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

procedure BT(`Grafo` $g1$, `Grafo` $g2$, `vector<int>` $vertices1$, `vector<int>` $vertices2$, `Isomorfismo` iso)
    if $solucion.aristas == MIN(g1.m, g2.m)$ **then** ▷ $O(1)$
        return
    if $!ordenado\_asc(iso)$ **then** ▷ $O(n_1)$
        return
    if $iso.size == g1.n$ **then** ▷ $O(1)$
        $aristas \leftarrow contar\_aristas\_isomorfismo(g1, g2, iso)$ ▷ $O(n_1^2)$
        if $aristas > solucion.aristas$ **then** ▷ $O(1)$
            $solucion.isomorfismo \leftarrow iso$ ▷ $O(n_1)$
            $solucion.aristas \leftarrow aristas$ ▷ $O(1)$
        return
    for $u \in vertices1$ **do** ▷ v_1 veces $O(1)$
        for $v \in vertices2$ **do** ▷ $v_1 v_2$ veces $O(1)$
            $nuevo\_iso = iso$ ▷ $v_1 v_2$ veces $O(n_1)$
            $nuevo\_iso.push\_back(make\_pair(u, v))$ ▷ $v_1 v_2$ veces $O(1)$
            $bt(g1, g2, copiar\_sin(vertices1, u), copiar\_sin(vertices2, v), nuevo\_iso)$ ▷ $v_1 v_2$ veces $T(n_1, n_2, v_1 - 1, v_2 - 1)$

Algorithm 2 Pseudocódigo del procedimiento contar aristas isomorfismo

procedure CONTAR_ARISTAS_ISOMORFISMO(`Grafo` $g1$, `Grafo` $g2$, `Isomorfismo` iso) $\rightarrow$ `Int`
    $aristas \leftarrow 0$ ▷ $O(1)$
    for $p \in iso$ **do** ▷ n_1 veces $O(1)$
        $vg1 = p.first$ ▷ n_1 veces $O(1)$
        $vg2 = p.second$ ▷ n_1 veces $O(1)$
        for $q \in iso$ **do** ▷ n_1 veces $O(1)$
            $ug1 = q.first$ ▷ n_1^2 veces $O(1)$
            $ug2 = q.second$ ▷ n_1^2 veces $O(1)$
            if $g1.adj\_matrix[vg1][ug1] \wedge g2.adj\_matrix[vg2][ug2]$ **then** ▷ n_1^2 veces $O(1)$
                $aristas{+}{+}$ ▷ 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,

$$\begin{aligned}
T(v_1, v_2) &< 4n_1^2n_2 + v_1v_2T(v_1 - 1, v_2 - 1) \\
&< 4n_1^2n_2 + v_1v_2(4n_1^2n_2 + (v_1 - 1)(v_2 - 1)T(v_1 - 2, v_2 - 2)) \\
&= 4n_1^2n_2 + v_1v_24n_1^2n_2 + v_1v_2(v_1 - 1)(v_2 - 1)T(v_1 - 2, v_2 - 2) \\
&< 4n_1^2n_2 + 4n_1^3n_2^2 + v_1v_2(v_1 - 1)(v_2 - 1)T(v_1 - 2, v_2 - 2) \\
&< ... \\
&< 4n_1^2n_2 + ... + 4n_1^{v_1}n_2^{v_1-1} + v_1(v_1 - 1)...(v_1 - v_1 + 1)v_2(v_2 - 1)...(v_2 - v_1 + 1)T(0, v_2 - v_1) \\
&< 4n_1^2n_2 + 4n_1^3n_2^2 + ... + 4n_1^{v_1}n_2^{v_1-1} + 4n_1^{v_1+1}n_2^{v_1} \\
&< 4n_1^2n_2 + 4n_1^3n_2^2 + ... + 4n_1^{v_1+1}n_2^{v_1} \\
&< 4^{v_1+1}n_1^{v_1+1}n_2^{v_1+1} \\
&= (4n_1n_2)^{v_1+1}
\end{aligned}$$

