Metaheurística
Doble Grado en Ingeniería Informática y Matemáticas
Universidad de Granada

Práctica 1
Greedy y Busqueda Local en Problema de Asignación Cuadrática(QAP)

Iván Sevillano García

DNI: 77187364-P

E-mail: ivansevillanogarcia@correo.ugr.es

Grupo del martes, 17:30h-19:30h

29 de abril de 2018

Índice

1. Descripción del problema QAP

El problema que se nos plantea es el Problema de Asignación Cuadrática(en adelante QAP por las siglas). En él, tenemos una serie de instalaciones las cuales tienen que interactuar entre ellas una cierta çantidad de trabajo", produciendo un coste. Las instalaciones tienen, además, unas determinadas localizaciones en las que se tienen que situar. Dependiendo de la distancia entre cada dos instalaciones, el coste que producen al interactuar es mayor o menor.

Cabe destacar una serie de detalles en el problema:

- **No Euclideo(No Métrico).** Este problema no pone ninguna objeción a que la distancia de una localización de un lugar a si mismo sea mayor que cero. Además, una instalación puede interactuar consigo misma por consiguiente.
- **No simétrico.** Tampoco pone objeción a que la distancia de una localización a otra no sea la misma que de otra a una.

El problema entonces consiste en repartir las instalaciones en las localizaciones de forma que el coste total o trabajo sea mínimo. Esto es fácil de representar con una permutación, que a cada instalación i le asigna una localización $\pi(i)$. Si llamamos d_{ij} a la distancia que hay de la localización i a la j, y f_{ij} la cantidad de trabajo que tiene que mandar la instalación i a la j, la función de coste asociada al problema sería la siguiente:

$$Coste(\pi) = \sum_{i=1}^{N}\sum_{j=1}^{N} f_{ij}d_{\pi(i)\pi(j)}$$

Si consideramos las matrices de distancias y flujos, D y F y las matrices de que representan a cada permutación y su inversa, Π y Π^{-1} respectivamente, se puede representar el coste de una manera más sencilla:

$$Coste(\pi) =< F, \Pi^{-1}D\Pi >$$

Donde la aplicación $<,>$ tiene como argumentos dos matrices de mismas dimensiones y se aplica en la suma de la multiplicación de sus componentes una a una.

2. Breve descripción de los algorítmos utilizados.

En esta sección vamos a explicar brevemente la configuración de una solución concreta del problema, el funcionamiento de la función de coste y de cómo trabajan los dos algoritmos que se describen en el enunciado.

2.1. Solución.

Una solución está perfectamente determinada por un vector de N componentes donde cada una de sus componentes es distinta unas de otras y el rango de valores difiere de 1 hasta N, o lo que es lo mismo, la solución es una permutación. Según la misma, la localización i tendrá alojada la instalación con el número que ocupa en el vector la posición i. Para agilizar cálculos, cada permutación, además, guarda su coste asociado. Así, será fácil calcular soluciones vecinas.

2.2. Función de coste.

La función de coste es la descrita anteriormente en la primera sección. A continuación el psudo-código. Los parámetros hacen referencia a la matriz de distancias(D), la matriz de flujos(F) y a la permutación que representa nuestra solución:

```
Parametros:D,F,P
Output: coste
coste = 0
Para cada elemento de la matriz (i,j):
  coste <- coste + F[j][i]*D[P(j)][P(i)]
```

2.3. Algoritmos genéticos.

Los aspectos fundamentales de los algoritmos genéticos que vamos a programar son los siguientes:

- Creación de una población de 50 individuos aleatorios. Se usará el mismo método que se describe en la práctica anterior para generar 50 individuos.
- Un mecanismo de selección de soluciones para el posterior cruce. En esta práctica se usará una selección por torneo binario. Consiste en escogerá la mejor solución de dos soluciones dadas:

```
Input: P1,P2
Si P1 tiene mejor coste que P2:
  Devuelve P1
Si no:
  Devuelve P2
```

- Un algoritmo de mutación de soluciones. En nuestro caso, se considera que una mutación es la aplicación de una trasposición (i, j) a la solución. Esto es equivalente a generar el vecino de nuestra solución que surge al aplicar la trasposición (i, j) a la solución, que a la hora del cálculo de coste nos hará más rápido el cálculo. Este método ya está explicado en la práctica anterior.
- Un algoritmo de cruce de soluciones o cromosomas.
- Un modelo de evolución.

Para esta práctica se han detallado dos modelos de evolución y dos algoritmos de cruce de cromosomas, lo que nos deja con cuatro algoritmos genéticos básicos que surgen al usar una u otra metodología. A continuación, se detalla cada uno de ellos:

Algoritmos de cruce

- **Primer método**. El primer método de cruce que utilizamos copia las componentes iguales de las permutaciones y mezcla aleatoriamente los índices no iguales. P hace referencia a las permutaciones padre y H a las permutaciones hija:

```
Input: P1,P2
Output: H1, H2
H1, H2 = [-1..-1],[-1..-1]
En H1,H2 se copian las componentes iguales de P1,P2
en su posicion.

Los indices no usados se mezclan de dos formas distintas
(S1,S2) usando la funcion shuffle()

Se meten en H1(resp. H2) por orden los indices ordenados
de S1(resp. S2).
```

