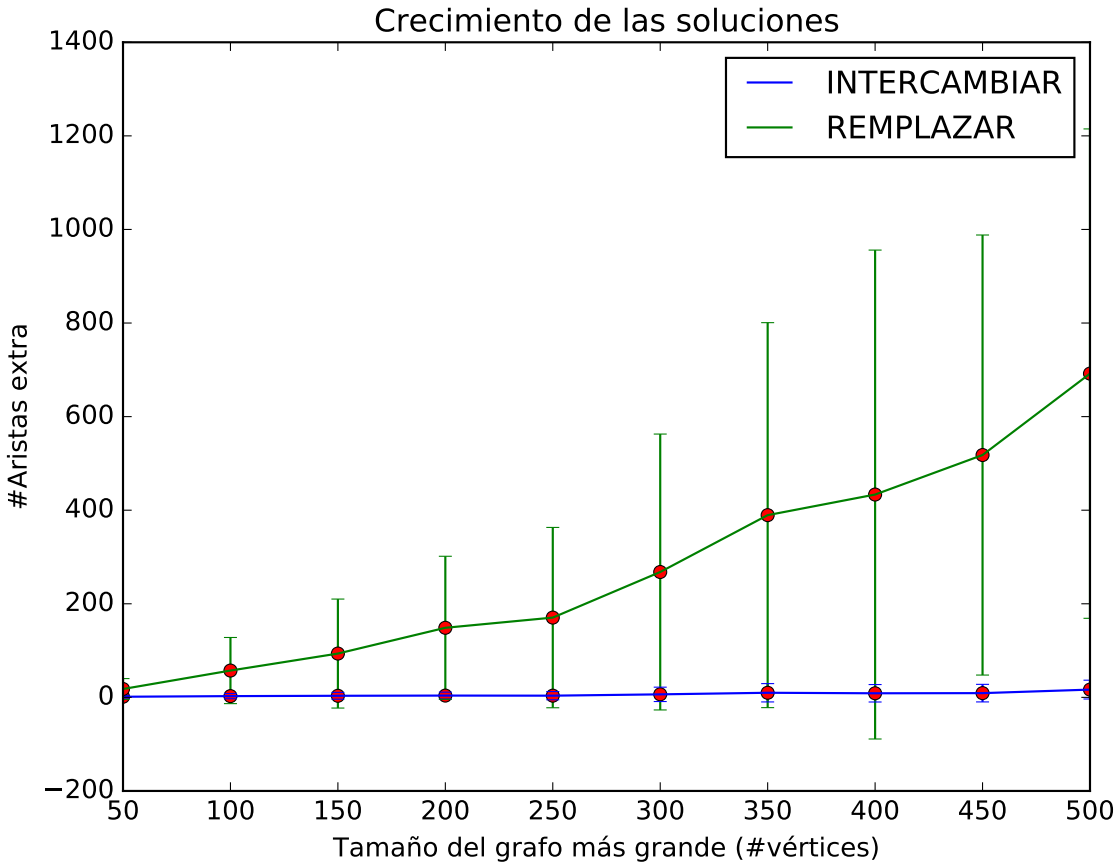

Crecimiento de las soluciones
INTERCAMBIAR
REMPLAZAR
#Aristas extra
Tamaño del grafo más grande (#vértices)
1400
1200
1000
800
600
400
200
0
−200
50
100
150
200
250
300
350
400
450
500

Figura 16:

6. Ejercicio 6: Tabu Search

6.1. Explicación detallada del algoritmo

En la sección anterior analizamos varios algoritmos de búsqueda local. La búsqueda local es una heuristica para algoritmos aproximados. Sin embargo, las heurísticas de búsqueda local pueden verse como un algoritmo de *gradient descent*. Estos algoritmos tienen el problema de los mínimos locales: cuando encuentran un mínimo local se quedan trabados y no pueden mejorar esa solución no óptima.

