UNIVERSIDAD DE GRANADA

Práctica de algoritmos evolutivos

QAP: Problemas de optimización combinatoria

Autor
Juan Manuel Castillo Nievas

Máster Profesional en Ingeniería Informática

—

Granada, 1 de febrero de 2021

Índice

1. Introducción

En este documento se va a presentar la resolución del problema de la asignación cuadrática utilizando técnicas de computación evolutiva.

El objetivo del problema de la asignación cuadrática (QAP) es asignar un conjunto de instalaciones a un conjunto de localizaciones de manera que se minimice el coste total de asignación. Este coste es una función entre el flujo de materiales que tiene que existir entre las distintas instalaciones y las distancias que hay entre las localizaciones.

En la Figura 1 se puede visualizar un ejemplo que muestra gráficamente el objetivo de este problema [1].

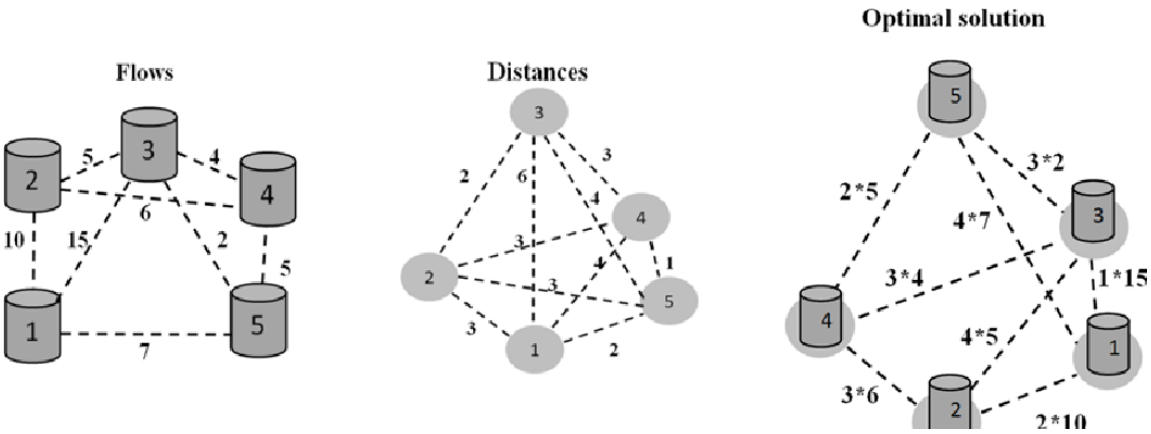

Figura 1: Ejemplo del problema de la asignación cuadrática

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`, se ha de encontrar la asignación de instalaciones a localizaciones que minimice la función de coste:

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

Para este problema se ha utilizado el dataset **tai256c** [2].

2. Entorno de trabajo

En una primera versión se ha implementado el problema usando **Python** como lenguaje de programación y **Sublime Text** [3] como editor de texto de código. Se implementó la versión `Standard` y se hicieron varias pruebas con distintos parámetros. El problema llegó cuando se implementó la versión `Baldwiniana` y `Lamarckiana` con sus respectivas técnicas greedy de optimización local. Tras 4 horas de ejecución, el algoritmo había conseguido avanzar apenas 1 iteración. Esto se debe a que es un lenguaje de programación interpretado.

Tras ver la cantidad de horas que se necesitaría para hacer distintas pruebas y obtener un resultado, se decidió implementar el problema en **Java** usando **Intellij** [4] como IDE.

En los resultados que se muestran en la Sección 4 se puede comprobar la diferencia de tiempo que hay entre las ejecucios en Python y las ejecuciones en Java.

Resumiendo en una lista el entorno de trabajo:

- **Python 3.6.9** como lenguaje de programación en una primera implementación del problema
- **Java 11.0.9.1** como lenguaje de programación en una segunda y última implementación del problema
- **Sublime Text 3** como editor de texto para la implementación en Python
- **IntelliJ IDEA** como IDE para la implementación en Java

3. Implementación

Toda la implementación que se va a comentar a continuación está subida en mi repositorio de **Github**:

`https://github.com/Jumacasni/IC/tree/main/QAP`

En el repositorio se encuentra tanto la versión en Python como la versión en Java, aunque durante esta sección se va a explicar la implementación del algoritmo genético en Java ya que es la versión que tarda menos tiempo en ejecutarse y que obtiene el mejor resultado.

