Práctica 1

Búsqueda basada en Poblaciones y Trayectorias Simples: Búsqueda Local, Algoritmos Genéticos y Algoritmos Meméticos.

Problema de la Asignación Cuadrática (QAP)

Metaheurísticas Curso Academico 2016/2017

Algoritmos considerados:

- Greedy
- Búsqueda local del primer mejor (con Don't Look Bits)
- Algoritmo genético generacional: Cruce Posicional
- Algoritmo genético generacional: Cruce OX
- Algoritmo genético generacional: Cruce PMX
- Algoritmo genético estacionario: Cruce Posicional
- Algoritmo genético estacionario: Cruce OX
- Algoritmo genético estacionario: Cruce PMX
- Algoritmo memético 10 generaciones, toda la población.
- Algoritmo memético 10 generaciones, 10 % población aleatorio.
- Algoritmo memético 10 generaciones, 10 % población mejor.

Nombre: David Criado Ramón

Correo electrónico: davidcr96@correo.ugr.es

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

1. Descripción del problema…… 3
2. Breve descripción de los algoritmos aplicados al problema……. 4
2.1 Representación de la solución…… 4
2.1.1 Cálculo directo del coste…….. 4
2.1.2 Cálculo factorizado del coste…….. 4
2.2 Generación de soluciones aleatorias……. 5
2.2.1 Generación de una solución aleatoria…… 5
2.2.2 Generación de una población aleatoria……. 5
2.3 Operación de generación de vecino (y mutación)…….. 5
2.4 Operador de selección para algoritmos genéticos……. 6
2.5 Operadores de cruce para algoritmos genéticos…… 6
2.5.1 Cruce posicional…… 6
2.5.2 Cruce OX…….. 7
2.5.3 Cruce PMX…… 7
3. Método de búsqueda y operadores de los algoritmos…… 8
3.1 Algoritmo greedy…… 8
3.2 Búsqueda local del primer mejor …….. 8
3.3 Algoritmos genéticos generacionales…….. 9
3.4 Algoritmos genéticos estacionarios……10
3.5 Algoritmos meméticos……..11
3.5.1 Algoritmo memético selección porcentaje mejor……..11
3.5.2 Algoritmo memético selección porcentaje aleatorio……11
4. Procedimiento considerado para realizar la práctica……..12
5. Experimentos y análisis de resultados……13
5.1 Resultados de la aplicación de los algoritmos a los casos con una única semilla……..13
5.2 Resultados de la aplicación de los algoritmos a los casos con 100 semillas……22

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, nuestra función objetivo viene determinada 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 Calculo 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 Calculo 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])
Devolver coste
```

2.2 Generación de soluciones aleatorias

Para obtener una solución inicial o una población inicial completa se utilizan las siguiente formas de crear aleatorios. Puesto que se inicializan los mecanismos generadores de aleatorios dentro de las 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 vacío de soluciones únicas con su coste

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 (y mutación)

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]

2.4 Operador de selección para algoritmos genéticos

Para la selección en los algoritmos genéticos utilizamos el torneo binario. El torneo binario consiste en la selección de la mejor solución de entre dos aleatorias. Para optimizar su cálculo trabajamos directamente con el índice puesto que la población ya está ordenada.

```
// Partiendo de una población ordenada de menor a mayor coste (la mejor solución está antes) y un generador de aleatorios en uso
Hacer:
    competidor1 = aleatorio en [1, tamaño de población]
    competidor2 = aleatorio en [1, tamaño de población]
mientras competidor 1 sea igual a competidor2

Devolver el mínimo de entre competidor1 y competidor2
```

2.5 Operadores de cruce para algoritmos genéticos

Cada cruce dará lugar a dos hijos uno formado por uno de los progenitores como padre y el otro como madre y el otro justo del revés.

2.5.1 Cruce Posicional

El cruce posicional se basa en la semejanza directa de los padres. Deja en la misma posición los elementos de los padres que coincidan y el resto los rellenamos de manera aleatoria.

```
hijo = permutación vacía
restos = vector vacío
para cada i desde 1 hasta tamaño de los padres:
      si padre[i] es igual a madre[i]:
            hijo[i] = madre[i]
      si no:
            insertar madre[i] en restos

restos = baraje aleatorio de restos con generador en uso
para cada i desde 1 hasta tamaño del hijo:
      si hijo[i] está vacío:
            hijo[i] = elemento actual en restos
            pasar al siguiente elemento en restos
hijo.coste = cálculo directo del coste de hijo
Devolver hijo
```

2.5.2 Cruce OX

Este operador de cruce genera un hijo a partir de una subcadena del padre y preservando el orden de la madre para los elementos que no forman parte de dicha subcadena.

```
// Siendo N el tamaño de la permutación
tam subcadena = aleatorio con generador en uso en [N/8, N/6]
inicio = aleatorio con generado en uso en [1, N – padre – tam subcadena-1]
fin = inicio + tam
Para cada i desde inicio hasta fin:
      hijo[i] = padre[i]
      Quitamos hijo[i] de la madre
Para cada i desde 0 hasta N que no esté en [inicio, fin]:
      hijo[i] = primer elemento de madre
      quitar primer elemento de madre
hijo.coste = Cálculo directo del coste de hijo
Devolver hijo
```

2.5.3 Cruce PMX

De manera similar al OX, en el cruce PMX obtenemos una subcadena del padre y preservamos los datos de la madre, de manera que si el elemento fuera de la subcadena ya está en el hijo seguimos las correspondencias en la subcadena central entre los datos del padre y la madre para obtener un valor que no se encuentre en el hijo.