- **Segundo método,Partially Mapped Crossover(PMX)**. Este método copia una subcadena central de cada padre en hijos cruzados(P1 en H2, P2 en H1). Tras esto, se intenta copiar cada valor de cada padre en cada hijo. En caso de que ese valor ya esté en la subcadena copiada, se intenta introducir el elemento que sustituye a tal valor en la subcadena copiada del padre hasta que este sea un valor que no esté copiado ya en el hijo.

```
Input: P1,P2
Output: H1, H2

Se escoge un inicio y un final para la subcadena.
En H1(H2) se copia la subcadena de P2(P1).
```

```
Para el resto de indices i:
  valor = P1[i]
  Mientras valor este copiado en el hijo:
    valor <- P1[(posicion del indice valor en P2)]
  H1[i] = valor

(Se actua de la misma forma con H2)
```

2.3.1. Algoritmo genético con reemplazo VS estacionario

Se plantean dos formas de hacer evolucionar a la población:

- **Modelo de reemplazo con elitismo.** Con este diseño se conserva la idea de evolución generacional. En cada paso, la población es sustituida completamente por una nueva generación creada a partir de la misma población. Se sigue el siguiente esquema:

  - Se escogen por torneo binario copias de elementos de la población actual.
  - Con una probabilidad de p_{cruce} se escogen parejas de estas copias para que sus cruces pasen a la siguiente generación. En caso de no cruzar, son las propias copias las que formarán parte de la próxima generación.

    Para ahorrar tiempo de computo se ha decidido que se cruzarán los primeros cromosomas, una proporción equivalente a p_{cruce}, y los demás no. Puesto que a la hora de seleccionar la siguiente generación ya se ha reordenado aleatoriamente, este método de selección de cruce no supone una variación del algoritmo demasiado significativa en cuanto a aleatoriedad.
  - Para cada gen de cada cromosoma de la nueva generación y de acuerdo a p_{muta}(probabilidad de mutación del gen), se produce mutación del individuo.

    Si tam es el número de localizaciones del problema, cada cromosoma tiene exactamente $\frac{tam(tam-1)}{2}$ genes, cada una de las posibles trasposiciones. De la misma forma que en el apartado anterior, escogemos una cantidad proporcional a p_{muta} de genes de entre toda la población de forma aleatoria. Puesto que hay $n_{individuos} \times \frac{tam(tam-1)}{2}$ genes, hacemos $p_{muta} \times n_{individuos} \times \frac{tam(tam-1)}{2}$ elecciones de genes que mutan:

    ```
    Input: Pobl[indiv],p_muta,n_indiv,tam
    Hacemos (p_muta x n_indiv x tam(tam-1)/2) elecciones:
      Escogemos aleatoriamente un individuo
    ```

```
Escogemos aleatoriamente el gen al que se lo aplicamos(i,j).
Sustituimos al individuo por su mutacion(vecino(i,j)).
```

Cabe destacar que la forma en la que escogemos el gen a mutar es usando un número aleatorio entre 0 y $n^2 - n - 1$ y lo llamamos E. Si llamamos $i = Emod(n)$, $j = E/n$, la pareja (i, j) no tiene por qué ser una trasposición. Pero si $j >= i$ y lo modificamos por $j <= j + 1$, no solo estamos seguros de que será una trasposición. Además sabemos que cada trasposición (i, j) tiene las mismas probabilidades de ser escogida para la mutación.

- Se sustituye la población anterior por la nueva población. En caso de que la mejor solución anterior no haya sido igualada o superada por otra solución de la nueva generación, esta se inyecta en la población actual, desechando la peor solución de la nueva población.

- **Modelo estacionario.** Este modelo no sustituye totalmente a la generación anterior, si no que a las soluciones hijo que se generan se les hace competir con las peores soluciones para entrar en la población. Además, no hay posibilidad de que no crucen los padres escogidos. El algoritmo en cada iteración es como sigue:

  - Se seleccionan por torneo binario a dos padres, los cuales se cruzan y generan dos hijos.
  - De la misma forma que en el apartado anterior, se seleccionan los genes que van a mutar de entre los genes de los hijos generados. Puesto que el tamaño de algunos problemas es demasiado pequeño, imponemos que obligatoriamente mute un gen como poco. Esto lo hacemos para que, en caso de que entre los dos hijos no junten los suficientes cromosomas(1000), también muten. Esto se da si el tamaño del problema es menor de $33(33 * 32 = 1056)$.
  - Sacamos las dos soluciones que tienen peor función coste de la población actual.
  - De entre esas dos soluciones y las dos soluciones hijo, metemos las dos con mejor función coste.

2.4. Algoritmos meméticos.

En la práctica se nos propone realizar tres mejoras al algoritmo genético que mejor comportamiento tenga de entre los cuatro utilizados. Estas mejoras se basan en la aplicación de búsqueda local a dependiendo qué soluciones cada 10 generaciones con un límite de 400 evaluaciones para cada búsqueda local. Las mejoras propuestas son las siguientes:

- **Primera mejora.** Aplicar la búsqueda local a todas las soluciones de la población.
- **Segunda mejora.** Aplicar la búsqueda local a un 10 % aleatorio de la población.
- **Tercera mejora.** Aplicar la búsqueda local al 10 % mejor de la población.

