**Metaheurística**
Doble Grado en Ingeniería Informática y Matemáticas
Universidad de Granada

---

# Práctica 1

## Algorítmos basados en trayectorias en Problema de Asignación Cuadrática(QAP)

---

Iván Sevillano García

DNI: 77187364-P

E-mail: ivansevillanogarcia@correo.ugr.es

Grupo del martes, 17:30h-19:30h

6 de junio de 2018

# Índice

# 1. Descripción del problema QAP

El problema que se nos plantea es el Problema de Asignación Cuadrática(en adelante QAP por las siglas). En él, tenemos una serie de instalaciones las cuales tienen que interactuar entre ellas una cierta çantidad de trabajo", produciendo un coste. Las instalaciones tienen, además, unas determinadas localizaciones en las que se tienen que situar. Dependiendo de la distancia entre cada dos instalaciones, el coste que producen al interactuar es mayor o menor.

Cabe destacar una serie de detalles en el problema:

- **No Euclideo(No Métrico).** Este problema no pone ninguna objeción a que la distancia de una localización de un lugar a si mismo sea mayor que cero. Además, una instalación puede interactuar consigo misma por consiguiente.
- **No simétrico.** Tampoco pone objeción a que la distancia de una localización a otra no sea la misma que de otra a una.

El problema entonces consiste en repartir las instalaciones en las localizaciones de forma que el coste total o trabajo sea mínimo. Esto es fácil de representar con una permutación, que a cada instalación $i$ le asigna una localización $\pi(i)$. Si llamamos $d_{ij}$ a la distancia que hay de la localización $i$ a la $j$, y $f_{ij}$ la cantidad de trabajo que tiene que mandar la instalación $i$ a la $j$, la función de coste asociada al problema sería la siguiente:

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

Si consideramos las matrices de distancias y flujos, $D$ y $F$ y las matrices de que representan a cada permutación y su inversa, $\Pi$ y $\Pi^{-1}$ respectivamente, se puede representar el coste de una manera más sencilla:

$$Coste(\pi) =< F, \Pi^{-1}D\Pi >$$

Donde la aplicación $<,>$ tiene como argumentos dos matrices de mismas dimensiones y se aplica en la suma de la multiplicación de sus componentes una a una.

# 2. Breve descripción de los algorítmos utilizados.

En esta sección vamos a explicar brevemente la configuración de una solución concreta del problema, el funcionamiento de la función de coste y de cómo trabajan los dos algoritmos que se describen en el enunciado.

## 2.1. Solución.

Una solución está perfectamente determinada por un vector de $N$ componentes donde cada una de sus componentes es distinta unas de otras y el rango de valores difiere de 1 hasta N, o lo que es lo mismo, la solución es una permutación. Según la misma, la localización $i$ tendrá alojada la instalación con el número que ocupa en el vector la posición $i$. Para agilizar cálculos, cada permutación, además, guarda su coste asociado. Así, será fácil calcular soluciones vecinas.

## 2.2. Función de coste.

La función de coste es la descrita anteriormente en la primera sección. A continuación el psudo-código. Los parámetros hacen referencia a la matriz de distancias(D), la matriz de flujos(F) y a la permutación que representa nuestra solución:

```
Parametros:D,F,P
Output: coste
coste = 0
Para cada elemento de la matriz (i,j):
  coste <- coste + F[j][i]*D[P(j)][P(i)]
```

## 2.3. Algoritmos basados en trayectorias.

Esta práctica consistirá en desarrollar cuatro algoritmos basado en trayectorias y un algoritmo híbrido entre dos de ellos. Veamos ahora las descripciones de cada uno de ellos:

### 2.3.1. Búsqueda multiarranque básico.

Este algorítmo consiste en utilizar la ya conocida búsqueda local a partir de varios puntos o soluciones de arranque. Puesto que condición de parada son las mismas evaluaciones, se realizarán menos evaluaciones en cada una de las búsquedas locales del algoritmo.

