Práctica 2

Búsqueda por Trayectorias

Problema de la Asignación Cuadrática (QAP)

Metaheurísticas Curso Académico 2016/2017

Algoritmos considerados:

- Greedy
- Búsqueda local del primer mejor (con Don't Look Bits)
- Enfriamiento Simulado
- Búsqueda Multiarranque Básica
- GRASP
- Búsqueda local reiterada (ILS)
- Hibridación de ILS y ES (ILS-ES)

Nombre: David Criado Ramón

D.N.I.:

Correo electrónico: davidcr96@correo.ugr.es

Grupo: MH3 (lunes 17:30-19:30)

1. Descripción del problema

El problema tratado en esta práctica es el clásico **Problema de la Asignación Cuadrática *(Quadratic Assigment Problem)***. El problema consiste en *asignar n unidades a n localizaciones*, y nuestro objetivo es minimizar el producto de dos variables conocidas: la distancia (coste de ir de una unidad a otra) y el flujo (medida de tráfico entre las unidades). Para simplificar el estudio realizado la versión estudiada en esta práctica parte de un número igual de localizaciones y unidades y hemos de asignarlas todas.

Así, pues nuestro objetivo viene determinado por:

$$\min_{s \in \Pi} \sum_{i=1}^{n} \sum_{j=1}^{n} f_{ij}\, d_{S(i)S(J)}$$

Existen ciertas posibles aplicaciones de este problema en la vida real como podrían ser el diseño de hospitales, el cableado óptimo de placas madre o cualquier otra relación entre dos variables, una que mide un coste unitario y otra que mide la frecuencia de uso para las variables asignadas.

2. Breve descripción de los algoritmos empleados al problema

2.1 Representación de la solución

La representación de la solución al problema viene determinada por una permutación almacenada en un vector donde el índice de la posición indica la unidad a colocar y el dato guardado en la misma permutación indica la localización a usar. Junto a la permutación solución guardamos su coste asociado.

Durante la práctica se utilizan dos maneras de calcular el coste:

2.1.1 Cálculo directo del coste

Esta es la manera de calcular el coste habitualmente (exceptuando en el uso del 2-OPT). En ella aplicamos directamente la fórmula presentada en el apartado anterior

```
// Siendo S el vector solución a evaluar, flujo(a,b) el flujo de la unidad a la
// unidad b y distancia(a,b) la distancia de a a b

coste = 0
Para cada i desde 1 hasta n:
  Para cada j desde 1 hasta n:
      coste = coste + flujo(i,j) + distancia(S[i], S[j])
Devolver coste
```

2.1.2 Cálculo factorizado del coste

Esta es la manera utilizada para calcular el coste en el 2-OPT (*este operador se encuentra explicado en la sección 2.3).* Puesto que el 2-OPT intercambia dos posiciones sólo realizamos las cuentas asociadas directamente a los valores asociados al intercambio recalculando los costes de los arcos con las posiciones adyacentes a las posiciones a intercambiar conforme a la siguiente fórmula:

$$\sum_{k=1}^{n}[f_{rk}(d_{S(s)S(k)} - d_{S(r)S(k)}) + f_{sk}(d_{S(r)S(k)} - d_{S(s)S(k)}) + f_{kr}(d_{S(k)S(s)} - d_{S(k)S(r)}) + f_{ks}(d_{S(k)S(r)} - d_{S(k)S(s)})]$$

```
// Siendo S la solución si haber realizado el intercambio, flujo y distancia con
// el mismo significado que en la sección anterior, con la precondición de que
// el coste de S esté calculado en la variable coste y las variables r y s las
// posiciones a intercambiar

Para cada k desde 1 hasta n:
  S.coste = coste + flujo(r,k) * (distancia(p[s],p[k]) - distancia(p[r],p[k])
                  + flujo(s,k) * (distancia(p[r],p[k]) - distancia(p[s],p[k])
                  + flujo(k,r) * (distancia(p[k],p[s]) - distancia(p[k],p[r])
                  + flujo(k,s) * (distancia(p[k],p[r]) - distancia(p[k],p[s])
```

2.2 Generación de soluciones aleatorias

Para obtener una solución inicial o una población inicial completa se utilizan las siguientes formas de crear aleatorios. Puesto que se inicializan los mecanismos generadores de

aleatorios dentro de los propios métodos para una misma semilla se obtendrá siempre la misma solución inicial en la búsqueda local (única que usa sólo una solución aleatoria inicial) o la misma población en todas las versiones genéticas (en la que el primer elemento de la población sin ordenar será la permutación generada en una única solución aleatoria). Además, en la versión en la que se genera una población me aseguro de que todas las soluciones de la población aleatoria sean distintas.