```
tam subcadena = aleatorio con generador en uso en [N/8, N/6]
inicio = aleatorio con generado en uso en [1, N – padre – tam subcadena-1]
fin = inicio + tam
Para cada i desde inicio hasta fin:
      hijo[i] = padre[i]
      insertar { padre[i] → madre[i]} en correspondencias.
Para cada elemento A en correspondencias:
      B = correspondencia de A (A → ? = B)
      mientras B esté en correspondencias y no se formen ciclos:
            C = correspondencia de B (B → C)
            arreglar correspondencia de A (A → C)
            B = C
Para cada i desde 0 hasta N que no esté en [inicio, fin]:
            si madre[i] está en correspondencias:
                  hijo[i] = correspondencias(madre[i])
            si no:
                  hijo[i] = madre[i]
hijo.coste = Cálculo directo del coste del hijo
Devolver hijo
```

3. Método de búsqueda y operaciones de los algoritmos

3.1 Algoritmo greedy

El algoritmo greedy nos permite obtener unas primeras soluciones (que no suelen ser de mucha calidad) en un tiempo muy rápido. El algoritmo greedy usado es el siguiente:

Para cada i desde 1 hasta el tamaño del problema:

Calcular su potencial de flujo $\hat{f}_i = \sum_{j=1}^{N} f_{ij}$ *y meterlo en flujos.*

Calcular su potencial de distancia $\hat{d}_i = \sum_{j=1}^{N} d_{ij}$ *y meterlo en distancias.*

solución = Solución vacía

para cada i desde 1 hasta el tamaño del problema:

unidad = Escoger la unidad de flujos con mayor flujo que no haya sido usada.

distancia = Escoger la localización de distancias con menor distancia que no haya sido usada.

solución[unidad] = distancia

Calcular coste de solución

Devolver solución con su coste

3.2 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
```

3.3 Algoritmo genético generacional

poblacion = Inicializar una población de soluciones aleatorias y ordenarla
*ncruces = Probabilidad de cruce * tamaño población / 2*
*nmutaciones = Probabilidad de mutación * tamaño población * tamaño individuo*
número de generaciones = 0
Mientras no se alcancen 50000 evaluaciones de la función objetivo:
seleccionados = Selección usando Torneo Binario tamaño población veces
i = 0

para cada par en seleccionados mientras que i < ncruces:
primero = cruce(padre1, padre2)
segundo = cruce(padre2, padre1)
insertar primero y segundo en nuevaPoblación
incrementar i

Completamos nuevaPoblación con el resto de seleccionados
Para cada i desde 1 hasta nmutaciones:
cromosoma = entero aleatorio en [1, tamaño de población]
gen1 = entero aleatorio en [1, tamaño del problema]
gen2 = entero aleatorio distinto de gen1 en [1, tamaño del problema]
Aplicar 2-Opt a nuevaPoblación[cromosoma] sobre gen1, gen2.

Ordenamos nuevaPoblación de menor a mayor coste
Si la mejor solución de población, A, no está en nuevaPoblación:
Si A es mejor solución que la peor de nuevaPoblación:
peor solución de nuevaPoblación = A

Intercambiar población y nuevaPoblación
Ordenar la población de menor a mayor coste
*Sumar a evaluaciones de la función objetivo 2 * ncruces + nmutaciones*
Incrementar número de generaciones

3.4 Algoritmo genético estacionario

poblacion = Inicializar una población de soluciones aleatorias

*nmutaciones = Probabilidad de mutación * tamaño población * tamaño individuo*

frecuencia = 1/nmutaciones (subido al siguiente entero)

Mientras no se alcancen 50000 evaluaciones de la función objetivo:

Ordenar la población de menor a mayor coste

seleccionados = Selección usando Torneo Binario 2 veces

i = 0

Insertar en nuevaPoblación cruce(seleccionado[1], seleccionado[2])

Insertar en nuevaPoblación cruce(seleccionado[2], seleccionado[1])

Si generación es múltiplo de frecuencia:

Para cada i desde 1 hasta nmutaciones:

cromosoma = entero aleatorio en [1, tamaño de población]

gen1 = entero aleatorio en [1, tamaño del problema]

gen2 = entero aleatorio distinto de gen1 en [1, tamaño del problema]

Aplicar 2-Opt a nuevaPoblación[cromosoma] sobre gen1, gen2.

Insertar en nuevaPoblación los dos últimos elementos de población

Ordenar nuevaPoblación de menor a mayor coste

Los dos últimos elementos de población son sobreescritos con los dos primeros elementos de nuevaPoblación

Sumar a evaluaciones de la función objetivo 2 + nmutaciones

Incrementar número de generación

3.5 Algoritmos meméticos

Los algoritmos meméticos utilizan el mismo esquema de evolución que el algoritmo generacional genético salvo que se realiza la integración justo al inicio del bucle que comprueba el número de evaluaciones de la siguiente manera:

3.5.1 Algoritmo memético selección porcentaje mejor

Aplicado también al caso en que se selecciona toda la población (porcentaje 1.0)

```
Si el número de generaciones es distinto de 0 y módulo 10:
      para cada i desde 1 hasta porcentaje * tamaño población:
            Aplicar BL a población[i]
            Aumentar número de evaluaciones usadas por la BL.
```

3.5.2 Algoritmo memético selección porcentaje aleatorio

```
Si el número de generaciones es distinto de 0 y módulo 10:
      s = generar permutación aleatoria del vector {1, 2 …, tamaño de la
          población }
      para cada i desde 1 hasta porcentaje * tamaño población:
            Aplicar BL a población[s[i]]
            Aumentar número de evaluaciones usadas por la BL.
```

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. La razón principal del uso de C++17 es el uso de *if constexpr*. Esto nos permite utilizar if en tiempo de compilación en templates y mantener el mismo código para AGG y algoritmos meméticos sin gastar tiempo inncesario en resolver los if que indican que está activado el memético en tiempo de ejecución.

El compilador utilizado para ejecutar el código es clang 3.9. El makefile depende directamente de este compilador (también es posible compilarlo con gcc-7 pero no puedo garantizar que funcione correctamente).

Para instalar clang 3.9 en Ubuntu 16.04 LTS realizar lo siguiente:

```
wget -O - http://apt.llvm.org/llvm-snapshot.gpg.key | sudo apt-key add -
sudo apt-add-repository "deb http://apt.llvm.org/xenial/ llvm-toolchain-xenial-3.9
main"
sudo apt-get update
sudo apt-get install clang-3.9 lldb-3.9
```

Se proporciona un único ejecutable que se puede generar con el makefile (hecho suponiendo que dispone de /usr/bin/clang++3.9). Ese ejecutable se encuentra precompilado para linux de 64 bits. Como parámetros se pasan en el siguiente orden: el código del algoritmo, el nombre del archivo a analizar o ALL (que ejecuta todos los casos propuestos de la práctica suponiendo que están en el directorio del trabajo), la semilla a usar para los pseudoaleatorios y un último parámetro que si está puesto a "on" muestra ciertos estadísticos extras para ver la convergencia (consumen más tiempo que la ejecución normal de los algoritmos).

5. Experimentos y análisis de resultados

5.1 Resultados de la aplicación de los algoritmos a los casos con una única semilla

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.