### 2.3.2. Enfriamiento simulado(ES).

Este algoritmo se basa en cómo se comportan una partícula dentro de un metal al enfriarse el mismo. Al principio, esta partícula tiene mucha energía por la alta temperatura, por lo que se mueve mucho aunque vaya a posiciones con más ."$^{\text{en}}$ergía". Conforme se va enfriando el metal, lo mismo le pasa a la partícula y tiene menos energía, por lo que los saltos que

hace se empiezan a parecer más a una búsqueda por descenso, intentando ir a posiciones con cada vez menos energía. Pasamos a describir cada una de las características:

- **Aceptación de soluciones.** Este algoritmo se basa en poder aceptar soluciones de peor calidad dependiendo de la temeratura actual. Así, una solución será aceptada cuando:

  - La solución S' es mejor que la solución S actual con una probabilidad de 1.
  - La solución S' es peor que la solución S actual con una probabilidad de $e^{\frac{-dC}{T_i}}$, donde $dC$ es la diferencia de coste entre la solución actual S y S'. $dC$ es un número positivo ya que S' no mejora a la solución actual. $T_i$ es la temperatura actual. Por tanto, la expresión antes descrita es un número entre 0 y 1 que podremos usar como probabilidad. Además se pueden hacer varias observaciones.

    La primera es que si la temperatura es muy alta la probabilidad con la que vamos a aceptar una solución es prácticamente 1. Si es muy baja, es casi 0.

    La segunda, que conforme va avanzando el algoritmo, $dC$ empieza a tener más importancia. Tanto cuando la temperatura es muy alta como cuando es muy baja, $dC$ no afecta demasiado a la aceptación de la solución. Sin embargo, en mitad de la ejecución si es más relevante. Se aceptarán soluciones dependiendo de cuanta diferencia de coste hay entre soluciones.

- **Esquema de enfriamiento.** En cada etapa del algoritmo modificaremos la temperatura de la siguiente manera:

  $$T_{k+1} = \frac{T_k}{1 + \beta T_k}, \beta = \frac{T_0 - T_f}{M T_0 T_f}$$

  Donde M es el número de enfriamientos. Este es distinto a número de evaluaciones de la función objetivo. Se harán, dentro de una misma etapa de enfriamiento, varias evaluaciones de la función objetivo.

- **Operador de vecino.** Se utiliza el mismo operador de vecino que en la búsqueda local de la primera práctica. En cada iteración se explorará sólo un vecino que pueda sustituir a la solución actual.

- **Condición de enfriamiento.** La condición que se quiere considerar en la práctica para enfriar es haber alcanzado un máximo de evaluaciones de nuevos vecinos o haber alcanzado un máximo de vecinos aceptados.

- **Condición de parada.** El algoritmo finalizará si se llega al máximo de evaluaciones o si en la iteración actual no ha encontrado ninguna solución que hayamos

aceptado. Esta última consideración se impone en la práctica pero veremos que pasa si la suprimimos.

Los valores de los parámetros escogidos en esta práctica son los siguientes:

$$T_0 = \frac{\mu C(S_0)}{-ln(\phi)}, T_f = 10^{-3}, \mu = \phi = 0{,}3$$

Se debe comprobar que la temperatura final es menor que la inicial. Como número máximo de vecinos en cada enfriamiento se ha escogido 10 veces el tamaño del problema. Como número de éxitos, la décima parte de este número. Así, el número de enfriamientos será $50000/max\_vecinos$.

### 2.3.3. Búsqueda local iterativa(ILS).

En este algoritmo ejecutaremos de forma repetida una búsqueda local. Cada cierto tiempo, modificaremos una parte significativa de la solución generando una solución vecina más lejana a la vecina de la búsqueda local, y volveremos a aplicar sobre esta nueva solución la búsqueda local.