2.2.1 Generación de una solución aleatoria

```
Inicializamos el generador de aleatorios con la semilla proporcionada
sol = {1, 2, …, n} // Ponemos la solución con la permutación estrictamente creciente
sol = baraje aleatorio de sol
sol.coste  = Cálculo directo del coste de sol
Devolver sol
```

2.2.2 Generación de una población aleatoria

```
Inicializamos el generador de aleatorios con la semilla proporcionada
sol = {1, 2, …, n} // Ponemos la solución con la permutación estrictamente creciente
C = {} // Conjunto de soluciones con su coste vacío

Mientras que el tamaño del conjunto C sea menor que el tamaño deseado de población:
      sol = baraje aleatorio de sol
      sol.coste = Cálculo directo del coste de sol
      Insertar sol en C

Devolver C
```

2.3 Operador de generación de vecino

El operador utilizado en la búsqueda local para la generación del vecindario es el 2-OPT. Dicho operador intercambia dos posiciones de la permutación que representa la solución cambiando por tanto la asignación de dos unidades con la localización de la contraria. Este operador también es utilizado en los algoritmos genéticos como operador de mutación.

```
S.coste = Cálculo de coste factorizado de S sobre r , s
Intercambiar S[r] con S[s]
```

3. Método de búsqueda y operaciones de los algoritmos

3.1 Búsqueda local del primer mejor

La búsqueda local del primer mejor hace uso de la técnica Don't Look Bits junto al operador 2-OPT para explorar el vecindario de las soluciones. Esta técnica reduce un poco la eficacia de las soluciones, pero aumenta considerablemente la eficiencia del algoritmo. Las condiciones de parada son que se haya superado un número máximo de evaluaciones o que tras una vuelta completa no haya mejorado la solución.

```
// Partiendo de una solución S (aleatoria o previa para meméticos).
// dlb es un vector de booleanos de tamaño N inicializado a Falso
Mientras que S mejore y el número de evaluaciones sea menor que 50000:
      para cada i desde 1 hasta N tal que dlb[i] es Falso:
            hemosMejorado = Falso
            para cada j desde 1 hasta N:
                  aux = Calculamos 2-OPT de S con respecto a i,j
                  Aumentar contador de evaluaciones
                  si aux es mejor que la solución actual:
                        dlb[i] = Falso
                        dlb[j] = Falso
                        S = aux
                        hemosMejorado = Verdadero
            si hemosMejorado es Falso:
                  dlb[i] = Verdadero
Devolver S
```

3.2 Enfriamiento simulado

3.2.1 Cálculo de la temperatura inicial

Para calcular la temperatura inicial utilizamos la siguiente fórmula:

$$T_0 = \frac{\mu \cdot Coste(S)}{\ln(\Phi)}, donde \Phi = \mu = 0.3 y S es la solución inicial creada aleatoriamente.$$

3.2.2 Esquema de enfriamiento

Para actualizar la temperatura en cada iteración utilizamos el esquema de Cauchy modificado. Esto quiere decir que al inicio calcularemos la siguiente beta:

$$\beta = \frac{T_0 - T_f}{M \cdot T_0 \cdot T_f}$$

M hace referencia al número máximo de enfriamientos que vamos a realizar en nuestro caso. En mi caso M vale 50000/10n, donde n es el tamaño del caso que se está evaluando.

