Máster Universitario en Ingeniería Informática

INTELIGENCIA COMPUTACIONAL

PRÁCTICA 2: Algoritmos Evolutivos
QAP

UNIVERSIDAD DE GRANADA

ETSIIT
Escuela Técnica Superior de Ingenierías Informática y de Telecomunicación

Carlos Santiago Sánchez Muñoz

Grupo de prácticas 1 - Lunes

Email: carlossamu7@correo.ugr.es

4 de enero de 2020

Índice

1. Descripción del problema

El problema de la asignación cuadrática o QAP [Quadratic Assignment Problem] es un problema de optimización combinatoria con numerosas aplicaciones. El problema se puede describir de la siguiente forma.

Supongamos que queremos decidir dónde construir n instalaciones (p.ej. fábricas) y tenemos n posibles localizaciones en las que podemos construir dichas instalaciones. Conocemos las distancias que hay entre cada par de instalaciones y también el flujo de materiales que ha de existir entre las distintas instalaciones (p.ej. la cantidad de suministros que deben transportarse de una fábrica a otra). El problema consiste en decidir dónde construir cada instalación de forma que se minimize el coste de transporte de materiales.

Formalmente, si llamamos $d(i,j)$ a la distancia de la localización i a la localización j y $w(i,j)$ al peso asociado al flujo de materiales que ha de transportarse de la instalación i a la instalación j, hemos de encontrar la asignación de instalaciones a localizaciones que minimice la función de coste

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

donde $p()$ define una permutación sobre el conjunto de instalaciones.

Igual que en el problema del viajante de comercio, que se puede considerar un caso particular del QAP, una solución para el problema es una permutación del conjunto de instalaciones que indica dónde se debe construir cada una.

El problema de la asignación cuadrática es un problema habitual en Investigación Operativa y, además de utilizarse para decidir la ubicación de plantas de producción, también se puede utilizar como modelo para colocar los componentes electrónicos de un circuito sobre una placa impresa o los módulos de un circuito integrado en la superficie de un microchip.

Por su interés teórico y práctico, existe una variedad muy amplia de algoritmos que abordan la resolución del problema de la asignación cuadrática. Al ser un problema NP-completo, el diseño y aplicación de algoritmos exactos para su resolución no es viable cuando n es grande. Nos centraremos, por tanto, en el diseño de algoritmos evolutivos y evaluaremos su rendimiento sobre instancias concretas del problema.

2. Operadores

Los algoritmos genéticos son algoritmos bioinspirados basados en la evolución. Como tales tienen operadores de cruce, mutación y selección. En algunos casos hay varios de ellos. A lo largo de este trabajo se realizará una comparativa de como tomar una decisión u otra puede afectar a los resultados.

2.1. Operador de cruce

Estos operadores deciden cómo se construyen los hijos de una nueva generación. Es decir, como cruzar dos padres para obtener los hijos.

El primer operador que se ha implementado es el **cruce OX** en donde se escoge una subcadena de uno de los padres y la subcadena complementaria del otro padre completando el hijo. En la elección de esta subcadena he usado los dos cuartos centrales del primer ascendiente y completo con el primer y último cuarto del otro ascendiente.

El otro operador es el de **cruce de posición** que mantiene en los descendientes aquellos genes comunes y el resto eligiendo de forma aleatoria.

2.2. Operador de mutación

Al igual que en la genética biológica también se implementan operadores de mutación que permite alterar la población. Esto resulta muy útil ya que permite la aparición de nuevas soluciones y muchas veces permite salir de óptimos locales para viajar a mejores soluciones.

El operador de mutación implementado es tan sencillo como intercambiar dos genes cualesquiera.

2.3. Operador de selección

El otro ingrediente fundamental de los algoritmos evolutivos es el operador de selección. Éste decide qué elementos de la población continúan en la siguiente generación. Se ha implementado un **torneo binario**. Dos individuos de la población son escogidos aleatoriamente y compiten entre ellos mediante su coste o *fitness*. He ordenado la población de modo que en el torneo binario el mejor es aquel que tiene un índice inferior. De este modo los padres son los mejores posibles manteniendo el elitismo.

