METAHEURÍSTICAS-2015/2016

****

PRÁCTICA 3.

Algoritmos Genéticos para el Problema de la Asignación Cuadrática

****

ALGORITMOS:

- Modelo generacional (AGG)-posición
- Modelo generacional (AGG)-OX
- Modelo estacionario (AGE)-posición.
- Modelo estacionario (AGE)-OX.

Autora: Cristina Zuheros Montes-50616450

- Correo: zuhe18@gmail.com
- Github: https://github.com/cristinazuhe

Horario prácticas: Viernes 17:30-19:30

Fecha: 12 Mayo 2016

ÍNDICE:

1. Descripción del problema.

EL problema de la asignación cuadrática (QAP) que vamos a analizar, trata de conseguir una asignación óptima de n unidades en n localizaciones, conociendo la distancia entre las unidades y el flujo entre las localizaciones. Tanto las distancias entre las unidades como el flujo entre las localizaciones vendrán dados por matrices, que denotaremos como "d" y "f" respectivamente. Así pues, f_{ij} detona el flujo existente entre la unidad "i" y la unidad "j". Análogo para la matriz de distancias.

Una vez tengamos las asociaciones de unidades-localidades, vamos a obtener su coste. El objetivo es conseguir el mínimo coste posible, que nos dará la solución más cercana a la óptima. La función de coste viene determinada por:

coste solucion: $$\min_{\pi \in \Pi_N} \left(\sum_{i=1}^{n} \sum_{j=1}^{n} f_{ij} \cdot d_{\pi(i)\pi(j)} \right)$$

donde π es una solución al problema que consiste en una permutación que representa la asignación de la unidad "i" a la localización "π (i)"

<u>Ejemplo:</u>
Para hacernos una idea más clara, supongamos que tenemos las unidades 0,1,2 y las localidades 0,1,2.
Si la solución encontrada es la permutación {2 0 1} quiere decir que a la unidad 0 le corresponde la localización 2, a la unidad 1 le corresponde la localización 0 y a la unidad 2, la localización 1.

2. Descripción de aplicación de los Algoritmos.

●Greedy.

Mediante este algoritmo asignamos las unidades que tengan mayor flujo a localizaciones que se encuentren céntricas.
La idea es obtener vectores de flujo y de distancias como marginalizaciones de las correspondientes matrices. A partir de dichos vectores, iremos asociando la unidad libre con mayor flujo a la localización libre con menor distancia.

●Funciones auxiliares.

Vamos a ver varias funciones auxiliares que aplicaremos en los algoritmos (mat1 representa flujos y mat2 representa distancias).

- Pseudocódigo de la función coste.

Vemos el pseudocódigo de la función coste vista en el primer apartado que llamábamos descripción del problema:

Coste_Solución:

```
for i=0...n-1
        for j=0...n-1
                si i!=j
                coste += mat1(i,j)*mat2(sol(i),sol(j))
```

- Pseudocódigo de generación de soluciones aleatorias.

También tendremos una función auxiliar para generar una solución aleatoria inicial para los algoritmos. Para ello haremos uso de una misma semilla para todos los algoritmos, que nos irá generando números aleatorios.

Permutar_Solucion_n:

```
for i=0...n-1
        gen1 <- aleatorio entre [0,n-1]
        gen2 <- aleatorio entre [0,n-1] != gen1
        permuto en la solución los indices gen1 y gen2
```

- Nota: Volvemos a insistir en la representación de soluciones que vamos a emplear:

π hace referencia a una solución al problema que consiste en una permutación que representa la asignación de la unidad "i" a la localización "π (i)" (Ver ejemplo comentado en el primer apartado donde explicamos la descripción del problema).

Mecanismo de selección.

Para realizar la selección de la población de padres, realizamos torneo binario. Consiste en tomar aleatoriamente dos individuos de la población original y seleccionar el mejor de ellos. Este mejor individuo será el que añadiremos a la población de padres.

El mecanismo es el mismo para AGG y para AGE, con la diferencia de que en AGG aplicamos tantos torneos como individuos hay en la población, mientras que en AGE aplicamos sólamente dos veces el torneo binario con el fin de conseguir los dos padres que serán posteriormente cruzados y mutados. (M representará tamaño población)
Podemos crear una función auxiliar común para ambos algoritmos.

