Inteligencia Computacional

QAP

Autor

María Matilde Cabrera González

Escuela Técnica Superior de Ingenierías Informática y de Telecomunicación

—

Granada, Enero de 2020

ÍNDICE:

Introducción

El objetivo de esta práctica es resolver un problema de asignación cuadrática utilizando
técnicas de computación evolutiva. El problema de asignación cuadrática (QAP) es un problema fundamental de optimización combinatoria que vamos a aplicar al siguiente problema:

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.

Vamos a implementar las siguientes variantes del problema:

- Algoritmo genético sin optimización local.
- Algoritmo genético con variante lamarckiana.
- Algoritmo genético con variante baldwidiana.

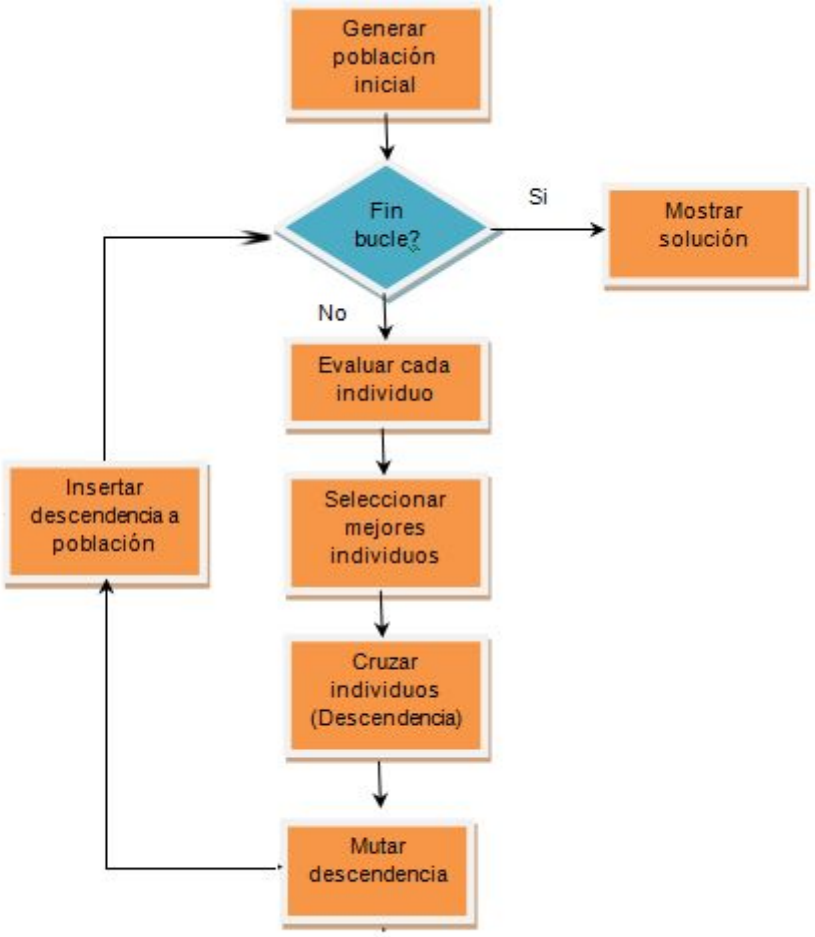
Compararemos los resultados de las variantes mencionadas usando varios archivos para obtener los resultados, aunque el principal archivo para nuestro estudio es “tai256c.dat”.

Implementación

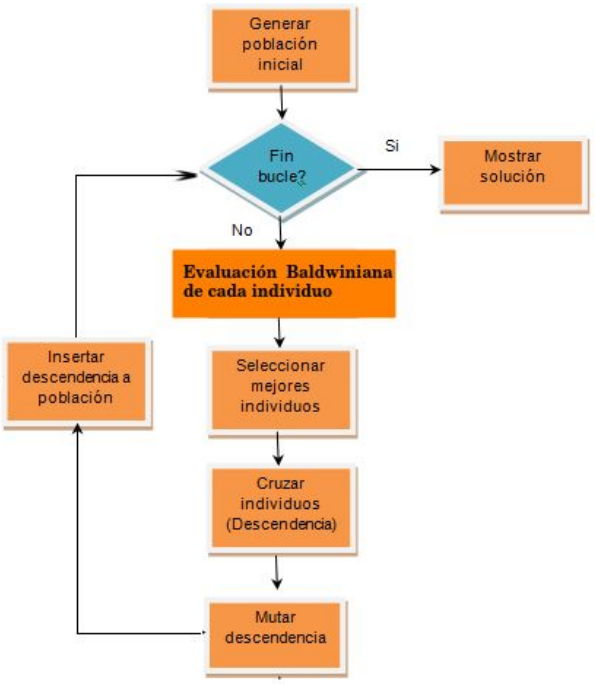
Después de varias pruebas en diferentes lenguajes, la implementación se ha realizado en en el lenguaje de programación c++. No se ha utilizado ningún IDE, ejecutamos desde la misma terminal, como ejemplo de ejecución:

```
g++ -std=c++11 main.cpp -o qap -O2
./qap datos/tai256c.dat 20_individuos 10000_iteraciones normal/lamarck/baldwi
```

