METAHEURÍSTICA
GRADO EN INGENIERÍA INFORMÁTICA
UNIVERSIDAD DE GRANADA

Práctica 3

José Luis Molina Aguilar

10 de junio de 2022

Curso 2021-2022
DNI : 77556436E
Correo : joselu201@correo.ugr.es
Grupo : A3, MARTES 17:30 - 19:30

Índice

Índice de figuras

1. Descripción Problema de Mínima Dispersión Diferencial

El problema de Mínima Dispersión Diferencial es un problema de optimización combinatoria que entra en la clase de problemas **NP-Completo**
Este es un problema en el que las heurísticas obtienen buenas soluciones en menos tiempo.

1.1. Descripción

Dado un conjunto de n elementos todos ellos conectados entre sí, representado por una matriz de distancias de tamaño nxn obtener un subconjunto m tal que la diferencia entre la máxima distancia acumulada y la mínima distancia acumulada de los elementos de m se minimiza.
El conjunto $m < n$ y por lo tanto lo que estamos buscando es $m \subset n \mid MinimizeDD(S_m)$
donde $DD(S_m)$ es la Dispersión Diferencial del conjunto de Soluciones de tamaño m

1.2. Consideraciones

En mi representación de este problema la matriz de distancias descrita anteriormente será una matriz de flotantes llamada **datos**

Además implementaré un vector **distan** la cual almacena la distancia desde un punto al resto, será útil para factorizar en BL.

2. Greedy

El algoritmo Greedy se basa en la heurística de ir añadiendo a la solución el elemento más óptimo de los disponibles, el cual es el que minimice la dispersión.
Elegiremos el primer elemento de m de forma aleatoria para ganar variedad en los resultados.
Después, el resto de elementos a elegir hasta completar la solución será, sobre todos los posibles candidatos, calculamos la dispersión cuando añadimos ese elemento a la solución m y el elemento que la minimice será escogido y añadido a la solución.
Esta aproximación cae fácilmente en óptimos locales ya que es muy dependiente de los del punto de inicio y en cada paso aunque escojamos el elemento que minimiza la Dispersión no significa que, como conjunto solución, sea el correcto.
La ventaja principal del greedy es que obtiene una solución relativamente buena en mucho menos tiempo que el algoritmo perfecto que resuelve este problema.
Para ayudarnos en el desarrollo del Greedy usaremos 3 funciones:

- **distPuntoRestoElemenetos**, que calcula la distancia acumulada de un punto al resto del vector.

- **diff**, el cual dado un vector de soluciones calcule las distancias acumuladas (distPunto-RestoElemenetos) y devuelva la dispersión para ese conjunto.

- **fit_adding**, esta función simplemente calcula la dispersión (mediante diff) si añadimos un nuevo elemento al vector de soluciones.

La representación de la solución la realizaremos con un vector de enteros que almacena los índices de los elementos escogidos.
Por lo que el algoritmo Greedy quedaría:

Algorithm 1 Greedy

```
1:  function GREEDY
2:      Solucion ← ∅
3:      Candidatos ← V                                  ▷ V son todos los indices, n
4:      v_0 ← SelectRandomFrom(Candidatos)
5:      Solucion ← Solucion ∪ {v_0}
6:      Candidatos ← Candidatos \ {v_0}
7:      while |Solucion| < m do
8:          for ele in Candidatos do
9:              min ← FLOATMAX
10:             new_fitness ← fit_adding(Solucion, ele)
11:             if new_fitness < min then
12:                 ele_pos ← ele                       ▷ Guardo el mejor elemento
13:                 min ← new_fitness                   ▷ Actualizo el minimo actual
14:             end if
15:         end for
16:         Solucion ← Solucion ∪ {ele_pos}             ▷ Añado a la solucion
17:         Candidatos ← Candidatos \ {ele_pos}
18:     end while
19:     return Solucion
20: end function
```

Pseudocodigo de **distPuntoRestoElemenetos**

Algorithm 2 distPuntoRestoElemenetos

```
1: function DISTPUNTORESTOELEMENETOS(FILA,VECTOR)
2:     dist ← 0
3:     for i ← 0 to length(vector) do
4:         dist ← dist + datos[fila][vector[i]]
5:     end for
6:     return dist
7: end function
```

Pseudocodigo de **diff**

Algorithm 3 diff

```
1: function DIFF(POSIBLES)
2:     distancias ← ∅
3:     for i ← 0 to length(posibles) do
4:         distancias ← distancias ∪ distPuntoRestoElemenetos(posibles[i], posib))
5:     end for
6:     sort(distancias)
7:     return distancias[length(posibles)] − distancias[0]
8: end function
```

Pseudocodigo de **fit_adding**

Algorithm 4 fit_adding

```
1: function fit_adding(posibles, new_i)
2:     posibles ← posibles ∪ new_i
3:     new_diff ← diff(posibles)
4:     posibles ← posibles \ new_i
5:     return new_diff
6: end function
```

3. Busqueda por Trayectorias Simples (BL)

La búsqueda local se basa en generar una solución aleatoria, la cual como solución válida tiene que satisfacer las restricciones de

- No puede tener elementos repetidos
- Tiene que tener exactamente m elementos
- El orden no es relevante

Para obtener una solución BL aplica un Operador de intercambio, este es:
Dada una solución, intercambiar un elemento de esa solución por otro elemento del conjunto Candidatos, (el cual esto formado por todos los índices menos los que están en solución, S-Solucion = Candidatos), el cual minimice el valor del fitness.
Esto provoca que el espacio de posibilidades de cambio sea de $m \cdot (m-n)$, por lo que a la hora de aplicar esto, una vez que encontremos un elemento que minimice la dispersión se añadirá a la solución, (con añadir me refiero a intercambiar los valores) y seguidamente buscaremos otra vez para el siguiente elemento de la solución, puede llegar un punto en el que una vez recorrido todo el conjunto de soluciones e intentar intercambiarlo por algún elemento del conjunto de Candidatos ninguno minimice el valor actual, en ese caso terminaremos y devolveremos la solución actual.
También utilizaremos un número limitado de iteraciones.