Seleccion_padres(poblacion_actual, poblacion_padres, numero_padres)

```
for i=0...numero_padres_a_generar-1
        genero aux1, aux2 aleatorios ∈ [0, M-1]
        si aux1 < aux2
                guardo la solución aux1 de poblacion_actual en población_padres
        sino
                guardo  la solución aux2 de poblacion_actual en población_padre
```

Nota: en población_actual tenemos la población de soluciones almacenadas según su coste de menor a mayor.

Operador de cruce: posición.

Partimos de dos padres y vamos a generar dos hijos con el mismo procedimiento:
Paso 1: Almacenamos en los hijos aquellas posiciones que contengan el mismo valor en ambos padres.
Paso 2: Las asignaciones restantes se seleccionan en un orden aleatorio para completar los hijos.
De este modo nos vamos quedando con las asignaciones más prometedoras.

Lo más sencillo es ver un ejemplo explicativo, sean los padres siguientes:
Padre1 = (1 2 3 4 5 6 7)
Padre2 = (1 4 2 6 5 3 7)
Realizando el paso 1 obtenemos
Hijo1 = (1 -1 -1 -1 5 -1 7) e Hijo 2 = (1 -1 -1 -1 5 -1 7)
Los elementos restantes son: {2,3,4,6}.
Realizamos el paso 2:
Supongamos que obtenemos el orden aleatorio {2,4,3,6} para el hijo1, quedaría: Hijo1 = (1 2 4 3 5 6 7)
Supongamos que obtenemos el orden aleatorio {6,4,2,3} para el hijo2, quedaria: Hijo2 = (1 6 4 2 5 3 7)

Veamos el pseudocógido:

Cruze_Posicion(poblacion_padres, poblacion_intermedia)
Paso1:
for j=0...n-1
    si (elemento j del primer padre = elemento j del segundo padre)
        añado a poblacion_intermedia 2hijos con los elementos j's de los padres
        almaceno los elementos j's en solcomunes[n]
    sino
        almaceno los elementos j's en solsueltas[n]
        almaceno las posiciones de dichos j's en poslibres[n]

Paso2:

Permutar_Solucion_n(solsueltas)

añado la permutación en las posiciones libres de hijo1 en orden
obtengo coste de primer hijo
añado coste de primer hijo al final del primer hijo (*)

Permutar_Solucion_n(solsueltas)

añado la permutación en las posiciones libres de hijo2 en orden
obtengo coste de segundo hijo
añado coste de segundo hijo al final del segundo hijo (*)

(*) Esto lo hacemos para poder recuperar los costes de las soluciones fácilmente en los algoritmos genéticos. Además así será más cómodo ordenar luego las poblaciones.

● Operador de cruce: OX.

Partimos de dos padres y vamos a generar dos hijos. El primer hijo se genera con el siguiente procedimiento:
Paso 1: Elegimos una subcadena del primer padre y la copiamos en el primer hijo.
Paso 2: Nos quedamos con los elementos que no hemos añadido al hijo y los ordenamos según el orden del segundo padre.
Paso 3: Finalmente los añadimos al primer hijo partiendo de la última posición que dejamos libre en el paso 1.

EL segundo hijo se genera intercambiando los papeles de los padres en el proceso anterior.

Lo más sencillo es ver un ejemplo explicativo, sean los padres siguientes:
Padre1 = (1 2 3 4 5 6 7)
Padre2 = (1 4 2 6 5 7 3)

Sean los aleatorios 3 y 5 que nos marcan la subcadena fija en el primer paso, obtenemos:
Generando hijo1:
Hijo1 = (-1 -1 -1 4 5 6 -1) y nos quedarían libres {1,2,3,7}.
Realizando el paso2 nos quedarían libres {1, 2, 7, 3}
Realizo el paso 3, queda Hijo1 = (1 2 7 4 5 6 3)

Generando hijo2:
Hijo2 = (-1 -1 -1 6 5 7 -1) y nos quedarían libres {1, 4,2, 3}.
Realizando el paso 2 nos quedarían libres {1,2, 3, 4}
Realizo el paso 3, queda Hijo2 = (1 2 3 6 5 7 4)

Cruze_OX(pob_padres, pob_intermedia)
Paso1:
genero auxi1 y auxi2 aleatorios de 0 a n-1 tal que auxi1<auxi2
copio en los hijos los intervalos fijos de los padres

Paso2:
los elementos restantes los ordeno conforme al padre contrario

Paso3:
añado los elementos restantes en las posiciones libres de los hijos a partir de la posicion (aux2+1)%n
obtengo coste de los hijos
añado los costes de los hijos al final de los hijos (*)

(*) Esto lo hacemos para poder recuperar los costes de las soluciones fácilmente en los algoritmos genéticos. Además así será más cómodo ordenar luego las poblaciones.

