Universidad de Granada

Inteligencia Computacional

Máster en Ingeniería Informática

Resolución de Problemas de Asignación Cuadrática con Algoritmos Evolutivos

Autor

Pedro Manuel Gómez-Portillo López

gomezportillo@correo.ugr.es

Escuela Técnica Superior de Ingenierías Informática y de Telecomunicación

—

Granada, 19 de enero de 2019

El formato de la documentación de este trabajo ha sido basado en la plantilla LaTeX de https://github.com/erseco.

Resolución de Problemas de Asignación Cuadrática con Algoritmos Evolutivos

Pedro Manuel Gómez-Portillo López

Resumen

Los algoritmos evolutivos son métodos de optimización y búsqueda de soluciones basados en la Teoría de la Evolución de las especies.

En esta práctica se trabajará con algoritmos evolutivos para resolver un problema de asignación cuadrática. Para ello, tras diseñar e implementar un algoritmo evolutivo estándar, se diseñarán sus variantes lamarckiana y baldwiniana y se compararán los resultados obtenidos tras su ejecución con el fin de descubrir cuál se desempeña mejor en este problema en concreto.

Palabras clave

algoritmos evolutivos, problemas de asignación cuadrática, qap

Keywords

evolutionary algorithm, quadratic assignment problem, qap

Índice general

Índice de figuras

Capítulo 1

Introducción

Los algoritmos evolutivos son métodos de optimización y búsqueda de soluciones basados en la Teoría de la Evolución de las especies. Según ésta, un conjunto de individuos forma una generación y éstos, al reproducirse entre ellos, generan nuevos individuos con características comunes de ambos padres.

Actualmente existe una gran cantidad de aplicaciones que hacen uso de los algoritmos evolutivos para toda clase de tareas; desde estrategias de búsqueda a problemas de optimización.

Un subtipo de los problemas de optimización son los problemas de optimización cuadrática, o QAP[1] por sus siglas en inglés, que son problemas de optimización combinatoria. Dicho problema puede describirse 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 ellas. El problema consiste en decidir dónde ubicar cada fábrica para minimizar 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

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

[1] Quadratic Asignation Problem

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

El ejemplo clásico de este tipo de problemas es el *TSP*[2] o Problema del Viajante, donde el objetivo es encontrar la ruta más corta entre varias ciudades.

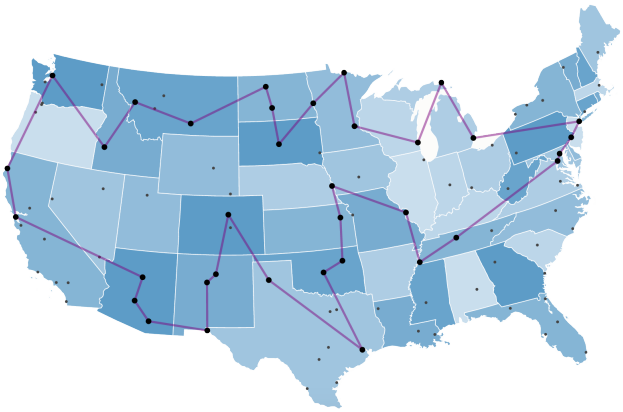

Figura 1.1: Ejemplo de rutas entre ciudades

Los casos de prueba para comprobar el funcionamiento de las distintas heurísticas se han obtenido de la biblioteca *QAPLIB*[3]. Cada uno de los archivos contiene el tamaño del problema, la matriz de flujo y la matriz de distancias.

En esta práctica sólo se trabajará con el archivo `tai256c`, cuyo tamaño es de 256. A día de hoy, la cota inferior global conocida (la mejor solución que se ha obtenido) tiene un fitness de 44,095,032, por lo que se supondrá que cualquier resultado que se obtenga menor a este será erróneo.

1.1. Entorno de desarrollo

Mi ordenador personal, en el cual se ha realizado esta práctica, cuenta con las siguientes características.

- **Sistema operativo**. Windows 10 Educational
- **Procesador**. Intel Core i7-6700HQ Quad-Core 2.6GHz
- **Cache**. 6M
- **Memoria RAM**. 8GB DDR4 2,133MHz

[2] Travel Salesman Problem
[3] `(http://www.seas.upenn.edu/qaplib/`

- **Tarjeta gráfica**. NVIDIA 950M
- **Disco duro**. SanDisk SSD Plus 256GB
- **Lenguaje de programación**. Python 3.7

1.2. Repositorio

El desarrollo y la evolución de esta práctica pueden verse en el siguiente repositorio.

`https://github.com/gomezportillo/qap`

Capítulo 2

Implementación

La figura 2.1 presenta el diagrama de clases del proyecto, que ha sido creada con la herramienta de generación automática de clases a partir de código de *Visual Paradigm*[1].

Se ha diseñado la clase `Genetic_Algorithm`, de la que heredan el resto de algoritmos, que contiene las variables y métodos comunes; esta clase define el tamaño y el número de las generaciones, parsea el fichero de datos, crea las generaciones...

De ella heredan tres clases; la implementación estándar del problema, `Standard`, la variante lamarckiana, `Lamarckian`, y la variante Baldwiniana, `Baldwinian`.

