Algoritmos evolutivos:

Problemas de optimización combinatoria QAP

Curso 2020/2021

PEDRO MANUEL FLORES CRESPO

Inteligenica Computacional

Máster en Ingeniería Informática

Índice

1. Introducción

En objetivo principal de esta práctica consiste en resolver el problema que se conoce como QAP o Problema de la Asignación Cuadrática, problema fundamental de optimización combinatoria. Para ello, utilizaremos un algoritmo genético clásico y posteriormente, unas variantes sobre el mismo. Más concretamente, estas variantes son la baldwiniana y lamarckiana que implementan algunas técnicas de optimización basándose en teorías de la evolución. Formalmente el problema se puede formular como sigue:

"Trata de asignar N instalaciones a una cantidad N de sitios o locaciones en donde se considera un costo asociado a cada una de las asignaciones. Este costo dependerá de las distancias y flujo entre las instalaciones. De este modo se buscará que este costo, en función de la distancia y flujo, sea mínimo."

El lenguaje utilizado para llevar a cabo el desarrollo de la práctica ha sido `C++` (al ser compilado presenta mayor velocidad que los interpretados). El mismo se adjunta en la entrega en la carpeta `código`. El "grueso" de la implementación se encuentra en la carpeta `src` donde hay un archivo para cada una de las partes que componen el algoritmo además de los programas principales. A lo largo de la memoria iremos explicando cada una de los procedimientos llevados a cabo para las etapas que componen el algoritmo así como el proceso de optimización para sus variantes.

2. Implementación del algoritmo

Como hemos visto en la asignatura, un algoritmo evolutivo estándar se compone de varias partes diferenciadas entre sí y que recogemos en la figura 1.

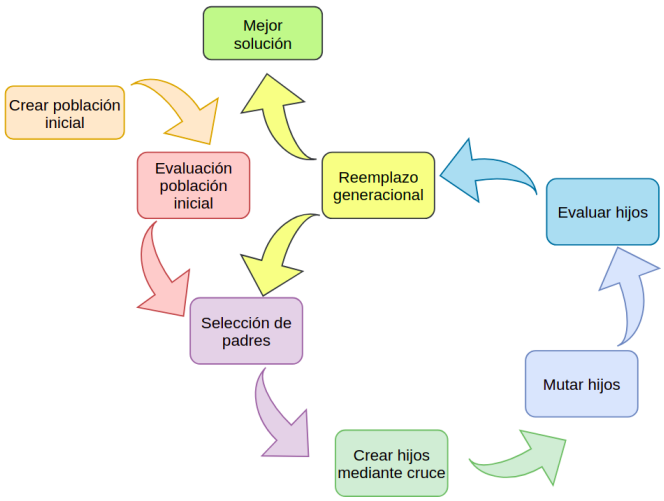


Figura 1: Algoritmo genético clásico.

En la mayoría de las mismas se pueden optar por diferentes estrategias para llevar a cabo dicho procedimiento. En las secciones siguientes explicaremos cada una de las soluciones llevadas a cabo.

2.1. Representación y creación de la población inicial

Como hemos visto en la definición del problema, se trata de asignar N instalaciones en N localizaciones diferentes. Por ello, para representar las posibles soluciones al problema vamos a usar una representación mediante permutaciones. Así, la posición i-ésima de la permutación representa la instalación i para $i = 1, \ldots, N$. De este modo, su valor, llamémoslo $p(i)$, representa la localización de dicha

instalación que también cumple $p(i) \in \{1, \ldots, N\}$. Por ello, el flujo entre las instalaciones i, j viene determinado por $w(i, j)$ y la distancia entre las mismas por $d(p(i), p(j))$ con $i, j, p(i), p(j) = 1, \ldots, N$. En el código, esta permutación la hemos representado mediante la estructura `Cromosoma` como sigue:

```
struct Cromosoma{
    vector<int> permutacion;
    int fitness;
};
```

Código 1: Representación de las soluciones.

Vemos que simplemente consta de un vector con la permutación y al que le añadimos el valor de aptitud o *fitness* que explicaremos más adelante. Partiendo de esta base, para crear la población inicial lo único que tenemos que hacer es rellenar el vector con los números ordenados y luego mezclarlos aleatoriamente.

```
void poblacion_inicial(vector<Cromosoma> &poblacion){
    // Rellenamos primero con los números ordenados
    vector<int> aux(size,0);
    for(int i=0; i<size; ++i){
        aux[i] = i;
    }

    //Asignamos y "permutamos"
    for(int i=0; i<poblacion.size(); ++i){
        poblacion[i].permutacion = aux;
        random_shuffle(poblacion[i].permutacion.begin(),
          poblacion[i].permutacion.end());
        poblacion[i].fitness = fitness(poblacion[i].permutacion);
    }


}
```

Código 2: Creación de la población inicial.