Siendo k la iteración (o número de enfriamientos que llevamos hasta el momento) la actualización de la temperatura viene dada por:

$$T_{k+1} = \frac{T_k}{1 + \beta \cdot T_k}$$

3.2.3 Algoritmo

```
Solución = mejorSolución = Calcular solución aleatoria
T = T_inicial = Calcular temperatura inicial
T_final = 0.001
B = Calcular beta
M = 50000/10n (siendo n el tamaño del caso).
Mientras T sea superior a la temperatura final:
   contador_vecinos = contador_exitos = 0
   Mientras contador_vecinos < 10n y contador_exitos < n:
      r = Posición aleatoria de la permutación
      s = Posición aleatoria de la permutación
      nuevaSolución = 2-Opt de Solución con respecto a r y s
      Aumentar el contador_vecinos en 1
      f = coste de nuevaSolución – coste de Solución
      Si f > 0 or un aleatorio entre 0 y 1 es menor o igual que exp(-f/k*T):
            Aumentar el contador de exitos en 1
            Solución = nuevaSolución
            Si Solución tiene un coste menor que mejorSolución:
                  mejorSolución: Solución

    T = Enfriamiento (Esquema de Cauchy modificado).
Devolver mejor solución
```

3.3 Búsqueda multiarranque básica

```
Población = generar población aleatoria de 25 individuos
Para cada individuo en Población:
      individuo = Aplicar búsqueda local sobre individuo con un máximo de 50000
                  evaluaciones
Devolver mejor individuo (menor coste) de población.
```

3.4 GRASP

La creación de la solución aleatoria se divide en dos etapas:

3.4.1 Lista de candidatos restringida: Etapa 1

Para la primera etapa se crean dos listas de candidatos, una para seleccionar las unidades y otra las localizaciones basándonos, respectivamente, en el potencial de flujo y el de distancia.

LCU (unidades) = lista de pares formado por {n.º unidad, potencial de flujo}

LCL (localizaciones) = lista de pares formado por {n.º localización, potencial de distancia}

El potencial de flujo de una unidad se calcula como:

$$\sum_{i=1}^{n} flujodesdeunidadhastai$$

El potencial de distancia de una localización se calcula como:

$$\sum_{i=1}^{n} distanciadesdelocalizaciónhastai$$

Ahora cada una de ellas pasará a ser una lista restringida basándonos en las siguientes restricciones (la lista se actualiza eliminando aquellos elementos que no cumplan la restricción).

Siendo min_flujo y max_flujo el menor y mayor potencial de flujo respectivamente y, análogamente, min_distancia y max_distancia para los potenciales de distancia.

Sólo se quedarán en LCU todos los pares cuyo potencial de flujo, pf, tales que:

$$pf \geq max_{flujo} - \alpha \cdot (max_{flujo} - min_{flujo}), siendo \alpha = 0.3$$

Sólo se quedarán en LCL todos los pares cuyo potencial de distancia, pd, tales que:

$$pd \leq min_{distancia} + \alpha \cdot (max_{distancia} - min_{distancia}), siendo \alpha = 0.3$$

Obtenemos dos unidades aleatorias y dos flujos aleatorios de las listas recién restringidas y asignamos en la permutación solución la primera unidad a la primera localización y la segunda unidad a la segunda localización.

3.4.1 Lista de candidatos restringida: Etapa 2

En la etapa 2 tenemos que hacer las n-2 asignaciones restantes.

En cada iteración:

Creamos la lista de candidatos (cada una de las posibles asignaciones que sean factibles con un coste dado por $\sum_{i=1,S[i]\neq-1}^{n} flujo(unidad, i) \cdot distancia(localización, S[i])$.

Restringimos la lista de candidatos aplicando la siguiente restricción:

// Siendo max_coste y min_coste el máximo y el mínimo de los costes

$coste \leq min_{coste} + \alpha \cdot (max_{coste} - min_{coste})$

Escogemos una asignación aleatoria y la aplicamos en la permutación.

Una vez ha terminado hemos obtenido una solución greedy aleatoria

3.4.3 Algoritmo

```
Durante 25 iteraciones:
   Solución = Construir solución greedy aleatoria (explicado en 3.4.1 y 3.4.2)
   Solución = Aplicar búsqueda local sobre Solución (max 50000 evaluaciones)
   Insertar Solución en Población

Devolver la solución con menor coste de Población
```

3.5 ILS

3.5.1 Mutación

```
tam = n/4
pos_inicial = Seleccionar posiciona aleatoria desde 1 hasta 3*n/4
Hacer una permutación aleatoria de los elementos de la permutación de Solución
Solución.coste = Cálculo del coste directo de Solución
Devolver solución
```

