Inteligencia Computacional

Práctica de algoritmos evolutivos.

Problemas de optimización combinatoria QAP.

Máster en Ingeniería Informática.

Curso 2019-2020.

Lidia Sánchez Mérida.

Índice

Índice de figuras

Índice de tablas

Problema a resolver: QAP

El problema de asignación cuadrática o QAP (Quadratic Assignment Problem) es un problema de optimización bastante conocido en el que cual se trata de asignar cada una de las instalaciones disponibles a una determinada localización. El objetivo es minimizar el coste de la asignación de modo que sea lo más liviano posible el flujo entre cada par de unidades.

Una solución para este problema se puede representar como una permutación en la que, para cada una de las unidades disponibles, se le asocia una localización no asignada. De este modo, se facilita bastante su representación puesto que se reduce a gestionar un vector, en este caso de enteros, en el que sus índices son las instalaciones a construir y los elementos del vector representan las localizaciones donde se situarán.

Para evaluar el coste de una solución se considera el flujo existente dadas dos unidades o instalaciones así como la distancia entre ambas. De este modo, en este problema se considera la siguiente fórmula, donde el coste se calcula como:

$$\sum_{i,j} w(i,j)\ d(p(i),p(j))$$

Figura 1. Fórmula para evaluar el coste de una solución.

Al ser un problema NP-completo no existe ningún modelo concreto capaz de calcular la solución óptima cuando el número de instalaciones a considerar es demasiado elevado. Por lo tanto, en este documento se desarrollarán y explicarán diversas variantes de algoritmos genéticos aplicados a los ficheros de datos concretos que se ofrecen en esta práctica.

Algoritmos genéticos

Los algoritmos genéticos son capaces de imitar el comportamiento reproductivo mediante la creación, cruce, mutación y reemplazamiento de un conjunto de individuos, los cuales, para este problema, representan una posible solución. Para cada una de estas operaciones existen diversas técnicas con las que se consiguen diferentes variantes de algoritmos genéticos, y por ende, distintos resultados. Es por ello por lo que en este documento se expondrán los detalles de las que se han utilizado para realizar esta práctica.

Generar la población inicial

La primera decisión que debemos tomar es el método con el que inicializamos la población, es decir, cómo se van a generar las distintas soluciones para comenzar la ejecución del algoritmo genético. En mi caso se han desarrollado dos variantes: **generar la población**

inicial aleatoriamente y generarla mediante un algoritmo Greedy constructivo. En el primer caso basta con inicializar tantas soluciones como individuos va a tener la población, generando de forma aleatoria y sin repetir las localizaciones para cada una de las instalaciones disponibles. Para ello haré uso de un generador de números aleatorios proporcionado en las prácticas de *Metaheurísticas* del Grado denominado *pseudoaleatorio.h*.

En el segundo caso se ha implementado una variante de los algoritmos voraces consistente en asignar a cada unidad la localización disponible **maximizando el flujo y minimizando la distancia**. Para ello será necesario disponer de dos vectores potenciales de flujo y distancia, respectivamente, en el que para cada unidad se recoja la sumatoria de sus flujos y distancias con respecto a las demás instalaciones. Si bien este algoritmo no suele proporcionar buenas soluciones, puede ser un buen punto de partida desde el que comenzar la ejecución de un algoritmo genético.

Operador de seleccion: torneo binario

A continuación debemos decidir el método por el cual seleccionamos las permutaciones que posteriormente cruzaremos para obtener nuevas soluciones. En mi caso he optado por la **selección por torneo binario**, con el cual se eligen dos individuos de la población al azar para competir por quién posee el mejor coste. Cabe destacar dos aspectos importantes: el primero consiste en asegurarse que ambos individuos son diferentes. El objetivo de ello es conseguir que la población de padres esté compuesta por las mejores soluciones de la población actual, de modo que aumentemos la posibilidades de generar hijos igual de óptimos o mejores.

El segundo aspecto consiste en **ordenar previamente la población en función del coste**, de modo que las soluciones con un menor coste se encuentren al principio de la población. Así, el procedimiento de la selección de padres se simplifica hasta el punto de devolver aquella permutación cuyo índice sea menor, puesto que será la que disponga de un menor coste y por tanto será más óptima que su respectivo rival.

Operadores de cruce

Los operadores de cruce permiten generar nuevas soluciones dadas dos permutaciones conocidas como *padres*. Estos, en mi caso, serán escogidos aplicando el procedimiento anterior basado en el torneo binario. Por norma general, se suelen obtener dos descendientes de cada dos padres aplicando, para ello, alguna de las diversas variantes de los operadores de cruce. En mi caso, he desarrollado dos métodos diferentes: **basado en posición y el operador de cruce OX**.

En el primer caso particular, la nueva solución que se genera **hereda los genes comunes a ambos padres** para mantener las asignaciones prometedoras, y posteriormente se completan el resto de unidades de forma aleatoria entre las localizaciones que aún queden disponibles. Mientras que en el operador OX se escoge una **subcadena del primer padre** para asignarla al hijo, manteniendo las mismas posiciones, y posteriormente se completa el descendiente utilizando al **segundo padre desde la última unidad asignada** hasta que todas

las unidades estén asociadas a una localización sin repetir. Para este operador en particular existen diversas formas de obtener dicha subsecuencia. En mi caso he calculado, en primer lugar, el elemento que ocupa la posición central del vector para que a partir de él, se asignen al hijo todos aquellos elementos que se encuentren a una distancia media del elemento central tanto a derecha como a izquierda. Es decir, si por ejemplo el conjunto de datos dispone de 12 instancias, el elemento central será el que ocupe la posición número 6. Por lo tanto, el descendiente heredará los genes comprendidos entre el tercer y el noveno elemento. De este modo, la subsecuencia se adaptará al tamaño de cada conjunto de datos, heredando más genes cuantas más instalaciones se consideren.

Operador de mutación

Con este operador se genera una nueva solución vecina dada una permutación a la que se le intercambian las posiciones de dos genes diferentes escogidos al azar. Generalmente, este operador se aplica cada cierto número de generaciones para introducir cierta diversidad en la población con el objetivo de evitar los óptimos locales.

Una vez han sido explicados los elementos comunes a cualquier variante de los algoritmos genéticos, procedo a detallar las distintas versiones que he considerado para la resolución del problema de optimización QAP.

Algoritmos Generacionales

Este primer modelo se caracteriza por un esquema de evolución basado en sustituir la población actual por la nueva generada, de modo que cada conjunto de individuos solo sobreviven durante una generación. Para ello, en primer lugar, se debe elegir un conjunto de **padres del tamaño de la población**. Es lógico que al utilizar el operador de selección por torneo binario muchos de estos padres sean soluciones repetidas puesto que, como se comentó anteriormente, en este procedimiento solo se escogen a los mejores.

Una vez escogidos los cromosomas, se procede a cruzarlos por parejas para generar **dos hijos** por cruce. En mi caso, he establecido una **probabilidad de cruce del 70%** por lo que solo el 70% de los padres seleccionados se cruzarán para generar dos hijos por pareja. No obstante, como el tamaño de la población debe permanecer el mismo, se **completa la nueva población añadiendo los padres no cruzados**. Este esquema de cruce es válido tanto para el operador basado en posición como para el OX.

A continuación se aplica el operador de mutación para intercambiar tantos genes como dicte su propia probabilidad así como el número de instalaciones del que disponemos y el tamaño de la población. En mi caso, he establecido una **probabilidad de mutación del 0.001% por gen,** por lo que para una población de 50 individuos y un fichero de hasta 256 datos se mutarían unos 12.8 genes.

Una vez generada la nueva población, esta sustituye directamente a la población anterior pasando a ser la población actual. Sin embargo, este esquema generacional suele estar caracterizado por el **elitismo**, consistente en asegurarse que siempre sobrevive la mejor

solución de la anterior población. Es por ello por lo que antes de sustituir la población anterior por la nueva generada, se comprueba si la mejor solución de la anterior se encuentra dentro de la nueva generación. Si no lo está entonces se introduce a cambio de la peor solución de la nueva población.

Algoritmos Estacionarios

Este modelo se diferencia del anterior en dos aspectos principales. En esta variante en particular solo se **seleccionan dos padres**, también mediante el torneo binario, **y siempre se cruzan**, por lo que la probabilidad de cruce para este modelo es del 100%.

La otra discrepancia con respecto al modelo anterior se centra en el esquema de reemplazamiento puesto que en este caso solo se incluirán los dos hijos generados si son mejores que las dos peores soluciones de la población actual. Este esquema tiene como principal ventaja el hecho de que al no ser tan agresivo, se conservan ciertas soluciones que en futuras iteraciones pueden llegar a convertirse en permutaciones bastante próximas a la óptima. Asimismo, al ser este procedimiento más sencillo, como veremos a lo largo de este documento, los tiempos invertidos en encontrar una solución serán menores que en su análogo generacional.