2.2. Función de *fitness*

Cada una de las permutaciones tiene un costo o valor asociado que representan la bondad de la solución. En nuestro caso, este valor viene determinado por el flujo y las distancias entre las instalaciones y localizaciones. Estos valores los tenemos disponibles mediante dos matrices para cada uno de los parámetros. El valor por tanto se obtiene mediante la fórmula

$$\sum_{i=1}^{N} \sum_{j=1}^{N} w(i, j) d(p(i), p(j)).$$

Esta fórmula es "asumible" cuando solo tenemos que ejecutarla pocas veces. Sin embargo, para las variantes del algoritmo estándar necesitamos hacer su cálculo gran cantidad de ocasiones ya que vamos intercambiando dos valores y el proceso de cálculo puede ser bastante costoso. Debido a la naturaleza del cambio, podemos acelerar el proceso partiendo del valor actual y recalculándolo mediante las diferencias existentes, ya que estamos variando solo dos columnas/filas de la matriz (tal y como hemos visto en clase). El mecanismo para obtener el *fitness* sería por tanto el siguiente, que depende de si es un cálculo nuevo o no:

```
int fitness(const vector<int> &permutacion, int coste_ant, int i_ant, int j_ant){
    if( permutacion.size() == size){
        int coste = 0;

        if(coste_ant == -1){ // Lo calculamos por primera vez
            for(int i=0; i<size; ++i){
```

```
                for(int j=0; j<size; ++j){
                    coste += m_flujo[i][j]*m_distancias[permutacion[i]][permutacion[j]];
                }
            }
        } else{ // Lo recalculamos. Permutación NO lleva el cambio
            int diff1 = 0;
            int diff2 = 0;
            coste = coste_ant;
            for (int i=0; i<size; ++i){
                if(i != i_ant && i != j_ant){
                    int val = permutacion[i];
                    diff1 = m_distancias[val][permutacion[j_ant]]
                         - m_distancias[val][permutacion[i_ant]];
                    diff2 = m_distancias[permutacion[j_ant]][val]
                          - m_distancias[permutacion[i_ant]][val];

                    coste += (m_flujo[i][i_ant] - m_flujo[i][j_ant])*diff1
                          + (m_flujo[i_ant][i] - m_flujo[j_ant][i])*diff2;
                }
            }
        }

        return coste;
    }

    return -1;
}
```

Código 3: Cálculo del *fitness*.

2.3. Mecanismos de selección

Ahora nos centraremos en la selección de padres. Existen una gran cantidad de procedimientos: selección proporcional o ruleta, basada en ranking lineal, mediante ranking exponencial, etc. En nuestro caso, hemos optado por usar la selección por torneo. Seleccionamos k individuos aleatoriamente y nos quedamos con el mejor de ellos. Este proceso lo repetimos hasta obtener el tamaño deseado. A su vez, en este algoritmo puede hacerse con reemplazamiento o sin él, dependiendo de si queremos aumentar la presión selectiva o no. Para ver cómo se comporta en un caso u otro, hemos implementado las dos versiones y posteriormente veremos cuál da mejores resultados.

```
void seleccion(const vector<Cromosoma> &poblacion, int k, int tam_selec,
               vector<Cromosoma> &seleccionados){
    srand(time(NULL));
    seleccionados.resize(tam_selec);

    vector<Cromosoma> pob_aux(poblacion);
    int introducidos = 0;

    while(introducidos < tam_selec){
        vector<int> indices;
        indices.resize(pob_aux.size());
        for(int i=0; i<pob_aux.size(); ++i){
            indices[i] = i;
        }
        random_shuffle(indices.begin(), indices.end()); //Los mezclamos
        indices.resize(k);


        vector<int> aux(k);  //Guarda la posición
        vector<Cromosoma> aux2(k); //Guarda el cromosoma
        for(int i=0;i<indices.size();++i){
            aux[i] = indices[i];
            aux2[i] = pob_aux[indices[i]];
        }

```

```
        pair<Cromosoma, int> par= mejor(aux2); //Obtenemos el mejor
        seleccionados[introducidos] = par.first;
        pob_aux.erase(pob_aux.begin()+aux[par.second]); //Borramos recuperando la
    posición

        ++introducidos;
    }

}
```

Código 4: Selección por torneo sin reemplazamiento.

```
void seleccionReemplazamiento(const vector<Cromosoma> &poblacion, int k, int tam_selec,
                              vector<Cromosoma> &seleccionados){
    srand(time(NULL));
    seleccionados.resize(tam_selec);

    int introducidos = 0;
    while(introducidos < tam_selec){
        vector<Cromosoma> aux(poblacion);
        random_shuffle(aux.begin(), aux.end()); //Los mezclamos de manera aleatoria
        aux.resize(k);

        seleccionados[introducidos] = mejor(aux).first;

        ++introducidos;
    }
}
```

Código 5: Selección por torneo con reemplazamiento.

