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

Práctica 3.a:
Búsquedas por Trayectorias 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 **2**

2. Consideraciones generales **3**

2.1. Representación de la solución 3

2.2. Representación de los Flujos y Distancias 3

2.3. Función objetivo 3

2.4. Búsqueda Local 4

2.5. Generación aleatoria de soluciones 5

3. Búsqueda Local Multiarranque Básica (BMB) **5**

4. Algoritmo GRASP **6**

5. Enfriamiento Simulado **7**

6. Iterated Local Search (ILS) **8**

7. Hibridación entre ILS y Enfriamiento Simulado **9**

8. Estructuración y Realización de la práctica **10**

9. Experimentos y análisis de resultados **11**

9.1. Tablas de resultados 11

9.2. Resultados Globales en el QAP 13

9.3. Conclusión 13

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 $n!$, 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 algunos de los algoritmos y que son comunes a todos ellos.

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
```

2.4. Búsqueda Local

La búsqueda local se aplica en los algoritmos GRASP, BMB e ILS, por lo que he decidido colocar su pseudocódigo dentro de este apartado. Es exactamente la misma que en la primera práctica, junto con Don't Look Bits y la factorización del coste. Su pseudocódigo es como sigue:

Algorithm 2 Algoritmo de Búsqueda Local Primero Mejor para QAP

Input: Recibe la solución de la que parte la búsqueda local y las matrices F y D

```
1: function LS(π, F, D)
2:     evaluaciones ← 0
3:     DLB ← {0, . . . , 0}                                          ▷ Se inicializa el vector DLB
4:     repeat
5:         for i = 1 to TamañoProblema and evaluaciones < 50000 do
6:             if DLB(i) = 0 then
7:                 for j = 1 to TamañoProblema and evaluaciones < 50000 do
8:                     variation ← DeltaCost(π, i, j, F, D)
9:                     evaluaciones ← evaluaciones + 1
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:
23:    until evaluaciones > 50000 or no se encuentre una mejor solución
24: end function
```

Algorithm 3 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
```

Algorithm 4 Cálculo del coste factorizado

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

function DeltaCost(π, r, s, F, D)
  variation $\leftarrow 0$
  for $k = 1$ **to** n **do**
    if $k \neq r, s$ **then**
      $variation+ = \begin{bmatrix} f_{rk} \cdot (d_{\pi(s)\pi(k)} - d_{\pi(r)\pi(k)}) & + & f_{sk} \cdot (d_{\pi(r)\pi(k)} - d_{\pi(s)\pi(k)}) & + \\ f_{kr} \cdot (d_{\pi(k)\pi(s)} - d_{\pi(k)\pi(r)}) & + & f_{ks} \cdot (d_{\pi(k)\pi(r)} - d_{\pi(k)\pi(s)}) & + \end{bmatrix}$
    end if
  end for
  return variation
end function

2.5. Generación aleatoria de soluciones

En este apartado se describe el procedimiento que se sigue para generar una solución aleatoria del problema del QAP. Dicha solución aleatoria se tomará como punto de partida para el algoritmo ILS, y también para cada una de las soluciones iniciales en el agoritmo BMB. Su pseudocódigo es el siguiente:

Algorithm 5 Generar una solución aleatoria

function GenerateRandom(π, F, D)
  index $\leftarrow \{1, \ldots, n\}$ ▷ Crear un vector de índices
  lim $\leftarrow$ n
  for $i = 1$ **to** n **do**
    in $\leftarrow$ Random(1, lim)
    $\pi(i) \leftarrow$ index(in)
    Borrar index(in) y decrementar lim
  end for
  QAP_cost(π, F, D)
end function

3. Búsqueda Local Multiarranque Básica (BMB)

Es un algoritmo muy sencillo. Simplemente, consiste en ejecutar varias búsquedas locales partiendo desde distintas soluciones generadas aleatoriamente. Al final, se devuelve la mejor solución encontrada de entre todas las búsquedas locales realizadas. Aquí se muestra el pseudocódigo del algoritmo:

Algorithm 6 Algoritmo Búsqueda Multiarranque Básica

Input: Recibe las matrices F y D.
Output: Devuelve una solución al QAP

function BMB(F, D)
  mejorSolución.coste $\leftarrow \infty$
  for $i = 1$ **to** 25 **do**
    GenerateRandom(S, F, D)
    $S' \leftarrow$ BúsquedaLocal(S, F, D)
    if S'.coste $<$ mejorSolución.coste **then**
      mejorSolución $\leftarrow S'$
    end if
  end for
  return mejorSolución
end function

4. Algoritmo GRASP

Empezaremos en primer lugar describiendo el pseudocódigo del algoritmo Greedy Aleatorizado. Este es empleado para obtener la solución inicial de la que partirán las iteraciones en el algoritmo GRASP. Su implementación es bastante compleja, por lo que para hacer más entendible el pseudocódigo, se describirá a un nivel más alto de abstracción del habitual.

Algorithm 7 Greedy Aleatorizado

Input: Recibe las matrices F y D.
Output: Devuelve un solución Greedy Aleatorizada.

1: **function** RandomizedGreedy(F, D)
2: PF ← CalculaMatrizPotencial(F) ▷ Calculamos los potenciales
3: DF ← CalculaMatrizPotencial(D)
4: $\mu_f \leftarrow$ Max(PF) $-\ \alpha\ \cdot$ (Max(PF)−Min(PF)) ▷ Calcular los umbrales
5: $\mu_d \leftarrow$ Min(PD) $+\ \alpha\ \cdot$ (Max(PD)−Min(PD))
6: **for** $i = 1$ **to** n **do** ▷ Construir las LCR
7: **if** PF[i] $\geq \mu_f$ **then**
8: $LCR_f \leftarrow LCR_f \cup \{PF[i]\}$
9: **end if**
10: **if** PD[i] $\leq \mu_d$ **then**
11: $LCR_d \leftarrow LCR_d \cup \{PD[i]\}$
12: **end if**
13: **end for**
14: $u_1, u_2, l_1, l_2 \leftarrow$ SeleccionarAleatoriamente(LCR_f, LCR_d) ▷ 2 unidades y 2 localizaciones aleatorias
15: $\pi[u_1] \leftarrow l_1$
16: $\pi[u_2] \leftarrow l_2$
17: S ← $\{\{u_1, l_1\},\ \{u_2, l_2\}\}$ ▷ Construir S
18: LC ← Construir todos los pares $\{u_i, l_i\}$ que no incluyan a los ya seleccionados
19: **for** $i = 1$ **to** $n - 2$ **do**
20: C ← Construir la matriz C tal que $C_{ik} = \sum_{(j,l)\in S} f_{ij} \cdot d_{kl}$
21: $\mu \leftarrow$ Min(C) $+\ \alpha\ \cdot$ (Max(C)−Min(C)) ▷ Calculamos el umbral
22: **for all** $i, j \in C$ **do** ▷ Construir la LCR
23: **if** C[i,j] $\leq \mu$ **then**
24: LCR ← LCR ∪ $\{\{u_i, l_j\}\}$
25: **end if**
26: **end for**
27: elemento ← SeleccionarParAleatorio(LCR) ▷ Se coge un par unidad, localización aleatorio
28: π[elemento.uni] ← elemento.loc
29: S ← S ∪ {{elemento.uni, elemento.loc}}
30: Borrar todos los elementos de LC con la unidad elemento.uni o con la localización elemento.loc
31: **end for**
32: QAP_cost(π, F, D)
33: **return** π
34: **end function**

Una vez implementado el Greedy Aleatorizado, el código del algoritmo GRASP es muy simple. Es exactamente el mismo que la búsqueda local multiarranque básica, la única diferencia es que la solución de partida no es totalmente aleatoria, sino que es la que devuelve el algoritmo Greedy antes mencionado.