3.1. Lectura del fichero tai256c.data

Para la lectura de matrices de flujo y distancias de este dataset se ha creado una clase llamada `Reader` que tiene tres atributos:

- `flow_matrix:` es una matriz de 256x256 que contiene la matriz de flujos.
- `distance_matrix:` es una matriz de 256x256 que contiene la matriz de distancias.
- `size`: el tamaño de las matrices y el número de cromosomas que tiene cada individuo. En este dataset su valor es 256.

Su constructor recibe un `String` como parámetro que es el nombre del fichero que se debe parsear.

Esta es la primera clase que se crea. A partir de este momento se puede acceder a las matrices en cualquier momento de la ejecución.

```
public class Reader {
    private static int flow_matrix[][];
    private static int distance_matrix[][];
    private static int size;

    public Reader(String file) {
        ...
    }
}
```

3.2. Representación de individuos

Para la representación de individuos se ha creado una clase llamada **texttt** que tiene dos atributos:

- `chromosomes:` es un vector de enteros que contiene los cromosomas del individuo. Con este dataset en concreto, cada individuo tiene 256 cromosomas y el vector contiene los números del 0 al 255 de manera única, es decir, que ningún elemento se puede repetir.
- `fitness:` contiene la evaluación del individuo, es decir, el coste total de la asignación.

Consta de dos constructores: el primero recibe un tamaño que es el número de cromosomas del individuo y un booleano que indica si el individuo se va a inicializar como vacío (rellenando todos sus elementos a -1) o si se quiere inicializar con una solución aleatoria. El segundo constructor es un constructor de copia que crea un individuo a partir de otro que se le pasa como parámetro.

```
public class Individual {
    private int[] chromosomes;
    private int fitness;

    public Individual(int size, boolean empty){
        ...
    }

    public Individual(Individual i){
        ...
    }
}
```

3.3. Representación de la población

Para la representación de la población se ha creado una clase llamada `Population` que tiene un atributo:

- `individuals:` es un vector de individuos que contiene todos los individuos que forman parte de la población.

Su constructor recibe un parámetro que es el tamaño de la población que se va a crear.

```
public class Population {
    private Individual[] individuals;

    public Population(int size){
        ...
    }
}
```

Adicionalmente se han creado dos funciones, `getFittest` y `getWorst`, que devuelven el mejor y peor individuo de la población, respectivamente.

```
public Individual getFittest(){
    ...
}

public Individual getWorst(){
    ...
}
```

3.4. Función fitness

Para calcular el fitness de cada individuo se usa una función dentro de la clase `Individual` llamada `calculateFitness`.

```
public void calculateFitness(){
    int totalFitness = 0;
    for(int i=0; i<Reader.getFlowMatrix().length; ++i){
        for(int j=0; j<Reader.getFlowMatrix().length; ++j){
            int chromosomeI = this.chromosomes[i];
            int chromosomeJ = this.chromosomes[j];

            totalFitness += Reader.getFlowMatrix()[i][j] *
                Reader.getDistanceMatrix()[chromosomeI][chromosomeJ];
        }
    }

    if(totalFitness < 44759294){
        System.out.println("Error al calcular el fitness");
    }
```

```
    this.fitness = totalFitness;
}
```

Lo que se hace es calcular el coste total de la asignación usando las matrices de flujo y de distancias. También hay una sentencia `if` que comprueba si el fitness obtenido es mejor que el record mundial actual, ya que se ha supuesto que si se supera este record es que el fitness se ha calculado mal.

3.5. Algoritmo genético: versión Standard

Se ha creado una clase llamada `GeneticAlgorithm` que contiene todo el proceso del algoritmo genético. Esta clase cuenta con los siguientes parámetros:

- `POPULATION_SIZE`: es el tamaño inicial de la población.
- `N_ITERATIONS`: es el número de iteraciones que se ejecutarán, es decir, el número de generaciones que habrá en toda la ejecución.
- `TOURNAMENT_SIZE`: es el número de individuos que se seleccionan aleatoriamente en la selección por torneo.
- `MUTATION_PROBABILITY`: es la probabilidad de mutación de un individuo.
- `version`: indica la versión que se va a ejecutar (`Standard`, `Baldwiniana` o `Lamarckiana`).

```
public class GeneticAlgorithm {

    private int POPULATION_SIZE;
    private int N_ITERATIONS;
    private int TOURNAMENT_SIZE;
    private double MUTATION_PROBABILITY;
    private String version;

    public GeneticAlgorithm(String t, int p, int i, int ts, double m){
        ...
    }
```