3. Pseudocódigos y explicaciones.

A continuación se detallan los pseudocódigos de los algoritmos immplementados:

3.1. AG generacional.

```
Pobl = 50 soluciones aleatorias.
Hasta que se superen las 50.000 evaluaciones:
  Selecciona por torneoBinario 50 padres(copias).
  Sustituye el primer 70% por los hijos generados al cruzar cada 2.

  Para cada mutacion que debamos hacer en H hijo, (i,j):
    Sustituimos H por su vecino (i,j)
  Si la nueva poblacion no mejora a la mejor solucion
  anterior:
    Se sustituye la peor solucion por la mejor solucion de la
    poblacion anterior.

  Se sustituye la poblacion anterior por la actual.
```

3.2. AG estacionario.

```
Pobl = 50 soluciones aleatorias.
Hasta que se superen las 50.000 evaluaciones:
  Selecciona por torneoBinario 2 padres(copias).
  Genera dos hijos con los cruces seleccionados.

  Para cada mutacion que debamos hacer en H hijo, (i,j):
    Sustituimos H por su vecino (i,j)

  Sacamos de la poblacion anterior a las dos peores soluciones.

  Introducimos en la poblacion a las dos soluciones con
  mejor coste de entre las dos peores y los dos hijos.
```

3.3. Algoritmos meméticos

Las posibles mejoras meméticas se pueden modelizar con dos variables: la proporción de la población a la que se le va a aplicar la búsqueda local y si esa proporción va a ser la de los mejores:

```
Input: Pobl, p_pobl, mejores

Si se escogen los mejores:
  Se ordena el vector de poblacion por coste ascendente.
```

Si no y p_pobl < 1:
  Se mezcla el vector Pobl.

Se aplica la busqueda local con un tope de 400 pasos a las primeras soluciones de la poblacion(Cantidad proporcional a p_pobl).

3.4. Consideraciones de parada del algoritmo.

En la práctica se pide que en el momento en el que se evalúe 50.000 veces la función de coste los algoritmos deben parar. A partir de aquí, se hacen las siguientes consideraciones:

- Cada solución guarda su coste y partir del mismo, se puede cacular facilmente el coste del vecino. Pero si dos soluciones se cruzan no hemos visto cómo factorizar el coste y habrá que evaluar la función objetivo sin factorización.
- En caso de mutar alguno de estos hijos la mutación no contará como evaluación, ya que no tendrá coste de la solución anterior. Se calculará el coste despues de haber mutado. Así nos ahorramos una evaluación de la función coste.
- Si a una copia de un padre se le hacen varias mutaciones, estas si se consideran distintas evaluaciones ya que si se puede factorizar el coste paso a paso. Es más rápido, por descontado, que calcular la función de coste al final completa pero consideramos que se ha evaluado varias veces la función.
- Dentro de las búsquedas locales de los algoritmos genéticos cabe destacar que están muy desmejorados ya que si se tiene más de 400 genes, nunca llegará a mejorar los últimos genes(en tai256c hay 32640 genes por individuo). Además, la idea de Don't look bits pierde sentido ya que apenas nos da tiempo en 400 iteraciones a tachar bits y ahorrar tiempo de ejecución.

Estas consideraciones nos pueden dar ideas para posibles mejoras.

4. Algoritmo de comparación.

En esta sección se compararán los resultados de los algoritmos Greedy y búsqueda local. Para dicha comparación, atenderemos a dos estadísticos:

- **Desviación a la solución óptima.** Este estadístico evaluará proporcionalmente la diferencia de coste de la mejor solución y la solución obtenida en cada instancia.La fórmula que describe este estadístico es el siguiente:

$$Desv = \frac{1}{|casos|} \sum_{i \in casos} 100 \frac{valorAlg_i - mejorVal_i}{mejorVal_i}$$

- **Tiempo medio de ejecución.** La media de los tiempos de ejecución de cada algoritmo.

Consideraremos entonces que un algoritmo es mejor que otro si la desviación a la solución óptima es menor. En caso de ser iguales, el algoritmo que tarde menos será considerado mejor.

5. Manual de uso.

Para la implementación de la práctica hemos utilizado el lenguaje precompilado Python3, por lo que debe de estar instalado en el sistema. También hemos hecho uso de la biblioteca random del mismo lenguaje, más en concreto de las funciones randint(inicio,fin), que escoge aleatoriamente un valor entero entre *inicio* y $fin - 1$, y shuffle(vector), que mezcla el vector que se le pasa como argumento.

La forma de utilizar el programa es la siguiente:

- **Programas.** Esta practica tiene un total de siete ejecutables. Cadaejecutable recibe como argumento el nombre del archivo de entrada con los datos(terminado en '.dat'), que debe de estar en el directorio *./qapdata/*. Tiene que haber también un archivo solución con el mismo nombre pero con terminación '.sln' en el directorio *./qapsoln/*. Es posible también introducir una semilla como segundo argumento y un fichero de salida:

  ./nombreAlgoritmo.py ./qapdata/nombredatos.dat semilla $fichero_salida$

  Nos dará como resultado:

  - Solución Algoritmo. La primera linea será la solución del algoritmo en cuestión obtenida. La segunda, cuanto tiempo ha tardado el programa en calcularla y cuanto coste total tiene esta solución.
  - Mejor solución. La primera linea es la configuración de la mejor solución y la siguiente es su coste.

- **Experimento total.** Para ejecutar el experimento completo, nos hemos ayudado de un script que ejecuta todos los casos de prueba y un makefile que lo llama. Así, la forma de obtener todas las soluciones será tan simple como ejecutar el siguiente comando:

  make all

  Es importante mencionar que dentro de la ejecución de los programas se les ha introducido una semilla en concreto, 200000, que no modificaremos.