La necesidad de encontrar un vecino más alejado es porque en la búsqueda local ya utilizamos un vecino para descender. Si modificásemos la solución sólo en una trasposición, la búsqueda local nos dará la misma solución. Por ello, en la práctica se nos da la siguiente solución:

- **Mutación por sublista.** Por este operador, escogeremos una posición aleatoria $i$ y mezclaremos todos los elementos entre $i$ y $i+t$, siendo t el tamalo de la sublista que iremos a mezclar. En caso de que la lista sea menor que $i+t$, mezclaremos también los índices que falten del principio de la lista. Para mezclar esta lista, utilizaremos el operador $shuffle$ de la librería $random$ de $python$.
- **Mutación por $t$ índices distintos.** En la práctica hemos modificado este operador por el de modificar $t$ índices distintos, sin tener estos que estar contiguos. Esto lo hacemos porque en el QAP la relación entre dos posiciones cualquiera es la misma que la de dos posiciones contiguas. Nuestra forma de generar soluciones lejanas estaría sesgada en tal caso.

El tamaño de sublista en nuestro caso entonces será de $t = n/4$, siendo n el tamaño del problema. Aplicaremos el método ILS a 25 soluciones, la primera vez a una solución aleatoria y las siguientes 24 a soluciones mutadas. Mutaremos siempre la mejor solución encontrada.

### 2.3.4. Búsqueda por procedimiento greedy aleatorizado(GRASP).

Este algoritmo se basa en crear soluciones greedy aleatoria y aplicarle la búsqueda local a esta. La solución greedy general es determinista en el sentido de que en cada paso de la creación de la solución, incrusta en la solución el mejor valor. En GRASP, se mete uno de los mejores de forma aleatoria. Así nos aseguramos tener soluciones parcialmente buenas y distintas unas de otras para poder escoger soluciones más diversas sobre las que partir y aplicar la búsqueda local.

A continuación se describe la forma de crear soluciones greedy aleatorias iniciales:

- **Primeras dos ciudades.** A nuestra solución todavía por construir comenzamos metiendo, de forma aleatoria, dos de los elementos con mejor puntuación greedy. Esto significa que ordenamos las localizaciones por mejor valor considerado en el algoritmo greedy. Seleccionamos las mejores y, de esas, escogemos dos.
- **Siguientes ciudades.** Para las siguientes, se ordenan por el coste relativo de insertar en una localización una instalación concreta. Esto es, cuanto coste añade esta asignación con respecto a las instalaciones ya asignadas:

$$C_{rel}(r,s) = \sum_{i \in S_p} f_{r,i} d_{s,S_p(i)} + f_{i,r} d_{S_p(i),s}$$

  Se sigue el mismo criterio para seleccionar a las mejores y, de estas, se escoge una asignación aleatoria y se aplica a la solución parcial $S_p$.

Las mejores asignaciones a las que nos referimos en la creación de la solución parcial son las que añadan como mucho una cantidad un 30 % peor que la mejor de todas las anteriores.

### 2.3.5. Algoritmo híbrido. ILS-ES

Esta hibridación utiliza el mismo método de ILS pero con la diferencia de que utiliza el enfriamiento simulado en vez de la búsqueda local. Utilizaremos la misma estructura que en los dos algoritmos por separado con la diferencia de que en el enfriamiento simulado habrá menos pasos de enfriamiento, ya que tendrán menos pasos en cada iteración.

# 3. Pseudocódigos y explicaciones.

A continuación se detallan los pseudocódigos de los algoritmos immplementados:

## 3.1. BMB.

La búsqueda local que utilizamos es la explicada en la primera práctica (DLB).

---

```
Generamos una solucion aleatoria S
Le aplicamos la busqueda local con 5000 pasos.
De 0 a 24:
  Genereamos una solucion aleatoria S'
  Le aplicamos la busqueda local a S'
  Si el coste de S' es menor que la de S
    S = S'

Devolvemos S
```

---

## 3.2. ES.

Para este algoritmo necesitamos una variable aleatoria U(0,1) a la que llamaremos con *random*.