Además de las variantes presentadas, se ha hecho una variante de cada una de ellas. Al hacer la selección por torneo las soluciones con peor valor de *fitness* tienen menos posibilidades de salir. Como es interesante tener una población diversa, se ha implementado una variante de los anteriores en el que se introducen directamente el mejor y peor individuo. Su implementación parte de los presentados y se puede ver más detenidamente en el archivo `código/src/seleccion.cpp`.

2.4. Mecanismos de cruce

Una vez tenemos los padre seleccionados, hay que cruzar los mismos para generar descendientes. El operador de cruce seleccionado es el de *cruce en un punto*. el punto seleccionado es por defecto la mitad de uno de los cromosomas padre. Al tener una representación mediante permutaciones tenemos que hacerlo con cierto cuidado para que los hijos resultantes sigan siendo soluciones válidas. El proceso es por tanto:

- Seleccionar los padres de manera aleatoria.
- Tomar un padre y copiar la mitad del mismo en el hijo.
- Tomar la segunda parte del otro padre y copiarlo en la segunda mitad del hijo (desde el punto de cruce) de la siguiente manera:
  - Insertar los número que no estén ya insertados en el orden que aparecen en el padre.
  - Cuando lleguemos al final, volvemos al principio del padre.
- El segundo hijo se crea igual intercambiando el papel de los padres.

```
void cruzar(vector<Cromosoma> seleccionados, int tamanio_pob_hijos,  vector<Cromosoma> &
    hijos){
    hijos.resize(tamanio_pob_hijos+1);
    int aniadidos = 0;

    int tam_seleccionados = seleccionados.size();
```

```
    while(aniadidos < tamanio_pob_hijos){
        int i = rand() % tam_seleccionados;
        int j = rand() % tam_seleccionados;

        if(i != j){
            vector<Cromosoma> hijos_aux = crucePadres(seleccionados[i], seleccionados[j
    ]);

            hijos[aniadidos] = hijos_aux[0];
            hijos[aniadidos+1] = hijos_aux[1];

            aniadidos += 2;
        }
    }


    hijos.resize(tamanio_pob_hijos); //Por si nos pasamos al meter más hijos
}
```

Código 6: Selección de padres para el cruce y llamada a cruzarlos.

```
vector<Cromosoma> crucePadres(Cromosoma p1, Cromosoma p2, float k_pctge = 0.5){
    vector<Cromosoma> hijos;
    hijos.resize(2);
    int k = static_cast<int>(k_pctge*size);

    Cromosoma h1, h2;
    h1.permutacion = p1.permutacion;
    h2.permutacion = p2.permutacion;

    int index_1 = k;
    int index_2 = k;
    vector<int>::iterator it1, last1;
    vector<int>::iterator it2, last2;
    last1 = h1.permutacion.begin()+k;
    last2 = h2.permutacion.begin()+k;

    for(int i=k; i<size; ++i){
        //Vemos si podemos rellenar el hijo 1
        it1 = find(h1.permutacion.begin(), last1, p2.permutacion[i]);

        if(it1 == last1){ //Si no lo ha encontrado
            h1.permutacion[index_1] = p2.permutacion[i];
            ++last1;
            ++index_1;
        }

        //Vemos si podemos rellenar el hijo 2
        it2 = find(h2.permutacion.begin(), last2, p1.permutacion[i]);

        if(it2 == last2){ //Si no lo ha encontrado
            h2.permutacion[index_2] = p1.permutacion[i];
            ++last2;
            ++index_2;
        }


    }

    //Volvemos a rellenar el hijo 1 empezando desde el principio del padre
    int index_1_2 = 0;
    last1 = h1.permutacion.begin()+index_1;
    while(index_1 < size){
        it1 = find(h1.permutacion.begin(), last1, p2.permutacion[index_1_2]);

        if(it1 == last1){ //Si no lo ha encontrado
            h1.permutacion[index_1] = p2.permutacion[index_1_2];
            ++index_1;
        }

```

```
50          ++index_1_2;
51      }
52
53      ///Volvemos a rellenar el hijo 2 empezando desde el principio del padre
54      int index_2_2 = 0;
55      last2 = h2.permutacion.begin()+index_2;
56      while(index_2 < size){
57          it2 = find(h2.permutacion.begin(), last2, p1.permutacion[index_2_2]);
58
59          if(it2 == last2){ //Si no lo ha encontrado
60              h2.permutacion[index_2] = p1.permutacion[index_2_2];
61              ++index_2;
62          }
63
64          ++index_2_2;
65      }
66
67      //noBienCruzado(h1);
68      //noBienCruzado(h2);
69
70      hijos[0] = h1;
71      hijos[1] = h2;
72
73
74      return hijos;
75  }
```

Código 7: Obtener los dos hijos a partir de los padres.