Variante *Baldwiniana*

La variante *baldwiniana* consiste en incluir una de las técnicas de optimización local mediante la implementación de un **algoritmo Greedy de transposición**, en concreto he utilizado el denominado **2-*opt***. Con él se realizará una búsqueda local para cada uno de los individuos de la población de forma que se explora su vecindario en busca de un óptimo local. Si su coste es menor que la mejor solución actual, se actualiza el mismo pero sin sustituir el cromosoma por el obtenido. De este modo, en esta variante solo se actualiza el *fitness* de los individuos de la población pero el material genético es el mismo.

Variante *Lamarckiana*

Este tipo de algoritmo incorpora, además del **algoritmo Greedy de transposición** explicado anteriormente, la capacidad de utilizar el aprendizaje obtenido mediante el **2-*opt*** para realizar los cruces y obtener la nueva población. Es por ello por lo que, además de actualizar el coste de la solución sobre la que se le aplica este tipo de búsqueda local, también se sustituye la solución obtenida con esta técnica por la original.

Experimentos con los algoritmos genéticos clásicos

En esta sección se estudiará el comportamiento de los dos algoritmos genéticos anteriores incluyendo los dos operadores de cruce, que hacen un total de cuatro modelos de

poblaciones. Si bien el comportamiento de estos algoritmos puede variar en función del conjunto de datos que se maneje, hay muchísimos ficheros de prueba y no da tiempo a ejecutarlos todos para cada una de las variantes. Por ello, he escogido los tres ficheros siguientes: ***lipa30a.dat***, por conocer el rendimiento de los cuatro algoritmos con un conjunto de datos pequeño, ***tai100a.dat*** para comprender su comportamiento con un conjunto de datos más grande, y por último, ***tai256c.dat*** que es el protagonista de esta práctica.

La evolución del coste de la mejor solución por generación se puede visualizar a continuación, en el que por ser el primer experimento, se ha utilizado el primero de los ficheros mencionados anteriormente. Para él se han realizado **ocho ejecuciones**, cuatro de ellas pertenecen a las dos variantes del algoritmo generacional con su cruce de posición y OX en las que para dos de esas cuatro se ha partido de una población inicial aleatoria y para las otras dos se ha generado mediante el algoritmo *Greedy constructivo*, explicado en secciones anteriores. La configuración de este experimento se basa en construir una **población de 50 individuos, respetando las probabilidades mencionadas anteriormente y con hasta 50.000 iteraciones estableciendo como semilla: 33**. Si bien no se incluye en la memoria, se ha llevado a cabo un procedimiento en el que se han ejecutado los algoritmos con un número de semillas diferente para realizar estos experimentos con aquella que aportase los mejores resultados.

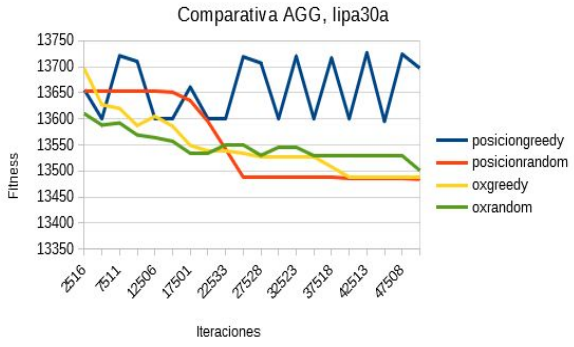

Figura 2. Comparativa AGG lipa30a.dat

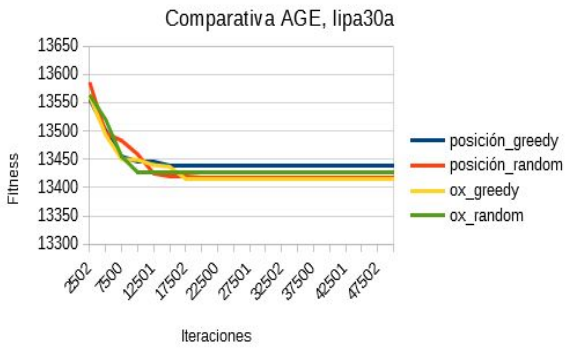

Figura 3. Comparativa AGE lipa30a.dat

En primer lugar, las diferencias entre las cuatro variantes de los dos esquemas de población son bastante visibles, puesto que mientras que en el algoritmo generacional se aprecia un comportamiento con una gran cantidad de variaciones en los costes de las mejores soluciones por generación, en el estacionario se observa la forma de una función decreciente que se estabiliza a partir de la iteración 20.000 aproximádamente. Esta característica es debida a los esquemas de reemplazamiento aplicados a cada variante, puesto que en la generacional se sustituye toda la población, conllevando el riesgo de eliminar soluciones que podrían ser cercanas a las óptimas y por ende, se producen muchos saltos en los costes de las mejores soluciones por generación. Sin embargo, en el estacionario, al solo reemplazar un máximo de dos individuos por población esta variación no se produce por lo que la convergencia de este algoritmo es más constante.

En relación a los operadores de cruce, podemos comprobar que por norma general las ejecuciones realizadas con **OX convergen a soluciones con costes similares**, tanto si la población ha sido inicializada con un algoritmo Greedy o de forma aleatoria. Sin embargo, en el operador basado en **posición sí se aprecia una diferencia de comportamiento si es utilizado en un esquema generacional o estacionario**, proporcionando una solución nada buena en el primer caso.

Por último, podemos comprobar que la mayor diferencia entre inicializar la población de forma aleatoria o con un algoritmo Greedy se encuentra en el esquema generacional, puesto que en el estacionario apenas influye este parámetro. Para estudiar más en profundidad este último aspecto, se adjunta a continuación la tabla con los costes de las soluciones finales de cada ejecución así como el tiempo invertido en encontrarlas.

	AGG Posición	**AGG Posición Greedy**	**AGG OX**	**AGG OX Greedy**
Coste	13484	13697	13501	13488
Tiempo	0.432581 s.	0.647814 s.	0.359198 s.	0.343146 s.

Tabla 1. Costes y tiempos Algoritmos Generacionales, lipa30a.dat

	AGE Posición	**AGE Posición Greedy**	**AGE OX**	**AGE OX Greedy**
Coste	13418	13438	13427	13416
Tiempo	0.342446 s.	0.341244 s.	0.336641 s.	0.353302 s.

Tabla 2. Costes y tiempos Algoritmos Estacionarios, lipa30a.dat,

Tal y como podemos comprobar, en relación a los tiempos de ejecución apenas hay diferencia entre las ocho ejecuciones, exceptuando el caso del algoritmo generacional con cruce posición inicializado mediante un algoritmo Greedy, puesto que invierte 20 segundos más que el resto. Esto es debido a que tanto el esquema generacional de esta variante como el propio operador de cruce consumen más tiempo que el resto de alternativas, tal y como se puede observar en las tablas. Si a ello le añadimos el coste computacional de generar una solución mediante el algoritmo Greedy, el tiempo aumenta aunque no es demasiado considerable.

Asimismo, en relación a los resultados inicializando la población de forma aleatoria o con el algoritmo Greedy constructivo, para este conjunto de datos no supone una gran diferencia de coste.

A continuación pasamos a estudiar el comportamiento de estos algoritmos con un fichero más grande: **tai100a.dat**. De nuevo, se repetirá el procedimiento anterior con la misma configuración descrita anteriormente, por lo que directamente se muestran las evoluciones de los costes de las mejores soluciones por generación en los ocho casos dispuestos.

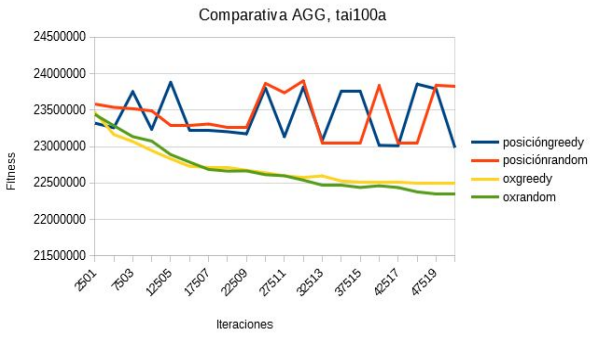

Figura 4. Comparativa AGG tai100a.dat

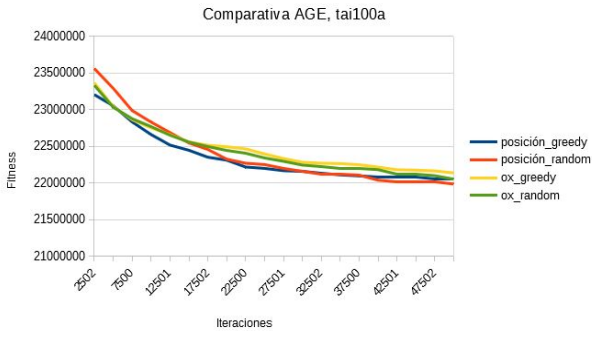

Figura 5. Comparativa AGE tai100a.dat

En primer lugar, destaca el cambio en el comportamiento del **algoritmo generacional**, puesto que con este conjunto de datos podemos visualizar el comportamiento general del operador de cruce basado en posición, con su tendencia a la gran variación de costes, como ocurría en el caso anterior. Sin embargo, el operador de cruce OX presenta una convergencia más constante, a diferencia del fichero anterior. Este hecho puede ser debido a que, como se comentó anteriormente, este operador de cruce escoge una subcadena central del primer padre para que la herede el descendiente. Como es dependiente del número de datos que manejemos, a mayor cantidad, mayor número de genes hereda el descendiente, por lo que estará más fuertemente influenciado por dicho padre. De ese modo, se preservan muchos de los genes del primer padre en el hijo y por tanto la solución obtenida tiene un coste similar al del padre.

