UNIVERSIDAD DE GRANADA
E.T.S.I. INFORMÁTICA Y TELECOMUNICACIÓN

Práctica 1.a:
Técnicas de Búsqueda Local y Algoritmos Greedy para el Problema de la Asignación Cuadrática

Álvaro Fernández García

DNI: 75572750V
3º Curso
Grupo 3: Miércoles 17:30 - 19:30
alvaro89@correo.ugr.es

Granada, curso 2017-2018

Índice

1. Formulación del Problema

El problema de asignación cuadrática (en inglés, quadratic assignment problem, QAP) es uno de los problemas de optimización combinatoria más conocidos. Fue planteado por Koopmans y Beckmann en 1957 como un modelo matemático para un conjunto de actividades económicas indivisibles.

En él se dispone de n unidades y n localizaciones en las que situarlas, por lo que el problema consiste en encontrar la asignación óptima de cada unidad a una localización. La nomenclatura "cuadrático" proviene de la función objetivo que mide la bondad de una asignación, la cual considera el producto de dos términos, la distancia entre cada par de localizaciones y el flujo que circula entre cada par de unidades. El QAP se puede formular como:

$$\underset{\pi \in \prod_n}{min} \left(\sum_{i=1}^{N}\sum_{j=1}^{N} f_{ij} \cdot d_{\pi(i)\pi(j)}\right)$$

donde:

- π es una solución al problema que consiste en una permutación que representa la asignación de la unidad i a la localización $\pi(i)$.
- f_{ij} es el flujo que circula entre la unidad i y la j.
- d_{kl} es la distancia existente entre la localización k y la l.

Se trata de un problema NP-completo. Incluso la resolución de problemas pequeños de tamaño superior a 25 se considera una tarea computacionalmente muy costosa.

Está asociado a las siguientes restricciones:

- No se puede asignar una misma unidad a dos localizaciones distintas.
- No se puede asignar más de una unidad a una misma localización.

Su espacio de búsqueda es de 2^{n^2}, donde n es el número de unidades y localizaciones.

En los siguientes ejemplos de aplicaciones se puede observar que resolver este problema para un gran número de instancias es de vital importancia, y a la vez que tratar de resolver el problema mediante técnicas completas puede resultar infactible por el alto número de instancias.

- Diseño de centros comerciales donde se quiere que el público recorra la menor cantidad de distancia para llegar a tiendas de intereses comunes para un sector del público.
- Diseño de terminales en aeropuertos, en donde se quiere que los pasajeros que deban hacer un transbordo recorran la distancia mínima entre una y otra terminal teniendo en cuenta el flujo de personas entre ellas.
- Procesos de comunicaciones.
- Diseño de teclados de computadora, en donde se quiere por ejemplo ubicar las teclas de una forma tal en que el desplazamientos de los dedos para escribir textos regulares sea el mínimo.
- Diseño de circuitos eléctricos, en donde es de relevante importancia dónde se ubican ciertas partes o chips con el fin de minimizar la distancia entre ellos, ya que las conexiones son de alto costo.

Como dato curioso, la versión de Koopmans y Beckmann tenía como entrada tres matrices $F = (f_{ij})$, $D = (d_{kl})$, $B = (b_{ik})$ del tipo real, donde (f_{ij}) especifica el flujo entre las instalaciones i y j, (d_{kl}) especifica la distancia entre las instalaciones k y l y (b_{ik}) el costo de instalar la instalación i en la locación k.

Como vemos, en la actualidad no se suele tener en cuenta la matriz B en los problemas de QAP.

2. Consideraciones generales

En este apartado, se describirán todos aquellos aspectos del problema que son utilizados en los algoritmos y que son comunes a ambos.

2.1. Representación de la solución

Para representar una solución del QAP, se ha considerado un vector permutación del conjunto

$$N = \{0, 1, \ldots, n-1\}$$

donde n representa el número de unidades/localizaciones de nuestro problema.

Según esta interpretación, cada posición del vector permutación representa una unidad, y el contenido de esa posición representa la localización asociada a dicha unidad. Por ejemplo:

0	1	2	3
0	3	1	2

vendría a significar que la unidad 0 se asigna a la localización 0, la unidad 1 a la localización 3, la unidad 2 a la localización 1 y la unidad 3 a la localización 2.