2.5. Mecanismos de mutación

El proceso de mutación es bastante sencillo, y nos da la población definitiva de hijos. Lo que hacemos es tomar dos índices aleatorios de cada uno de los hijos y los intercambiamos.

```
1   void mutacion(Cromosoma &crom, int i, int j){
2       int tmp= crom.permutacion[i];
3       crom.permutacion[i] = crom.permutacion[j];
4       crom.permutacion[j] = tmp;
5   }
6
7   void mutar_hijos(vector<Cromosoma> &hijos){
8       srand(time(NULL));
9       int tam_hijos = hijos.size();
10
11
12      for(int index = 0; index<tam_hijos; ){
13          int i = rand() % size;
14          int j = rand() % size;
15
16
17          if(i != j){
18              mutacion(hijos[index], i, j);
19              hijos[index].fitness = fitness(hijos[index].permutacion);
20              ++index;
21          }
22
23      }
24  }
```

Código 8: Mutación de los hijos.

2.6. Mecanismos de reemplazo

Finalmente, la última etapa es la de reemplazo en la que seleccionamos la población que sobrevive a la siguiente generación de entre los padres y los hijos. Para ello, nos podemos basar en la edad, el

fitness o usando ambos (elitismo). Como se ha comentado, para converger al óptimo global es necesario que haya elitismo, es decir, el mejor individuo siempre sobrevive sea padre o hijo. Por ello, vamos a ver dos propuestas, una que toma directamente las mejores soluciones a partir del *fitness* (de padres e hijos) y otra en la que los hijos reemplazan totalmente a los padres, salvo cuando un padre presenta un mejor valor, en cuyo caso reemplaza a uno de los hijos.

```
vector<Cromosoma> reemplazoElitismo(vector<Cromosoma> &padres, vector<Cromosoma> &hijos,
     int tamanio_pob){
    vector<Cromosoma> nueva_poblacion(hijos);

    Cromosoma mejor_padre = mejor(padres).first;
    Cromosoma mejor_hijo = mejor(hijos).first;

    if(mejor_padre.fitness < mejor_hijo.fitness){
        nueva_poblacion[0] = mejor_padre;
    }

    nueva_poblacion.resize(tamanio_pob);
    return nueva_poblacion;
}
```

Código 9: Reemplazo con elitismo.

```
vector<Cromosoma> reemplazoFitness(vector<Cromosoma> &padres, vector<Cromosoma> &hijos,
    int tamanio_pob){
    vector<Cromosoma> nueva_poblacion(padres);

    for(int i = 0; i<hijos.size(); ++i){
        nueva_poblacion.push_back(hijos[i]);
    }


    sort(nueva_poblacion.begin(), nueva_poblacion.end(), miOrden);

    nueva_poblacion.resize(tamanio_pob);
    return nueva_poblacion;
}
```

Código 10: Reemplazo basado en el *fitness*.

Al igual que el caso de la selección de padres, es interesante que la población sea diversa por lo que también hemos creado las variantes de los anteriores introduciendo el peor de los hijos y de los padres (se pueden ver en el archivo `código/src/reemplazo.cpp`).

3. Pruebas algoritmo clásico

Una vez planteadas los procedimientos para cada una de las etapas, vamos a hacer una serie de pruebas a partir de diferentes configuraciones de las mismas. Para ello, usamos el conjunto de prueba disponible `tai256c.dat`. En todos ellos partimos de una población de 50 individuos y durante 2 000 generaciones. Para el torneo, se van a seleccionar un total de 30 padres enfrentados por parejas, es decir, $k = 2$. Las combinaciones y los resultados aparecen en la tabla 1. Aclaramos que cuando marcamos como variante nos referimos a las que hemos notado anteriormente consistentes en tener una mayor variedad en la población incluyendo soluciones peores.

Vemos que los tiempos de ejecución son bastantes parecidos en todos los casos. Sin embargo, hemos obtenido mejores resultados a nivel general cuando el torneo es con reemplazamiento. Además, el mejor resultado se ha obtenido usando la variante indicada de la selección con reemplazamiento junto con la variante del reemplazo basado en el el *fitness*.