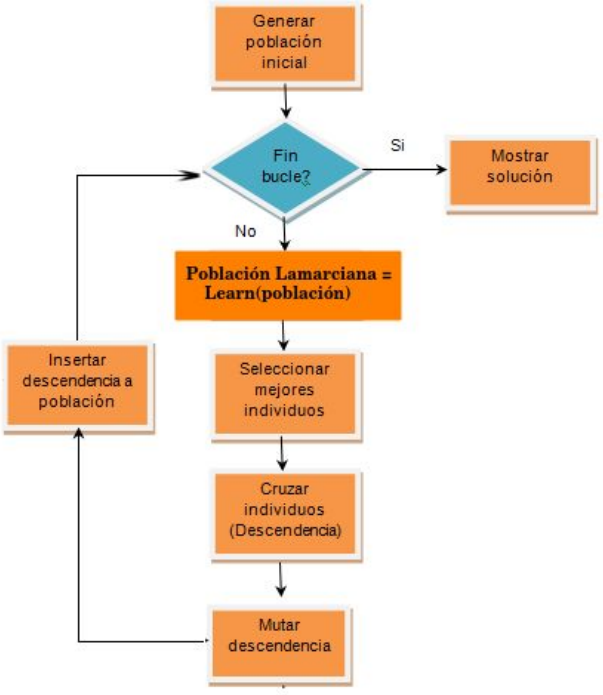
Como ya hemos mencionado el algoritmo genético a realizar consta de tres variantes, todas parten de una misma base, la cual mostramos a continuación:



La diferencia con la variante baldwiniana del algoritmo genético, para evaluar el fitness de cada individuo, se utiliza dicho individuo como punto inicial de una búsqueda local hasta que se alcanza un óptimo local. El valor de ese óptimo local determina el fitness del individuo. Sin embargo, a la hora de formar descendientes, se utiliza el material genético del individuo original.



La diferencia del algoritmo genético con la variante lamarckiana, es que se implementa una función de aprendizaje, en la cual se busca un óptimo local y se reemplaza ese individuo por su evolución optimizada.



Además, hemos repetido el proceso sin cruzar los individuos, los seleccionados mutan y estos pasarían a ser los hijos, después se inserta en la población de la misma manera.

Algoritmo Genético

Generar población inicial

La población inicial se genera de forma aleatoria con tantos individuos como le indiquemos, y el valor de cada gen para el individuo lo determinará el archivo elegido. Los individuos se representan como permutaciones.

Para nuestro algoritmo genético hemos diseñado la siguiente función para crear la solución inicial al problema:

```
void pob_inicial(string& tampob, vector<vector<int> > & poblacion){
        tam = stoi(tampob);
        vector <int> aux;
    for(int i = 0; i < tam; ++i){
       for(int j = 0; j < fileSize; ++j){
                aux.push_back(j);
        }
            std::random_shuffle(aux.begin(), aux.end());
            poblacion.push_back(aux);
            aux.clear();
    }
}
```

En el código mostrado creamos una secuencia de soluciones, aplicamos sobre esta una función aleatoria (Random) para crear diversidad de individuos y los introducimos en una matriz de posibles soluciones.

Fitness

Es la medida de la calidad del individuo, la capacidad de adaptación del individuo, dicho en otras palabras, es la medida de evaluación de una posible solución, a mejor calidad del individuo mejor será la solución que encontremos.

```
int fitness(vector<int> cromosoma,vector<vector<int> > & distancia, vector<vector<int> > & flujo){
        int f;
        f = 0;
        int tamanio;
        tamanio = cromosoma.size();
        for( int i = 0; i < tamanio ; i++) {
                for( int j = 0; j < tamanio ; j++) {
                        f = f + (distancia[i][j] * flujo[cromosoma[i]][cromosoma[j]]);
                }
        }
```

```
	return -f;
}
```