3. Algoritmos

En esta sección se va a contar brevemente los algoritmos implementados. Antes de entrar en los tecnicismo de los algoritmos hay que decidir cómo establecer la población inicial. Hacerlo de forma aleatoria sin repetición es claramente una posibilidad.

¿Existe alguna forma mejor? La respuesta es afirmativa. Por ejemplo podemos usar como población inicial una generada mediante un **algoritmo greedy**. Dicho algoritmo busca minimizar la distancia a la vez que maximizar el flujo ya que es un problema de optimización múltiple. El tamaño de la población elegido es `20`.

3.1. Algoritmo Genético Generacional

Este algoritmo en cada generación sustituye la población actual por la nueva. Se escogen los padres mediante el torneo binario y existe una **probabilidad de cruce** de que estos se reproduzcan. Los padres no cruzados se mantienen de modo que la población de una iteración con la de la siguiente tenga la misma dimensión.

Asimismo también se usa el operador de mutación que permite intercambiar dos genes en función de una **probabilidad de mutación**. El último componente esencial de estos algoritmos es que el mejor individuo de la población ha de sobrevivir. Esto se conoce como **elitismo**.

Tras realizar pruebas los mejores parámetros hallados son:

```
Probabilidad de cruce: 0.7
Probabilidad de mutación. 0.001
```

3.2. Algoritmo Genético Estacionario

Esta otra modalidad tiene diferencias notables respecto a la anterior. En cada iteración se escogen dos padres (aleatoriamente) y se les aplica los operadores genéticos. Del mismo modo sólo se incluyen los hijos si son mejores que las dos peores soluciones de la generación en cuestión.

Lógicamente este algoritmo toma menos tiempo que el Generacional y a veces permite mantener soluciones que a la postre nos permiten llegar a soluciones muy buenas.

3.3. Variante Baldwiniana

Esta variante incorpora técnicas de optimización local al algoritmo genético estándar para dotar a los individuos de aprendizaje. Se evalúa el fitness teniendo en cuenta esta optimización pero los individuos a usar son con el material genético original. Se ha implementado el **algoritmo greedy de transposición 2-opt** como técnica de optimziación.

Obsérvese un pequeño fragmento de código, que permite entender muy bien esta idea:

```
void Geneticos::Baldwiniana(vector<Cromosoma> &pop) {
  for (unsigned i=0; i<pop.size(); ++i) {
    algGreedy.Alg2opt(pop[i], datos);
    pop[i].fitness = algGreedy.solutionGreedy.fitness;
  }
}
```

3.4. Variante Lamarckiana

Esta variante es análoga a la anterior pero manteniendo el material genético aprendido en el proceso de aprendizaje. A nivel de código:

```
void Geneticos::Lamarckiana(vector<Cromosoma> &pop) {
  for (unsigned i=0; i<pop.size(); ++i) {
    algGreedy.Alg2opt(pop[i], datos);
    pop[i].solution = algGreedy.solutionGreedy.solution;
    pop[i].fitness = algGreedy.solutionGreedy.fitness;
  }
}
```

4. Comparativa

Para dicha comparativa se ha usado el conjunto de datos `tai60a.dat`.

4.1. AGE

Comenzamos con los Algoritmos Estacionarios.

Algoritmo estándar con población inicial greedy

Tabla de resultados:

IterGenetic	FitnessGenetic	FitnessGenetic OX
20000	7637546	7727468
40000	7523292	7536962
60000	7523292	7503284
80000	7523292	7503284
100000	7523292	7503284
120000	7523292	7503284
140000	7523292	7503284
160000	7523292	7503284
180000	7523292	7503284
200000	7523292	7503284

Gráfica:

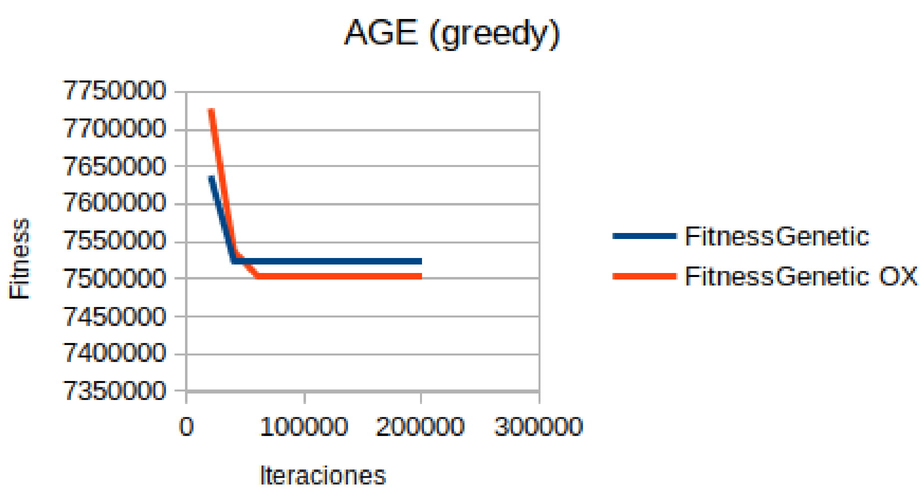

Imagen 1: Algoritmo estándar con población inicial greedy (AGE)

Comienza desde un fitness decente debido a su inicialización greedy y decae rapidísimamente para estacionarse en un óptimo local. El cruce OX es algo superior.

Baldwiniana

Tabla de resultados:

IterBaldwiniana	FitnessBaldwiniana	FitnessBaldwiniana OX
10000	7641366	7727526
20000	7598224	7617646
30000	7598224	7593314
40000	7598224	7575026
50000	7598224	7557268
60000	7598224	7557268
70000	7598224	7557268
80000	7598224	7557268
90000	7598224	7557268
100000	7598224	7557268

Gráfica:

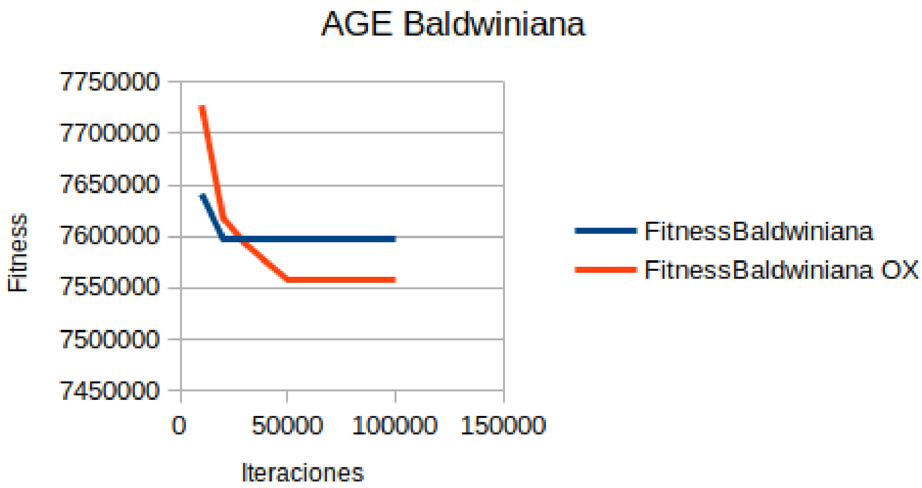

Imagen 2: Baldwiniana (AGE)

En este caso se observa que la mejora del *fitness* es más progresiva con una buena pendiente de decrecimiento. Posteriormente también estaciona. El cruce OX es bastante superior al de posición.

Lamarckiana

Tabla de resultados:

IterLamarckiana	FitnessLamarckiana	FitnessLamarckiana OX
1000	7560410	7560410
2000	7560410	7560410
3000	7560410	7560410
4000	7560410	7560410
5000	7560410	7560410
6000	7560410	7560410
7000	7560410	7560410
8000	7560410	7560410
9000	7560410	7560410
10000	7560410	7560410

Gráfica:

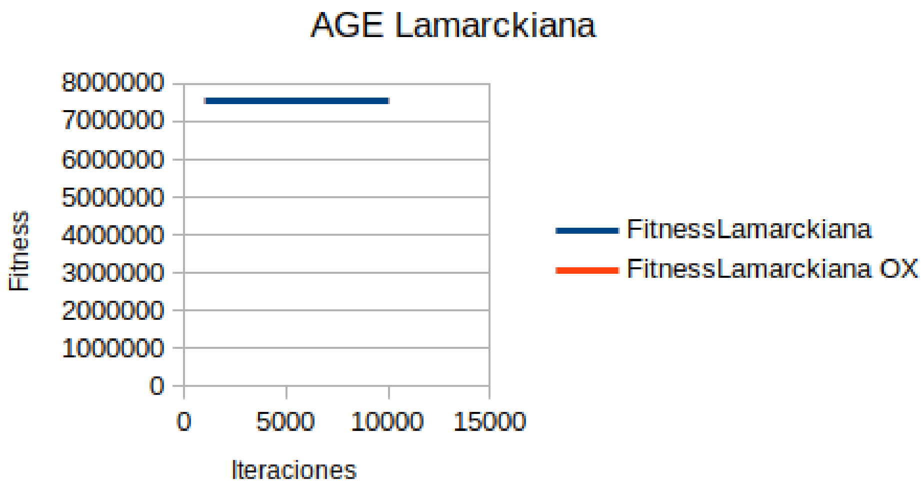

Imagen 3: Lamarckiniana (AGE)

No existe variación alguna entre usar 1000 iteraciones o 10000, ambas proveen el mismo *fitness*. Quizás se debería probar con valores más pequeños como trabajo futuro pero el hecho de que el aprendizaje permanezca entre iteraciones genera este efecto, al menos sobre este conjunto de datos.

4.2. AGG

Ahora pasamos a comparar resultados en los Algoritmos Generacionales.

Algoritmo estándar con población inicial greedy

Tabla de resultados:

IterGenetic	FitnessGenetic	FitnessGenetic OX
20000	7776278	7832018
40000	7725174	7745818
60000	7682876	7646762
80000	7670010	7631202
100000	7640942	7584130
120000	7619880	7583892
140000	7607828	7580972
160000	7587250	7569986
180000	7587250	7534020
200000	7583398	7534020

Gráfica:

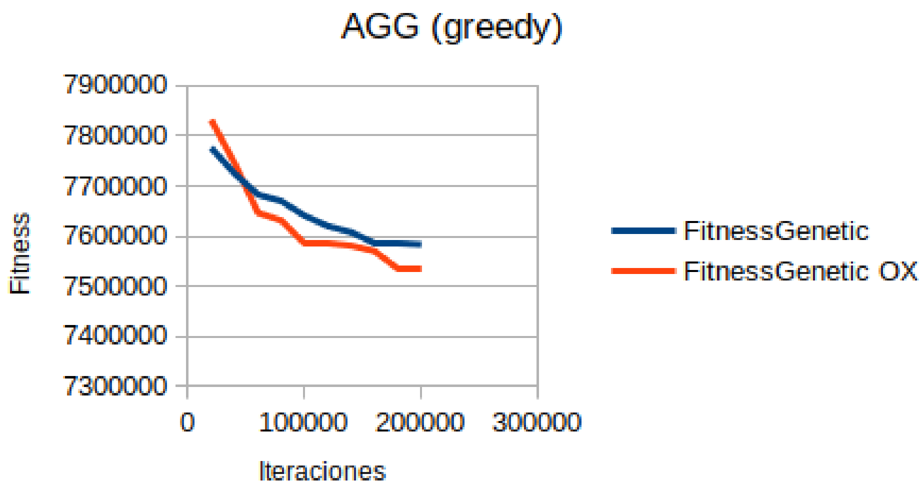

Imagen 4: Algoritmo estándar con población inicial greedy (AGG)

En este caso la caída es muy diferente a lo que ya conocíamos. El cambio de población es más brusco aumentando la complejidad computacional pero también se puede mejorar el coste. Parece que en las últimas iteraciones ya ha estacionado. Como hasta ahora, el cruce OX es superior.