En cuanto al **algoritmo estacionario** podemos comprobar que su tendencia es muy parecida al del fichero anterior, por lo que podemos concluir que esta variante es más fuerte que la generacional en cuanto a la influencia en el resultado del número de datos que gestione, es decir, su comportamiento es similar independientemente del número de instalaciones. A continuación mostramos las tablas con los resultados finales para este fichero así como los tiempos invertidos por las distintas variantes.

	AGG Posición	**AGG Posición Greedy**	**AGG OX**	**AGG OX Greedy**
Coste	23824804	22983802	22354666	22496836
Tiempo	4.66269 s.	4.77301 s.	2.9418 s.	2.97957 s.

Tabla 3. Costes y tiempos Algoritmos Generacionales, tai100a.dat

	AGE Posición	**AGE Posición Greedy**	**AGE OX**	**AGE OX Greedy**
Coste	21982898	22050168	22049352	22134928
Tiempo	2.99062	2.90978 s.	2.85993 s.	2.88221 s.

Tabla 4. Costes y tiempos Algoritmos Estacionarios, ta100a.dat

Tal y como podemos observar, los datos representados en las gráficas acompañan a los costes obtenidos en cada una de las ocho ejecuciones. Las asociadas al algoritmo estacionario son las que proporcionan mejores soluciones en cuanto a coste así como en tiempo, llegando a ser casi dos veces más rápidas que las del algoritmo generacional. Asimismo, si comparamos los costes de las ejecuciones que empezaron con una población aleatoria y las que comenzaron con una solución Greedy, de nuevo, en este fichero tampoco existen grandes diferencias pero por lo general utilizar una población aleatoria proporciona mejores resultados que utilizar un algoritmo Greedy constructivo.

Por último detallamos el tercer experimento de esta sección utilizando, para ello, el fichero de datos **tai256c.dat**. De nuevo, repetimos el mismo procedimiento anterior estableciendo los mismos parámetros de configuración explicados al comienzo. La evolución del *fitness* de las mejores soluciones por generación se pueden comprobar a continuación.

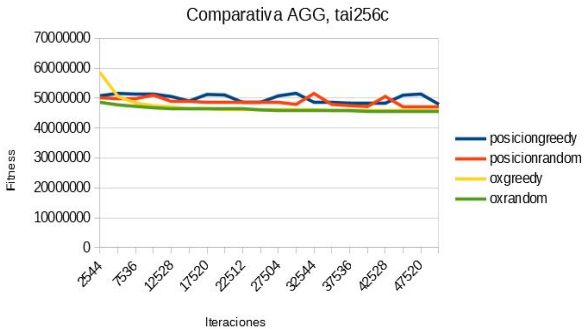

Figura 6. Comparativa AGG tai256c.dat

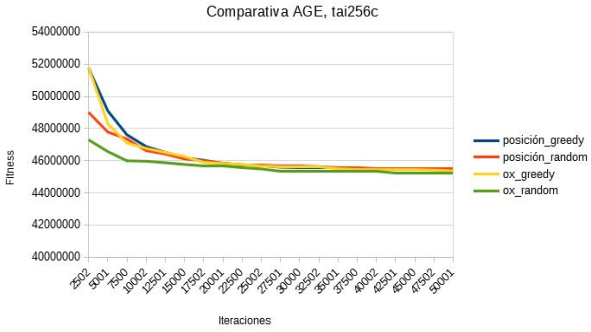

Figura 7. Comparativa AGE tai256c.dat

Con este conjunto de datos particular, podemos observar un drástico cambio en el comportamiento del **algoritmo generacional**, puesto que las grandes variaciones de costes que presentaba con los dos ficheros anteriores han mermado considerablemente. Es cierto que aún se pueden observar ciertos picos, pero nada comparado con la inestabilidad de los casos anteriores. Es por ello por lo que podemos concluir que este algoritmo es fácilmente influenciable por el conjunto de datos que gestione. Asimismo, podemos observar que su

convergencia es bastante lineal, lo cual indica que necesitará más iteraciones para encontrar una buena solución más próxima a la óptima.

En cuanto al **algoritmo estacionario** podemos observar que su comportamiento también es bastante similar al de los casos anteriores, por lo tanto se reafirma su fortaleza frente al conjunto de datos que gestione. A continuación observaremos con más detalle los costes de las soluciones finales y los tiempos de estas ocho nuevas ejecuciones.

	AGG Posición	**AGG Posición Greedy**	**AGG OX**	**AGG OX Greedy**
Coste	47052408	47919104	45626742	45971212
Tiempo	26.7845 s.	27.655 s.	18.1552 s.	18.1566 s.

Tabla 5. Costes y tiempos Algoritmos Generacionales, tai256c.dat

	AGE Posición	**AGE Posición Greedy**	**AGE OX**	**AGE OX Greedy**
Coste	45518210	45435230	45200118	45362718
Tiempo	21.7865 s.	22.9692 s.	17.7943 s.	18.0031 s.

Tabla 6. Costes y tiempos Algoritmos Estacionarios, tai256c.dat

Tal y como podemos observar, con un conjunto de datos más amplio comienzan a ser palpables las diferencias entre los dos operadores de cruce, proporcionando soluciones con mejor coste el OX frente al basado en posición. Esto nos indica que el hecho de que el descendiente herede una mayor cantidad de genes es sumamente beneficioso, así como el hecho de utilizar al segundo padre para terminar de completarlo en lugar de hacerlo de forma aleatoria como en el operador de cruce basado en posición. Asimismo, también dispone de otra ventaja puesto que como podemos contemplar, invierte un cantidad menor de tiempo en converger hacia una solución próxima al óptimo, para este conjunto de datos en particular.

En estos experimentos también podemos comprobar que el uso de un algoritmo Greedy para inicializar la solución suele provocar como resultado una solución con un coste más elevado que si iniciamos la población de forma aleatoria. Sin embargo, tras realizar otro diverso número de experimentos **variando el tamaño de la población**, he podido llegar a la conclusión de que si inicializamos la población con este algoritmo, serán necesarios **un mayor número de iteraciones** para obtener una buena solución. Asimismo, si reducimos la población, por ejemplo a 20 individuos y aumentamos el número de iteraciones, los costes de las soluciones proporcionadas son menores que si aumentamos el tamaño de la población, por ejemplo a 100. La conclusión a la que he llegado con estos otros

experimentos, es que **cuanto menor es el tamaño de la población, antes converge el algoritmo** puesto que más rápido pasan a formar parte los mejores individuos. De ese modo, es más rápido obtener una solución más cercana a la óptima, y de hecho, ha sido a través de esta nueva configuración con 20 individuos y 200.000 iteraciones como he obtenido el coste más cercano al óptimo de todos los algoritmos genéticos: **44980020** con un algoritmo estacionario y operador de cruce OX.

Experimentos con las variantes *Baldwiniana* y *Lamarckiana*

Si bien ambas variantes son fáciles de implementar, son **muy ineficientes en conjuntos de datos grandes, como *tai256c***. Es por ello por lo que, para realizar un estudio acerca de la evolución del coste de las soluciones he tenido que modificar la configuración que vengo utilizando hasta el momento. En este caso, para la variante ***Baldwiniana*** se genera una población de **20 individuos** y en lugar de aplicar el algoritmo *2-opt* en todas las generaciones, lo he aplicado **cada 10 generaciones** pues tras probar distintos valores para estos parámetros, este ha sido el número de generaciones que mejor calidad-tiempo proporciona. Asimismo, cabe destacar que tras varios experimentos el **número óptimo de iteraciones de este algoritmo es 10**, por tanto la búsqueda local finalizará si se han realizado 10 iteraciones o si no se obtiene una solución mejor explorando el vecindario.

Para el caso de la variante ***Lamarckiana***, se han reducido el número de **iteraciones** del algoritmo genético hibridado a **1.000** puesto que para ficheros de mayor tamaño, como el *tai256c*, es el número de iteraciones óptimo en relación a la calidad de la solución final obtenida y el tiempo de ejecución.

A continuación se presentan los resultados obtenidos tras incluir estas variantes en los dos tipos de algoritmos genéticos, cada uno, además, con los dos operadores de cruce disponibles. Del mismo modo, también se realizarán tres experimentos para comprobar su comportamiento con ficheros con distinto número de datos.

Variante Baldwiniana

Comenzamos esta sección mostrando las gráficas asociadas a la evolución de las mejores soluciones de cada generación aplicado al conjunto de datos del fichero ***lipa30a.dat***.