Esta clase tiene un método llamado `start` que es el encargado de realizar todas las iteraciones siguiendo el modelo de un algoritmo genético. El algoritmo implementado sigue los siguientes pasos:

1. Creación de la población inicial de tamaño `POPULATION_SIZE`
2. Se recorre `N_ITERATIONS`:
   a) Creación de la población con los descentiendes de la generación anterior con un tamaño de `POPULATION_SIZE*2`.
   b) Se recorre `POPULATION_SIZE`:
      - Selección por torneo de dos padres
      - Creación de dos hijos mediante el cruce de los padres
      - Mutación de los hijos con una probabilidad de `MUTATION_PROBABILITY`
      - Añadir los hijos a la nueva población creada
   c) Calcular el fitness de todos los individuos de la nueva población
   d) Reemplazar el peor individuo de la nueva población por el mejor individuo de la población anterior
   e) Volver al punto 2 con la nueva población creada

```
public Individual start(){
    Population currentPopulation = createPopulation(POPULATION_SIZE);

    for(int i=0; i<N_ITERATIONS; ++i){
        Population newPopulation = new Population(POPULATION_SIZE*2);
        System.out.println("Iteracion
            "+String.valueOf(i+1+"/"+String.valueOf(N_ITERATIONS)));

        for(int j=0; j<POPULATION_SIZE; ++j){
            Individual parent1 = tournamentSelection(currentPopulation);
            Individual parent2 = tournamentSelection(currentPopulation);
            // Para evitar que se escoja el mismo padre
            while(parent1.equals(parent2.getChromosomes())){
                parent2 = tournamentSelection(currentPopulation);
            }

            Individual[] children = uniformLikeCrossover(parent1, parent2);

            swapMutation(children[0]);
            swapMutation(children[1]);

            newPopulation.addIndividual(children[0], 2*j);
            newPopulation.addIndividual(children[1], 2*j+1);
        }

        calculateAllFitness(newPopulation);

        Individual worstIndividual = newPopulation.getWorst();
        int indexWorst =
            Arrays.asList(newPopulation.getIndividuals()).indexOf(worstIndividual);
        Individual bestIndividual = currentPopulation.getFittest();
```

```
        newPopulation.addIndividual(bestIndividual, indexWorst);

        currentPopulation = newPopulation;
    }

    return currentPopulation.getFittest();
}
```

3.5.1. Selección por torneo

Para la selección de padres para el cruce se ha usado la selección por torneo. Esta selección consiste en escoger aleatoriamente `TOURNAMENT_SIZE` individuos de la población y de esos individuos escoger el que mejor fitness tenga.

```
private Individual tournamentSelection(Population population){
    Population tournament = new Population(TOURNAMENT_SIZE);
    Random rnd = new Random();
    Individual pickIndividual = null;
    ArrayList<Integer> pickedNumbers = new ArrayList<Integer>();
    for(int i=0; i<TOURNAMENT_SIZE; ++i){
        int intRandom = rnd.nextInt(population.getIndividuals().length);
        pickIndividual= population.getIndividual(intRandom);
        // Para evitar que un individuo este repetido
        while (pickedNumbers.contains(intRandom)){
            intRandom = rnd.nextInt(population.getIndividuals().length);
            pickIndividual = population.getIndividual(intRandom);
        }
        tournament.addIndividual(pickIndividual, i);
        pickedNumbers.add(intRandom);
    }

    return tournament.getFittest();
}
```

3.5.2. Operador de cruce

Para el operador de cruce se ha usado un algoritmo que fue propuesto por Tate y Smith en 1995 [5] y se le llama **Uniform Like Crossover** (ULX). Este mecanismo se puede ver como un cruce uniforme adaptado a soluciones basadas en permutaciones, como es nuestro caso. Funciona de la

siguiente manera:

- Todos los cromosomas que sean iguales y que además coincidan en la posición de ambos padres se copian al hijo.
- Se recorren de izquierda a derecha las posiciones que no han sido asignadas. Si el cromosoma no está en el hijo, se escoge aleatoriamente el cromosoma de alguno de los dos padres y se inserta en el hijo.
- Los cromosomas que no han sido añadidos se insertan aleatoriamente en el hijo

En la Figura 2 se muestra un ejemplo de cruce de dos padres siguiendo este mecanismo.