3.1. Factorización del Movimiento de Intercambio

Ejemplo.
Dado el conjunto solución (0,4,6), cambio el elemento 0 por 1, quedaria (1,4,6) por lo que el vector distan quedaría:
D0 = D04 + D06 //Esta ya no lo necesito
D4 = D40 + D46 //Si cambio el 0 por un 1, D4 = D4 -D04 + D14
D6 = D60 + D64

D1 = D14 + D16 //Este tengo que recalcularlo entero
D4 = D4 -D04 + D14
D6 = D6 -D06 + D16

De esta forma no tengo que volver a calcular de nuevo todas las distancia de un punto al resto, sino que simplemente tendré que actualizar el valor de la forma anteriormente descrita.
Lo cual me hace pasar de una complejidad $\mathcal{O}(n^2)$ a $\mathcal{O}(n)$

Algorithm 5 BL

```
1: function BL
2:     Solucion ← SelectRandomSolution
3:     Candidatos ← V                                   ▷ V son todos los indices de n
4:     Shuffle(Candidatos)
5:     index ← 0                                                  ▷ Indice de solucion
6:     MaxIters ← 1000000
7:     cambia ← true
8:     iter ← 0
9:     while iter < MaxIters and cambia do
10:        for i ← 0 to length(candidatos) do
11:            actual_disp ← diff(Solucion)
12:            intercambio ← (index, cand[i])   ▷ Cambio el elemento index por un candidato
13:            new_disp ← distFactorizada(Solucion, intercambio)
14:            if new_disp < actual_disp then
15:                Solucion ← Solucion ∪ {cand[i]}                  ▷ Añado a la solucion
16:                index ← index + 1                ▷ Avanzo a otro elemento de solucion
17:                cambio ← true                          ▷ Anoto que ha habido cambio
18:                i ← 0                          ▷ Vuelvo a mirar con el siguiente elemento
19:            end if
20:            if !cambio and solucion[index] != solucion[solucion.size()] then
21:                        ▷ Si no ha habido cambio, pero no he he comprobado todo Solucion
22:                index ← index + 1                          ▷ Avanzo al siguiente elemento
23:                i ← 0                          ▷ Vuelvo a mirar si algun candidato mejora
24:            else if solucion[index] == solucion[solucion.size()] then
25:                                                          ▷ Si he comprobado todas
26:                cambio ← false
27:            end if
28:            iter ← iter + 1
29:        end for
30:        index ← index + 1
31:    end while
32:    return Solucion
33: end function
```

Pseudocodigo de **distFactorizada**

```
1: function distFactorizada(solucion, cambio)
2:     distan[cambio.first] = distPuntoRestoElemenetos(cambio.second, solucion);  ▷ Recalculo el punto que he cambiado
3:     for i ← 0 to length(solucion) do
4:         distan[i] ← distan[i] − datos[solucion[cambio.first]][solucion[i]] + datos[cambio.second][solucion[i]];
5:     end for
6:     Sort(distan)
7:     return (distan[distan.size() − 1] − distan[0]);
8: end function
```

4. Analisis Práctica 1

- La información de uso se encuentra en el README.md.
- Las semillas con las que se han obtenido los resultados entan el main.cpp y son {0,1,2,3,4}.

Finalmente para representar los resultados obtenidos por estos dos algoritmos respecto del perfecto tenemos la siguinte grafica

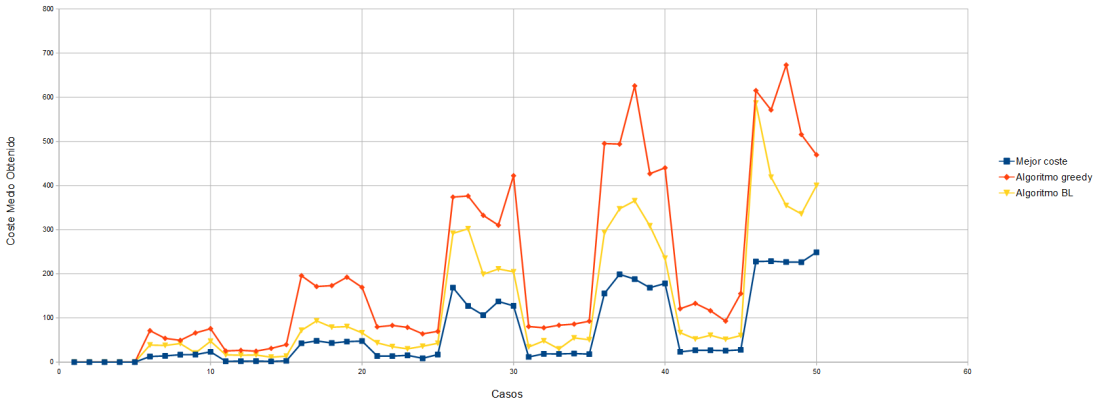

Figura 4.1: Resultados distintos Algoritmos

En la Gráfica 4.1 podemos ver como se comporta cada algoritmo, como vemos la BL es mucho mejor que el Greedy en este caso ya que esta mas cerca del resultado que nos ofrece el algoritmo perfecto.

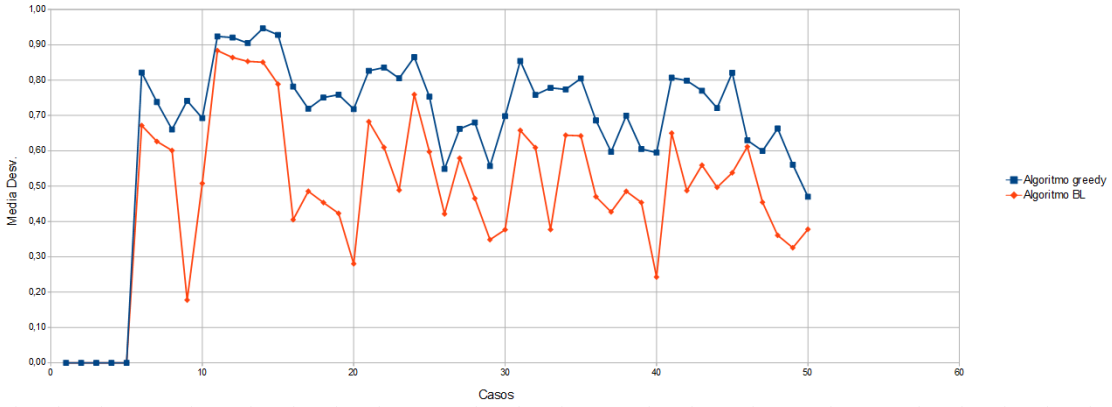

Figura 4.2: Desviación distintos Algoritmos

Pero esta mejora en la precisión conlleva como podemos ver el la Gráfica 4.3 un aumento sustancial en el tiempo de computo aunque si pudiesemos comparar el tiempo que le tomo al algoritmo perfecto, la BL en comparación sería muy rapida.
Además podemos ver que que el tiempo que tarda en dar una solucion es directamente proporcional al numero de elementos de m ya que, como podemos ver podemos diferenciar intervalos que tienen en comun una cosa n, por ejemplo [31,40], $n = 125$, dentro de ese intervalo tenemos que [31-35] $m = 12$ y [35 - 40] $m = 37$, en la Gráfica podemos ver que tarda mucho menos en [31-35] ya que el valor de m es menor que en el intervalo [35 - 40]

