DEPARTAMENTO
DE COMPUTACION
Facultad de Ciencias Exactas y Naturales - UBA

Trabajo Práctico Número 3

24 de Junio 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 *solucion*, que tiene 2 campos, *aristas* e *isomorfismo*. En esta variable global irá guardando la mejor solución encontrada hasta el momendo. *solucion.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_1 v_2 veces O(1)
14:            nuevo_iso = iso                                                       ▷ v_1 v_2 veces O(n_1)
15:            nuevo_iso.push_back(make_pair(u, v))                                  ▷ v_1 v_2 veces O(1)
16:            bt(g1, g2, copiar_sin(vertices1, u), copiar_sin(vertices2, v), nuevo_iso)   ▷ v_1 v_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

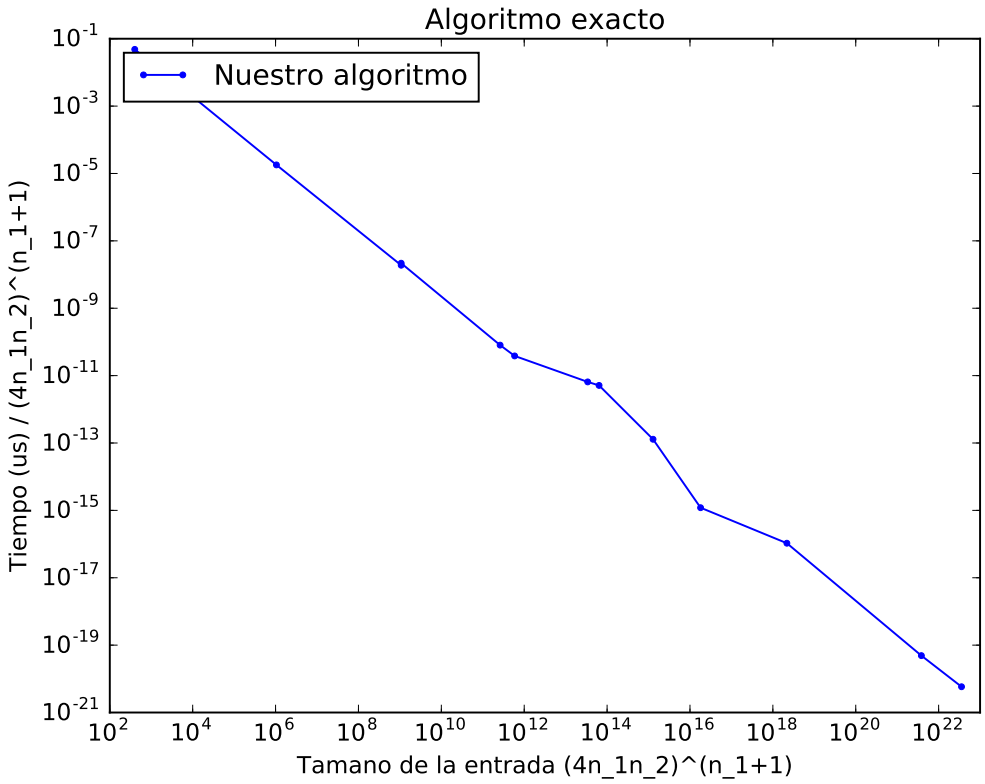


Figura 1

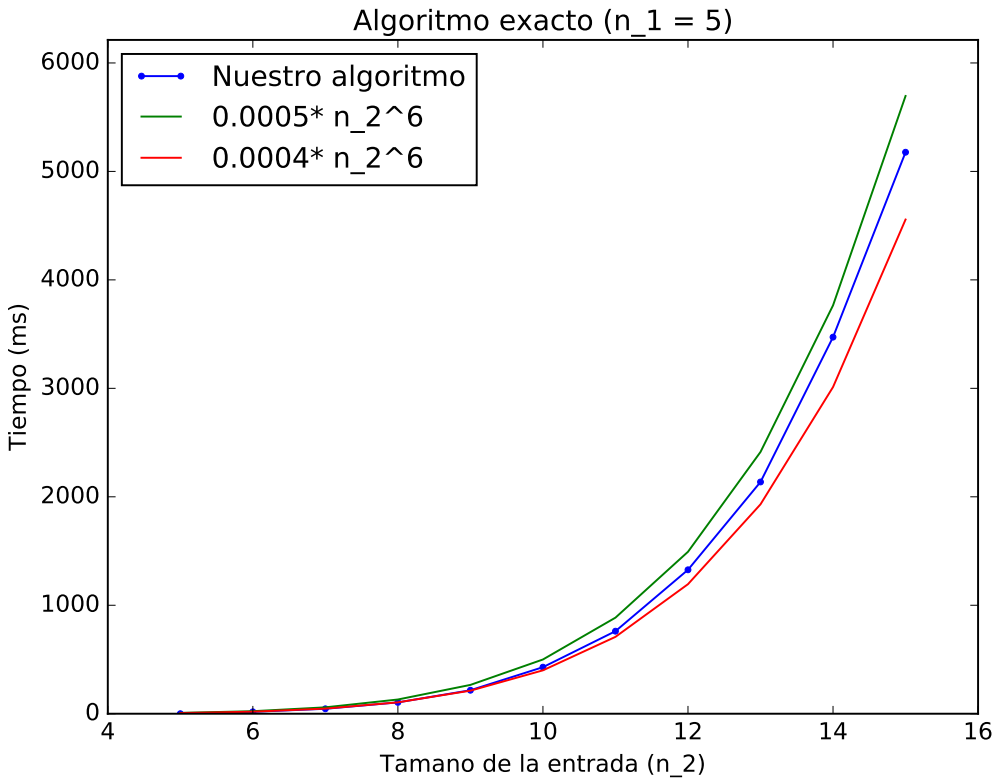

Algoritmo exacto (n_1 = 5)
Nuestro algoritmo
0.0005* n_2^6
0.0004* n_2^6
Tiempo (ms)
Tamano de la entrada (n_2)
0
1000
2000
3000
4000
5000
6000
4
6
8
10
12
14
16

Figura 2

3. Ejercicio 3

3.1. Problema del MCS entre cografos y grafos completos

En esta sección se presenta un algoritmo exacto para resolver el problema en cuestión para el caso particular en que los dos grafos de entrada son un cografo y un grafo completo, respectivamente. El problema del MCS para este tipo de grafos de entrada puede resolverse en tiempo polinomial, debido a características particulares de este tipo de grafos.

En principio, si consideramos el problema del MCS entre un grafo cualquiera G y un grafo completo Kn, se deduce fácilmente que si $|Kn| \geq |G| \Rightarrow MCS(G, Kn) = G$. Por lo tanto, el caso interesante que se busca resolver es cuando $|G| > |Kn|$.

Para esto, el algoritmo presentado aprovecha la siguiente definición recursiva de un cografo:

def. ***Cografo***

1. *El grafo trivial es un cografo.*
2. *La unión ($\cup$) disjunta de dos cografos es un cografo.*
3. *El join ($\vee$) de dos cografos es un cografo, donde* $G1 \vee G2 = \overline{\overline{G1} \cup \overline{G2}}$.

3.2. Explicación detallada del algoritmo

El algoritmo propuesto utiliza la técnica de programación dinámica. Aprovechando la definición recursiva de los cografos, dividimos el problema en subproblemas más chicos. Dada la definición, un cografo puede ser un grafo trivial, unión o join de dos cografos. Se define la siguiente función recursiva:

$f(G, K, i) = MCS(G, K)$ usando i nodos, con $i \leq min\{|G|, |K|\}$

donde:

$$f(G,K,i) = \begin{cases} G & si\ G\ trivial \\ \underset{0 \leq j \leq i}{max Aristas}\ \{MCS(G1,K,j) \cup MCS(G2,K,i-j)\} & si\ G = G1 \cup G2 \\ \underset{0 \leq j \leq i}{max Aristas}\ \{MCS(G1,K,j) \vee MCS(G2,K,i-j)\} & si\ G = G1 \vee G2 \end{cases}$$

La solución que buscamos, entonces, es: $f(G, K, min\{|G|, |K|\})$.

3.2.1. Correctitud de la formulación recursiva

- **Caso trivial.** El MCS de un grafo trivial con un grafo completo es ($\{v\}$, $\{\}$), donde v es el único nodo de G.

- **Caso G es unión de cografos $G1, G2$.** Por transitividad, MCS$(Gi, K, j) \in G$ pues es subgrafo de Gi y Gi es subgrafo de G $\forall i \in (1, 2)$. Además, toda arista y nodo del $MCS(G, K, j)$ de j vertices pertenece bien a $G1$ o a $G2$. Por lo que podemos calcularlo quedándonos con el MCS de máxima cantidad de aristas resultante entre todas las combinaciones posibles de usar x cantidad de nodos de $G1$ y $j - x$ nodos de $G2$ ($x \in [0..j]$). Para ver cuál es el mejor MCS resultante es necesario calcular el x que maximice la cantidad de aristas de la unión entre los $MCS(G1, K, x)$ y $MCS(G2, K, j-x)$ (para todo x válido), ya que vale el principio de optimalidad. Si utilizáramos otro subgrafo de $G1$ o $G2$ que no fuera MCS del mismo, entonces el MCS resultante para G no sería óptimo, pues podríamos intercambiarlo por el óptimo y obtener un MCS mejor para G.

