Práctica 3

Algoritmos Genéticos para el
Problema de la Asignación Cuadrática
AGG-POS • AGG-PMX • AGE-POS • AGG-PMX

Mario Ruiz Calvo
mariorc@correo.ugr.es
33958755-Z

Grupo 2
(J 17:30-19:30)

Curso 2014-2015

INDICE

1. Descripción del Problema

El problema de asignación cuadrática es un problema de optimización combinatioria.. En éste se trata de asignar N unidades a una cantidad N de localizaciones en donde se considera un costo asociado a cada una de las asignaciones. Este costo dependerá de las distancias y flujo entre las instalaciones. Por lo que el problema consiste en encontrar la asignación óptima de caada unidad a una localización. La función objetivo mide la bondad de una asignación, considerada el producto de la distancia entre cada par de localizaciones y el flujo que circula entre cada par de unidades.

2. Componentes de los algoritmos

- Esquema de representación: La solución se representa con un vector en forma de permutación π de tamaño n, donde los indices del vector corresponden a las unidades y el contenido a las distancias.

- Función objetivo:

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

- Generación de soluciones aleatorias:

```
GenerarSolucionAleatoria(sol)
        Desde i=1 hasta n (tamaño del problema)
                Hacer
                        r = rand(0,n)
                Mientras r este contenido en sol
                sol[i] = r
        Fin
Fin
```

- Descripción del mecanismo de selección:

```
seleccion(poblacion,padres,npadres)
        Desde i=1 hasta npadres hacer
                j=rand(0,N_CROMOSOMAS)
                k=rand(0,N_CROMOSOMAS) (k distinto de j)
                Si ( coste(poblacion[j]) < coste(poblacion[k]) )
                        padres[i]=poblacion[j]
                Sino
                        padres[i]=poblacion[k]
        Fin
Fin
```

4

- Descripción del operador de cruce:

```
crucePosicion(padres,ncruces)
        Desde i=1 hasta ncruces hacer
                        Desde j=1 hasta n hacer
                                Si padres[i*2][j] != padres[(i*2)+1][j]
                                        Insertar j en posiciones
                                        Insertar padres]i*2][j] en asignaciones
                                FinSI
                        Fin
                        Mientras posiciones no sea vacio
                                pos=rand(0,posiciones.size())
                                padres[i*2][posiciones[pos]]=asignaciones[0]
                                Eliminar posiciones[pos]
                                Eliminar asignaciones[0]
                        FinMientras
        Fin
Fin
```

```
cruceOX(padres,npadres,ncruces,nsubcadena)
        Inicial = (n/2) – (nsubcadena/2)
        Desde i=1 hasta ncruces hacer
                        Desde j=1 hasta inicial hacer
                                Insertar padres[i*2][j] en aux
                        Desde j=inicial hasta inicial+nsubcadena
                                hijo[j]=padres[i*2][j]
                        Desde j=inicial+nsubcadena hasta n
                                Insertar padres[i*2][j] en aux

                        Desde j=1 hasta n
                                Si padres[(i*2)+1][j] esta en aux
                                        Insertar padres[(i*2)+1][j] en aux2

                        K=0
                        Desde j=inicial+nsubcadena hasta n
                                hijo[j]=aux2[k]
                                K++
                        Desde j=1 hasta inicial
                                hijo[j]=aux2[k]

                        padres[i*2]=hijo

                        Limpiar aux
                        Limpiar aux2
        Fin
Fin
```

```
mutacion(padres,npadres,prob_mutacion)
        Desde i=1 hasta npadres
                        p=rand(0,1)
                        Si prob_mutacion<p
                                GeneraVecinoAleatorio(padres[i])
        Fin
Fin
```

```
GeneraVecinoAleatorio(sol)
        r=rand(0,n)
                s=rand(0,n) (r distinto de s)
                intercambiar(sol[r],sol[s])
Fin
```

Para evitar genererar muchos números aleatorios que después no se van a utilizar en el esquema generacional, se calcula previamente el número de mutaciones que se van a realizar como *prob_mutacion*n*N_CROMOSOMAS* y se mutan directamente.

```
mutacionG(padres,npadres,nmutaciones)
        Desde i=1 hasta nmutaciones
                        p=rand(0,N_CROMOSOMAS) (p no repetido)
                        GeneraVecinoAleatorio(padres[p])
        Fin
Fin
```