A nivel de código, se ha utilizado una estructura que contiene el vector permutación y el coste de la misma llamada **QAP_solution**.

2.2. Representación de los Flujos y Distancias

Para guardar la información asociada a los flujos y distancias se han utilizado dos matrices $n \times n$ de números en coma flotante:

- F: Representa la matriz de flujos: la posición f_{ij} representa el flujo existente entre la unidad i y la j. Es una matriz simétrica cuya diagonal principal es nula.
- D: Representa la matriz de distancias: la posición d_{ij} representa la distancia existente entre la localización i y la j. Es una matriz simétrica cuya diagonal principal es nula.

$$F = \begin{pmatrix} 0 & \ldots & f_{1n} \\ \vdots & \ddots & \vdots \\ f_{n1} & \ldots & 0 \end{pmatrix} \qquad D = \begin{pmatrix} 0 & \ldots & d_{1n} \\ \vdots & \ddots & \vdots \\ d_{n1} & \ldots & 0 \end{pmatrix}$$

2.3. Función objetivo

La función objetivo (que nos permite calcular el coste total de una solución para determinar su bondad) no es más que una traducción a código de la función matemática explicada en la sección 1:

$$coste(\pi) = \sum_{i=1}^{N} \sum_{j=1}^{N} f_{ij} \cdot d_{\pi(i)\pi(j)}$$

Su pseudocódigo es el siguiente:

Algorithm 1 Función Coste

Input: Recibe una solución, los flujos y las distancias

```
1: function QAP_COST(π, F, D)        ▷ Donde π es la solución, F la matriz de flujos y D la de distancias
2:     cost ← 0
3:     for i = 1 to n do
4:         for j = 1 to n do
5:             cost ← cost + F_ij · D_π(i)π(j)
6:         end for
7:     end for
8:     π.coste ← cost
9: end function
```

3. Algoritmo Greedy

Como primera aproximación para intentar obtener una solución al problema del QAP en un tiempo razonable, se ha diseñado un algoritmo de tipo Greedy, los cuales se caracterizan por su sencillez y su eficiencia.

Para este algoritmo, se ha considerado la heurística de asociar unidades de gran flujo con localizaciones céntricas en la red y viceversa. Para ello, se calculan dos vectores, el potencial de flujo y el potencial de distancia, que se componen respectivamente de la sumatoria de flujos de una unidad al resto ($\hat{f}_i = \sum_{j=1}^{N} f_{ij}$) y de distancias desde una localización al resto ($\hat{d}_k = \sum_{l=1}^{N} d_{kl}$). Para una unidad concreta, cuanto mayor sea $\hat{f}_i$ más importante es la unidad en el intercambio de flujos. Por otro lado, cuanto menor sea $\hat{d}_k$ más céntrica es una localización concreta. Por tanto, el algoritmo irá seleccionando la unidad i libre con mayor $\hat{f}_i$ y le asignará la localización k libre con menor $\hat{d}_k$.

Una vez dicho esto, los elementos formales de está técnica, hablando en términos del QAP son los siguientes:

- Conjunto de candidatos: Serán el conjunto de unidades y localizaciones.
- Conjunto de selccionados: Aquellas unidades para las cuales se ha asociado una localización.
- Función solución: El conjunto de candidatos se encuentra vacío, es decir, se ha asignado una localización a cada unidad.
- Función de selección: Selecciona la unidad con mayor potencial de flujo y la localización con menor potencial de distancia.
- Función objetivo: La mencionada en la Sección 2.