Algoritmo Greedy

Caso	*Desv*	*Tiempo*	*Caso*	*Desv*	*Tiempo*
Chr20b	*365,80*	*1,26E-04*	***Sko100a***	*13,23*	*0,00204918*
Chr22a	*119,92*	*1,27E-04*	***Sko100b***	*13,49*	*0,00190636*
Els19	*124,42*	*1,02E-04*	***Sko100c***	*14,53*	*0,00305694*
Esc32b	*90,48*	*2,22E-04*	***Sko100d***	*12,53*	*0,00213131*
Kra30b	*29,61*	*3,09E-04*	***Sko100e***	*13,25*	*0,000974227*
Lipa90b	*29,06*	*0,00255208*	***Tai30b***	*117,73*	*1,05E-04*
Nug25	*18,54*	*3,62E-04*	***Tai50b***	*71,83*	*0,000235959*
Sko56	*19,29*	*0,000759848*	***Tai60a***	*15,82*	*0,000327277*
Sko64	*17,63*	*0,000839182*	***Tai256c***	*125,05*	*0,00455488*
Sko72	*15,64*	*0,00118959*	***Tho150***	25,02	0,00145802

Tabla 5.1.1: 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.

Búsqueda Local

Caso	*Desv*	*Tiempo*	*Evaluaciones usadas*	*Caso*	*Desv*	*Tiempo*	*Evaluaciones usadas*
Chr20b	55,79	2,04E-03	600	***Sko100a***	2,87	0,341318	25100
Chr22a	8,48	2,23E-03	858	***Sko100b***	2,12	0,387337	28400
Els19	61,40	1,72E-03	817	***Sko100c***	2,62	0,429092	30100
Esc32b	19,05	7,57E-03	1888	***Sko100***	2,52	0,392947	26400

				d			
Kra30b	7,00	8,96E-03	1620	***Sko100e***	2,70	0,464806	27600
Lipa90b	22,13	0,148555	14670	***Tai30b***	31,78	7,22E-03	1920
Nug25	2,30	4,18E-03	1375	***Tai50b***	5,77	0,0488259	7350
Sko56	2,62	0,0498418	7672	***Tai60a***	5,57	0,0554907	6480
Sko64	4,02	0,0885282	11136	***Tai256c***	2,97	1,81283	50176
Sko72	3,00	0,136816	14976	***Tho150***	10,40	1,00824	50100

Tabla 5.1.2: Búsqueda Local

La búsqueda local nos ofrece soluciones mucho mejores que el algoritmo greedy gracias a la explotación que hace en el espacio de búsqueda a partir de la solución inicial que obtiene aleatoriamente. De hecho, en muchos casos concretos este es el algoritmo que obtiene la mejor solución. Gracias a la técnica *Don't Look Bits* para evitar explorar zonas que potencialmente no lleven a ninguna solución y al cálculo del coste factorizado ahorramos una gran cantidad de tiempo en comparación con lo que tardan el resto de metaheurísticas propuestas. Por tanto, si las soluciones Greedy no nos satisfacen que es lo habitual, lanzar una búsqueda local puede ayudarnos a encontrar soluciones ya de una cierta calidad en muy poco tiempo. El principal problema de la búsqueda local es quedar atascados en un óptimo local que difiera demasiado del óptimo global. Por temas de optimización puede superarse ligeramente el número de evaluaciones objetivos. No obstante, lo más habitual es que el algoritmo converja rápidamente hacia una mínimo local.

Para evaluar la convergencia se añaden nuevos estadísticos.

Algoritmo Genético Generacional – Cruce Basado en Posición

Caso	*Desv*	*Tiempo*		*Caso*	*Desv*	*Tiempo*
Chr20b	47,17	7,02E-01		***Sko100a***	9,86	10,261
Chr22a	35,38	8,78E-01		***Sko100b***	9,12	10,2841
Els19	44,30	6,09E-01		***Sko100c***	12,06	10,2373
Esc32b	30,95	1,40E+00		***Sko100d***	10,06	10,3312
Kra30b	9,04	1,21E+00		***Sko100e***	9,32	10,2882
Lipa90b	27,85	8,45392		***Tai30b***	11,84	1,21E+00
Nug25	13,14	8,75E-01		***Tai50b***	12,81	2,98552

Sko56	8,66	3,59508		***Tai60a***	6,63	4,02553
Sko64	11,60	4,51064		***Tai256c***	8,33	55,3575
Sko72	12,00	5,81379		***Tho150***	22,13	21,4285

Tabla 5.1.3: AGG-Posición

Algoritmo Genético Generacional – Cruce OX

Caso	*Desv*	*Tiempo*	*Caso*	*Desv*	*Tiempo*
Chr20b	33,42	8,94E-01	***Sko100a***	2,87	0,341318
Chr22a	15,11	1,01E+00	***Sko100b***	2,12	0,387337
Els19	51,31	8,04E-01	***Sko100c***	2,62	0,429092
Esc32b	38,10	1,80E+00	***Sko100d***	2,52	0,392947
Kra30b	9,31	1,65E+00	***Sko100e***	2,70	0,464806
Lipa90b	23,41	10,9407	***Tai30b***	31,78	7,22E-03
Nug25	5,34	1,20E+00	***Tai50b***	5,77	0,0488259
Sko56	3,67	4,70735	***Tai60a***	5,57	0,0554907
Sko64	4,00	5,87752	***Tai256c***	2,97	1,81283
Sko72	4,08	7,30797	***Tho150***	10,40	1,00824