La clase `Individual`, que usan el resto de clases, representa a los individuos del algoritmo; estos individuos tienen un array que representa sus **cromosomas** (generados aleatoriamente) y otras herramientas, como funciones que permiten mutarlos conforme a una probabilidad, y un **fitness**, que indica cómo de óptimos son sus cromosomas.

Como **mecanismo de selección** se ha utilizado un **torneo binario**. En él, se eligen dos individuos al azar de la población actual y se devuelve aquél con menor fitness.

Como **mecanismo de reemplazo** se utiliza el **elitismo**; el mejor individuo de una generación pasa automáticamente a la siguiente, sustituyendo al peor de ésta.

Como **operador de mutación** se utiliza la **técnica del intercambio** adaptada para que dependa de una probabilidad. Cada individuo tiene una probabilidad de mutar como individuo y de que mute cada uno de sus cromosomas por separado.

[1] `https://www.visual-paradigm.com/`

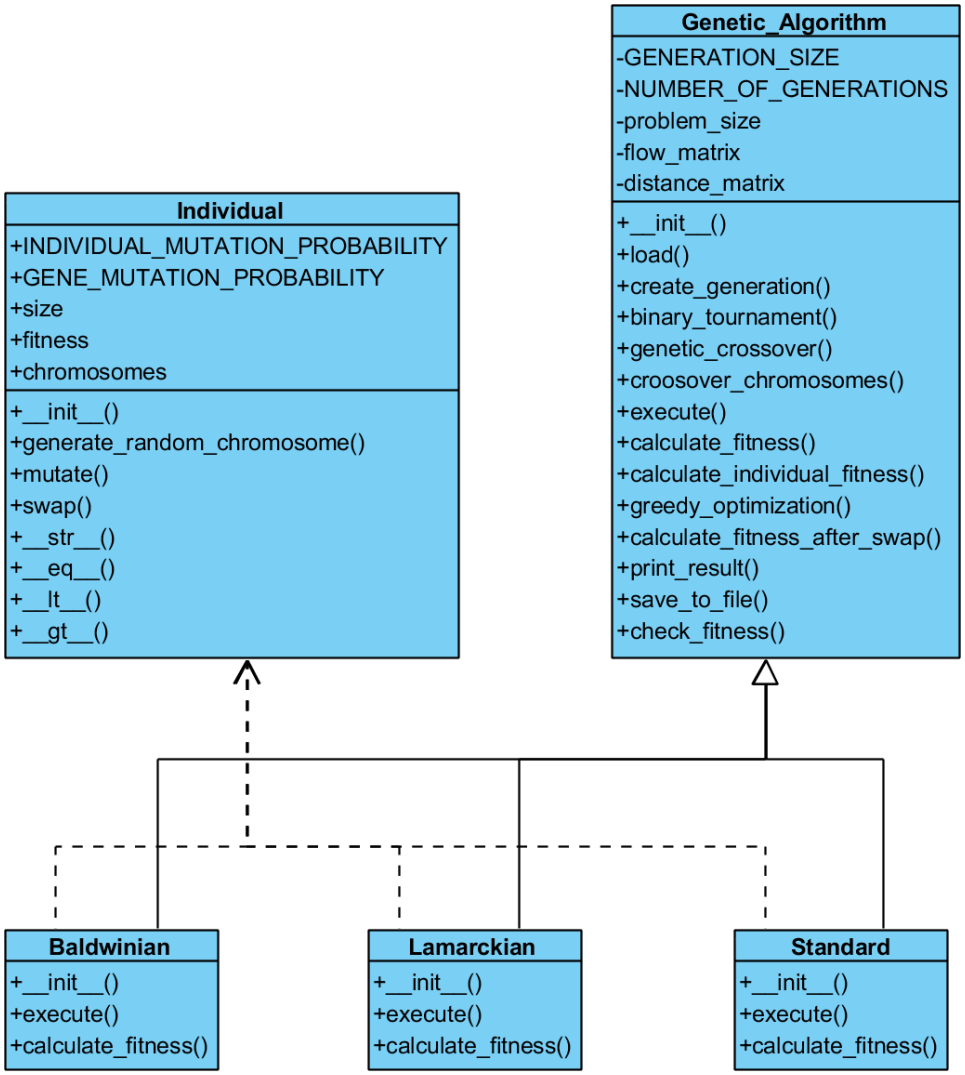

Figura 2.1: Diagrama de clases del proyecto

Como **operador de cruce** se ha utilizado **recombinación en un punto**. Esta técnica permite cortar a ambos padres en un punto y recombinar sus trozos. La imagen 2.2 representa este algoritmo, aunque específicamente en este problema hay que tener cuidado de no repetir sus cromosomas, pasando al siguiente número en caso de que ya exista.

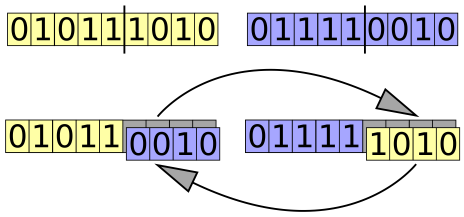

Figura 2.2: Representación gráfica de la recombinación por punto
Referencia: Wikipedia.