3.5.2 Algoritmo

```
// Nota: busquedaTrayectoria en esta práctica es la búsqueda local para ILS y el
// enfriamiento simulado para ILS-ES
mejorSolución = Solución = Generar solución aleatoria
Repetir 24 veces:
      Solución2 = Aplicar Mutación sobre mejorSolución
      Solución2 = Aplicar busquedaTrayectoria sobre Solución2
      Solución = la de menor coste de entre Solución y Solución2
      mejorSolución = la de menor coste entre Solución y mejorSolución

Devolver mejorSolución
```

4. Procedimiento considerado para realizar la práctica

El código correspondiente a los algoritmos desarrollados en C++17 de cero a partir de los conocimientos adquiridos en las clases teóricas y los seminarios de la asignatura para continuar usando lo mismo que en la entrega de la primera práctica. Se proporciona un makefile para poder compilarlo y una versión precompilada para Linux de 64 bits para poder usarlo.

5. Experimentos y análisis de resultados

5.1 Resultados de la aplicación de los algoritmos a los casos

Nota: Para todas las tablas mostradas a continuación se ha utilizado la semilla 18. Si no se especifica una semilla diferente a la hora de ejecutar los algoritmos esta será la semilla utilizada.

Todos los casos proporcionados han sido ejecutados.

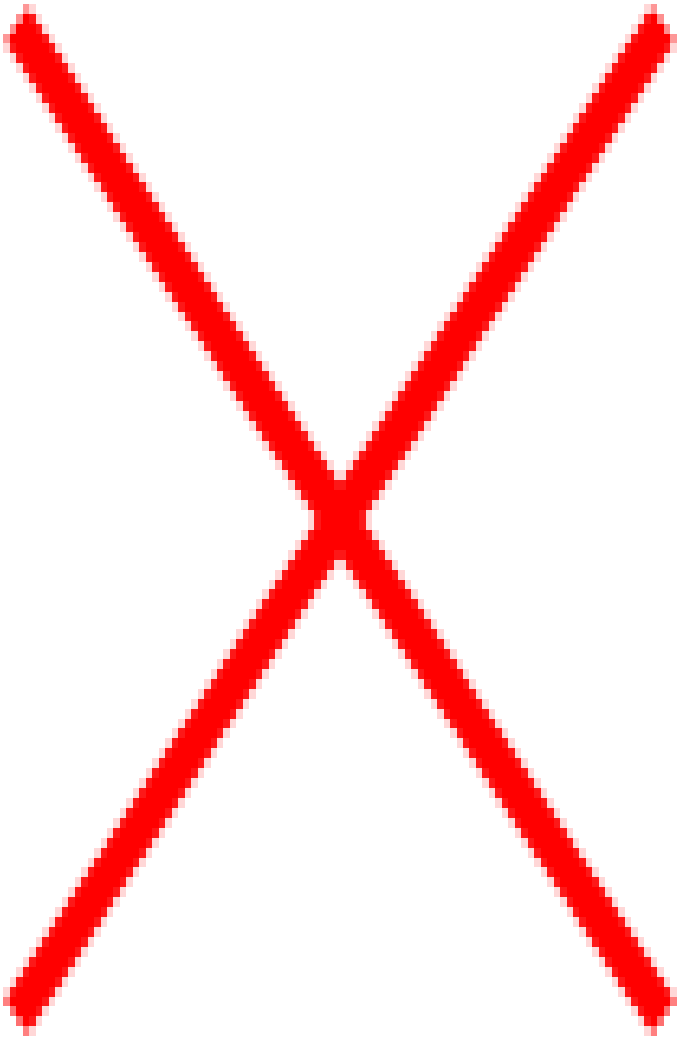

Tabla 5.1: Algoritmo Greedy

Como podemos observar el algoritmo greedy no nos ofrece las mejores soluciones, pero sí que ofrece unas soluciones iniciales de una manera muy rápida. Debido a lo poco que tarda es probable que nos interese comprobar el caso a analizar con el algoritmo greedy y determinar si la calidad de la solución es suficientemente buena o debemos utilizar alguna de las metaheurísticas propuestas en el estudio.