Aquí se muestra su pseudocódigo:

Algorithm 8 Algoritmo GRASP

Input: Recibe las matrices F y D.
Output: Devuelve una solución al QAP

```
1:  function GRASP(F, D)
2:      mejorSolución.coste ← ∞
3:      for i = 1 to 25 do
4:          S ← RandomizedGreedy(F, D)
5:          S' ← BúsquedaLocal(S, F, D)
6:          if S'.coste < mejorSolución.coste then
7:              mejorSolución ← S'
8:          end if
9:      end for
10:     return mejorSolución
11: end function
```

5. Enfriamiento Simulado

En primer lugar definiremos el operador de vecino que utilizaremos para el Enfriamiento Simulado. La idea es la misma que la que se emplea en la búsqueda local: seleccionar dos posiciones de la permutación e intercambiarlas. No obstante, están implementados de manera diferente.

Aquí se muestra el pseudocódigo:

Algorithm 9 Operador de vecino para ES

Input: Recibe una solución y las matrices F y D.
Output: Devuelve una nueva solución al QAP, con dos posiciones intercambiadas

```
1: function NeighbourOP(S, F, D)
2:     out ← S
3:     i, j ← GenerarDosAleatoriosSinRepetición(1, size(out))
4:     out[i] ← out[j]
5:     out[j] ← out[i]
6:     QAP_cost(out, F, D)
7:     return out
8: end function
```

Visto esto, el pseudocódigo del enfriamiento simulado es el siguiente:

Nota, la fórmula empleada para calcular la temperatura inicial es la siguiente, donde μ y ϕ toman los valores de 0.3:

$$T_0 = \frac{\mu \cdot c(S_0)}{-\log \phi}$$

Con respecto al método de enfriamiento, se han probado dos variantes:

- Esquema proporcional: T ← T · 0.99
- Cauchy modificado:

  $$T = \frac{T}{1 + \beta \cdot T}$$

  Donde β se calcula como

  $$\beta = \frac{T_0 - T_f}{(50000/\text{max\_vecinos}) \cdot T_0 \cdot T_f}$$

  .

Algorithm 10 Algoritmo de Enfriamiento Simulado

Input: Recibe una solución (generada aleatoriamente previamente) y las matrices F y D.
Output: Devuelve una solución al QAP

```
1: function ES(S, F, D)
2:     maxVecinos ← 10 · n
3:     maxÉxitos 0,1· maxVecinos
4:     mejorSolución ← S
5:     T_f ← 0.001
6:     T_0 ← CalcularTemperaturaInicial()          ▷ La fórmula empleada es la explicada antes
7:     if T_0 < T_f then                           ▷ Por si la temperatura incial es muy pequeña
8:         T_f ← T_0 · 0,001
9:     end if
10:    T ← T_0
11:    repeat
12:        vecinosActuales ← 0
13:        éxitosActuales ← 0
14:        while vecinosActuales ≤ maxVecinos and éxitosActuales ≤ maxÉxitos do
15:            S_n ← NeighbourOP(S, F, D)
16:            vecinosActuales ← vecinosActuales + 1
17:            Δf ← S_n.coste - S.coste
18:            if Δf < 0 or Uniforme(0,1) ≤ exp(−Δf/T) then
19:                S ← S_n
20:                éxitosActuales ← éxitosActuales + 1
21:                if S.coste < mejorSolución.coste then
22:                    mejorSolución ← S
23:                end if
24:            end if
25:        end while
26:        if éxitosActuales = 0 then
27:            break
28:        end if
29:        T ← Enfriar(T)                          ▷ Cualquiera de las dos anteriores
30:
31:    until T ≤ T_f
32:    return mejorSolución
33: end function
```

6. Iterated Local Search (ILS)

El algoritmo ILS consiste en generar una solución inicial aleatoria y aplicar el algoritmo de búsqueda local sobre ella. Una vez obtenida la solución optimizada, se estudiará si es mejor que la mejor solución encontrada hasta el momento y se realizará una mutación sobre la mejor de estas dos, volviendo a aplicar el algoritmo de búsqueda local sobre esta solución mutada. Este proceso se repetirá un determinado número de veces, devolviéndose la mejor solución encontrada en toda la ejecución. Por tanto, se seguirá el criterio del mejor como criterio de aceptación de la ILS.