Los tres algoritmos comparten la ejecución básica, ya que solo se diferencian en la manera de calcular el fitness de los individuos. En el listado 2.1 puede verse el pseudocódigo de esta ejecución común.

Fragmento de código 2.1: Ejecución base de los algoritmos

```
1  leer_fichero()
2  generacion_actual = crear_generacion()
3  generacion_actual.calcular_fitness()
4
5  for i=0..NUM_GENERACIONES:
6      for j=0..TAM_GENERACION:
7          padre1 = torneo_binario(generacion_actual)
8          padre2 = torneo_binario(generacion_actual) && padre1 != padre2
9
10         hijo1, hijo2 = recombinar(padre1, padre2)
11         hijo1.mutar()
12         hijo2.mutar()
13
14         nueva_generacion.añadir(hijo1, hijo2)
15
16     mejor = generacion_actual.obtener_mejor()
17     nueva_generacion.sustituior_peor_por(mejor)
18     generacion_actual = nueva_generacion
19     generacion_actual.calcular_fitness()
20
21     comprobar_que_el_mejor_no_se_repite_o_reinicializar(mejor)
22
23 return generacion_actual.obtener_mejor()
```

En la linea 21, la única no explicada anteriormente, se comprueba que el mejor individuo de la población no lleva repitiéndose más veces que las permitidas, y de ser así reinicia la población conservando el mejor individuo de la anterior.

2.1. Algoritmo estándar

De las tres implementaciones de la que consta la práctica, el algoritmo estándar fue el primero que se programó. Su fitness es el más sencillo, ya que basta con aplicar la fórmula 1 explicada en la introducción.

2.2. Variantes evolutivas

La técnica de optimización local aplicada es un algoritmo *greedy 2-opt*. El fragmento de pseudocódigo 2.2 muestra cómo funciona.

Fragmento de código 2.2: Optimización greedy 2-opt

```
1 S = candidato inicial con coste c(S)
2 mejor = S
3 for i=1..n:
4     for j=i+1..n:
5         T = S tras intercambiar i con j
6         if c(T) < c(S):
7             S=T
8 return S
```

Se ha simplificado con respecto a la versión propuesta en el guión de prácticas eliminando el bucle `while`, ya que es prácticamente imposible que tras iterar cuadráticamente sobre el tamaño del problema `S` y `T` terminen siendo iguales, y tras realizar pruebas se comprobó que esto nunca sucede, por lo que se decidió eliminar de la implementación.

Por otro lado Python, por ser un lenguaje interpretado, funciona especialmente lento en comparación a otros lenguajes compilados, por lo que se ha dedicado un esfuerzo extra en intentar optimizar todo lo posible el algoritmo. Concretamente, y como fue propuesto en clase por el profesor Berzal, se ha diseñado una función específica para recalcular el fitness de los individuos tras intercambiar sus cromosomas (línea 5 del listado 2.2), modificando solo los valores que han cambiado. Así, se ha conseguido reducir la complejidad de esa función en concreto de $O(n^2)$ a $O(n)$.

2.2.1. Variante lamarckiana

Esta variante está basada en la teoría evolutiva del naturalista francés **Jean-Baptiste Lamarck** (1744-1829), quien afirmaba que las mejoras fisiológicas que obtenía un individuo a lo largo de su vida quedaban grabadas en sus genes, por lo que sus descendientes adquirían esta mejoras en su código genético.

Así, tras obtener un individuo optimizado con la función presentada en el litado 2.2, el individuo original el sustituido por su versión mejorada.

2.2.2. Variante baldwiniana

Esta variante está basada en la teoría evolutiva del psicólogo estadounidense **James Mark Baldwin** (1861-1934), quien afirmaba que las mejoras fisiológicas de un individuo obtenidas a lo largo de su vida no se graban en sus genes, por lo que su descendencia no las obtiene.

Así, tras aplicar una optimización local en cada generación, éstas no se usan para generar la siguiente población, sino que *mueren* con sus individuos.

Capítulo 3

Resultados

Tras implementar y ejecutar las tres variantes del algoritmo genético se pasó a analizar los resultados.

Para los resultados presentados en esta sección se han usado los siguientes parámetros.

En lo que respecta a la población,

- **Tamaño de población.** 60
- **Número de generaciones.** 100
- **Máximo número de repeticiones del mejor individuo.** 20

Inicialmente se probó con tamaños de población más pequeños y mayor número de generaciones, pero el hecho de tener pocos individuos hacía que se obtuviera rápidamente el mínimo de la ejecución, y por lo general no solían ser muy buenos.

Por otro lado, la variable `Máximo número de repeticiones del mejor individuo` indica cuántas veces puede repetirse el mejor individuo una generación tras otra antes de reinicializar la población, asumiendo que ese individuo es especialmente bueno y la población actual no va a mejorarlo.

En lo que respecta a los individuos,

- **Probabilidad individual de mutación.** 50 %
- **Probabilidad de mutación de cada cromosoma.** 5 %

Se ha intentado no depender mucho en el RNG[1] para obtener los resultados, por lo que las probabilidades de mutación no son muy altas.

[1] Random Number Generation

3.1. Comparación de resultados

En el gráfico 3.1 se presentan los resultados tras ejecutar los tres algoritmos. En el anexo 5.1 pueden verse tanto el tiempo de ejecución como el fitness y los cromosomas del mejor individuo de cada ejecución.

