

\*\*\*\*\*

# **PRÁCTICA 5.**

## **Búsquedas Híbridas para el Problema de la Asignación Cuadrática**

\*\*\*\*\*

**ALGORITMOS:**

● **Algoritmos Meméticos.**

**Autora: Cristina Zuheros Montes-50616450**

● Correo: [zuhe18@gmail.com](mailto:zuhe18@gmail.com)

● Github: <https://github.com/cristinazuhe>

**Horario prácticas: Viernes 17:30-19:30**

**Fecha: 11 Julio 2016**

# **ÍNDICE:**

## **1. Descripción del problema**

## **2. Descripción de aplicación de los Algoritmos.**

- Greedy.
- Funciones auxiliares.
- Búsqueda Local.
- Algoritmos Meméticos (AM).

## **3. Descripción en pseudocódigo de los Algoritmos.**

- Búsqueda Local.
- AM1.
- AM2.
- AM3.

## **4. Experimentos y análisis de resultados**

## 1. Descripción del problema.

EL problema de la asignación cuadrática (QAP) que vamos a analizar, trata de conseguir una asignación óptima de  $n$  unidades en  $n$  localizaciones, conociendo la distancia entre las unidades y el flujo entre las localizaciones. Tanto las distancias entre las unidades como el flujo entre las localizaciones vendrán dados por matrices, que denotaremos como “ $d$ ” y “ $f$ ” respectivamente. Así pues,  $f_{ij}$  denota el flujo existente entre la unidad “ $i$ ” y la unidad “ $j$ ”. Análogo para la matriz de distancias.

Una vez tengamos las asociaciones de unidades-localidades, vamos a obtener su coste. El objetivo es conseguir el mínimo coste posible, que nos dará la solución más cercana a la óptima. La función de coste viene determinada por:

coste solucion: 
$$\min_{\pi \in \Pi_N} \left( \sum_{i=1}^n \sum_{j=1}^n f_{ij} \cdot d_{\pi(i)\pi(j)} \right)$$

donde  $\pi$  es una solución al problema que consiste en una permutación que representa la asignación de la unidad “ $i$ ” a la localización “ $\pi(i)$ ”

### Ejemplo:

Para hacernos una idea más clara, supongamos que tenemos las unidades 0,1,2 y las localidades 0,1,2.

Si la solución encontrada es la permutación {2 0 1} quiere decir que a la unidad 0 le corresponde la localización 2, a la unidad 1 le corresponde la localización 0 y a la unidad 2, la localización 1.

## 2. Descripción de aplicación de los Algoritmos.

### ● Greedy.

Mediante este algoritmo asignamos las unidades que tengan mayor flujo a localizaciones que se encuentren céntricas.

La idea es obtener vectores de flujo y de distancias como marginalizaciones de las correspondientes matrices. A partir de dichos vectores, iremos asociando la unidad libre con mayor flujo a la localización libre con menor distancia.

### ● Funciones auxiliares.

Vamos a ver varias funciones auxiliares que aplicaremos en los algoritmos (mat1 representa flujos y mat2 representa distancias).

#### ● Pseudocódigo de la función coste.

Vemos el pseudocódigo de la función coste vista en el primer apartado que llamábamos descripción del problema:

**Coste\_Solución:**

```
for i=0...n-1
  for j=0...n-1
    si i!=j
      coste += mat1(i,j)*mat2(sol(i),sol(j))
```

#### ● Pseudocódigo de generación de soluciones aleatorias.

También tendremos una función auxiliar para generar una solución aleatoria inicial para los algoritmos. Para ello haremos uso de una misma semilla para todos los algoritmos, que nos irá generando números aleatorios.