Como mutación, se utilizará el operador de modificación por sublista aleatoria de tamaño un cuarto. Aquí se muestran los pseudocódigos del operador de mutación y del algoritmo en sí:

Algorithm 11 Operador de mutación para ILS

Input: Recibe una solución y las matrices F y D.
Output: Devuelve una nueva solución al QAP mutada.

```
1: function MUTATE(S, F, D)
2:     index ← RandInt(1, 3/4·n)          ▷ Así nos aseguramos que nunca queda fuera del vector
3:     offset ← n · 4
4:     out ← S
5:     RandomShuffle(out, index, index+offset)          ▷ Barajamos la sublista
6:     QAP_cost(out, F, D)
7:     return out
8: end function
```

Algorithm 12 Algoritmo ILS

Input: Recibe las matrices F y D.
Output: Devuelve una nueva solución al QAP.

```
1: function ILS(F, D)
2:     GenerateRandom(S, F, D)          ▷ Solución inicial aleatoria
3:     S ← LS(S, F, D)          ▷ Optimizamos
4:     mejorSolución ← S
5:     for i = 1 to 24 do          ▷ 24 porque ya se ha hecho una antes (la primera)
6:         S ← Mutate(S, F, D)          ▷ Mutamos siempre la mejor
7:         S ← LS(S, F, D)          ▷ Optimizamos
8:         if S.coste < mejorSolución.coste then
9:             mejorSolución ← S
10:        end if
11:    end for
12:    return mejorSolución
13: end function
```

7. Hibridación entre ILS y Enfriamiento Simulado

La idea es la misma que antes, solo que en lugar de utilizar búsqueda local para optimizar, utilizamos enfriamiento simulado:

Algorithm 13 Algoritmo ILS-ES

Input: Recibe las matrices F y D.
Output: Devuelve una nueva solución al QAP.

```
1: function ILSES(F, D)
2:     GenerateRandom(S, F, D)          ▷ Solución inicial aleatoria
3:     S ← ES(S, F, D)          ▷ Optimizamos con ES
4:     mejorSolución ← S
5:     for i = 1 to 24 do          ▷ 24 porque ya se ha hecho una antes (la primera)
6:         S ← Mutate(S, F, D)          ▷ Mutamos siempre la mejor
7:         S ← ES(S, F, D)          ▷ Optimizamos con ES
8:         if S.coste < mejorSolución.coste then
9:             mejorSolución ← S
10:        end if
11:    end for
12:    return mejorSolución
13: end function
```

Nota: en este caso, enfriamiento simulado se ha aplicado usando Cauchy modificado como método de enfriamiento.