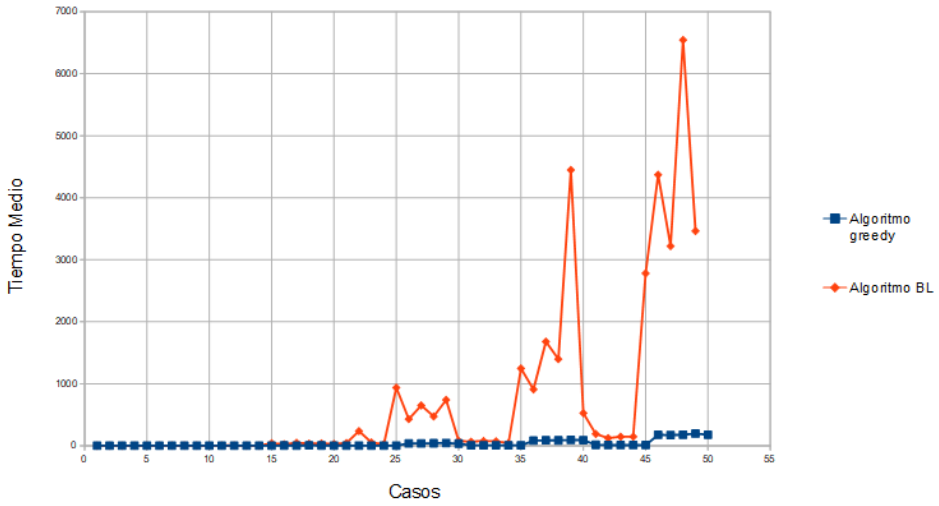

Figura 4.3: Tiempos(ms) para diferentes Algoritmos

5. Práctica2a : Técnicas de Búsqueda Basados en Poblaciones

En esta practica tendremos que cambiar la representacion de las solución de enteras a binarias, por lo que ahora tendremos que cada solución tendrá un vector de n elementos donde m de esos elementos tienen que ser obligatoriamente 1's lo que significará que esos genes estan activos. Cambiaremos a esta representacion ya que es idonenea para realizar ejercicios basados en poblacion gracias a la facilidad de generar, cruzar y mutar individuos.

Nos ayudaremos de funciones para realizar estos algoritmos:

- **generarPoblacion()** Esta funcion nos permite generar individuos aleatoriamente respentando las restricciones que se nos proponen, si ejecutamos esta funcion N veces obtendremos una poblacion de N individuos.
  Una vez que tenemos una poblacion el siguiendo punto importante seria ser capaz de seleccionar individuos de la misma con el fin de cruzarlos y generar hijo, para ello:

- **seleccion(poblacion, n)** Esta funcion nos permitira elegir n individuos de la poblacion. Una vez que tenemos a los individuos seleccionados tenemos que generar una seria de hijos de estos individuos seleccionados, para ello tenemos dos opciones.

  - **generarHijosPosicion(padre1, padre2)** Esta funcion cruzara dos padres, manteniendo los genes comunes y eligira aleatoriamente el resto de la siguiente forma, restará los genes que no son coincidente los mezclará y insertara en los hijos, de esta forma si partimos de padres factibles los hijos tambien lo serán.

    De esta forma los hijos generados comparten un poco menos de informacion con los padres.
  - **generarHijosUniforme(padre1, padre2)** Esta función cruzamos dos padres, manteniendo los genes comunes y eligiendo aleatoriamente el resto, esto puede provocar que los hijos que obtengamos no cumplan las restricciones, por lo que a veces en necesario reparlos, para ello:

  - **repair(hijo)** La manera en la que reparamos se basa en si faltan elementos, añadir los que minimizen la disperion y si sobran elementos eliminar aquellos que minimizen la dispersion.

Pseudocodigo de **generarPoblacion**

```
1:  function generarPoblacion()
2:      poblacion ← ∅                          ▷ Creamos una poblacion vacia
3:      while count(poblacion, 1)! = m do      ▷ El numero de 1 sea distinto de m
4:          idx ← Random(0, n)                 ▷ Obtenemos un numero de 0 a n
5:          if poblacion[idx] == 0 then
6:              poblacion[idx] = 1
7:          else continue;
8:          end if
9:      end while
10:     return poblacion
11: end function
```

Entonces a la hora de crear una poblacion ejecutaremos esta funcion tantas veces como individuos queramos, para después guardarlo en una matriz de todos los individuos llamada poblacion. Ademas tendremos un vector llamadao **fitness_i** en la cual guardaremos el fitness de cada individuo.

Pseudocodigo de **torneo(poblacion,fitness_i, indiv)**

```
1:  function torneo(poblacion, fitness_i, indiv)
2:      torneo ← ∅                             ▷ Creamos un torneo vacia
3:      worst ← ∞
4:      while |torneo|! = indiv do             ▷ Tenemos indiv diferentes
5:          torneo ← torneo ∪ Random(0, |poblacion|)   ▷ Obtenemos un numero de 0 a n
6:      end while
7:      for i in torneo do
8:          if fitness_i[i] < worst then
9:              worst ← fitness_i[i]
10:             winner ← i
11:         end if
12:     end for
13:     return winner
14: end function
```

Funcion que escoge n elementos ganadores del torneo.
Pseudocodigo de **seleccion(poblacion,fitness_i, indiv,n)**

```
1: function seleccion(poblacion, fitness_i, indiv, n)
2:     winner ← ∅
3:     for i in n do
4:         winner ← winner ∪ torneo(poblacion, fitness_i, n)
5:     end for
6:     return winner
7: end function
```

Pseudocodigo de **generarHijosUniforme(padre1,padre2)**

```
1: function generarHijosUniforme(padre1, padre2)
2:     hijos ← ∅
3:     for i = 0in|padre| do
4:         if padre1[i] == padre2[i] then
5:             hijos[i] ← padre1[i]
6:         else
7:             hijos[i] ← Random(0, 1)
8:         end if
9:     end for
10:    reparar(hijos)
11:    return hijos
12: end function
```

Pseudocodigo de **reparar(hijo)** ||| Completar esto con lo del codigo

```
1: function reparar(hijo)
2:     v ← count(hijos, 1)
3:     if v == m then
4:         return hijo
5:     else if v > m then                          ▷ Si te sobran elementos
6:         while v !=m do
7:             eliminar los genes que minimizan la dispersion
8:         end while
9:     else if v < m then                          ▷ Si te faltan elementos
10:        while v != m do
11:            escoger los genes que minimizan la dispersion
12:        end while
13:    end if
14:    return hijo
15: end function
```

Pseudocodigo de **generarHijosPosicion(padre1,padre2)**

```
1:  function generarHijosPosicion(padre1, padre2)
2:      hijos ← ∅
3:      restos ← ∅
4:      if padre1 == padre2 then
5:          hijos = padre1, padre2                    ▷ Devolvemos los mismos padres
6:      end if
7:      for i = 0in|padre| do
8:          if padre1[i] != padre2[i] then
9:              if padre1[i] == 0 and padre2[i] == 1 then
10:                 resto ← resto ∪ 0
11:             else if padre1[i] == 1 and padre2[i] == 0 then
12:                 resto ← resto ∪ 1
13:             end if
14:         else
15:             continue
16:         end if
17:     end for
18:     shuffle(resto)
19:     j ← 0
20:     for i = 0in|padre| do
21:         if padre1[i] == padre2[i] then
22:             hijos[i] ← padre1[i]
23:         else
24:             hijos[i] ← resto[j]
25:             j ← j + 1
26:         end if
27:     end for
28:     return hijos
29: end function
```

5.1. Algoritmos Genéticos

Implentaremos 2 tipos de algorimos geneticos,
AGG(algorimo genetico generacional)
AGG se basa en un algorimo que genera una poblacion aleatoria, despues se escogen mediante un torneo binario tantos individuos como poblacion del mismo y se cruzan, en nuestro caso diferenciaremos dos tipos de cruce uniforme y posicion (cuyos Pseudocodigo estas explicado arriba) para seguidamente mutar algunos de sus genes con probabilidad P_M y reemplazar la nueva poblacion obtenida por la anterior y siempre conservando el mejor de ambas(elitismo).