Selección

Hemos creado una función que selecciona los individuos para su reproducción, la selección se hará por torneo, es decir, se propone una probabilidad de selección, en una primera instancia seleccionaremos al 80% de los mejores individuos, estos son los individuos con mejor fitness (aptitud, calidad del individuo).

```
vector<int> seleccion(float probselect,vector<int> &fitness){
	int probseleccion = 0;
	probseleccion = (int) (probselect * tam);
	vector<int> selecc;
	selecc.reserve(probseleccion);
	for( int i = 0; i < probseleccion ; i++) {
		int select1 = 0;
		int select2 = 0;
		int mejorselect = 0;
		select1 = rand() % tam;
		do {
			select2 = rand() % tam;
		}while(select1==select2);

		if(fitness[select1]>fitness[select2]){
			mejorselect = select1;
		}else{
			mejorselect = select2;
		}
		// controla que no haya individuos repetidos, si el seleccionado ya está entre los elegidos damos otra ronda.
		if(std::find(selecc.begin(), selecc.end(), mejorselect) != selecc.end()){
			i--;
		}else{
			selecc.push_back(mejorselect);
		}
	}
	return selecc;
}
```

La selección por torneo tiene un coste computacional muy bajo por su sencillez, elegimos dos individuos al azar, comparamos cuál de ellos tiene menor coste y nos quedamos con el mejor evaluado. Posteriormente comprobamos que ese individuo no haya sido ya seleccionado anteriormente, con esto pretendemos cierto grado de elitismo sin producir una convergencia genética prematura cuando la población es amplia, es decir, que sean los mejores los que se reproducen.

Cruce

El operador cruce representa la reproducción de la población, exploramos las soluciones almacenadas hasta el momento y la combinamos para crear mejores soluciones.

Nos hemos decantado por un operador de cruce en un punto, de forma que obtendremos el primer 30 % de los genes del primero de los progenitores, y el resto del segundo progenitor.

Nuestra función cruce adjudica al hijo la primera parte de la cadena de un progenitor, y posteriormente introduce los del segundo progenitor en el orden en el que se encuentran en el segundo progenitor y excluyendo los que ya tiene el hijo, nos cercioramos que sigamos teniendo una permutación.

```
vector<int> cruce(vector<int> padre1, vector<int> padre2){
	int crucep1 =0;
	crucep1 = padre1.size()/3;
	vector<int> p1;
	p1.reserve(crucep1);
	vector<int> hijo;
	hijo.reserve(padre1.size());
	// introducimos la tercera parte del primer padre
	for (int i = 0 ; i < crucep1 ; i++) {
		hijo.push_back(padre1[i]);
		p1.push_back(padre1[i]);
	}
	// introducimos el resto en el orden del padre 2
	for (int i = 0 ; i < padre2.size() ; i++) {
		if(std::find(p1.begin(), p1.end(), padre2[i]) == p1.end()) {
			hijo.push_back(padre2[i]);
		}
	}
	p1.clear();
	return hijo;
}
```

Mutación

La mutación representa la evolución de la población. Este operador proporciona diversidad entre los individuos de la población, para evitar la temprana convergencia hacia óptimos locales, los individuos al mutar añade diversificación al método de cruce. Pretendemos obtener nuevas soluciones a partir de una solución existente para no quedarnos en una solución local.

La mutación se hará por intercambio de dos posiciones al azar. Se eligen dos posiciones y se intercambian.

```
void mutacion(vector<int> & mutado){
	int azar1, azar2;
	azar1 = rand() % mutado.size();
	azar2 = rand() % mutado.size();
	if (azar1 == azar2) {
		if (azar2 > (mutado.size() - 1)) {
		azar2 = azar2 - 1;
		}
		if (azar2 < 1) {
		azar2 = azar2 + 1;
		}
	}
	swap(mutado[azar1],mutado[azar2]);
}
```