---

```
Generamos S una solucion.
Calculamos los parametros necesarios maxExitos, maxVecinos, M, t0,
tf y beta descritos anteriormente.
T = t0
Mientras T sea mayor que tf y se hayan realizado exitos
  exitos = generados = 0
  Mientras exitos < maxExitos y generados < maxVecinos
    Generamos un vecino S' de S.
    generados = generamos + 1
    difCoste = Coste(S')-Coste(S)
    Si difCoste < 0 o random() < exp(-difCoste/T)
      S = S'
      exitos = exitos + 1
  Actualizamos la temperatura actual T = T/(1+beta*T)

Devolvemos la mejor solucion que hayamos obtenido.
```

## 3.3. ILS.

En este algoritmo hemos tenido que programar, además del algoritmo, una función de mutación que pasamos a describir en pseudocódigo:

---

```
P es la permutacion a mutar.
Mezclamos con shuffle una lista de indices y cogemos los n/4 primeros, L1.
Sacamos y guardamos en L2 los indices marcados de P y los sustituimos por -1 e
Mezclamos con shuffle la lista L2.
Sustituimos los -1 en P por los indices de L2 en orden.
```

---

El algoritmo ILS:

---

```
Generamos una solucion S aleatoria
Le aplicamos la busqueda local con 50000/25 pasos.
Durante 24 veces:
  Mutamos la solucion S y la guardamos en S'.
  Aplicamos busqueda local a S' con 50000/25 pasos.
  Si la solucion S' tiene mejor coste que S
    S = S'

Devolvemos S'
```

## 3.4. GRASP.

Este algoritmo tiene varias funciones auxiliares que ayudan a determinar cómo construir la solución greedy aleatorizada. Las funciones a explicar son las siguientes:

**Función** *OrdenSuma*. Ordena las diagonales de una matriz por cuanto vale la suma de su columna y su fila. Devuelve un vector con los índices ya ordenados junto con su coste. Utilizamos la función *sort* de python para este último propósito.

---

```
M matriz
L = {} lista de los indices con sus costes
Para i de 0 hasta N
  sumaI = 0
  Para cada elemento de la columnna o la fila que contiene a i,j
    sumaI = sumaI + M[i][j](elemento de la fila)
    sumaI = sumaI + M[j][i](elemento de la columna)
  L <- (i,sumaI)
```

```
Devolvemos la lista ordenada por el valor de sumaI.
```

---

La segunda función auxiliar se encarga de calcular cuanto añade de costo a la solución parcial $S_p$ con las condiciones iniciales de matrices de flujo y de distancia $F, D$ respectivamente, la asignación de una instalación $r$ no asignada a una localización $s$ no utilizada dentro de la solución:

---

```
F,D,Sp,r,s valores de entrada.
coste = 0
Para cada localizacion que ya tenga instalacion asignada
  // Aniadimos a coste lo que cuesta colocar (r,s) con respecto a (i,Sp[i])
  coste = coste + F[r][i]*D[s][Sp[i]]+F[i][r]*D[Sp[i]][s]

Devolvemos coste
```

---

La tercera función se encarga de ordenar las posibles inserciones de una instalación no ubicada en $S_Pp$ en una ubicación no utilizada por $S_p$ ordenadas por el coste de la segunda función auxiliar.