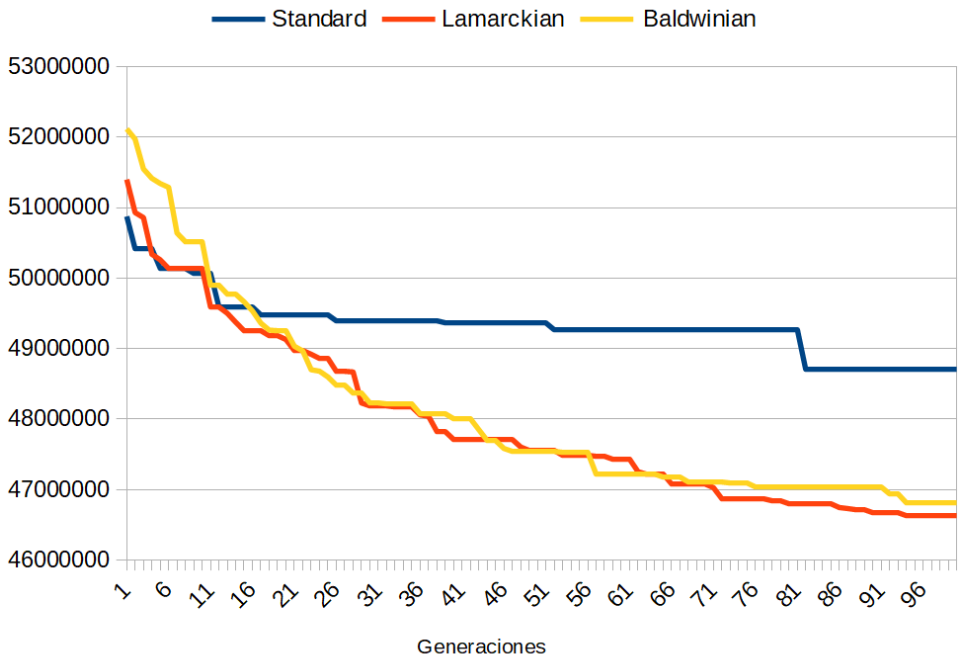

Figura 3.1: Comparación del resultado de la ejecución de los tres algoritmos

Se han repetido las ejecuciones varias veces con resultados muy parecidos, por lo que se concluye que estos datos son representativos.

Como puede verse, el algoritmo que peor desempeño tiene es el estándar, ya que mejora muy poco y muy lentamente a lo largo de su ejecución. Por otro lado, sus dos variantes tienen resultados bastante parecidos, aunque la implementación lamarckiana da mejores resultados. Esto tiene sentido, ya que transmitir las mejoras de un individuo en sus cromosomas, aunque no sea realista, proporciona mejores resultados que de no hacerlo.

3.2. Mejor resultado obtenido

Actualmente, el mejor resultado obtenido ha sido con la ejecución de la implementación **lamarckiana** con un fitness de **46634000**, un ∼1.06 % ma-

yor que la cota inferior global conocida (44095032). Sus cromosomas pueden verse en el anexo 5.1.

Capítulo 4

Conclusiones

Los algoritmos genéticos son muy profundos y, aunque esta práctica me ha servido sólo como introducción, me alegro mucho de haberla realizado.

Además, el hecho de haber utilizado un lenguaje interpretado, más lento, me ha llevado a tener que optimizar la práctica, algo que de haber obtenido unos tiempos de ejecución razonablemente bajos no me habría planteado hacer.

La mayoría de mis compañeros ya tenían nociones básicas y conocían los conceptos de los algoritmos genéticos por haberlos visto en el Grado, pero en la UCLM no se estudian así que yo apenas sí los conocía de oídas. Por eso, aunque me haya podido costar algo más de trabajo realizar la práctica, creo que me ha resultado especialmente útil.

Capítulo 5

Anexos

5.1. Anexo 1. Resultados de la comparación de los tres algoritmos

Standard

- **Tiempo de ejecución.** ∼178 segundos

- **Fitness del mejor elemento.** 48711852

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

Lamarckian

- **Tiempo de ejecución.** ∼7552 segundos
- **Fitness del mejor elemento.** 46634000
- **Cromosomas** [169, 56, 148, 224, 235, 32, 12, 250, 103, 243, 50, 65, 150, 192, 60, 254, 214, 147, 255, 173, 95, 241, 76, 45, 137, 26, 191, 167, 126, 84, 248, 125, 101, 201, 186, 58, 245, 199, 88, 4, 17, 228, 107, 178, 0, 180, 197, 226, 23, 31, 53, 156, 110, 62, 217, 68, 105, 122, 203, 188, 206, 81, 161, 189, 253, 54, 159, 165, 129, 93, 231, 51, 18, 43, 130, 135, 212, 116, 154, 72, 112, 86, 21, 160, 221, 9, 90, 118, 98, 79, 184, 193, 213, 215, 5, 89, 66, 113, 70, 82, 151, 67, 171, 209, 97, 227, 195, 142, 102, 205, 20, 219, 111, 61, 237, 96, 52, 49, 174, 7, 211, 37, 69, 64, 104, 223, 55, 15, 200, 41, 247, 157, 119, 74, 39, 35, 181, 216, 220, 128, 27, 6, 106, 11, 34, 141, 179, 83, 131, 204, 230, 194, 40, 145, 138, 46, 77, 59, 10, 28, 225, 244, 120, 187, 190, 149, 71, 196, 42, 123, 182, 47, 172, 152, 121, 124, 198, 36, 94, 78, 164, 99, 146, 218, 24, 87, 117, 13, 44, 249, 108, 14, 239, 242, 234, 33, 85, 163, 1, 238, 75, 8, 251, 38, 166, 229, 185, 127, 240, 162, 202, 3, 233, 246, 144, 73, 236, 133, 153, 57, 30, 155, 177, 115, 140, 222, 183, 109, 100, 176, 19, 16, 232, 158, 168, 207, 134, 63, 208, 136, 29, 132, 170, 252, 92, 91, 48, 2, 210, 143, 175, 114, 80, 139, 22, 25]