- **Caso G es join de cografos $G1, G2$.** Para este caso, debemos considerar las aristas que se agregan al hacer el join. Al igual que en el caso anterior, todo nodo del $MCS(G, K, j)$ de j vertices pertenece bien a $G1$ o a $G2$. Por lo que podemos calcularlo quedándonos con el MCS de máxima cantidad de aristas resultante entre todas las combinaciones posibles de usar x cantidad de nodos de $G1$ y $j - x$ nodos de $G2$ ($x \in [0..j]$). Sin embargo, no podemos quedarnos con la unión de los MCS de los subgrafos tal que la suma de las aristas es la máxima como en el caso anterior, ya que hay aristas que no estamos considerando. Éstas aristas son las aristas que se agregan cuando se realiza el join entre dos grafos, y son todas aquellas que unen cada nodo de una componente del join a todos los nodos de la otra. Por lo que, para calcular la mejor combinación posible de nodos entre $G1$ y $G2$ debemos sumar la cantidad de aristas entre el $MCS(G1, K, x)$ y $MCS(G2, K, j - x)$. Entonces, buscamos la combinación que maximice lo siguiente:

  $$aristas(MCS(G1, K, x)) + aristas(MCS(G2, K, j - x)) + x * (j - x) \forall x \in [0..j]$$

  donde $x * (j - x)$ corresponde a la cantidad de aristas con un extremo en $G1$ y otro en $G2$. Al igual que en el caso anterior, si utilizáramos otro subgrafo de $G1$ o $G2$ que no fuera MCS del mismo, entonces el MCS resultante para G no sería óptimo, pues podríamos intercambiarlo por el óptimo y obtener un MCS mejor para G.

□

Como hay soluciones que se deben calcular más de una vez, utilizando programación dinámica nos guardamos las soluciones parciales resueltas para utilizar luego, reduciendo la complejidad final del algoritmo. Para implementar esta función, se realizó una implementación de una clase *CographTree* basada en una estructura de árbol binario de cografos, para representar el grafo de entrada, y poder aplicar la recursión. La raíz del árbol es el cografo original de entrada, y tiene dos cografos hijos en caso de ser unión o join de éstos (las hojas del árbol son grafos triviales).

A grandes rasgos, el algoritmo para resolver este problema se divide en 4 partes:

1. Lectura de entrada
2. Generación de la estructura *CographTree* a partir del grafo de entrada

3. Cálculo del MCS

4. Impresión de la solución

A diferencia del resto de los algoritmos, los datos de entrada se guardan en una estructura *Grafo_ adjlist* basada en una lista de adyacencias. Dicha estructura utiliza un diccionario(nodo, lista<nodos>) para representar un grafo. Esta estructura permite, una vez calculados los nodos correspondientes al MCS, calcular fácilmente el subgrafo inducido por dichos nodos en el grafo original, manteniendo la numeración original de los nodos.

3.2.2. Generación del CographTree

A continuación se encuentra el pseudocódigo de la función utilizada para inicializar el CographTree correspondiente al grafo de entrada:

Algorithm 3 Pseudocódigo del constructor de CographTree

```
1: procedure CographTree(Grafo_adjlist g)
2:     n = g.n
3:     m = g.m
4:     nodos = nodosIds(g)
5:     sol =< false, Solucion > (n + 1)
6:       ▷ El vector sol se utiliza para almacenar las soluciones parciales usando de 0 a g.n
       nodos. Se inicializa en false para indicar que la dicha solución todavía no fue resuelta.
7:     componentes = obtenerComponentesCografo(g, false)
8:     if |componentes| == 2 then
9:         tipo = unionDisjunta
10:        izq = CographTree(subgrafoInducido(g, componentes[0])
11:        der = CographTree(subgrafoInducido(g, componentes[1])
12:    else if |componentes| == 1 then
13:        tipo = trivial
14:        izq = null
15:        der = null
16:    else
17:        tipo = join
18:        componentesJoin = obtenerComponentesCografo(g, true)
19:        izq = CographTree(subgrafoInducido(g, componentesJoin[0])
20:        der = CographTree(subgrafoInducido(g, componentesJoin[1])
```

El algoritmo se basa primero en verificar si el cografo de entrada es conexo. En caso de que no lo sea, entonces por definición el cografo es unión de otros dos (un join de dos grafos siempre es conexo, ver demostración debajo). Si el cografo es conexo, puede ser un grafo trivial o un join de otros dos cografos.

dem. *Sea G un grafo tal que $G = G1 \vee G2$, $\Rightarrow$ G es conexo.* Sea v un vértice cualquiera. Por definición de join, $\forall v \in G1, u \in G2, \exists (u, v) \in G$. Spge, asumo $v \in G1$. Entonces, existe un camino desde v a todos los nodos pertenecientes a $G2$, ya que está

conectado a todos ellos. Sea u un nodo $\in G2$. Por la misma razón u está conectado a todos los nodos de $G1$. Por lo tanto existe un camino desde v a todos los demás nodos de $G1$ pasando por u. Entonces, G es conexo. □

Para verificar si el grafo es conexo, se calculan las componentes conexas llamando a *obtenerComponentesCografo*. Esta función está basada en BFS, pero a diferencia de éste, si el grafo tiene más de dos componentes conexas: $C_1, C_2, ..., C_k$, sólo devuelve dos componentes, $C'_1 = C_1$ y $C'_2 = C_2 \cup C_3 \cup ... \cup C_k$. De esta manera se logra que el CographTree generado finalmente sea binario. El segundo parámetro de la función es un booleano que se usa para indicar si lo que se buscan son las dos componentes de un cografo Join. (es decir se buscan $G1, G2$ tal que $G = G1 \vee G2$). Para esto, se introduce una modificación al BFS, tal que en lugar de recorrer los adyacentes a cada nodo, se recorren los nodos no adyacentes de los mismos, para calcular los nodos correspondientes a uno de los grafos del join. A continuación una demostración de que el algoritmo con dicha modificación calcula efectivamente las dos componentes de un join.

dem. Dado $G, G1, G2$ grafos, tal que $G = G1 \vee G2$. Es decir, $\forall v \in G1, u \in G2, \exists (u, v) \in G$. Comenzando con un nodo aleatorio x, marco todos los nodos no adyacentes. Spge, supongo $x \in G1$. Estos nodos se encuentran necesariamente en G1, ya que x es adyacente a todos los nodos de G2. Luego, marco los nodos no adyacentes a dichos nodos, que también, por la misma razón, pertenecen a G1. De esta manera, iterativamente, marcamos todos los nodos pertenecientes a G1. El resto de los nodos que quedó sin marcar, pertenece a la otra componente del join. □

Una vez determinadas las dos componentes del cografo (sea unión o join), se aplica recursivamente para construir los cografos correspondientes a los subgrafos inducidos por los nodos de dichas componentes y continuar armando el árbol de cografos hasta completar las n hojas con grafos triviales.

Notar que la estructura `CographTree` sólo almacena los ids de los nodos del grafo original, y no las adyacencias. Esto basta ya que una vez calculada la solución puede obtenerse el subgrafo inducido por los nodos de la misma sin necesidad de almacenar y recalcular las adyacencias de cada cografo del árbol.

3.2.3. Pseudocódigo del algoritmo MCS para cografos

A continuación, el pseudocódigo de la función para calcular el MCS dado una cantidad de nodos y el tamaño del grafo completo: (notar que la función devuelve los ids de los nodos del grafo original que corresponden al MCS).

