

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

## Algoritmos y Estructuras de datos III

Segundo Cuatrimestre de 2011

### Trabajo Práctico 3

Cartero Chino en grafo mixto.

**Grupo: '1'**

Integrante	LU	Correo electrónico
Cammi, Martín	676/02	<a href="mailto:martincammi@gmail.com">martincammi@gmail.com</a>
Garbi, Sebastián	179/05	<a href="mailto:garbyseba@gmail.com">garbyseba@gmail.com</a>
Kretschmayer, Daniel	310/99	<a href="mailto:daniak@gmail.com">daniak@gmail.com</a>

# Índice

0.1. Lenguaje utilizado . . . . .	3
0.2. Como ejecutar el TP . . . . .	3
<b>1. Introducción</b>	<b>5</b>
<b>2. Situaciones de la vida real</b>	<b>6</b>
<b>3. Heurística Constructiva</b>	<b>7</b>
<b>4. Heurística de búsqueda Local</b>	<b>9</b>
<b>5. Metaheurística Grasp</b>	<b>11</b>
5.1. Gráficos de variación de ALFA . . . . .	12
5.2. Gráficos de cantidad de iteraciones . . . . .	13
<b>6. Pseudo-algoritmo</b>	<b>14</b>
<b>7. Complejidad</b>	<b>15</b>

## Ejecución del TP

### 0.1. Lenguaje utilizado

El lenguaje utilizado para el trabajo práctico ha sido *Java*, compilado con la versión 1.5 de la Virtual Machine.

El trabajo se acompaña con los fuentes de la solución que puede importarse en IDE de Eclipse o ejecutarse desde línea de comandos.

### 0.2. Como ejecutar el TP

#### Desde línea de comandos

- Posicionarse en el directorio Algo3Tp3
- Copiar allí el archivo de entrada para el problema i, por ejemplo Ej.in
- Ejecutar el comando: `java -cp ./bin problema1.Ej`

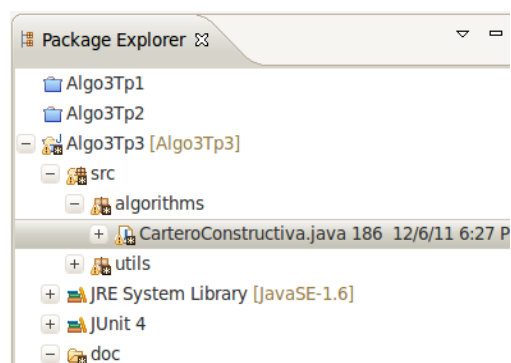
Esto generará el archivo Ej.out con la solución en el mismo directorio Algo3Tp3.

#### Desde el Eclipse

Primero importaremos el proyecto:

- Seleccionar File ⇒ Import.
- Seleccionar General ⇒ Existing Projects into Workspace ⇒ Next.
- Seleccionar el directorio llamado Algo3Tp3.
- Finish.

Desde la vista de **Package Explorer** bajo el paquete **src** aparecerá el paquete *algorithms* y dentro del el archivo de java:



Para ejecutar un problema:

- Posicionarse en el directorio Algo3Tp3
- Copiar en el directorio Algo3Tp3 el archivo de entrada para el problema i, por ejemplo Ej.in
- Con botón derecho Run As  $\Rightarrow$  Java Application. Se ejecutará el problema seleccionado.

Esto generará el archivo Ej.out con la solución en el mismo directorio Algo3Tp3.



## 2. Situaciones de la vida real

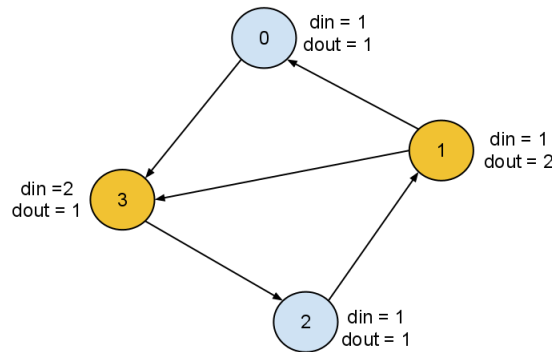
El algoritmo del cartero chino tiene múltiples aplicaciones en la vida cotidiana, como por ejemplo:

- *Camiones de basura:* en donde los camiones deban recorrer las calles de una ciudad y volver al centro de tratamiento de basura. Las calles estarían representadas por aristas y las intersecciones por vértices. Las calles podrían bien tener diferentes sentidos o sentidos únicos, y los pesos podrían corresponder a la cantidad de basura que deben recolectar en ciertas zonas (ej zonas industriales).
- *Reparto de Volantes:* Cálculo de ruta óptima que pase por todas las calles de una ciudad al menos una vez para poder repartir volantes de publicidad. Las calles estarían representadas por aristas y las intersecciones por vértices. Podría asignarse un peso de 1 a cada arista siendo que no tiene mayor complejidad repartir volantes en una u otra cuadra.