Baldwiniana

Tabla de resultados:

IterBaldwiniana	FitnessBaldwiniana	FitnessBaldwiniana OX
10000	7899320	8075532
20000	7717128	7821946
30000	7760356	7852368
40000	7630506	7795604
50000	7663012	7695824
60000	7622826	7712322
70000	7634528	7722406
80000	7634528	7722406
90000	7567130	7722128
100000	7619172	7697428

Gráfica:

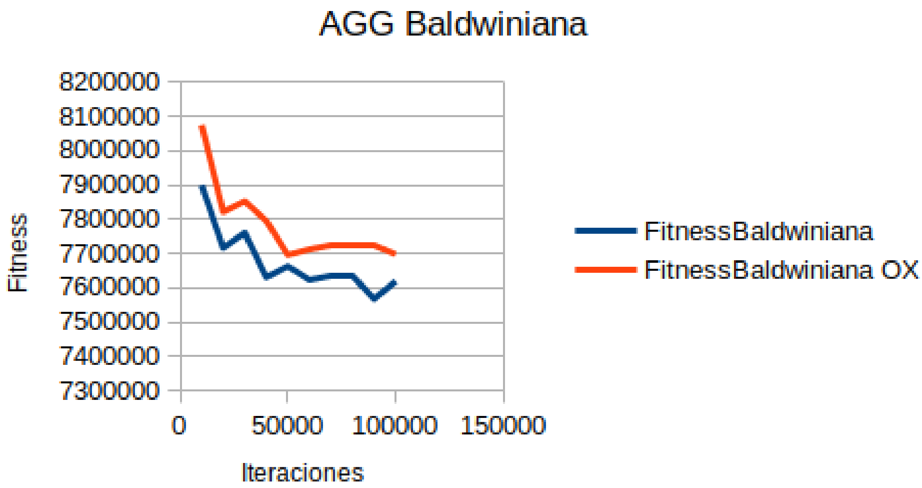

Imagen 5: Baldwiniana (AGG)

Se observan oscilaciones y a veces un aumento de iteraciones puede llevar a un *fitness* peor pero a la larga las soluciones son mejores. En este caso el operador de cruce de posición es mejor. Las mutaciones y cambios drásticos en la población explican dichas leves oscilaciones.

Lamarckiana

Tabla de resultados:

IterLamarckiana	FitnessLamarckiana	FitnessLamarckiana OX
1000	7947652	7947652
2000	7607836	7693590
3000	7542780	7630272
4000	7542780	7630272
5000	7542780	7630272
6000	7542780	7630272
7000	7542780	7630272
8000	7542780	7630272
9000	7542780	7630272
10000	7542780	7630272

Gráfica:

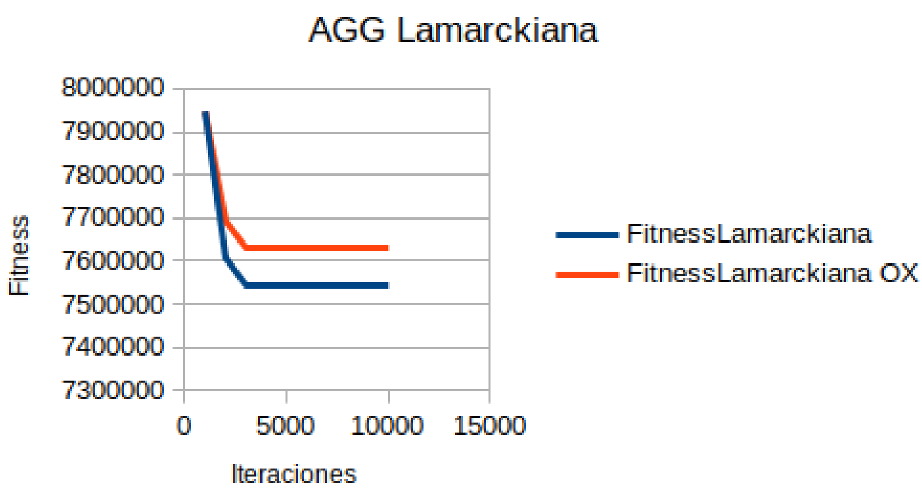

Imagen 6: Lamarckiniana (AGG)

Como era de esperar el coste cae rápidamente para establecerse en un óptimo. Al igual que el caso anterior el cruce por posición es mejor al OX.