Por esa razón surgen las metaheuristicas. Una metaheurística es un método heurístico para resolver un problema computacional general. Una meteheurística usa los parámetros dados por el usuario sobre unos procedimientos genéricos y abstractos. Normalmente, estos procedimientos son heurísticos.

En particular, nosotros utilizaremos la metaheurística del tabu search, que a su vez se basa en la heurística de búsqueda local. Una explicación completa de la heurística de tabú search puede encontrarse en [Tab].

Expliquemos los detalles de nuestra implementación antes de ver el pseudocódigo.

Nuestra lista tabú en nuestro caso debería contener soluciones posibles, es decir, isomorfismos. Sin embargo, como comparar isomorfismos es muy caro, decidimos usar una función de hash y almacenar este valor. De esta manera, buscar soluciones en la lista tabú se vuelve mucho menos costoso.

Nuestra función de aspiración, es decir, nuestro método para elegir el isomorfismo inicial de la siguiente iteración hace lo obvio si hay alguna solución no tabú: elige la mejor. Sin embargo, si todas las soluciones son tabú, elegiremos la mejor solución tabú en cuestión.

No creemos que fuera adecuado elegir ningun *movimiento prohibido*, como vimos en la teórica (la definición de movimiento prohibido no existe en la definición original de la metaheuristica tabu search, si no que es una adición posterior), dado que no parecía razonable prohibir ningún movimiento ni ninguna solución particular (más allá de la tabú). Esto se debe principalmente a que no hay características que hagan que, inmediatamente, podamos descartar una solución y declararla inviable.

Sin más que explicar, veamos nuestra implementación.

Algorithm 8 Pseudocódigo del procedimiento Tabu Search

```
1: procedure TABU_SEARCH(Grafo g1, vector<int> vertices1, Grafo g2, vector<int> vertices2)
2:     source ← goloso(g1, vertices1, g2, vertices2)
3:     lista_tabu ← lista(1000, make_pair(0, 0))
4:     indice_lista_tabu ← 0
5:     Inicializar estructuras relacionadas con el criterio de parada.
6:     while criterio de parada do
7:         mejor_tabu ← {.isomorfismo = Isomorfismo(), .aristas = 0}
8:         mejor_solucion ← {.isomorfismo = Isomorfismo(), .aristas = 0}
9:         for  dovecino ∈ vecindad(source)
10:            aristas ← buscar(lista_tabu, hash(vecino))
11:            if aristas = 0 then                          ▷ Source es una solución tabu.
12:                if aristas > mejor_tabu.aristas then
13:                    mejor_tabu.aristas ← aristas
14:                    mejor_tabu.isomorfismo ← vecino
15:                    continue
16:            else
17:                aristas = contar_aristas_isomorfismo(g1, g2, vecino)
18:                if aristas > mejor_tabu.aristas then
19:                    mejor_solucion.aristas ← aristas
20:                    mejor_solucion.isomorfismo ← vecino
21:                    continue
22:        if mejor_solucion.aristas > 0 then
23:            lista_tabu.push_back(
24:              make_pair(hash(mejor_solucion.isomorfismo), mejor_solucion.aristas))
25:            if lista_tabu.size() > lista_tabu_limite then
26:                lista_tabu.pop_front()
27:        if mejor_solucion.aristas = 0 then               ▷ Todas las soluciones son tabu.
28:            source ← mejor_tabu
29:        else
30:            source ← mejor_solucion
31:        Actualizar estructuras relacionadas con el criterio de parada.
```

Para la experimentación, decidimos plantear tres tamaños distintos de lista tabú. A continuación, detallaremos nuestras expectativas. El tamaño más pequeño funcionará muy rápido, dado que la búsqueda es muy efectiva, pero la calidad de los resultados será la peor de entre todas, dado que no podremos guardar muchos resultados previos. El tamaño más grande nos proveerá de los mejores resultados (en promedio) pero la búsqueda en la lista será terriblemente lenta, por lo cual no valdrá tanto la pena. El tamaño intermedio será, según nuestra creencia, un balance entre velocidad y calidad de los resultados.