---

```
Tenemos Sp una solucion parcial
L1 = [] lista de posibles inserciones con sus costes
Para cada instalacion i no asignada
  Para cada localizacion j no utilizada
    Aniadimos a L1 la asignacion de la instalacion i en la localizacion j
    junto con su coste

Devolvemos la lista L1 por los valores antes calculados.
```

---

El pseudocódigo del algoritmo de una iteración del algoritmo GRASP se detalla a continuación. Esta llamada la realizaremos 25 veces, es decir, generaremos 25 soluciones GRASP. El método de escoger dos elementos será mezclar la lista de los mejores y escoger los dos primeros elementos:

---

```
Solucion vacia S.
Obtenemos los valores ordenados de las diagonales de la matriz F,Fl
Obtenemos los valores ordenados de las diagonales de la matriz D,Dl
Invertimos Dl(Ya que queremos distancias cortas para flujos grandes)
Calculamos el umbral de mejores uM y peores uP de cada lista.
```

```
Nos quedamos en Dl con los indices cuyo valor no sobrepase
  uM+0.3(uP-uM)
Nos quedamos en Fl con los indices cuyo valor no rebaje
  uM-0,3*(uM-uP)

Escogemos aleatoriamente dos elementos de cada lista, f1, f2, d1, d2
Aniadimos a la solucion las asignaciones (f1,d1), (f2,d2)

Para el resto de posibles asignaciones:
  cand es la lista de asignaciones ordenadas calculada para la
    solucion parcial S.
  Calculamos los umbrales uM, uP para S
  Nos quedamos en cand las asignaciones cuyo valor
    no sobrepase uM+0.3*(uM-uP)
  Escogemos aleatoriamente una asignacion de cand y
  la aniadimos a la solucion S

Le aplicamos a S la busqueda local con 50000/25 pasos
```

---

### 3.5. ILS-ES.

Como hemos dicho, este algoritmo tiene la misma estructura que el ILS pero, en vez de utilizar la búsqueda local, utilizamos enfriamiento simulado para optimizar la solución mutada.

---

```
Generamos una solucion S aleatoria
Le aplicamos la busqueda local con 50000/25 pasos.
Durante 24 veces:
  Mutamos la solucion S y la guardamos en S'.
  Aplicamos enfriamiento simulado a S' con 50000/25 pasos.
  Si la solucion S' tiene mejor coste que S
    S = S'

Devolvemos S'
```

---

# 4. Algoritmo de comparación.

En esta sección se compararán los resultados de los algoritmos Greedy y búsqueda local. Para dicha comparación, atenderemos a dos estadísticos:

- **Desviación a la solución óptima.** Este estadístico evaluará proporcionalmente la diferencia de coste de la mejor solución y la solución obtenida en cada instancia.La fórmula que describe este estadístico es el siguiente:

$$Desv = \frac{1}{|casos|} \sum_{i \in casos} 100 \frac{valorAlg_i - mejorVal_i}{mejorVal_i}$$

- **Tiempo medio de ejecución.** La media de los tiempos de ejecución de cada algoritmo.

Consideraremos entonces que un algoritmo es mejor que otro si la desviación a la solución óptima es menor. En caso de ser iguales, el algoritmo que tarde menos será considerado mejor.

# 5. Manual de uso.

Para la implementación de la práctica hemos utilizado el lenguaje precompilado Python3, por lo que debe de estar instalado en el sistema. También hemos hecho uso de la biblioteca random del mismo lenguaje, más en concreto de las funciones randint(inicio,fin), que escoge aleatoriamente un valor entero entre *inicio* y $fin - 1$, y shuffle(vector), que mezcla el vector que se le pasa como argumento.

La forma de utilizar el programa es la siguiente:

- **Programas.** Esta practica tiene un total de cinco ejecutables nuevos. Cada ejecutable recibe como argumento el nombre del archivo de entrada con los datos(terminado en '.dat'), que debe de estar en el directorio *./qapdata/*. Tiene que haber también un archivo solución con el mismo nombre pero con terminación '.sln' en el directorio *./qapsoln/*. Es posible también introducir una semilla como segundo argumento y un fichero de salida:

  *./nombreAlgoritmo.py ./qapdata/nombredatos.dat semilla* $fichero_salida$

  Nos dará como resultado:

  - Solución Algoritmo. La primera linea será la solución del algoritmo en cuestión obtenida. La segunda, cuanto tiempo ha tardado el programa en calcularla y cuanto coste total tiene esta solución.
  - Mejor solución. La primera linea es la configuración de la mejor solución y la siguiente es su coste.