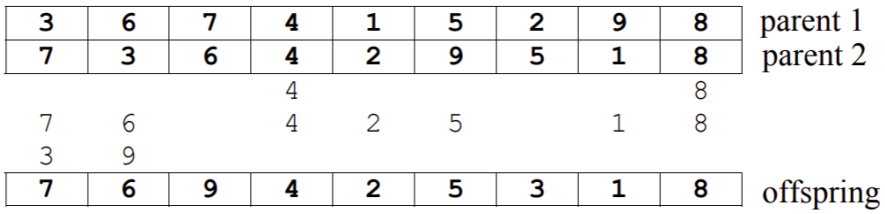

Figura 2: Ejemplo del ULX [5]

En el código se utilizan una función para el cruce llamada `uniformLikeCrossover` que a su vez llama a la función `generateChild` que es la que contiene el algoritmo **ULX**.

```
private Individual[] uniformLikeCrossover(Individual parent1, Individual parent2){

    Individual[] children = new Individual[2];

    children[0] = generateChild(parent1, parent2);
    children[1] = generateChild(parent1, parent2);

    return children;
}

private Individual generateChild(Individual parent1, Individual parent2){
    // Algoritmo ULX
    ...
}
```

3.5.3. Operador de mutación

Para la mutación de los hijos se ha usado lo que se conoce como **swap mutation**. En esta operacion se seleccionan dos posiciones aleatorias del individuo y se intercambian los cromosomas. Esto sólo ocurre con una cierta probabilidad definida por `MUTATION_PROBABILITY`.

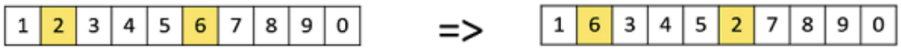

Figura 3: Ejemplo de *swap mutation* [6]

```
private void swapMutation(Individual individual){
    double prob = Math.random();
    Random rnd = new Random();

    if(prob < MUTATION_PROBABILITY){
        int indexChromosome1 = rnd.nextInt(individual.getChromosomes().length);
        int indexChromosome2 = rnd.nextInt(individual.getChromosomes().length);
        while(indexChromosome1 == indexChromosome2){
            indexChromosome2 = rnd.nextInt(individual.getChromosomes().length);
        }

        individual.swap(indexChromosome1, indexChromosome2);
    }
}
```

3.5.4. Mecanismo de reemplazo

En este mecanismo de reemplazo se utiliza un modelo elitista, esto es, se selecciona el peor individuo de la nueva población creada y se reemplaza por el mejor individuo de la anterior población.

```
Individual worstIndividual = newPopulation.getWorst();
int indexWorst = Arrays.asList(newPopulation.getIndividuals()).indexOf(worstIndividual);
Individual bestIndividual = currentPopulation.getFittest();

newPopulation.addIndividual(bestIndividual, indexWorst);
```

3.6. Versión Baldwiniana

En esta versión lo único que cambia es la forma de evaluar el fitness de un individuo. Ahora lo que se hace es realizar una búsqueda local que parte del individuo hasta alcanzar un óptimo local. Dicho óptimo local es el nuevo fitness del individuo. Sin embargo, sus cromosomas no cambian y siguen siendo los mismos que los del individuo general.

Para la búsqueda del óptimo local se ha usado el **algoritmo greedy 2-opt** con una estrategia de transposición. La idea es comenzar con una permutación inicial e ir intercambiando los cromosomas del individuo.

Algorithm 1: Algoritmo greedy 2-opt

```
S = individuo inicial;
best = S;
for i ← 1 to n do
    for j ← i + 1 to n do
        T = S tras intercambiar i con j;
        if fitness(T) ¡fitness(S) then
            S = T;
        end
    end
end
```

Esta implementación difiere de la mostrada en el guión de prácticas en el bucle `while` ya que, al implementarlo con la condición `S.fitness < this.fitness`, el coste computacional ascendía notoriamente y el tiempo de ejecución se disparaba.