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

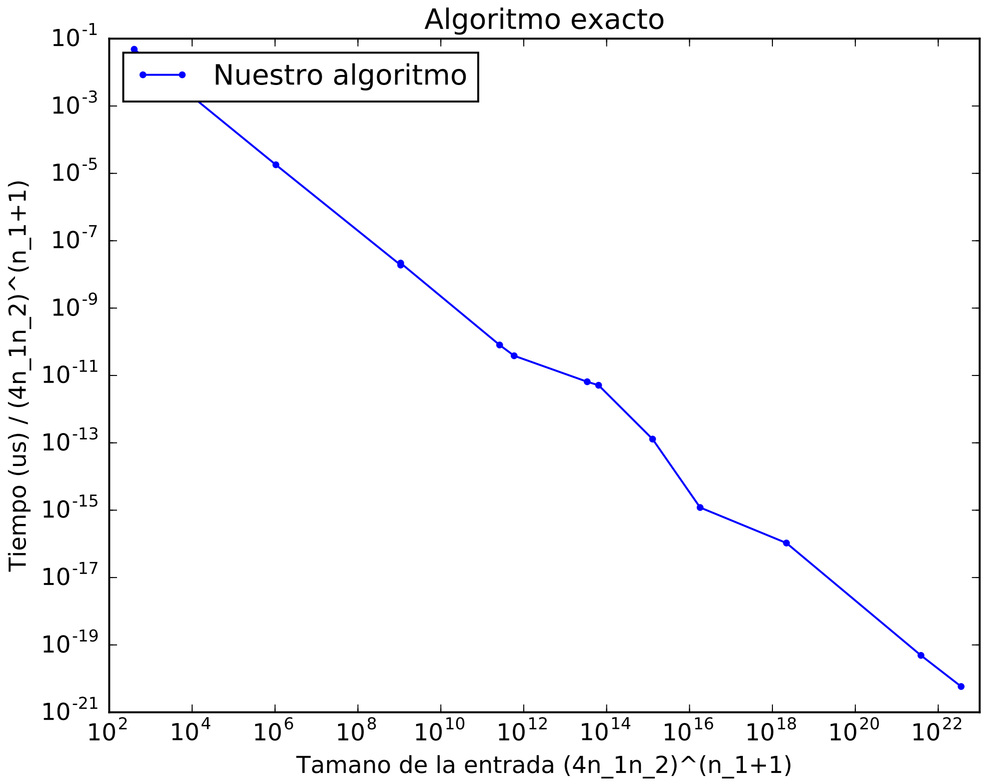


Figura 1

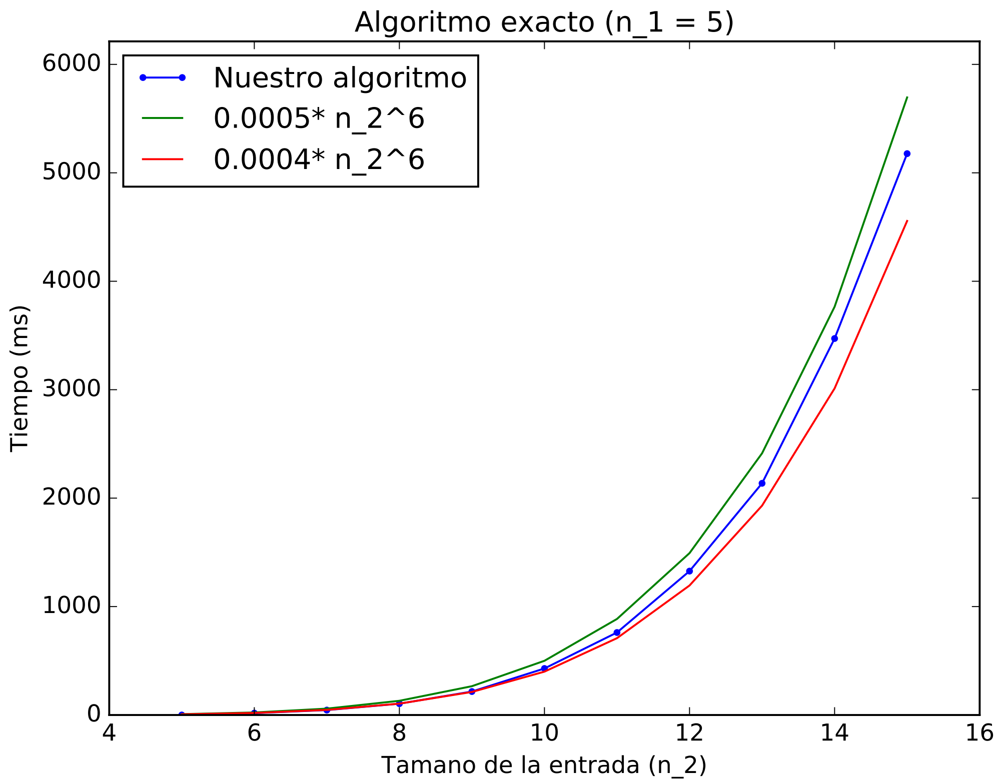

Algoritmo exacto (n_1 = 5)
Nuestro algoritmo
0.0005* n_2^6
0.0004* n_2^6
Tiempo (ms)
Tamano de la entrada (n_2)
0
1000
2000
3000
4000
5000
6000
4
6
8
10
12
14
16

Figura 2

3. Ejercicio 3

4. Ejercicio 4: Algoritmo Goloso

4.1. Explicación detallada del algoritmo

Como vimos en el ejercicio 2 este problema no parece poder ser resuelto en tiempo polinomial, para poder hacercarnos a una solución en tiempos razonables sacrificaremos la seguridad de conseguir siempre la opción óptima a cambio de mejorar la complejidad. Esto se hará a partir de la implementación de una heuristica golosa, un programa que a partir de ciertas suposiciones y reglas, no necesariamente validas siempre pero muchas veces útiles, nos permiten tomar desiciones rapidamente. Será golosa porque tomará desiciones en base a mejorar su estado actual sin pensar a la larga.

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

procedure GOLOSO1(`Grafo` $g1$, `Grafo` $g2$, `set<int>` $vertices1$, `set<int>` $vertices2$) $\rightarrow$ `MCS`
    $ordenar\_por\_grado(vertices1, g1)$ ▷ $O(n_1^2)$
    $ordenar\_por\_grado(vertices2, g2)$ ▷ $O(n_2^2)$
    `MCS` $solucion$ ▷ $O(1)$
    for `int` $i = 0$, $i < vertices1.tamanio()$, $i++$ **do**
        $solucion.isomorfismo.insertar\_atras(< vertices1[i], vertices2[i] >)$ ▷ n_1 veces $O(1)$
    `int` $aristas$
    $aristas = contar\_aristas\_isomorfismo(g1, g2, u, v, solucion.isomorfismo)$ ▷ $O(n_1^2)$
    return $solucion$

La segunda heuristica tiene una mayor complejidad temporal pero da soluciones de calidad superior. 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 que maximice la cantidad de aristas del isomorfismo. Si hay empates se queda con la primera.

