Metaheurísticas

Práctica 1.a: Técnicas de Búsqueda basadas en Poblaciones para el Problema de la Asignación Cuadrática

Curso 2016-2017. Grado en Ingeniería Informática

Alumno:
Sergio Carrasco Márquez

Índice

1. Descripción del problema

El problema consiste en la asignación de unidades que tiene un flujo asociado entre ellas a localizaciones con un valor de distancia que las separa, de forma que las unidades con más flujo entre ellas estén separadas por distancias más cortas. Expresado de forma matemática el problema consiste en reducir el resultado de la función $\sum_{i=1}^{n}\sum_{j=1}^{n}(f_{ij}\ d_{\pi(i)\pi(j)})$. De tal forma que f_{ij} Simboliza el flujo entre las unidades i y j y $d_{\pi(i)\pi(j)}$ la distancia entre la localización a la que se asignan dichas unidades.

2. Descripción de la aplicación de los algoritmos

2.1Función de evaluación

La solución al problema se representa como un vector en el que cada casilla indica la unidad a la que se hace referencia y el contenido de dicha casilla es la localización de dicha unidad. Para evaluar una solución se llama a una función que calcula el valor de la función explicada en el apartado anterior, $\sum_{i=1}^{n}\sum_{j=1}^{n}(f_{ij}\ d_{\pi(i)\pi(j)})$, dependiendo de los valores asignados a cada posición del vector. La función se implementa de la siguiente manera:

```
Evaluación = 0
para i=0 hasta tamaño_del_vector
        para j=0 hasta tamaño_del_vector
                //v es el vector que almacena la solución
                evaluación += FlujoEntre(i,j)*Distancia(v[i],v[j])
return evaluación
```

2.2 Generación de vecinos y mutación

Para generar un vecino o una mutación se intercambian dos posiciones del vector y se debe calcular la evaluación de nuevo, pero en este caso se factoriza dicho cálculo.

```
Swap = v[i]
V[i] = v[j]
V[j] = v[i]
```

En el caso de la mutación los índices *i* y *j* se generan de forma aleatoria, en el caso de la búsqueda local la generación de un vecino escoge varios índices de forma aleatoria, sin repetir la pareja de valores asignados a *i* y a *j*, hasta encontrar un vector que mejore al original.

En el caso de tener que evaluar un vector que es resultado de la permutación de dos elementos se puede factorizar la función de evaluación para que sea computacionalmente menos costosa. Para realizar dicha factorización se debe conocer que dos elementos han permutado y el valor heurístico del vector antes de realizar dicha permutación

```
//Primero se invierte el cambio realizado en el vector
Swap = v[p1]
V[p1] = v[p2]
 v[p2] = swap

incremento = 0
para i = 0 hasta tamaño_del_vector
        si(i != p1 and i != p2)
                incremento += flujo(p1,i)*(distancia(v[p1],i) – distancia(v[p2],v[i] )) +
                flujo(p1,i)*( distancia( v[p2],v[i] ) – distancia( v[p1],v[i] ) ) +
                flujo(i,p2)*( distancia( v[i],v[p1] )-distancia(v[i],v[p2]) )+
                flujo(i,p1)*( distancia(v[i],v[p2])-distancia(v[i],v[p1]) )

//Deshacer el cambio en el vector
 Swap = v[p1]
V[p1] = v[p2]
 v[p2] = swap

eval+=incremento
return eval
```

2.3 Generación de soluciones aleatorias

La generación de soluciones aleatorias se realiza de forma simple, pero cumpliendo la restricción del problema sobre las soluciones, que no se asigne la misma localización a dos unidades distintas. En nuestro caso esa restricción implica que el en vector solución no pueden aparecer dos valores repetidos. Para generar vectores que cumplan esta restricción se crea un vector ordenado con todos los valores posibles, es decir desde 0 hasta el tamaño del vector y luego se baraja.

```
Para i = 0 hasta tamaño_del_vector
        V[i] = i
//Barajar el vector
Para i = 0 hasta tamaño_del_vector
        position = rand(i,tamaño_del_vector-1)//aleatorio entre i el el tamaño del vector -1
        swap = v[i]
        v[i] = v[posicion]
        v[posicion] = swap
```