En cuanto a los criterios de parada, hay varios de entre los cuales elegir, pero todos se reducen a los mismos dos criterios de parada. Analicemoslos.

1. Primero, podemos setear una cantidad de iteraciones global k, y simplemente parar

luego de que la cantidad de iteraciones supera k. Este es el criterio más simple y esperamos que funcione peor que el siguiente.

2. El segundo criterio de parada, es un criterio que se fija cuantas iteraciones pasaron desde que la solución fue mejorada por última vez. Una vez que esa cantidad de iteraciones supera un límite k, el algoritmo terminará.

Esperamos que el segundo criterio funcione mucho mejor, dado que se adapta a lo que esta sucediendo en cada momento: si encuentra cada vez mejores soluciones sigue buscando y si deja de encontrar mejores soluciones termina. Sin embargo, lo bueno del otro criterio de parada, es que tenemos asegurado que va a terminar dentro de un margen de tiempo que podemos determinar muy fácilmente con el límite de iteraciones k.

6.2. Performance del algoritmo

7. Ejercicio 7

A. Apéndice

A.1. Generación de grafos conexos aleatorios

Algorithm 9 Pseudocódigo del procedimiento para generar grafos al azar

```
1: procedure GRAFO_RANDOM(int n, int m)→ Grafo
2:     k_n ← {(0,1),(0,2),...,(0,n),(1,2),(1,3),...,(n−2,n−1)}
3:     grafo ← random.choice(k_n, m) return grafo
```

Este fue el algoritmo que usamos para generar grafos al azar. A diferencia de trabajos anteriores, no era necesario que los grafos fueran conexos.

Aunque el algoritmo es muy simple, nos parece importante mostrarlo y aclarar que los grafos con los que testeamos no necesariamente son conexos.

A.2. Partes relevantes del código

Durante algunas partes del informe al explicar nuestras implementaciones mencionamos u obviamos algunas funciones para no volver demasiado extensa o densa la explicación. Dejamos aquí el pseudocódigo de algunos de los métodos utilizados para evacuar posibles dudas.

Los algoritmos de lectura y escritura no fueron tomados en cuenta para el cálculo de complejidad ya que las mediciones de I/O son muy variables y suelen producir gráficos muy ruidosos. Por otro lado lo que intentabamos cuantificar era la complejidad de los algoritmos para el calculo del MCS propiamente dichos, por lo que no nos pareció provechoso agregar un estudio de estos métodos de lectura y escritura, los cuales son agenos al objetivo principal de estudio.

Algorithm 10 Pseudocódigo del procedimiento para leer la entrada

```
1: procedure LEER_ENTRADA(Grafo g1, Grafo g2)
2:     leo_entrada >> g1.n >> g1.m >> g2.n >> g2.m                          ▷ O(1)
3:     g1.adj_matrix ← vector < vector < bool >> nuevo_vector1[g1.n][g1.n, false]   ▷
   O(g1.n)
4:     g2.adj_matrix ← vector < vector < bool >> nuevo_vector2[g2.n][g2.n, false]   ▷
   O(g2.n)
5:     g1.grafos = vector<int> vector[g1.n, 0]                              ▷ O(g1.n)
6:     g2.grafos = vector<int> vector[g2.n, 0]                              ▷ O(g2.n)
7:     for int i = 0; i < g1.m; i++ do
8:         int u, v                                                         ▷ g1.m veces O(1)
9:         leo_entrada >> u >> v                                            ▷ g1.m veces O(1)
10:        g1.adj_matrix[u][v] = true                                       ▷ g1.m veces O(1)
11:        g1.adj_matrix[v][u] = true                                       ▷ g1.m veces O(1)
12:        g1.grados[u]++                                                   ▷ g1.m veces O(1)
13:        g2.grados[v]++                                                   ▷ g1.m veces O(1)
14:    for int i = 0; i < g2.m; i++ do
15:        int u, v                                                         ▷ g2.m veces O(1)
16:        leo_entrada >> u >> v                                            ▷ g2.m veces O(1)
17:        g2.adj_matrix[u][v] = true                                       ▷ g2.m veces O(1)
18:        g2.adj_matrix[v][u] = true                                       ▷ g2.m veces O(1)
19:        g2.grados[u]++                                                   ▷ g2.m veces O(1)
20:        g2.grados[v]++                                                   ▷ g2.m veces O(1)
```