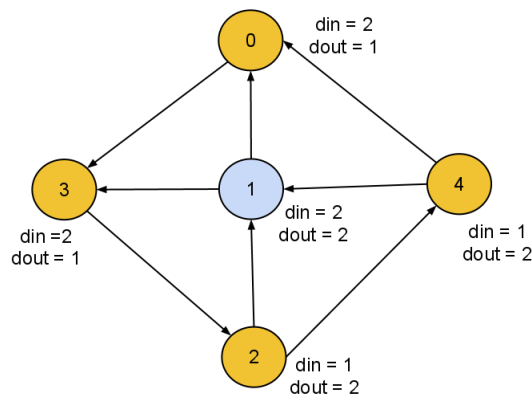
### 3. Heurística Constructiva

El *algoritmo constructivo* se encargará de tomar un grafo mixto y orientar sus aristas no orientadas y al mismo tiempo establecer un matching inicial.

Un matching consiste en agrupar pares de nodos  $(v, w)$  tales que  $d_{in}(v) > d_{out}(v)$  y  $d_{out}(w) > d_{in}(w)$ .



En el grafo, los nodos 1 y 3 están desbalanceados con respecto a  $d_{in}$  y  $d_{out}$ , más aún  $d_{in}3 > d_{out}3$  y  $d_{out}1 > d_{in}1$  con lo cual pueden formar parte de un matching:  $[(3,1)]$



En el grafo, los nodos 0, 2, 3 y 4 están desbalanceados con respecto a  $d_{in}$  y  $d_{out}$ , un matching posible para equiparar sus  $d_{in}$  y  $d_{out}$  sería:  $[(0,2)(3,4)]$

Para ello ideamos diferentes métodos para la orientación de todas las aristas y fuimos experimentando con 5 métodos distintos, todos ellos basándose en los grados de los nodos para la elección de un nodo con el cual empezar a orientar sus aristas. Los métodos elegir:

- 1) Los nodos que tienen igual cantidad de aristas sin orientar que orientadas, y que la suma de sus grados de entrada y salida sea mayor o igual a  $ALFA * \text{nodoGradoMaximo}$ .
- 2) Los nodos a partir de un cierto número, que viene como parámetro, y pondrá como máximo  $ALFA * \text{nodoGradoMaximo}$  nodos.

- 3) Los nodos que tengan grado de nodo (sin orientar) mayor o igual al  $ALFA * nodoGradoMaximo$ .
- 4) Los nodos que tengan grado de entrada (orientado) mayor o igual al  $ALFA * nodoGradoMaximo$ .
- 5) Los nodos que tengan grado de salida (orientado) mayor o igual al  $ALFA * nodoGradoMaximo$ .

Se entiende como *nodoGradoMaximo* al nodo de mayor grado, contando aristas no orientadas.

Con cualquiera de estas posibilidades, siempre se intenta hacer un balance entre los grados que quedarán (solo quedarán grados de entrada y salida, ya que será completamente orientado). Realizando pequeñas pruebas terminamos optando por el método número 3) ya que por ejemplo con el método número 1) no podíamos asegurar que siempre existieran tales nodos que cumplieran la condición.

El segundo método se basa en encontrar los primeros  $ALFA * nodoGradoMaximo$  nodos, pero nos estaba dando nodos que podía tener grados menores que  $ALFA * nodoGradoMaximo$ , por lo que descartamos también esta opción.

En el cuarto y quinto método se buscan los nodos que tengan  $d_{in}$  o  $d_{out}$  mayor o igual a la proporción  $ALFA$ , pero esta proporción estaba basada en el grado no orientado del nodo con mayor grado y estábamos comparando grados no orientados contra grados orientados que no tenían necesariamente que tener relación.