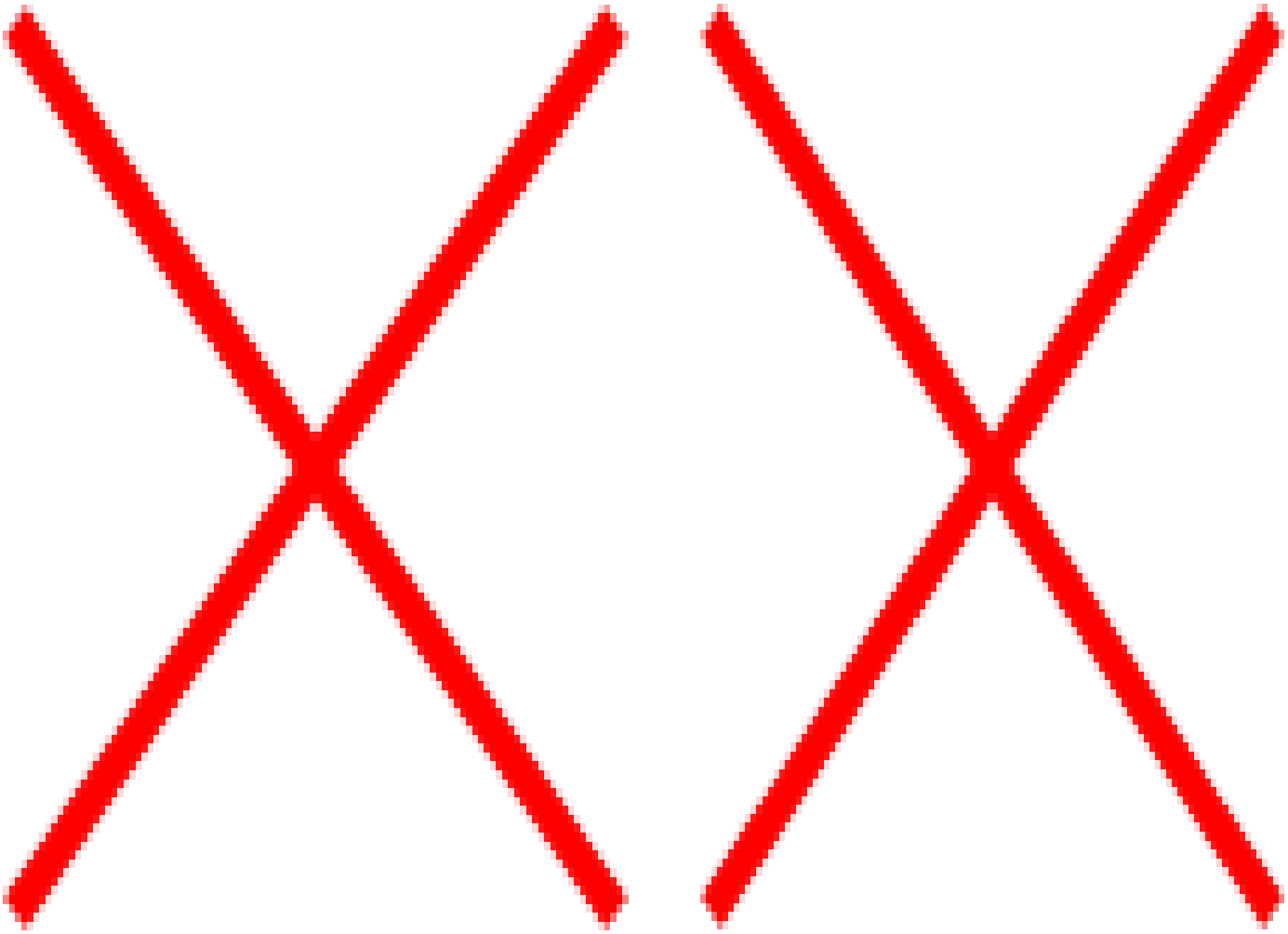

Tabla 5.2: Búsqueda Local *Tabla 5.3: Enfriamiento Simulado*

A diferencia de la búsqueda Local, el enfriamiento simulado, a coste de un ligero mayor tiempo de ejecución nos permite movernos más por el espacio de búsqueda, evitando ciertas situaciones de convergencia prematura, gracias a que permite volver a soluciones peores, por esa razón cabe esperar que enfriamiento simulado supere en bastantes casos a la búsqueda local y que de media ofrezca una desviación ligeramente menor.