3. Algoritmos Genéticos

```
reemplazamientoG(poblacion,padres,mejor_sol)
        poblacion=padres
                Si Min(costes(padres)) == coste(mejor_sol)
                        poblacion[Max(costes(padres))]=coste(mejor_sol)
Fin
```

```
reemplazamientoE(poblacion,padres,mejor_sol)
        ordenar(poblacion)
        Si padres[0] && padres[1] son menores que poblacion[n] y poblacion[n-1]
                Intercambiar(padres[0],poblacion[n])
                Intercambiar(padres[1],poblacion[n-1])
Fin
```

Se utilizará el mismo esquema para las cuatro versiones del algoritmo genético cambiando en cada caso el número de padres ha considerar, el operador de cruce (OX o Posicion) y el tipo de reemplazamiento.

```
esquemaGenetico(sol,coste, prob_cruce, prob_mutacion)
        Ncruces = prob_cruce*(N_CROMOSOMAS)/2
        Nmutaciones = prob_mutacion*n*N_CROMOSOMAS

        inicializar(poblacion,padres)
        evaluar(poblacion, costes_poblacion, sol, coste)

        Desde i=1 hasta 25000
                seleccion(poblacion,padres,npadres)
                cruce(padres,ncruces)
                mutacion(padres,npadres,prob_mutacion)
                reemplazamiento(poblacion,padres,mejor_sol)
                evaluar(poblacion, costes_poblacion, sol_l, coste_l)
                ActualizarMejorSolucion(sol, sol_l,coste,coste_l)
        FinDesde
Fin
```

4. Procedimiento para la práctica

Para el desarrollo de la práctia se ha partido de un código desde cero, siguiendo como referencia el código proporcionado en la plataforma DECSAI y en el pseudocódigo y explicación de los seminarios y las tranparencias de teoría.

La practica se orgainza en las siguientes carpetas:

- bin: contiene los ficheros ejecutables
- data: contiene los ficheros .dat de donde se extraen los flujos y distancias.
- Include: contiene las ficheros .h
- lib: contien la librerias con todos lof ficheros objeto
- obj: contiene los ficheros objeto .o
- resultados: contiene las tablas con los resultados que se han obtenido para los distintos algoritmos y las desviaciones de tiempo y costo.
- Src: contiene el código fuente.

Además el directorio raíz contiene un fichero makefile para compilar los ejecutables y dos scripts con los que se obtienen los resultados en un formato fácil de trasladar a las tablas.

Ejecutando el fichero makefile se obtienen dos ejecutables: main y obtener_resultados:

main: Ejecuta la búsqueda local primer mejor, la búsqueda multiarranque básica y la busqueda local retroactiva, con una semilla fija (5555) , mostrando el costo obtenido y el tiempo.

-modo de empleo (desde directorio practica2):

./bin/main data/nombre_del_fichero.dat

obtener_resultado: Realiza una ejecución de un algoritmo pasado por parámetro con una semilla variable (también pasada como

parámetro) e imprime el coste y el tiempo de ejcución en un formato adecuado para la extracción de datos para las 20 instancias.

-modo de empleo (desde directorio practica2):

./bin/obtener_resultado id_algoritmo semilla

identificador de algoritmo:
- 0: greedy
- 1: búsqueda local primero mejor
- 2: BMB
- 3: GRASP
- 4: ILS

Ejecutando el script obtener_resultado.ssh se realizan 20 ejecuciones DE LAS 20 INSTANCIAS del algoritmo pasado por parámetro, cada una con diferente semilla. Se imprimran los resultados en un formato adecuado para recoger los datos en tablas.

-modo de empleo (desde directorio practica2):

./obtener_resultados.ssh id > resultados/nombre

5. Análisis de resultados

Para obtener los resulados se ha realizado una única ejecuciones de las 20 instancias con semilla 5555.

Greedy					
Caso	**Desv**	**Tiempo**	**Caso**	**Desv**	**Tiempo**
Chr20b	342,82	0,000046	**Sko64**	18,39	0,000132
Chr20c	456,63	0,00003	**Sko72**	15,16	0,000161
Chr22a	130,31	0,000045	**Sko81**	15,11	0,000208
Chr22b	134,48	0,000034	**Sko90**	13,49	0,000278
Els19	124,42	0,000027	**Sko100a**	14,37	0,000185
Esc32b	140,48	0,00007	**Sko100b**	13	0,000207
Kra30b	28,04	0,000056	**Sko100c**	12,89	0,00018
Lipa90b	29,36	0,000422	**Sko100d**	14,07	0,000227
Nug30	21,69	0,00002	**Sko100e**	14,64	0,000178
Sko56	16,09	0,000104	**Wil50**	10,59	0,000056