El pseudocódigo es el siguiente

Algorithm 4 Pseudocódigo de la heurística golosa

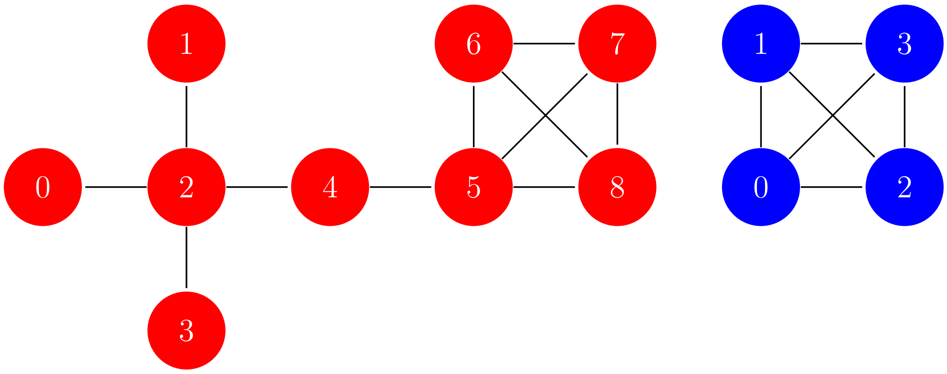
```
procedure GOLOSO(Grafo g1, Grafo g2, vector<int> vertices1, vector<int> vertices2) → MCS
    MCS solucion                                                        ▷ O(1)
    solucion.aristas = 0
    int vertice1 = mayor_adj(vertices1, g1)                             ▷ O(n_1)
    int vertice2 = mayor_adj(vertices2, g2)                             ▷ O(n_2)
    solucion.isomorfismo.insertar_atras(< vertice1, vertice2 >)         ▷ O(1)
    vertices1.borrar(vertice1)                                          ▷ O(log(n_1))
    vertices2.borrar(vertice2)                                          ▷ O(log(n_2))
    while vertices1.tamanio() ≠ 0 do
        par<int,int> par_mayor_deg =< vertices1.primero(), vertices2.primero() >   ▷ n_1 veces O(1)
        for u ∈ vertices1 do
            for v ∈ vertices2 do
                int aristas                                             ▷ n_1^2 n_2 veces O(1)
                aristas = contar_aristas_isomorfismo(g1, g2, u, v, solucion.isomorfismo)   ▷ n_1^2 n_2 veces O(n_1^2)
                if aristas > solucion.aristas then                      ▷ n_1^2 n_2 veces O(1)
                    solucion.aristas = aristas                          ▷ n_1^2 n_2 veces O(1)
                    par_mayor_deg =< u, v >                             ▷ n_1^2 n_2 veces O(1)
        solucion.isomorfismo.insertar_atras(par_mayor_deg)              ▷ n_1 veces O(1)
        vertices1.borrar(par_mayor_deg.primero)                         ▷ n_1 veces O(log(n_1))
        vertices1.borrar(par_mayor_deg.segundo)                         ▷ n_1 veces O(log(n_2))
    return solucion
```

4.2. Complejidad temporal de peor caso

La complejidad del algoritmo es precisamente $O(n_1^4 n_2)$.

4.3. Instancias no óptimas

Un ejemplo donde la heuristica golosa versión 2 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.



Para estos casos la heuristica 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 sería G_n, o sea tendría $\frac{n.(n-1)}{2}$ aristas, por lo cual para n suficientemente grande tenemos una diferencia entre el óptimo y el resultado obtenido tan grande como queramos.