Tabla 5.1.4: AGG-OX

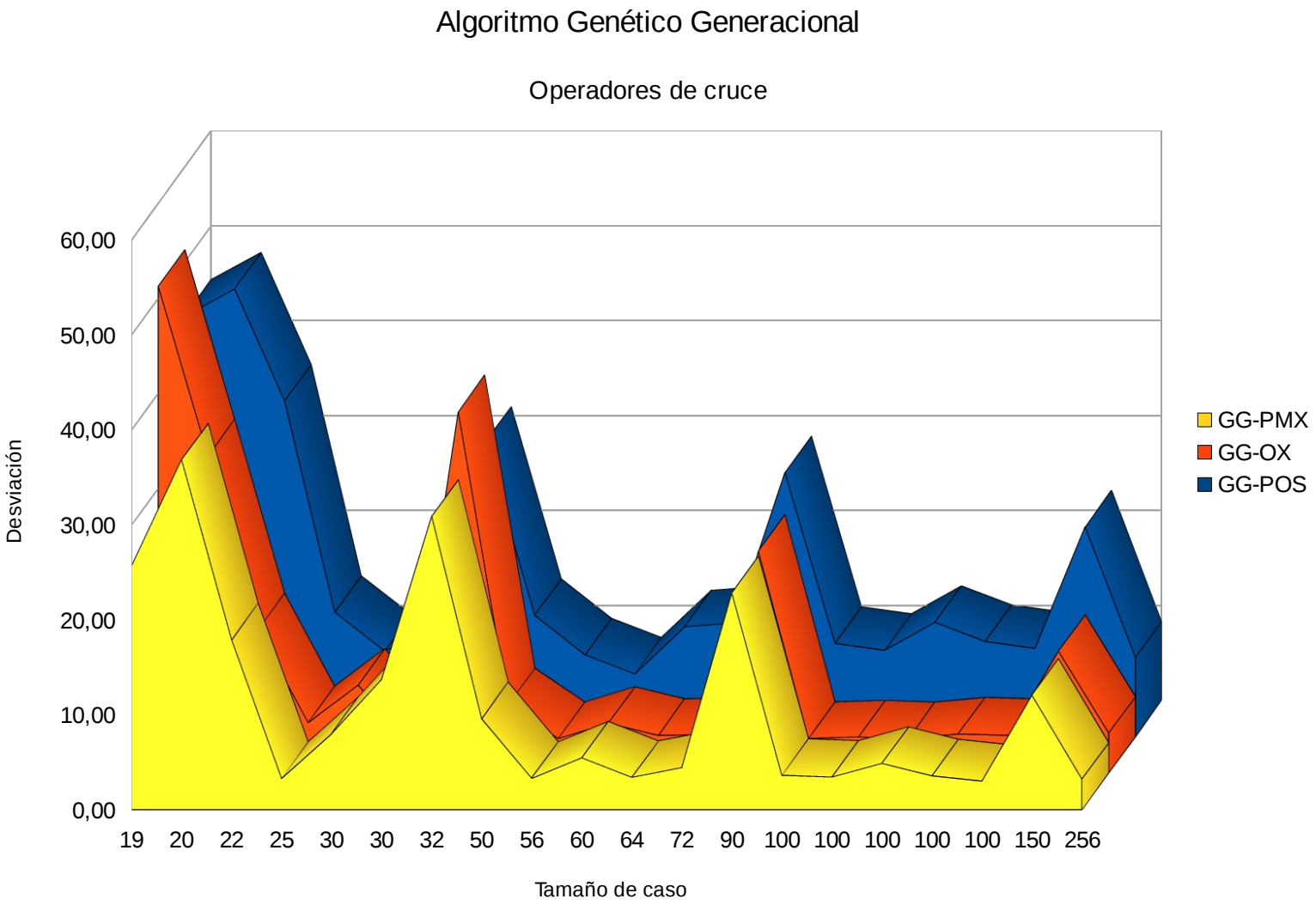
Algoritmo Genético Generacional – Cruce PMX

Caso	*Desv*	*Tiempo*	*Caso*	*Desv*	*Tiempo*
Chr20b	36,90	6,34E-01	***Sko100a***	3,65	9,93312
Chr22a	17,93	7,05E-01	***Sko100b***	3,46	9,92183
Els19	25,68	5,81E-01	***Sko100c***	4,90	9,91349
Esc32b	30,95	1,29E+00	***Sko100d***	3,59	9,93186
Kra30b	8,02	1,16E+00	***Sko100e***	3,04	9,89237
Lipa90b	22,87	8,25128	***Tai30b***	13,75	1,18E+00
Nug25	3,31	8,65E-01	***Tai50b***	9,60	2,84576
Sko56	3,32	3,5296	***Tai60a***	5,48	3,95356
Sko64	3,43	4,39627	***Tai256c***	3,25	55,15
Sko72	4,45	5,51884	***Tho150***	12,07	21,0246

Tabla 5.1.5: AGG-PMX

A diferencia de la búsqueda local, los algoritmos genéticos toman una estrategia más explorativa, partiendo de diferentes soluciones aleatorias vamos combinándolas en los cruces, permitiendo así utilizar soluciones ligeramente más malas para generar hijos que

potencialmente serían mejores que siguiendo sólo una trayectoria simple como ocurre en el caso de la búsqueda local. Esta, no obstante es también la técnica más costosa en cuanto a tiempo de ejecución relativo (principalmente por que para la mutación y la búsqueda local podemos aplicar un cálculo de coste factorizado), pero puede llegar a ofrecer resultados muy interesantes en ciertos casos. Se han considerado 3 operadores de cruce para los AGG: basado en posición, OX y PMX



Mientras que es evidente que el basado en posición, como cabía esperar, da unos resultados bastante peores, no podemos asegurar que el PMX sea mejor que el OX o viceversa.

Algoritmo Genético Estacionario – Cruce Basado en Posición

Caso	*Desv*	*Tiempo*	*Caso*	*Desv*	*Tiempo*
Chr20b	78,68	9,47E-01	***Sko100a***	9,47	3,88342

Chr22a	10,62	9,78E-01	***Sko100b***	10,24	3,90147
Els19	41,41	1,24E+00	***Sko100c***	10,52	3,89058
Esc32b	26,19	1,41E+00	***Sko100d***	10,30	3,86958
Kra30b	6,10	1,34E+00	***Sko100e***	10,40	3,89058
Lipa90b	25,59	3,64906	***Tai30b***	0,24	1,30E+00
Nug25	4,01	1,10E+00	***Tai50b***	7,12	1,95473
Sko56	4,11	2,34252	***Tai60a***	8,52	2,15971
Sko64	7,25	2,38741	***Tai256c***	10,05	12,014
Sko72	8,16	2,92716	***Tho150***	21,17	6,53227

Tabla 5.1.6: AGE-Posicional

Algoritmo Genético Estacionario – Cruce OX

Caso	*Desv*	*Tiempo*	*Caso*	*Desv*	*Tiempo*
Chr20b	32,55	1,10E+00	***Sko100a***	8,44	4,87363
Chr22a	9,84	1,20E+00	***Sko100b***	7,31	4,85685
Els19	38,13	1,50E+00	***Sko100c***	8,48	4,86913
Esc32b	14,29	1,87E+00	***Sko100d***	8,56	4,87398
Kra30b	1,24	1,69E+00	***Sko100e***	8,96	4,88921
Lipa90b	25,41	4,65429	***Tai30b***	0,38	1,62E+00
Nug25	3,31	1,34E+00	***Tai50b***	9,19	2,49173
Sko56	4,12	2,99691	***Tai60a***	6,37	2,73459
Sko64	4,02	3,01846	***Tai256c***	5,64	15,1156
Sko72	5,26	3,68826	**Tho150**	19,63	8,15999