AGE(algorimo genetico Estacional)
Este es practicamente igual, la unica diferencia es que a laa hora de selecionar los individuos, solo se seleccionan 2, estos seran los que se cruzaran y cuyos hijos mutaran, para seguidamente, si mejoran los peores individuos de la poblacion, sustituirlos.

```
1: function AGG_uniforme()
2:     TAM = 50
3:     MAX_ITERS = 100000
4:     iters = 0
5:     poblacion ← ∅                                   ▷ Matriz de TAM individuos
6:     hijo ← ∅                                        ▷ Este sera el hijo solucion
7:     hijos ← ∅
8:     fitness_i ← ∅
9:     for row in poblacion do
10:        row ← generarPoblacion()
11:        fitness_i ← fitness_i ∪ diff(row)
12:    end for
13:    while iters < MAX_ITERS do
14:        best_father ← best(poblacion)        ▷ Individuo de la poblacion con la menor
   dispersión
15:        for row in |poblacion| do
16:            padre2 ← next(row)                          ▷ Siguiente individuo
17:            if boolRandom(0,7) then
18:                hijos ← hijos ∪ generarHijosUniforme(row, padre2)
19:                fitness_i ← fitness_i ∪ diff(hijos)   ▷ Actualizamos la dispersion de los
   nuevos hijos
20:            else
21:                hijos ← hijos ∪ row
22:            end if
23:        end for
24:        spected_mutations ← 0,1 * n * m
25:        count = 0
26:        while count! = spected_mutations do
27:            cromosoma ← Random(0, TAM)              ▷ Elegimos el Individuo a mutar
28:            gen1 ← Random(0, n)
29:            gen2 ← Random(0, n)                     ▷ Elegimos dos genes Diferentes
30:            if hijosatcromosoma[gen1]! = hijosatcromosoma[gen2] then ▷ Si los genes del
   individuos elegido en hijos son distintos
31:                swap(hijosatcromosoma[gen1], hijosatcromosoma[gen2])
32:            end if
33:            count ← count + 1
34:        end while
35:        poblacion ← hijos                    ▷ Los hijos sustituyen a la poblacion actual
36:        if !find(fitness_i, best_father) then   ▷ Si no encuentro el mejor individuo de la
   poblacion sustituyo el peor por el mejor de la anterior
37:            poblacion at peor elemento ← best_father
38:        end if
39:        iters ← iters + 1
40:    end while
41:    hijo ← min(poblacion)
42:    return hijo
43: end function
```

```
1: function AGG_posicion()
2:     TAM = 50
3:     MAX_ITERS = 100000
4:     iters = 0
5:     poblacion ← ∅                                   ▷ Matriz de TAM individuos
6:     hijo ← ∅                                        ▷ Este sera el hijo solucion
7:     hijos ← ∅
8:     fitness_i ← ∅
9:     for row in poblacion do
10:        row ← generarPoblacion()
11:        fitness_i ← fitness_i ∪ diff(row)
12:    end for
13:    while iters < MAX_ITERS do
14:        best_father ← best(poblacion)               ▷ Individuo de la poblacion con la menor
   dispersión
15:        for row in |poblacion| do
16:            padre2 ← next(row)                      ▷ Siguiente individuo
17:            if boolRandom(0,7) then
18:                hijos ← hijos ∪ generarHijosPosicion(row, padre2)
19:                fitness_i ← fitness_i ∪ diff(hijos)  ▷ Actualizamos la dispersion de los
   nuevos hijos
20:            else
21:                hijos ← hijos ∪ row
22:            end if
23:        end for
24:        spected_mutations ← 0,1 * n * m
25:        count = 0
26:        while count! = spected_mutations do
27:            cromosoma ← Random(0, TAM)              ▷ Elegimos el Individuo a mutar
28:            gen1 ← Random(0, n)
29:            gen2 ← Random(0, n)                     ▷ Elegimos dos genes Diferentes
30:            if hijosatcromosoma[gen1]! = hijosatcromosoma[gen2] then ▷ Si los genes del
   individuos elegido en hijos son distintos
31:                swap(hijosatcromosoma[gen1], hijosatcromosoma[gen2])
32:            end if
33:            count ← count + 1
34:        end while
35:        poblacion ← hijos                           ▷ Los hijos sustituyen a la poblacion actual
36:        if !find(fitness_i, best_father) then       ▷ Si no encuentro el mejor individuo de la
   poblacion sustituyo el peor por el mejor de la anterior
37:            poblacion at peor elemento ← best_father
38:        end if
39:        iters ← iters + 1
40:    end while
41:    hijo ← min(poblacion)
42:    return hijo
43: end function
```

```
1: function AGE_uniforme()
2:     TAM = 50
3:     MAX_ITERS = 100000
4:     iters = 0
5:     poblacion ← ∅                                   ▷ Matriz de TAM individuos
6:     hijo ← ∅                                        ▷ Este sera el hijo solucion
7:     hijos ← ∅
8:     fitness_i ← ∅
9:     fitness_hijo ← ∅
10:    for row in poblacion do
11:        row ← generarPoblacion()
12:        fitness_i ← fitness_i ∪ diff(row)
13:    end for
14:    while iters < MAX_ITERS do
15:        selected ← seleccion(poblacion, fitness_i, 2)          ▷ Eligo los 2 padres
16:        hijos ← generarHijosUniforme(selected[0], selected[1])  ▷ Cruzo los 2 padres
17:        spected_mutations ← 0,1 * n * 2
18:        count = 0
19:        while count! = spected_mutations do
20:            cromosoma ← Random(0, |hijos|)              ▷ Elegimos el Individuo a mutar
21:            gen1 ← Random(0, n)
22:            gen2 ← Random(0, n)                         ▷ Elegimos dos genes Diferentes
23:            if hijosatcromosoma[gen1]! = hijosatcromosoma[gen2] then ▷ Si los genes del
    individuos elegido en hijos son distintos
24:                swap(hijosatcromosoma[gen1], hijosatcromosoma[gen2])  ▷ Intercambio
    cromosomas
25:                fitness_hijo ← fitness_hijo ∪ hijosatcromosoma  ▷ Actualizo dispersion
26:            end if
27:            count ← count + 1
28:        end while
29:        worst_i, second_worst ← max(fitness_i)          ▷ Obtengo los dos peores padres
30:        poblacion ← poblacion ∩ {worst_i, second_worst}
31:        poblacion ← poblacion ∪ hijos  ▷ Sustituyo los peores padres por los mejores hijos
32:        iters ← iters + 1
33:    end while
34:    hijo ← min(poblacion)
35:    return hijo
36: end function
```

```
1: function AGE_posicion()
2:     TAM = 50
3:     MAX_ITERS = 100000
4:     iters = 0
5:     poblacion ← ∅                                   ▷ Matriz de TAM individuos
6:     hijo ← ∅                                        ▷ Este sera el hijo solucion
7:     hijos ← ∅
8:     fitness_i ← ∅
9:     fitness_hijo ← ∅
10:    for row in poblacion do
11:        row ← generarPoblacion()
12:        fitness_i ← fitness_i ∪ diff(row)
13:    end for
14:    while iters < MAX_ITERS do
15:        selected ← seleccion(poblacion, fitness_i, 2)        ▷ Eligo los 2 padres
16:        hijos ← generarHijosPosicion(selected[0], selected[1])   ▷ Cruzo los 2 padres
17:        spected_mutations ← 0,1 * n * 2
18:        count = 0
19:        while count! = spected_mutations do
20:            cromosoma ← Random(0, |hijos|)          ▷ Elegimos el Individuo a mutar
21:            gen1 ← Random(0, n)
22:            gen2 ← Random(0, n)                     ▷ Elegimos dos genes Diferentes
23:            if hijosatcromosoma[gen1]! = hijosatcromosoma[gen2] then ▷ Si los genes del
   individuos elegido en hijos son distintos
24:                swap(hijosatcromosoma[gen1], hijosatcromosoma[gen2])   ▷ Intercambio
   cromosomas
25:                fitness_hijo ← fitness_hijo ∪ hijosatcromosoma   ▷ Actualizo dispersion
26:            end if
27:            count ← count + 1
28:        end while
29:        worst_i, second_worst ← max(fitness_i)      ▷ Obtengo los dos peores padres
30:        poblacion ← poblacion ∩ {worst_i, second_worst}
31:        poblacion ← poblacion ∪ hijos  ▷ Sustituyo los peores padres por los mejores hijos
32:        iters ← iters + 1
33:    end while
34:    hijo ← min(poblacion)
35:    return hijo
36: end function
```