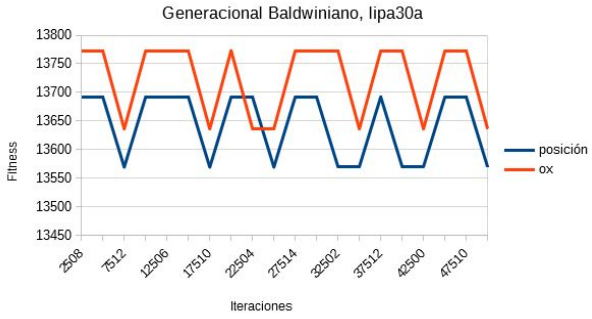

Figura 8. Comparativa Generacional Baldwiniano utilizando lipa30a.dat

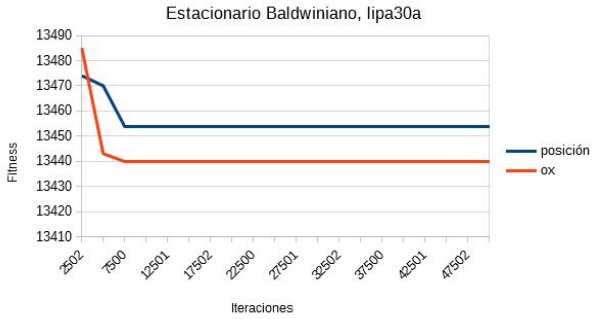

Figura 9. Comparativa Estacionaria Baldwiniano utilizando lipa30a.dat.

Tal y como podemos observar, la variante **generacional** presenta mucha variación en relación a los costes de las mejores soluciones por generación, tanto en el operador de cruce basado en posición como el OX. Si bien es una cualidad característica de esta variante, se acentúa aún más al incluir este tipo de búsqueda local. Mientras que en la variante **estacionaria** podemos visualizar un comportamiento propio de este tipo de algoritmo genético cuya convergencia es más pausada y constante. Sin embargo, si observamos la gráfica, podemos contemplar que cae en un óptimo local sobre la iteración 10.000, lo cual es sumamente negativo puesto que el algoritmo converge hacia una solución demasiado pronto. A continuación se muestra la tabla que resume los costes resultantes así como los tiempos invertidos.

	AGG Posición	**AGG OX**	**AGE Posición**	**AGE OX**
Coste	13569	13636	13454	13440
Tiempo	5.70666 s.	1.02212 s.	0.086357 s.	0.08628 s.

Tabla 7. Costes y tiempos Algoritmos Genéticos Baldwinianos, lipa30a.dat

Tal y como se puede observar, la tendencia es similar a los resultados obtenidos con los algoritmos genéticos originales. El esquema generacional es el que obtiene unas soluciones con coste mayor, mientras que por el contrario el estacionario es el que consigue reducir el coste de las soluciones finales. Si bien los resultados de la variante *baldwiniana* son peores que los de los algoritmos genéticos originales, las diferencias no son demasiado significativas. Sin embargo, los tiempos sí que son sumamente mayores a los de los algoritmos genéticos por lo que hasta el momento, para este conjunto de datos, los algoritmos genéticos originales siguen siendo los más adecuados para resolver este problema dado este fichero en concreto.

En segundo lugar procedo a repetir el mismo procedimiento anterior aplicado al fichero ***tai100a*** para comprobar el rendimiento de esta variante con un fichero de datos de mayor

tamaño. A continuación se presentan las gráficas que representan la evolución del *fitness* de las mejores soluciones por generación.

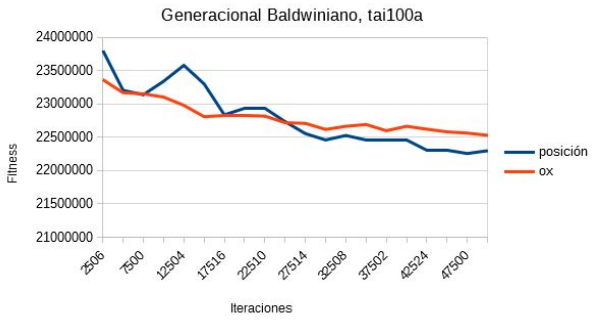

Figura 10. Comparativa Generacional Baldwiniano utilizando tai100a.dat

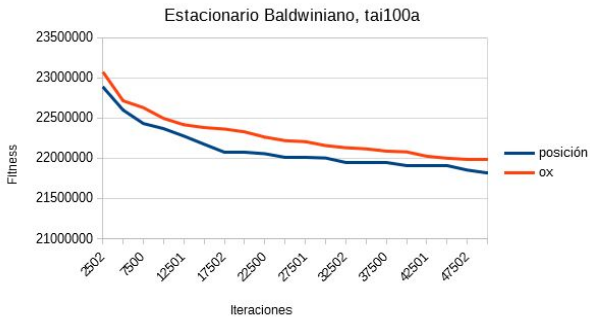

Figura 11. Comparativa Estacionaria Baldwiniano utilizando tai100a.dat.

Para este conjunto de datos, las variaciones que se mostraban con el anterior han quedado bastante reducidas. Por lo tanto, podemos concluir que esta variante es bastante influenciable por el conjunto de datos que gestione, modificando su comportamiento hasta conseguir una convergencia más constante y suave. Estas cualidades se acentúan aún más en el **estacionario** puesto que realza su naturaleza y por lo tanto, presenta una gráfica bastante más parecida al algoritmo estacionario original. Otro aspecto interesante a destacar, consiste en la **mejoría del operador de cruce basado en posición** frente al empeoramiento de los resultados proporcionados por el operador OX. En esta variante, queda claro que el gran beneficiado es el primero de ellos puesto le aporta una mayor estabilidad y por ello es capaz de proporcionar mejores resultados. A continuación se muestra la tabla correspondiente a los costes de las soluciones resultantes y los tiempos invertidos en obtenerlas.

	AGG Posición	**AGG OX**	**AGE Posición**	**AGE OX**
Coste	22297550	22527820	21815750	21990592
Tiempo	187.922 s.	89.9524 s.	1.28501 s.	1.35639 s.

Tabla 8. Costes y tiempos Algoritmos Genéticos Baldwinianos, tai100a.dat

Como podemos apreciar los costes de las distintas variantes son bastante similares entre sí y también comparados con los resultados obtenidos con los algoritmos genéticos originales. De hecho, en algunos de ellos se mejora. Sin embargo, las grandes diferencias se presentan en los tiempos de los **generacionales** que llegan a ser hasta 100 veces más costosos computacionalmente, mientras que las variantes **estacionarias** se presentan más asequibles.

Por último analizaremos el conjunto de datos del fichero ***tai256c.dat*** aplicando el mismo procedimiento que en los dos casos anteriores. Por ello, a continuación se muestran las gráficas pertenecientes a los dos algoritmos genéticos a los que se les ha incluido esta variante en particular, para comprobar la evolución del coste de sus soluciones durante cada generación.

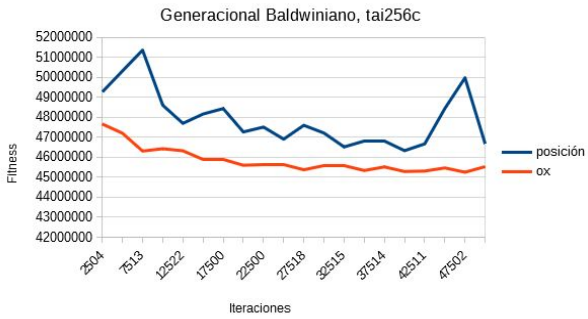

Figura 12. Comparativa Generacional Baldwiniano utilizando tai100a.dat

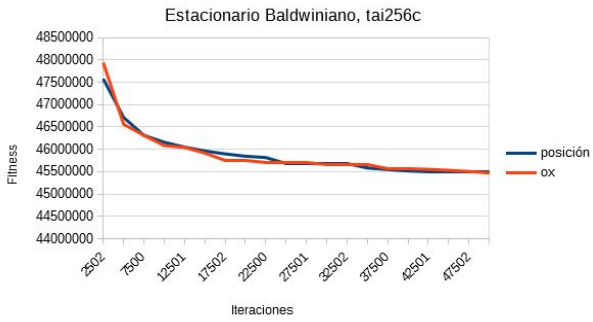

Figura 13. Comparativa Estacionaria Baldwiniano utilizando tai100a.dat.

Tal y como se puede observar en el caso del algoritmo **generacional** el operador OX vuelve a proporcionar el mejor resultado, comparado con el operador de cruce basado en Posición. Asimismo, este presenta una función con grandes variaciones mientras que el operador de cruce OX dispone de una función decreciente más constante. Ambos son comportamientos típicos que anteriormente hemos observado, especialmente en ficheros con conjuntos de datos numerosos.

Por otro lado la variante **estacionaria** presenta una gran similitud entre la evolución de los costes de las soluciones para ambos operadores de cruce, lo cual también lo hemos podido observar anteriormente en los algoritmos genéticos originales, en particular aplicados a este mismo fichero. A continuación se visualiza la tabla con los costes exactos y los tiempos de las cuatro ejecuciones.

	AGG Posición	**AGG OX**	**AGE Posición**	**AGE OX**
Coste	46668972	45526104	45490690	45466494
Tiempo	7079.25 s.	24970.1 s.	20.1916 s.	17.7836 s.