8. 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 -O3 -march=native
```

9. 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 todos los 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:

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

9.1. Tablas de resultados

Algoritmo GRASP					
Caso	**Desv**	**Tiempo**	**Caso**	**Desv**	**Tiempo**
Chr22a	8.09	0.018341	Sko100a	1.53	3.30826
Chr22b	12.46	8.84568	Sko100f	1.6	3.30176
Chr25a	30.87	0.020363	Tai100a	3.31	2.29113
Esc128	0	5.23298	Tai100b	1.76	3.23962
Had20	0.12	0.017317	Tai150b	6.42	10.8558
Lipa60b	19.72	0.342017	Tai256c	1.07	67.567
Lipa80b	21.31	0.990048	Tho40	2.64	0.105332
Nug28	1.94	0.031819	Tho150	6.17	10.9777
Sko81	1.69	1.48322	Wil50	1.02	0.22941
Sko90	1.63	2.24251	Wil100	0.94	3.38527

Algoritmo BMB					
Caso	**Desv**	**Tiempo**	**Caso**	**Desv**	**Tiempo**
Chr22a	10.36	0.008983	Sko100a	1.1	1.74139
Chr22b	6.23	0.006718	Sko100f	1.58	1.74182
Chr25a	32.03	0.010585	Tai100a	3.69	0.75824
Esc128	0	1.02249	Tai100b	1.26	1.68299
Had20	0.06	0.006946	Tai150b	6.17	2.93706
Lipa60b	19.57	0.143306	Tai256c	1.18	6.08632
Lipa80b	21.19	0.374972	Tho40	3.1	0.054936
Nug28	1.05	0.014594	Tho150	6.75	2.94616
Sko81	1.69	0.779667	Wil50	0.88	0.131937
Sko90	1.94	1.1509	Wil100	0.87	1.74292

Algoritmo ES-Cauchy-mod					
Caso	**Desv**	**Tiempo**	**Caso**	**Desv**	**Tiempo**
Chr22a	27.03	0.001486	Sko100a	2.41	0.752618
Chr22b	14.98	0.00279	Sko100f	2.25	0.701062
Chr25a	55.48	0.005112	Tai100a	3.92	0.470737
Esc128	21.88	0.387393	Tai100b	1.62	0.925475
Had20	0.09	0.001057	Tai150b	3.41	3.02008
Lipa60b	20.55	0.055077	Tai256c	1.75	1.96129
Lipa80b	21.96	0.218106	Tho40	7.95	0.014928
Nug28	4.84	0.00433	Tho150	2.8	2.9279
Sko81	1.59	0.250695	Wil50	2.09	0.049435
Sko90	2.57	0.514087	Wil100	0.73	0.661038

Algoritmo ES-Proporcional					
Caso	**Desv**	**Tiempo**	**Caso**	**Desv**	**Tiempo**
Chr22a	6.08	0.084677	Sko100a	0.68	9.16547
Chr22b	6.1	0.083355	Sko100f	0.52	9.62992
Chr25a	23.34	0.142225	Tai100a	3.62	5.07099
Esc128	0	10.4498	Tai100b	0.74	13.9411
Had20	0.38	0.06363	Tai150b	0.68	53.8057
Lipa60b	19.78	1.00777	Tai256c	0.36	209.357
Lipa80b	21.32	2.61774	Tho40	1.34	0.476187
Nug28	2.21	0.117532	Tho150	0.55	31.0199
Sko81	0.53	4.65257	Wil50	0.22	1.18289
Sko90	0.53	6.45736	Wil100	0.34	8.77119

Algoritmo ILS					
Caso	**Desv**	**Tiempo**	**Caso**	**Desv**	**Tiempo**
Chr22a	8.74	0.005971	Sko100a	1.18	0.523828
Chr22b	8.65	0.004128	Sko100f	1.2	0.528836
Chr25a	31.56	0.005881	Tai100a	2.67	0.480629
Esc128	0	0.851577	Tai100b	3.55	0.614756
Had20	2.05	0.003317	Tai150b	2.28	2.25314
Lipa60b	19.09	0.091695	Tai256c	0.47	6.16093
Lipa80b	20.17	0.243633	Tho40	1.78	0.023844
Nug28	3.21	0.008418	Tho150	1.84	1.94115
Sko81	1.62	0.263633	Wil50	1.09	0.051597
Sko90	2.03	0.370295	Wil100	1.11	0.514747

Algoritmo ILS-ES					
Caso	**Desv**	**Tiempo**	**Caso**	**Desv**	**Tiempo**
Chr22a	8.25	0.081021	Sko100a	1.22	16.6384
Chr22b	8.36	0.045588	Sko100f	1.4	15.2234
Chr25a	30.72	0.087832	Tai100a	3.95	9.5242
Esc128	0	9.88136	Tai100b	1.22	18.9676
Had20	0.12	0.050721	Tai150b	2.16	76.652
Lipa60b	19.87	1.45116	Tai256c	0.99	49.9091
Lipa80b	21.53	3.97279	Tho40	2.69	0.412731
Nug28	1.97	0.120546	Tho150	1.66	71.2383
Sko81	1.08	7.35679	Wil50	0.86	1.15512
Sko90	1.54	11.5397	Wil100	0.66	16.6008

Nota: Los tiempos aparecen en segundos.

9.2. Resultados Globales en el QAP

Algoritmo	**Desv**	**Tiempo**
Greedy	55,11	0,00011
BL	9,11	0,078
GRASP	6,22	6,22428
BMB	6,04	1,16715
ES Cauchy Mod.	9,99	0,64623
ES Proporcional	4,47	18,40486
ILS	5,72	0,74710
ILS-ES	5,51	15,54547

9.3. Conclusión