●Mutación.

Es una adaptación de la función Permutar_Solucion_n que vimos en "Funciones auxiliares". La diferencia es que antes procedíamos sobre una solución, y ahora tenemos que hacer la mutación sobre una población. Vamos a tener que considerar alguna de las soluciones de la población y aplicarle a ella la permutación.

Además ahora vamos a tener un número de mutaciones a realizar (no vamos a mutar las n veces como hacíamos antes).

Veamos cómo quedaría el pseudocódigo:

Mutar_Poblacion(poblacion, num_mutaciones,n_soluciones_poblacion) :
for i=0...num_mutaciones-1
alea1 <- aleatorio entre [0, n_sol_poblacion-1]
gen1 <- aleatorio entre [0,n-1]
gen2 <- aleatorio entre [0,n-1] != gen1
permuto en la solución alea1 de la población los indices gen1 y gen2
actualizo el coste de la solucion alea1 conforme a la permutacion

●Algoritmos genéticos.

Los algoritmos genéticos siguen el siguiente esquema, que se ejecuta un número determinado de veces:

De una población actual de M individuos (en nuestro caso M=50) hacemos una selección. Esa selección nos dará la población de padres, a los cuales se les realizará un cruce (en nuestro caso, de posición o OX) con una determinada probabilidad.

De este modo obtenemos una población intermedia a la cual se le hará una mutación con otra determinada probabilidad (en nuestro caso usamos 0,001). Este proceso nos genera la población de los hijos.

Estos hijos realizarán un reemplazamiento sobre la población actual dependiendo de unas determinadas características según el algoritmo empleado

●Modelo generacional (AGG).

Seguimos el modelo que hemos visto anteriormente, con las siguientes características:
Selección: se realizará un torneo binario de M candidatos en el que nos quedaremos con las mejores soluciones. Se selecciona una población de padres del mismo tamaño que la población genética. Iremos haciendo parejas de padres tipo 0-1, 2-3, 4-5...
Cruce: usaremos los dos tipos de cruces que explicamos anteriormente.
Reemplazamiento: la población de hijos sustituye automáticamente a la actual. Si la mejor solución de la generación anterior no sobrevive, sustituimos la peor solución de la nueva población.

●Modelo estacionario (AGE).

Seguimos el modelo que hemos visto anteriormente, con las siguientes características:
Selección: realizamos dos veces torneo binario para elegir a los dos padres que serán posteriormente recombinados.
Cruce: usaremos los dos tipos de cruces que explicamos anteriormente.
Reemplazamiento: los dos descendentes recombinados, sustituirán a las dos peores soluciones de la población mejor, en caso de que sean mejores.

3. Descripción en pseudocódigo de los Algoritmos.

●Modelo generacional (AGG).

Inicializo:

M <- 50 será el tamaño de la población actual
Genero M soluciones iniciales aleatorias y las almaceno en poblacion_actual junto con sus costes
Ordeno la población actual de menor a mayor coste
mejorpadre <- poblacion_actual[0]

Bucle principal:

mientras (num_evaluaciones < 25000)
    Seleccion_padres(poblacion_actual, poblacion_padres, M)
    Ordeno la población de padres de menor a mayor coste

    *num_cruces <-0.7 *(M/2)*
    for i=0...M-1
        si i < num_cruces
            si tipo_operador=0 (con 0 indico cruce posicion)
                Cruze_Posicion(pob_padres, pob_intermedia)
                num_evaluciones+=2
            sino si tipo_operador=1 (con 1 indico cruce OX)
                Cruze_OX(pob_padres, pob_intermedia)
                num_evaluciones+=2
        sino
            añado los padres usados y las siguientes soluciones de mejor calidad a la poblacion_intermedia

    num_mutaciones <- 0.001*(M*n)
    Mutar_Poblacion(pob_intermedia, num_mutaciones, M)
    num_evaluaciones <- num_evaluaciones + num_mutaciones

    meto el mejor padre en caso de que no esté en la pob_intermedia
    sustituyo pob_actual por pob_intermedia ordenada por coste

Calculo el coste de la solución.