- **Experimento total.** Para ejecutar el experimento completo, nos hemos ayudado de un script que ejecuta todos los casos de prueba y un makefile que lo llama. Así, la forma de obtener todas las soluciones será tan simple como ejecutar el siguiente comando:

  *make all*

  Si se quiere ejecutar un algoritmo completo sin los demás, hay que ejecutar:

  *make todoAlgoritmo*

  Reemplazando *Algoritmo* por el nombre del algoritmo. También se pueden obtener los estadísticos ejecutando el siguiente comando después del comando anterior:

*make estadisticosAlgoritmo*

Es importante mencionar que dentro de la ejecución de los programas se les ha introducido una semilla en concreto, 200000, que no modificaremos.

# 6. Experimento y análisis de resultados.

Las tablas obtenidas para cada algoritmo son las siguientes:

**Algoritmo BMB**

| Caso | MejorCoste | Coste obtenido | Desviación | tiempo |
|---|---|---|---|---|
| had20 | 6156 | 6724 | 9.22 | 2.12 |
| chr22a | 6194 | 6666 | 7.62 | 2.07 |
| chr22b | 3796 | 4378 | 15.33 | 3.37 |
| chr25a | 64 | 114 | 78.12 | 31.93 |
| nug28 | 6922 | 6926 | 0.05 | 1.98 |
| tho40 | 2520135 | 3041614 | 20.69 | 15.07 |
| wil50 | 7763962 | 9544338 | 22.93 | 20.57 |
| lipa60b | 5166 | 5228 | 1.2 | 4.67 |
| lipa80b | 90998 | 98294 | 8.01 | 20.75 |
| sko81 | 115534 | 125448 | 8.58 | 23.34 |
| sko90 | 152002 | 166146 | 9.3 | 25.46 |
| sko100a | 149036 | 162588 | 9.09 | 25.45 |
| sko100f | 21052466 | 22260648 | 5.73 | 25.76 |
| tai100a | 1185996137 | 1374946874 | 15.93 | 26.32 |
| tai100b | 498896643 | 594443569 | 19.15 | 39.57 |
| wil100 | 44759294 | 49791988 | 11.24 | 68.96 |
| esc128 | 240516 | 250658 | 4.21 | 10.11 |
| tai150b | 8133398 | 9149516 | 12.49 | 40.09 |
| tho150 | 48816 | 49638 | 1.68 | 12.67 |
| tai256c | 273038 | 287216 | 5.19 | 25.77 |

Tabla 6.1: Datos obtenidos para el algoritmo BMB

**Algoritmo ES**

| Caso | MejorCoste | Coste obtenido | Desviación | tiempo |
|---|---|---|---|---|
| had20 | 6156 | 6862 | 11.46 | 0.12 |
| chr22a | 6194 | 7634 | 23.24 | 0.08 |
| chr22b | 3796 | 6956 | 83.24 | 0.24 |
| chr25a | 64 | 78 | 21.87 | 7.88 |
| nug28 | 6922 | 6938 | 0.23 | 0.14 |
| tho40 | 2520135 | 3044214 | 20.79 | 1.25 |
| wil50 | 7763962 | 9438677 | 21.57 | 4.63 |
| lipa60b | 5166 | 5388 | 4.29 | 0.37 |
| lipa80b | 90998 | 92866 | 2.05 | 5.83 |
| sko81 | 115534 | 117570 | 1.76 | 7.22 |
| sko90 | 152002 | 155124 | 2.05 | 12.21 |
| sko100a | 149036 | 152106 | 2.05 | 10.04 |
| sko100f | 21052466 | 21914850 | 4.09 | 7.55 |
| tai100a | 1185996137 | 1252179834 | 5.58 | 14.45 |
| tai100b | 498896643 | 513278727 | 2.88 | 36.83 |
| wil100 | 44759294 | 45598736 | 1.87 | 19.46 |
| esc128 | 240516 | 253326 | 5.32 | 0.68 |
| tai150b | 8133398 | 8314986 | 2.23 | 34.21 |
| tho150 | 48816 | 49876 | 2.17 | 1.42 |
| tai256c | 273038 | 276036 | 1.09 | 14.27 |