AGGpos					
Caso	**Desv**	**Tiempo**	**Caso**	**Desv**	**Tiempo**
Chr20b	13,75	4,661348	Sko64	**2,16**	43,399052
Chr20c	49,38	4,741295	Sko72	2,16	54,73518
Chr22a	6,69	5,397944	Sko81	2,57	68,915314
Chr22b	**5,94**	5,569252	Sko90	**1,97**	90,476707
Els19	**0**	4,357876	Sko100a	**1,46**	104,445107
Esc32b	**9,52**	11,96969	Sko100b	**1,79**	104,601967
Kra30b	4,88	9,860563	Sko100c	2,74	104,687782
Lipa90b	**22,07**	84,647217	Sko100d	**2,05**	104,136032
Nug30	4,41	9,678495	Sko100e	**1,79**	104,032654
Sko56	2,98	33,489449	Wil50	1,45	27,716801

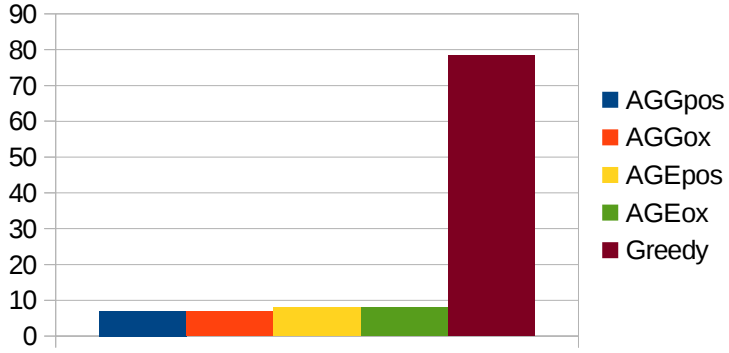
AGGox					
Caso	**Desv**	**Tiempo**	**Caso**	**Desv**	**Tiempo**
Chr20b	**5,92**	6,38619	Sko64	2,57	51,532604
Chr20c	35,17	6,464711	Sko72	2,27	64,776062
Chr22a	11,7	7,436623	Sko81	**2,48**	81,889687
Chr22b	9,78	7,391784	Sko90	2,67	100,524246
Els19	**0**	5,900079	Sko100a	1,64	124,15818
Esc32b	19,05	14,297687	Sko100b	2,35	122,646873
Kra30b	6,23	12,867749	Sko100c	**2,2**	122,631126
Lipa90b	22,08	100,360207	Sko100d	2,34	124,422592
Nug30	**2,02**	13,135749	Sko100e	1,98	123,881088
Sko56	4,48	41,045452	Wil50	1,65	35,433498

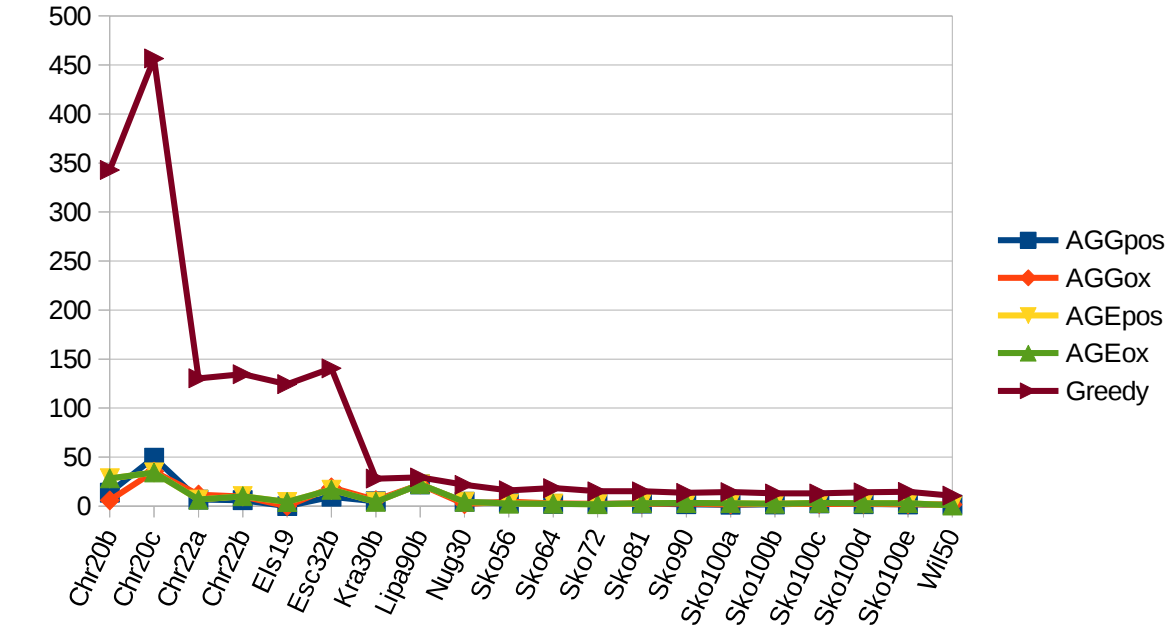
AGEpos					
Caso	**Desv**	**Tiempo**	**Caso**	**Desv**	**Tiempo**
Chr20b	28,37	4,326752	Sko64	2,41	39,533844
Chr20c	**34,24**	4,312407	Sko72	**1,9**	52,390858
Chr22a	**6,66**	4,997578	Sko81	3,08	63,938206
Chr22b	10,17	5,061188	Sko90	3,13	78,651848
Els19	4,21	4,058217	Sko100a	2,99	99,169136
Esc32b	16,67	10,600887	Sko100b	2,44	98,925339
Kra30b	**4,34**	9,515444	Sko100c	3,17	106,019417
Lipa90b	22,21	84,818176	Sko100d	2,82	103,68071
Nug30	4,47	9,135925	Sko100e	3,08	98,220627
Sko56	**2,58**	30,686382	Wil50	**0,68**	24,387119