2.4 Selección en algoritmos genéticos

La selección en los distintos algoritmos genéticos se lleva a cabo mediante la realización de torneos binarios entre individuos aleatorios. El número de torneos binarios realizados en el modelo generacional es distinto al los que se realizan en el modelo estacionario.

```
//Modelo generacional
vector candidatos1 //vector que guarda los índices de la población a los que se les aplicara el
//torneo binario
para i = 0 hasta  numero_padres
        candidatos1[i] = randint(0,tamaño_poblacion-1)
vector candidatos2
para i = 0 hasta  numero_padres
        candidatos2[i] = randint(0,tamaño_poblacion-1)
//Ahora se enfrentan candidatos1[i] contra candidatos2[i] en un torneo binario
Vector padres //almacena los índices de los padres que serán cruzados
Para i = 0 hasta numero_padres
        Si (población[candidatos1[i]].evaluación < población[candidatos2[i]].evaluación)
                padres[i] = candidatos1[i]
        sino
                padres[i] = candidatos2[i]

//Modelo estacionario

candidato1 = randint(0,tamaño_población-1)
candidato2 = randint(0,tamaño_población-1)
candidato3 = randint(0,tamaño_población-1)
candidato4 = randint(0,tamaño_población-1)
si (población[candidato1].evaluacion < (población[candidato2].evaluacion)
        padre1 = candidato1
sino
        padre1 = candidato2
si (población[candidato3].evaluacion < (población[candidato4].evaluacion)
        padre2 = candidato3
sino
        padre2 = candidato4
```

2.5 Operadores de cruce

Se implementan dos operadores de cruce, el basado en posición y el OX

```
//Cruce basado en posición

vector used
vector son =  -1 //hijo formado por padre1 y padre2 inicializado a -1
para i = 0 hasta tamaño_del_vector
        p1 = padre1[i]
        p2 = padre1[i]
        si(p1 == p2)
                son[i] = p1
                used[p1] = true
```

```
        sino
                used[p1] = false
vector pos
pos = RandomVector()//Vector aleatorio sin repeticiones
posición_escritura = 0
para i = 0 hasta tamaño_del_vector
        si (used[pos1] == false)
                mientras(son[posición_escritura]!=-1)
                        posición_escritura++

//Operador OX
Inicio =posición_media -  n_elementos/2
Fin = posición_media + n_elementos/2
vector escogidos
para i = 0 hasta tamaño_vector
        si(i >= inicio and i <= fin)
                escogidos[padre1[i]] = true
        sino
                escogidos[padre1[i]] = false
último_no_usado = 0
para i = 0 hasta tamaño_vector
        si (i >= inicio and i <= fin)
                son[i] = parent[i]
        sino
                mientras(escogidos[padre2[último_no_usado]] == true)
                        último_no_usado ++
                son[i] = padre2[último_no_usado]
                último_no_usado ++
```

3 Pseudocódigo de los algoritmos

3.1 Búsqueda local

```
Procedure EncuentraMejorVecino
        vector orden = GeneraVectorAleatorio()
        para i = 0 hasta tamaño_vector y encontrado == false
                posi =orden[i]
                si (DLB[posi] == true)
                        para j = 0 hsata tamaño_vector == encontrado == false
                                posj =order[j]
                                si (i i= j)
                                        swap = v[posi]
                                        v[posi] = v[posj]
                                        v[posj] =swap
```

```
                                        ev = FactorizaciónVector(v,solución_actual,posi,posj)
                                        iteraciones++
                                        si ev <solución_actual
                                                encontrado = true
                                                DLB[posj] = true
                                                DLB[posi] = true
                                        sino
                                                swap = v[posi]
                                                v[posi] = v[posj]
        endfor                                  v[posj] =swap
        si(encontrado == false)
                DLB[i] = false
        Return encontrado
endfor

Procedure:BúsquedaLocal
vector DLB = true //Dont look bits inicializado a true
mientras(iteraciones <max_iters  and encontrado )
        encontrado = EncuentraMejorVecino(v,iteraciones,DLB)
return v
```