  Todos los resultados nos los encontramos en las carpeta *./solutionAlgoritmo/* con el mismo nombre del archivo de datos pero con terminación ".sol". Se han generado también tablas en Latex e imágenes que nos ayudarán en el estudio de los algoritmos. Para la obtención de gráficos se han creado tres scripts los cuales se ejecutan

de la siguiente manera:

./AllGraphics.sh
./graficosComparativa.sh
./graficosComparativaGeneracionales.sh

6. Experimento y análisis de resultados.

Las tablas obtenidas para cada algoritmo son las siguientes:

Caso	MejorCoste	Coste obtenido	Desviación	tiempo
chr22a	6156	6890	11.92	5.93
chr22b	6194	6788	9.58	5.89
chr25a	3796	6760	78.08	6.44
esc128	64	192	200.0	20.82
had20	6922	6948	0.37	5.19
lipa60b	2520135	3114899	23.6	14.04
lipa80b	7763962	9820192	26.48	16.46
nug28	5166	5346	3.48	7.31
sko81	90998	100144	10.05	16.64
sko90	115534	128098	10.87	17.47
sko100a	152002	169456	11.48	18.63
sko100f	149036	165620	11.12	18.57
tai100a	21052466	23246764	10.42	18.79
tai100b	1185996137	1509118098	27.24	19.07
tai150b	498896643	612722637	22.81	24.67
tai256c	44759294	49845632	11.36	36.47
tho40	240516	253076	5.22	10.42
tho150	8133398	9348502	14.93	24.76
wil50	48816	50926	4.32	11.91
wil100	273038	290268	6.31	18.41

Tabla 6.1: Datos obtenidos para el algoritmo Genetico1

Caso	MejorCoste	Coste obtenido	Desviación	tiempo
chr22a	6156	6878	11.72	6.31
chr22b	6194	6918	11.68	6.39
chr25a	3796	5496	44.78	7.07
esc128	64	170	165.62	24.22
had20	6922	7106	2.65	6.02
lipa60b	2520135	3086900	22.48	16.29
lipa80b	7763962	9785619	26.03	19.14
nug28	5166	5258	1.78	8.58
sko81	90998	98592	8.34	18.71
sko90	115534	127210	10.1	20.91
sko100a	152002	168030	10.54	22.55
sko100f	149036	164018	10.05	22.37
tai100a	21052466	23202536	10.21	23.31
tai100b	1185996137	1467627923	23.74	23.56
tai150b	498896643	606998457	21.66	29.46
tai256c	44759294	49708242	11.05	48.5
tho40	240516	247892	3.06	12.22
tho150	8133398	9415020	15.75	29.9
wil50	48816	50092	2.61	13.54
wil100	273038	288504	5.66	21.87

Tabla 6.2: Datos obtenidos para el algoritmo Genetico1PMX

Caso	MejorCoste	Coste obtenido	Desviación	tiempo
chr22a	6156	7126	15.75	4.69
chr22b	6194	6854	10.65	4.69
chr25a	3796	6596	73.76	5.64
esc128	64	122	90.62	100.88
had20	6922	6944	0.31	4.19
lipa60b	2520135	3021338	19.88	26.96
lipa80b	7763962	9733433	25.36	46.99
nug28	5166	5344	3.44	6.74
sko81	90998	98816	8.59	44.99
sko90	115534	126774	9.72	54.97
sko100a	152002	167330	10.08	67.25
sko100f	149036	163698	9.83	68.22
tai100a	21052466	23058986	9.53	77.37
tai100b	1185996137	1406411030	18.58	71.58
tai150b	498896643	604787441	21.22	154.37
tai256c	44759294	48899728	9.25	437.79
tho40	240516	250114	3.99	12.52
tho150	8133398	9341026	14.84	158.34
wil50	48816	49170	0.72	17.91
wil100	273038	286802	5.04	66.38

Tabla 6.3: Datos obtenidos para el algoritmo Genetico2

Caso	MejorCoste	Coste obtenido	Desviación	tiempo
chr22a	6156	7072	14.87	4.72
chr22b	6194	6840	10.42	4.63
chr25a	3796	5602	47.57	5.51
esc128	64	74	15.62	98.23
had20	6922	6922	0.0	4.06
lipa60b	2520135	3023777	19.98	26.9
lipa80b	7763962	9534143	22.79	45.89
nug28	5166	5306	2.71	6.82
sko81	90998	94872	4.25	44.22
sko90	115534	121362	5.04	53.62
sko100a	152002	160428	5.54	66.14
sko100f	149036	157454	5.64	66.18
tai100a	21052466	22523706	6.98	70.44
tai100b	1185996137	1290564064	8.81	69.53
tai150b	498896643	582733862	16.8	153.33
tai256c	44759294	48057102	7.36	440.28
tho40	240516	254506	5.81	12.17
tho150	8133398	9158476	12.6	156.49
wil50	48816	49998	2.42	17.5
wil100	273038	282292	3.38	65.5

Tabla 6.4: Datos obtenidos para el algoritmo Genetico2PMX

Tras considerar los estadísticos colocados más abajo, se ha decidido mejorar con algoritmos meméticos el algoritmo estacionario con cruce PMX, teniendo una desviación de 10,93 y tardando de media 70,61 segundos.

Caso	MejorCoste	Coste obtenido	Desviación	tiempo
chr22a	6156	7140	15.98	4.55
chr22b	6194	7278	17.5	4.51
chr25a	3796	6624	74.49	5.23
esc128	64	198	209.37	27.8
had20	6922	6938	0.23	4.24
lipa60b	2520135	3092355	22.7	14.38
lipa80b	7763962	9748520	25.56	19.06
nug28	5166	5338	3.32	6.21
sko81	90998	101284	11.3	19.04
sko90	115534	130340	12.81	21.73
sko100a	152002	171506	12.83	23.64
sko100f	149036	167510	12.39	25.11
tai100a	21052466	23055766	9.51	24.95
tai100b	1185996137	1519526483	28.12	25.1
tai150b	498896643	610497203	22.36	40.26
tai256c	44759294	51140576	14.25	63.47
tho40	240516	270292	12.38	9.4
tho150	8133398	9523768	17.09	36.29
wil50	48816	51322	5.13	11.91
wil100	273038	292406	7.09	24.88