**Permutar\_Solucion\_n:**

```
for i=0...n-1
  gen1 <- aleatorio entre [0,n-1]
  gen2 <- aleatorio entre [0,n-1] != gen1
  permuto en la solución los indices gen1 y gen2
```

- **Nota:** Volvemos a insistir en la representación de soluciones que vamos a emplear:

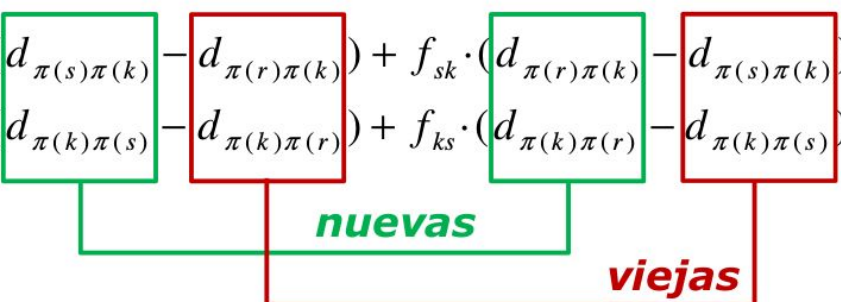
$\pi$  hace referencia a una solución al problema que consiste en una permutación que representa la asignación de la unidad “i” a la localización “ $\pi(i)$ ” (Ver ejemplo comentado en el primer apartado donde explicamos la descripción del problema).

Reemplazamiento: la población de hijos sustituye automáticamente a la actual. Si la mejor solución de la generación anterior no sobrevive, sustituimos la peor solución de la nueva población.

### ● Búsqueda Local.

Partiremos de una solución inicial, en la que permutaremos dos unidades-localizaciones (solución vecina). Si esta nueva solución mejora a la solución anterior, se aplicará el movimiento y se tomará dicha solución como la actual.

Para ver si esta nueva solución mejora a la solución anterior, basta con obtener el coste del intercambio. Para ello calcularemos:

$$\sum_{k=1, k \neq r, s}^n \left[ f_{rk} \cdot (d_{\pi(s)\pi(k)} - d_{\pi(r)\pi(k)}) + f_{sk} \cdot (d_{\pi(r)\pi(k)} - d_{\pi(s)\pi(k)}) + \right. \\ \left. f_{kr} \cdot (d_{\pi(k)\pi(s)} - d_{\pi(k)\pi(r)}) + f_{ks} \cdot (d_{\pi(k)\pi(r)} - d_{\pi(k)\pi(s)}) \right]$$


The diagram highlights the terms in the formula using colored boxes and labels:

- nuevas (green boxes):**  $d_{\pi(s)\pi(k)}$ ,  $d_{\pi(k)\pi(s)}$ ,  $d_{\pi(r)\pi(k)}$ , and  $d_{\pi(k)\pi(r)}$ .
- viejas (red boxes):**  $d_{\pi(r)\pi(k)}$ ,  $d_{\pi(s)\pi(k)}$ ,  $d_{\pi(k)\pi(r)}$ , and  $d_{\pi(k)\pi(s)}$ .

Para obtener la lista de candidatos usaremos la técnica don't look bits. Finalmente el algoritmo se detendrá cuando se haya explorado el vecindario sin encontrar mejoras o bien cuando se hayan evaluado 400 vecinos distintos.

## ● **Algoritmos Meméticos.**

El AM consistirá en hibridar el algoritmo genético generacional (AGG) que hicimos y explicamos en la Práctica 3 con la Búsqueda local que hemos comentado anteriormente. En la Práctica 3 hicimos dos versiones de AGG (con cruce de posición y cruce OX). En vista a los resultados que vimos en la misma Práctica 3, hemos decidido que lo mejor para esta práctica será trabajar con el cruce de posición.

Se estudiarán las tres posibilidades de hibridación siguientes:

**AM0 → AM-(10,1.0):** Cada 10 generaciones, se aplica la BL sobre todos los cromosomas de la población: Recorremos toda la población intermedia.