Finalmente, en la figura 2 vemos cómo ha sido la evolución en cada uno de los casos. Observamos que los presentan un mayor grado de convergencia son las pruebas 1, 2, 6 y 7.

ID	Selección	Reemplazo	Mejor *fitness*	Tiempo (s)
1	Con reemplazamiento	Basado en *fitness*	45 269 474	91.6559
2	Sin reemplazamiento	Basado en *fitness*	45 278 166	87.7938
3	Sin reemplazamiento	Elitismo	45 868 500	88.9837
4	Sin reemplazamiento (variante)	Elitismo	45 716 578	89.5240
5	Con reemplazamiento (variante)	Elitismo (variante)	45 618 422	89.2145
6	Con reemplazamiento (variante)	Basado en *fitness* (variante)	44 999 918	89.2771
7	Con reemplazamiento	Basado en *fitness* (variante)	45 255208	90.2870

Tabla 1: Resultados tras aplicar distintos procedimientos en un algoritmo genético.

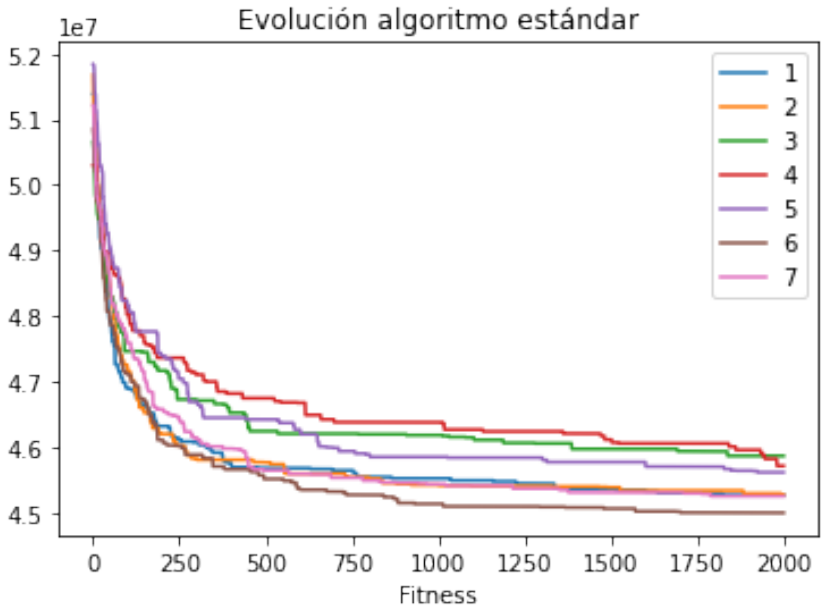


Figura 2: Evolución de los resultados del algoritmo genético clásico.

4. Variantes al algoritmo estándar

En esta sección incluiremos dos variantes del algoritmo estándar aunque tienen una base común. La idea es dotar a a la población de capacidad de aprendizaje. Para ello, utilizamos un algoritmo *greedy* para realizar una búsqueda local partiendo de cada una de los individuos hasta alcanzar un óptimo local. En nuestro caso hemos optado por el algoritmo basado en 2-opt. Es decir, vamos intercambiando dos posiciones y vemos si es mejor que la que teníamos o no. Además, debemos decidir cuántas transposiciones se deben llevar a cabo antes de devolver el resultado. Por las pruebas realizadas, hemos decidido que este grado sea adaptativo. Al principio tenemos una población "peor" por lo que es más fácil encontrar mejores soluciones. Sin embargo, conforme avanzan las generaciones es más complicado. Por ello, partimos devolviendo la solución con diez cambios mientras que a partir de la generación 100 solo llevamos a cabo 1. De este modo, el algoritmo para el caso baldwiniano quedaría como sigue:

```
void greedyBald(vector<Cromosoma> &poblacion, int iteracion){
    int cambio_max = 10;

    if(iteracion > 100){
        cambio_max = 1;
    }

    for(int i=0; i<poblacion.size(); ++i){
        int cambio;

        Cromosoma S = poblacion[i];
```

```
        cambio = 0;

        for(int k=0; k<size && cambio < cambio_max; ++k){
            for(int j=k+1; j<size && cambio < cambio_max; ++j){
                int fT = fitness(S.permutacion, S.fitness, k, j);

                if(fT < S.fitness){
                    Cromosoma T = mutacionNoModifica(S, k, j);
                    S.permutacion = T.permutacion;
                    S.fitness = fT;
                    ++cambio;
                }

            }
        }

        poblacion[i].fitness = S.fitness; //Solo actualizamos el fitness
    }
}
```

Código 11: Optimización en la variante baldwiniana.