Reemplazo

En cada iteración reemplazamos la población manteniendo el mismo número de individuos, vamos a conservar siempre el mejor de los progenitores, de entre el resto nos quedaremos con los que tengan mejor fitness y por último cogeremos el peor de los hijos. De esta manera mantenemos la diversidad mientras avanzamos en nuestra búsqueda de la mejor solución.

```
void reemplazo(vector<int> & mejor_individuo,  vector<int> & fitness, vector<vector<int> > & popu,
vector<vector<int> > & hijos){
	vector<vector<int> >	nuevapoblacion;
	// conservamos la mejor solución encontrada hasta el momento
	nuevapoblacion.push_back(mejor_individuo);
	//nueva población con el mejor y el peor de los padres
	int peor;
	peor = 0;
	int fitnesspeor = fitness[0];
	for(int j = 0 ; j < tam ; j++) {
		if(fitness[j]< fitnesspeor){
			peor = j;
			fitnesspeor = fitness[j];
		}
	}
	nuevapoblacion.push_back(popu[peor]);

	//introduzco los hijos
	for(int j = 0 ; j < hijos.size() ; j++) {
		if(nuevapoblacion.size()<popu.size()){
			nuevapoblacion.push_back(hijos[j]);
		}
	}
	//relleno de los padres hasta completar tamaño
```

```
	int a = 0;
	while(nuevapoblacion.size() < popu.size()){
		nuevapoblacion.push_back(popu[a]);
		a++;
	}

	//reemplazo poblacion
	popu = nuevapoblacion;
	nuevapoblacion.clear();
}
```

Con un reemplazo elitista tratamos de evitar que en el transcurso del algoritmo, individuos con buenas cualidades pueden no generar descendencia, perdiendo de esta forma su información genética y con ella un buen resultado.

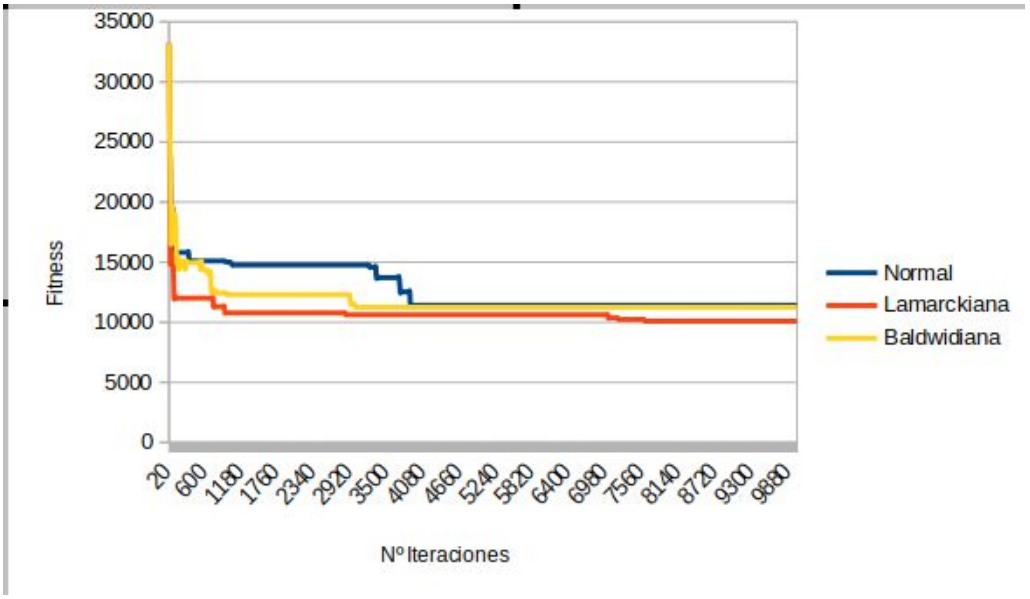
Resultados

Se ha pedido que hagan varias pruebas y que se centren las soluciones para el problema “tai256c.dat”. Para nuestro estudio hemos seleccionado varios problemas para comparar soluciones.

Para obtener los resultados que se mostraran a continuación, se han elegido los siguientes parámetros:

- Hemos definido la población en 20 individuos. A partir de ahí la solución empeora.
- Iteraciones 10000.
- Variantes :
  - normal(sin optimización)
  - lamarck
  - baldwi.
- Probabilidad de selección 80%.
- Selección por torneo
- Mutación por intercambio en un punto
- Cruce en un punto.