Tabla 5.1.7: AGE-OX

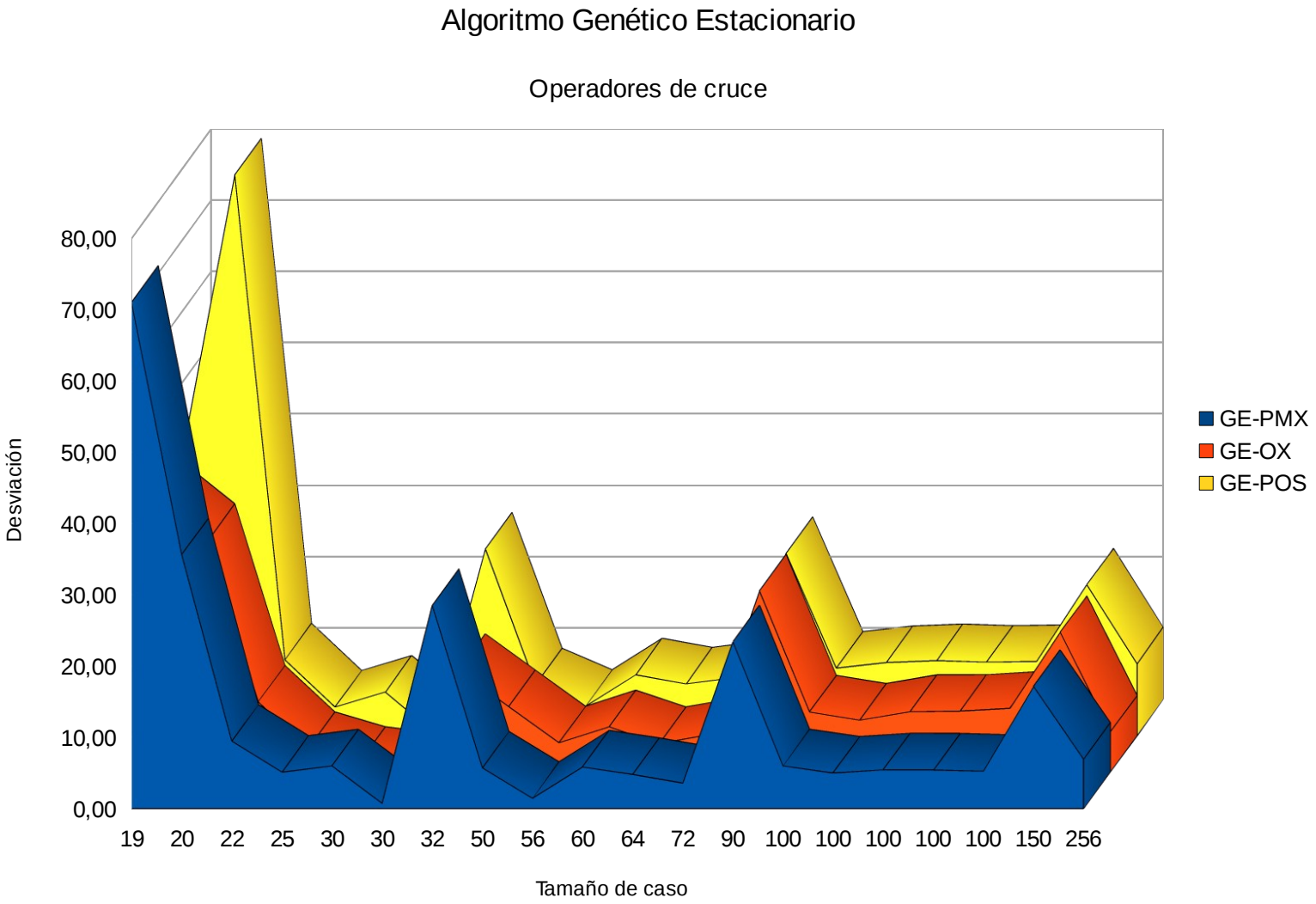
Algoritmo Genético Estacionario – Cruce PMX

Caso	*Desv*	*Tiempo*	*Caso*	*Desv*	*Tiempo*
Chr20b	35,68	9,52E-01	***Sko100a***	6,02	4,15752
Chr22a	9,49	9,98E-01	***Sko100b***	5,02	3,9541
Els19	71,12	1,27E+00	***Sko100c***	5,47	3,93121
Esc32b	28,57	1,43E+00	***Sko100d***	5,44	3,94928
Kra30b	6,02	1,36E+00	***Sko100e***	5,26	3,92655
Lipa90b	23,45	3,68658	***Tai30b***	0,75	1,33E+00
Nug25	5,13	1,10E+00	***Tai50b***	5,73	1,98532
Sko56	1,45	2,37159	***Tai60a***	5,86	2,18544

Sko64	4,82	2,42697	***Tai256c***	6,95	12,3068
Sko72	3,60	2,93507	**Tho150**	17,13	6,55225

Tabla 5.1.8: AGE-PMX

Los algoritmos genéticos estacionarios nos ofrecen soluciones de una calidad similar a la de los generacionales (a veces un poco peores, a veces un poco mejores) reduciendo considerablemente el tiempo requerido para su ejecución.



Mientras que el operador de cruce basado en posición ha mejorado considerablemente para el caso estacionario, OX ha mantenido una relación similar con su equivalente generacional mientras que PMX ha empeorado considerablemente en los tamaños de caso más bajos, aumentando mucho su desviación media. Si quitamos dichos casos iniciales, que pueden darse debido a que las soluciones iniciales no fuera suficientemente buenas. Se sigue manteniendo el hecho de que por el basado en posición queda por debajo de OX/PMX que se mantienen en un entorno similar. No obstante, el cruce basado en posición ha mejorado considerablemente con respecto a su equivalente en el AGG.