Su pseudocódigo es el siguiente:

Algorithm 2 Versión Greedy para el QAP

Input: recibe la matriz de flujos y distancias.
Output: devuelve una solución y su coste, (estructura QAP_solution)

```
1: function GQAP(F, D)
2:     f̂ ← Calcular potencial de flujo para cada unidad
3:     d̂ ← Calcular potencial de distancia para cada localización
4:     for i = 1 to n do
5:         maxPos ← max(f̂_i)          ▷ Seleccionar la unidad con máximo potencial de flujo
6:         minPos ← min(d̂_i)          ▷ Seleccionar la loc. con mínimo potencial de distancia
7:         π(maxPos) ← minPos          ▷ Asignar la localización a la unidad correspondiente
8:         f̂_maxPos ← −∞               ▷ Los "borramos"de candidatos asignando un
9:         d̂_minPos ← ∞                ▷ valor que asegura que no volverán a ser max. ni min.
10:    end for
       return π, QAP_cost(π, F, D)
11: end function
```

4. Algoritmo de Búsqueda Local

Como segundo algoritmo para encontrar una solución factible al problema del QAP en un tiempo razonable, se ha realizado una Búsqueda Local primero el mejor. Este tipo de búsqueda local se caracteriza porque cuando se explora el vecindario, la nueva solución pasa a ser la primera que mejore la solución actual. Se realizan un total de 50.000 exploraciones (o hasta que no se encuentre una solución vecina que mejore a la solución actual) y se parte de una solución generada aleatoriamente:

Algorithm 3 Generar una solución aleatoria

```
1: function GenerateRandom(π)
2:     index ← {1, ..., n}                          ▷ Crear un vector de índices
3:     lim ← n
4:     for i = 1 to n do
5:         in ← Random(1, lim)
6:         π(i) ← index(in)
7:         Borrar index(in) y decrementar lim
8:     end for
9: end function
```

4.1. Operador de movimiento

Para generar el vecindario de la solución actual se propone el operador de intercambio: El entorno de una solución π está formado por las soluciones accesibles desde ella a través de un movimiento de intercambio. Dado un vector permutación π que representa una solución al QAP, y dos unidades cualesquiera r y s, el operador de movimiento intercambiará las dos localizaciones asignadas a las unidades r y s.

$$\pi = \{\pi(1), \ldots, \pi_r(a), \pi_s(b), \ldots, \pi(n)\} \quad \pi' = \{\pi(1), \ldots, \pi_s(b), \pi_r(a), \ldots, \pi(n)\}$$

4.2. Factorización de coste

Para evitar volver a recalcular el coste de la solución después de aplicar el operador de intercambio, calcularemos el coste de la nueva solución a partir de la antigua factorizando, es decir, calcularemos exclusivamente la diferencia de coste existente entre ambas soluciones ($\Delta C(\pi, r, s) = C(\pi') - C(\pi)$ intercambiando r y s) . Eso se consigue mediante la siguiente expresión:

$$\sum_{k=1, k\neq r,s}^{N} \begin{bmatrix} f_{rk} \cdot (d_{\pi(s)\pi(k)} - d_{\pi(r)\pi(k)}) & + & f_{sk} \cdot (d_{\pi(r)\pi(k)} - d_{\pi(s)\pi(k)}) & + \\ f_{kr} \cdot (d_{\pi(k)\pi(s)} - d_{\pi(k)\pi(r)}) & + & f_{ks} \cdot (d_{\pi(k)\pi(r)} - d_{\pi(k)\pi(s)}) & + \end{bmatrix}$$

Una vez dicho esto, los pseudocódigos de los ultimos procedimientos son los siguientes:

Algorithm 4 Cálculo del coste factorizado

Input: Recibe π, r, s y las matrices de flujo y distancia.
Output: $\Delta C(\pi, r, s)$

```
1: function DeltaCost(π, r, s, F, D)
2:     variation ← 0
3:     for k = 1 to n do
4:         if k ≠ r, s then
5:             variation+ = [ f_rk · (d_π(s)π(k) − d_π(r)π(k))  +  f_sk · (d_π(r)π(k) − d_π(s)π(k))  + ]
                            [ f_kr · (d_π(k)π(s) − d_π(k)π(r))  +  f_ks · (d_π(k)π(r) − d_π(k)π(s))  + ]
6:         end if
7:     end for
       return variation
8: end function
```

Algorithm 5 Aplicar el operador de movimiento

Input: Recibe π, r, s y $\Delta C(\pi, r, s)$.

```
1: function ApplyMove(π, r, s, variation)
2:     π(r) ← π(s)
3:     π(s) ← π(r)
4:     π.coste ← π.coste + variation
5: end function
```

4.3. Don't Look Bits

Nuesta Búsqueda Local también dispondrá de un vector binario de tamaño n denominado DLB.