**AM1 → AM-(10,0.1):** Cada 10 generaciones, se aplica la BL sobre un subconjunto de cromosomas de la población seleccionado aleatoriamente con probabilidad pLS igual a 0.1 para cada cromosoma: Recorremos  $0.1 \cdot \text{tamaño de la población intermedia}$  cromosomas. Además, los cromosomas que tomamos de la población serán aleatorios.

**AM2 → AM-(10,0.1mej):** Cada 10 generaciones, aplicar la BL sobre los  $0.1 \cdot N$  mejores cromosomas de la población actual (N es el tamaño de ésta). Recorremos  $0.1 \cdot \text{tamaño de la población intermedia}$  cromosomas. Estos cromosomas serán los primeros de la población pues los tenemos ordenados de mejor a peor.

### 3. Descripción en pseudocódigo de los Algoritmos.

#### ● Búsqueda Local.

**Inicializamos el vector dlb a 0's.**

**Mientras que se mejore solución, hacemos** dontlookbits:

**mejora** <- false

*for i=0...n-1 && !mejora && < 400 vecinos generados*

*si dlb[i]=0*

*mejora* <- false

*for j=0...n-1*

*si i!=j && mejora\_permutar(i,j)*

*mejora* <- true

*marco unidades i y j en dlb a 0*

*si !mejora*

*marco unidad i en dlb a 1*

Pseudocódigo funciones auxiliares:

**Mejora\_permutar(i,j):**

*mejora* <- false

*costecambio* <- **coste\_Posicion** de permutar unidades i,j

*Si costecambio negativo*

*permuto unidades i y j en solución actual*

*actualizo coste de solución actual*

*mejora* <- true

**coste\_Posicion(sol, i, j):**

*Obtengo el coste de permutar las unidades i, j de solución usando la formula vista en la página 5.*

Tras el proceso de mutación en AGG, al pasar 10 generaciones, realizaremos el siguiente proceso:

### ● AM0.

```
for i=0...M-1
  for j=0...n-1
    soluciones_aux[j] <- poblacion_intermedia[i][j]
    coste_solucion_aux <- poblacion_intermedia[i][n]
    alg_Busqueda_Local con soluciones_aux y su coste
    Si el coste de la nueva solucion es mejor que la anterior
      cambio la solucion de poblacion_intermedia por la nueva
      cambio el coste de esa solucion en la poblacion_padres
    incremento n_evaluaciones para criterio de parada
```

### ● AM1.

```
for i=0...(0.1*M)-1
  k <- aleatorio de 0 a M-1
  for j=0...n-1
    soluciones_aux[j] <- poblacion_intermedia[k][j]
    coste_solucion_aux <- poblacion_intermedia[k][n]
    alg_Busqueda_Local con soluciones_aux y su coste
    Si el coste de la nueva solucion es mejor que la anterior
      cambio la solucion de poblacion_intermedia por la nueva
      cambio el coste de esa solucion en la poblacion_padres
    incremento n_evaluaciones para criterio de parada
```

### ● AM2.

```
for i=0...(0.1*M)-1
  for j=0...n-1
    soluciones_aux[j] <- poblacion_intermedia[i][j]
    coste_solucion_aux <- poblacion_intermedia[i][n]
    alg_Busqueda_Local con soluciones_aux y su coste
    Si el coste de la nueva solucion es mejor que la anterior
      cambio la solucion de poblacion_intermedia por la nueva
```



*cambio el coste de esa solucion en la poblacion\_padres  
incremento n\_evaluaciones para criterio de parada*

## 4. Experimentos y análisis de resultados

Usando AM0.