El método que implementa este algoritmo se llama `calculateFitnessBaldwinian` y está implementado en la clase `Individual`.

```
public void calculateFitnessBaldwinian(){
    Individual best = new Individual(this);
    int bestFitness = best.getFitness();

    for(int i=0; i<best.getChromosomes().length; ++i) {
        for (int j = i + 1; j < best.getChromosomes().length; ++j) {
            Individual T = new Individual(best);
            T.setChromosome(best.getChromosomes()[j], i);
            T.setChromosome(best.getChromosomes()[i], j);
            int optFitness = T.calculateOptFitness(T.getChromosomes());

            if (optFitness < best.getFitness()) {
```

```
                best.fitness = optFitness;
                best.chromosomes = T.getChromosomes();
                this.fitness = optFitness;
            }
        }
    }
}
```

3.7. Versión Lamarckiana

Esta versión sigue el mismo esquema de la versión Baldwiniana con la única diferencia de que en esta versión los cromosomas del individuo original sí cambian y son reemplazados por los cromosomas que obtienen el mejor fitness durante la búsqueda del óptimo local.

El método que implementa este algoritmo se llama `calculateFitnessLamarckian` y está implementado en la clase `Individual`.

```
public void calculateFitnessLamarckian(){
    Individual best = new Individual(this);

    for(int i=0; i<best.getChromosomes().length; ++i){
        for(int j=i+1; j<best.getChromosomes().length; ++j){
            Individual T = new Individual(best);
            T.setChromosome(best.getChromosomes()[j],i);
            T.setChromosome(best.getChromosomes()[i],j);
            int optFitness = T.calculateOptFitness(T.getChromosomes());

            if(optFitness < best.getFitness()){
                best.fitness = optFitness;
                best.chromosomes = T.getChromosomes();
                this.fitness = optFitness;
                this.chromosomes = best.getChromosomes();
            }
        }
    }
}
```

4. Resultados

4.1. Versión en Python

Tal y como se ha comentado anteriormente en este documento, en una primera versión se implementó este problema en Python y se obtuvieron resultados para la versión `Standard` con diferentes parámetros. El código se ha subido a mi repositorio de **Github** por si se quiere saber cómo se ha implementado:

`https://github.com/Jumacasni/IC/tree/main/QAP/Python`

En la Figura 4 se muestra una tabla con el coste de la solución y el tiempo empleado en segundos para diferentes parámetros. Tal y como se ve en la columna del tiempo empleado, para ser una versión `Standard` se consume demasiado tiempo.

En las versiones `Baldwiniana` y `Lamarckiana` fue imposible obtener solución alguna después de 4 horas de ejecución.

La mejor solución obtenida en esta versión es la siguiente:

- **Tamaño de la población:** 100
- **Número de iteraciones:** 500
- **Tamaño del torneo:** 10
- **Probabilidad de mutación:** 0.1
- **Coste de la solución:** 45434544
- **Tiempo de ejecucion:** 1473.88 segundos

Tamaño población	Iteraciones	Tamaño del torneo	Prob. mutación	Coste de la solución	Tiempo empleado (s)
20	100	5	0.05	46948136	71.73
50	100	10	0.05	46528586	125.75
100	100	10	0.05	46039430	252.88
100	300	10	0.05	45808552	766.23
100	300	5	0.05	45580078	787.99
50	500	10	0.05	45593502	686.09
50	500	5	0.05	45451428	711.68
50	300	5	0.05	45694618	449.77
50	300	10	0.05	46153780	465.05
100	500	10	0.05	45515202	1479.30
100	500	10	0.1	45434544	1473.88
50	500	5	0.1	45773928	640.04
50	500	25	0.05	45542310	701.96
50	500	25	0.1	45597294	735.35

Figura 4: Resultados obtenidos de la versión `Standard` en Python

4.2. Versión en Java

La versión en Java ha sido menos costosa en cuanto a tiempo de ejecución y se han podido obtener resultados para las tres versiones. Para estas tres versiones se han hecho 5 pruebas en cada una con los mismos parámetros, para de esa forma poder hacer una comparación.