4.4. Performance del algoritmo

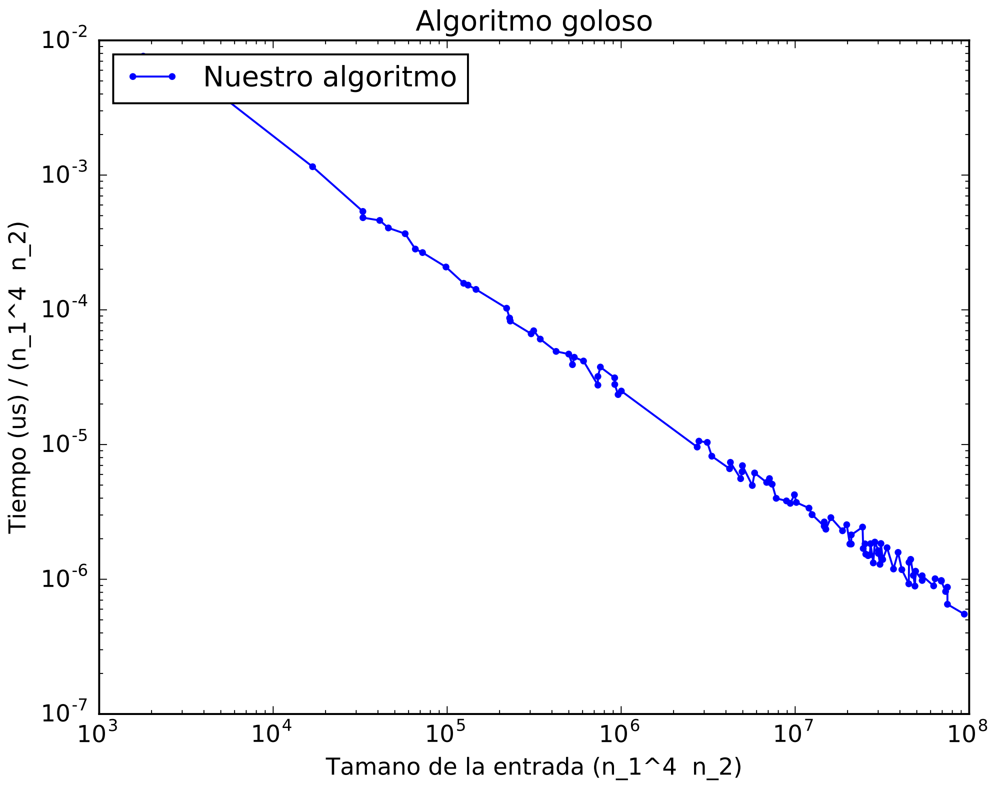


Figura 3

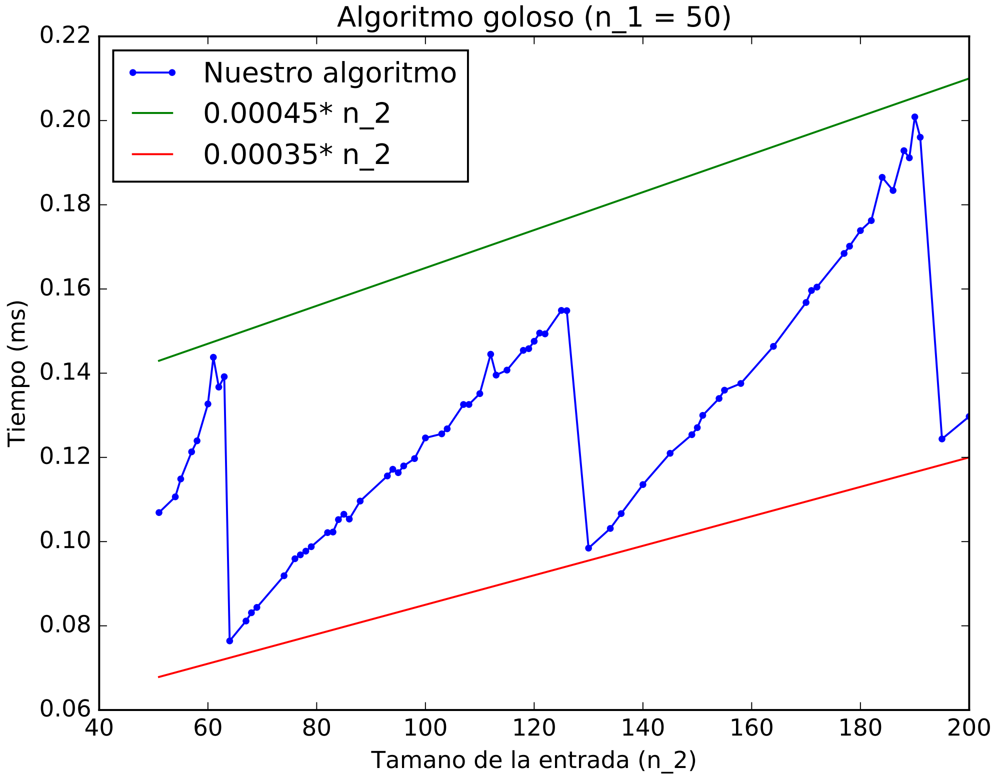

Algoritmo goloso (n_1 = 50)
Nuestro algoritmo
0.00045* n_2
0.00035* n_2
Tiempo (ms)
Tamano de la entrada (n_2)
0.22
0.20
0.18
0.16
0.14
0.12
0.10
0.08
0.06
40
60
80
100
120
140
160
180
200