Eligiendo el método 3), podemos asegurarnos de siempre encontrar al menos un nodo ya que buscaremos aquellos que tengan grado mayor o igual a  $ALFA * nodoGradoMaximo$  donde el nodo de mayor grado estará implícitamente incluido ya que  $ALFA$  es un valor entre 0 y 1 que denota un porcentaje del  $nodoGradoMaximo$ .

Una vez que tenemos todo el grafo orientado, realizamos luego una etapa de matching entre los nodos que quedan con diferentes grado de entrada y de salida, y balanceamos a todos los nodos, para que queden con iguales grados de entrada como de salida, y el grafo quede Euleriano. Este matching lo calculamos con el peso del camino mínimo entre cada uno de los nodos, calculado sobre el grafo original utilizando Dantzig, ya que en esta etapa del algoritmo solo nos interesa saber el peso del camino mínimo y no como esta compuesto.



## 4. Heurística de búsqueda Local

El algoritmo de búsqueda local se encargará, dado uno un grafo orientado completamente, un matching inicial y el grafo original, de buscar en la vecindad el mejor matching.

Un Matching está compuesto por pares de la forma (nodoOrigen, nodoDestino).

Lo que haremos para conseguir una solución vecina es intercambiar en un matching los nodos destinos de dos pares para obtener así un nuevo matching. Calculamos de esta manera todas las posibles soluciones vecinas intercambiando solo los nodos destinos de dos pares y luego de todas ellas obtenemos la que sea mínima y así poder acercarnos a una solución mejor.

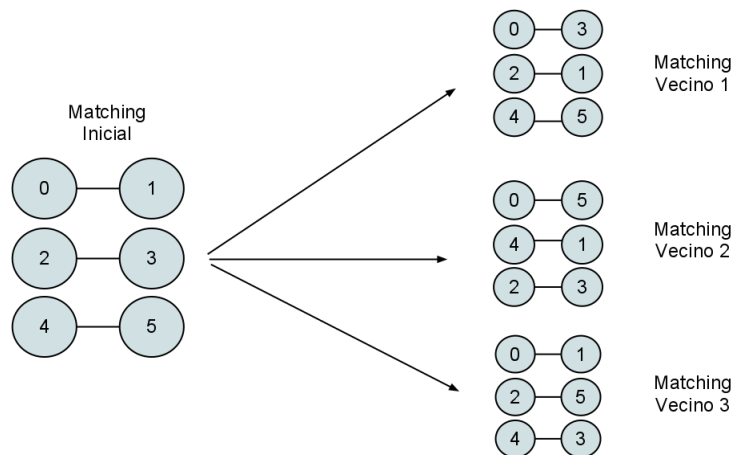
Esta definición de vecindad tiene un tamaño de  $\frac{k \cdot (k-1)}{2}$  donde  $k$  es la cantidad de pares en el matching. Esto puede verse ya que para el intercambio de un par con otro existen  $k - 1$  posibilidades de otros pares. A su vez como la cantidad de pares en el matching está determinada por la diferencia de grados de entrada y salida de los nodos, ésta equivale a:

$$\begin{aligned} & \frac{\sum_{i=1}^n |d_{in}(i) - d_{out}(i)|}{2} \\ & \frac{\sum_{1 \leq i \leq n/d_{in} > d_{out}} d_{in}(i) - d_{out}(i) + \sum_{1 \leq i \leq n/d_{out} > d_{in}} d_{out}(i) - d_{in}(i)}{2} \leq \\ & \frac{\sum_{1 \leq i \leq n/d_{in} > d_{out}} d_{in}(i) + \sum_{1 \leq i \leq n/d_{out} > d_{in}} d_{out}(i)}{2} \leq \\ & \frac{m + m}{2} = \frac{2 \cdot m}{2} = m \end{aligned}$$

Así, una vecindad tiene a lo sumo  $m$  elementos.

A su vez la cantidad de posibles matchings diferentes que pueden existir es del orden de  $k!$  ya en la elección de los nodoDestino para un nodoOrigen determinado existen  $k$  posibilidades para su destino, una vez establecido para el siguiente nodo habrá  $k - 1$  posibles destinos y así sucesivamente.

Un ejemplo de una vecindad dado un matching inicial sería la siguiente:



Una solución es vecina de otra si y solo si los matchings de ambas soluciones difieren sólo en un intercambio entre dos pares de matchings, así, en el Vecino 1 se han intercambiado los pares  $(0,1)$  y  $(2,3)$  del matching original, en el Vecino 2 los pares  $(0,1)$  y  $(4,5)$  y en el Vecino 3 los pares  $(2,3)$  y  $(4,5)$ .