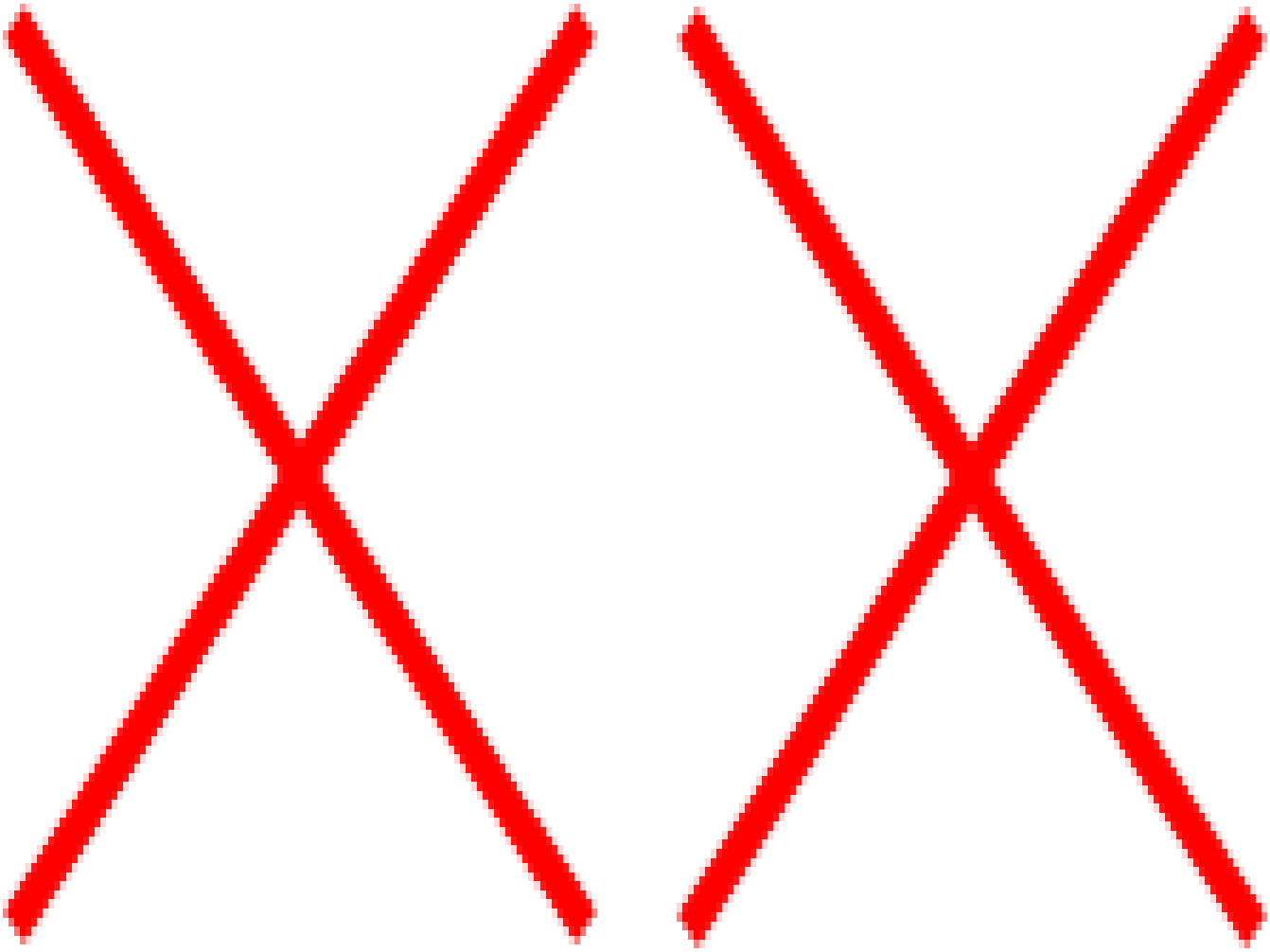

Tabla 5.4 : Búsqueda Multiarranque Básica *Tabla 5.5: GRASP*

Para solucionar la convergencia prematura muchas metaheurísticas parten de diversas soluciones con el fin de evitarlas. Dos de estas alternativas (BMB y GRASP) generan X soluciones aleatorias y aplican un algoritmo básico de búsqueda de trayectorias para cada una (en esta experimentación Búsqueda Local). Este mecanismo bastante simple y que no parece muy fiable da soluciones muy buenas. Mientras que BMB se basa en soluciones completamente aleatorias, GRASP se basa en un procedimiento greedy aleatorizado que aumenta muy considerablemente el tiempo de ejecución sin obtener resultados mejores que BMB.