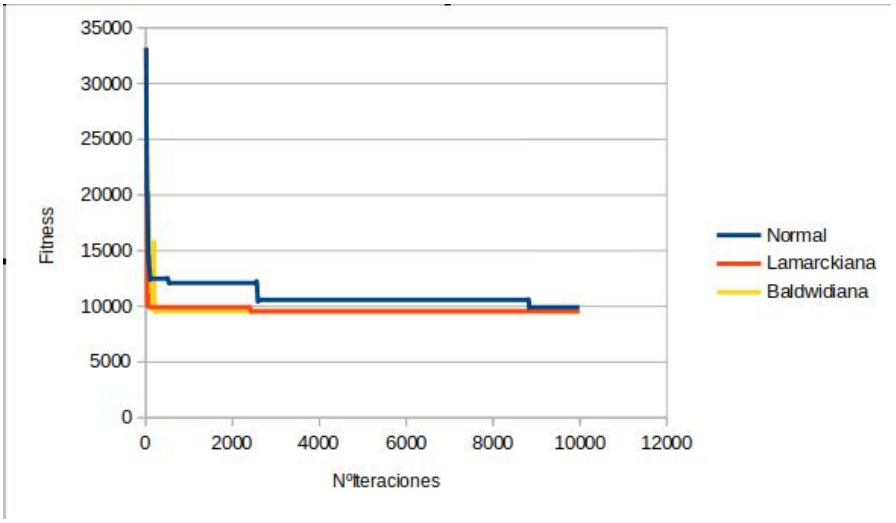
En primer lugar evaluamos los resultados para chr12a.dat, en primer lugar generamos la solución introduciendo el cruce en el algoritmo:



Mejores soluciones encontradas:

normal	11414
lamarck	10096
baldwi	11248

Evaluamos la misma semilla sin cruzar los individuos:

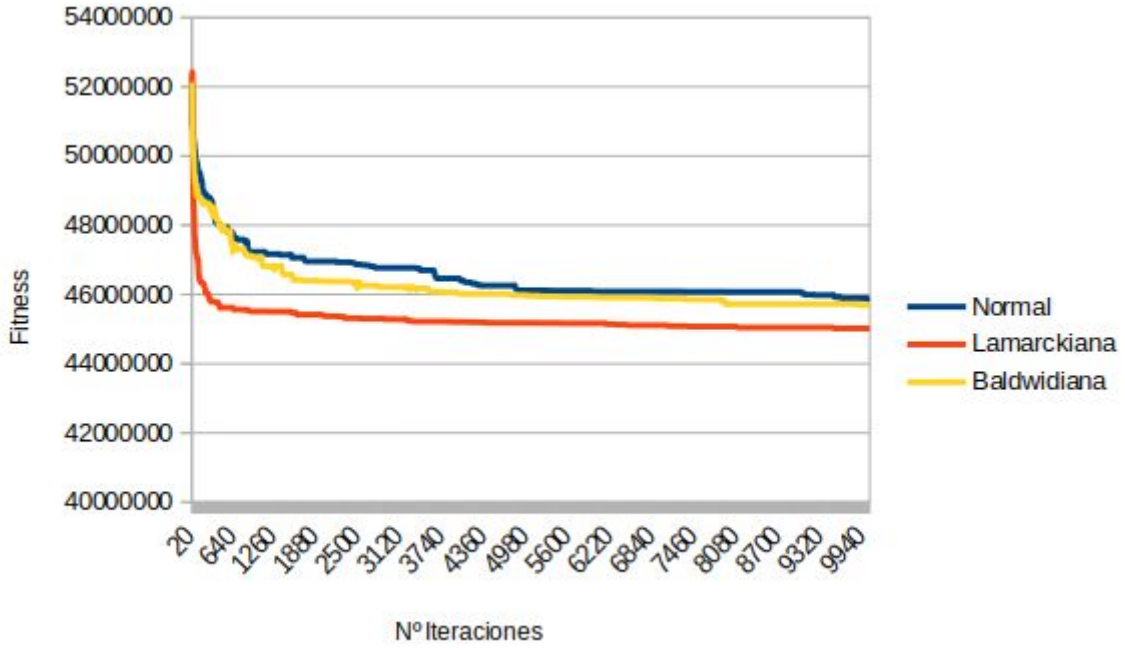


Mejores soluciones encontradas:

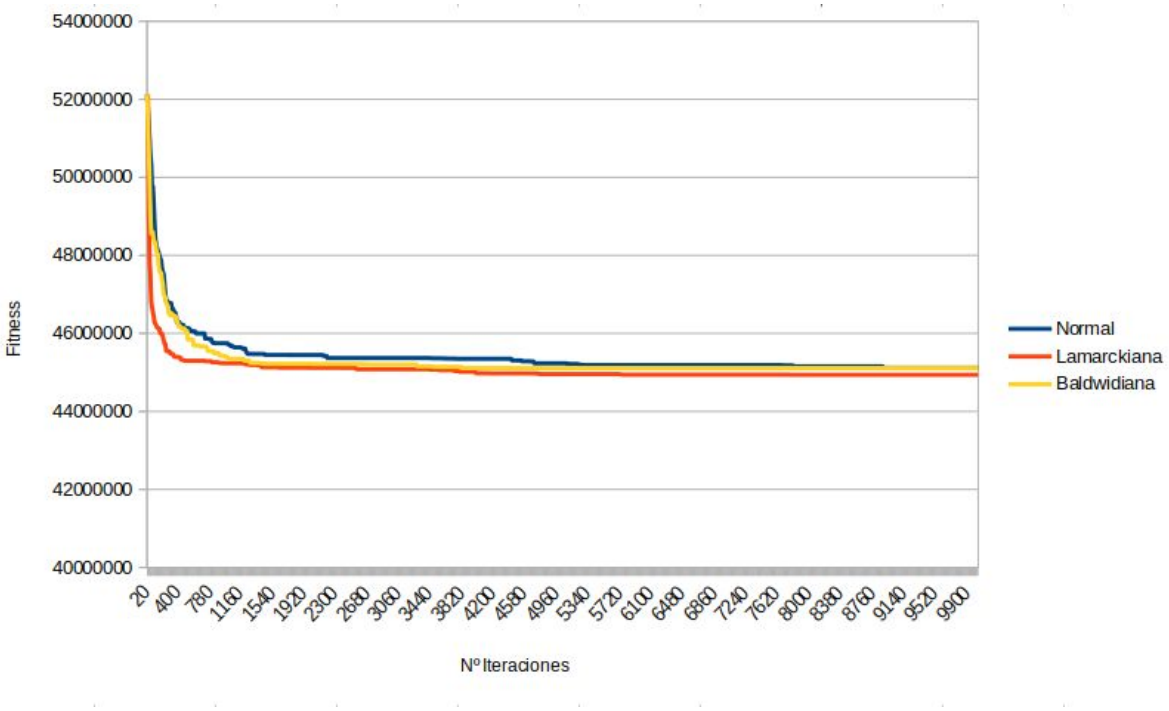
normal	9916
lamarck	9552
baldwi	9552

Para ser un problema de pocas dimensiones, hay una diferencia significativa. Sin cruce los algoritmos encuentran una solución óptima mejor más rápidamente.

Vamos a evaluar de la misma manera el problema que se pide solucionar para este proyecto, tai256c.dat, en primer lugar usamos el algoritmo que incluye la función de cruce.



Hacemos lo mismo para el algoritmo que no incluye la función de cruce.



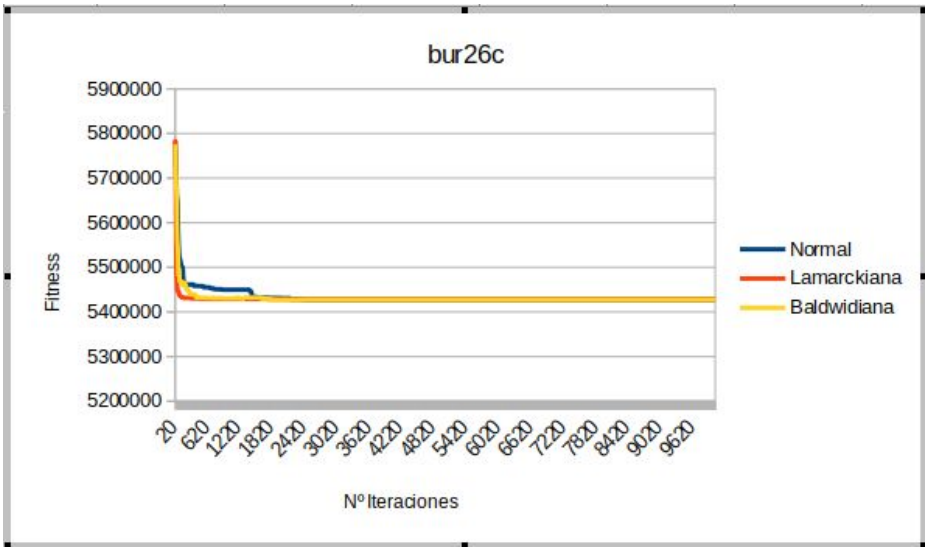
Comparamos los resultados obtenidos. Cómo obtenemos mejores resultados y más rápidamente, seguimos nuestro estudio con el mejor de nuestros algoritmos:

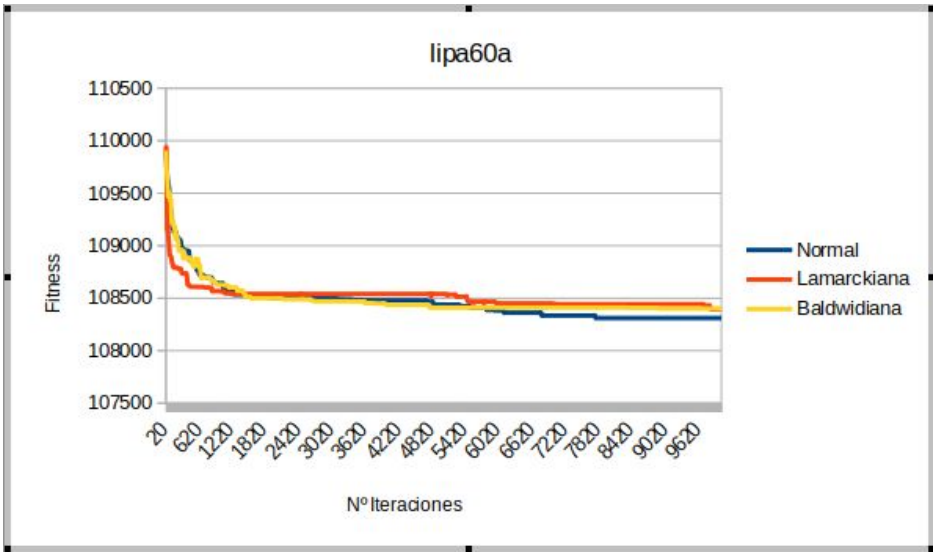
	Con función cruce	Sin función cruce
Sin optimizar	45861546	45117232
Lamarckiana	45017912	44938492
Baldwidiana	45693350	45110264

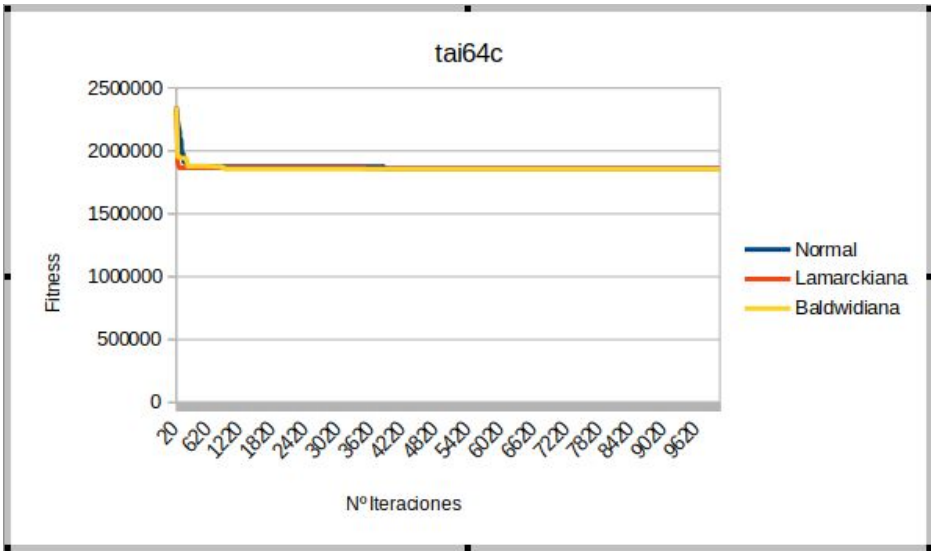
Lo que se pretende con el método de cruce es que no obtengamos descendientes mucho peores que los padres, pero puede tener adaptaciones completamente aleatorias.

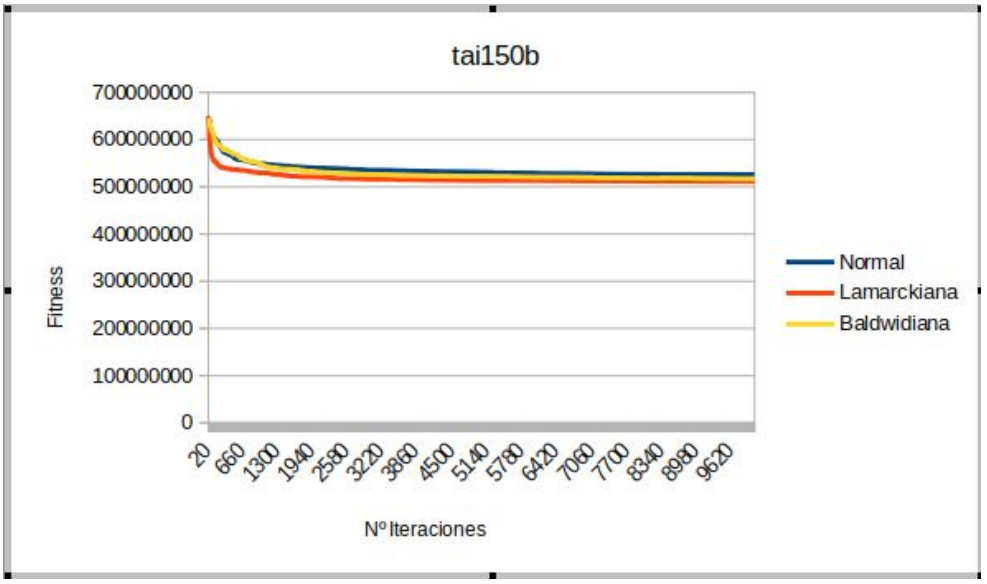
Según nuestro estudio el efecto del cruce en la búsqueda es inferior al que se esperaba, así que vamos a usar una evolución primitiva, en la cual solo se consta de selección y mutación, hay estudios que demuestran que esta evolución primitiva supera con creces a una evolución basada exclusivamente en selección y cruce.