La salida en consola que se obtiene en cada ejecución se muestra en la Figura 5.

```
PARÁMETROS:
Versión: Standard
Tamaño de la población: 20
Número de iteraciones: 10
Tamaño del torneo: 5
Probabilidad de mutación: 0.1


Iteración 1/10
Iteración 2/10
Iteración 3/10
Iteración 4/10
Iteración 5/10
Iteración 6/10
Iteración 7/10
Iteración 8/10
Iteración 9/10
Iteración 10/10

RESULTADOS:
Vector solución: [222, 75, 186, 6, 2, 71, 95, 116, 9, 172, 86, 65, 218, 39, 210, 52, 48, 88, 205,
Fitness: 49257816
Nota obtenida: -5.0504756
Tiempo transcurrido: 0.136759841
```

Figura 5: Salida en consola de la ejecución en Java

4.2.1. Standard

En la Figura 6 se muestra la tabla con los resultados obtenidos en esta versión. Se obtienen peores resultados que en Python porque se hacen muchas menos iteraciones. Aunque en Java el tiempo de ejecución sea mucho menor que en Python, si el número de iteraciones es muy alto, en las versiones `Baldwiniana` y `Lamarckiana` el tiempo de ejecución vuelve a ser inviable.

Tamaño población	Iteraciones	Tamaño del torneo	Prob. mutación	Coste de la solución	Tiempo empleado (s)
20	50	5	0.05	46477590	0.34
20	10	10	0.05	48198134	0.13
50	20	20	0.05	47637696	0.41
50	20	10	0.05	48119458	0.37
100	20	10	0.1	47004164	0.64

Figura 6: Resultados obtenidos de la versión `Standard` en Java

La mejor solución obtenida en esta versión es la siguiente:

- **Tamaño de la población:** 20
- **Número de iteraciones:** 50
- **Tamaño del torneo:** 5
- **Probabilidad de mutación:** 0.05
- **Coste de la solución:** 46477590
- **Tiempo de ejecucion:** 0.34 segundos

4.2.2. Baldwiniana

En la Figura 7 se muestra la tabla con los resultados obtenidos en esta versión. A pesar de obtener una solución mejor que en la versión `Standard`, en general los resultados son mucho peores y con un tiempo de ejecución bastante más grande.

Tamaño población	Iteraciones	Tamaño del torneo	Prob. mutación	Coste de la solución	Tiempo empleado (s)
20	50	5	0.05	44901074	3864.21
20	10	10	0.05	52046204	764.26
50	20	20	0.05	57765420	4793.76
50	20	10	0.05	54683074	4893.14
100	20	10	0.1	52858798	10016.03

Figura 7: Resultados obtenidos de la versión `Baldwiniana` en Java

La mejor solución obtenida en esta versión es la siguiente:

- **Tamaño de la población:** 20
- **Número de iteraciones:** 50
- **Tamaño del torneo:** 5
- **Probabilidad de mutación:** 0.05
- **Coste de la solución:** 44901074
- **Tiempo de ejecucion:** 3864.21 segundos

4.2.3. Lamarckiana

En la Figura 8 se muestra la tabla con los resultados obtenidos en esta versión. Esta versión es la que mejores resultados ha obtenido, y es que en esta variante se van heredando los mejores cromosomas gracias a la búsqueda del óptimo local.

Tamaño población	Iteraciones	Tamaño del torneo	Prob. mutación	Coste de la solución	Tiempo empleado (s)
20	50	5	0.05	44879002	3867.91
20	10	10	0.05	44938950	761.89
50	20	20	0.05	44842840	4026.60
50	20	10	0.05	44866248	4910.27
100	20	10	0.1	44925714	8640.69

Figura 8: Resultados obtenidos de la versión `Lamarckiana` en Java

La mejor solución obtenida en esta versión es la siguiente:

- **Tamaño de la población:** 50
- **Número de iteraciones:** 20
- **Tamaño del torneo:** 20
- **Probabilidad de mutación:** 0.05
- **Coste de la solución:** 44842840
- **Tiempo de ejecucion:** 4026.60 segundos

5. Conclusiones

Un algoritmo genético es mejor que sea implementado en un lenguaje de programación compilado ya que incluye optimizaciones que los lenguajes interpretados no son capaces de alcanzar.

Durante las pruebas realizadas se ha podido comprobar que es mejor aumentar el número de iteraciones que el tamaño de la población, ya que si se aumenta la población pero el número de iteraciones es bajo, el resultado no va a mejorar demasiado.