3.2 Algoritmos genéticos

3.2.1 Modelo Generacional

Para este modelo se usan vectores que contienen el vector solución y la evaluación heurística del mismo y vectores que contienen el índice y el valor del heurístico del vector en la posición indicada por el índice para hacer las operaciones de ordenado de forma más eficiente.

```
//Inicializa la población
para i = 0 hasta tamaño_población
        población[i].v = GeneraVectorAleatorio
        población[i].evaluacion = evaluar(población[i].v)
        índices[i].índice = i
        índices[i].evaluación = población[i].evaluación
endfor
mientras(iteraciones < máximo_iteraciones)
        padres = SeleccionaPadres(v) //De la forma explicada en el apartado 2.4
        n_hijos = n_padres*prob_hijos
        para i = 0 hasta n_hijos
                si(operador == posición)
                        hijos[i].v = posición(padres[i*2],padres[2*i+1])
                sino
                        hijos[i].v = ox(padres[i*2],padres[2*i+1])
                hijos[i].evaluación = Evalua(hijos[i].v)
```

```
        iteraciones++
        si(iteraciones == max_iteraciones)
                n_hijos = i
        índice_hijos[i].índice = i
        índice_hijos[i].evaluación = hijos[i].evaluación
endfor
n_mutaciones = n_hijos*tamaño_problema*prob_mutación
para i = 0 hasta n_mutaciones
        hijo = Randint(0,n_hijos-1)
        gen1 = Randint(0,tamaño_problema-1)
        gen2 = Randint(0,tamaño_problema-1)
        solución_actual = hijos[hijo].evaluación
        swap = hijos[hijo].v[gen1]
        hijos[hijo].v[gen1] = hijos[hijo].v[gen2]
        hijos[hijo].v[gen2] = swap
        hijos[hijo].evaluación = factorización(solución_actual,hijos[hijo].v,gen1,gen2)
        índice_hijos[hijo].evaluación = hijos[hijo].evaluación
endfor
ordenar(índice_hijos)
ordenar(índice_población)
posición_insertado = tamaño_población – n_hijos
si(posición_insertado > 0)
        para i = 0 hasta n_hijos or hasta tamaño_población
                índice_pob = índice_población[posición_insertado+i].posición
                índice_son = índice_hijo[i].posición
                población[índice_pob] = hijos[índice_son]
        endfor
sino
        si (población[i] != hijos[índice_hijos[i-1].posición])
                para i = 0 hasta tamaño_población
                        población[i] = hijos[índice_hijos[i-1].posición]
        sino
                para i = 1 hasta tamaño_población
                        población[índice_población[i].posición] =
                         hijos[índice_hijos[i-1].posición]
para i = 0 hasta tamaño_población
        índice_población[i].posición = i
        índice_población[i].evaluación = población[i].evaluación
ordena(índice_población)
return población[índice_población.posición].v
```

3.2.1 Modelo estacionario

En el modelo estacionario se seleccionan dos parejas de padres y se obtiene una pareja de hijos. Esa pareja compite contra los dos peores elementos de la población y los mejores son los que pasan a formar parte de la población.

```
mutaciones = 0
frecuencia_mutaciones = 1/prob_mutación
iteraciones = 0
Mientras interaciones < n_iteraciones
        padre1 = GeneraPadreEstacionario() //La selección de padres es igual a la del
                                                apartado 2.4
        padre2 = GeneraPadreEstacionario()
        padre3 = GeneraPadreEstacionario()
        padre4 = GeneraPadreEstacionario()

        si(ox)
                hijo1 = OX(padre1,padre2)
                hijo2 = OX(padre3,padre4)
        sino
                hijo1 = Posición(padre1,padre2)
                hijo2 = Posición(padre3,padre4)
        mutaciones += 2*tamaño_vector
        //Muta un gen si es necesario
        si(mutaciones > frecuencia_mutacion)
                hijo = randint(0,1)
                gen1 = randint(0,tamaño_vector-1)
                gen2 = randint(0,tamaño_vector-1)
                si(hijo == 0)
                        swap = hijo1.v[gen1]
                        hijo1.v[gen1] = hijo1.v[gen2]
                        hijo1.v[gen2] = swap
                sino
                        swap = hijo2.v[gen1]
                        hijo2.v[gen1] = hijo2.v[gen2]
                        hijo2.v[gen2] = swap
                mutaciones = mutaciones%frecuencia_mutación
        //Torneo entre los peores de la generación y los hijos
        vector torneo
        torneo[0].posición = índice_segundo_peor
        torneo[0].evalaución = evalaución_segundo_peor
        torneo[1].posición = índice_peor
        torneo[1].evaluación = evaluación_peor
        torneo[2].posición = -1 //representa al hijo1
        torneo[2].evalaución = hijo1.evaluación
```

```
        torneo[3].posición = -2 //representa al hijo2
        torneo[3].evalaución = hijo2.evaluación
        sort(torneo)
        si torneo[0].posición == -1
                población[índice_peor] = hijo1
                si torneo[1].posición == -2
                        población[índice_segundo_peor] = hijo2
        sino
                si torneo[0].posición == -2
                        población[índice_peor] = hijo2
                        si torneo[1].posición == -1
                                población[índice_segundo_peor] = hijo1
finmientras
return población[índice_mejor]
```