Baldwinian

- **Tiempo de ejecución.** ∼7029 segundos
- **Fitness del mejor elemento.** 46815666
- **Cromosomas** [20, 50, 153, 186, 85, 96, 162, 53, 201, 250, 70, 221, 74, 194, 214, 0, 103, 145, 19, 12, 197, 160, 152, 248, 172, 158, 150, 232, 14, 97, 2, 33, 92, 78, 218, 130, 42, 121, 180, 45, 135, 244, 211, 63, 168, 31, 139, 5, 89, 165, 247, 192, 148, 39, 183, 190, 25, 251, 65, 226, 163, 100, 128, 188, 159, 56, 255, 44, 208, 38, 125, 67, 126, 93, 241, 131, 230, 87, 199, 237, 27, 95, 206, 52, 82, 59, 252, 141, 203, 117, 107, 24, 13, 134, 127, 3, 90, 32, 105, 220, 98, 176, 88, 200, 146, 124, 102, 187, 84, 8, 73, 253, 240, 51, 182, 207, 222, 104, 149, 119, 198, 108, 181, 242, 75, 238, 204, 22, 16, 216, 157, 254, 154, 151, 166, 129, 58, 114, 174, 120, 116, 35, 26, 169, 6, 202, 17, 106, 28, 227, 76, 109, 137, 179, 10, 72, 40, 205, 1, 60, 210, 15, 225, 41, 57, 43, 47, 223, 55, 101, 49, 142, 178, 23, 193, 115, 91, 122, 118, 235, 184, 140, 189, 94, 86, 156, 246, 138, 99, 213, 245, 136, 212, 224, 219, 234, 144, 164, 34, 71, 233, 29, 175, 54, 239, 249, 143, 48, 36, 209, 9, 30, 133, 111, 11, 66, 110, 37, 217, 62, 147, 64, 177, 228, 21, 77, 161, 167, 191, 196, 68, 18, 69, 112, 229, 46, 123, 79,

83, 80, 231, 81, 4, 173, 185, 113, 236, 170, 215, 61, 195, 7, 155, 171, 132, 243]

5.2. Anexo 2. Código fuente de la práctica

main.py

```
from auxiliary import *

from standard import Standard
from baldwinian import Baldwinian
from lamarckian import Lamarckian


standard   = Standard()
baldwinian = Baldwinian()
lamarckian = Lamarckian()


if __name__ == '__main__':

    elapsed_time = execute_algorithm( standard, 'tai256c.dat' )
    print("Standard executing time: {:.3f}s\n".format(elapsed_time))

    elapsed_time = execute_algorithm( baldwinian, 'tai256c.dat' )
    print("Baldwinian executing time: {:.3f}s\n".format(elapsed_time))

    elapsed_time = execute_algorithm( lamarckian, 'tai256c.dat' )
    print("Lamarckian executing time: {:.3f}s\n".format(elapsed_time))
```

auxiliary.py

```
import os
import time

from genetic_algorithm import GeneticAlgorithm
from standard import Standard

DATA_DIR = os.path.join('src', 'data', 'qap')


def get_data_files( dir ):
    """
    Reference https://stackoverflow.com/questions/3207219/how-do-i-list-all-files-of-a-directory
    """
    def exists(file):
        return os.path.isfile(os.path.join(dir, file))

    return [file for file in os.listdir(dir) if exists(file)]


def execute_algorithm( algorithm, datafile ):
    """
    Executes a genetic algorithm and returns its computing time
    """
    if issubclass(type(algorithm), GeneticAlgorithm):
        start_time = time.time()
        algorithm.execute( datafile )
        return time.time() - start_time

    else:
        raise Exception('The algorithm is not a subclass of GeneticAlgorithm')


def check_files( ):
    """
    Runs the Standar genetic algorithm with all the files and checks for AssertionErrors, that happens when the files are not well-structured Corret structure: Problem size, flow matrix and distance matrix with size equal to the problem size
    Current erros: 19
    """
    n_assertion_err = 0
    for file in get_data_files( DATA_DIR ):
        try:
            execute_algorithm( Standard(), file )
        except AssertionError:
            print("========== AssertionError exception on file", file)
            n_assertion_err += 1
    print(n_assertion_err)
```