Figura 4

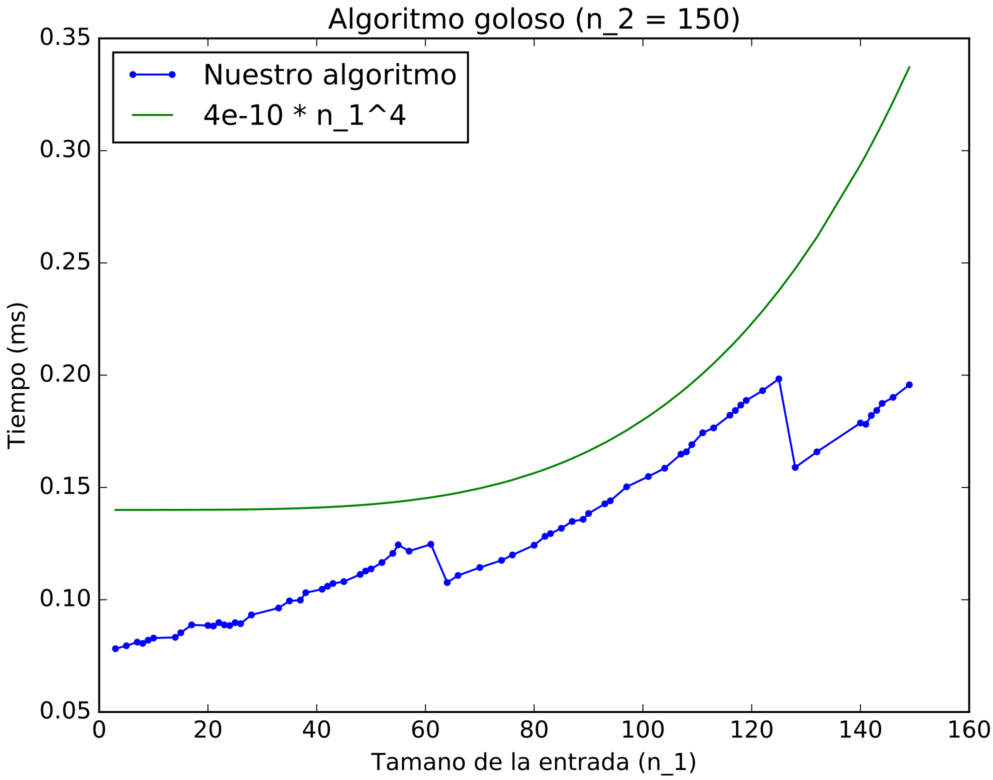

Algoritmo goloso (n_2 = 150)
Nuestro algoritmo
4e-10 * n_1^4
Tiempo (ms)
Tamano de la entrada (n_1)
0.35
0.30
0.25
0.20
0.15
0.10
0.05
0
20
40
60
80
100
120
140
160

Figura 5

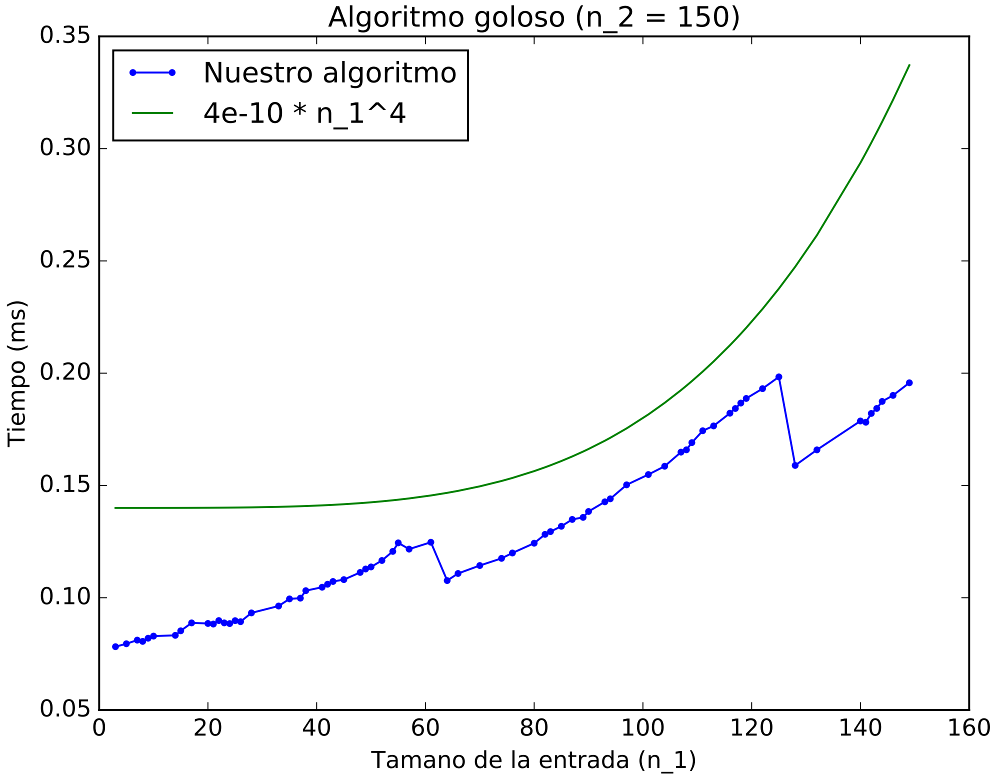

Algoritmo goloso (n_2 = 150)
Nuestro algoritmo
4e-10 * n_1^4
Tiempo (ms)
Tamano de la entrada (n_1)
0.35
0.30
0.25
0.20
0.15
0.10
0.05
0
20
40
60
80
100
120
140
160

Figura 6

5. Ejercicio 5

Algorithm 5 Pseudocódigo de INTERCAMBIAR

```
procedure INT(Grafo G1, set<int> vertices1, Grafo G2, set<int> vertices2) → MCS
    MCS source ← goloso(G1, vertices1, G2, vertices2)                          ▷ O()
    bool mejore ← true                                                         ▷ O(1)
    while mejore do                                                            ▷ O(min{m1, m2})
        mejore ← false                                                         ▷ O(1)
        for i ← 0 ... |source.isomorfismo| do                                  ▷ n1 veces
            for j ← 0 ... |source.isomorfismo|, i ≠ j do                       ▷ n1 veces
                swap(source.isomorfismo[i].first, source.isomorfismo[j].first) ▷ O(1)
                int aristas ← contar_aristas_isomorfismo(G1, G2, source.isomorfismo) ▷ O(n1^2)
                if aristas > source.aristas then                               ▷ O(1)
                    source.aristas ← aristas                                   ▷ O(1)
                    mejore ← true                                              ▷ O(1)
```