5.2. Algoritmo Memético

Este algoritmo consiste en hibridar el AGG con una busqueda local, sabemos que BL obtiene buenos resultados en poco tiempo, por lo que la idea de este algoritmo seria mejorar la poblacion mediante BL, de esta forma la solucion seria bastante aproximada, pero si seguimos ejecutando cruces y mutaciones poder salir del espacio de busqueda local para seguir explorando mejores soluciones.
AM obtiene muy buenos resultados en muy poco tiempo, vamos a implementar 3 tipos de AM.
La diferencia de entre estos tres algoritmos Memeticos será sobre que individuos aplicar la BL.

- **AM(10,1.0)** Este tipo se caracteriza por realizar BL sobre toda la poblacion, de esta forma obtenemos muy buenas poblaciones, por lo que a la hora de seguir cruzando y mutando los individuos, como comparten genes seguiremos obtendendo buenos resultados.
  Consume más tiempo que el resto.
- **AM(10,0.1)** Si aplicamos BL sobre un subconjunto de la poblacion mejoraremos, un poco esos individuos, pero como el cruce se hace con toda la poblacion no se mejora tanto la poblacion, o en cualquir caso tardara mas iteraciones en mejorar toda la solucion
- **AM(10,0.1mej)** En el ultimo caso solo aplicamos BL sobre los mejores individuos (0.1) esto implica que esos individuos obtienen muy buenas soluciones, pero al cruzarse con una poblacion no tan buenas tienden a no ser muy buena idea, aunque igualmente obtiene buenos resultados