●Modelo estacionario (AGE).

Inicializo:

M <- 50 será el tamaño de la población actual
Genero M soluciones iniciales aleatorias y las almaceno en poblacion_actual junto con sus costes
Ordeno la población actual de menor a mayor coste

Bucle principal:

mientras (num_evaluaciones < 25000)

Seleccion_padres(poblacion_actual, poblacion_padres, 2)

si tipo_operador=0 (con 0 indico cruce posicion)

Cruze_Posicion(pob_padres, pob_intermedia)

num_evaluciones+=2

sino si tipo_operador=1 (con 1 indico cruce OX)

Cruze_OX(pob_padres, pob_intermedia)

num_evaluciones+=2

sino

añado los padres usados y las siguientes soluciones de mejor calidad a la poblacion_intermedia

num_mutaciones <- 0.001*(M*n)
Mutar_Poblacion(pob_intermedia, num_mutaciones, 2)
num_evaluaciones <- num_evaluaciones + num_mutaciones

busco los dos peores individuos y los comparo con los dos hijos
me quedo con los dos mejores de estas cuatro soluciones
ordeno la población actual de menor a mayor coste

Calculo el coste de la solución.

4. Experimentos y análisis de resultados

Algoritmo AGG-posicion			
Caso	Coste obtenido	Desv	Tiempo
Chr20a	3144	43,43	0,08817
Chr20c	17610	24,52	0,08332
Chr22b	7324	18,24	0,09356
Chr25a	5852	54,16	0,11276
Esc32a	158	21,54	0,17927
Esc64a	124	6,90	0,58713
Esc128	78	21,88	2,11885
Kra32	96330	8,60	0,18072
Lipa90a	364000	0,93	1,07574
Sko42	16312	3,16	0,27139
Sko49	24282	3,83	0,35816
Sko81	94264	3,59	0,88193
Sko90	120490	4,29	1,07764
Sko100f	154022	9,48	1,30425
Tai64c	2732114	47,21	0,60483
Tai80a	15793896	17,00	0,90090
Tai100a	24087528	52,02	1,35755
Tai150b	657434738	48,81	2,86930
Tai256c	53947022	23,03	8,33640
Tho150	9795822	28,54	3,03033

Usando AGG con operador de cruce de posición:

Vemos que los resultados que obtenemos no son buenos en general, aunque encontramos un intervalo (desde esc64a hasta sko100f) con resultados aceptables. Cabe destacar que para lipa90a encontramos una solución muy cercana a la óptima.

Por otra parte, vemos que los tiempos son bastante reducidos en la mayoría de los casos. Destacando negativamente tai256c (es comprensible por la cantidad de unidades-localidades de las que dispone).

Algoritmo AGG-OX			
Caso	Coste obtenido	Desv	Tiempo
Chr20a	4732	115,88	0,10102
Chr20c	36484	157,98	0,09725
Chr22b	8136	31,35	0,10773
Chr25a	9924	161,43	0,13087
Esc32a	274	110,77	0,20558
Esc64a	158	36,21	0,63752
Esc128	182	184,38	2,28232
Kra32	111600	25,82	0,20569
Lipa90a	366402	1,60	1,18062
Sko42	18142	14,74	0,30610
Sko49	26614	13,80	0,39973
Sko81	102886	13,06	0,97972
Sko90	129622	12,19	1,18545
Sko100f	166692	18,48	1,42711
Tai64c	3037342	63,66	0,63598
Tai80a	15554544	15,23	0,95505
Tai100a	24269932	53,17	1,43014
Tai150b	654868636	48,23	3,04459
Tai256c	51871254	18,29	8,78186
Tho150	9898294	29,89	3,15104

Usando AGG con operador de cruce OX:

Usando dicho algoritmo obtenemos resultados bastante pésimos. A penas nos encontramos casos en los que la solución sea cercana a la óptima. Sin embargo, sí que nos encontramos muchos casos en los que la solución dista mucho de ser buena, como pueden ser esc128 y chr20c.