Este vector actuará como una máscara, de tal forma que solo se tendrá en cuenta para el operador de intercambio aquellas unidades cuyo bit del vector DLB esté desactivado.

En consecuencia, en la primera iteración, todos los bits están a 0, es decir, todos las unidades están activadas en un bucle externo y todos sus movimientos pueden ser considerados para explorar el entorno:

- Si tras probar todos los movimientos asociados a esa unidad, ninguno provoca una mejora, se pone su bit a 1 para desactivarla en el futuro
- Si una unidad está implicada en un movimiento que genera una solución vecina con mejor coste, se pone su bit a 0 para reactivarla

4.4. Pseudocódigo de la Búsqueda Local

Hechas todas las consideraciones, aquí se muestra su pseudocódigo:

Algorithm 6 Algoritmo de Búsqueda Local Primero Mejor para QAP

Input: Recibe las matrices de flujo y distancias.
Output: devuelve una solución y su coste, (estructura QAP_solution)

```
1: function LSQAP(F, D)
2:     π ← Solución aleatoria
3:     QAP_cost(π, F, D)                          ▷ Se calcula su coste la primera vez
4:     DLB ← {0, ..., 0}                          ▷ Se inicializa el vector DLB
5:     for k = 1 to 50000 and se encuentre una mejor solución do
6:         for i = 1 to n do
7:             if DLB(i) = 0 then
8:                 for j = 1 to n do
9:                     variation ← DeltaCost(π, i, j, F, D)   ▷ Calcular la variación de coste del nuevo vecino
10:                    if variation < 0 then                  ▷ Si la variación es negativa, el vecino es mejor
11:                        ApplyMove(π, i, j, variation)      ▷ Se aplica el operador
12:                        DLB(i) ← 0          ▷ Como han producido un buen movimiento, se desactivan
13:                        DLB(j) ← 0
14:                        Salir de los dos bucles más internos   ▷ Búsqueda Primero el Mejor!
15:                    end if
16:                end for
17:                if no se produjo ninguna mejora al intercambiar π(i) then
18:                    DLB(i) ← 1
19:                end if
20:            end if
21:        end for
22:    end for
       return π, π.coste
23: end function
```

5. Estructuración y Realización de la práctica

Los algoritmos han sido implementados desde cero en C++. Se han estructurado en un solo fichero nombrado QAP.cpp. También se ha utilizado el generador de números aleatorios proporcionado en la web de la asignatura.

El proyecto está estructurado de la siguiente forma:

```
root/
├── Memoria.pdf
└── software/
    ├── bin/
    │   ├── LEEME.txt
    │   ├── QAP
    │   └── data/
    │       ├── *.dat
    │       └── *.sln
    └── fuentes/
        ├── QAP.cpp
        ├── random.cpp
        └── random.h
```

- Memoria.pdf: Este mismo documento.
- software: Directorio que ontiene el programa en sí.
- bin: Directorio que contiene el ejecutable y los datos.
- LEEME.txt: Contiene esta misma información
- QAP: Ejecutable.
- data: Directorio que contiene los datos de QAPLIB.
- *.dat: Datos del problema.
- *.sln: Solucion al problema.
- fuentes: Contiene los códigos fuente.
- QAP.cpp: Contiene el programa principal de la practica.
- random.cpp, random.h: Generador de números aleatorios proporcionado

Para utilizar el binario basta con introducir en la términal lo siguiente:

```
bash $> cd software/bin
bash $> ./QAP data/<filename>.dat <seed> <mode>
```

Donde seed representa la semilla para el generador de números aleatorios y mode tomará como valores 0 (para imprimir solo los costes y tiempo, es decir información reducida) o 1 (para mostrar también el vector permutación obtenido).

Para la realización de la práctica se ha lanzado este ejecutable para cada uno de los archivos *.dat con una semilla de 88. La información reducida se ha utilizado para agilizar la recolección de tiempos y costes en un fichero.

