Metaheurística
Doble Grado en Ingeniería Informática y Matemáticas
Universidad de Granada

Práctica 1

Algorítmos basados en trayectorias 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

5 de junio 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 basados en trayectorias.

Esta práctica consistirá en desarrollar cuatro algoritmos basado en trayectorias y un algoritmo híbrido entre dos de ellos. Veamos ahora las descripciones de cada uno de ellos:

2.3.1. Búsqueda multiarranque básico.

Este algorítmo consiste en utilizar la ya conocida búsqueda local a partir de varios puntos o soluciones de arranque. Puesto que condición de parada son las mismas evaluaciones, se realizarán menos evaluaciones en cada una de las búsquedas locales del algoritmo.

2.3.2. Enfriamiento simulado(ES).

Este algoritmo se basa en cómo se comportan una partícula dentro de un metal al enfriarse el mismo. Al principio, esta partícula tiene mucha energía por la alta temperatura, por lo que se mueve mucho aunque vaya a posiciones con más .^{en}ergía". Conforme se va enfriando el metal, lo mismo le pasa a la partícula y tiene menos energía, por lo que los

saltos que hace se empiezan a parecer más a una búsqueda por descenso, intentando ir a posiciones con cada vez menos energía.

2.3.3. Búsqueda local iterativa(ILS).

En este algoritmo ejecutaremos de forma repetida una búsqueda local. Cada cierto tiempo, modificaremos una parte significativa de la solución generando una solución vecina más lejana a la vecina de la búsqueda local, y volveremos a aplicar sobre esta nueva solución la búsqueda local.

2.3.4. Búsqueda por procedimiento greedy aleatorizado(GRASP).

2.3.5. Algoritmo híbrido. ILS-ES

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 de cada algoritmo.

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:

Algoritmo BMB

Caso	MejorCoste	Coste obtenido	Desviación	tiempo
chr22a	6156	6724	9.22	2.12
chr22b	6194	6666	7.62	2.07
chr25a	3796	4378	15.33	3.37
esc128	64	114	78.12	31.93
had20	6922	6926	0.05	1.98
lipa60b	2520135	3041614	20.69	15.07
lipa80b	7763962	9544338	22.93	20.57
nug28	5166	5228	1.2	4.67
sko81	90998	98294	8.01	20.75
sko90	115534	125448	8.58	23.34
sko100a	152002	166146	9.3	25.46
sko100f	149036	162588	9.09	25.45
tai100a	21052466	22260648	5.73	25.76
tai100b	1185996137	1374946874	15.93	26.32
tai150b	498896643	594443569	19.15	39.57
tai256c	44759294	49791988	11.24	68.96
tho40	240516	250658	4.21	10.11
tho150	8133398	9149516	12.49	40.09
wil50	48816	49638	1.68	12.67
wil100	273038	287216	5.19	25.77

Tabla 6.1: Datos obtenidos para el algoritmo BMB

Algoritmo ES

Caso	MejorCoste	Coste obtenido	Desviación	tiempo
chr22a	6156	6924	12.47	4.68
chr22b	6194	6626	6.97	4.5
chr25a	3796	4998	31.66	5.12
esc128	64	66	3.12	43.27
had20	6922	6926	0.05	4.34
lipa60b	2520135	3037170	20.51	14.02
lipa80b	7763962	9445973	21.66	18.39
nug28	5166	5350	3.56	6.02
sko81	90998	92686	1.85	18.55
sko90	115534	117786	1.94	21.19
sko100a	152002	155366	2.21	23.98
sko100f	149036	151812	1.86	23.62
tai100a	21052466	21823886	3.66	24.73
tai100b	1185996137	1246330318	5.08	24.18
tai150b	498896643	509380051	2.1	37.15
tai256c	44759294	45007954	0.55	96.87
tho40	240516	246914	2.66	8.99
tho150	8133398	8282656	1.83	37.29
wil50	48816	49656	1.72	11.62
wil100	273038	275082	0.74	21.25

Tabla 6.2: Datos obtenidos para el algoritmo ES

Algoritmo ILS

Caso	MejorCoste	Coste obtenido	Desviación	tiempo
chr22a	6156	6428	4.41	1.57
chr22b	6194	6610	6.71	1.61
chr25a	3796	4374	15.22	2.69
esc128	64	104	62.5	26.58
had20	6922	6922	0.0	1.29
lipa60b	2520135	2685111	6.54	13.01
lipa80b	7763962	9531633	22.76	18.25
nug28	5166	5258	1.78	3.42
sko81	90998	96858	6.43	18.54
sko90	115534	123836	7.18	21.74
sko100a	152002	164162	7.99	22.61
sko100f	149036	160264	7.53	22.44
tai100a	21052466	22292828	5.89	22.47
tai100b	1185996137	1294015153	9.1	22.73
tai150b	498896643	574223346	15.09	36.0
tai256c	44759294	48720216	8.84	59.34
tho40	240516	248288	3.23	8.85
tho150	8133398	8994402	10.58	35.61
wil50	48816	49302	0.99	11.41
wil100	273038	284826	4.31	22.94