Tabla 6.2: Datos obtenidos para el algoritmo ES

**Algoritmo ILS**

| Caso | MejorCoste | Coste obtenido | Desviación | tiempo |
|---|---|---|---|---|
| had20 | 6156 | 6428 | 4.41 | 1.57 |
| chr22a | 6194 | 6610 | 6.71 | 1.61 |
| chr22b | 3796 | 4374 | 15.22 | 2.69 |
| chr25a | 64 | 104 | 62.5 | 26.58 |
| nug28 | 6922 | 6922 | 0.0 | 1.29 |
| tho40 | 2520135 | 2685111 | 6.54 | 13.01 |
| wil50 | 7763962 | 9531633 | 22.76 | 18.25 |
| lipa60b | 5166 | 5258 | 1.78 | 3.42 |
| lipa80b | 90998 | 96858 | 6.43 | 18.54 |
| sko81 | 115534 | 123836 | 7.18 | 21.74 |
| sko90 | 152002 | 164162 | 7.99 | 22.61 |
| sko100a | 149036 | 160264 | 7.53 | 22.44 |
| sko100f | 21052466 | 22292828 | 5.89 | 22.47 |
| tai100a | 1185996137 | 1294015153 | 9.1 | 22.73 |
| tai100b | 498896643 | 574223346 | 15.09 | 36.0 |
| wil100 | 44759294 | 48720216 | 8.84 | 59.34 |
| esc128 | 240516 | 248288 | 3.23 | 8.85 |
| tai150b | 8133398 | 8994402 | 10.58 | 35.61 |
| tho150 | 48816 | 49302 | 0.99 | 11.41 |
| tai256c | 273038 | 284826 | 4.31 | 22.94 |

Tabla 6.3: Datos obtenidos para el algoritmo ILS

**Algoritmo GRASP**

| Caso | MejorCoste | Coste obtenido | Desviación | tiempo |
|---|---|---|---|---|
| had20 | 6156 | 6908 | 12.21 | 2.33 |
| chr22a | 6194 | 7082 | 14.33 | 2.3 |
| chr22b | 3796 | 5290 | 39.35 | 3.68 |
| chr25a | 64 | 128 | 100.0 | 257.36 |
| nug28 | 6922 | 7122 | 2.88 | 1.86 |
| tho40 | 2520135 | 3049821 | 21.01 | 24.76 |
| wil50 | 7763962 | 9597892 | 23.62 | 53.46 |
| lipa60b | 5166 | 5470 | 5.88 | 4.52 |
| lipa80b | 90998 | 100286 | 10.2 | 54.78 |
| sko81 | 115534 | 125894 | 8.96 | 81.5 |
| sko90 | 152002 | 168410 | 10.79 | 118.38 |
| sko100a | 149036 | 161862 | 8.6 | 116.73 |
| sko100f | 21052466 | 22554300 | 7.13 | 122.89 |
| tai100a | 1185996137 | 1471648942 | 24.08 | 129.37 |
| tai100b | 498896643 | 592341248 | 18.73 | 512.6 |
| wil100 | 44759294 | 51406510 | 14.85 | 3910.65 |
| esc128 | 240516 | 261158 | 8.58 | 12.51 |
| tai150b | 8133398 | 9150920 | 12.51 | 502.23 |
| tho150 | 48816 | 50132 | 2.69 | 17.91 |
| tai256c | 273038 | 287414 | 5.26 | 111.13 |