Algorithm 4 Pseudocódigo de MCS entre Cografo y Grafo Completo

```
1: procedure MCS_SOL(CographTree cografo, int cantNodos, tamGrafoCompleto ) → Solucion: <cantAristasSol, nodosSol>
2:     if cantNodos > cografo.n then
3:         return mcs_Sol(cografo.n, tamGrafoCompleto)
4:     else
5:         int cantAristasSol
6:         vector<int> nodosSol
7:         if cantNodos == 0 then
8:             cantAristasSol = 0
9:         else if cografo.sol[cantNodos].resuelto? then
10:            return cografo.sol[cantNodos].solucion
11:        else if cantNodos == 1 OR cografo.tipo == trivial then
12:            cantAristasSol = 0
13:            nodosSol = cografo.nodos[0]
14:        else if cografo.n ≤ tamGrafoCompleto AND cantNodos == cografo.n then
15:            cantAristasSol = 0
16:            nodosSol = cografo.nodos
17:        else
18:            int maxAristas = 0
19:            for i ∈ [0..cantNodos] do
20:                solIzq = mcs_Sol(cografo.izq, i, tamGrafoCompleto)
21:                solDer = mcs_Sol(cografo.der, cantNodos − i, tamGrafoCompleto)
22:                aristas = solIzq.cantAristas + solDer.cantAristas
23:                if cografo.tipo == join then
24:                    aristas+ = |solIzq.nodosSol| ∗ |solDer.nodosSol|
25:                if aristas > maxAristas AND (|solIzq.nodosSol| + |solDer.nodosSol| == cantNodos) then
26:                    maxAristas = aristas
27:                    nodosSolIzq = solIzq.nodosSol
28:                    nodosSolDer = solDer.nodosSol
29:            cantAristasSol = maxAristas
30:            if cantAristasSol == 0 then
31:                for i ∈ [0..cantNodos) do
32:                    nodosSol.agregar(nodos[i])
33:            else
34:                nodosSol = unirNodos(nodosSolIzq, nodosSolDer)
35:        Solucion s =< cantAristasSol, nodosSol >
36:        cografo.sol[cantNodos].resuelto? = true
37:        cografo.sol[cantNodos].solucion = s
38:        return s
```

3.3. Complejidad del Algoritmo

La complejidad del algoritmo se puede calcular como: C (Leer entrada) + C (Generación de CographTree) + C (MCS) + C (Imprimir)

- `Leer entrada.` $O(n+m)$
- `Generación de CographTree.` Para generar esta estructura recursiva, es necesario inicializar cada cografo del árbol. Para eso, para cada uno de ellos se ejecuta `obtenerComponentesCografo` al menos una vez para chequear si el grafo correspondiente es conexo usando BFS $O(n+m)$. Además, si el grafo es conexo se ejecuta con la modificación sobre el BFS, recorriendo los nodos no adyacentes en lugar de los adyacentes. Con esta modificación, la complejidad del algoritmo es $O(n^2)$. Por lo tanto la complejidad quedaría:

$$\sum_{i=1}^{tam(CographTree)} O((n_i)^2 + m_i) =$$

$$= O(tam(CographTree) * n^2) + O(tam(CographTree) * m)$$

notar que la suma de los n_i y los m_i pertenecientes a cada nivel del CographTree es $O(n)$ y $O(m)$ respectivamente.

$$= O(n * n^2) + O(n * m)$$

acotamos $tam(CographTree)$ por n pues el cotree tiene obviamente n hojas (una por cada nodo) y como es binario completo, la cantidad de nodos es aproximadamente $2n$.

$$= O(n^3 + n * m)$$

donde n_i y m_i son los correspondientes al i-ésimo cografo del CographTree.

- `MCS.` Por cada cografo del CographTree debemos chequear cuál es la mejor combinación de nodos, para eso debemos probar con n_i posibles combinaciones. Cada solución parcial se resuelve una sola vez, ya que vamos guardando las soluciones en el vector `sol`. Por lo que la complejidad queda:

$$\sum_{i=1}^{tam(CographTree)} O(n_i) =$$

$$= O(tam(CographTree) * n)$$
$$= O(n * n)$$
$$= O(n^2)$$

- `Imprimir.` $O(min(n, n_K) + min(m, m_K))$, donde n_K y m_K corresponden al grafo completo de entrada.

La complejidad total queda: $O(n^3 + n * m + min(n, n_K) + min(m, m_K))$

3.4. Performance del Algoritmo

Confirmemos experimentalmente las complejidades enunciadas anteriormente. Como siempre, dejamos de lado la parte de entrada salida, dado que hace que la varianza sea muy alta, y no vale la pena en general medir esta parte porque no es lo central del algoritmo. Por lo tanto, esperaremos que lo que medimos tenga complejidad $O(n^3 + nm)$.

Para experimentar, utilizamos el algoritmo de generación de cografos al azar que analizaremos en la siguiente sección. Medimos los tiempos manualmente, y los graficamos como explicamos en la introducción.

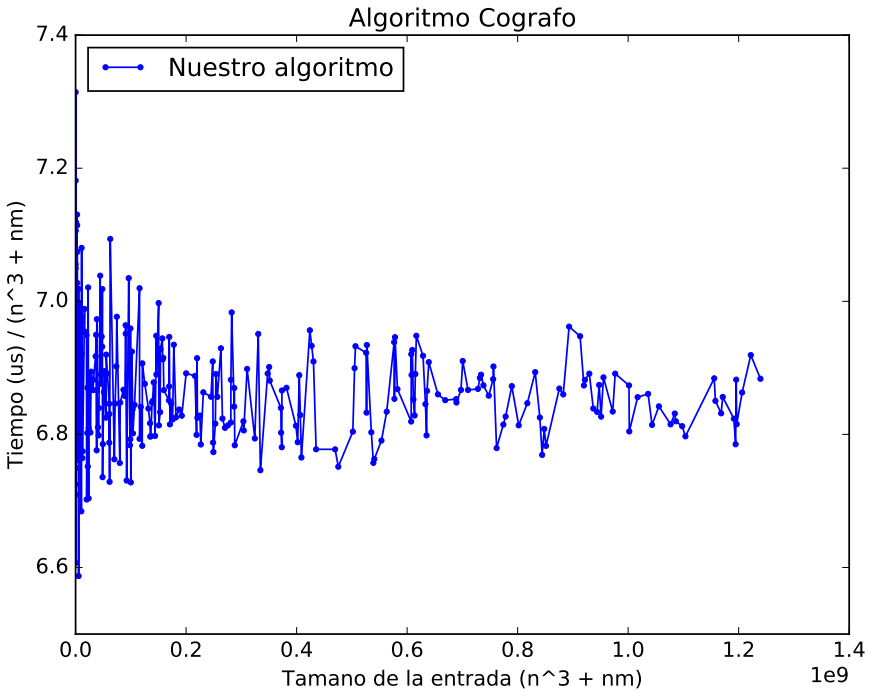


Figura 3

Con este experimento confirmamos lo que esperabamos.

Además, podemos comprobar que si fijamos n, todo anda como debería, es decir, la complejidad es lineal en m.

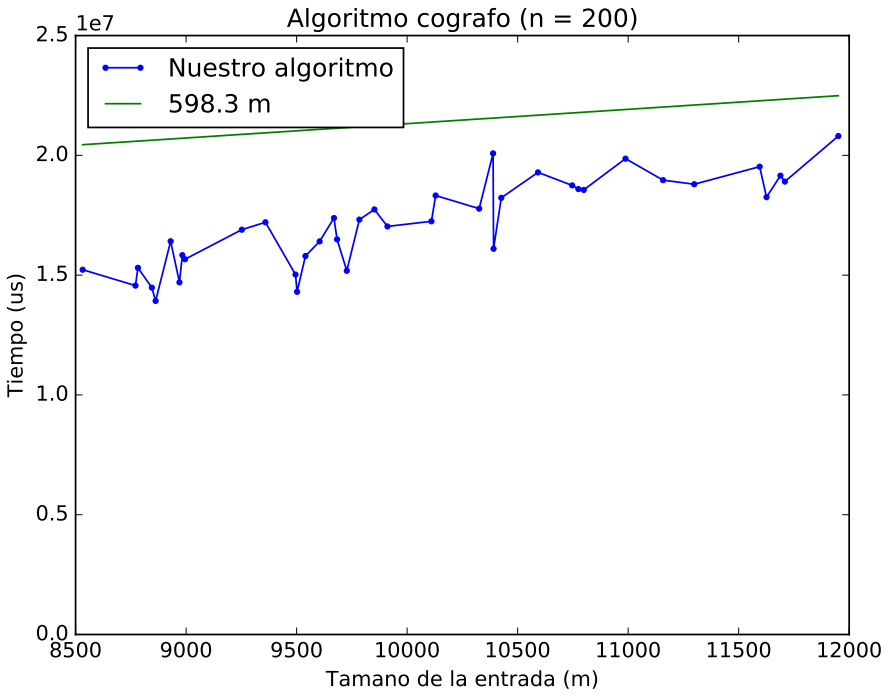


Figura 4

Por último, por completitud, es importante decir que es muy díficil, de hecho imposible, hacer el experimento en el que se fija m y se varia n, porque nada nos garantiza que existan suficientes cografos con cantidad de aristas m de distinto n como para hacer un experimento significativo, y además no es claro como generarlos (es decir, dado un m como encontrar todos o varios cografos de distintos n).

3.5. Generación y testing de casos random

Para testear la correctitud del algoritmo, primero se ejecutó el mismo con grafos de entrada más chicos y se comparó la solución con el algoritmo exacto (Problema anterior). Además, se implementó un generador de cografos random para calcular la performance, testear la correctitud y utilizar como parámetro de comparación en la posterior experimentación de las heurísticas. A continuación se presenta un algoritmo para generar cografos de manera random:

Algorithm 5 Pseudocódigo del constructor random de un CographTree

```
1:  procedure CographTree(int size, int nodoInicial)
2:                                  ▷ nodoInicial corresponde al id del primer nodo.
3:      n = size
4:      for i ∈ [nodoInicial..nodoInicial + size] do
5:          nodos.agregar(i)
6:      sol =< false, Solucion > (n + 1)
7:      if size == 1 then
8:          tipo = trivial
9:          izq = null
10:         der = null
11:     else
12:         nodosIzq = (rand() mod (n − 1)) + 1
13:         nodosDer = n − nodosIzq
14:         izq = CographTree(nodosIzq, nodoInicial)
15:         der = CographTree(nodosDer, nodoInicial + nodosIzq)
16:         int trand = rand() mod 2
17:         if trand == 0 then
18:             tipo = unionDisjunta
19:             m = izq.m + der.m
20:         else
21:             tipo = join
22:             m = izq.m + der.m + (nodosIzq ∗ nodosDer)
```

Junto con el informe y el resto de los problemas se agrega un archivo `testerproblema3`, que puede ser utilizado para generar casos random y chequear correctitud de los mismos usando el algoritmo planteado anteriormente.