Cabe destacar que el algoritmo de BL usado es el mismo que el de la practica 1 excepto que en vez de generar una solucion se la pasamos como parametro, como esa solucion que le pasamos esta en representacion binaria tenemos que transformarla a representacion entera y una vez encontrada la solucion volver a pasarla a representación binaria.
En todos los AM la busqueda local se ejecutará cada 10 iteraciones.

```
1: function AM_all()
2:     TAM = 50
3:     MAX_ITERS = 100000
4:     iters = 0
5:     poblacion ← ∅                                   ▷ Matriz de TAM individuos
6:     hijo ← ∅                                        ▷ Este sera el hijo solucion
7:     hijos ← ∅
8:     fitness_i ← ∅
9:     for row in poblacion do
10:        row ← generarPoblacion()
11:        fitness_i ← fitness_i ∪ diff(row)
12:    end for
13:    while iters < MAX_ITERS do
14:        best_father ← best(poblacion)               ▷ Individuo de la poblacion con la menor
   dispersión
15:        for row in |poblacion| do
16:            padre2 ← next(row)                      ▷ Siguiente individuo
17:            if boolRandom(0,7) then
18:                hijos ← hijos ∪ generarHijosPosicion(row, padre2)
19:                fitness_i ← fitness_i ∪ diff(hijos)  ▷ Actualizamos la dispersion de los
   nuevos hijos
20:            else
21:                hijos ← hijos ∪ row
22:            end if
23:        end for
24:        spected_mutations ← 0,1 ∗ n ∗ m
25:        count = 0
26:        while count! = spected_mutations do
27:            cromosoma ← Random(0, TAM)              ▷ Elegimos el Individuo a mutar
28:            gen1 ← Random(0, n)
29:            gen2 ← Random(0, n)                     ▷ Elegimos dos genes Diferentes
30:            if hijosatcromosoma[gen1]! = hijosatcromosoma[gen2] then ▷ Si los genes del
   individuos elegido en hijos son distintos
31:                swap(hijosatcromosoma[gen1], hijosatcromosoma[gen2])
32:            end if
33:            count ← count + 1
34:        end while
35:        poblacion ← hijos                           ▷ Los hijos sustituyen a la poblacion actual
36:        if !find(fitness_i, best_father) then       ▷ Si no encuentro el mejor individuo de la
   poblacion sustituyo el peor por el mejor de la anterior
37:            poblacionatpeorelemento ← best_father
38:        end if
39:        if iters == 10 ∗ k then                     ▷ Cada 10 iters
40:            fitness_i ← ∅
41:            for row in |poblacion| do
42:                row = BL(row)                       ▷ Actualizoel individuo aplicandole la BL
43:                fitness_i ← fitness_i ∪ diff(row)                                    20
44:            end for
45:        end if
46:        iters ← iters + 1
47:    end while
48:    hijo ← min(poblacion)
49:    return hijo
50: end function
```

```
1: function AM_subset()
2:     TAM = 50
3:     MAX_ITERS = 100000
4:     iters = 0
5:     poblacion ← ∅                                   ▷ Matriz de TAM individuos
6:     hijo ← ∅                                        ▷ Este sera el hijo solucion
7:     hijos ← ∅
8:     fitness_i ← ∅
9:     for row in poblacion do
10:        row ← generarPoblacion()
11:        fitness_i ← fitness_i ∪ diff(row)
12:    end for
13:    while iters < MAX_ITERS do
14:        best_father ← best(poblacion)               ▷ Individuo de la poblacion con la menor
   dispersión
15:        for row in |poblacion| do
16:            padre2 ← next(row)                      ▷ Siguiente individuo
17:            if boolRandom(0,7) then
18:                hijos ← hijos ∪ generarHijosPosicion(row, padre2)
19:                fitness_i ← fitness_i ∪ diff(hijos)  ▷ Actualizamos la dispersion de los
   nuevos hijos
20:            else
21:                hijos ← hijos ∪ row
22:            end if
23:        end for
24:        spected_mutations ← 0,1 * n * m
25:        count = 0
26:        while count! = spected_mutations do
27:            cromosoma ← Random(0, TAM)              ▷ Elegimos el Individuo a mutar
28:            gen1 ← Random(0, n)
29:            gen2 ← Random(0, n)                     ▷ Elegimos dos genes Diferentes
30:            if hijosatcromosoma[gen1]! = hijosatcromosoma[gen2] then ▷ Si los genes del
   individuos elegido en hijos son distintos
31:                swap(hijosatcromosoma[gen1], hijosatcromosoma[gen2])
32:            end if
33:            count ← count + 1
34:        end while
35:        poblacion ← hijos                           ▷ Los hijos sustituyen a la poblacion actual
36:        if !find(fitness_i, best_father) then       ▷ Si no encuentro el mejor individuo de la
   poblacion sustituyo el peor por el mejor de la anterior
37:            poblacionatpeorelemento ← best_father
38:        end if
39:        if iters == 10 * k then                     ▷ Cada 10 iters
40:            fitness_i ← ∅
41:            for row in |poblacion| do
42:                if Random < bool > (0,1) then       ▷ Con un 10% de probabilidad de aplicar
   BL
43:                    row = BL(row)
44:                    fitness_i ← fitness_i ∪ diff(row)  ▷ Actualizo fitness para ese
   individuo
45:                end if
46:            end for
47:        end if
48:        iters ← iters + 1
49:    end while
50:    hijo ← min(poblacion)
```

```
1: function AM_bests()
2:     TAM = 50
3:     MAX_ITERS = 100000
4:     iters = 0
5:     poblacion ← ∅                                   ▷ Matriz de TAM individuos
6:     hijo ← ∅                                        ▷ Este sera el hijo solucion
7:     hijos ← ∅
8:     fitness_i ← ∅
9:     for row in poblacion do
10:        row ← generarPoblacion()
11:        fitness_i ← fitness_i ∪ diff(row)
12:    end for
13:    while iters < MAX_ITERS do
14:        best_father ← best(poblacion)               ▷ Individuo de la poblacion con la menor
   dispersión
15:        for row in |poblacion| do
16:            padre2 ← next(row)                      ▷ Siguiente individuo
17:            if boolRandom(0,7) then
18:                hijos ← hijos ∪ generarHijosPosicion(row, padre2)
19:                fitness_i ← fitness_i ∪ diff(hijos)  ▷ Actualizamos la dispersion de los
   nuevos hijos
20:            else
21:                hijos ← hijos ∪ row
22:            end if
23:        end for
24:        spected_mutations ← 0,1 * n * m
25:        count = 0
26:        while count! = spected_mutations do
27:            cromosoma ← Random(0, TAM)              ▷ Elegimos el Individuo a mutar
28:            gen1 ← Random(0, n)
29:            gen2 ← Random(0, n)                     ▷ Elegimos dos genes Diferentes
30:            if hijosatcromosoma[gen1]! = hijosatcromosoma[gen2] then ▷ Si los genes del
   individuos elegido en hijos son distintos
31:                swap(hijosatcromosoma[gen1], hijosatcromosoma[gen2])
32:            end if
33:            count ← count + 1
34:        end while
35:        poblacion ← hijos                           ▷ Los hijos sustituyen a la poblacion actual
36:        if !find(fitness_i, best_father) then       ▷ Si no encuentro el mejor individuo de la
   poblacion sustituyo el peor por el mejor de la anterior
37:            poblacionatpeorelemento ← best_father
38:        end if
39:        best_5 ← sort_ind(poblacion, 0,1 * TAM)     ▷ Obtengo el 0.1 de los mejores
   individuos
40:        if iters == 10 * k then                     ▷ Cada 10 iters
41:            fitness_i ← ∅
42:            for row in best_5 do                                              22
43:                row = BL(row)
44:                fitness_i ← fitness_i ∪ diff(row)   ▷ Actualizo fitness para ese individuo
45:            end for
46:        end if
47:        iters ← iters + 1
48:    end while
49:    hijo ← min(poblacion)
50:    return hijo
51: end function
```

6. Conclusión Práctica 2.

Recordando las conclusiones que obtuvimos en la practica 1 ampliamos.

Estos tipos de algoritmos son muy dependientes del número de iteraciones que realizan, ya que en muchos casos mientras realizaba pruebas encontraba mejores resultados para los test simplemente aumentado el número de iteraciones, eso si, penalizando por un consumo de tiempo mayor.
Ademas la mayoria de estos algorimos encuentra la solucion puesta como mejor e incluso la mejoran en los primeros casos, con esto observamos que cuando nos enfrentamos a problemas con un tamaño pequeño, tenemos una alta probabilida de sacar la solucion perfecta o muy aproximada.