Los tiempos son muy buenos con alguna que otra excepción (como tahi256c), pero al generar soluciones tan malas no nos merece la pena. Este algoritmo no nos proporciona unos buenos resultados, de modo que tener un buen tiempo no aporta demasiado.

Algoritmo AGE-posicion			
Caso	Coste obtenido	Desv	Tiempo
Chr20a	3840	75,18	0,11548
Chr20c	34068	140,90	0,11624
Chr22b	7118	14,92	0,13044
Chr25a	5654	48,95	0,15052
Esc32a	164	26,15	0,22626
Esc64a	118	1,72	0,59713
Esc128	138	115,63	2,08283
Kra32	95940	8,16	0,22510
Lipa90a	365799	1,43	1,08442
Sko42	16562	4,74	0,30079
Sko49	24478	4,67	0,38986
Sko81	100096	10,00	0,89261
Sko90	128662	11,36	1,09302
Sko100f	164448	16,89	1,30451
Tai64c	3080600	65,99	0,60334
Tai80a	15659120	16,00	0,88512
Tai100a	24082030	51,99	1,30520
Tai150b	649495292	47,02	2,93375
Tai256c	52388720	19,47	8,16724
Tho150	9873390	29,56	3,01327

Usando AGE con operador de cruce de posición:

Vemos que en general no obtenemos soluciones cercanas a la óptima aunque hay algunos casos (desde kra32 a sko100f) que son aceptables.

En cuanto al tiempo que tarda en ejecutarse vemos que es bastante reducido, incrementándose en los casos en los que el número de unidades-localizaciones es mayor.

Algoritmo AGE-OX			
Caso	Coste obtenido	Desv	Tiempo
Chr20a	2898	32,21	0,12252
Chr20c	21554	52,41	0,12281
Chr22b	7360	18,82	0,13878
Chr25a	5594	47,37	0,16129
Esc32a	196	50,77	0,23997
Esc64a	140	20,69	0,61588
Esc128	168	162,50	2,12656
Kra32	99890	12,62	0,23821
Lipa90a	366214	1,55	1,12009
Sko42	17424	10,19	0,31543
Sko49	25868	10,61	0,40864
Sko81	102008	12,10	0,91765
Sko90	129088	11,73	1,12065
Sko100f	165724	17,79	1,33776
Tai64c	3009672	62,17	0,61514
Tai80a	15529818	15,04	0,89881
Tai100a	24263854	53,14	1,32906
Tai150b	652467671	47,69	3,01640
Tai256c	53445462	21,88	8,27984
Tho150	9895054	29,85	3,06398

Usando AGE con operador de cruce OX:

En general vemos que con este método no obtenemos soluciones cercanas a las óptimas, incluso hay casos en los que las desviaciones son excesivamente grandes (ver esc128). Hay una pequeña zona comprendida entre kra32 y sko100f que obtiene resultados mejores, aunque siguen sin ser buenos.

En cuanto al tiempo, vemos que es bastante pequeño, superando un par de segundos en pocos casos.

Veamos la comparación entre los algoritmos:

Algoritmo	Desv	Tiempo
Greedy	104,55	0
AGG-pos	22,05846	1,27561
AGG-OX	56,30804	1,36227
AGE-pos	35,53648	1,28086
AGE-OX	34,55624	1,30947

Cabe comentar antes de nada, que para todos estos resultados que mostramos hemos usado la semilla 5500055.

Comparando algoritmos:
Vemos que con el Greedy obtenemos una desviación realmente alta, luego no nos aporta ningún beneficio utilizarlo. Sin embargo, nos va a permitir hacer comparaciones con los demás algoritmos.

Usando algoritmos genéticos generacionales elitistas con operador de posición obtenemos una desviación media aceptable en comparación con los demás algoritmos. Ya hemos visto antes, que no dicho algoritmo no nos proporciona buenos resultados, pero sí que nos proporciona mejores resultados que los demás algoritmos usando dicha semilla y dichos datos. Es más, el tiempo también es el más pequeño en media, lo que nos lleva a posicionarlo como el algoritmo que nos proporciona mejores resultados en relación desviación-tiempo.