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 nos permitirá tomar desiciones rapidamente a partir de ciertas suposiciones o atajos no necesariamente validas siempre, pero muchas veces útiles. 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 para este ejercicio. La primer versión consiste en dados los dos grafos, G_1 y G_2, 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.

A continuación se puede ver el pseudo-código de dicho programa.

Algorithm 6 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, 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 cuales son los ejes que 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 la 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 7 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(n1)
5:     int vertice2 = mayor_adj(vertices2, g2)                               ▷ O(n2)
6:     solucion.isomorfismo.insertar_atras(< vertice1, vertice2 >)           ▷ O(1)
7:     vertices1.borrar(vertice1)                                            ▷ O(log(n1))
8:     vertices2.borrar(vertice2)                                            ▷ O(log(n2))
9:     while vertices1.tamanio() ≠ 0 do
10:        par<int,int> par_mayor_deg =< vertices1.primero(), vertices2.primero() >
   ▷ n1 veces O(1)
11:        for u ∈ vertices1 do
12:            for v ∈ vertices2 do
13:                int aristas                                               ▷ n1^2 n2 veces O(1)
14:                aristas = contar_aristas_isomorfismo(g1, g2, u, v, solucion.isomorfismo)
   ▷ n1^2 n2 veces O(n1^2)
15:                if aristas > solucion.aristas then                        ▷ n1^2 n2 veces O(1)
16:                    solucion.aristas = aristas                            ▷ n1^2 n2 veces O(1)
17:                    par_mayor_deg =< u, v >                               ▷ n1^2 n2 veces O(1)
18:        solucion.isomorfismo.insertar_atras(par_mayor_deg)                ▷ n1 veces O(1)
19:        vertices1.borrar(par_mayor_deg.primero)                           ▷ n1 veces O(log(n1))
20:        vertices1.borrar(par_mayor_deg.segundo)                           ▷ n1 veces O(log(n2))
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 una precondición del programa).

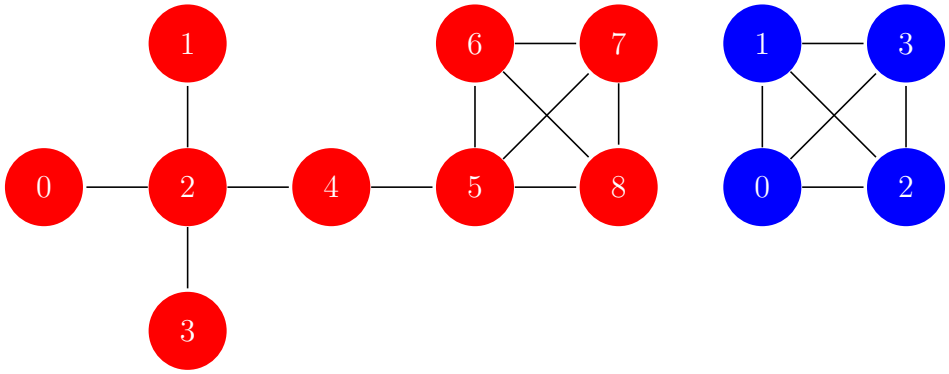
En la segunda heuristica la complejidad es precisamente $O(n_1^4 n_2)$. Esto resulta de los 3 ciclos anidados que hay mas una operación de costo cuadratico respecto de n_1. Si sumamos todo lo que se detalló en el pseudocódigo se puede ver que queda $O(n_1 + n_2 + \log(n_1) + \log(n_2) + n_1^2.n_2 + n_1^2.n_2.n_1^2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

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 $n-1$ 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. En rojo se observa a G_1 y en azul a G_2



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 algún 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, G_2, con el grafo completo contenido en G_1.

Para estos casos la salida de la heuristica, llamemosla $H(n)$, 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 ρ-aproximado, ya que 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 principalmente a corroborar empíricamente ciertas hipótesis sobre nuestra segunda heurística 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.

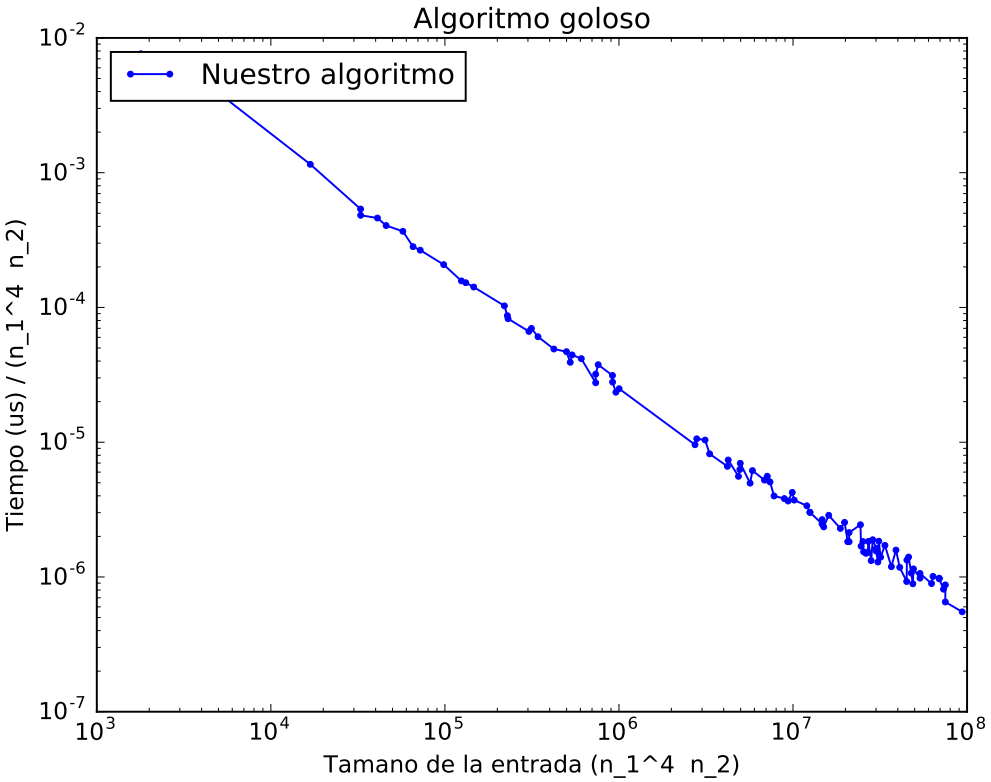


Figura 5

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 al mismo tiempo 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.

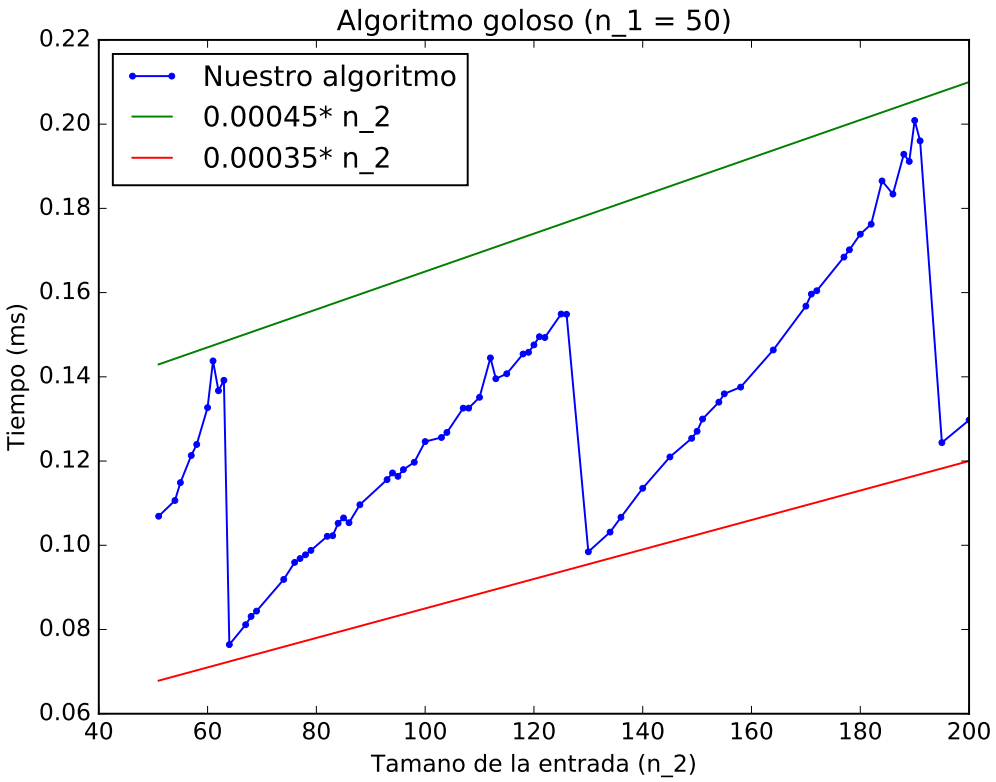


Figura 6

Este gráfico tiene dos particularidades que vale 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.