genetic_algorithm.py

```
"""
Parent class of the rest of the algorithms holding its variables
"""

import os
import sys
import random
from copy import deepcopy

from individual import Individual


class GeneticAlgorithm:
    """
    Parent class of the rest of algorithms, holding their common
        functions and variables
    """

    def __init__(self):
        """
        Fixed variables
        """
        self.GENERATION_SIZE       = 60
        self.NUMBER_OF_GENERATIONS = 100
        self.MAX_NUMBER_REPETITION_BEST_ONE = 20
        """
        Problem variables
        """
        self.problem_size    = -1
        self.flow_matrix     = []
        self.distance_matrix = []

        """
        Counts the number of times the best individual of a generation
             is repeated to reinitialise it
        """
        self.last_best_one = Individual(0)
        self.repetition_best_one = 0

        """
        Stores the best one of each generation to (manually) make a
            graph
        """
        self.bests = []


    def load(self, filename):
        """
        Parses a file and stores its flow and distance matrices as
            class variables that can be accessed by the class
            inheriting from it
        """
        self.filename = filename
        datafile = os.path.join('src', 'data', 'qap', self.filename)

        if os.path.isfile( datafile ):
            with open( datafile, 'r' ) as f:
                lines = f.readlines()
                lines = [line.split() for line in lines]
                lines = [list(map(int, line)) for line in lines if
                    line != []] # avoids empty lines