Algorithm 6 Pseudocódigo de REMPLAZAR

```
procedure REMP(Grafo G1, set<int> vertices1, Grafo G2, set<int> vertices2) → MCS
    MCS source ← goloso(G1, vertices1, G2, vertices2)
    bool mejore ← true                                                         ▷ O(1)
    vector<int> vertices ← set_to_vector(vertices2)                            ▷ O(n2 − n1)
    while mejore do                                                            ▷ O(min{m1, m2})
        mejore ← false                                                         ▷ O(1)
        for i ← 0 ... |vertices| do                                            ▷ n2 − n1 veces
            for j ← 0 ... |source.isomorfismo| do                              ▷ n1 veces
                swap(vertices[i], source.isomorfismo[j].second)                ▷ O(1)
                int aristas ← contar_aristas_isomorfismo(G1, G2, source.isomorfismo) ▷ O(n1^2)
                if aristas > source.aristas then                               ▷ O(1)
                    source.aristas ← aristas                                   ▷ O(1)
                    mejore ← true                                              ▷ O(1)
                else
                    swap(vertices[i], source.isomorfismo[j].second)            ▷ O(1)
```

Algorithm 7 Pseudocódigo de 3-ROTACION

```
procedure 3-ROT(Grafo G1, set<int> vertices1, Grafo G2, set<int> vertices2)→ MCS
    MCS source ← goloso(G1, vertices1, G2, vertices2)
    bool mejore ← true                                                        ▷ O(1)
    while mejore do                                                           ▷ O(min{m1, m2})
        mejore ← false                                                        ▷ O(1)
        for i ← 0 . . . |source.isomorfismo| do                               ▷ O(n1) veces
            for j ← 0 . . . |source.isomorfismo|, i ≠ j do                    ▷ n1 veces
                for k ← 0 . . . |source.isomorfismo| do                       ▷ n1 veces
                    swap(source.isomorfismo[i].first, source.isomorfismo[k].first)   ▷ O(1)
                    swap(source.isomorfismo[k].first, source.isomorfismo[j].first)   ▷ O(1)
                    int aristas ← contar_aristas_isomorfismo(G1, G2, source.isomorfismo)   ▷ O(n1^2)
                    if aristas > source.aristas then                          ▷ O(1)
                        source.aristas ← aristas                              ▷ O(1)
                        source.isomorfismo ← source.isomorfismo               ▷ O(n1)
                        mejore ← true                                         ▷ O(1)
```

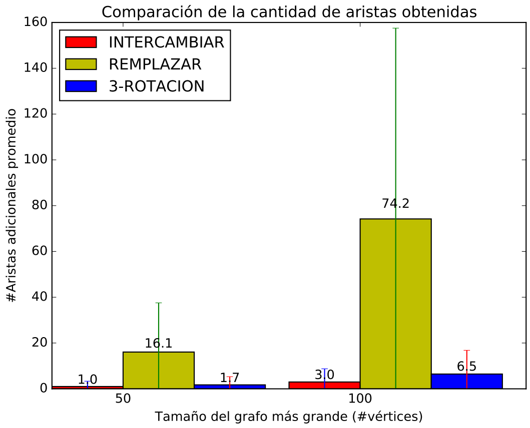


Figura 7:

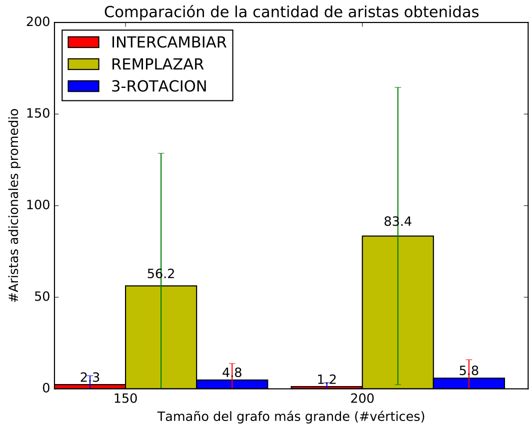


Figura 8:

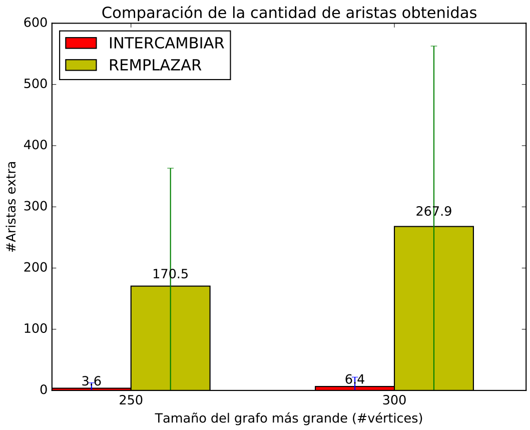


Figura 9:

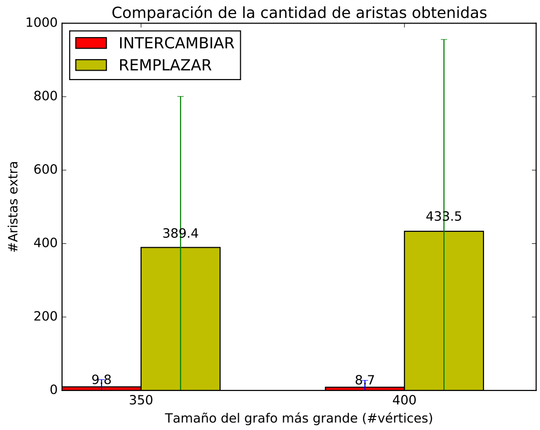


Figura 10:

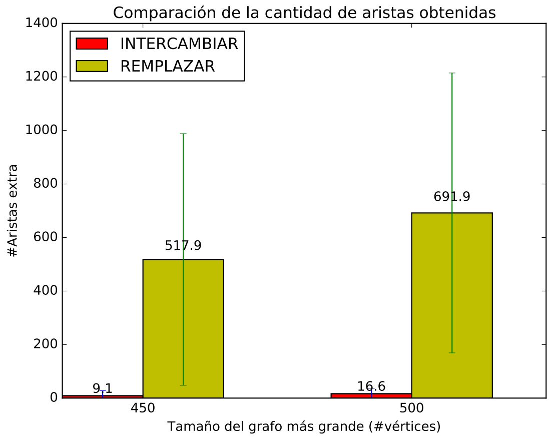


Figura 11:

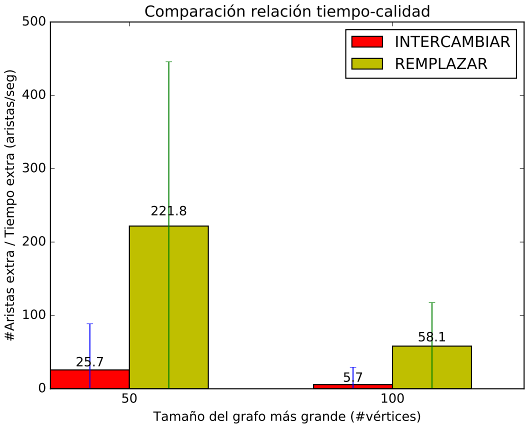


Figura 12:

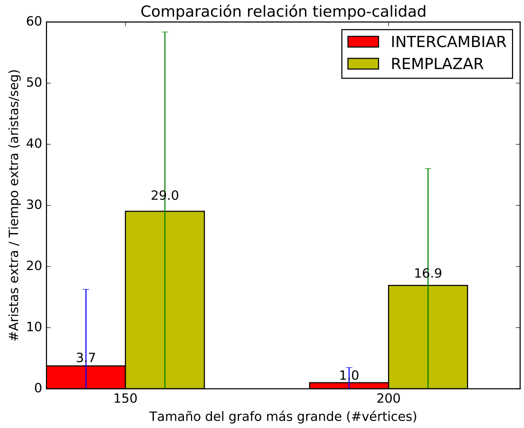


Figura 13:

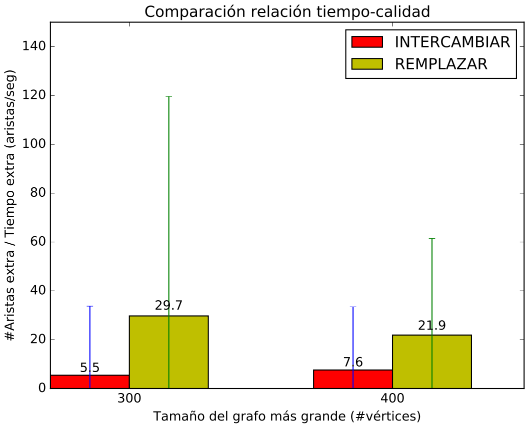


Figura 14:

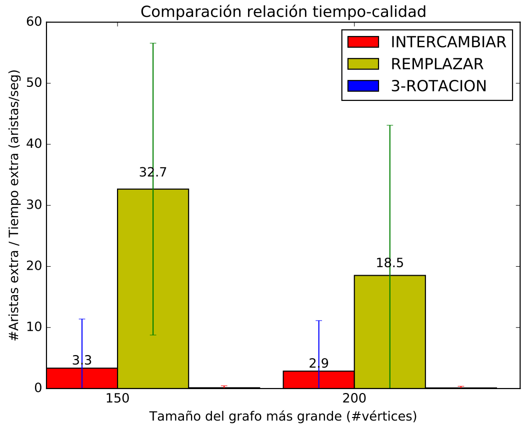


Figura 15:

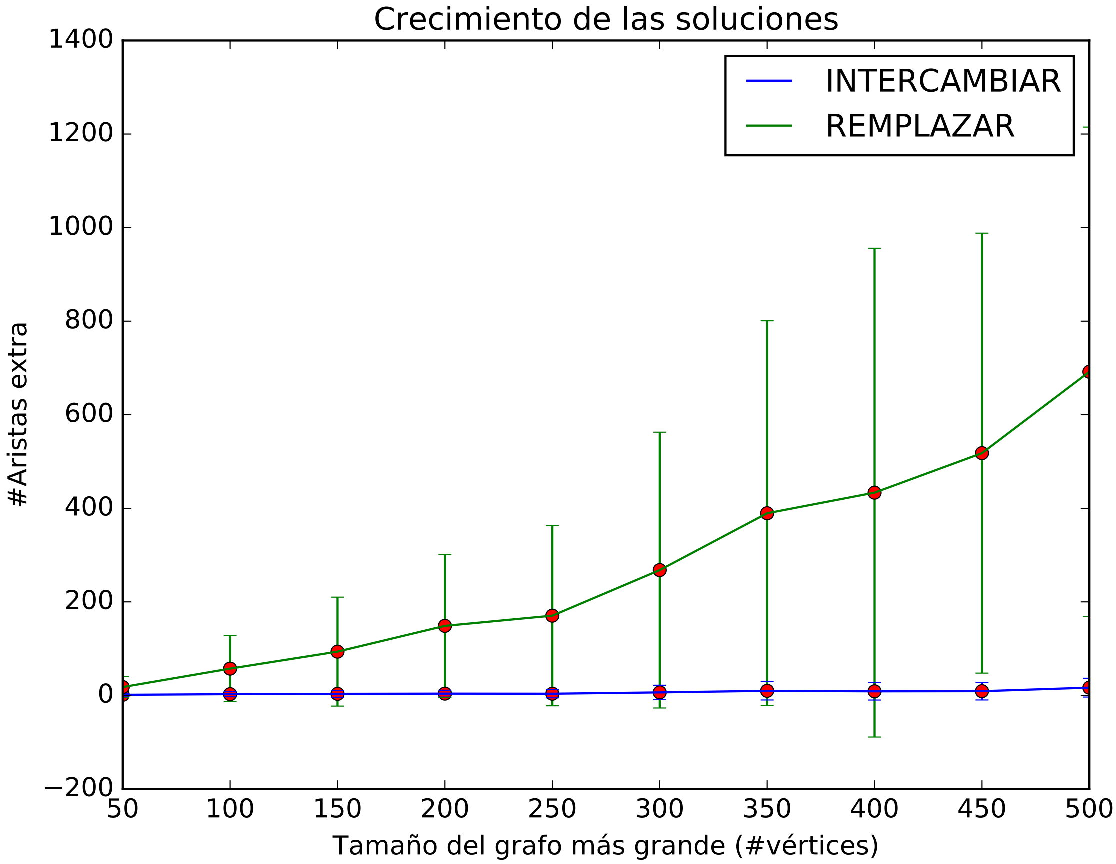

Crecimiento de las soluciones
INTERCAMBIAR
REMPLAZAR
#Aristas extra
Tamaño del grafo más grande (#vértices)
1400
1200
1000
800
600
400
200
0
−200
50
100
150
200
250
300
350
400
450
500

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

```
procedure TABU_SEARCH(Grafo g1, vector<int> vertices1, Grafo g2, vector<int> vertices2)
    source ← goloso(g1, vertices1, g2, vertices2)
    lista_tabu ← lista(1000, make_pair(0, 0))
    indice_lista_tabu ← 0
    Inicializar estructuras relacionadas con el criterio de parada.
    while criterio de parada do
        mejor_tabu ← {.isomorfismo = Isomorfismo(), .aristas = 0}
        mejor_solucion ← {.isomorfismo = Isomorfismo(), .aristas = 0}
        for dovecino ∈ vecindad(source)
            aristas ← buscar(lista_t abu, hash(vecino))
            if aristas = 0 then                              ▷ Source es una solución tabu.
                if aristas > mejor_tabu.aristas then
                    mejor_tabu.aristas ← aristas
                    mejor_tabu.isomorfismo ← vecino
                    continue
            else
                aristas = contar_aristas_isomorfismo(g1, g2, vecino)
                if aristas > mejor_tabu.aristas then
                    mejor_solucion.aristas ← aristas
                    mejor_solucion.isomorfismo ← vecino
                    continue
        if mejor_solucion.aristas > 0 then
            lista_tabu.push_back(
             make_pair(hash(mejor_solucion.isomorfismo), mejor_solucion.aristas))
            if lista_tabu.size() > lista_tabu_limite then
                lista_tabu.pop_front()
        if mejor_solucion.aristas = 0 then                   ▷ Todas las soluciones son tabu.
            source ← mejor_tabu
        else
            source ← mejor_solucion
        Actualizar estructuras relacionadas con el criterio de parada.
```

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 conexos al azar

```
procedure GRAFO_RANDOM(int n, int m) → Grafo
    k_n ← {(0,1), (0,2), ..., (0,n), (1,2), (1,3), ..., (n−2, n−1)}
    vertices ← {random.range(0,n)}                    ▷ Empiezo con un vértice al azar
    agm ← {}
    while vertices.size() < n do
        aristas ← "aristas (u,v) de k_n tal que u ∈ vertices y v ∉ vertices o viceversa"
        arista_nueva ← random.choice(aristas)
        agm.add(arista_nueva)
        k_n.remove(arista_nueva)
        vertices.add("extremo de arista_nueva que no estaba en vertices")
                ▷ Cuando termina este ciclo tenemos un árbol de n vertices y n − 1 aristas
    grafo ← agm
    while grafo.size() < m do
        arista ← random.choice(k_n)
        grafo.add(arista)
        k_n.remove(arista)
    for arista ∈ grafo do
        peso(arista) ← random.random()
    return grafo
```

El algoritmo, se basa en generar un grafo conexo minimal (es decir, un árbol) de n vértices. Para lograr esto, técnicamente lo que hacemos es empezar con K_n, es decir, el grafo completo de n vértices, con todos sus aristas de igual peso, y le encontramos un árbol generador mínimo utilizando Prim. Todo esto es obviamente trivial en este caso, dado que todas las aristas tienen igual peso, así que básicamente lo que hacemos es elegir una arista al azar en cada paso.

Luego, una vez que tenemos el árbol terminado, lo completamos con aristas al azar, hasta llegar al objetivo de m aristas.

Finalmente, se eligen pesos al azar para cada arista.

A.2. Partes relevantes del código

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.