Tabla 6.3: Datos obtenidos para el algoritmo ILS

Algoritmo GRASP

Caso	MejorCoste	Coste obtenido	Desviación	tiempo
chr22a	6156	6980	13.38	2.39
chr22b	6194	7408	19.59	2.32
chr25a	3796	5334	40.51	3.62
esc128	64	152	137.5	308.05
had20	6922	7054	1.9	1.92
lipa60b	2520135	3056446	21.28	27.77
lipa80b	7763962	9609494	23.77	59.4
nug28	5166	5408	4.68	5.13
sko81	90998	99980	9.87	60.63
sko90	115534	126136	9.17	82.92
sko100a	152002	167186	9.98	116.93
sko100f	149036	162980	9.35	116.81
tai100a	21052466	22518484	6.96	123.11
tai100b	1185996137	1462137390	23.28	129.65
tai150b	498896643	599411314	20.14	535.07
tai256c	44759294	50395560	12.59	4053.33
tho40	240516	260566	8.33	11.73
tho150	8133398	9169366	12.73	526.79
wil50	48816	50354	3.15	17.47
wil100	273038	287476	5.28	114.94

Tabla 6.4: Datos obtenidos para el algoritmo GRASP

Algoritmo ILS-ES

Caso	MejorCoste	Coste obtenido	Desviación	tiempo
chr22a	6156	6456	4.87	5.32
chr22b	6194	6742	8.84	5.23
chr25a	3796	5238	37.98	5.84
esc128	64	98	53.12	59.34
had20	6922	6922	0.0	4.44
lipa60b	2520135	3043180	20.75	19.21
lipa80b	7763962	9547423	22.97	26.83
nug28	5166	5264	1.89	7.14
sko81	90998	94486	3.83	28.52
sko90	115534	120792	4.55	34.67
sko100a	152002	161208	6.05	32.19
sko100f	149036	157370	5.59	32.24
tai100a	21052466	22370186	6.25	33.07
tai100b	1185996137	1273972590	7.41	34.25
tai150b	498896643	544843315	9.2	75.7
tai256c	44759294	49180730	9.87	137.33
tho40	240516	251164	4.42	10.65
tho150	8133398	8707874	7.06	73.11
wil50	48816	49952	2.32	13.92
wil100	273038	280364	2.68	34.88

Tabla 6.5: Datos obtenidos para el algoritmo ILS-ES

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 6.6: 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.

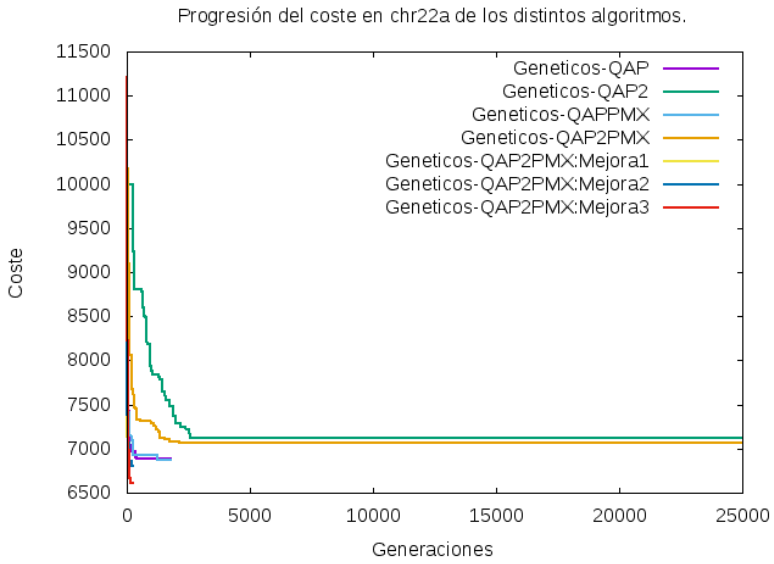

Figura 6.1: Comparativa chr22a

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:

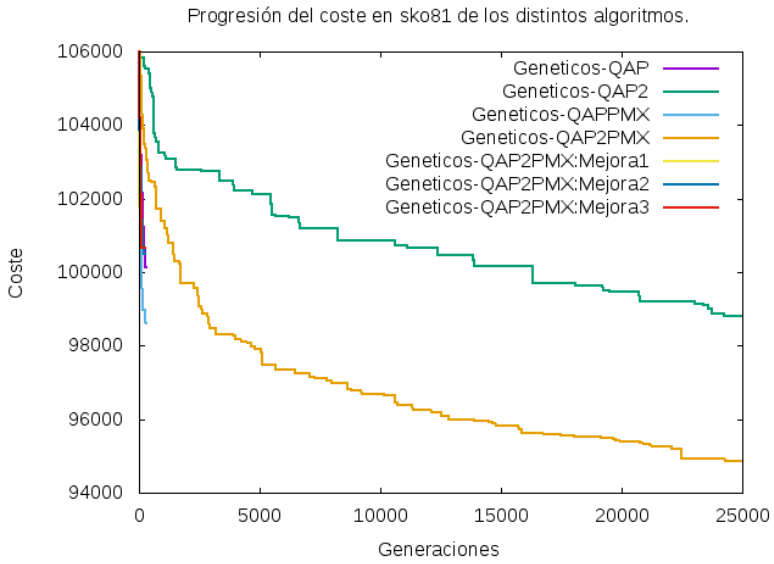

Figura 6.2: Comparativa sko81

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:

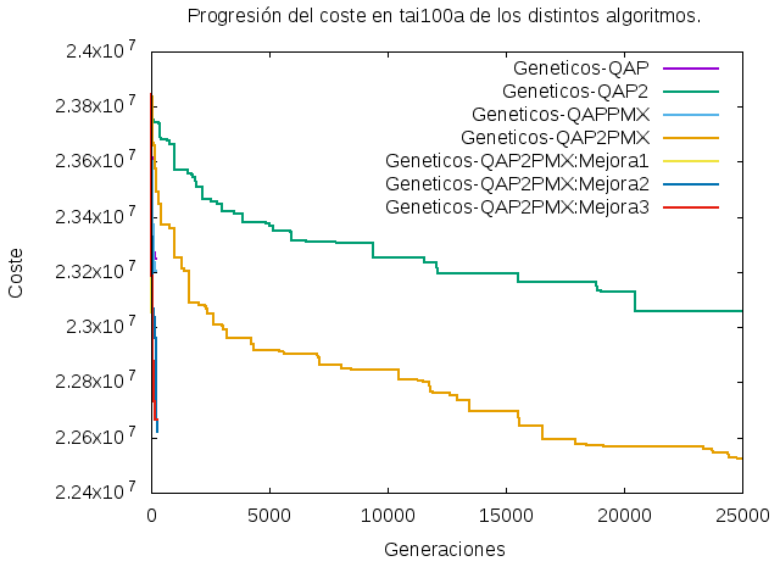

Figura 6.3: Comparativa tai100a

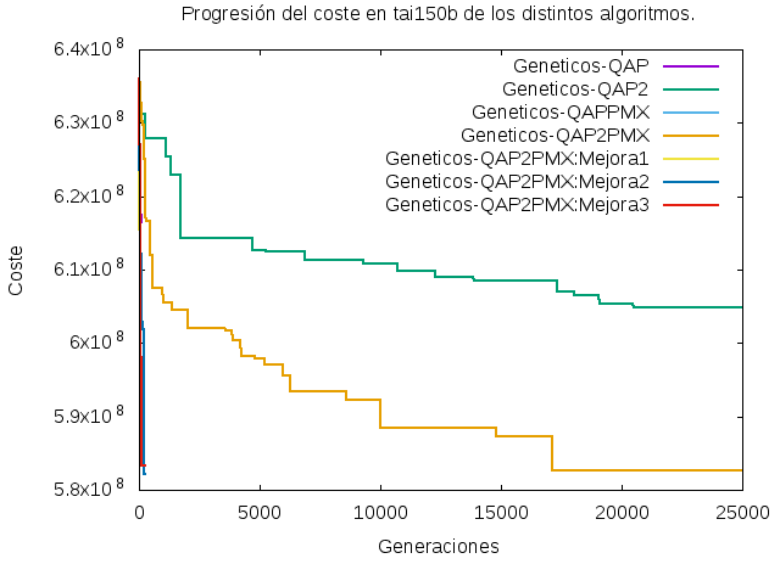

Figura 6.4: Comparativa tai150b

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

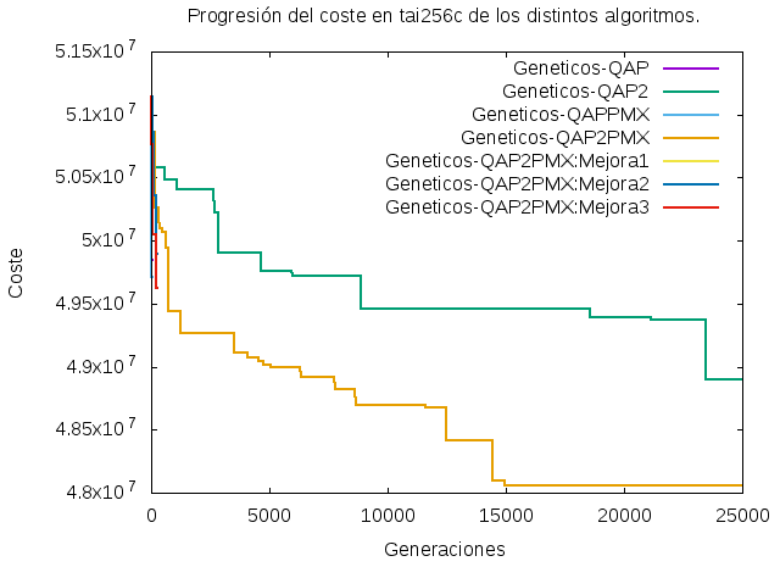

Figura 6.5: Comparativa 256c

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:

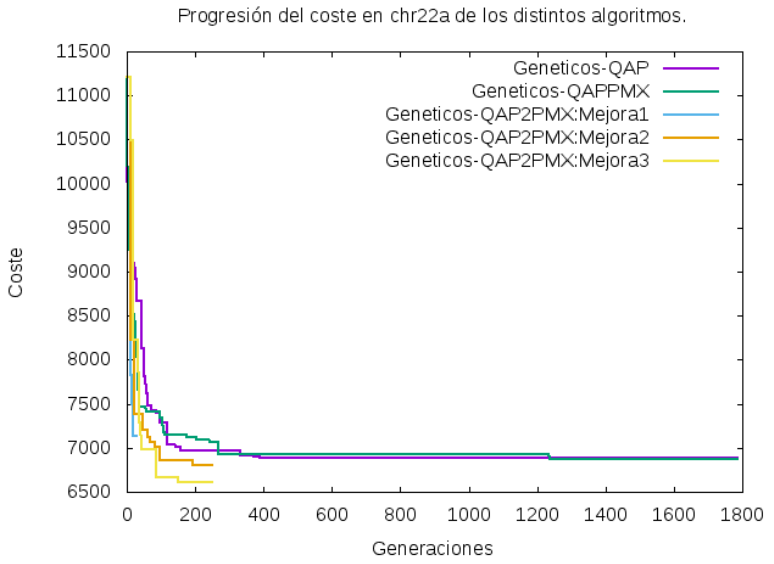

Figura 6.6: Comparativa generacional vs meméticos estacionarios chr22a

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:

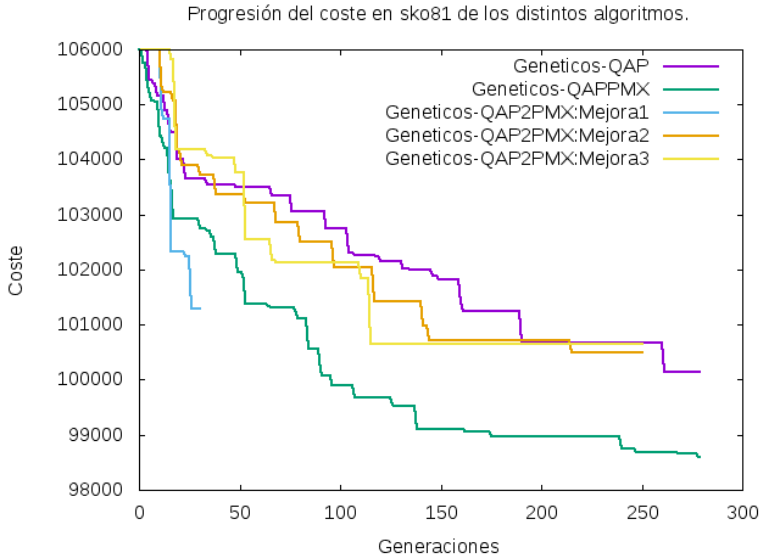

Figura 6.7: Comparativa generacional vs meméticos estacionarios sko81

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:

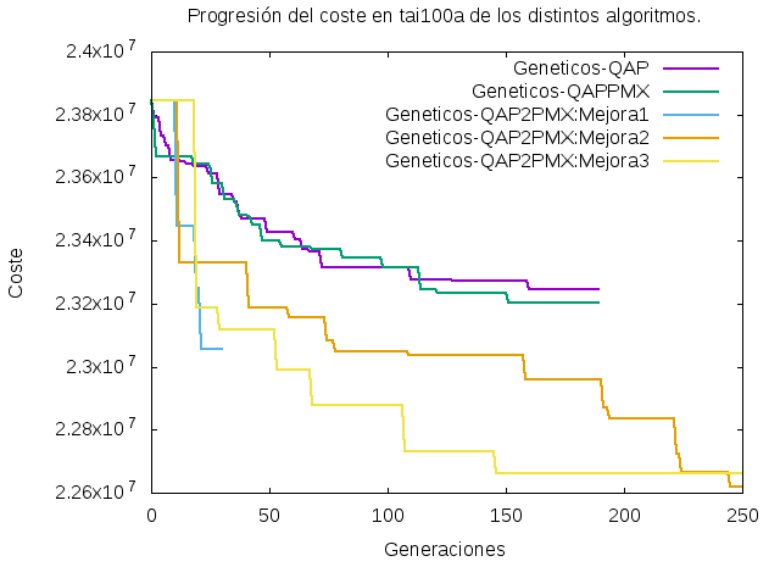

Figura 6.8: Comparativa generacional vs meméticos estacionarios tai100a

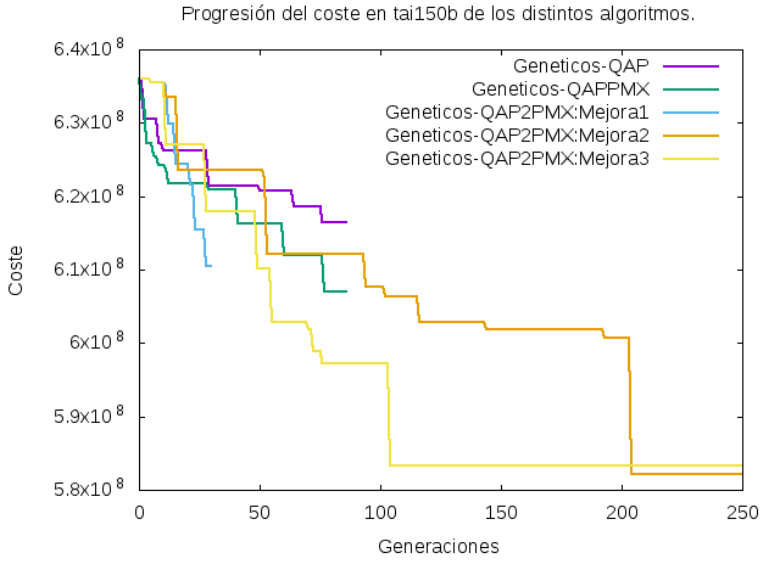

Figura 6.9: Comparativa generacional vs meméticos estacionarios tai150b

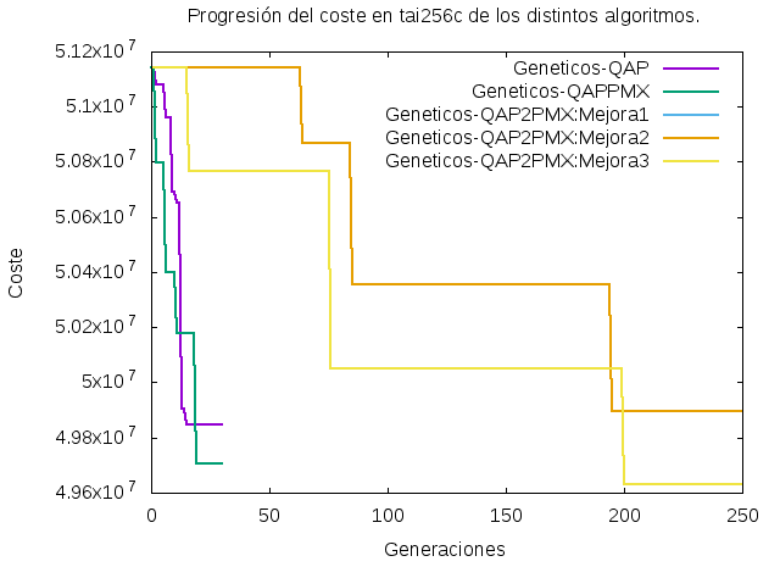

Figura 6.10: Comparativa generacional vs meméticos estacionarios tai256c

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).