Para la lamarckiana sería igual solamente que en vez de actualizar el *fitness*, también actualizamos el valor de la permutación para permitir que lo aprendido se herede de padres a hijos. Además, por las pruebas realizadas se ha visto que un número de intercambios muy alto lleva un alto coste computacional por lo que usamos directamente `cambio_max = 1` para todas las generaciones.

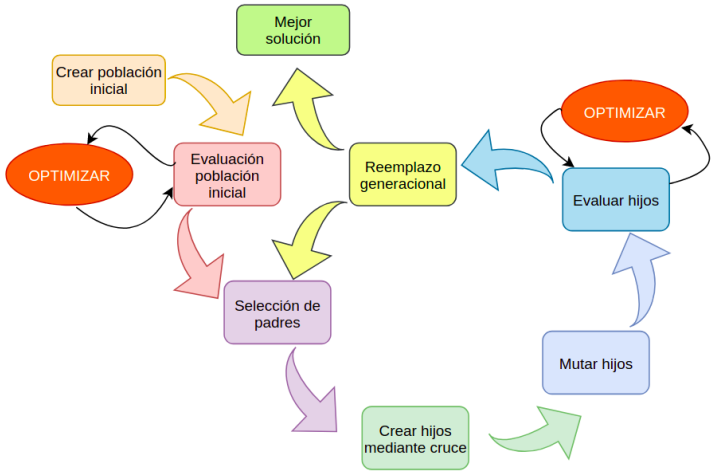


Figura 3: Variante algoritmo genético baldwiniano y lamarckiano.

4.1. Baldwininana

Ahora, llevaremos a cabo las pruebas. Debido a que la carga computacional es este caso es mayor, vamos a reducir el tamaño de la población a un total de 30 individuos y por tanto ahora seleccionamos un total de 18 padres. Tanto el número de generaciones como el torneo por parejas se mantiene. Destacamos que en este caso los mejores valores no son los que correspondería a dicha permutación si no los obtenidos tras la optimización. Las pruebas se muestran en la tabla 2 y la figura 4.

Vemos que algoritmo aprende más rápido que el caso anterior aunque se queda en mínimos locales con mayor frecuencia. Una excepción es el caso del B5 donde vemos cómo es capaz de salir de los mismos.

ID	Selección	Reemplazo	Mejor *fitness*	Tiempo (s)
B1	Con reemplazamiento	Basado en *fitness*	46 737 796	117.945
B2	Sin reemplazamiento	Basado en *fitness*	46 896 390	120.075
B5	Con reemplazamiento (variante)	Elitismo (variante)	45 617 458	233.27
B6	Con reemplazamiento (variante)	Basado en *fitness* (variante)	46 402 690	118.817
B7	Con reemplazamiento	Basado en *fitness* (variante)	46 848 942	211.159

Tabla 2: Resultados tras aplicar distintos procedimientos en un algoritmo genético baldwiniano.

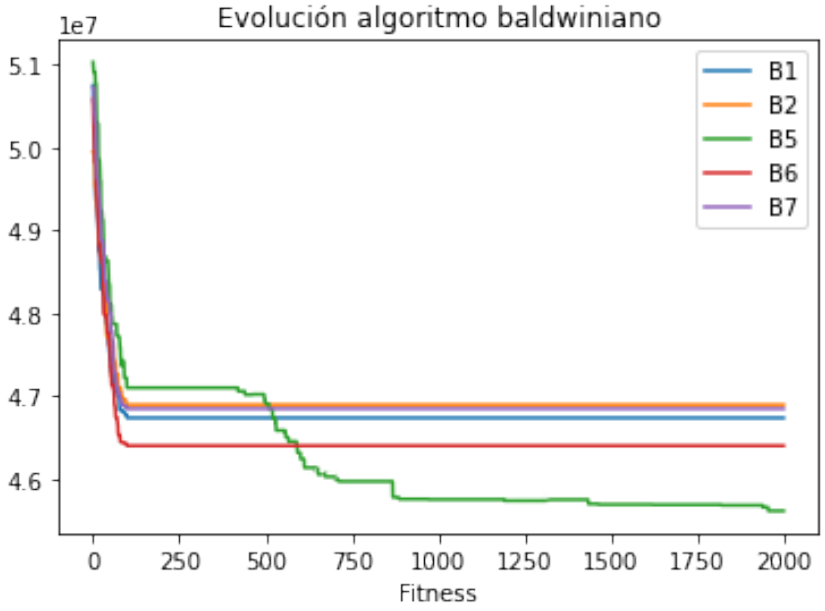


Figura 4: Evolución de los resultados del algoritmo genético baldwiniano.