Algoritmo	Desv	Tiempo
Greedy	**0,6646633915**	**32,88**
BL	**0,4815112495**	**706,84**
AGG-uniforme	**0,409546259**	**25331,42**
AGG-posición	**0,5215188632**	**3517,38**
AGE-uniforme	**0,483662062**	**37863,06**
AGE-posición	**0,470069908**	**1843,76**
AM-(10,1.0)	**0,322516584**	**5011,3**
AM-(10,0.1)	**0,427105774**	**3654,3**
AM-(10,0.1mej)	**0,415776803**	**3664,36**

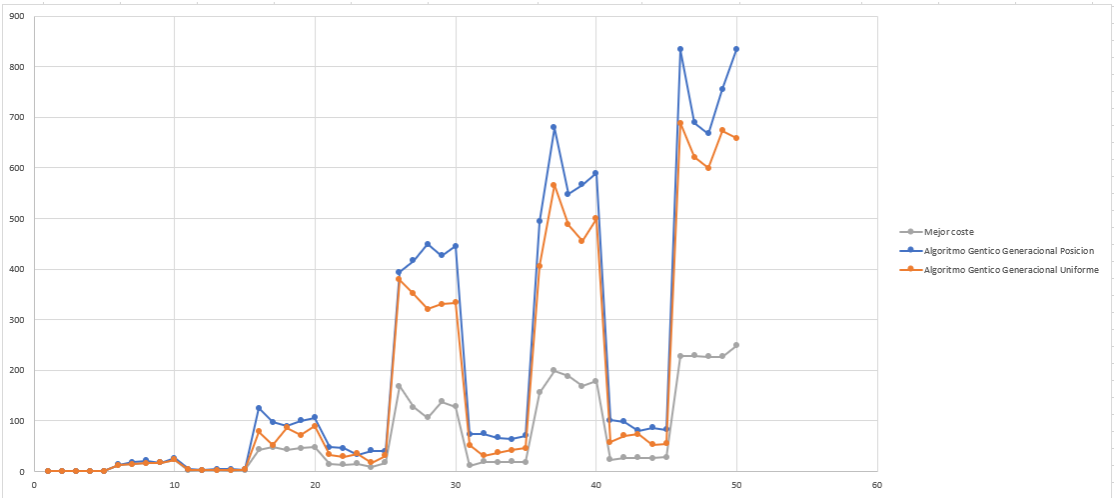

Figura 6.1: Desviación AGG

Comparando los resultados obtenidos de los AGG observamos que obtenemos mejores resultados mediante el cruce Uniforme como habiamos indicado antes, pero sin embargo cuando vemos los tiempos en Figura 6.4 observamos que este cruce es muy costoso en tiempo

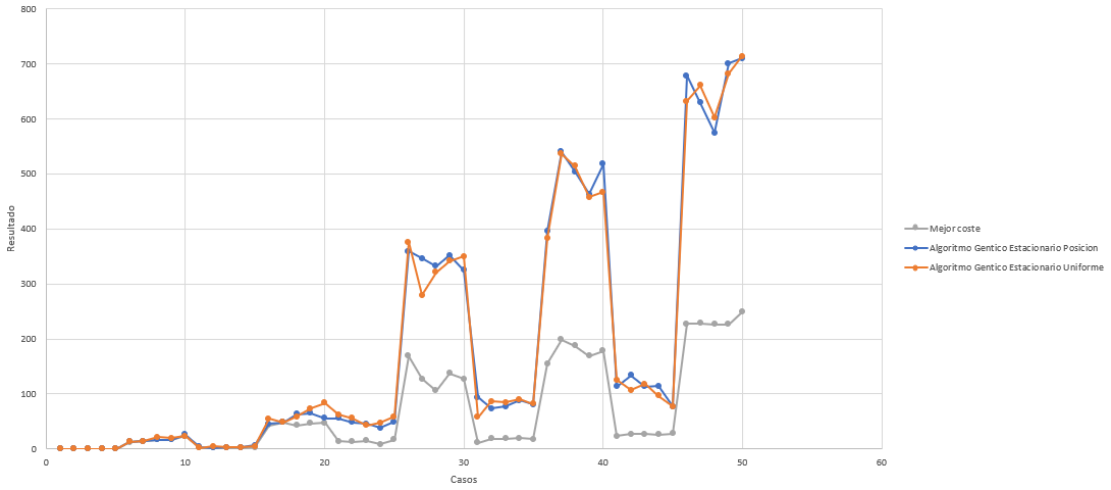

Figura 6.2: Desviación AGE

Si seguimos comparando los AGE llegamos a la misma conclusion, y de media vemos que obtenemos mejores resultados que el AGG

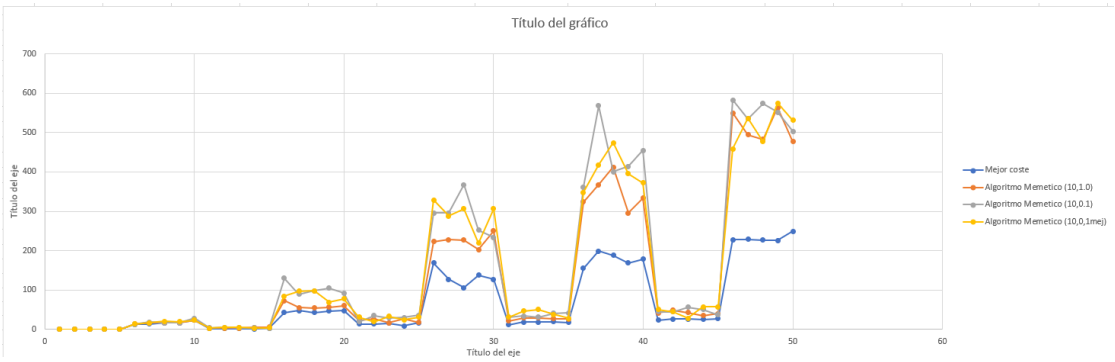

Figura 6.3: Desviación AM

Como vemos de media AM obiene mejores resultados que AGG y AGE ademas no tenemos el problema de tiempos provocado por el cruce uniforme. Como dije antes corroboramos que el mejor algoritmo de AM es el que ejecuta BL sobre toda la poblacion sin embargo, tambien vemos que eso conlleva un ligero incremento en tiempo de cómputo,

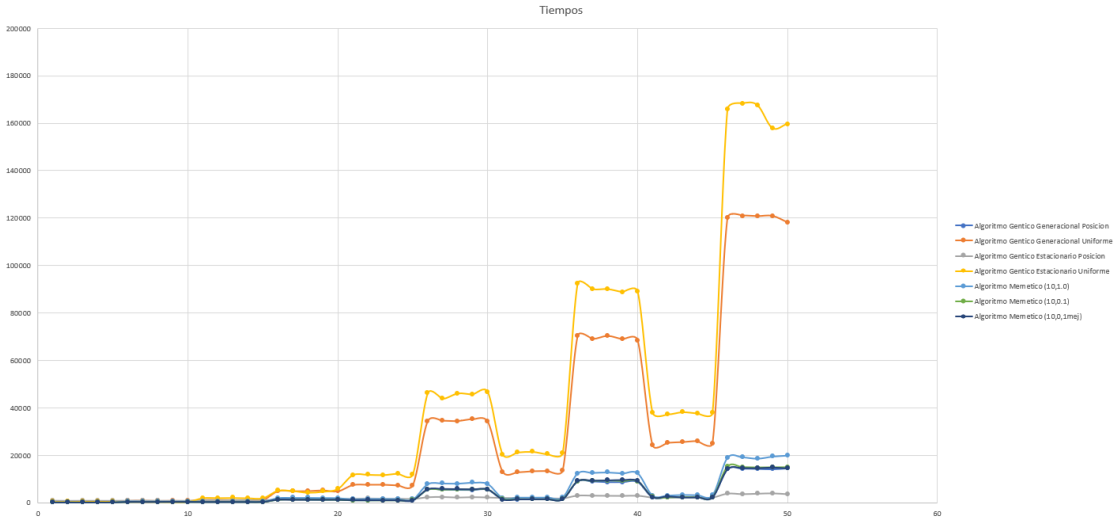

Figura 6.4: Desviación distintos Algoritmos

7. P3: Trayectorias

Tabla de resultados de la practica 3

Algoritmo	Desv	Tiempo
Greedy	**0,6646633915**	**32,88**
BL	**0,4815112495**	**706,84**
BMB	**0,4013**	**1848,1**
Enfriamiento Simulado	**0,5353**	**110,26**
ILS	**0,55171**	**2438,48**
ILS-ES	**0,38508**	**1075,46**