Por si fuese necesario, también se indica a continuación la orden para compilar el ejecutable:

```
bash $> cd software/fuentes
bash $> g++ -o QAP -I. *.cpp -std=c++11
```

6. Experimentos y análisis de resultados

Como ya se ha mencionado en el apartado anterior, para evaluar la bondad de los algoritmos implementados se ha procedido a la ejecución de dichos algoritmos para todos y cada uno de los 20 ficheros de datos de la librería QAPLIB con una semilla de 88 para el generador de números aleatorios.

De cada ejecución se ha medido tanto el coste de la solución obtenida por ambos métodos, como el tiempo que tardan los algoritmos en calcular la solución.

Para determinar como de buenas son las soluciones encontradas, se ha procedido a calcular la desviación existente entre el coste de la solución presente en los ficheros *.sln, y el coste que encuentra nuestro algoritmo. por último, para obtener una visión global de los algoritmos, se ha calculado la media en porcentaje de la desviación obtenida en cada caso así como la media aritmética del tiempo:

$$\frac{1}{|casos|} \cdot \sum_{i \in casos} 100 \cdot \frac{valorAlgoritmo_i - mejorValor_i}{mejorValor_i}$$

6.1. Tablas de resultados

Algoritmo Greedy					
Caso	**Desv**	**Tiempo**	**Caso**	**Desv**	**Tiempo**
Chr22a	119,92	0,000124	Sko100a	13,23	0,000877
Chr22b	92,80	0,000125	Sko100f	14,38	0,001087
Chr25a	362,49	0,000104	Tai100a	13,65	0,000859
Esc128	125,00	0,001780	Tai100b	32,79	0,000894
Had20	10,11	0,000107	Tai150b	24,54	0,003035
Lipa60b	28,25	0,000360	Tai256c	119,84	0,003790
Lipa80b	29,42	0,000654	Tho40	30,11	0.000283
Nug28	19,09	0,000084	Tho150	17,14	0,001888
Sko81	16,30	0,000611	Wil50	11,99	0,000339
Sko90	13,78	0,001109	Wil100	7,37	0,000905

Algoritmo de Búsqueda Local					
Caso	**Desv**	**Tiempo**	**Caso**	**Desv**	**Tiempo**
Chr22a	13,55	0,007022	Sko100a	3,35	1,326400
Chr22b	6,88	0,006041	Sko100f	2,01	1,316845
Chr25a	59,27	0,012097	Tai100a	3,66	0,583315
Esc128	21,88	0,806851	Tai100b	1,46	1,251209
Had20	1,07	0,008395	Tai150b	3,78	4,803096
Lipa60b	20,44	0,100490	Tai256c	0,46	7,566162
Lipa80b	21,79	0,268598	Tho40	6,62	0,044795
Nug28	4,18	0,012167	Tho150	3,20	5,000590
Sko81	2,74	0,540780	Wil50	2,47	0,089452
Sko90	2,15	0,956031	Wil100	1,17	1,278674

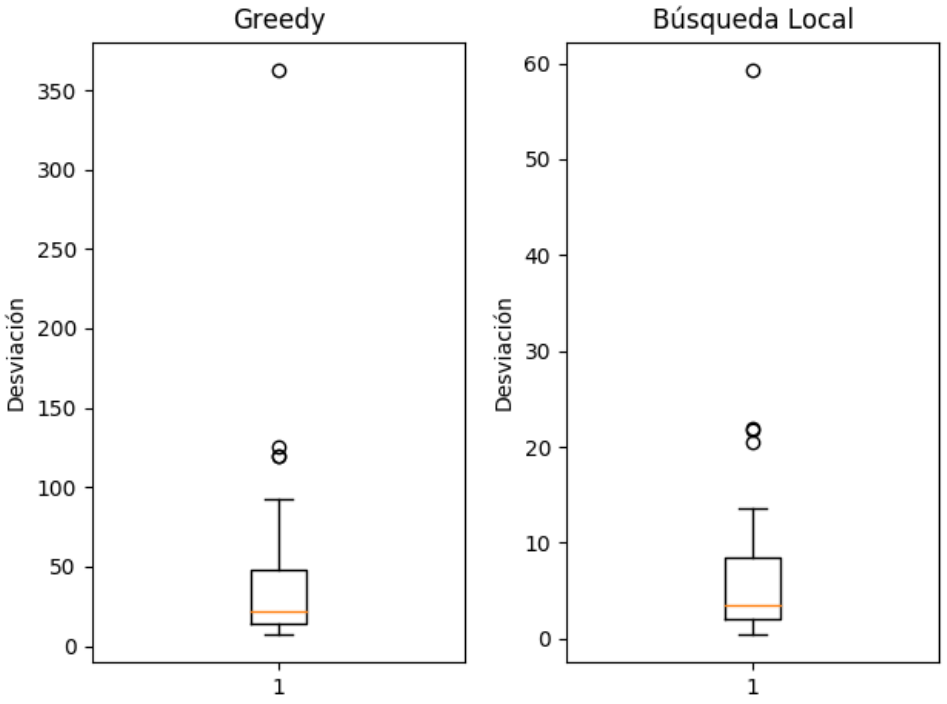
Nota: Los tiempos aparecen en segundos.