```

```
57                  self.problem_size = int(lines[0][0])
58
59                  assert(len(lines) == self.problem_size*2+1) # checks
                        the file has a correct structure
60
61                  self.flow_matrix = lines[1:self.problem_size+1]
62                  self.distance_matrix = lines[self.problem_size+1:]
63
64          else:
65              raise FileNotFoundError("Cannot find file {}".format(
                    datafile ))
66
67
68      def create_generation(self):
69          """
70          Creates a generation with the size of the problem initialised
                with random individuals
71          """
72          generation = []
73          for i in range(self.GENERATION_SIZE):
74              generation.append( Individual( self.problem_size ) )
75
76          return generation
77
78
79      def binary_tournament(self):
80          """
81          Randomly selects two different individuals from the current
                generation and returns the optimal one
82          """
83          rand_numb1, rand_numb2 = random.sample(range(0, self.
                GENERATION_SIZE), 2)
84
85          individ1 = self.current_generation[rand_numb1]
86          individ2 = self.current_generation[rand_numb2]
87
88          return min([individ1, individ2])
89
90
91      def genetic_crossover(self, parent1, parent2):
92          """
93          Mixes the two parent individuals into two children slicing
                them by a random index and avoiding repeating chromosomes
                in each child
94          """
95          slice_index = random.randint(1, self.problem_size-1)
96
97          child1_chrom = self.croosover_chromosomes(slice_index,
98                                                    parent1.chromosomes,
99                                                    parent2.chromosomes)
100         child2_chrom = self.croosover_chromosomes(slice_index,
101                                                   parent2.chromosomes,
102                                                   parent1.chromosomes)
103
104         assert(len(child1_chrom) == len(child2_chrom) == self.
                problem_size)
105
106         child1 = Individual( self.problem_size )
107         child1.chromosomes = child1_chrom
108
109         child2 = Individual( self.problem_size )
110         child2.chromosomes = child2_chrom
```

```
111
112         return child1, child2
113
114
115     def croosover_chromosomes(self, slice_index, parent1, parent2):
116         numbers_left = self.problem_size - slice_index
117         child = parent1[:slice_index]
118
119         index = slice_index
120         while len(child) < self.problem_size:
121             if parent2[index] not in child:
122                 child.append(parent2[index])
123
124             index += 1
125             if index >= self.problem_size:
126                 index = 0
127
128         return child
129
130
131     def execute(self, datafile):
132         """
133         Genetic algorithm's execution function. It is inherited by all
                its children, which will overload the default '
            calculate_fitness' function.
134         """
135         print("Executing algorithm with file {}".format(datafile))
136
137         self.load( datafile )
138
139         self.current_generation = self.create_generation()
140         self.calculate_fitness( self.current_generation )
141
142         for i in range( self.NUMBER_OF_GENERATIONS ):
143             print("Executing generation {}/{}... Best {}".format(i+1,
                    self.NUMBER_OF_GENERATIONS, self.last_best_one.fitness
                    ), end="\r")
144
145             new_generation = []
146             for j in range( 0, int(self.GENERATION_SIZE), 2 ): # step
                    = 2
147                 parent1 = self.binary_tournament()
148
149                 parent2 = None
150                 while parent1 != parent2:
151                     parent2 = self.binary_tournament()
152
153                 child1, child2 = self.genetic_crossover(parent1,
                        parent2)
154
155                 child1.mutate()
156                 child2.mutate()
157
158                 new_generation.append( child1 )
159                 new_generation.append( child2 )
160
161
162             """
163             Pops out the worst one of the current generation and
                    inserts in its place the best one of the previous
                    generation
164             """
```

```
165             old_best = min( self.current_generation )
166             new_worst = max( new_generation )
167             new_worst_index = new_generation.index( new_worst )
168             new_generation.pop( new_worst_index )
169             new_generation.append( old_best )
170             self.current_generation = new_generation
171 
172             self.calculate_fitness( self.current_generation )
173 
174             """
175             Check the best one is not stuck being repeated over
                    generations. Otherwise, reinitialise the population
                    keeping the best one
176             """
177             best_one = min( self.current_generation )
178             self.check_best_one_from_generation( best_one )
179 
180             self.bests.append(best_one)
181 
182         best_one = min( self.current_generation )
183         return best_one
184 
185 
186     def calculate_fitness(self, generation):
187         """
188         All the child classes inheriting from this one shall override
                this function.
189         """
190         raise NotImplementedError
191 
192 
193     def calculate_individual_fitness(self, individual):
194         """
195         Calculates the fitness of a single individual and checks that
                it is not greater than the possible maximum.
196         """
197         new_fitness = 0
198         for i in range(self.problem_size):
199             for j in range(self.problem_size):
200                 chrom_i = individual.chromosomes[i]
201                 chrom_j = individual.chromosomes[j]
202 
203                 new_fitness += self.flow_matrix[i][j] * \
204                                self.distance_matrix[chrom_i][chrom_j]
205 
206         self.check_fitness( new_fitness )
207         return new_fitness
208 
209 
210     def greedy_optimization(self, individual):
211         """
212         Greedy 2-opt algorithm implementing the pseudocode given on
                the problem statement. The do/while loop has been ommited
                from the algorithm as it is virtually impossible that
                after all the permutations S == best, and even though if
                that was the case it would not change for executing for
                loops again.
213         """
214         S = deepcopy( individual )
215         S.fitness = self.calculate_individual_fitness( S )
216 
217         best = deepcopy( S )
```

```
218
219         for i in range( S.size ):
220             for j in range(i + 1, S.size):
221                 T = deepcopy( S )
222                 T.chromosomes[i] = S.chromosomes[j]
223                 T.chromosomes[j] = S.chromosomes[i]
224
225                 self.calculate_fitness_after_swap( S, T, i, j )
226
227                 if T < S:
228                     S = deepcopy( T )
229
230         return S
231
232
233     def calculate_fitness_after_swap(self, S, T, i, j):
234         """
235         Instead of calculating again the whole fitness, it is only a
                matter of calculing the chromosomes that have been changed
                , reducing complexity from n^2 to 2n
236         """
237         new_fitness = T.fitness
238         chrom_S_i = S.chromosomes[i]
239         chrom_S_j = S.chromosomes[j]
240         chrom_T_i = T.chromosomes[i]
241         chrom_T_j = T.chromosomes[j]
242
243         for k in range(self.problem_size):
244             chrom_S_k = S.chromosomes[k]
245             chrom_T_k = T.chromosomes[k]
246
247             # recalculate i
248             new_fitness -= self.flow_matrix[i][k] * \
249                            self.distance_matrix[chrom_S_i][chrom_S_k]
250
251             new_fitness -= self.flow_matrix[i][k] * \
252                            self.distance_matrix[chrom_T_i][chrom_T_k]
253
254             # recalculate j
255             new_fitness -= self.flow_matrix[j][k] * \
256                            self.distance_matrix[chrom_S_j][chrom_S_k]
257
258             new_fitness += self.flow_matrix[j][k] * \
259                            self.distance_matrix[chrom_T_j][chrom_T_k]
260
261             # recalculate the rest of the values of the loop
262             if k not in [i, j]:
263                 # recalculate i
264                 new_fitness -= self.flow_matrix[k][i] * \
265                                self.distance_matrix[chrom_S_k][
                                    chrom_S_i]
266
267                 new_fitness += self.flow_matrix[k][i] * \
268                                self.distance_matrix[chrom_T_k][
                                    chrom_T_i]
269
270                 # recalculate j
271                 new_fitness -= self.flow_matrix[k][j] * \
272                                self.distance_matrix[chrom_S_k][
                                    chrom_S_j]
273
274                 new_fitness += self.flow_matrix[k][j] * \
```

```
275                                 self.distance_matrix[chrom_T_k][
                                        chrom_T_j]
276
277
278     def check_best_one_from_generation(self, best_one):
279         """
280         Check the best one is not stuck being repeated over
                generations. Otherwise, reinitialise the population
                keeping the best one
281         """
282         if best_one == self.last_best_one:
283             self.repetition_best_one += 1
284             if self.repetition_best_one > self.
                    MAX_NUMBER_REPETITION_BEST_ONE:
285                 print("\nStuck population! Best one {}. Reinitialising
                        ...".format(best_one.fitness))
286                 self.reinitialise_population( best_one )
287                 self.repetition_best_one = 0
288
289         else:
290             self.repetition_best_one = 0
291
292         self.last_best_one = best_one
293
294
295     def reinitialise_population(self, best_one):
296         """
297         Generates another random generation, pops one individual at
                random and inserts the best one from the previous
                generation
298         """
299         self.current_generation = self.create_generation()
300         self.calculate_fitness( self.current_generation )
301         self.current_generation.pop()
302         self.current_generation.append(best_one)
303
304
305     def print_result(self, best_one):
306         """
307         Prints the final result.
308         """
309         print("_________________________________________________")
310         print("Recombination operator: Crossover")
311         print("Mutation operator: Index swap")
312         print("Problem size: ", self.problem_size)
313         print("Number of generations: ", self.NUMBER_OF_GENERATIONS)
314         print("Generation size: ", self.GENERATION_SIZE)
315         print("Fitness of the final best individual: ", best_one.
                fitness)
316         print("Chromosomes:\n", best_one.chromosomes )
317
318
319     def save_to_file(self, best_one):
320         """
321         Redirects the stdout to a file and saves the results
322         """
323         filename = 'result.txt'
324         filename = os.path.join('results', filename)
325         f = open( filename, 'w')
326
327         orig_stdout = sys.stdout
328         sys.stdout = f
```

```
329
330         self.print_result( best_one )
331
332         sys.stdout = orig_stdout
333         f.close()
334
335
336     def check_fitness(self, fitness):
337         """
338         Check the fitness is not lesser than the possible value
339         """
340         if self.filename == 'tai256c.dat' and fitness < 44095032:
341             raise Exception("Fitness cannot be lesser than 44095032 on
                    file \
342                              tai256c, current ", fitness)
```