Tabla 9. Costes y tiempos Algoritmos Genéticos Baldwinianos, tai256c.dat

Como se puede observar, las dos ejecuciones del algoritmo generacional son las que más tiempo invierten en conseguir sus soluciones, las cuales, además, tienen un coste mayor que la variante estacionaria. Este comportamiento, de nuevo, se ha podido comprobar en las variantes genéticas originales en tanto en cuanto la variante generacional suele tardar más y proporcionar soluciones peores, solo que en el caso de esta variante la variable del

tiempo se acentúa mucho más debido a la inclusión de una búsqueda local para cada uno de los miembros de la población.

No obstante, en relación a los costes proporcionados por esta variante y los obtenidos con los genéticos tradicionales, podemos observar que no existen diferencias significativas como para utilizar estos algoritmos en lugar de los originales, los cuales disponen de una gran ventaja por ser más eficientes.

Variante Lamarckiana

Una vez establecida la configuración a utilizar, la cual se ha descrito al comienzo de esta sección, empezamos exponiendo la evolución de las soluciones en cada generación para el fichero de datos ***lipa30a.dat***.

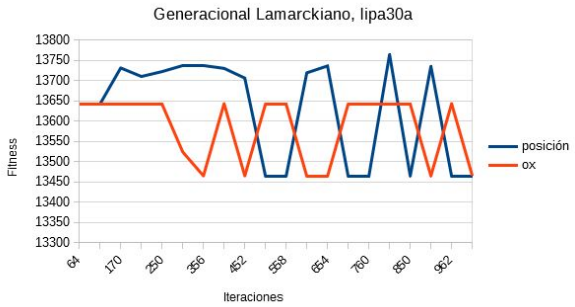

Figura 14. Comparativa Generacional Lamarckiano utilizando lipa30a.dat

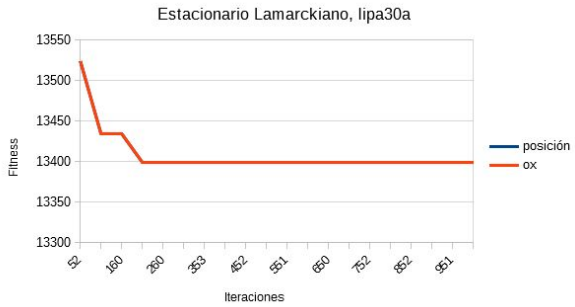

Figura 15. Comparativa Estacionaria Lamarckiano utilizando lipa30a.dat.

Como podemos observar en la variante **generacional** ambos operadores de cruce presentan una gran variación en las soluciones obtenidas, lo cual de nuevo representa el comportamiento de este tipo de algoritmo genético. Sin embargo, en ambas ejecuciones se obtienen soluciones iguales o muy similares. En contraposición a este comportamiento, se presenta la variante **estacionaria** que dispone de una función decreciente más monótona en la que ambas ejecuciones presentan el mismo comportamiento, de ahí que no se visualice la evolución del operador de posición. Comprobaremos más adelante si este comportamiento se repite para ficheros de mayor tamaño. Mientras tanto adjunto la tabla con los resultados exactos así como el tiempo transcurrido para cada ejecución.

	AGG Posición	**AGG OX**	**AGE Posición**	**AGE OX**
Coste	13465	13465	13399	13399
Tiempo	0.092124 s.	0.110634 s.	1.49964 s.	1.47526 s.

Tabla 10. Costes y tiempos Algoritmos Genéticos Lamarckianos, lipa30a.dat

Tal y como podemos observar, los costes de las soluciones obtenidas respaldan la teoría de que ambos operadores del algoritmo generacional evolucionan de forma diferente pero llegan a la misma solución. Del mismo modo, los algoritmos estacionarios además de obtener la misma solución, también disponen del mismo comportamiento con sendos operadores de cruce.

Comparando los resultados obtenidos con la variante *Baldwiniana* se puede observar a simple vista que con la variante *Lamarckiana* se obtienen soluciones con mejores costes. Aunque las diferencias no son demasiado significativas para este conjunto de datos particular, parece que aplicar el aprendizaje en la población ayuda a obtener mejores soluciones y evitar los óptimos locales tan comunes de la búsqueda local. Otro punto a favor de la variante que nos ocupa en esta sección es que también **obtiene las soluciones en bastante menos tiempo que la *Baldwiniana***.

En segundo lugar se presentan las evoluciones de los costes de las mejores soluciones por generación aplicando esta variante en particular al fichero de datos ***tai100a.dat***. El principal objetivo es comprobar si se repite el comportamiento visualizado con el fichero de datos anterior para comenzar a extraer conclusiones acerca de esta variante.

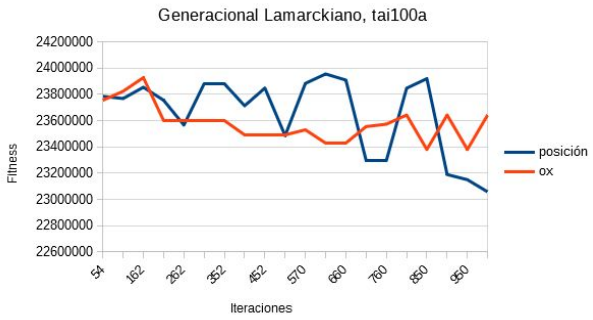

Figura 16. Comparativa Generacional Lamarckiano utilizando tai100a.dat

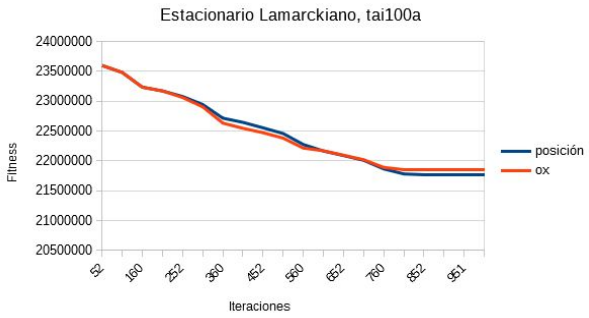

Figura 17. Comparativa Estacionaria Lamarckiano utilizando tai100a.dat.

Tal y como podemos observar, en las ejecuciones del algoritmo **generacional** de nuevo se puede apreciar un comportamiento muy cambiante, solo que en este caso ambos operadores de cruce obtienen soluciones con costes muy diferentes, siendo en este caso el de posición el que obtiene la más cercana a la óptima. Por otra parte, en el algoritmo **estacionario** también se visualiza el comportamiento con el fichero anterior, pues ambos operadores de cruce disponen de una evolución muy similar concluyendo en soluciones con costes también muy parecidos. A continuación se muestra la tabla con los costes exactos y los tiempos de las ejecuciones.

	AGG Posición	**AGG OX**	**AGE Posición**	**AGE OX**
Coste	23057892	23643330	21759826	21846474

Tiempo	0.76401 s.	0.519572 s.	127.382 s.	137.092 s.

Tabla 11. Costes y tiempos Algoritmos Genéticos Lamarckianos, tai100a.dat

Como se puede apreciar, las soluciones obtenidas por cada operador de cruce son bastante similares, algo que ya se podía intuir visualizando las gráficas evolutivas. En cuanto al tiempo, con este fichero de datos, la variante más tardía es la estacionaria, al contrario de lo que sucedía con la variante *Baldwiniana*. Observando ambas ejecuciones de este tipo de genético, he podido comprobar que el proceso se ralentiza considerablemente en las últimas iteraciones. Este hecho debe ser debido a que en las últimas iteraciones, la búsqueda local necesita un mayor número de iteraciones para poder obtener una solución mejor de la que se le presenta, y por ello, esta variante tarda más que la generacional.

Si comparamos los resultados obtenidos con los de la variante ***Baldwiniana***, podemos observar que **los generacionales obtienen soluciones con un mayor coste mientras que los estacionarios mejoran las soluciones levemente**. Por lo tanto, para este conjunto de datos, las diferencias existentes entre los costes tampoco son significativas como para que en base a este criterio se prefiera una variante u otra.

Por último presentamos los resultados obtenidos para el fichero ***tai256c.dat*** aplicando esta variante mediante el mismo procedimiento llevado a cabo con los dos conjuntos de datos anteriores.

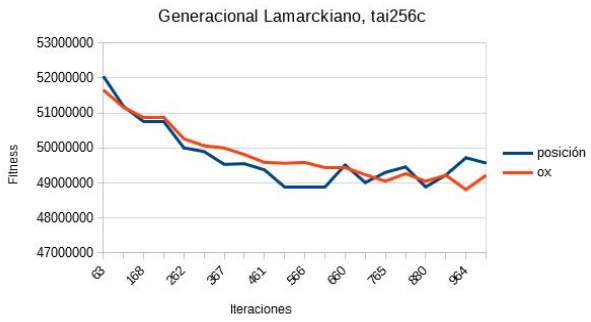

Figura 18. Comparativa Generacional Lamarckiano utilizando tai256c.dat

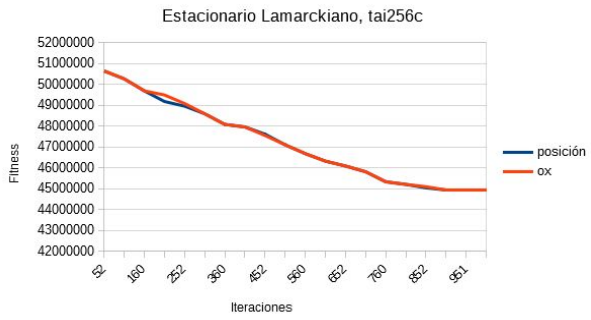

Figura 19. Comparativa Estacionaria Lamarckiano utilizando tai256c.dat.