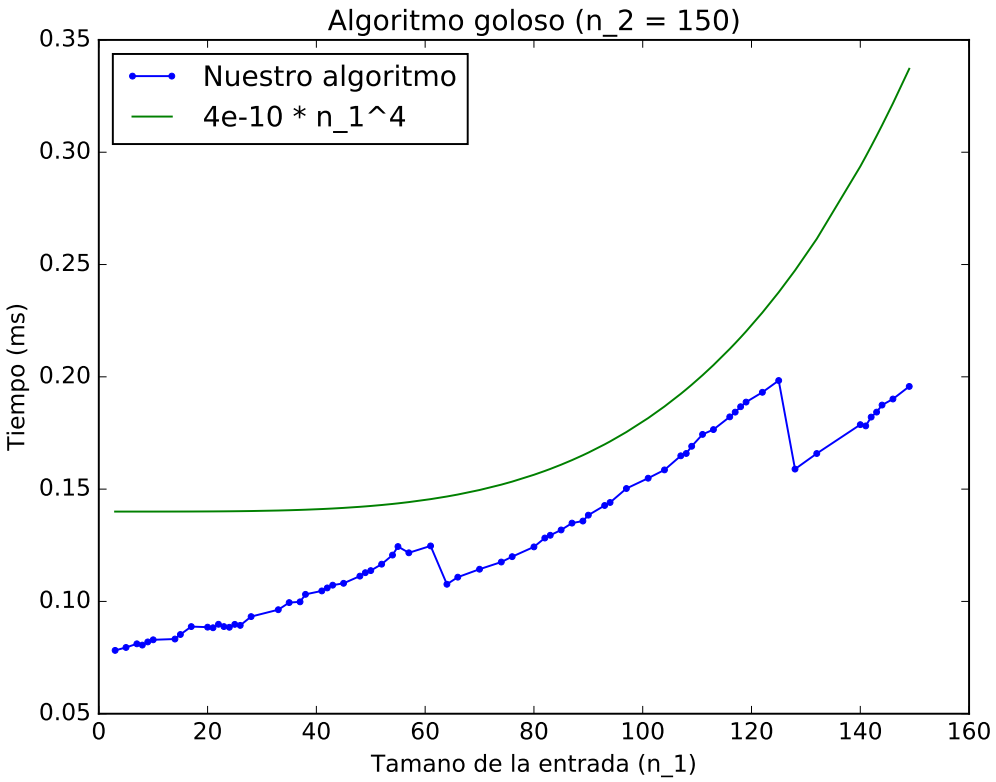


Figura 7

En este gráfico se ve como acotamos superiormente las instancias generadas con la complejidad que habiamos calculado, 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.

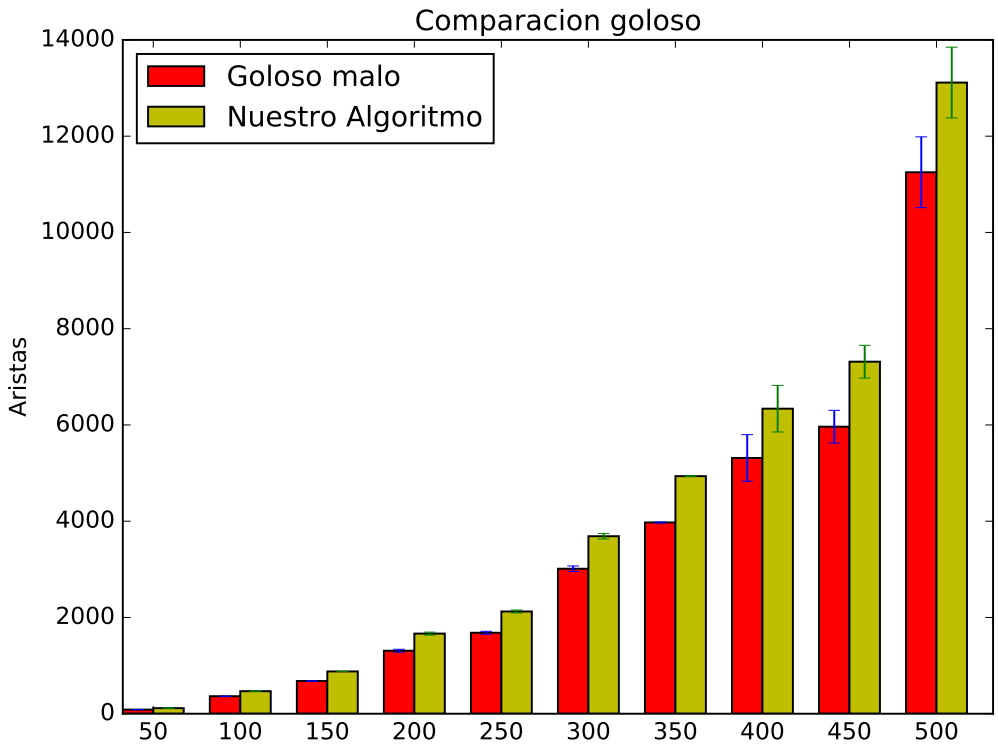


Figura 8

Se puede ver como nuestro algoritmo es siempre superior en promedio a la otra heuristica. Aunque la ventaja no es inmensa tampoco, 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: Búsqueda Local

5.1. Introducción

En la sección anterior implementamos y analizamos una heurística golosa para atacar el problema planteado. Si bien resulta bastante deseable desde el punto de vista de la complejidad temporal (polinomial de grado razonable), tiene el problema de que, como vimos, la solución que nos provee puede ser arbitrariamente mala y en el caso general no tenemos ningún tipo de garantía de qué tan cerca puede estar de una solución verdaderamente óptima.

Lo que quisiéramos entonces es poder refinar la solución que nos devuelve la heurística golosa para que se acerce más al máximo real. Para esto probaremos 3 algoritmos de búsqueda local. Dichos algoritmos se basan en la noción de "vecindad" de soluciones. Dada una solución fuente, una vecindad de S, $N(S)$, es un subconjunto del conjunto de posibles soluciones (en nuestro caso un subconjunto del conjunto de subgrafos comunes) que a diferencia de este último tiene un cardinal lo suficientemente pequeño como para poder ser recorrido en tiempo polinomial. Entonces los algoritmos de búsqueda local se diferenciarán a partir de la vecindad escogida en cada caso.

Es importante resaltar que los algoritmos de búsqueda local no dejan de ser algoritmos heurísticos cuya calidad dependerá tanto de la solución fuente utilizada como de la vecindad escogida. Los algoritmos solo nos garantizan que encontraremos el máximo de la vecindad (máximo local) que no necesariamente es el máximo global.

Otra cosa importante que también será común a todas las búsquedas es que cuando encontremos un máximo en la vecindad, si es mejor que la solución *source*, no nos conformaremos con esto sino que pasaremos a explorar la vecindad correspondiente a esta nueva solución. De forma que se iterará la búsqueda hasta que se llegue a una solución que sea máxima en su vecindad.

En particular nosotros escogimos 3 vecindades para el problema, bajo los siguientes rótulos:

- INTERCAMBIAR
- REEMPLAZAR
- 3-ROTACION

En todos los casos la solución fuente inicial será la dada por el algoritmo goloso.

5.2. Explicación detallada de los algoritmos

En pos de la claridad separamos la explicación y el pseudocódigo de las tres vecindades.

La notación usada será $n_1 = |V(G_1)|$, $n_2 = |V(G_2)|$, $m_1 = |E(G_1)|$ y $m_2 = |E(G_2)|$. Asumimos, por ser precondición para usar el algoritmo goloso y la heurística REMPLAZAR,

que $n_1 \leq n_2$.

5.2.1. INTERCAMBIAR

Un aspecto importante de nuestro algoritmo goloso es que siempre mapea exactamente todos los vértices del grafo con menor número de vértices a la hora de devolver la solución. En esta primera vecindad lo que planteamos es considerar todos los *swapeos* posibles para el mapeo de la solución *source*. Formalmente, si $S = \{(v_0, w_{i_1}), \ldots, (v_n, w_{i_n})\}$ es el mapeo de la solución *source*, entonces la vecindad de tipo INTERCAMBIAR asociada a S es $N_I(S) = \{S' : S' = \{(v_0, w_{i_1}), \ldots, (v_p, w_{i_q}), \ldots, (v_q, w_{i_p}), (v_n, w_{i_n})\}\}$.