El tamaño de la selección por torneo también influye ya que no es lo mismo hacer un torneo con un 50 % de los individuos que con un 5 % de los individuos, ya que cuanta más probabilidad haya, se seleccionarán más individuos y por lo tanto habrá más posibilidades de seleccionar individuos con un mejor fitness. De hecho, la mejor solución obtenida es aquella en la que se seleccionan aleatoriamente 20 individuos de los 50 que hay disponibles. Es la prueba en la que mayor porcentaje de individuos se seleccionan.

La versión que mejor funciona es la `Lamarckiana`, mientras que la versión que peor resultados da es la `Baldwiniana`, y es que en esta última versión se calcula el fitness de un individuo usando una heurística greedy pero sus cromosomas van a seguir siendo los mismos, mientras que en la primera versión los individuos heredarán los mejores cromosomas. Sin embargo, a pesar de que la `Baldwiniana` obtiene peores resultados en general, el mejor resultado obtenido de su versión es mejor que el mejor obtenido en la `Standard`.

5.1. Mejor resultado obtenido

El mejor resultado obtenido se ha dado en la versión `Lamarckiana` con los siguientes datos:

- **Tamaño de la población:** 50
- **Número de iteraciones:** 20
- **Tamaño del torneo:** 20
- **Probabilidad de mutación:** 0.05
- **Coste de la solución:** 44842840
- **Tiempo de ejecucion:** 4026.60 segundos
- **Vector de la solución:** [106, 9, 195, 6, 213, 209, 98, 154, 141, 159, 128, 2, 180, 87, 68, 189, 43, 165, 224, 26, 41, 245, 234, 184, 145, 126, 102, 198, 226, 38, 172, 204, 80, 36, 76, 63, 162, 192, 174, 17, 117, 137, 115, 232, 177, 215, 191, 169, 55, 50, 24, 120, 187, 78, 132, 89, 206, 221, 135, 21, 243, 65, 72, 202, 236, 85, 150, 46, 19, 32, 217, 58, 83, 53, 147, 109, 12, 139, 0, 61, 29, 247, 238, 91, 228, 113, 167, 251, 255, 14, 111, 124, 201, 73, 231, 37, 210, 193, 49, 248, 56, 75, 214, 185, 242, 246, 112, 200, 95, 54, 39, 101, 108, 222, 94, 166, 99, 134, 28, 227, 23, 156, 114, 11, 170, 103, 179, 16, 116, 93, 199, 31, 1, 151, 104, 239, 125, 158, 88, 235, 22, 148, 203, 218, 69, 97, 30, 237, 252, 66, 160, 146, 133, 127, 244, 67, 40, 149, 110, 42, 51, 196, 155, 74, 130, 59, 25, 64, 92, 205, 122, 81, 100, 13, 223, 254, 140, 15, 152, 119, 188, 86, 52, 33, 182, 207, 47, 190, 153, 79, 163, 175, 219, 143, 90, 225, 129, 71, 138, 157, 173, 183, 70, 136, 10, 62, 3, 181, 176, 249, 27, 77, 48, 240, 164, 60, 250, 45, 211, 168, 178, 161, 18, 35, 107, 123, 8, 5, 57, 20, 142, 171, 208, 44, 105, 7, 241, 131, 4, 220, 197, 121, 118, 230, 186, 229, 84, 144, 96, 216, 212, 34, 82, 253, 194, 233]

Referencias

[1] Laila A. Shawky, Mohamed Abd El-Baset Metwally, and Abd El-Nasser H. Zaied. Quadratic assignment problem: A survey and applications. 2015.

[2] Dataset tai256c. `http://anjos.mgi.polymtl.ca/qaplib/data.d/tai256c.dat`.

[3] Sublime HQ Pty Ltd. Sublime text. `https://www.sublimetext.com/`.

[4] JetBrains. Intellij idea. `https://www.jetbrains.com/es-es/idea/`.

[5] Alfonsas Misevičius and Bronislovas Kilda. Comparison of crossover operators for the quadratic assignment problem. *Information Technology and Control*, 34(2), 2005.

[6] tutorialspoint. Genetic algorithms - mutation. `https://www.tutorialspoint.com/genetic_algorithms/genetic_algorithms_mutation.htm`.