AM0			
Caso	Coste obtenido	Desv	Tiempo
Chr20a	2492	13,69	1,81403
Chr20c	17370	22,83	2,02879
Chr22b	6676	7,78	2,50274
Chr25a	4676	23,18	3,79281
Esc32a	144	10,77	6,75922
Esc64a	116	0,00	38,77348
Esc128	64	0,00	286,80408
Kra32	91660	3,34	8,19062
Lipa90a	363118	0,69	143,27011
Sko42	16096	1,80	23,15730
Sko49	23860	2,03	39,45654
Sko81	92262	1,39	204,01459
Sko90	117292	1,52	303,90912
Sko100f	151332	7,56	428,86710
Tai64c	3082764	66,10	43,70454
Tai80a	15717270	16,43	114,19128
Tai100a	24012022	51,55	213,60796
Tai150b	650769637	47,30	1573,37952
Tai256c	53058506	21,00	2264,67578
Tho150	9729522	27,67	1522,10181

Vemos que en general los resultados no son demasiado buenos aunque encontramos un intervalo (desde esc32a hasta sko100f) con desviaciones relativamente leves.

Hay casos, como tai100a, en los que se obtienen resultados pésimos.

Además el tiempo de ejecución es muy alto, llegando en algunos casos a tardar unos 15/20 minutos.

### Usando AM1.

AM1			
Caso	Coste obtenido	Desv	Tiempo
Chr20a	2772	26,46	0,30930
Chr20c	23532	66,40	0,32097
Chr22b	6802	9,82	0,38272
Chr25a	4852	27,82	0,56930
Esc32a	156	20,00	0,99470
Esc64a	116	0,00	5,18923
Esc128	64	0,00	37,53840
Kra32	91450	3,10	1,16027
Lipa90a	363424	0,77	18,41688
Sko42	16282	2,97	3,01205
Sko49	24000	2,63	5,09421
Sko81	92324	1,46	25,26298
Sko90	117810	1,97	39,58003
Sko100f	151746	7,86	55,15745
Tai64c	2489166	34,12	5,72942
Tai80a	15600978	15,57	14,77783
Tai100a	24211090	52,80	26,59350
Tai150b	650893354	47,33	188,87380
Tai256c	51695640	17,89	268,01419
Tho150	9816056	28,81	182,46660

Los resultados obtenidos en general no son muy buenos: en varios casos como esc64a y esc128 llegamos a encontrar el óptimo, pero tenemos casos como chr20c y tai100a que no son para nada aceptables.

En cuanto al tiempo consumido, vemos que es bastante bueno en los primeros casos pero, en cuanto trabajamos con un gran número de unidades-localizaciones, el tiempo se incrementa notablemente.

## Usando AM2.

AM2			
Caso	Coste obtenido	<u>Desv</u>	Tiempo
Chr20a	2688	22,63	0,30181
Chr20c	22064	56,02	0,34459
Chr22b	6622	6,91	0,37798
Chr25a	4794	26,29	0,57618
Esc32a	150	15,38	1,00104
Esc64a	116	0,00	5,18675
Esc128	64	0,00	35,58659
Kra32	91700	3,38	1,14862
Lipa90a	363118	0,69	18,27931
Sko42	16122	1,96	3,01239
Sko49	23722	1,44	5,09031
Sko81	92392	1,53	26,05191
Sko90	117768	1,93	39,84710
Sko100f	151120	7,41	52,29962
Tai64c	2832596	52,62	5,63176
Tai80a	15598110	15,55	14,00986
Tai100a	24008718	51,52	26,32087
Tai150b	652422654	47,68	179,30669
Tai256c	55308018	26,13	268,70084
Tho150	9764126	28,13	182,37914

Obtenemos resultados bastante drásticos: muy buenos en algunos casos en los que incluso se llega a alcanzar el óptimo y muy malos en algunos casos en los que no tiene sentido consumir el tiempo que se consume para obtener esos resultados.

En los problemas con pocas unidades-localizaciones se obtienen tiempos aceptables pero en los últimos casos se produce un gran incremento del tiempo.

Cabe comentar antes de nada, que para todos estos resultados que mostramos hemos usado la semilla 5500055.