Si bien se aprecia un comportamiento variante en el algoritmo **generacional**, con este fichero de datos más numeroso, dicha variación se ve más reducida. Esto nos indica que con un conjunto de datos más voluminoso es más difícil encontrar soluciones mejores a las actuales por generación. Sin embargo, en el caso del algoritmo **estacionario** el comportamiento no varía, ambos operadores de cruce disponen de la misma evolución y en este caso, como en el primero, llegan a la misma solución. A continuación se adjunta la tabla con los resultados de los costes obtenidos así como los tiempos de las cuatro ejecuciones.

	AGG Posición	AGG OX	AGE Posición	AGE OX
Coste	49565336	49218786	44919026	44919026
Tiempo	101.041 s.	111.165 s.	5967.96 s.	7313.2 s.

Tabla 12. Costes y tiempos Algoritmos Genéticos Lamarckianos, tai256c.dat

Como podemos observar, de nuevo los costes obtenidos por los operadores de cruce de cada una de las variantes son muy similares, siendo el mismo en el caso de los estacionarios. Con el aumento del número de datos, también aumenta el tiempo invertido en obtener las soluciones, siendo los estacionarios los que de nuevo disponen de un proceso más lento. Sin embargo, para este conjunto de datos, sí existen diferencias significativas entre ambos tipos de genéticos, puesto que los **estacionarios obtienen soluciones bastante próximas a la óptima**, aunque se invierte una mayor cantidad de tiempo.

Como en el caso anterior, si comparamos las soluciones obtenidas con las de la variante ***Baldwiniana***, de nuevo observamos que los generacionales empeoran con respecto a esta pero los estacionarios consiguen soluciones mejores. Por lo tanto podemos concluir, que la variante ***Lamarckiana*** mejora considerablemente el comportamiento de los **algoritmos estacionarios** y consigue soluciones bastante próximas a la óptima con conjuntos de datos voluminosos como el *tai256c*. Sin embargo, el tiempo también es considerablemente mayor por lo que elegir una u otra variante dependerá de los resultados que queramos conseguir y del tiempo que estemos dispuestos a esperar.

Algoritmos meméticos

Estos algoritmos son una variante híbrida entre los algoritmos genéticos en combinación con un procedimiento denominado **búsqueda local**. Esta, normalmente, se aplica cada cierto número de generaciones y puede tomar a la población al completo o a un subconjunto de individuos.

Si bien existen un gran número de variantes para la búsqueda local, en mi caso he desarrollado la denominada ***el primero mejor***, consistente en actualizar la solución actual en cuanto se genere una mejor, pasando directamente a la siguiente iteración. La búsqueda se detendrá cuando se haya explorado el vecindario completo de una solución en particular sin haber obtenido ninguna mejor. Para controlar este hecho he usado una técnica denominada ***don't look bits***, para gestionar la lista de las posibles instalaciones que aún pueden mejorar su localización. Estos serán los que tengan asociado el bit 0, mientras que las localizaciones ya optimizadas tendrán asociado el bit 1, indicando que no se realizarán más búsquedas en sus vecindarios para intentar encontrar una mejor solución.

Inicialmente la máscara estará a 0 para todas las instalaciones, y conforme vaya avanzando el algoritmo realizando intercambios entre los genes y evaluando las soluciones

resultantes, si no se consigue obtener una mejor a la actual entonces se le asigna el bit 1 para dejar de explorar su vecindario. Por el contrario, si alguno de los movimientos genera una solución de menor coste, se le asigna el bit 0 para volver a explorar el vecindario en busca de una solución aún mejor.

Para esta sección se combinan las variantes genéticas anteriores, tanto el generacional como el estacionario con sus dos operadores de cruce cada uno, **aplicando la búsqueda local cada 10 generaciones con un máximo de hasta 400 iteraciones por ejecución**. De nuevo, se inicializa una población de 50 cromosomas de forma aleatoria, puesto que al intentar hacerlo con el algoritmo Greedy, en este tipo de algoritmos empeora muchísimo el coste de las soluciones finales, haciendo falta un mayor número de iteraciones para obtener una solución mejor. Por lo tanto, en este caso, siempre se generará la población inicial de forma aleatoria. Por último cabe destacar que las probabilidades de cruce y mutación son las mismas (0.7 y 0.001 por gen, respectivamente) y que se realizarán 50.000 iteraciones para cada una de las ejecuciones.

Asimismo, dentro de cada variante memética se aplicará la búsqueda local en los siguientes tres casos:

1. A la población completa.
2. Al 10 % de la población escogidos de forma aleatoria.
3. Al 10% de las soluciones mejores de la población.

De este modo podremos comprobar la eficacia de la búsqueda local en estos tres casos para comprobar cuál es el que mejor calidad-tiempo presenta.

Comenzamos con el fichero de menor número de datos: **lipa30a.dat**. La evolución del *fitness* de las mejores soluciones por generación para la variante generacional y para la variante estacionaria se pueden visualizar a continuación. En este caso se ha aplicado la búsqueda local a toda la población.

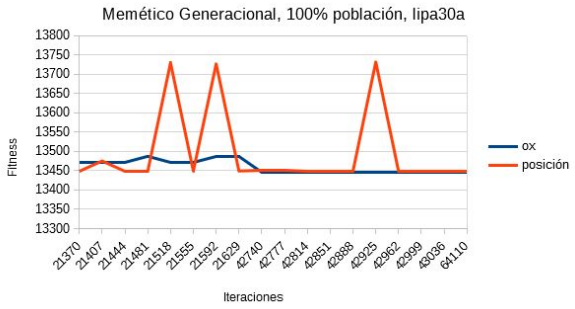

Figura 20. Comparativa Memético Generacional a la población completa utilizando lipa30a.dat

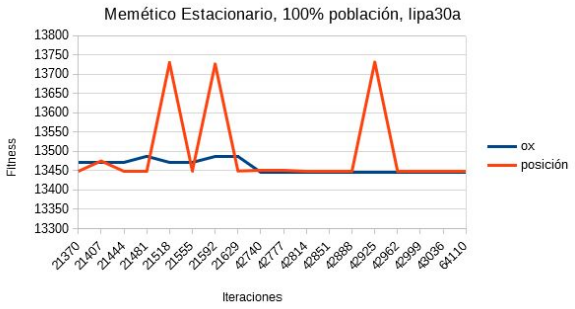

Figura 21. Comparativa Memético Estacionario a la población completa utilizando lipa30a.dat

Tal y como podemos observar, el comportamiento de sendos algoritmos es sumamente parecido. Principalmente destaca la naturaleza cambiante del operador de cruce, como se ha demostrado en ocasiones anteriores, frente a la constante convergencia del algoritmo al utilizar el operador de cruce OX. No obstante, para este fichero y esta variante de la búsqueda local no existen prácticamente diferencias entre los rendimientos de las dos variantes. A continuación se presentan los resultados de aplicar la búsqueda local a los dos casos restantes para sendos algoritmos meméticos.

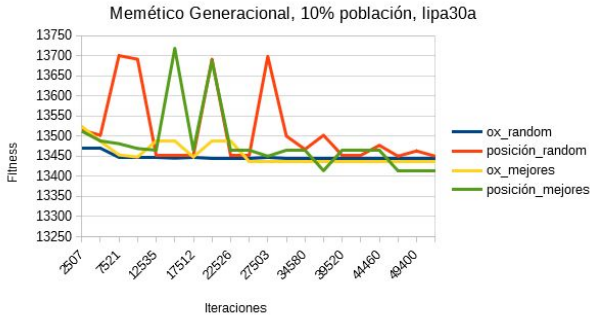

Figura 22. Comparativa Memético Generacional al 10% de la población utilizando lipa30a.dat

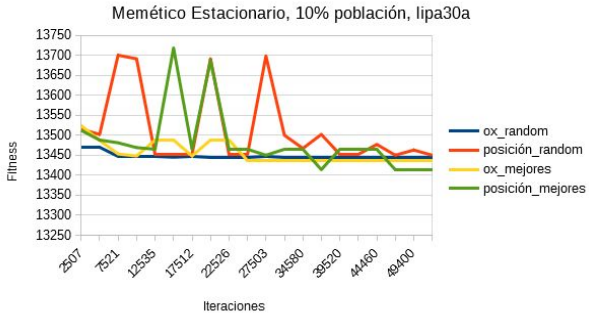

Figura 23. Comparativa Memético Estacionario al 10% de la población utilizando lipa30a.dat

De nuevo, podemos visualizar que sendos algoritmos comparten rendimiento tanto en la variante de aplicar la búsqueda local al 10% de la población de forma aleatoria (líneas roja y azul) como al 10% de los mejores de esta (líneas amarilla y verde). Este fenómeno puede deberse a la facilidad que tiene la búsqueda local para centrarse en óptimos locales, y dado un fichero con un conjunto de datos reducido, la posibilidad de encontrar un óptimo local y no poder salir de esa zona es sumamente elevada. A continuación se visualizan dos tablas relativas a los costes de las soluciones finales así como el tiempo invertido en cada variante.