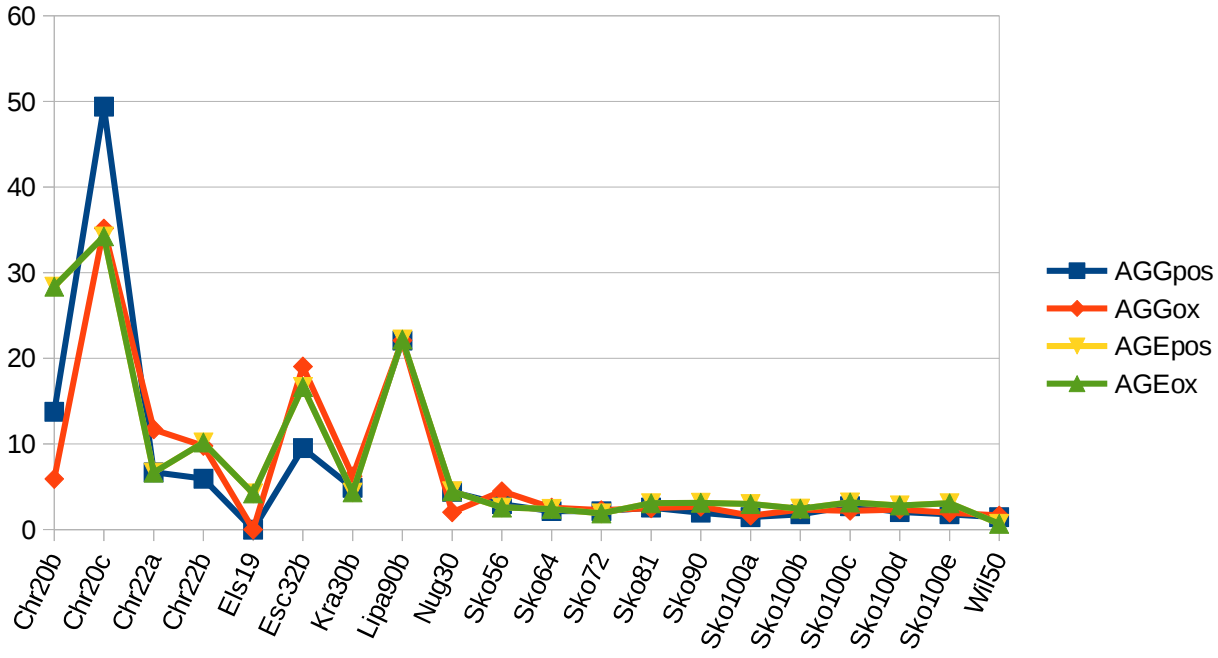
AGEox					
Caso	**Desv**	**Tiempo**	**Caso**	**Desv**	**Tiempo**
Chr20b	28,37	4,388088	Sko64	2,41	40,178318
Chr20c	**34,24**	4,387081	Sko72	**1,9**	51,142048
Chr22a	**6,66**	5,092771	Sko81	3,08	64,064247
Chr22b	10,17	5,12596	Sko90	3,13	80,21627
Els19	4,21	4,118004	Sko100a	2,99	98,628387
Esc32b	16,67	10,26131	Sko100b	2,44	98,059113
Kra30b	**4,34**	8,996072	Sko100c	3,17	97,374802
Lipa90b	22,21	79,578407	Sko100d	2,82	97,987144
Nug30	4,47	9,494042	Sko100e	3,08	97,819954
Sko56	**2,58**	31,174059	Wil50	**0,68**	25,342871