## 5. Metaheurística Grasp

Esta etapa del algoritmo realiza varias iteraciones de los algoritmos de las heurísticas constructiva y de búsqueda local, decidiendo mediante experimentación el nivel de aleatoriedad agregado al algoritmo constructivo, el tamaño de la vecindad a visitar y la cantidad de veces que se intentará una solución nueva.

El parametro *ALFA* es un valor perteneciente al intervalo  $(0, 1)$  y define el nivel de aleatoriedad del algoritmo constructivo.

El parametro *CANT\_ITERACIONES\_MAXIMA* define el máximo de veces que se construirá una solución inicial.

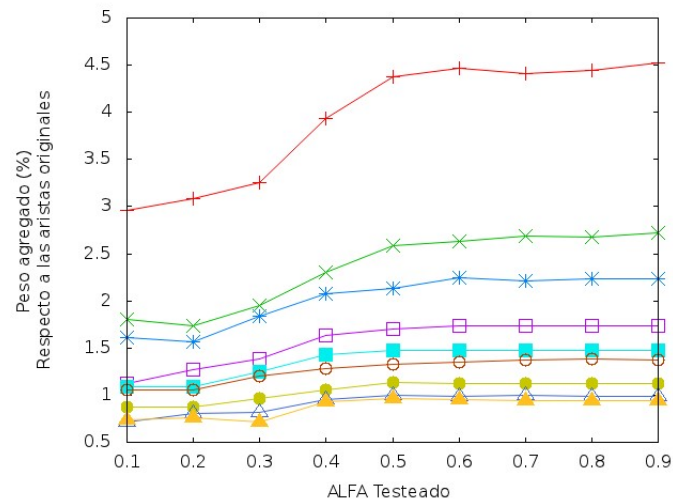
El parametro *CANT\_ITERACIONES\_SIN\_MEJORAR* se usa como condición de corte, si no se pudo encontrar una solución mejor en cierta cantidad de iteraciones de *GRASP* se corta la ejecución y se devuelve la mejor hasta el momento.

El parametro *CANT\_ITERACIONES\_BUSQUEDA\_LOCAL* define el tamaño máximo que tendrá la vecindad ya que este puede ser exponencial en el tamaño del matching.

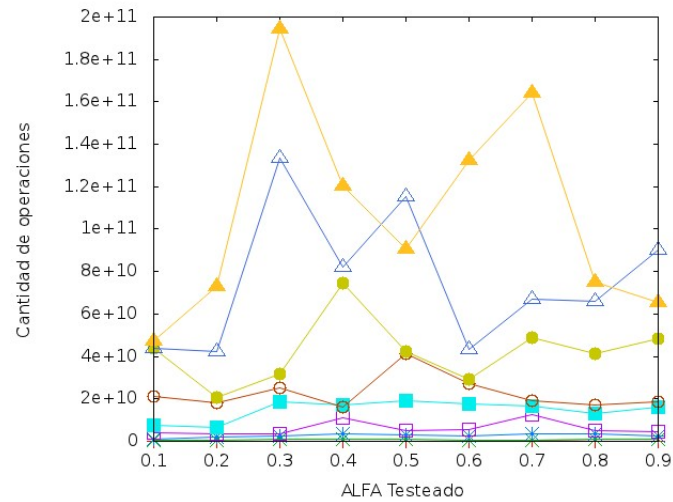
Como no tenemos el valor real de la solución óptima, medimos la calidad de las soluciones como el porcentaje de peso agregado por el algoritmo en la solución con respecto al peso total de las aristas del grafo original.  $\left( \frac{\text{PesoCircuito} - \text{PesoAristas}(G)}{\text{PesoAristas}(G)} \right) * 100$

Las instancias utilizadas para tomar las mediciones fueron grafos de 100 a 500 nodos generadas pseudo-aleatoriamente.