Búsqueda Local	**Memético Generacional Posición**	**Memético Generacional OX**	**Memético Estacionario Posición**	**Memético Estacionario OX**
100% población	13449	13445	13449	13445
10% aleatoria	13450	13445	13450	13445
10% mejores	13414	13437	13414	13437

Tabla 13. Costes Algoritmos Meméticos, lipa30a.dat

Búsqueda Local	Memético Generacional Posición	Memético Generacional OX	Memético Estacionario Posición	Memético Estacionario OX
100% población	0.039595 s.	0.035189 s.	0.039715 s.	0.036141 s.
10% aleatoria	0.118997 s.	0.07521 s.	0.118858 s.	0.073859 s.
10% mejores	0.094378 s.	0.074085 s.	0.095113 s.	0.074531 s.

Tabla 14. Tiempos Algoritmos Meméticos, lipa30a.dat

Tal y como podemos apreciar los resultados, como demuestran las gráficas, son muy similares por lo que con este conjunto de datos no podemos diferenciar el rendimiento que proporcionan ambas variantes. Sin embargo, si comparamos los costes con los **algoritmos genéticos** podemos observar que en el caso del **generacional,** la variante memética análoga produce unas soluciones mejores y con un tiempo considerablemente menor. Sin embargo, en el caso de la variante **estacionaria** su homólogo memético no es capaz de mejorar los resultados. Esto nos indica que la inclusión de la búsqueda local en la variante generacional ha sido beneficiosa puesto que ha dotado al algoritmo de una mayor estabilidad y constancia a la hora de converger hacia una solución. Sin embargo, como estas cualidades ya las poseía la variante estacionaria, esta no ha mejorado incluyendo este procedimiento para este fichero en concreto.

A continuación repetimos el mismo procedimiento anterior para el caso del fichero de datos ***tai100a.dat***. En primer lugar, mostramos la evolución del coste de la mejor solución por cada generación considerando a toda la población para aplicar la búsqueda local.

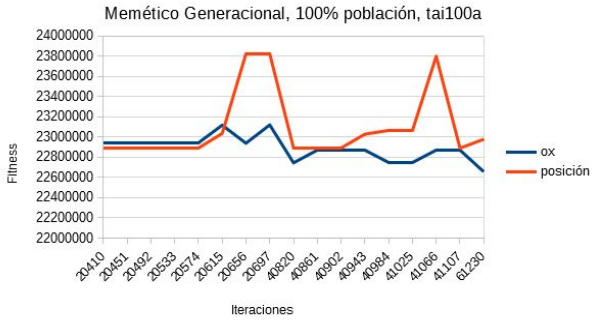

Figura 24. Comparativa Memético Generacional a la población completa utilizando tai100a.dat

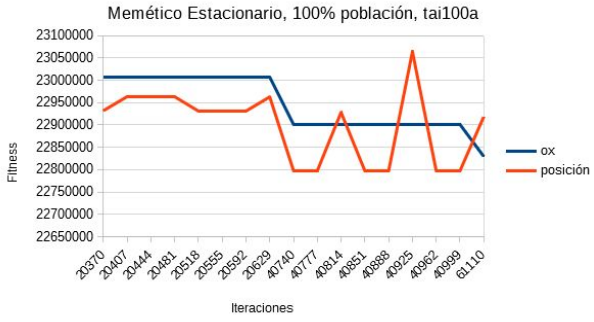

Figura 25. Comparativa Memético Estacionario a la población completa utilizando tai100a.dat

Para un fichero de mayores dimensiones como es este, a diferencia del caso anterior, sí existen diferencias significativas entre el comportamiento de ambos algoritmos. En el primer caso observamos la típica variedad de costes del operador de cruce basado en posición, frente a la convergencia más calmada del operador de cruce OX. Sin embargo, en el estacionario también se visualizan bastantes picos en el operador de cruce basado en

posición, cosa que en el genético estacionario original no ocurría. Por lo tanto, podemos concluir que la inclusión de la búsqueda local ha provocado que sea el algoritmo sea más influenciado por el conjunto de datos que maneje. A continuación se muestran las gráficas de los dos casos restantes de la búsqueda local para comprobar si este nuevo cambio ha sido beneficioso o no.

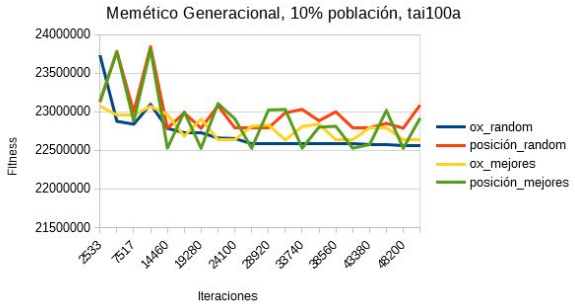

Figura 26. Comparativa Memético Generacional al 10% de la población utilizando tai100a.dat

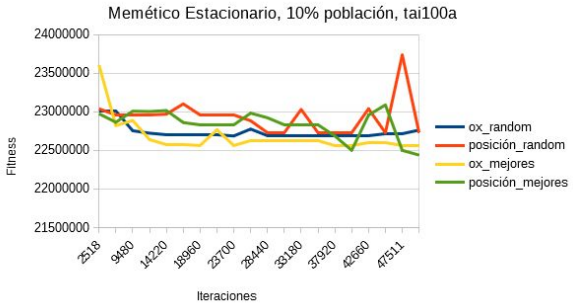

Figura 27. Comparativa Memético Estacionario al 10% de la población utilizando tai100a.dat

En estas dos gráficas se muestra mejor el comportamiento variante del operador de cruce basado en posición, en ambas variantes, así como la convergencia más constante con el operador OX también en sendos meméticos. No obstante, podemos visualizar diferencias entre el comportamiento de los algoritmos aplicando la búsqueda local de forma aleatoria, en cuyo caso el memético generacional proporciona el mejor resultado con el operador OX, mientras que en el estacionario es la búsqueda sobre el 10% de los mejores la que proporciona la mejor solución, y además con el operador de posición. A continuación se muestran los resultados con más detalle así como los tiempos de ejecución.

Búsqueda Local	**Memético Generacional Posición**	**Memético Generacional OX**	**Memético Estacionario Posición**	**Memético Estacionario OX**
100% población	22977484	22654456	22918444	22829178
10% aleatoria	23087016	22555986	22726830	22762188
10% mejores	22918192	22634844	22437836	22563102

Tabla 15. Costes Algoritmos Meméticos, tai100a.dat

Búsqueda Local	**Memético Generacional Posición**	**Memético Generacional OX**	**Memético Estacionario Posición**	**Memético Estacionario OX**

100% población	0.24198 s.	0.20154 s.	0.232757 s.	0.191792 s.
10% aleatoria	0.904023 s.	0.603923 s.	0.878943 s.	0.582374 s.
10% mejores	0.916647 s.	0.606342 s.	0.89154 s.	0.575424 s.

Tabla 16. Tiempos Algoritmos Meméticos, tai100a.dat

Efectivamente, tal y como podemos apreciar, el mejor resultado lo proporciona la variante estacionaria con un operador de posición. Esto nos indica que la inclusión de la búsqueda local sobre el 10% de los mejores de la población ha sido bastante beneficiosa para este operador puesto que le proporciona la diversidad necesaria para obtener soluciones mejores en cuanto al coste. Sin embargo, para conseguir esta mejora se paga un precio en tiempo puesto que es una de las ejecuciones que ha tardado más en terminar. En este conjunto de datos es más notable la gran diferencia de tiempo entre aplicar la búsqueda local a toda la población o aplicarla sobre el 10%, el cual hay que elegirlo según alguno de los dos criterios comentados anteriormente, lo que supone un aumento importante en la inversión temporal. Sin embargo, a la vista de los costes resultantes, este aumento temporal no merece la pena puesto que las diferencias entre los tres criterios de la búsqueda local apenas son significativos, ya que casi todos se encuentran en torno a los 22 millones.

Si comparamos estos resultados con los obtenidos en los **algoritmos genéticos**, podemos observar que, de forma general, los costes de las soluciones de los meméticos son más elevados que los obtenidos por los genéticos. Este hecho se debe en que como el conjunto de datos es mayor, también aumenta el número de soluciones posibles así como la probabilidad de caer en un óptimo local, algo muy común en la búsqueda local. Por lo tanto, por ahora podemos determinar que los algoritmos meméticos son mucho más débiles a quedar atrapados en mínimos locales y por ende aportan soluciones peores aunque en bastante menos tiempo.

Para terminar con este apartado, procedemos a analizar los resultados obtenidos con el fichero ***tai256c.dat*** del mismo modo que en los dos casos anteriores. Comenzamos mostrando la evolución del *fitness* para el primer caso de la búsqueda local, en el cual se aplica a toda la población.