### **Comparando algoritmos:**

Algoritmo	Desv	Tiempo
Greedy	104,55	0
AM0	16,33	361,25
AM1	18,38	43,97
AM2	18,36	43,27

Antes de nada merece la pena comentar que en la Práctica 3 obteníamos una desviación media para AGG (con cruce de posición) de 22.06.

Vemos que con cualquiera de los algoritmos meméticos se han mejorado los resultados. Hemos conseguido que la búsqueda local introduzca mayor explotación de las soluciones prometedoras.

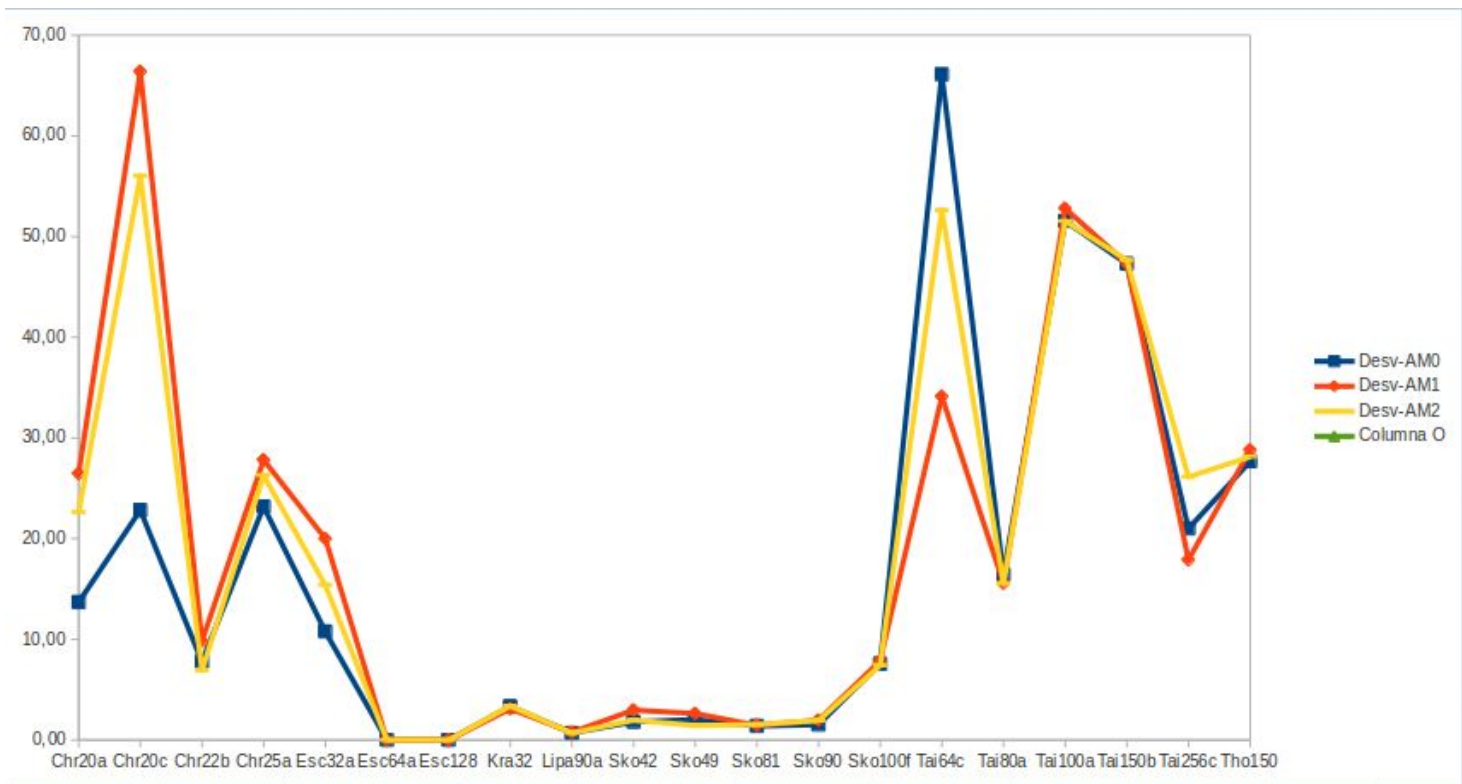
Con el algoritmo memético 0 (AM0) vemos que se consigue la mejor desviación pero en media se consume mucho tiempo. Esto se debe a que estamos aplicando la búsqueda local sobre todo los cromosomas de la población, teniendo que realizar un gran número de operaciones.