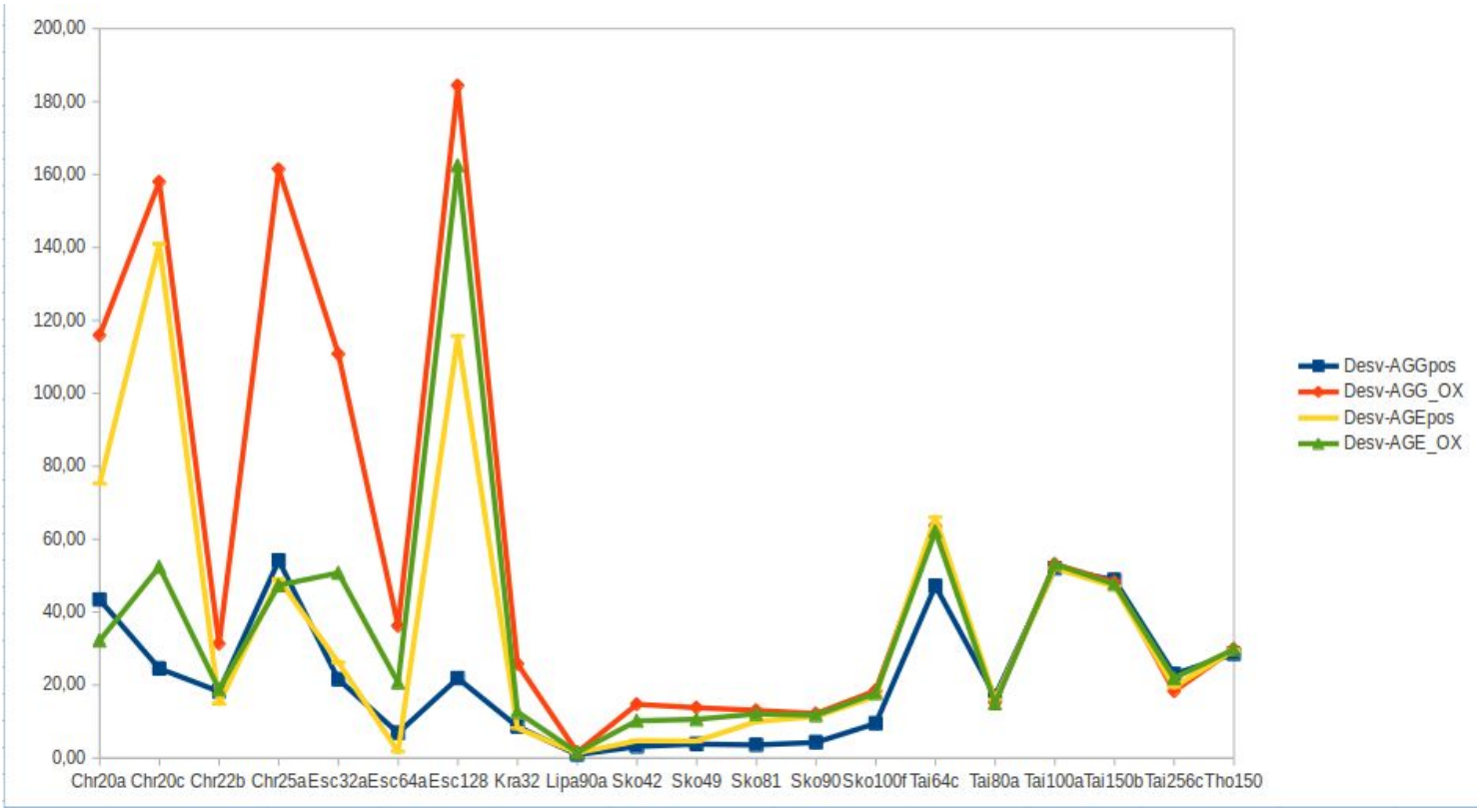
Al usar algoritmos genéticos generacionales elitistas con operador OX, vemos que se incrementa notablemente la desviación media. Obtenemos peores resultados. Estos nos lleva a darnos cuenta de que la elección del operador de cruce es muy importante para obtener

mejores o peores resultados. Probablemente, si hubiésemos usando otro operador como PMX, podríamos haber obtenido mejores desviaciones.
A continuación tenemos los algoritmos genéticos estacionarios con operador de posición. Vemos que la desviación en media es bastante elevada, empeorando con respecto a AGG con el mismo operador de cruce. El tiempo medio es muy bueno y prácticamente igual que con AGG-posicional.