Algoritmo Memético 10-1.0

Caso	*Desv*	*Tiempo*	*Caso*	*Desv*	*Tiempo*

Chr20b	9,57	1,87E-01	***Sko100a***	8,47	1,06551
Chr22a	2,99	2,27E-01	***Sko100b***	9,85	1,07704
Els19	4,56	1,69E-01	***Sko100c***	9,95	1,06496
Esc32b	19,05	3,00E-01	***Sko100d***	9,52	1,06556
Kra30b	5,09	2,81E-01	***Sko100e***	10,18	1,06195
Lipa90b	24,22	1,03132	***Tai30b***	4,41	2,81E-01
Nug25	2,56	2,21E-01	***Tai50b***	8,32	0,471841
Sko56	7,68	0,592616	***Tai60a***	6,26	0,609402
Sko64	8,16	0,690988	***Tai256c***	11,43	2,95596
Sko72	9,28	0,764943	***Tho150***	19,42	1,97528

Tabla 5.1.9: AM-(10,1.0)

Algoritmo Memético 10-0.1mej

Caso	*Desv*	*Tiempo*	*Caso*	*Desv*	*Tiempo*
Chr20b	36,29	2,37E-01	***Sko100a***	6,30	2,30375
Chr22a	13,39	2,58E-01	***Sko100b***	6,95	2,32177
Els19	4,56	2,09E-01	***Sko100c***	6,90	2,4049
Esc32b	35,71	4,01E-01	***Sko100d***	6,64	2,30806
Kra30b	8,19	3,62E-01	***Sko100e***	6,78	2,29649
Lipa90b	23,70	1,78532	***Tai30b***	4,75	3,66E-01
Nug25	4,65	2,84E-01	***Tai50b***	7,12	0,764906
Sko56	6,74	0,850635	***Tai60a***	5,85	0,988503
Sko64	6,30	1,02998	***Tai256c***	5,72	10,619
Sko72	5,73	1,29143	***Tho150***	15,70	4,34574

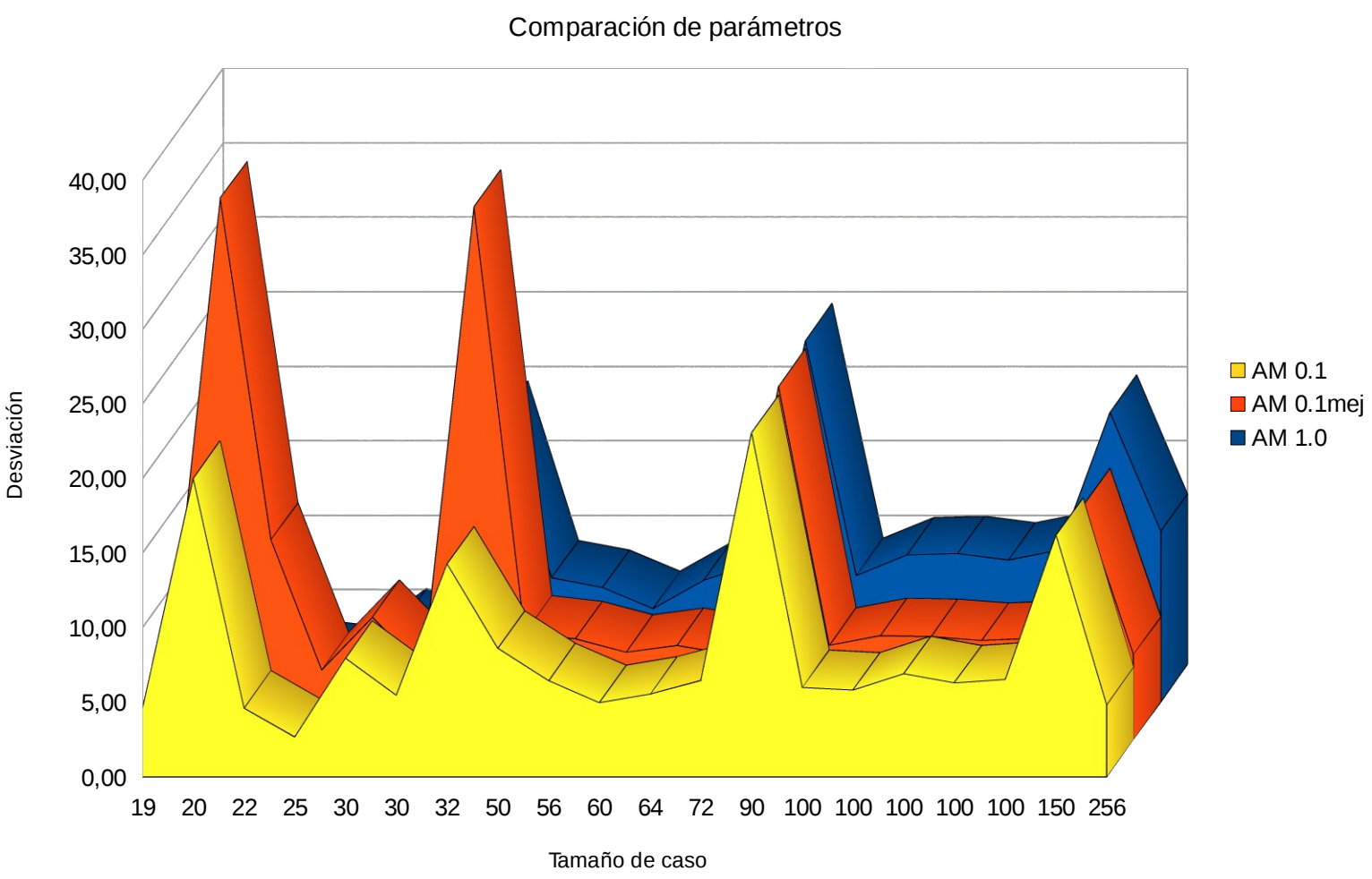
Tabla 5.1.10: AM-(10,0.1mej)

Algoritmo Memético 10-0.1

Caso	*Desv*	*Tiempo*	*Caso*	*Desv*	*Tiempo*
Chr20b	20,02	2,19E-01	***Sko100a***	5,99	4,15752
Chr22a	4,61	2,47E-01	***Sko100b***	5,82	3,9541
Els19	4,21	2,11E-01	***Sko100c***	6,91	3,93121
Esc32b	14,29	3,98E-01	***Sko100d***	6,30	3,94928
Kra30b	7,95	3,64E-01	***Sko100e***	6,53	3,92655
Lipa90b	23,11	1,79032	***Tai30b***	5,47	1,33E+00

Nug25	2,67	2,85E-01	***Tai50b***	8,62	1,98532
Sko56	6,45	0,856499	***Tai60a***	4,97	2,18544
Sko64	5,54	1,03104	***Tai256c***	4,83	12,3068
Sko72	6,46	1,29978	***Tho150***	16,22	6,55225

Tabla 5.1.11: AM-(10,0.1)