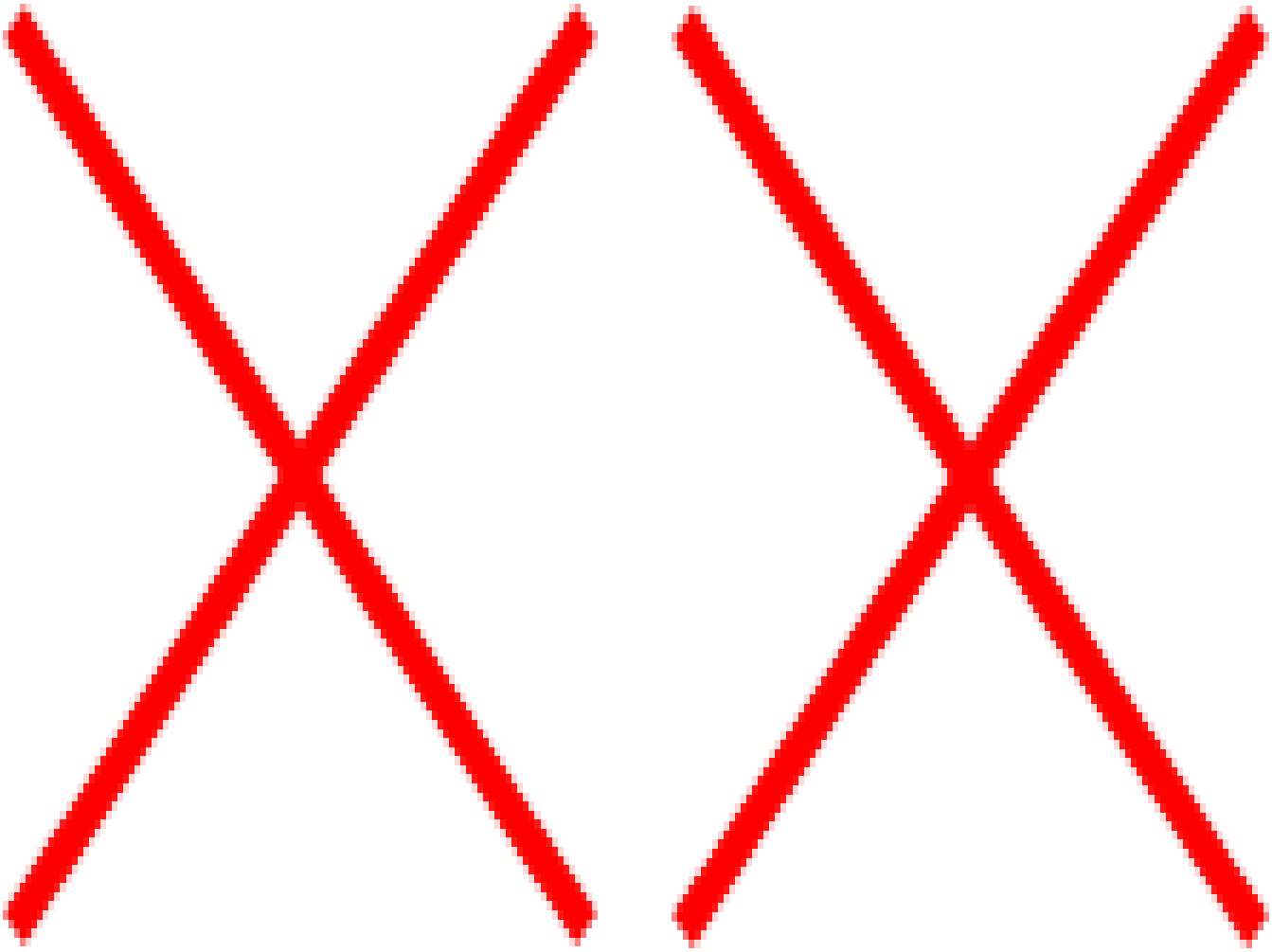

Tabla 5.6: ILS *Tabla 5.7: ILS-ES*

La última metaheurística vista para evitar la convergencia es ILS. En esta se utiliza un procedimiento de mutación a la solución que estamos usando para evitar que converja mientras guardamos los mejores resultados. Su versión híbrida con Enfriamiento Simulado ha dado lugar a los mejores resultados medios de todas las metaheurísticas probadas, con tan sólo un 4,5 % de desviación.