### 5.1. Gráficos de variación de ALFA



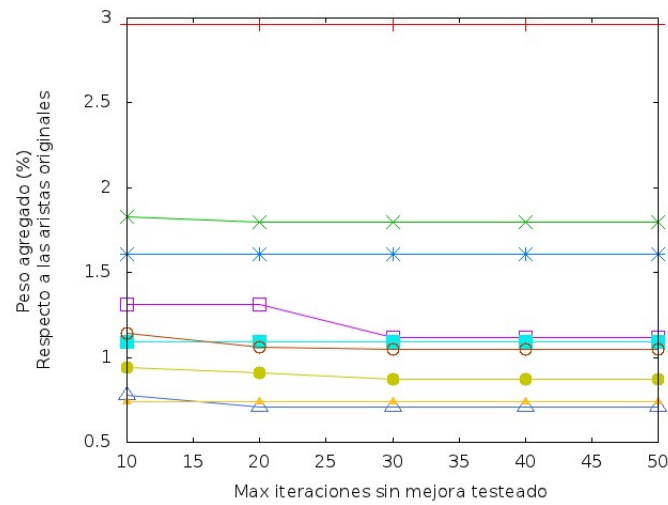
Este gráfico muestra la calidad de las soluciones con respecto a los valores para *ALFA* testeados.



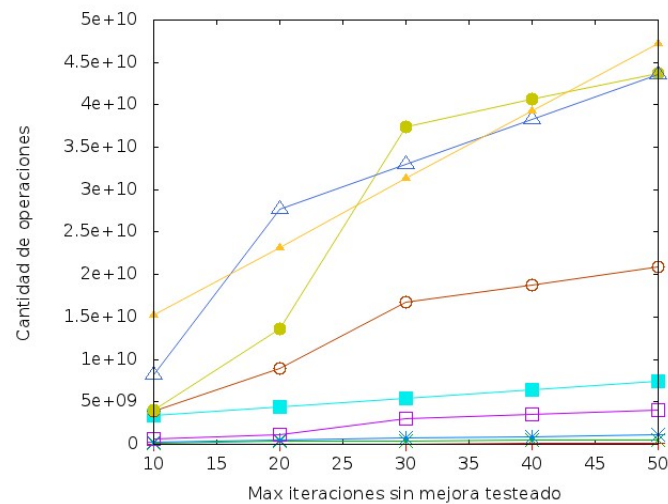
Este gráfico muestra la cantidad de operaciones realizadas para los los valores de *ALFA* testeados.

Como se puede observar en los gráficos de las mediciones tomadas para las mismas instancias con distintos valores para *ALFA*, se obtuvieron los mejores resultados para el valor 0,1 y las complejidades prácticas para ese valor son bastante razonables. El valor decidido para *ALFA* fue entonces de 0,1.

## 5.2. Gráficos de cantidad de iteraciones



El gráfico muestra la calidad de las soluciones con respecto a los los valores para *CANT\_ITERACIONES\_SIN\_MEJORAR* testeados



Este gráfico muestra la cantidad de operaciones realizadas para los los valores de *CANT\_ITERACIONES\_SIN\_MEJORAR* testeados.

Decidimos darle el valor 10 a *CANT\_ITERACIONES\_SIN\_MEJORAR* ya que en las mediciones tomadas se observa un incremento considerable en la cantidad de operaciones, mientras que las soluciones no mejoran de manera significativa.

El valor decidido para *CANT\_ITERACIONES\_MAXIMA* fue 200, ya que en ninguno de los casos evaluados llegó a cortar por esa cota.

## 6. Pseudo-algoritmo

El siguiente es un pseudo-código del algoritmo heurístico que realizamos para obtener el circuito Euleriano:

### Algoritmo GRASP

Mientras no se superen las  $I$  iteraciones totales y las  $J$  iteraciones sin mejorar de *GRASP* hacer:

- **Algoritmo Constructivo**

- 1) Calcular Dantzig sobre el grafo.
- 2) Orientar todas las aristas no orientadas del grafo.
- 3) Obtener un matching inicial entre los nodos que tengan  $d_{in} \neq d_{out}$ .