Tabla 6.5: Datos obtenidos para el algoritmo Genetico2PMXM1

Caso	MejorCoste	Coste obtenido	Desviación	tiempo
chr22a	6156	6802	10.49	4.66
chr22b	6194	6682	7.87	4.83
chr25a	3796	5416	42.67	5.43
esc128	64	144	125.0	28.73
had20	6922	6934	0.17	4.22
lipa60b	2520135	3082035	22.29	13.68
lipa80b	7763962	9636780	24.12	18.82
nug28	5166	5256	1.74	6.08
sko81	90998	100492	10.43	19.27
sko90	115534	128680	11.37	21.85
sko100a	152002	168432	10.8	23.89
sko100f	149036	166054	11.41	25.19
tai100a	21052466	22620144	7.44	24.9
tai100b	1185996137	1474626582	24.33	25.07
tai150b	498896643	582202312	16.69	39.89
tai256c	44759294	49895772	11.47	64.51
tho40	240516	259460	7.87	9.55
tho150	8133398	9388054	15.42	38.47
wil50	48816	50992	4.45	10.89
wil100	273038	288598	5.69	22.84

Tabla 6.6: Datos obtenidos para el algoritmo Genetico2PMXM2

Caso	MejorCoste	Coste obtenido	Desviación	tiempo
chr22a	6156	6612	7.4	4.62
chr22b	6194	6996	12.94	4.61
chr25a	3796	5740	51.21	5.17
esc128	64	134	109.37	26.8
had20	6922	6944	0.31	3.97
lipa60b	2520135	3062380	21.51	13.05
lipa80b	7763962	9624352	23.96	17.7
nug28	5166	5262	1.85	5.87
sko81	90998	100662	10.62	17.43
sko90	115534	127262	10.15	19.25
sko100a	152002	170008	11.84	21.66
sko100f	149036	163574	9.75	21.82
tai100a	21052466	22663316	7.65	21.8
tai100b	1185996137	1489745022	25.61	21.56
tai150b	498896643	583253116	16.9	33.48
tai256c	44759294	49629034	10.87	61.21
tho40	240516	270648	12.52	9.36
tho150	8133398	9293044	14.25	36.09
wil50	48816	50820	4.1	10.82
wil100	273038	289232	5.93	22.4

Tabla 6.7: Datos obtenidos para el algoritmo Genetico2PMXM3

Por último, la tabla de comparación de algoritmos:

Algoritmo	Desv	Tiempo
Greedy	71.75	0,00
BL	8.64	11.08
Genetico1	24.98	15.9
Genetico1PMX	20.98	19.05
Genetico2	18.06	71.62
Genetico2PMX	10.93	70.61
Genetico2PMX Mejora 1	26.72	20.59
Genetico2PMX Mejora 2	18.59	20.64
Genetico2PMX Mejora 3	18.44	18.93

Tabla 6.8: Tabla de comparación

6.1. Análisis de resultados

La primera consideración que hacemos es la comparativa entre los dos cruces que hemos implementado. El primer cruce tiene peores resultados que el PMX. Esto puede darse por la aleatoriedad en el cruce subyacente, que no aprovecha toda la información que podría utilizar de la permutación.

Según los estadísticos que hemos recogido queda claro que ningún algoritmo aquí implementado es mejor que la búsqueda local de la práctica anterior. Sin embargo hay algunos casos en los que los algoritmos si se comportan mejor. Un ejemplo de ello es el caso de $chr22a.dat$. Veamos cómo evoluciona en cada generación de cada algoritmo la mejor función de coste.

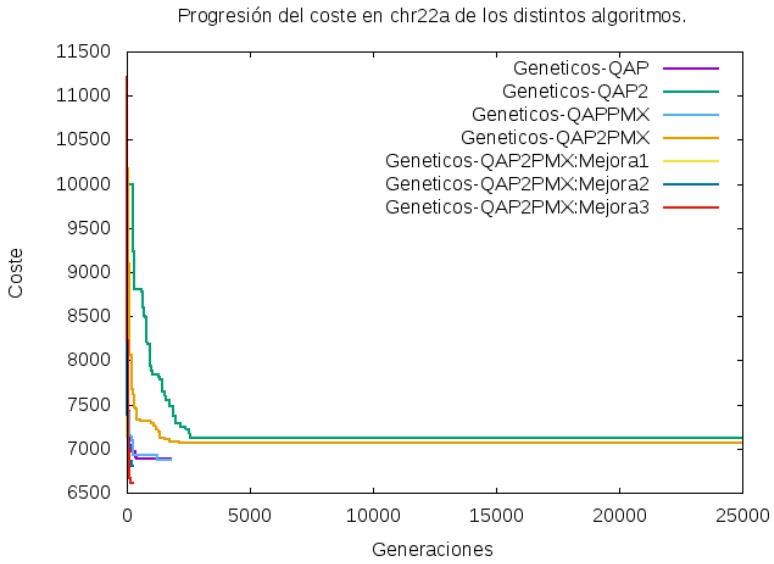

Figura 6.1: Comparativa chr22a

En este gráfico se ve claramente que se han obtenido muchas generaciones en los algoritmos genéticos estacionarios mientras que tanto los meméticos como los generacionales han utilizado menos generaciones. Sin embargo, ambos han conseguido rebajar mucho más rápido la función de coste, incluso han superado el mínimo local en el que se han quedado estancados los algoritmos estacionarios.