3.3 Algorítmos meméticos

Los algoritmos meméticos consisten en hacer potenciar a los algoritmos genéticos mejorando la calidad de la población. En este caso se usa una búsqueda local sobre algunos de los miembros de la población genética cada 10 generaciones. Se han desarrollado 3 modalidades, la primera que lanza la búsqueda local sobre toda la población, otra que lo hace sobre un número concreto de individuos aleatorios y otra que lo hace sobre los mejores elementos de la población. Para simplificar el pseudocódigo descrito posteriormente se entiende que la creación de generaciones se realiza de la forma descrita en el algoritmo genético generacional.

3.3.1 Memético 1

```
iteraciones = 0
generaciones = 0
vector generación_actual //tiene almacenado una generación inicial aleatoria
mientras iteraciones < max_iteraciones
        GeneraGeneracion(generación_actual) //crea una nueva generación según el modelo
                                            Generacional descrito anteriormente
        generaciones++
        si generaciones == 10
                //Lanza las búsquedas locales
                para i = 0 hasta tamaño_población
                        BusquedaLocal(generación_actual[i],tope_iteraciones,iteraciones)
                generaciones = generaciones % 10
        //Al necesita la generación genética un vector de índices, es necesario restablecerlo
        Para i = 0 hasta tamaño_población
                índices_población[i].posición = i
                índices_población[i].evaluación = generación_actual[i].evaluación
        endfor
mejor busca_mejor(generación_actual)
return generación_actual[mejor]
```

3.3.2 Memético 2

```
iteraciones = 0
generaciones = 0
vector generación_actual //tiene almacenado una generación inicial aleatoria
mientras iteraciones < max_iteraciones
        GeneraGeneracion(generación_actual) //crea una nueva generación según el modelo
                                                Generacional descrito anteriormente
        generaciones++
        si generaciones == 10
                //Lanza las búsquedas locales
                n_búsquedas = prob_búsqueda *tamaño_población
                para i = 0 hasta n_búsquedas
                        índice = randint(0,tamaño_población-1)
                BusquedaLocal(generación_actual[índice],tope_iteraciones,iteraciones)
                generaciones = generaciones % 10
        //Al necesita la generación genética un vector de índices, es necesario restablecerlo
        Para i = 0 hasta tamaño_población
                índices_población[i].posición = i
                índices_población[i].evaluación = generación_actual[i].evaluación
mejor busca_mejor(generación_actual)
return generación_actual[mejor]
```

3.3.3 Memético 3

```
iteraciones = 0
generaciones = 0
vector generación_actual //tiene almacenado una generación inicial aleatoria
mientras iteraciones < max_iteraciones
        GeneraGeneracion(generación_actual) //crea una nueva generación según el modelo
                                                Generacional descrito anteriormente
        generaciones++
        si generaciones == 10
                //Lanza las búsquedas locales
                //ordena la generación actual


                //primero restaura los índices
                Para i = 0 hasta tamaño_población
                        índices_población[i].posición = i
                        índices_población[i].evaluación = generación_actual[i].evaluación
                sort(índices_población)
                n_búsquedas = prob_búsqueda *tamaño_población
                para i = 0 hasta n_búsquedas
```

```
                índice = índice_población[i].posición
                BusquedaLocal(generación_actual[índice],
                tope_iteraciones,iteraciones)

        generaciones = generaciones % 10
    //Al necesitar la generación genética un vector de índices, es necesario restablecerlo
    Para i = 0 hasta tamaño_población
        índices_población[i].posición = i
        índices_población[i].evaluación = generación_actual[i].evaluación
mejor busca_mejor(generación_actual)
return generación_actual[mejor]
```

4 Análisis de los resultados obtenidos y replicación

Los resultados se han obtenido en base a casos de tamaños comprendidos entre 20 y 256. La semilla aleatoria que se ha usado para lanzar los algoritmos es 3.