Tabla 6.4: Datos obtenidos para el algoritmo GRASP

**Algoritmo ILS-ES**

| Caso | MejorCoste | Coste obtenido | Desviación | tiempo |
|---|---|---|---|---|
| had20 | 6156 | 6456 | 4.87 | 5.32 |
| chr22a | 6194 | 6742 | 8.84 | 5.23 |
| chr22b | 3796 | 5238 | 37.98 | 5.84 |
| chr25a | 64 | 98 | 53.12 | 59.34 |
| nug28 | 6922 | 6922 | 0.0 | 4.44 |
| tho40 | 2520135 | 3043180 | 20.75 | 19.21 |
| wil50 | 7763962 | 9547423 | 22.97 | 26.83 |
| lipa60b | 5166 | 5264 | 1.89 | 7.14 |
| lipa80b | 90998 | 94486 | 3.83 | 28.52 |
| sko81 | 115534 | 120792 | 4.55 | 34.67 |
| sko90 | 152002 | 161208 | 6.05 | 32.19 |
| sko100a | 149036 | 157370 | 5.59 | 32.24 |
| sko100f | 21052466 | 22370186 | 6.25 | 33.07 |
| tai100a | 1185996137 | 1273972590 | 7.41 | 34.25 |
| tai100b | 498896643 | 544843315 | 9.2 | 75.7 |
| wil100 | 44759294 | 49180730 | 9.87 | 137.33 |
| esc128 | 240516 | 251164 | 4.42 | 10.65 |
| tai150b | 8133398 | 8707874 | 7.06 | 73.11 |
| tho150 | 48816 | 49952 | 2.32 | 13.92 |
| tai256c | 273038 | 280364 | 2.68 | 34.88 |

Tabla 6.5: Datos obtenidos para el algoritmo ILS-ES

Por último, la tabla de comparación de algoritmos:

| Algoritmo | Desv | Tiempo |
|---|---|---|
| Greedy | 59.73 | 0,00 |
| BL | 8.34 | 13.45 |
| BMB | 13.29 | 21.31 |
| ES | 10.99 | 8.94 |
| ILS | 10.35 | 18.65 |
| GRASP | 19.67 | 309.15 |
| ILS-ES | 10.98 | 33.69 |

Tabla 6.6: Tabla de comparación

## 6.1. Análisis de resultados

Analizando los estadísticos observamos que ninguno de ellos supera a la búsqueda local en el estadístico Desv. Puesto que algunas de estas metaheurísticas utilizan por debajo una búsqueda local se nos plantea si realmente son buenas estas nuevas formas de abarcar nuestro problema.

Si observamos las instancias de tamaño n pequeño, el algoritmo de multi arranque básico mejora siempre a la solución de la búsqueda. Esto es debido a que el número máximo de evaluaciones sobrepasa de lejos la cantidad de pasos que necesita la búsqueda local para estancarse y no poder encontrar nuevas soluciones. Esto hace que la técnica multi arranque básica sea repetir búsquedas locales casi con la misma cantidad de pasos. Sin embargo, en tamaños más grandes, apenas llegan a conseguir una profundidad adecuada.

Si comparamos el enfriamiento simulado con la búsqueda local, vemos que no tenemos el mismo patrón. Se obtienen mejores o peores resultados de forma aleatoria. Sin embargo, en media, obtiene peores resultados. Cabe destacar también que tiene menos tiempo de ejecución.

En el algoritmo ILS pasa algo parecido a la búsqueda multi arranque básica. En instancias con un tamaño pequeño, la diversificación de las soluciones y la poca profundidad del problema hace que tenga muy buenos resultados, incluso llegando a conseguir la solución óptima. Sin embargo, en tamaños de solución más grande, la diversidad de soluciones no compensa a las evaluaciones de menos de cada búsqueda local. Aun así, en algunos tamaños grandes se consigue igualar o superar por poco a la solución de la búsqueda local.

GRASP

ILS-ES

## 6.2. Posibles mejoras de los algoritmos

- Algo