Esto no ocurre en otras instancias, como por ejemplo en *sko*81:

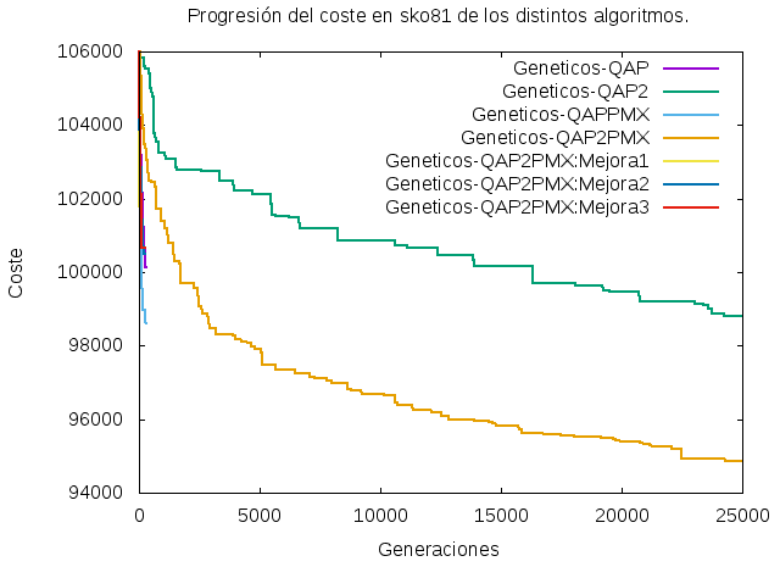

Figura 6.2: Comparativa sko81

Este gráfico nos muestra que, con las iteraciones máximas que le hemos puesto, los algoritmos tanto meméticos como generacionales no han tenido tiempo de dar una solución de calidad. Sin embargo, los algoritmos estacionarios si han podido ir mejorando su coste de forma progresiva. En los siguientes dos casos se tienen comportamientos similares, solo que en la pocas iteraciones si que se ha conseguido un coste parecido:

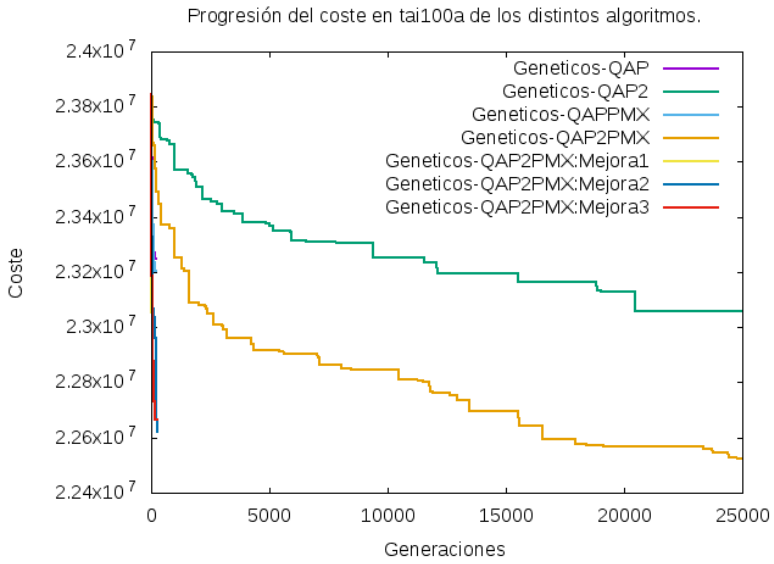

Figura 6.3: Comparativa tai100a

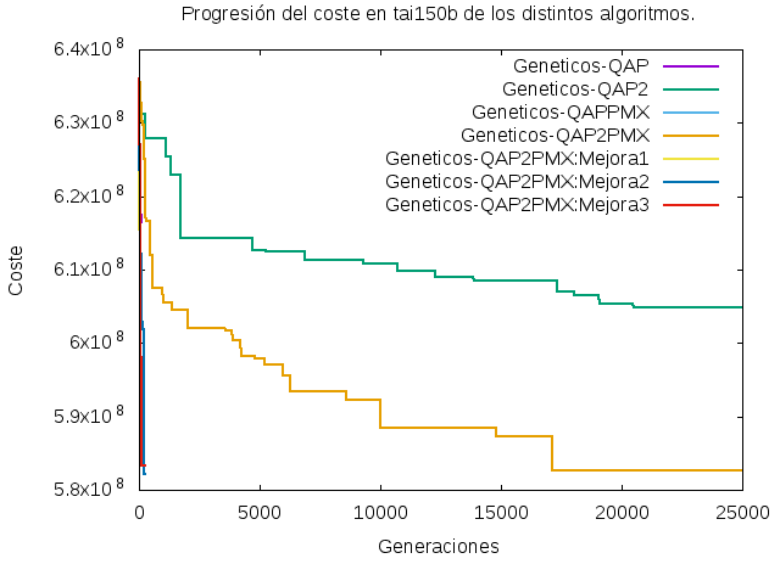

Figura 6.4: Comparativa tai150b

Por último, en la instancia más grande(tai256c) no se consigue apenas una mejora consistente, volvemos al caso de sko81:

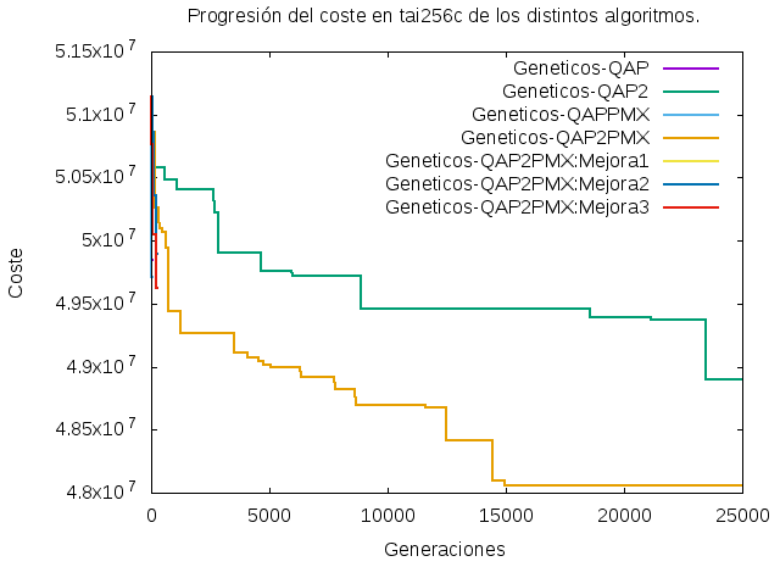

Figura 6.5: Comparativa 256c

6.2. Análisis de AG generacionales.

En las gráficas antes vistas se puede observar fácilmente el comportamiento de los algoritmos estacionarios ya que tienen muchas generaciones. Veamos ahora el comportamiento en concreto de los algoritmos generacionales y meméticos:

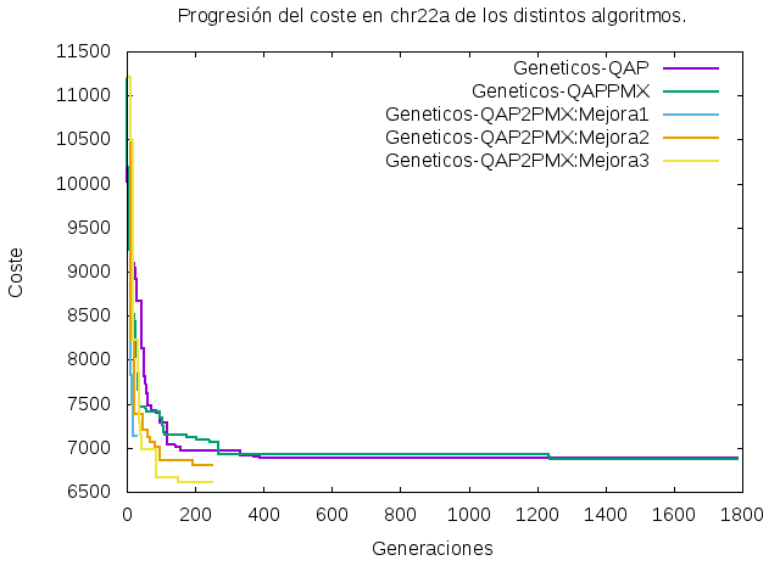

Figura 6.6: Comparativa generacional vs meméticos estacionarios chr22a

Lo primero que notamos en esta comparativa es que, aun evaluando muchísimas veces la función objetivo en cada generación de los algoritmos genéticos generacionales, en un problema de tamaño pequeño los algoritmos meméticos utilizan muy pocas generaciones en comparación. Esto se debe a la cantidad de mutaciones que hacemos. Veamos que ocurre en las siguientes instancias:

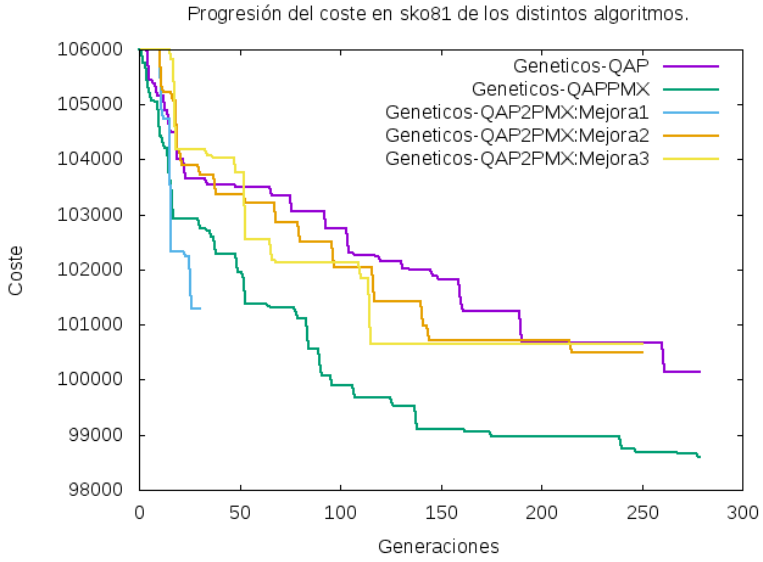

Figura 6.7: Comparativa generacional vs meméticos estacionarios sko81

En esta instancia el número de generaciones se igualan ya que se evalúa más veces la función objetivo gracias a las mutaciones. También se puede observar que la evolución de los algoritmos meméticos se produce a saltos. Otro dato importante a tener en cuenta es que, pese a tener mejor estadístico la tercera mejora, en esta instancia la segunda mejora lo supera. La primera mejora tiene muchas menos iteraciones ya que utiliza la mayoría de las evaluaciones de la función coste en búsquedas locales.