***nota: los resultados son iguales a AGEpos pero no se porque ocurre. Se han obtenido con obtener_resultados pero al ejecutarlos con main si salen distintos. No me ha dado tiempo ha ejecutarlo uno por uno para poner los valores correctos.**

Algoritmo	Desv	Tiempo
AGGpos	6,99	49,08
AGGox	6,93	58,36
AGEpos	7,98	46,62
AGEox	7,98	45,67
Greedy	78,3	0,0001333

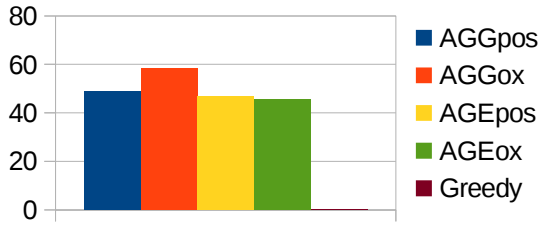


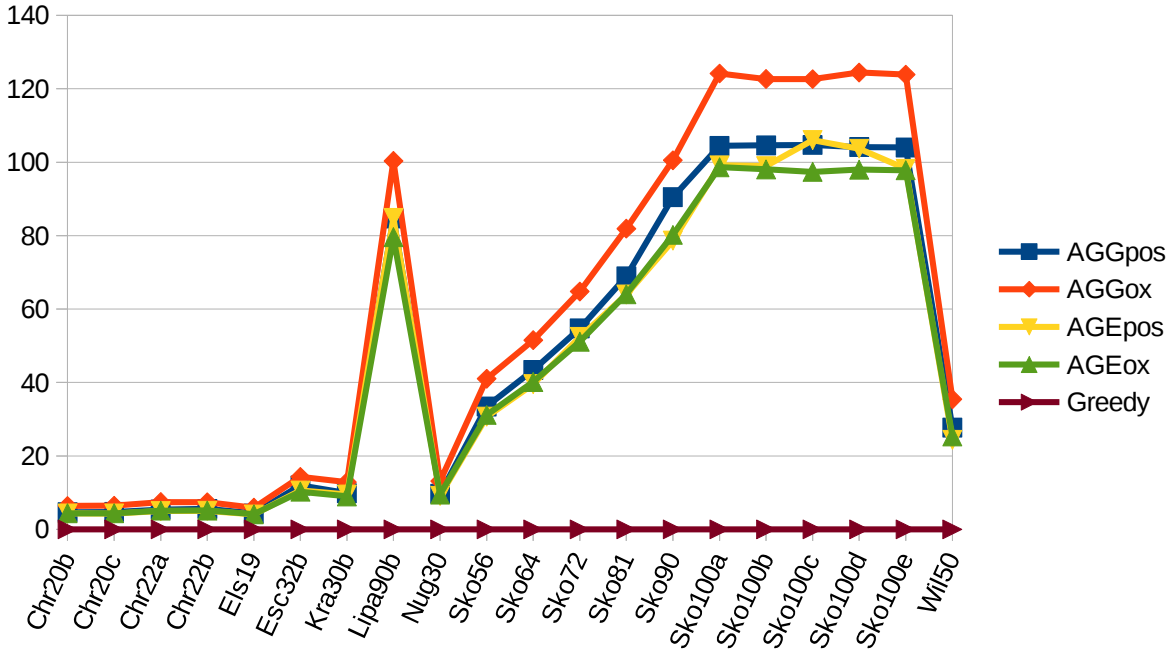




Los cuatro versiones se comportan de manera parecida. Cuando los tamaños de memoria son grandes la diferencia de desviaciones de un algoritmo a otro son menores y estan muy cercanas a 0 superando a los algoritmos de la práctica 2.

Las dos versiones generacionales, en media, tienen mejores resultados. Incluso llegan a alcanzar el óptimo en la instancia *els19*.





Los tiempos son muy altos en instancias con tamaño grande. Las versiones generacionales consumen mas tiempo.

En conclusión si queremos soluciones muy buenas en instancias grandes, podemos sacrificar un poco de tiempo para conseguirlo utilizando algoritmos genéticos. Pero si no podemos perder demasiado tiempo, podemos aceptar peores soluciones con algoritmos como los de la practica 2 que tienen tiempos mas bajos.