Para compilar el archivo .cpp solo es necesario usar g++ -O2 –o p1 p1.cpp en un terminal Linux y para ejecutar basta con ejecutar el script lanza.sh que lanza el programa para todos los archivos o usar ./p1 nombrearchivo semilla.

4.1 Búsqueda local

Búsqueda local			
Caso	**Coste obtenido**	**Desv**	**Tiempo**
Chr20b	3270,00	42,30	0,0005030
Chr22a	7050,00	14,52	0,0008860
Els19	23688500,00	37,62	0,0005020
Esc32b	228,00	35,71	0,0019480
Kra30b	98480,00	7,72	0,0019920
Lipa90b	15250729,00	22,10	0,0456590
Nug25	3828,00	2,24	0,0009030
Sko56	35348,00	2,58	0,0121000
Sko64	49766,00	2,61	0,0183750
Sko72	68668,00	3,64	0,0241390
Sko100a	155542,00	2,33	0,0852030
Sko100b	156922,00	1,97	0,0933340
Sko100c	150352,00	1,68	0,0826670
Sko100d	152214,00	1,76	0,0871600
Sko100e	153946,00	3,22	0,0852710
Tai30b	816324715,00	28,13	0,0023290
Tai50b	479142881,00	4,43	0,0116460
Tai60a	7531464,00	4,52	0,0123590
Tai256c	45274196,00	3,25	0,6035390
Tho150	8315108,00	9,11	0,2921450
		11,57	0,07

La búsqueda local produce unos resultados relativamente buenos, pero no mejores que algunos de los que se estudiarán a continuación. Este algoritmo presenta algunos resultados malos en algunos de los conjuntos de datos. Dichos conjuntos no son similares en tamaño, por lo cual podemos deducir que los malos resultados se deben a la componente aleatoria del algoritmo. El vector de partida puede significar mucho en cuanto al resultado. El hecho además de realizar la exploración quedándote con el primer mejor hace que dejemos zonas más prometedoras sin explorar. El principal fallo que presenta este algoritmo es que queda atrapado en óptimos locales muy rápidamente. Esto también puede verse como una ventaja en cuanto al tiempo de ejecución respecta. La rápida convergencia del algoritmo hace que presente resultados de forma veloz, y si el resultado no se puede considerar bueno se puede ejecutar varias veces. Aunque este caso lo que le otorga a este algoritmo esta ventaja temporal también es la posible factorización del resultado, lo que agiliza mucho la generación de vecinos. En otros problemas podemos encontrar que la exploración de vecinos es demasiado lenta y otras técnicas pueden ofrecer mejores resultados en tiempos similares.

4.2 Estacionario con operador OX

Estacionario OX			
Caso	**Coste obtenido**	**Desv**	**Tiempo**
Chr20b	2992	30,20	0,120719
Chr22a	6720	9,16	0,14865
Els19	20940212	21,66	0,113229
Esc32b	252	50,00	0,297086
Kra30b	96600	5,67	0,235578
Lipa90b	15251184	22,10	2,01755
Nug25	3864	3,21	0,191082
Sko56	35178	2,09	0,803349
Sko64	50212	3,53	1,0162
Sko72	68518	3,41	1,31264
Sko100a	157082	3,34	2,44161
Sko100b	159712	3,78	2,46145
Sko100c	152736	3,30	2,55577
Sko100d	154018	2,97	2,44083
Sko100e	155404	4,19	2,6633
Tai30b	641345356	0,66	0,237075
Tai50b	490574934	6,92	0,64114
Tai60a	7531178	4,51	0,892105
Tai256c	45167376	3,01	15,7615
Tho150	8528202	11,91	5,62057
		9,78	2,10

El algoritmo de generación estacionaria si presenta mejores resultados que la búsqueda local y también lo hace con respecto a el modelo generacional como veremos más adelante. El hecho de que los hijos deban competir para entrar en la población hace que se presente un mínimo de calidad en la población y que el algoritmo converja hacia mejores soluciones. El tiempo es más lento que la búsqueda local, pero eso se debe a que este algoritmo debe realizar más evaluaciones sin factorizar que la búsqueda local. Aunque los modelos genéticos se ven favorecidos de esta factorización, lo hacen en menor medida, ya que solo la usan a la hora de mutar a sus hijos.