4.2. Lamarckiana

Pasemos ahora a ver las mismas pruebas anteriores pero con la variante lamarckiana. Como decimos el proceso de optimización es el mismo salvo que también actualizamos la propia permutación y nos quedamos directamente con una solución mejor. Finalmente, también hemos modificado el número de generaciones, reduciendo el mismo a 150. El tamaño de la población sigue siendo 30 y utilizamos torneo seleccionando 18 padres que compiten por parejas.

ID	Selección	Reemplazo	Mejor *fitness*	Tiempo (s)
L1	Con reemplazamiento	Basado en *fitness*	44 943 834	102.3254
L2	Sin reemplazamiento	Basado en *fitness*	44 940 304	98.5581
L5	Con reemplazamiento (variante)	Elitismo (variante)	45 049 750	70.8333
L6	Con reemplazamiento (variante)	Basado en *fitness* (variante)	44 911 776	132.141
L7	Con reemplazamiento	Basado en *fitness* (variante)	44 996 886	172.174

Tabla 3: Resultados tras aplicar distintos procedimientos en un algoritmo genético baldwiniano.

En la tabla 3 y figura 5 vemos cómo hemos obtenido mejores resultados que en los casos anteriores y con un número mucho menor de generaciones. Sin embargo, también observamos cómo el tiempo de ejecución crece considerablemente.

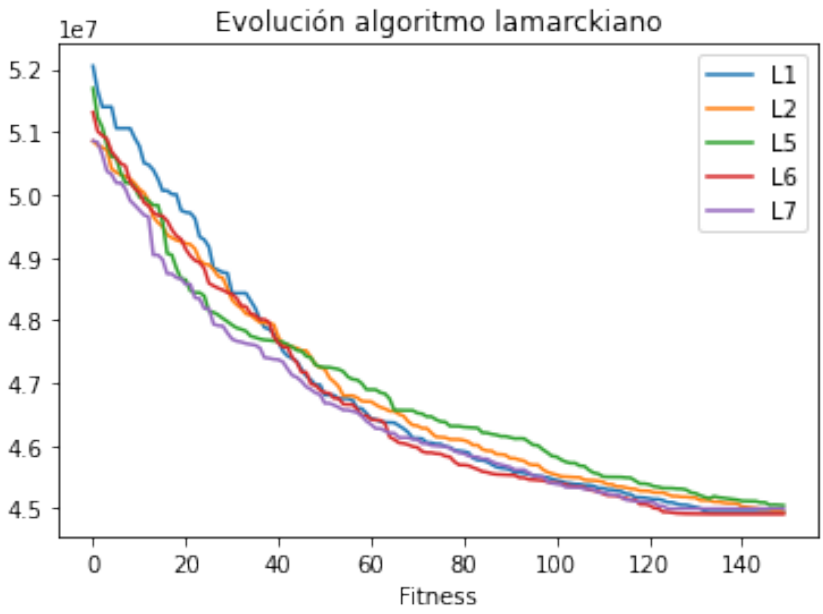


Figura 5: Evolución de los resultados del algoritmo genético lamarckiniano.

5. Comparación y mejores resultados

Una vez hemos hecho las pruebas, veamos gráficamente como se han comportado en cada uno de los casos como mostramos en la figura 6.

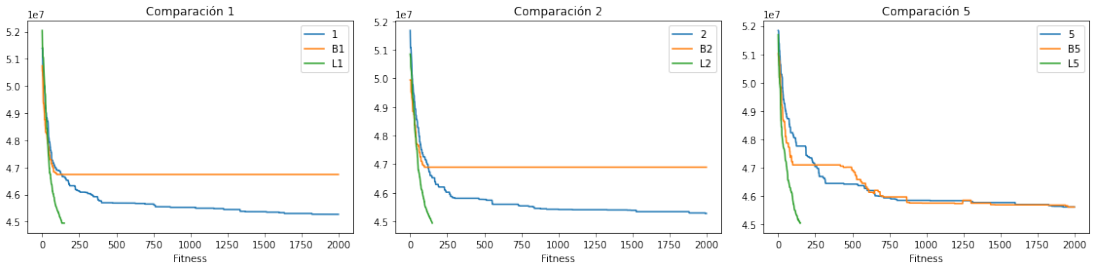
Vemos cómo hemos obtenido los mejores resultados con la variante lamarckiana. Esta ha conseguido mejores resultados incluso con un número menor de generaciones aunque el coste computacional es muy elevado. Por ejemplo, para *L6*, la combinación que mejor resultados nos ha dado, ha tardado 132.141 segundos (150 generaciones). Por su parte, esa misma combinación ha tardado 118.817 segundos en la baldwiniana y 89.2771 segundos en la estándar (con 2 000 generaciones). Por tanto, el mejor costo que se ha obtenido ha sido 44 911 776 que se corresponde con la permutación