6.2. Resultados Globales en el QAP

Algoritmo	**Desv**	**Tiempo**
Greedy	55,11	0,00094
BL	9,11	1,29895

6.3. Diagramas de Caja

Para ayudar en su interpretación, aquí se muestran dos diagramas de caja que representan la desviación para cada uno de los algoritmos:



6.4. Conclusión

Una de las primeras cosas que saltan a la vista cuando se mira la tabla, es que ninguno de los dos algoritmos consigue dar una respuesta óptima a ninguno de los 20 casos planteados. Ambos son algoritmos que nos dan una aproximación a las mejores soluciones obtenidas.

En el caso del algoritmo greedy la aproximación deja bastante que desear. En el mejor de los casos, el agoritmo consigue una desviación de 7,37 con respecto a la solución conocida, pero en el resto de los casos las desviaciones no bajan de 10, incluso habiendo casos desastrosos, en los que las desviaciones superan el 100 e incluso el 300 %. De todo esto podemos deducir que la heurística planteada para el algoritmo greedy es bastante imparcial y no tiene un comportamiento regular sobre los distintos problemas. Dicho de otro modo, para algunos problemas (aunque escasos) puede dar una respuesta razonable, pero en la mayoría, las respuestas no son representativas.

Con respecto a la búsqueda local, las soluciones son mucho más razonables que en el caso del algoritmo greedy. Tampoco llega a dar una respuesta exacta en ninguno de los casos pero si que es cierto que hay algunos en los que la respuesta es una aproximación al coste óptimo muy acertada (como en el caso de Tai256c, con una desviación del 0.46 %). Además este caso en concreto presenta una gran mejora con respecto al greedy (de 119 de desviación que había en el greedy a tan solo 0.46 con la búsqueda Local).

Observando los diagramas de caja, podemos ver que únicamente se producen cuatro resultados atípicos (nótese que hay un punto doble en los diagramas, por valores muy cercanos), es decir, los algoritmos tienen dificultades para encontrar una solución tan aproximada como las otras para los casos de Char25a, Esc128, Char22a, Tai256b en el algoritmo Greedy, y Char25a, Esc128, Lipa60b, Lipa80b en la búsqueda local. Ambos comparten Char25a (que es el más desastroso de todos) y Esc128. Sería interesante comprobar que características concretas tienen estos problemas que no tengan los demás, e intentar descubrir porque los algoritmos no funcionan tan bien como en el resto. No obstante esto no es sencillo, y siempre queda la

posibilidad de que simplemente se deba a que los algoritmos, con los parámetros establecidos, simplemente no llegan a una solución razonable.

Para terminar queda hablar un poco sobre tiempos. Es indiscutible que el algoritmo Greedy tiene un tiempo mucho menor que el de la búsqueda local. No obstante, gracias a las optimizaciones que hemos realizado sobre la búsqueda local (la máscara DLB y la factorización en el cálculo del coste), los tiempos no son para nada elevados y siguen manteniendose dentro de la razonabilidad, (de hecho, lo máximo que tarda son 7 segundos y medio para el caso de $n = 256$).

En definitiva, si tuviera que elegir entre los dos algoritmos, elegiría la búsqueda local. Realmente, la ganancia de velocidad que se adquiriría por utilizar el greedy, no compensa para nada la bondad de las soluciones alcanzadas por la búsqueda local.