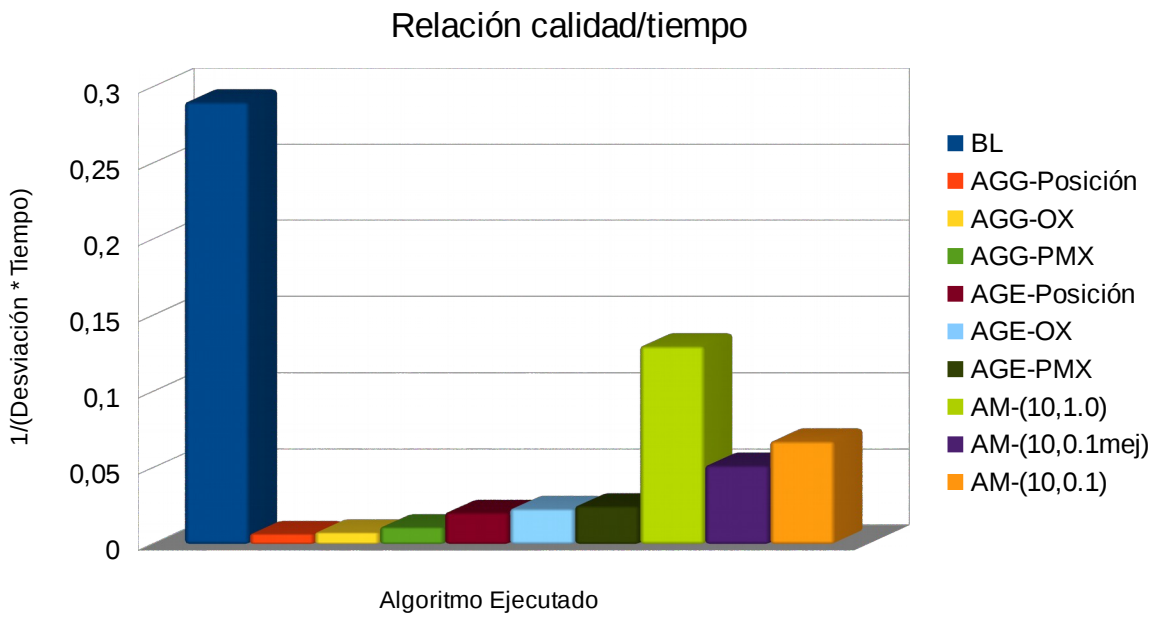


Los algoritmos meméticos (para todos se ha usado el operador PMX) intentan equilibrar la explotación de la búsqueda local con la exploración de los algoritmos genéticos con la finalidad de suplir sus respectivos problemas. Los tres algoritmos meméticos que hemos analizado son de baja intensidad. De las 3 variantes podemos observar que claramente una de ellas es mejor que el resto, la del 10 % aleatorio. Aplicar a toda la población la búsqueda local (AM 1.0) implica centrarnos excesivamente en las características explotativas del algoritmo y podemos perder muchas evaluaciones intentando analizar soluciones que ya han alcanzado un óptimo local. El del 10 % mejor va a sufrir de este mismo problema con ese 10 % de soluciones. Además, la gran mejora con respecto al resto de la población del 10 % mejor, hará más difícil la supervivencia de los individuos más malos de la población y probablemente nos lleve a una convergencia prematura. Por

tanto, parece razonable utilizar un mecanismo de selección aleatoria para la búsqueda local incorporada.

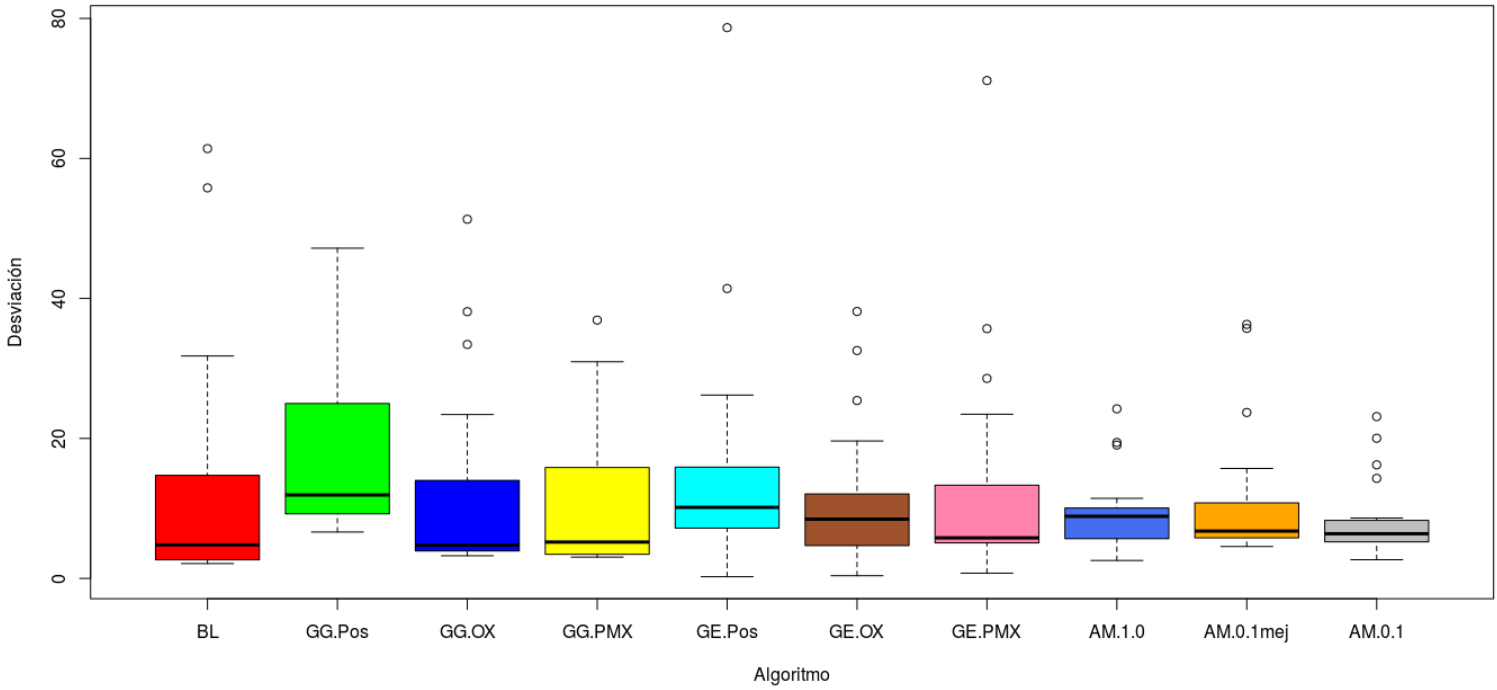
	Desviación	Tiempo
Greedy	62,64	0,00
BL	12,76	0,27
AGG-Posición	17,61	8,22
AGG-OX	12,00	10,57
AGG-PMX	10,98	8,03
AGE-Posición	15,51	3,09
AGE-OX	11,06	3,88
AGE-PMX	12,65	3,14
AM-(10,1.0)	9,55	0,80
AM-(10,0.1mej)	10,90	1,77
AM-(10,0.1)	8,37	1,77

Tabla 5.1.12: Comparativa de resultados (semilla 18)



Como cabía esperar exceptuando al Greedy para la que esta relación es todavía más exagerada, la búsqueda local es la siguiente en relación calidad/tiempo seguida por los meméticos donde el memético aplicado a toda la población es el más rápido seguido del del 10 % aleatorio.