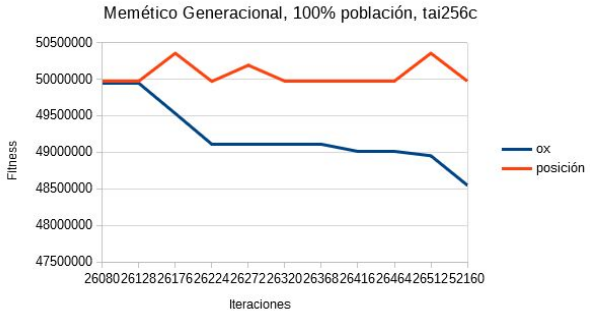

Figura 28. Comparativa Memético Generacional a la población completa utilizando tai256c.dat

Figura 29. Comparativa Memético Estacionario a la población completa utilizando tai256c.dat

Con este conjunto de datos, la gran variación que caracterizaba el comportamiento de sendos meméticos en el caso anterior ha desaparecido casi por completo. Asimismo, otro dato sorprendente, es que el comportamiento del memético **estacionario** vuelve a ser más sosegado, tal y como se mostraba en el genético original. De ese modo, la diferencia de costes entre el operador de cruce basado en posición y el OX apenas parece significativa, aunque esto lo confirmaremos más adelante con los costes exactos. Por el contrario, en la variante **generacional** ambos operadores proporcionan soluciones con costes muy dispares. Esta tendencia ya se ha podido visualizar en casos anteriores del genético generacional, como por ejemplo con el conjunto de datos *tai100a*. A continuación se muestran las gráficas relacionados con los dos casos restantes de la aplicación de la búsqueda local para este conjunto de datos particular.

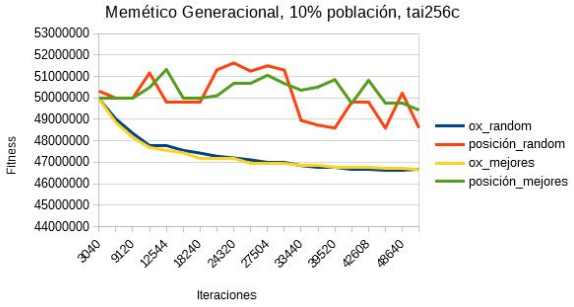

Figura 30. Comparativa Memético Generacional al 10% de la población utilizando tai256c.dat

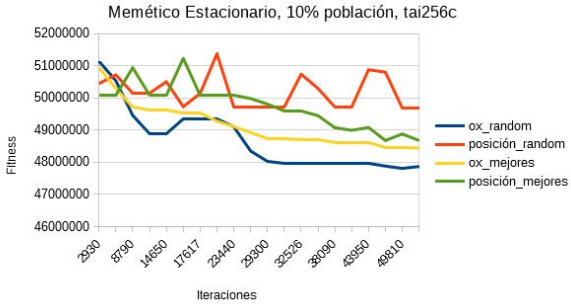

Figura 31. Comparativa Memético Estacionario al 10% de la población utilizando tai256c.dat

En este caso, con el conjunto de datos *tai256c*, podemos apreciar el comportamiento del algoritmo genético **generacional** en el memético, puesto que el operador de cruce basado en posición sigue caracterizado por sus numerosos picos en los costes de las soluciones encontradas, mientras que el OX tanto aplicado al 10% aleatorio como al 10% de los mejores muestra una suave curva descendente. Sin embargo, esto no sucede en el

estacionario ya que, a diferencia del genético original, presenta un mayor número de variaciones en relación a los costes de las mejores soluciones por generación. No obstante, al igual que el generacional, los mejores resultados también los proporciona el operador OX, aunque en este caso sí que existe una diferencia considerable entre aplicar la búsqueda local al 10% de la población de forma aleatoria o al 10% de los mejores, el cual proporciona una peor solución. Esto puede deberse a que no siempre se puede obtener una solución más próxima a la óptima escogiendo los mejores, si no que en ocasiones es mejor aportar un mayor grado de diversidad en la población para aplicar la búsqueda local con el objetivo de evitar los mínimos locales. Por último, se muestra a continuación la tabla con los resultados detallados así como sus tiempos para este fichero en cuestión.

Búsqueda Local	**Memético Generacional Posición**	**Memético Generacional OX**	**Memético Estacionario Posición**	**Memético Estacionario OX**
100% población	49969022	48548984	50145804	48548984
10% aleatoria	48591928	46673808	49691876	46673808
10% mejores	49424914	46652946	48673854	48439826

Tabla 17. Costes Algoritmos Meméticos, tai256c.dat

Búsqueda Local	**Memético Generacional Posición**	**Memético Generacional OX**	**Memético Estacionario Posición**	**Memético Estacionario OX**
100% población	0.890224 s.	0.72537 s.	0.810625 s.	0.717488 s.
10% aleatoria	4.69848 s.	3.30083 s.	4.25701 s.	3.31701 s.
10% mejores	4.79513 s.	3.29589 s.	3.87166 s.	2.69567 s.

Tabla 18. Tiempos Algoritmos Meméticos, tai256c.dat

Tal y como podemos observar las dos mejores soluciones las proporciona el memético generacional con el operador OX al aplicar la búsqueda local sobre el 10% aleatorio y sobre el 10% de los mejores de la población. Sin embargo, como hemos podido comprobar en el caso anterior, el tiempo necesario para llegar a ambas soluciones es superior que aplicar la búsqueda local a toda la población, y más considerando un fichero con un conjunto de datos de mayor envergadura. No obstante, estos resultados están lejos de ser competentes con los proporcionados por los **algoritmos genéticos**, y es que es en este conjunto de datos donde se hace más visible la pérdida de rendimiento a la hora de incluir una búsqueda local en un genético, puesto que es demasiado propensa en caer en óptimos locales y devolverlos como solución, especialmente cuando el conjunto de datos es voluminoso, como es este tercer caso. Sin embargo, cabe destacar que el **tiempo invertido es hasta**

cuatro veces menor, en algunos casos, comparado con los genéticos. Por lo tanto, escogeremos estos o los meméticos en función de las prestaciones de nuestro ordenador y del tiempo que deseemos invertir en buscar una solución próxima a la solución óptima.

Conclusiones finales

Como hemos podido comprobar a lo largo de este documento, por lo general los algoritmos desarrollados no presentan comportamientos demasiado diferentes dependiendo del conjunto de datos que gestionen. En el caso del fichero de datos de menor tamaño *lipa30a.dat*, los **algoritmos genéticos tradicionales** no presentan diferencias considerables en las soluciones que obtienen. De igual modo ocurre con un conjunto de datos mayor, como es *tai100a.dat*, con el que también se obtienen soluciones bastante similares entre las ejecuciones realizadas. Por último, con el conjunto de datos más voluminoso el comportamiento es similar al de los dos anteriores, por lo que podemos concluir que este tipo de algoritmos son poco influenciables en razón al conjunto de datos que se les suministre. Además, tal y como se ha comentado anteriormente, generalmente la variante **estacionaria** presenta mejores resultados que la **generacional** debido a que su esquema de reemplazamiento no es tan agresivo y por ende no descarta tantas soluciones como en el caso del generacional, pudiendo perder la más cercana a la óptima. Del mismo modo, el operador de cruce OX ha demostrado en la mayoría de ocasiones proporcionar mejores costes comparado con el de posición debido a que el proceso de generación de descendientes se adapta al conjunto de datos, transmitiendo más o menos genes en función de si hay más o menos datos, respectivamente.

En relación a las variantes ***Baldwinianas***, podemos observar que también la estacionaria presenta mejores resultados en la mayoría de casos pero no las diferencias no son significativas independientemente del conjunto de datos. Otro aspecto a tener en cuenta es el considerable aumento de tiempo de la variante generacional, por lo cual la inclusión del algoritmo 2-*opt* empeora el rendimiento de este tipo de genético en particular. De forma contraria, la variante ***Lamarckiana*** aumenta el tiempo en el otro genético, el estacionario, siendo tan desorbitado que se han reducido de nuevo las iteraciones para poder obtener resultados con los que realizar el análisis. Sin embargo, los compensa obteniendo soluciones muy próximas a la óptima con un número menor de iteraciones que los genéticos en ficheros de datos voluminosos, como el *tai256c*.

Por último, los **algoritmos meméticos** se caracterizan por su rapidez pero también por su propensión a caer en óptimos locales y proporcionar soluciones bastante distantes a la óptima. Este comportamiento se acentúa conforme el conjunto de datos es mayor, por lo que no son los más acertados para aplicarlos al fichero *tai256c.dat*. Asimismo, también hemos podido observar que aplicar la búsqueda local ***primer el mejor*** proporciona mejores resultados cuando se aplica solo a una parte de la población que a toda, puesto que si modificamos la población al completo la posibilidad de caer en óptimos locales es mayor. Sin embargo, aplicando este algoritmo a una parte reducida de la población se introduce

cierta diversidad que acaba mejorando las siguientes generaciones consiguiendo una solución mejor.