Finalmente nos encontramos con los algoritmos genéticos estacionarios usando el operador OX. En este caso, vemos que mejora levemente la desviación media que obtenemos con AGE-posicional, aunque sigue siendo peor que AGG-OX.

Estas diferencias entre un mismo algoritmo cambiando únicamente el operador, no nos pueden garantizar que un operador de cruce sea mejor que otro siempre. Aunque para nuestro caso de estudio, parece ser que el operador de posición da mejores resultados.

A continuación vamos a ver un gráfico donde comparamos las desviaciones particulares y medias para los distintos casos de estudio utilizando AGG-pos, AGG-OX, AGE-pos y AGE-OX:



Aquí se puede ver claramente que AGG-OX es el algoritmo que presenta peores resultados, obteniendo la desviación más elevado para prácticamente todos los casos.

Además vemos que, por el contrario, AGG-pos presenta los mejores resultados, obteniendo las menor desviación para prácticamente todos los casos.

Es curioso que estemos con el mismo algoritmo y únicamente cambiemos el operador de cruce. Esto revela que el operador de cruce tiene mucho peso en estos algoritmo pues es lo que nos va permitir ir construyendo unas poblaciones intermedias de buena calidad.

Valorando casos particulares:

Viendo las tablas anteriores, por un lado observamos que el caso esc128 es uno de los más problemáticos en el sentido de que con cualquiera de los algoritmos realizados (salvo con AGG-pos) obtenemos soluciones muy malas.
Otro caso bastante problemático es chr20c, pues tanto para AGG-OX como para AGE-pos, supone un gran incremento de la desviación media.

Por otro lado, destaca el caso lipa90a con el que obtenemos un resultado bastante cercano al óptimo con cualquiera de los algoritmos.

Número de veces que es mejor…:

Hemos recopilado para cada caso cuál es el mejor algoritmo obteniendo dichos resultados. Nuevo de veces que es mejor:

- Greedy: 3
- AGG-pos: 10
- AGG-OX: 1
- AGE-pos: 3
- AGE-OX: 3

Como ya comentábamos antes, usando AGG-pos obtenemos la mejor solución para la mayoría de los casos. En concreto, obtenemos la mejor solución para 10 casos.

Por otra parte tenemos AGG-OX, AGE-pox y AGE-OX que, a pesar de tener malas desviaciones medias, hay casos en los que obtiene la mejor solución con respecto a los otros algoritmos. Esto se debe a que nos encontramos fuertes picos de subida, e incluso bajada, con dichos algoritmos.

Finalmente, llama la atención el hecho de que usando Greedy tengamos 3 casos de estudio en los que encuentra la mejor solución. Esto se debe a que los algoritmos genéticos no nos han aportando buenas soluciones, de modo que con cualquier otro algoritmo, por simple que sea, podremos llegar a encontrar casos en los que obtengamos mejores resultados que con los genéticos.

Usando distintas semillas:
Finalmente en nuestro análisis hemos probado a ejecutar los algoritmos con distintas semillas.
Veamos las desviaciones de, por ejemplo:

- Con semilla 6060:
  AGG-pos: 25.37
  AGG-OX: 53.42
  AGE-pos: 33.30
  AGE-OX: 37.04

- Con semilla 1111:
  AGG-pos: 25.17
  AGG-OX: 53.03
  AGE-pos: 30.31
  AGE-OX: 32.87

- Con semilla 9090:
  AGG-pos: 29.68
  AGG-OX: 48.94
  AGE-pos: 32.90
  AGE-OX: 33.66

Vemos que, al igual que ocurría con con la semilla 550055, usando el algoritmo AGG con operador de cruce de posición obtenemos la mejor desviación en media.

Además, para todas las semillas, al usar el operador OX en lugar del de posición, vemos que la desviación se incrementa notablemente. Esto nos lleva a pensar que posiblemente el operador OX no sea el mejor para nuestro problema.

En general, AGG con operador de posición da los mejores resultados, mientras que AGG con operador OX da los peores resultados.

5. Procedimiento para el desarrollo de la práctica

Para la realización de la práctica he hecho uso de los pdfs seminario03-problemas-poblaciones, guión de prácticas y tema 5 de teoría. He seguido las ideas y pseudocódigos que vemos en dichos pdfs, teniendo en cuenta las condiciones que se imponen en el guión de prácticas.