Algorithm 11 Pseudocódigo del procedimiento para imprimir la solucion

```
1:  procedure IMPRIMIR_SOLUCIONH(bool inverso, vector<int,int> aristas, MCS solucion)
2:      imprimo_valores >> solucion.isomorfismo.tamanio() >> "" >> aristas.tamanio() >> "" >> final_de_linea                ▷ O(1)
3:      for p ∈ solucion.isomorfismo do
4:          if ¬inverso then                                  ▷ A lo sumo g1.n veces O(1)
5:              imprimo_valores >> p.primero >>               ▷ g1.n veces O(1)
6:          else
7:              imprimo_valores >> p.segundo >>               ▷ g1.n veces O(1)
8:      for p ∈ solucion.isomorfismo do
9:          if ¬inverso then                                  ▷ g1.n veces O(1)
10:             imprimo_valores >> p.segundo >>               ▷ g1.n veces O(1)
11:         else
12:             imprimo_valores >> p.primero >>               ▷ g1.n veces O(1)
```

El programa analiza de cuantas aristas sería el isomorfismo si se le agregara el mapeo entre $v1$ y $v2$, siendo estos nodos de $g1$ y $g2$ respectivamente.

Algorithm 12 Pseudocódigo del procedimiento para contar aristas del isomorfismo

```
1:  procedure CONTAR_ARISTAS_ISOMORFISMO_AGREGAR(Grafo g1, Grafo g2, int v1, int v2, Isomorfismo iso ) → int
2:      int aristas = 0                                       ▷ O(1)
3:      for i = 0; i ≤ iso.size(); i++ do
4:          if i < iso.tamanio then                           ▷ g1.n veces O(1)
5:              int vg1 = iso[i].primero                      ▷ g1.n veces O(1)
6:              int vg2 = iso[i].primero                      ▷ g1.n veces O(1)
7:          else
8:              int vg1 = v1                                  ▷ g1.n veces O(1)
9:              int vg2 = v2                                  ▷ g1.n veces O(1)
10:         for j = 0; j ≤ iso.size(); j++ do
11:             if i < iso.tamanio then                       ▷ g1.n veces O(1)
12:                 int ug1 = iso[j].primero
13:                 int ug2 = iso[j].primero
14:             else
15:                 int ug1 = v1                              ▷ g1.n veces O(1)
16:                 int ug2 = v2                              ▷ g1.n veces O(1)
17:             if g1.adj_matrix[vg1][ug1] ∧ g2.adj_matrix[vg2][ug2] then
18:                 aristas++                                 ▷ g1.n veces O(1)
        return aristas/2
```

Hay dos pequeñas variantes a esta función que son utilizadas en nuestro código, estas son *contar_aristas_isomorfismo* y *hallar_aristas_isomorfismo*, dado un isomorfismo la primera calcula la cantidad de aristas y la segunda averigua y almacena las aristas

del mismo.

Referencias

[Tab] "Tabu Search—Part I". En: *ORSA Journal on Computing* 1.3 (1989), págs. 190-206. DOI: 10.1287/ijoc.1.3.190. eprint: http://dx.doi.org/10.1287/ijoc.1.3.190.