Notas a tener en cuenta
->He modiciado un poca la busqueda local y la he llamado BL_2, esta se diferencia de la original en que no genera una solucion inicial sino que se le pasa a la funcion como parametro, ademas de pasarle el numero de iteraciones que tiene que reailizar antes de parar y tambien devuelve por referencia el valor de la dispersion para la solucion encontrada por la busquda local (esto lo hago para reducir el tiempo de computo ya que la funcion DIFF que es la que calcula el fitness de un solucion es costosa), pero en esencia es la misma que en practica 1.

Pseudocodigo de **generarSolucionAleatoria()**

```
1: function generarSolucionAleatoria()
2:     solucion ← ∅
3:     for i = 0 in m do
4:         rand ← Random(0, n − 1)
5:         solucion ← solucion ∩ rand          ▷ Añadimos a la solucion si rand ∉ solucion
6:     end for
7:     return solucion
8: end function
```

Pseudocodigo de **generarVecino(solucion,fit_vecino)**

```
1: function generarVecino(solucion, fit_vecino)
2:     vecino ← solucion
3:     cambio ← Random(0, m), Random(0, n)     ▷ Seleccione un indice y lo cambio por otro
   elemento
4:     fit_vecino ← distFactorizada(vecino, cambio)
5:     return vecino
6: end function
```

Pseudocodigo de **EnfriamientoSimulado(MAX_ITERS, fit_solucion)**

```
1: function EnfriamientoSimulado(MAX_ITERS, fit_solucion)
2:     solucion ← ∅
3:     best ← ∅
4:     s_i ← ∅
5:     max_vecinos ← 10 * n
6:     max_exitos ← 0,1 * max_vecinos
7:     M ← MAX_ITERS/max_vecinos
8:     n_vecinos ← 1
9:     n_exitos ← 1
10:    solucion ← generarSolucionAleatoria()
11:    coste_ini ← diff(solucion)                    ▷ Calculo el coste de la sol inicial
12:    best ← solucion
13:    best_fit ← coste_ini
                                                     ▷ Inicializamos la temperatura
14:    T_ini ← 0,3 * coste_ini / −log(0,3)
15:    T_fin ← 0,0001
16:    T_curr ← T_ini
17:    iters ← 0
18:    β ← (T_ini − T_fin) / (M * T_ini * T_fin)
19:    curr_fit ← coste_ini
20:    while iters < MAX_ITERS and n_exitos! = 0 do
21:        n_vecinos ← 0
22:        n_exitos ← 0
23:        while n_vecinos < max_vecinos and n_exitos < max_exitos do
24:            vecino_fit ← 0,0
25:            iters ← iters + 1
26:            n_vecnios ← n_venicos + 1
27:            s_i ← generarVecino(solucion, vecino_fit)
28:            diferencia ← |vecino_fit − curr_fit|
29:            if vecino_fit < curr_fit or RandFloat(0,1) < e^(−diferencia / T_curr) then
30:                n_exitoss ← n_exitos + 1
31:                solucion ← s_i
32:                curr_fit ← vecino_fit
33:                if vecino_fit < best_fit then
34:                    best ← solucion
35:                    best_fit ← vecino_fit
36:                end if
37:            end if
38:        end while
39:        T_curr ← T_curr / (1 + β * T_curr)
40:    end while
41:    sol_fit ← best_fit                            ▷ Lo devuelvo por referencia
42:    return best
43: end function
```

Pseudocodigo de **BMB(n_sol, MAX_ITERS)**

```
1:  function BMB(n_sol, MAX_ITERS)
2:      solucion ← ∅
3:      best ← ∅
4:      best_fit ← ∞
5:      iters ← 0
6:      while iters < n_sol do
7:          iters ← iters + 1
8:          solucion ← generarSolucionAleatoria()        ▷ Aplico la BL sobre esa solucion aleatoria
9:          BL_sol ← BL_2(solucion, MAX_ITERS, sol_fit)
10:         if sol_fit < best_fit then
11:             best ← BL_sol
12:             best_fit ← sol_fit
13:         end if
14:     end while
15:     return best
16: end function
```

Pseudocodigo de **mutarSolucion(solucion, percent)**

```
1:  function mutarSolucion(solucion, percent)
2:      mutacion ← solucion
3:      t ← percent * m
4:      count ← 0
5:      while count < t do  ▷ Eligo un gen para mutar y lo cambio por uno que no este ya en el individuo
6:          gen ← Random(0, m − 1)
7:          count ← count + 1
8:          cambio ← Random(0, n − 1)                    ▷ Mutamos si si cambio ∉ solucion
9:          mutacion.at(muta) ← cambio
10:     end while
11:     return mutacion
12: end function
```

Pseudocodigo de **ILS(n_sol, BL_ITERS)**

```
1: function ILS(n_sol, ILS_ITERS)
2:     solucion ← generarSolucionAleatoria()
3:     curr_fit ← 0
4:     best ← solucion
5:     best_fit ← diff(solucion)
6:     iters ← 0
7:     while iters < n_sol do
8:         iters ← iters + 1
9:         BL_sol ← BL_2(solucion, BL_ITERS, curr_fit)      ▷ devuelve el fit por referencia
10:        if curr_fit < best_fit then
11:            best ← BL_sol
12:            best_fit ← curr_fit
13:        end if
14:        solucion ← mutarSolucion(best, 0,3)
15:    end while
16:    return best
17: end function
```

Pseudocodigo de **ILS(n_sol, ES_ITERS)**

```
1: function ILS(n_sol, ILS_ITERS)
2:     solucion ← generarSolucionAleatoria()
3:     curr_fit ← 0
4:     best ← solucion
5:     best_fit ← diff(solucion)
6:     iters ← 0
7:     while iters < n_sol do
8:         iters ← iters + 1
9:         BL_sol ← EnfriamientoSimulado(solucion, ES_ITERS, curr_fit) ▷ devuelve el fit
   por referencia
10:        if curr_fit < best_fit then
11:            best ← BL_sol
12:            best_fit ← curr_fit
13:        end if
14:        solucion ← mutarSolucion(best, 0,3)
15:    end while
16:    return best
17: end function
```