4.2 Estacionario con operador de posición

Caso	Coste obtenido	Desv	Tiempo
Chr20b	2828	23,06	0,128637
Chr22a	7530	22,32	0,156073
Els19	17997928	4,56	0,125245
Esc32b	220	30,95	0,314405
Kra30b	95800	4,79	0,244878
Lipa90b	15246715	22,07	2,11781
Nug25	3868	3,31	0,236105
Sko56	35748	3,74	1,10063
Sko64	49962	3,02	1,08019
Sko72	68382	3,21	1,37423
Sko100a	156760	3,13	2,64108
Sko100b	157990	2,66	2,47442
Sko100c	154138	4,24	2,46816
Sko100d	152876	2,21	2,47283
Sko100e	154338	3,48	2,63403
Tai30b	669679891	5,11	0,255398
Tai50b	494215287	7,71	0,665722
Tai60a	7569036	5,04	0,930182
Tai256c	45821564	4,50	15,9896
Tho150	8496600	11,49	5,65855
		8,53	2,15

El operador de posición nos proporciona mejores soluciones, pero la diferencia apenas es significativa como para extraer conclusiones claras. Este hecho podría deberse simplemente a que las partes no comunes se han rellenado de forma aleatoria dando lugar a mejores soluciones o porque a medida que los padres convergen hacia mejores soluciones se parecen más entre ellos y los genes buenos permanecen intactos, mientras que unos pocos genes son los que se van variando y dando lugar a soluciones muy parecidas entre ellas.

4.3 Generacional con operador OX y posición

Generacional Posición			
Caso	Coste obtenido	Desv	Tiempo
Chr20b	5742	149,87	0,131839
Chr22a	9346	51,82	0,146906
Els19	28792292	67,28	0,120539
Esc32b	372	121,43	0,323856
Kra30b	118370	29,48	0,249504
Lipa90b	15369867	23,05	2,02421
Nug25	4484	19,76	0,185195
Sko56	36042	4,60	0,841411
Sko64	50406	3,93	1,03884
Sko72	69226	4,48	1,34619
Sko100a	157624	3,70	2,52625
Sko100b	160972	4,60	2,53748
Sko100c	153878	4,07	2,50132
Sko100d	156044	4,32	2,65496
Sko100e	157500	5,60	2,53003
Tai30b	893559554	40,25	0,257185
Tai50b	662137959	44,31	0,651658
Tai60a	7622692	5,78	0,932857
Tai256c	45552844	3,88	17,249
Tho150	8588962	12,71	5,8808
		30,25	2,21

Generacional OX			
Caso	Coste obtenido	Desv	Tiempo
Chr20b	6208	170,15	0,118833
Chr22a	9542	55,00	0,145957
Els19	27669572	60,75	0,107217
Esc32b	352	109,52	0,298913
Kra30b	121920	33,36	0,23358
Lipa90b	15476151	23,90	2,0407
Nug25	4398	17,47	0,169913
Sko56	36336	5,45	0,816677
Sko64	51016	5,19	1,0336
Sko72	69834	5,40	1,30915
Sko100a	160632	5,68	2,62884
Sko100b	162728	5,74	2,44058
Sko100c	156784	6,03	2,4712
Sko100d	158074	5,68	2,50933
Sko100e	158050	5,97	2,52686
Tai30b	952832870	49,55	0,249293
Tai50b	698975497	52,34	0,738124
Tai60a	7708906	6,98	0,992594
Tai256c	45370554	3,47	16,8452
Tho150	8663204	13,68	5,59376
		32,07	2,16

Este algoritmo es el que presenta peores solucione con mucha diferencia frente al resto. Estos resultados se deben al modelo de reemplazamiento generacional. En la implementación probada todos los hijos, independientemente de su calidad son introducidos en la nueva población, conservando solo el mejor de los anteriores. Esto posibilita la entrada a individuos que empeoran la calidad media del conjunto de elementos de la población. En algunos casos se muestran resultados con una desviación cercana al 5%, pero son casos en los que el resto de algoritmos han obtenido desviaciones menores. Al igual que en el caso anterior el modelo basado en el cruce de posición obtiene resultados algo mejores, tanto de forma global cono en cada uno de los casos.