**Fin Algoritmo Constructivo**

- **Algoritmo Búsqueda Local**

*Mientras* no se superen  $K$  iteraciones y no se haya encontrado un matching óptimo hacer

- 4) Obtener los matchings vecinos y quedarse con el mínimo vecino

*Fin Mientras*

**Fin Algoritmo Búsqueda Local**

- 5) Calcular los caminos mínimos entre los nodos del matching.
- 6) Calcular el circuito Euleriano

## 7. Complejidad

Complejidades donde  $n$  es la cantidad de nodos,  $m_1$  la cantidad de aristas,  $m_2$  la cantidad de arcos y  $m = m_1 + m_2$

- Inicializar grafo (lectura de archivo)

- Inicializar la matriz de pesos  $O(n^2)$
- Inicializar las listas de adyacencias  $O(n + m_1 + m_2)$

En total:  $O(n^2 + m_1 + m_2) \subseteq O(n^2)$

- FuertementeConexo

- Inicialización de listas de adyacencias  $O(n + m_1 + m_2)$
- resetVisitados y checkVisitados recorren un array de bools de tamaño  $n$ .  $O(n)$
- visitarNodos recorre recursivamente en DFS las adyacencias y marca los nodos que son accesibles desde el nodo 0.  $O(m_1 + m_2)$

En total:  $O(n + m_1 + m_2)$

- Calcular Dantzig

- Inicialización de la matriz de pesos de caminos mínimos  $O(n^2)$
- 3 for anidados de 0 a  $n$  con operaciones de tiempo constante  $O(n^3)$

En total:  $O(n^3)$

- sumaPesosEjes recorre todas las aristas y arcos de los nodos  $O(m_1 + m_2)$

- Clonar el grafo  $O(n^2 + m_1 + m_2)$  para copiar las estructuras

- orientarTodasAristas

- $m_1$  iteraciones de
    - encontrarNodoAOrientar: Busca cuales nodos cumplen la condición dada  $O(n)$
    - eleccionNodoAOrientar: Elije segun el parametro pasado por cual nodo de la lista anterior va a continuar  $O(n)$
    - orientarNodo  $O(1)$
- $O(1 + n + n) \subseteq O(n)$

En total:  $O(m_1 * n)$

- encontrarMatchingNodos

- Inicialización de arreglos de longitud  $n$   $O(n)$
- para cada exedente de  $D_{in}$  arma una pareja con un  $D_{out}$  y lo agrega en una lista con el peso calculado con *Dantzig*  $O(\frac{\sum_{i=1}^n |d_{in}(i) - d_{out}(i)|}{2}) \subseteq O(m)$

En total:  $O(m + n)$

- encontrarMatchingDeMenorPeso: Dado un matching de tamaño  $k$  devuelve el mejor matching que se puede obtener intercambiando los destinos de solo dos items del matching original.

En total:  $O(k^2)$

- pesoMatching recorre una lista de tamaño  $a$  lo sumo  $m$

En total:  $O(m)$

- agregarCaminosMatcheados

- Inicialización de matriz de tamaño  $n * n$ .  $O(n^2)$
- Para cada eje  $(i, j)$  del matching (a lo sumo  $m$  ejes)
  - Si no esta calculado el *Dijkstra* para el nodo  $i \rightarrow$  Calcula *Dijkstra* sobre el grafo original y lo guarda.  $O(n^2)$
  - Recuperar el camino del *Dijkstra* calculado para el nodo  $i$  hasta el nodo  $j$ .  $O(n)$
- $O(n^2)$  Si no esta calculado *Dijkstra* (a lo sumo  $n$  casos)
- $O(n)$  Si ya esta calculado *Dijkstra*

En total:  $O(m * n + n * n^2)$  como  $O(m) \subseteq O(n^2) \Rightarrow O(m * n + n^3) \subseteq O(n^3)$

- Dijkstra

- Inicialización de arreglos de int y bool de tamaño  $n$ .  $O(n)$
- Para cada nodo  $i$  ( $n$  elementos)
  - Minvertex  $O(n)$
  - Para cada adyacente del nodo  $i$  actualiza el valor  $O(d(i))$

En total:  $O(n + \sum_{i=1}^n (d(i) + n)) \subseteq O(n + 2 * m + n * n) \subseteq O(n^2)$

- encontrar circuito euleriano  $O(2 * \text{largo del circuito})$

Donde largo del circuito puede ser en peor caso  $m * n$  cuando todos los nodos tiene matching y todos los caminos agregados tiene longitud  $n$ .