En los siguientes gráficos se ve cómo decrece el número de generaciones de los algoritmos genéticos generacionales mientras que los meméticos estacionarios mantienen sus generaciones fijas:

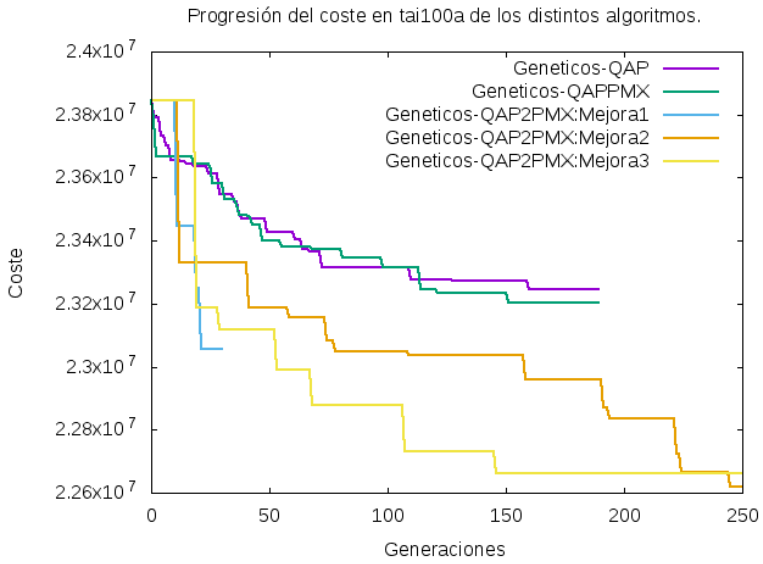

Figura 6.8: Comparativa generacional vs meméticos estacionarios tai100a

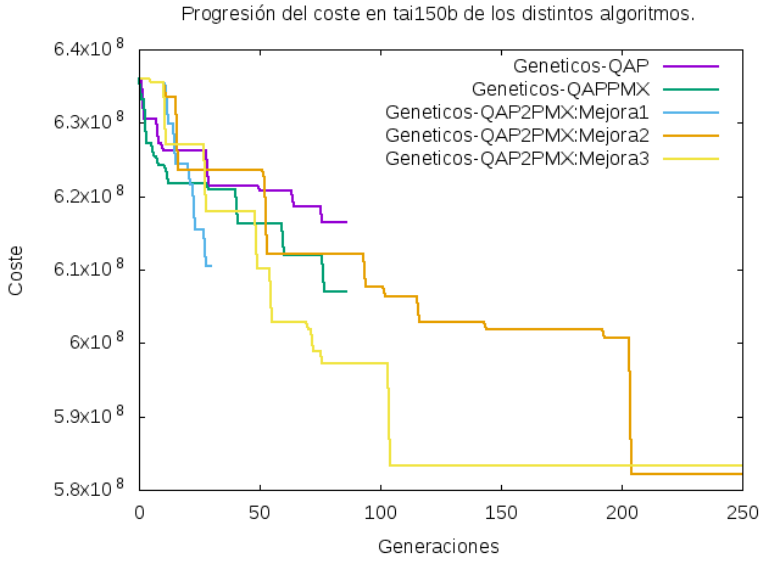

Figura 6.9: Comparativa generacional vs meméticos estacionarios tai150b

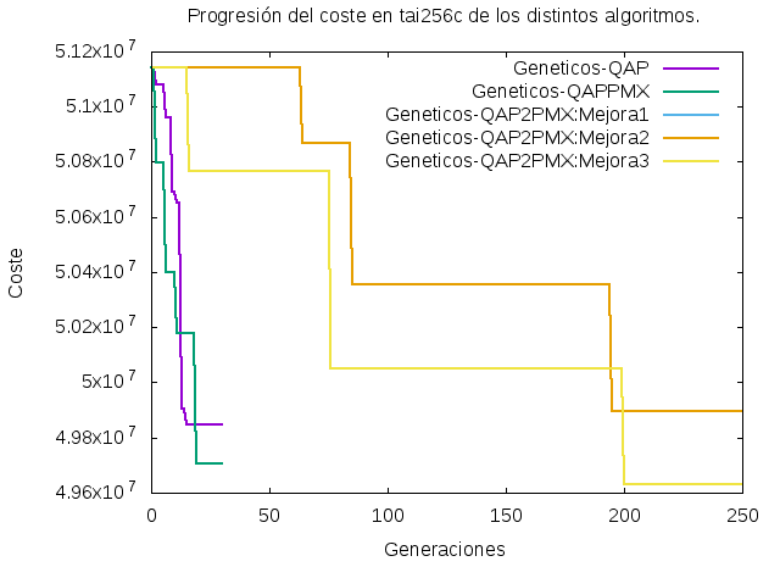

Figura 6.10: Comparativa generacional vs meméticos estacionarios tai256c

6.3. Posibles mejoras de los algoritmos

Como ya se dijo en el apartado de consideraciones de parada, hay varias ideas en los algoritmos que se deben pulir:

- **Tiempo de ejecución.** Las mutaciones, aunque sean evaluaciones de la función objetivo, tardan mucho menos en ser evaluadas. Se debería poder evaluar muchas veces tales mutaciones sin que ello repercutiese demasiado en el número de evaluaciones totales.
- **Evaluaciones por BL**. En los algoritmos meméticos la búsqueda local solo dispone de 400 evaluaciones. Si consideramos los primeros 400 genes, nunca se podrá mejorar los últimos. Para poder solventar esto, se podría empezar a profundizar en cada búsqueda local en un gen distinto o incluso aleatorizar la selección del gen a investigar. Otra forma de poder mejorar este aspecto es que el número de evaluaciones de la búsqueda local asegurase que intenta cambiar y mejorar al menos una vez cada gen. Es decir, que en vez de tener un número fijo de evaluaciones(400), este número dependa del número de genes(posibles trasposiciones (i, j)).
- **Parámetros modificables.** Dependiendo del tamaño del problema podríamos modificar cada cuanto se debe realizar una búsqueda local en los algoritmos meméticos, ya que estos consumen muchas evaluaciones y no dejan al algoritmo genético evolucionar para producir diversidad. De hecho, la última mejora se podría considerar que se comporta casi como una búsqueda local multiarranque en algunos casos(aplicando siempre la búsqueda local a los mejores).