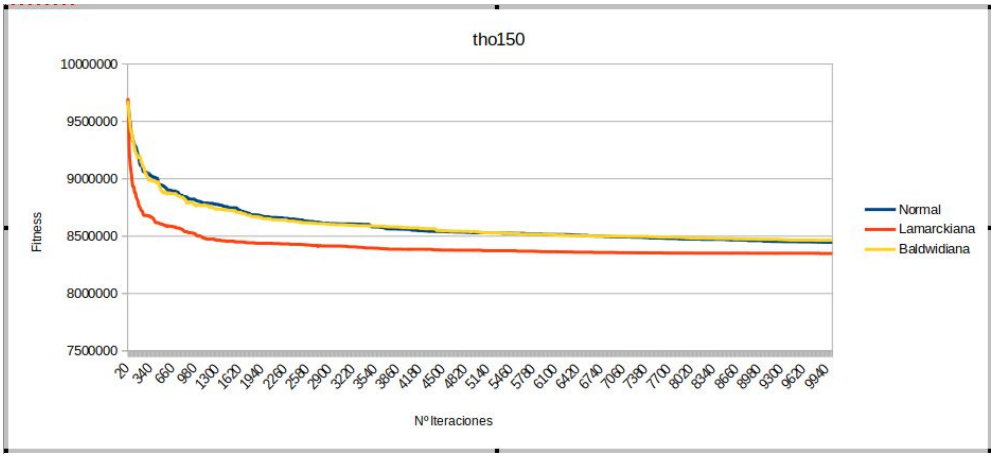
Vemos el comportamiento de los algoritmos con diferentes problemas:











En todos los resultados excepto en lipa60, en la variante Lamarckiana se observan mejores resultados, en todas las pruebas se obtiene un buen resultado relativamente pronto y luego su progreso es inapreciable.

Tabla comparativa

	Sin optimización	Lamarckiana	Baldwidiana
chr12a	9916	9552	9552
bur26c	5427184	5428356	5427346
lipa60a	108311	108395	108402
tai64c	1860702	1859480	1855928
tai150b	525774508	511283606	516937005
tai256c	45117232	44938492	45110264
tho150	8446782	8348626	8463504

Conclusiones

Usamos una evolución primitiva, en la cual solo se consta de selección y mutación, hay estudios que demuestran que esta evolución primitiva supera con creces a una evolución basada exclusivamente en selección y cruce. En todo caso nuestro algoritmo empeoraba con la función de cruce en un punto, deberíamos haber optado por un cruce en el que se mantengan los genes que coincidan en ambos padres.

Hemos realizado pruebas con diferentes tamaños de población y diferente cantidad de iteraciones, fijándonos más exhaustivamente en el problema tai256c, con ello hemos observado que con más de 20 individuos siempre nos devuelve una solución peor hasta la encontrada en el momento, con un número de iteraciones bajo, nos devuelve una mejor solución cuando elegimos menos individuos, como por ejemplo las mejores soluciones que mostramos a continuación.

Mejor solución

Destacamos las dos mejores soluciones obtenidas:

Variante Lamarckiana
Nº de individuos: 10
Iteraciones del algoritmo: 60000
Coste de la mejor solución: 44913762
Tiempo de ejecución: 2.57698e+08 segundos.

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

Individuo de la mejor solución obtenida:

Variante Lamarckiana
Nº de individuos: 20
Iteraciones del algoritmo: 100000
Coste de la mejor solución: 44899588
Tiempo de ejecución: 4.29495e+08 segundos.

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