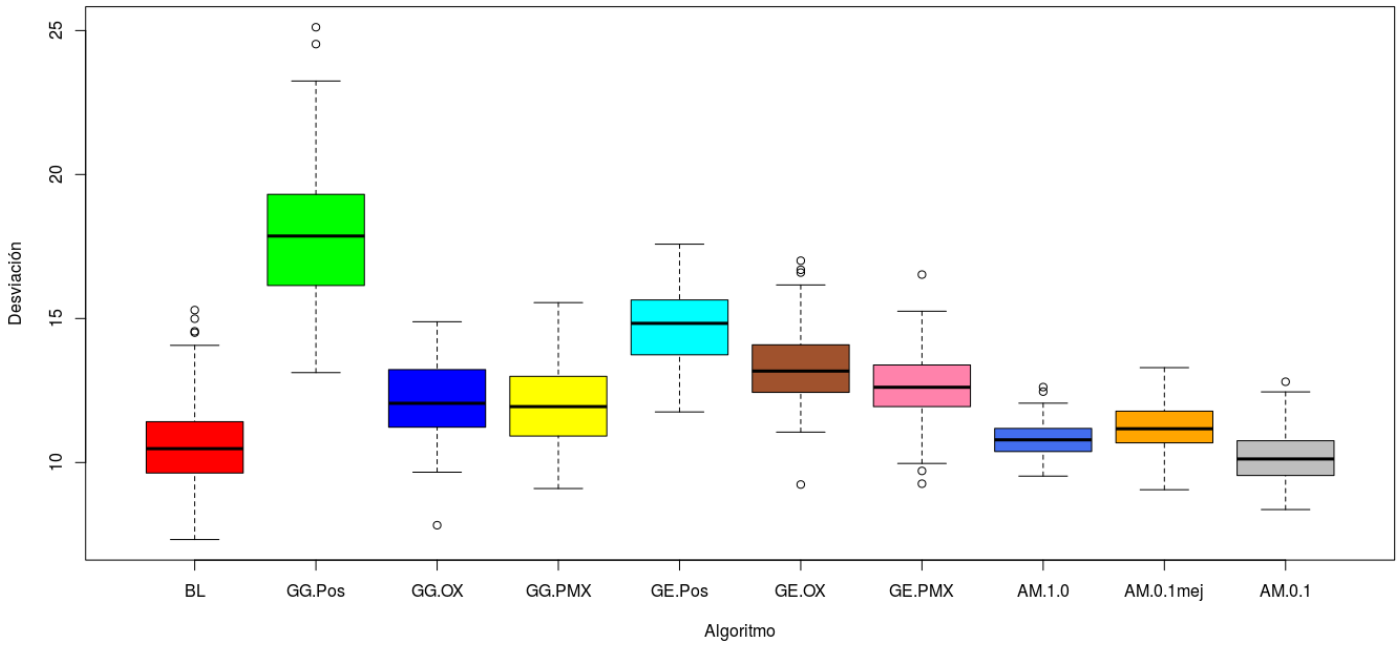
Para tener una visión final (aparte de la tabla anterior) cómo se comportan estos algoritmos para la semilla 18 veamos el siguiente boxplot basado en las desviaciones para todos los casos.



Podemos observar que la mediana de la Búsqueda Local es la más baja, no obstante la varianza entre los cuartiles 1 y 3, y en general es bastante amplia. Los algoritmos genéticos que usan basado en posición quedan son acotados por la parte inferior por sus equivalentes OX y PMX, por lo que podemos obviarlos como buenos algoritmos. Entre el PMX y el OX cuesta sacar una conclusión definitiva de cuál es mejor. Para el caso dado el OX se comporta ligeramente mejor que el PMX, pero no hay suficiente diferencia como para determinar que uno es mejor que otro. De entre los meméticos la versión con el 10 % aleatorio es claramente la ganadora ya que es la que menos mediana nos ofrece y tienes lo cuartiles mejor acotados exceptuando los 4 casos excepcionales representados con puntos. En relación calidad/tiempo la BL y el algoritmo memético son las dos opciones que mejor satisfacen el problema. La posible exploración de otras posibilidades en el memético cambiando los parámetros y analizando los problemas de convergencia del mismo puedo llevarnos a versiones que nos permitan alcanzar soluciones de mayor calidad.

5.2 Resultados de la aplicación de los algoritmos a los casos con 100 semillas

No obstante, limitarnos a una única semilla crea una dependencia directa entre la calidad de nuestro resultados y la calidad de la población/soluciones iniciales utilizadas para los algoritmos. Para conseguir una muestra más representativa de la muestra vamos a calcular los media de las desviaciones para las 100 primeras semillas (exceptuando el algoritmo Greedy) y ver si los datos que hemos obtenido en la muestra de la semilla 18 son representativos del caso general con otro boxplot.



Podemos observar que al aumentar el número de casos el AGG con cruce basado en posición se queda muy por detrás del resto. El resto de AGG nos da una gran similitud entre OX y PMX, con el OX con un caso excepcional en el que se comporta muy bien y el pmx ofreciendo por norma general una ligera variación más alta (tanto a mayor coste más alto como más bajo). No obstante, el rango entre los cuartiles primero y tercero queda ligeramente más bajo para PMX. En el caso del estacionario, PMX es el ganador sin duda. Los valores mínimos obtenidos han sido en casos de la búsqueda local, que tienen una gran varianza. Las diferencia entre los meméticos son menos exageradas que en el caso de la semilla 18. No obstante, sin lugar a dudas, el caso memético del 1 %, mantiene tanto la mediana como sus cuartiles del 1 al 3 lo más bajos, ofrece la posibilidad de ir a mínimos mejores pero su variabilidad ha aumentado considerablemente.