A la vista de las tablas, podemos afirmar que efectivamente, los algoritmos basados en trayectorias son los que dan unos mejores resultados para el QAP. Ya se venía advirtiendo desde la práctica anterior, donde los algoritmos genéticos apenas lograban superar a la búsqueda local simple, solo sobresaliendo el algoritmo memético, con el que se conseguía una media en la desviación de 7,91 gracias precisamente a la parte de intensificación que introducía la búsqueda local.

Ahora tenemos algoritmos con unos resultado más que satisfactorios, hablando tanto en términos de calidad de las soluciones encontradas, como el tiempo que tardan en dar con ellas.

Llama especialmente la atención como mejora la búsqueda local con simplemente ejecutarla varias veces desde distintas soluciones iniciales (búsqueda multiarranque básica) para finalmente devolver la mejor solución encontrada. Concretamente, disminuye la desviación en aproximadamente tres unidades, y solo estamos pagando como precio un segundo más en el tiempo medio que tarda en ejecutarse. Esta mejora tan sumamente sencilla, está logrando para este problema unos mejores resultados que el algoritmo GRASP, el cual incorpora una gran cantidad de conocimiento sobre el QAP gracias al Greedy Aleatorizado que emplea. No obstante, puede deberse a la casuística de este problema, y no ser así para otros.

Con respecto al Enfriamiento Simulado, decidí aplicar el criterio de enfriamiento de temperatura proporcional porque no estaba nada satisfecho con los resultados que mostraba el de Cauchy modificado. Este último criterio enfria demasiado rápido (y como justificación tenemos el poco tiempo que tarda en ejecutarse, además de la calidad de los resultados, prácticamente peores que la búsqueda local simple), por lo que estaba llegando velozmente a la etapa de convergencia del algoritmo, y no estaba dando la suficiente importancia a la etapa incial en la que se introduce diversidad en las soluciones. En contrapartida, posiblemente el utilizar 0,99 como α en el enfriamiento propocional quizás es pasarse un poco, ralentizando el algoritmo, ya que los tiempos se disparan hasta casi los 4 minutos en el problema Tai256c y 18 segundos de media. No obstante creo que merece la pena. Es cierto que no tarda milisegundos como el resto de algoritmos, pero 4 minutos continua siendo un tiempo razonable, y los resultados obtenidos son brillantes, de hecho, constituyen el mejor algoritmo implementado para el QAP.

Con respecto al ILS, también destaca por su sencillez y sus buenos resultados, situándolo como el segundo mejor algoritmo. En la hibridación de este último junto con el algoritmo de Enfriamiento Simulado (para el que se ha utilizado únicamente el criterio de Cauchy modificado), vemos que conseguimos reducir un 0,2 la varianza con respecto a la aplicación de la búsqueda local. Sin embargo, hemos pagado un alto precio, ya que el tiempo se ha disparado de menos de un segundo de media, a 15 segundos. Realmente no compensa para la poca mejora obtenida. No obstante, y aunque no se ha probado, si se utilizase el criterio propocional para optimizar las soluciones mutadas, posiblemente obtendríamos mejores soluciones que el algoritmo de ES en solitario, pero ahora sí que tendríamos un aumento drástico del tiempo (calculo, aproximadamente 25 veces lo que tarda de media el ES, es decir, 1 hora y 30 minutos para el problema de tai256c), por lo que sería necesario estudiar si las mejoras son palpables y realmente merece la pena el cambio.

Para terminar simplemente comentar algunos detalles sobre los resultados:

- Todos los algoritmos (a excepción del ES con Cauchy modificado), encuentran el óptimo para el mapa Esc128 (mientras que los algoritmos de prácticas anteriores fallaban)

- Como en el resto de prácticas, se siguen manteniendo los mismos valores atípicos: Chr25a, Lipa60b y Lipa80b.