standard.py

```
1
2  from genetic_algorithm import GeneticAlgorithm
3  from individual import Individual
4
5  class Standard(GeneticAlgorithm):
6      """
7      Standard implementation of the genetic algorithm
8      """
9
10     def __init__(self):
11         super(Standard, self).__init__()
12
13
14     def execute(self, datafile):
15         """
16         Loads the data, creates the first generation and executes the
               genetic algorithm on each generation overloading and
               executing the child 'calculate_fitness' function. Finally
               returns the best of 'em all.
17         """
18         best_one = super().execute( datafile )
19         super().print_result( best_one )
20         super().save_to_file( best_one )
21
22         return best_one
23
24
25     def calculate_fitness(self, generation):
26         """
27         Calculate the fitness of each individual on the generation
28         """
29         for individual in generation:
30             individual.fitness = super().calculate_individual_fitness(
                   individual)
```

lamarckian.py

```
from genetic_algorithm import GeneticAlgorithm
from individual import Individual


class Lamarckian(GeneticAlgorithm):
    """
    Lamarckian implementation of the genetic algorithm
    """

    def __init__(self):
        super(Lamarckian, self).__init__()


    def execute(self, datafile):
        """
        Loads the data, creates the first generation and executes the
            genetic algorithm on each generation overloading and
            executing the child 'calculate_fitness' function. Finally
            returns the best of 'em all.
        """
        best_one = super().execute( datafile )
        super().print_result( best_one )
        super().save_to_file( best_one )
        return best_one


    def calculate_fitness(self, generation):
        """
        Optimizes the individual and saves the changes for them to be
            inherited
        by its offspring so next generation have it. Works in the say
            proposed
        by the biologist J. B. Lamark.
        """
        for individual in generation:
            individual = super().greedy_optimization( individual )
```

baldwinian.py

```
from genetic_algorithm import GeneticAlgorithm
from individual import Individual


class Baldwinian(GeneticAlgorithm):
    """
    Baldwinian implementation of the genetic algorithm
    """

    def __init__(self):
        super(Baldwinian, self).__init__()


    def execute(self, datafile):
        """
        Loads the data, creates the first generation and executes the
            genetic algorithm on each generation overloading and
            executing the child 'calculate_fitness' function. Finally
            returns the best of 'em all.
        """
        best_one = super().execute( datafile )
        super().print_result( best_one )
        super().save_to_file( best_one )
        return best_one


    def calculate_fitness(self, generation):
        """
        Optimizes the individual but only changes its fitness without
            letting the optimisation, thus not allowing the
            optimization to be in it offspring. Works in the way
            propposed by the psicologist J. M. Baldwin.
        """
        for individual in generation:
            optimized_individual = super().greedy_optimization(
                individual )
            individual.fitness = optimized_individual.fitness
```

individual.py

```
import math
import random


class Individual:

    def __init__(self, size):
        self.INDIVIDUAL_MUTATION_PROBABILITY = 0.5
        self.GENE_MUTATION_PROBABILITY       = 0.05

        self.size    = size
        self.fitness = math.inf
        self.generate_random_chromosome( size )


    def generate_random_chromosome(self, size):
        self.chromosomes = list(range(size))
        random.shuffle(self.chromosomes)


    def mutate(self):
        if self.INDIVIDUAL_MUTATION_PROBABILITY > random.random():
            for chromosome in self.chromosomes:
                if self.GENE_MUTATION_PROBABILITY > random.random():
                    index1, index2 = random.sample(range(0, self.size)
                        , 2)
                    self.swap( index1, index2 )


    def swap(self, index1, index2):
        val1 = self.chromosomes[index1]
        val2 = self.chromosomes[index2]
        self.chromosomes[index1] = val2
        self.chromosomes[index2] = val1


    def __str__(self):
        return str(self.chromosomes)


    def __eq__(self, other):
        if not isinstance(other, Individual):
            return False

        equal_size = self.size == other.size
        equal_chrom = set(self.chromosomes) == set(other.chromosomes)

        return equal_size and equal_chrom


    def __lt__(self, other):
        if not isinstance(other, Individual):
            return False

        return self.fitness < other.fitness


    def __gt__(self, other):
        if not isinstance(other, Individual):
```

```
59              return False
60
61          return self.fitness > other.fitness
```