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

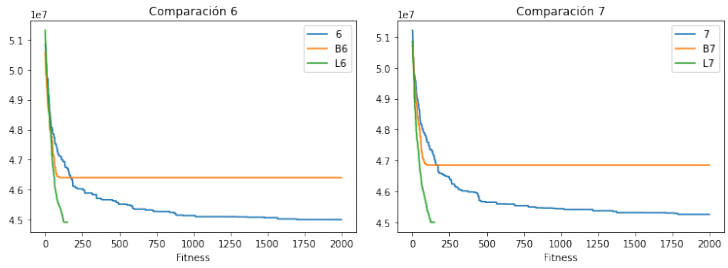
Código 12: Mejor permutación obtenida pra `tai256c` con coste 44 911 776.

6. Pruebas con otros conjuntos

Finalmente, para ver los algoritmos en algunos ejemplos más, vamos a usar otros dos conjuntos de datos disponibles. Más concretamente, vamos a usar `tai100a` y `lipa60a`. En todos los casos vamos a usar una población de tamaño 20 durante 500 generaciones. Usaremos selección de padres por torneo con reemplazamiento compitiendo por parejas hasta obtener un total de 12. El reemplazo de la población se hará mediante elitismo.

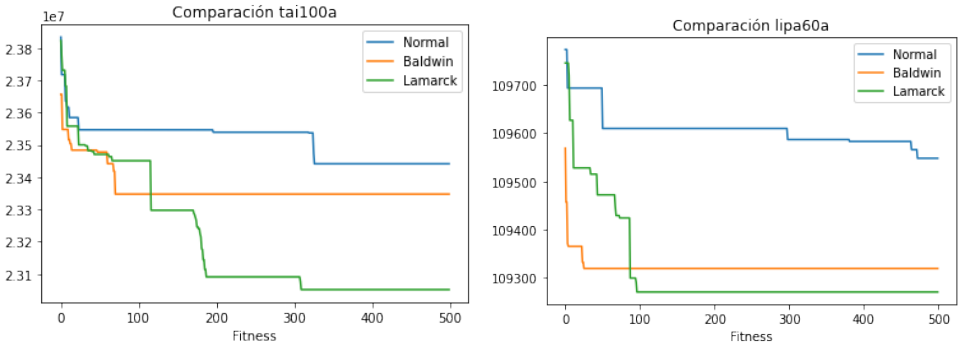


(a) Comparación tipo 1 (b) Comparación tipo 2 (c) Comparación tipo 5



(d) Comparación tipo 6 (e) Comparación tipo 7

Figura 6: Comparación de las evoluciones para cada una de las variantes según las combinaciones de los procedimientos.



(a) `tai100a` (b) lipa60a

Figura 7: Comparación para otros conjuntos de datos.

Vemos (figura 7) cómo también seguimos obteniendo mejores resultados con la variante de lamarck. En la tabla 4 vemos más concretamente los resultados.

7. Conclusiones

A modo conclusión, podemos decir que hemos sido capaces de encontrar soluciones válidas mediante un algoritmo genético. A partir de la versión clásica, hemos visto cómo funcionan algunas variantes del mismo sacadas de teorías de la evolución. La que mejores resultados ha dado ha sido la variante basada en al teoría de Lamarck. En ella, llevamos a cabo un proceso de optimización para obtener un mínimo local de cada una de los soluciones, es decir, la población aprende y lo hereda a sus hijos. Sin embargo, presenta un mayor esfuerzo computacional que las otras dos variantes así como un mayor tiempo de ejecución.

Variante	Coste tai100a	Tiempo tai100a	Coste lipa60a	Tiempo lipa60a
Estándar	23 441 724	2.12034	109 548	0.95541
Baldwin	23 347 764	2.763	109 319	1.81374
Lamarck	23 051 902	2.4064	109 270	1.27632

Tabla 4: Resultados con otros conjuntos de datos.

Bibliografía

1. `Algoritmos Genéticos, Apuntes de la asignatura`, Fernando Berzal.
2. `Práctica de algoritmos evolutivos. Problema de optimización combinatoria QAP`, Fernando Berzal.
3. `Wikipedia: Problema de la asignación cuadrática` https://es.wikipedia.org/wiki/Problema_de_la_asignación_cuadrática. Último acceso 03/01/2021.