4.5 Algoritmos meméticos

Memético 2				Memético 1			
Caso	Coste obtenido	Desv	Tiempo	Caso	Coste obtenido	Desv	Tiempo
Chr20b	2796	21,67	0,043644	Chr20b	2536	10,36	0,048233
Chr22a	6572	6,76	0,047204	Chr22a	6480	5,26	0,048371
Els19	17436428	1,30	0,040279	Els19	17245098	0,19	0,045099
Esc32b	200	19,05	0,091108	Esc32b	200	19,05	0,0722
Kra30b	95410	4,36	0,080797	Kra30b	94880	3,78	0,073065
Lipa90b	15504360	24,13	0,439672	Lipa90b	15554953	24,53	0,393259
Nug25	3814	1,87	0,054798	Nug25	3772	0,75	0,055803
Sko56	35748	3,74	0,204763	Sko56	36148	4,90	0,205003
Sko64	50906	4,97	0,263468	Sko64	51232	5,64	0,25561
Sko72	69780	5,32	0,30732	Sko72	70370	6,21	0,308536
Sko100a	160226	5,41	0,586801	Sko100a	161794	6,44	0,604593
Sko100b	161702	5,08	0,599112	Sko100b	163112	5,99	0,63739
Sko100c	156914	6,12	0,615667	Sko100c	157070	6,23	0,612205
Sko100d	157982	5,62	0,596694	Sko100d	158424	5,92	0,577427
Sko100e	157440	5,56	0,605747	Sko100e	158216	6,08	0,591483
Tai30b	653817727	2,62	0,077036	Tai30b	642059766	0,78	0,082299
Tai50b	489784589	6,75	0,181019	Tai50b	487963951	6,35	0,194512
Tai60a	7664352	6,36	0,213709	Tai60a	7710288	7,00	0,193505
Tai256c	47653996	8,68	2,31841	Tai256c	47633642	8,63	1,57933
Tho150	8654906	13,57	1,34624	Tho150	8853114	16,17	1,38477
		7,95	0,44			7,51	0,40

Memético 3			
Caso	**Coste obtenido**	**Desv**	**Tiempo**
Chr20b	2924	27,24	0,048645
Chr22a	7002	13,74	0,049266
Els19	20265368	17,74	0,042084
Esc32b	200	19,05	0,076808
Kra30b	94940	3,85	0,065694
Lipa90b	15289131	22,41	0,345119
Nug25	3788	1,18	0,053036
Sko56	35466	2,93	0,168036
Sko64	49536	2,14	0,21092
Sko72	67766	2,28	0,238936
Sko100a	155182	2,09	0,41619
Sko100b	157398	2,28	0,427305
Sko100c	153076	3,53	0,418109
Sko100d	153372	2,54	0,422584
Sko100e	153450	2,88	0,421062
Tai30b	656174831	2,99	0,065754
Tai50b	470046298	2,45	0,14233
Tai60a	7555230	4,85	0,174975
Tai256c	46204860	5,37	2,00479
Tho150	8490682	11,42	0,909796
		7,65	0,34

Las diferencias entre las tres versiones no son lo suficientemente relevantes como para establecer una ventaja clara de unos sobre otros. En el caso de la primera versión obtiene un promedio mejor, aunque en algunos casos empeora, pero por muy poco y es el que ha obtenido desviaciones menores al 1%. El hecho de potenciar a toda la población a costa de producir más hijos le permite tener individuos muy buenos para cruzar. Los otros dos modelos únicamente se diferencian en que el tercero potencia más a los mejores individuos de la población y el segundo crea una población algo más homogénea. Pero su estrategia es no gastar demasiadas iteraciones en mejorar al conjunto de la población y usarlas para generar más hijos. Esto hace que se presenten individuos de peor calidad en los cruces y generen poblaciones de peor calidad.

4.6 Vistazo general

Algoritmo	desviacón	tiempo
BL	11.57	0,07
AGG-posición	30,25	2,21
AGG-OX	32,07	2,16
AGE-posición	8,53	2,15
AGE-OX	9,78	2,1
Memético1	7,51	0,4
Memético2	7,95	0,44
Memético3	7,65	0,34
Greedy	76,12	tiende a 0