En total:  $O(m * n)$



---

**Algorithm 1** Main(*grafo*)

---



---

Grafo *GrafoOriginal*  $\leftarrow$  Inicializar con instancia

**if** es fuertemente Conexo *GrafoOriginal* **then**

*PesosCaminosMinimos*  $\leftarrow$  calcularDantzig(*GrafoOriginal*)

*SumaOriginalPesos*  $\leftarrow$  sumaPesosEjes(*GrafoOriginal*)

*SumaSolucionMejor*  $\leftarrow \infty$ , *MatchingMejor*  $\leftarrow \emptyset$

*GrafoCopiaMejor*  $\leftarrow NULL$

**while** no se alcanzan *MaximoIteracionesGrasp*  $\wedge$  *MaximoIteracionesSinMejoraGrasp*

**do**

Grafo *GC*  $\leftarrow$  copiarGrafo(*GrafoOriginal*)

*ParametroGRASP*  $\leftarrow$  Random

orientarTodasAristas(*GC*, *ParametroGRASP*)

*matching*  $\leftarrow$  encontrarMatchingNodos(*GC*, *ParametroGRASP*)

**while** no se alcanza *MaximoIteracionesBusquedaLocal*  $\wedge$  no se encontró un matching óptimo **do**

*matching*  $\leftarrow$  encontrarMatchingDeMenorPeso(*matching*, *pesosCaminosMinimos*)

**end while**

*pesoMatching*  $\leftarrow$  pesoMatching(*matching*)

*sumaSolucion*  $\leftarrow$  *pesoMatching* + *SumaOriginalPesos*

**if** mejoró la solucion **then**

*SumaSolucionMejor*  $\leftarrow$  *sumaSolucion*

*GrafoCopiaMejor*  $\leftarrow$  *GC*

*MatchingMejor*  $\leftarrow$  *matching*

**end if**

**end while**

agregarCaminosMatcheados(*GrafoCopiaMejor*, *MatchingMejor*, *GrafoOriginal*)

*circuito*  $\leftarrow$  encontrarCircuitoEuleriano(*GrafoCopiaMejor*)

**return** *circuito*

**else**

**return** No hay Tour porque el grafo no era fuertemente conexo

**end if**

---

Complejidad:

- $O(n^2)$  de inicialización
- $O(n + m_1 + m_2)$  de fuertementeConexo
- $O(n^3)$  de Dantzig
- $O(m_1 + m_2)$  de sumarPesosEjes
- $MaximoIteracionesGrasp *$ 
  - $O(n^2 + m_1 + m_2)$  de copiar el grafo
  - $O(m_1 * n)$  de orientarTodasLasAristas
  - $O(m_1 + m_2 + n)$  de encontrarMatchingNodos
  - $MaximoIteracionesBusquedaLocal *$ 
    - $O(m^2)$  de encontrarMatchingDeMenorPeso
  - $O(m)$  de calcular peso del matching
- $O(n^3)$  de agregarCaminosMatcheados
- $O(m * n)$  de encontrar el CircuitoEuleriano

Para grafos que no son fuertemente conexos la complejidad es  $O(n^2 + m)$

Para las instancias interesantes del problema la complejidad esta dada por:

$$O(n^2 + n + m + n^3 + m +$$

$$MaximoIteracionesGrasp * (n^2 + m + m_1 * n + m + n + MaximoIteracionesBusquedaLocal * (m^2) + m) + n^3 + m * n) \subseteq O(n^3 + m^2 + m * n) \subseteq O(n^3 + m^2)$$

Definimos el tamaño de entrada como:

$$T = \log(n) + \log(m_1) + \log(m_2) + \sum_{i=1}^{m_1+m_2} (\log(eje_{i_{nodo1}}) + \log(eje_{i_{nodo2}}) + \log(eje_{i_{peso}}))$$

Como  $\log(n) + \log(m_1) + \log(m_2) > 0 \Rightarrow$

$$T > \sum_{i=1}^{m_1+m_2} (\log(eje_{i_{nodo1}}) + \log(eje_{i_{nodo2}}) + \log(eje_{i_{peso}}))$$

También sabemos que para cada eje  $\log(eje_{i_{nodo1}}) + \log(eje_{i_{nodo2}}) + \log(eje_{i_{peso}}) > 1$  porque se necesitan al menos un bit para representar a cada eje  $\Rightarrow$

$$T > \sum_{i=1}^{m_1+m_2} 1 = m_1 + m_2 = m$$

Sabemos que el grafo es fuertemente conexo así que  $n$  es del orden de  $m \Rightarrow$

$$T > m \geq n$$

Dado la complejidad calculada y las cotas para  $T$  vale  $O(n^3 + m^2) \subseteq O(T^3 + T^2) \subseteq O(T^3)$