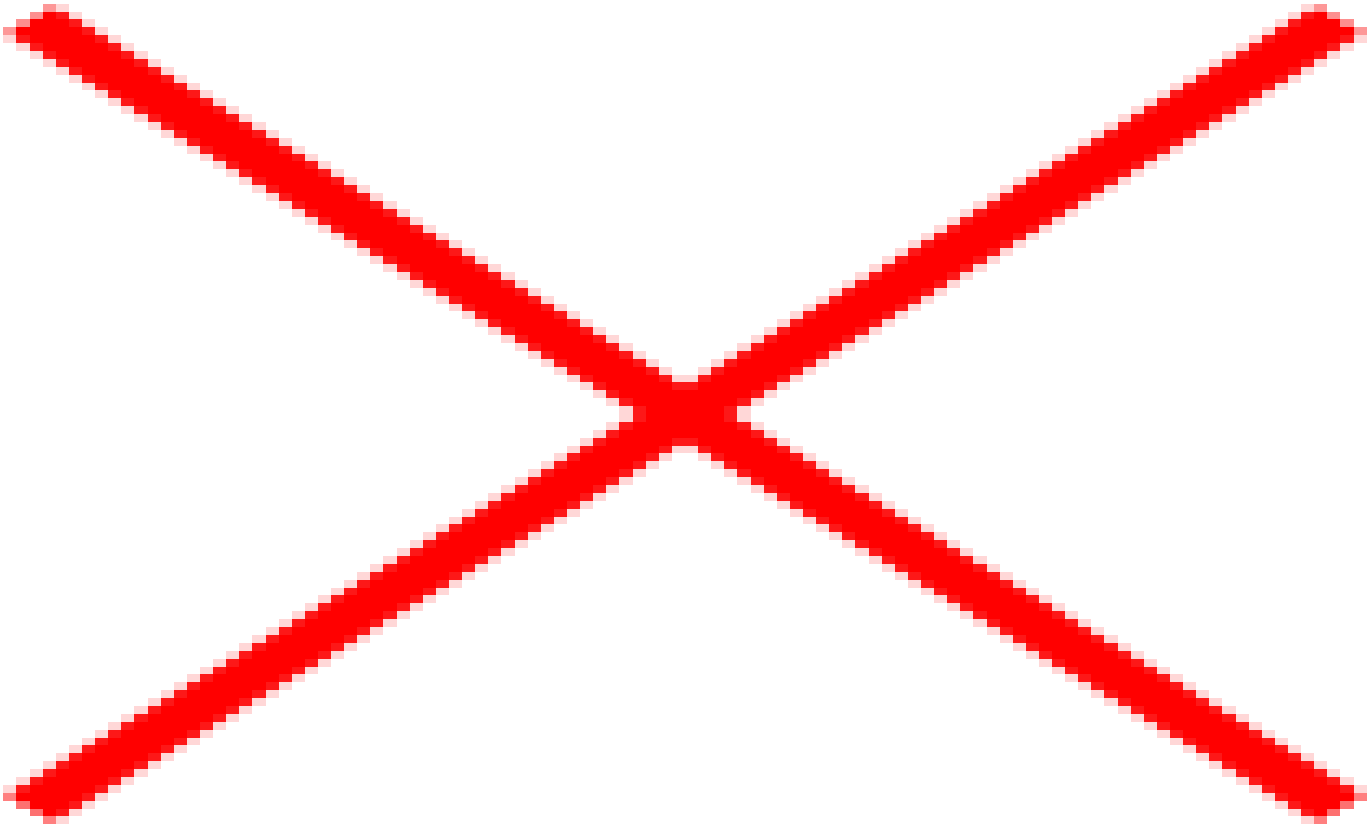

Tabla 5.8: Comparativa de resultados