Vemos que con AM1 se obtienen peores resultados que con AM0, pero el tiempo invertido en la ejecución es mucho menor, de modo que merece la pena AM1 frente a AM0.

Sin embargo, el algoritmo memético que parece más prometedor es AM2. Conseguimos unos resultados mejores que con AM1 e invertimos menos tiempo. Esto se debe a que sólo estamos haciendo búsqueda local sobre los mejores cromosomas de la población. En general, si tenemos que optar por algún algoritmo memético, este parece el más interesante por la calidad desviación-tiempo obtenidos. Sin embargo,

estas desviaciones siguen siendo bastante grandes así que sigue dejando bastante por desear.

### Valorando casos particulares:



En este gráfico comparamos las desviaciones particulares para los distintos casos de estudio utilizando AM0, AM1 y AM2.

Aquí se puede ver claramente que AM1 es el algoritmo que presenta peores resultados, obteniendo la desviación más elevada para prácticamente todos los casos.

Además vemos que, por el contrario, AM0 presenta los mejores resultados, obteniendo las menor desviación para prácticamente todos los casos.

Con esto podríamos pensar que AM0 es el más prometedor, pero si vemos la relación desviación obtenida / tiempo de ejecución, nos



damos cuenta de que AM0 no interesa tanto como puede parecer en dicha gráfica.

Viendo las tablas anteriores y la gráfica, por un lado observamos que los casos chr20c y tai64c son de los más problemáticos en el sentido de que con cualquiera de los algoritmos realizados obtenemos soluciones muy malas.

Por otro lado, destaca los casos esc64a y esc128 con el que obtenemos resultados bastante cercano al óptimo con cualquiera de los algoritmos.

#### **Número de veces que es mejor....:**

Hemos recopilado para cada caso cuál es el mejor algoritmo obteniendo dichos resultados. Nuevo de veces que es mejor:

Greedy: 3

AM0: 7

AM1: 3

AM2: 7

Tenemos que AM1, a pesar de tener malas desviaciones medias, hay casos en los que obtiene la mejor solución con respecto a los otros algoritmos. Esto se debe a que nos encontramos fuertes picos de subida, e incluso bajada.

Finalmente, llama la atención el hecho de que usando Greedy tengamos 3 casos de estudio en los que encuentra la mejor solución. Esto se debe a que los algoritmos meméticos no nos han aportando buenas soluciones, de modo que con cualquier otro algoritmo, por simple que sea, podremos llegar a encontrar casos en los que obtengamos mejores resultados que con los meméticos. Esto también ocurría con los algoritmos genéticos.

### **Usando distintas semillas:**

Finalmente en nuestro análisis hemos probado a ejecutar los algoritmos con distintas semillas.

Veamos las desviaciones de, por ejemplo:

- **Con semilla 6060:**

AM0: 17.15

AM1: 20.12

AM2: 20.01

- **Con semilla 1111:**

AM0: 18.04

AM1: 19.36

AM2: 19.24

- **Con semilla 9090:**

AM0: 19.14

AM1: 21.89

AM2: 20.17

En general vemos la misma traza que con la semilla con la que hemos hecho el estudio: AM0 obtiene las mejores desviaciones pero el tiempo invertido sigue siendo muy alto (aunque aquí no lo hemos indicado, se ha tardado mucho). AM1 obtiene los peores resultados pero están muy cercanos a AM2, siendo los tiempos de ambos relativamente cercanos.