Algorithm 8 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)
```

Complejidad: $O(minn_1, n_2)$

5.2.2. REEMPLAZAR

Recordar que $n_1 \leq n_2$ por precondición. En particular, para esta vecindad solo nos interesarán los casos en que $n_1 < n_2$. Si ambos tamaños fueran iguales el algoritmo no modificará la solución golosa.

Debido a esta condición extra que asumimos sobre el tamaño de los grafos, para cualquier mapeo que tengamos como solución (en particular, el dado por el algoritmo goloso) siempre hay vértices de G_2 que no se están mapeando.

La motivación de esta vecindad entonces es ver qué sucede con la cantidad de aristas cuando en el mapeo de la solución fuente reemplazamos nodos de G_2 con otros que no habíamos usado originalmente.

Formalmente, si $S = \{(v_0, w_1), \ldots, (v_n, w_n)\}$ es el mapeo de la solución fuente y $R = \{z_1, \ldots, z_r\}$ son los vértices de G_2 que no se mapearon en S, entonces
$N(S) = \{S' : S' = S \backslash \{(v_i, w_i)\} \cup \{(v_i, z_i)\}\}$.

Algorithm 9 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)
```

5.2.3. 3-ROTACION

Esta vecindad sigue un principio muy parecido al de INTERCAMBIAR. La idea surge por analogía de las búsquedas locales 2-opt y 3-opt vistas en la teórica. En INTERCAMBIAR solo considerábamos como vecindad las soluciones que estaban a un *swap* de diferencia del mapeo fuente. En ese caso los *swapeos* pueden considerarse como una rotación de solo dos elementos. En esta variación la vecindad estará compuesta por todas las soluciones que están a lo sumo a una rotación de 3 nodos de diferencia.