5. Resultados

Tras los resultados obtenidos en la comparativa anterior los diferentes algoritmos y variantes genéticos no ofrecen grandes diferencia en los resultados siendo todas ellas bastante buenas. Asimismo pasa con el operador de cruce, no hay evidencia para pensar que OX sea mejor que posición o viceversa.

En esta sección se va a ofrecer los resultados sobre el conjuto de datos pedido, `tai256.c`, el cual tiene un número de instalaciones considerablemente alto.

Mejor *fitness*: `45133724`.

Solución: `132 98 130 13 96 215 49 81 94 229 178 167 125 73 187 246 127 170 244 115 64 207 22 26 46 37 69 60 7 120 24 202 67 43 20 71 161 39 17 9 190 252 175 204 3 84 113 122 135 224 34 221 0 184 250 117 77 63 238 90 212 107 155 226 195 209 173 180 144 255 217 31 28 137 56 235 147 142 152 140 241 165 198 58 103 86 105 52 176 150 92 232 181 210 240 237 216 76 136 251 211 101 201 121 126 197 53 220 40 19 233 93 247 139 68 234 6 25 23 78 182 151 141 248 206 21 205 189 111 171 8 168 208 41 1 118 188 48 242 102 163 61 162 30 74 10 70 79 50 166 42 62 87 199 156 253 82 119 124 179 218 138 44 51 104 91 133 134 4 254 146 55 231 225 100 145 236 157 174 59 249 2 83 36 185 85 75 33 18 196 11 172 245 89 45 239 154 32 57 192 219 223 230 203 243 128 227 213 97 12 65 106 160 169 191 143 159 123 95 29 222 27 54 47 116 186 16 109 194 80 110 66 108 14 158 99 228 131 164 35 153 15 129 72 38 200 193 112 177 114 5 148 88 183 214 149`.

Resultados de AGE:

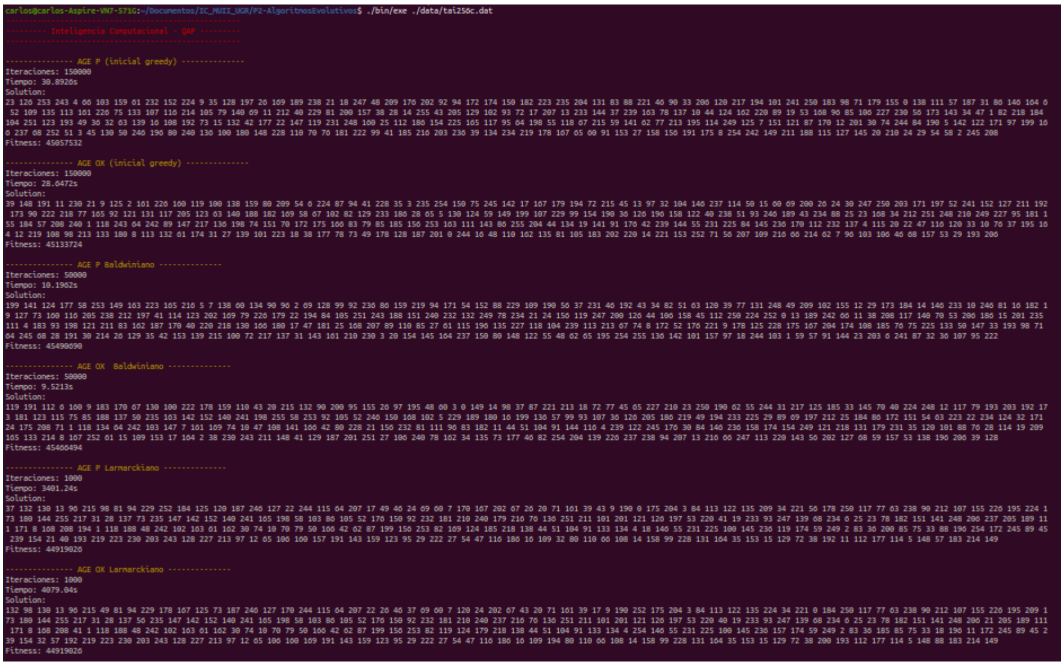

Imagen 7: Resultados AGE sobre `tai256c.dat`