Algorithm 10 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)
```

5.3. Complejidades

En este apartado se mantine la convención de que $n_1 = min\{n_1, n_2\}$. Para calcular las complejidades en peor caso de las búsquedas tengamos en cuenta lo siguiente: en todos los casos se tiene por un lado el costo para recorrer completamente cada vecindad, y por el otro la cantidad de veces que vamos a tener que efectivamente vamos a tener que recorrer una vecindad. Esto último es debido a que, en peor caso, por cada mejora que logramos tenemos que recorrer una vecindad nueva. Calcular el costo de realizar cada recorrido es tan simple como ver los algoritmos de la subsección anterior y usar álgebra de órdenes. En cambio encontrar una cota no demasiado holgada para la cantidad de veces que se mejora no resulta trivial.

Por tales motivos primero desdoblaremos el cálculo del costo de cada iteración del de la cantidad de iteraciones.

5.3.1. INTERCAMBIAR

$O(n_1^4)$

5.3.2. REMPLAZAR

$O((n_2 - n_1)n_1^3)$

5.3.3. 3-ROTACION

$O(n_1^5)$

5.3.4. Cantidad de iteraciones

Para hallar una cota en peor caso para la cantidad de iteraciones (que vale para las tres vecindades) pensemos en la siguiente situación: la solución *source* solo tiene una arista; además con cada iteración siempre mejoramos en solo una arista. Es claro que este es el peor escenario posible. También es relativamente fácil darse cuenta que la cantidad de iteraciones es $O(m_1)$, pues una solución máxima no puede tener más de m_1 aristas.

Es importante destacar que podría pasar que G_2 tenga menos aristas pero que las mismas usen más vértices de los que tiene G_1. Además si $m_2 < m_1$, $O(m_2) \subset O(m_1)$. Por eso está bien que sea $O(m_1)$ y no $O(min\{m_1, m_2\})$.

En definitiva esta cota podría no ser alcanzable (nosotros no encontramos una instancia que la realice), pero seguro acota la cantidad de iteraciones y no pudimos encontrar una cota mejor en peor caso.

5.4. Análisis de performance y calidad

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 11 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

Antes de empezar con la explicación, debemos comentar que los tests de calidad y tiempos del tabú search son muy intensos computacionalmente: debían ser corridos por varias horas para que terminen.

Lo que presentamos es el resultado final, que aunque es el resultado de una sola corrida de varias horas, necesito algunas decenas de intentos anteriores, con el objetivo de encontrar los parámetros más interesantes a ser mostrados en el informe.

Por esa razón debimos limitar la cantidad de muestras que tomamos, lo cual, como vermos impactará un poco en las conclusiones que podremos obtener. Sin embargo, los resultados son muy interesantes, y en el ejercicio 7 mostraremos algunos resultados que complementarán lo que no pudimos ver aquí.

Para testear, utilizamos los casos que pueden encontrarse en el apéndice (ACLARAR). En los gráficos, se verá que en el eje horizontal aparecen numeros de la pinta n/m. n será el criterio de parada utilizado, llamaremos 0 al criterio que cuenta la cantidad de iteraciones sin mejorar (el límite elegido será 500) y 1 al que harcodea el total de iteraciones (el límite elegido será 2000). m será el largo de la lista tabú.

Una aclaración importante es que probamos con distintos límites de iteraciones para cada criterio de parada y la diferencia en tiempo era la obvia (escalaba linealmente) y la diferencia en calidad era casi nula (a menos que las iteraciones fueran menores a 100. Por esa razón elegimos esas cantidades de iteraciones que nos parecieron lo suficientemente razonables.

No incluimos los gráficos de esos experimentos porque son mas que nada preliminares y no contribuyen a comparar las variantes, si no simplemente a optimizar los parámetros de cada una de ellas.

Primero analicemos la "calidad" de los algoritmos, es decir, que tan buenos son los resultados que devuelven. La métrica que utilizaremos es las aristas extra que devuelven, es decir, la cantidad de aristas que le agregan a la solución fuente, que es la de la heurística golosa.

Esta comparación será relativa: compararemos variantes del algoritmo unas contra

otras, sin saber cual es la solución optima. Compararemos las soluciones del tabu search contra soluciones optimas en el ejercicio 7, aquí sólo nos atañe comparar las diferentes variantes de nuestro algoritmo.

Como dijimos anteriormente, la intensidad computacional de los algoritmos no permitió que podamos testear con grafos extremadamente grandes, utilizamos grafos de tamaño 150 como máximo. Sin embargo, 150 es un tamaño suficientemente grande, dado que ya a partir de 500, simplemente una corrida de un algoritmo comienza a tardar en el orden de 15 minutos: ni mencionar lo que se demora un test entero.

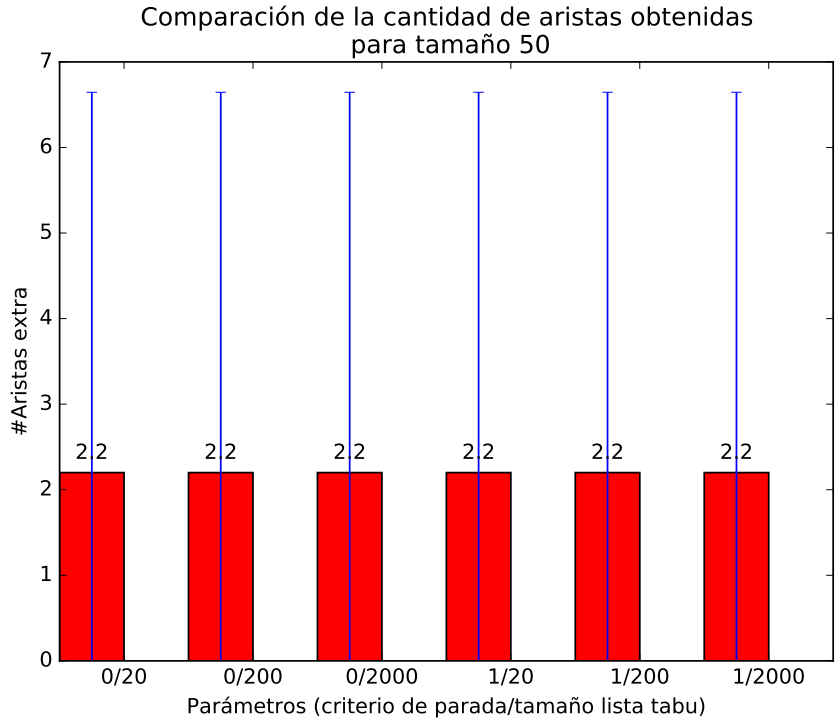


Figura 9

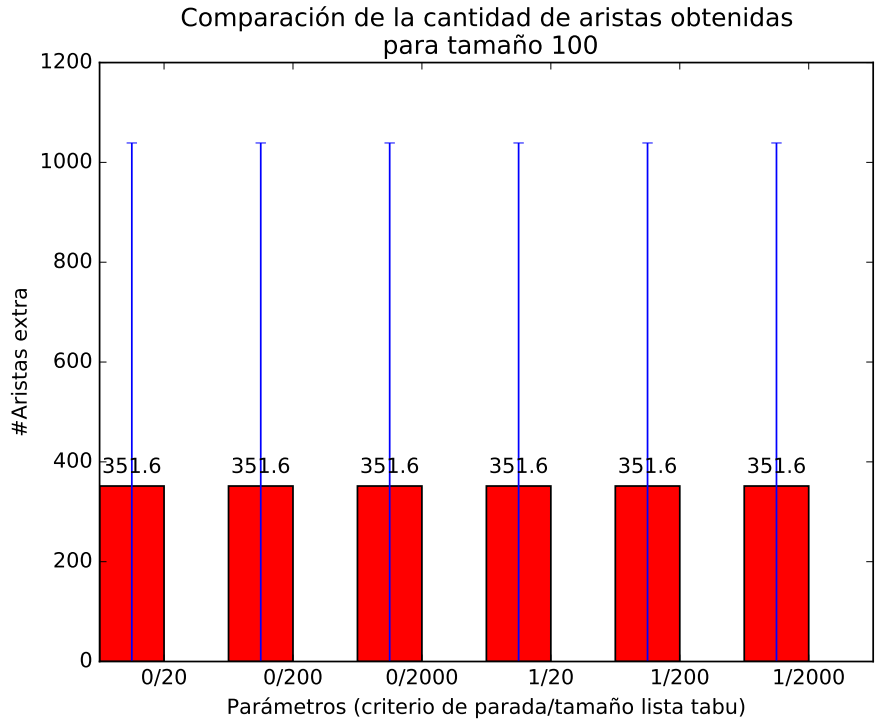


Figura 10

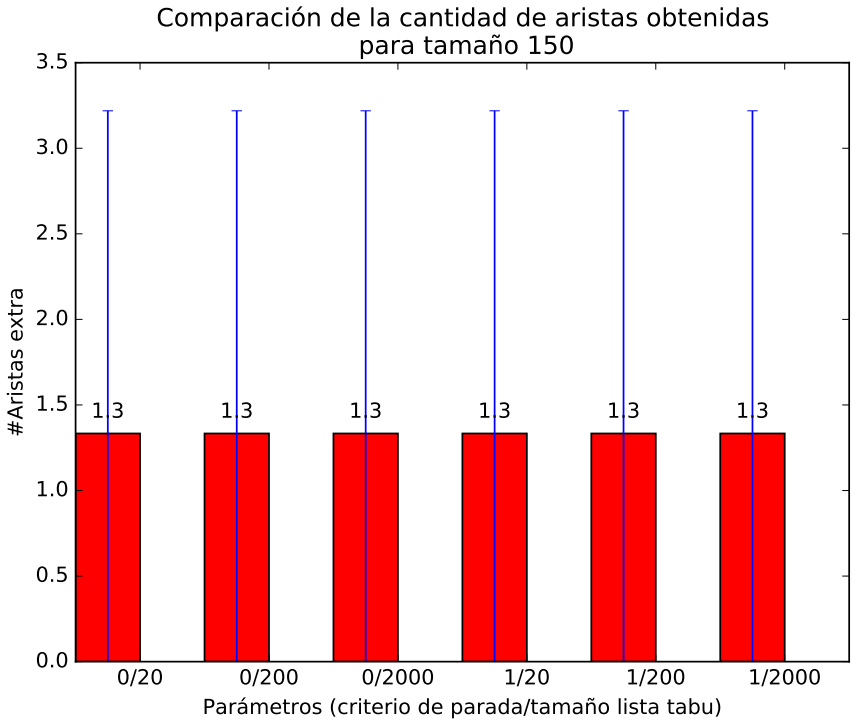


Figura 11

En los gráficos se marca el promedio con una barra y la desviación estándar con un

segmento.

Como podemos observar, no hay diferencia en el promedio de calidad de las mediciones. Sin embargo, pudimos observar algunas diferencias en los máximos y los mínimos. Los mínimos de la variante que cuenta cuantas iteraciones no se mejoró eran mas altos.

Esto se debe a que el tamaño de los grafos no es lo suficientemente grande como para que dos algoritmos buenos hagan diferencia. Sin embargo, los grafos son lo suficientemente grandes como para asegurarnos que la diferencia entre las variantes no será demasiada en ningún caso (salvo casos patológicos).

A continuación, analizaremos los tiempos de los algoritmos. Nuestra expectativa, es que el tiempo que tarda la variante 0, es decir, la que fija la cantidad de iteraciones máxima sin que se mejore la solucion, va a ser mejor dado que optimiza la cantidad de iteraciones .ºciosas", sobre todo en grafos chicos, donde la búsqueda se estanca relativamente rápido.

Además, esperamos que las variantes que tienen una lista tabú más larga tardan más, debido a que usamos una lista enlazada (std::list) para representarla, por lo que la búsqueda dentro de ella tiene complejidad lineal.

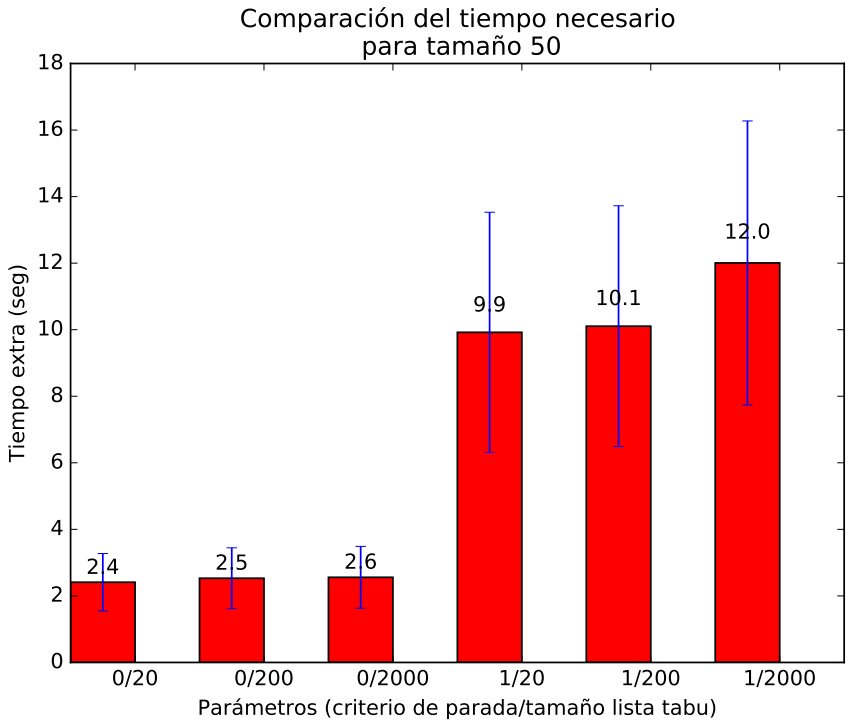


Figura 12

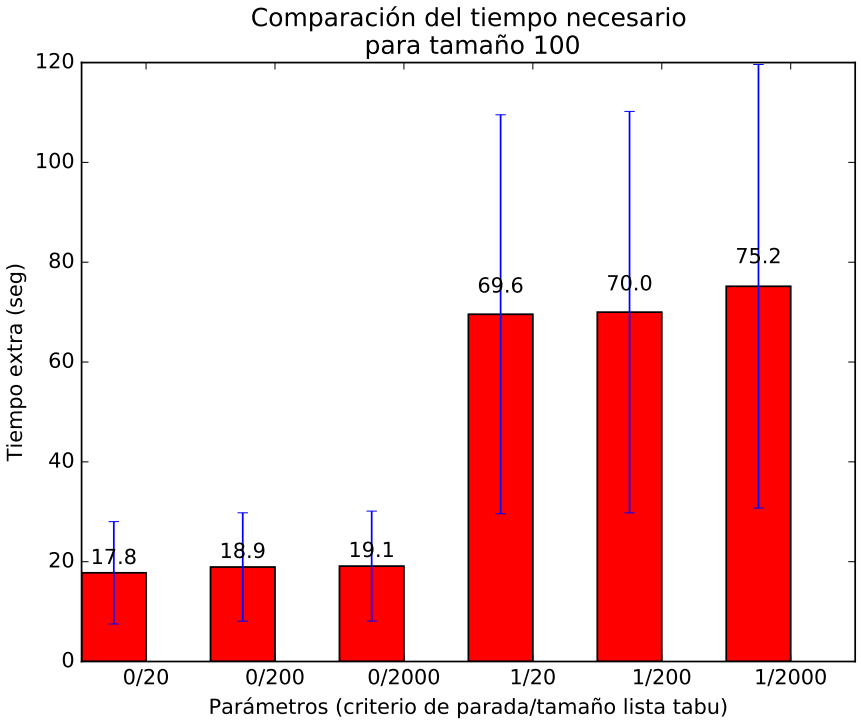


Figura 13

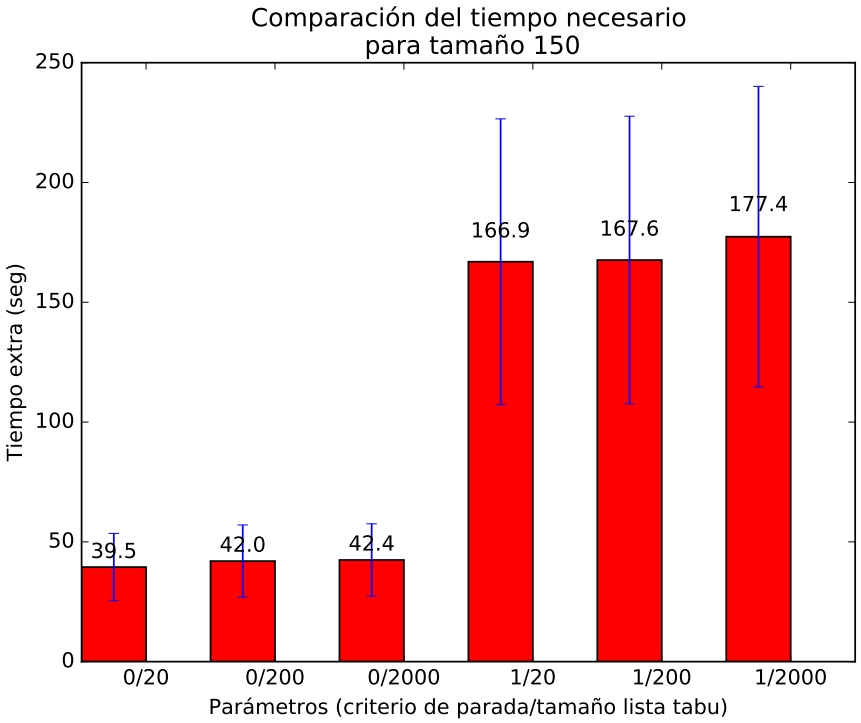


Figura 14

En los gráficos se observa que confirmamos nuestras teorias de una manera bastante

sólida.

Esto nos permite concluir que, aunque la calidad de las variantes es realmente similar, el tiempo que tardan los algoritmos es distinto, lo que nos permite concluir que la relación costo calidad de los algoritmos sí es diferente..

Esta relación costo calidad puede verse en los gráficos siguientes. En estos gráficos, dividimos la cantidad de aristas extra, sobre la el tiempo que tarda el algoritmo. Es decir, a más alto, mejor es el algoritmo.

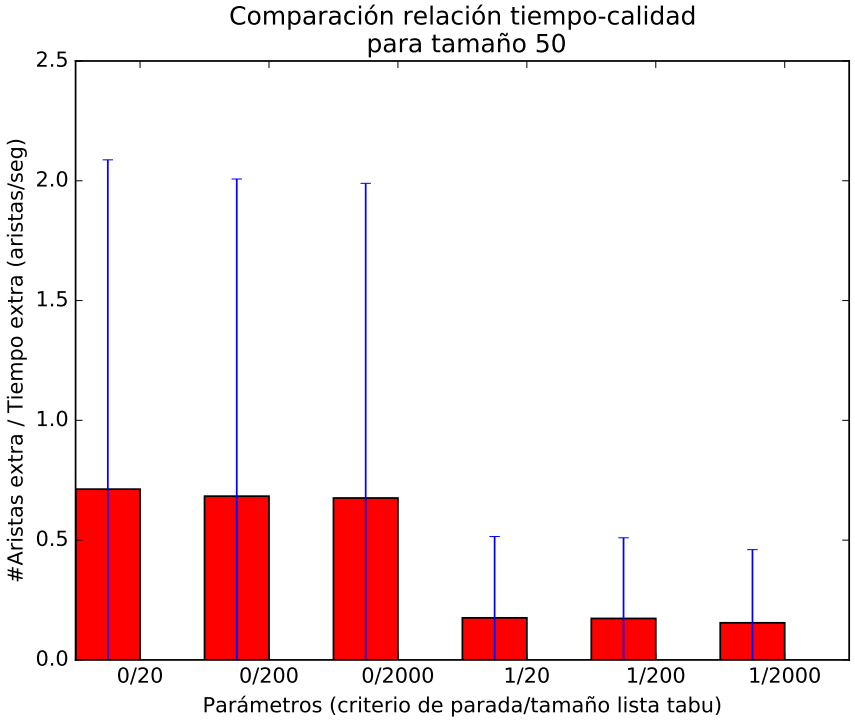


Figura 15

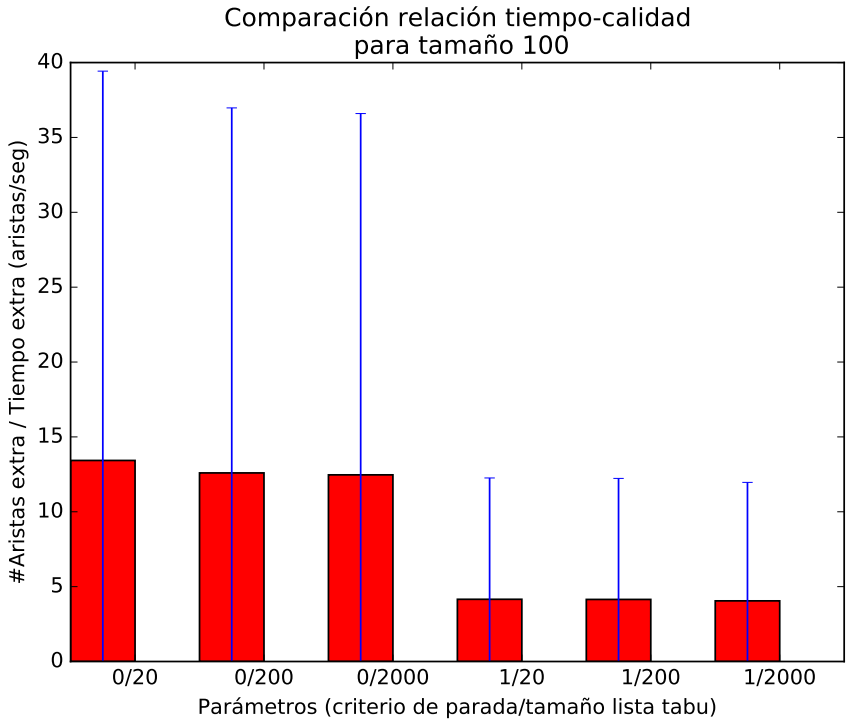


Figura 16

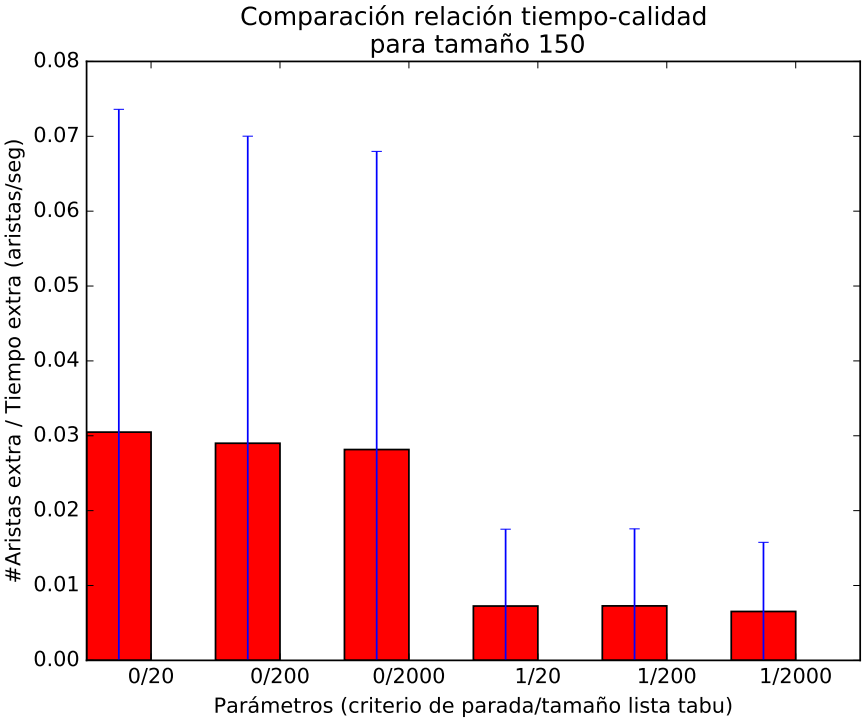


Figura 17

7. Ejercicio 7

A. Apéndice

A.1. Generación de grafos conexos aleatorios

Algorithm 12 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 ciertas partes del informe al explicar nuestras implementaciones mencionamos algunas funciones secundarias sin profundizar demasiado en como estaban definas para no volver demasiado extensa o densa la explicación. Dejamos aquí el pseudocódigo de algunos de estos métodos para evacuar posibles dudas.

Explicaremos a continuación, entre otras cosas, nuestros algoritmos de lectura y escritura, es importante aclarar que estos no fueron tomados en cuenta para los cálculos de complejidad ya que las mediciones de I/O son muy variables y suelen producir gráficos muy ruidosos. A demás, 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 13 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 14 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 15 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.