GENETIC ALGORITHM FOR TSP

*Implementation and Experimentation of a genetic in Python.*

### The next document describes the process followed to implement the “Traveling Salesperson” Problem in Python. It includes a memory of changes of the project and the results of experimentation obtained.

Alberto Bujalance Muñoz - Melchor Vázquez Martínez

07/09/2017

# INTRODUCTION

## Software

The software used to develop the task was mainly [Spider v.3](https://pythonhosted.org/spyder/) and the [Python Release 3.5](https://www.python.org/downloads/release/python-350/); those two were obtained by means of [Anaconda](https://www.anaconda.com/download/). In the latest stages of the project, [Git](https://git-scm.com/) was used to upload the project and make it public in addition to keep a better version control.

The project is uploaded to: <https://github.com/DeadPixelG/TSP-Gen>

## Hardware

We used a variety of different computers in development but the specs of the one used in the experimentation stage are the most important since it determines the time and performance of the algorithm.

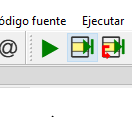
**CPU:** AMD FX-6300 3.5GHZ x 6 cores.

**RAM:** G.Skill Sniper 8GB (4GBx2) DDR3 1333.

## (Melchor, si usas tu PC para experimentación, añade aquí tus specs también)

# USER GUIDE

A visual user interface is not provided but the algorithm is not hard to use.

All you need is to have a python interpreter and a console to introduce some basic commands. Three steps are required:

* **Run the Code.** Using your python interpreter, click on the play button or whatever it uses to run the code. Once it’s done, prepare to introduce two things on console.
* **Introduce the sets of cities.** You need to manually create the cities that you are going to use, including the distances and roads to other cities. Note that you have to add any city that has a road pointing to it from another city. We provide 2 sets. A basic one with Andalusian capitals and a second, more complex one, with all Spaniard capitals. The format of each city is of the type: ***City(‘Name\_Of\_The\_City’, {‘City\_Joined\_By\_Road’ : Distance , … })***

Just copy one of the sets and press enter.

*City('Sevilla', {'Córdoba': 104.5,'Málaga': 204.8, 'Huelva':92.8, 'Cádiz':121.1,'Almería':412.4, 'Granada':249.7,'Jaén': 241.4})*

*City('Córdoba', {'Sevilla': 104.5,'Málaga': 162.6, 'Huelva':238.5, 'Cádiz':265.1,'Almería':342.1, 'Granada':207.8,'Jaén': 108.1})*

*City('Málaga', {'Sevilla': 204.8,'Córdoba': 162.6, 'Huelva':301.9, 'Cádiz':235.6,'Almería':201.5, 'Granada':126.5,'Jaén': 203.7})*

*City('Huelva', {'Sevilla': 92.8,'Córdoba': 238.5, 'Málaga':301.9, 'Cádiz':209.9,'Almería':508.7, 'Granada':346.0,'Jaén': 336.3})*

*City('Cádiz', {'Sevilla': 121.1,'Córdoba': 265.1, 'Málaga':235.6, 'Huelva':209.9,'Almería':435.7, 'Granada':292.8,'Jaén': 324.8})*

*City('Almería', {'Sevilla': 412.4,'Córdoba': 342.1, 'Málaga':201.5, 'Huelva':508.7,'Cádiz':435.7, 'Granada':167.6,'Jaén': 224.5})*

*City('Granada', {'Sevilla': 249.7,'Córdoba': 207.8, 'Málaga':126.5, 'Huelva':346.0,'Cádiz':292.8, 'Almería':167.6,'Jaén': 92.0})*

*City('Jaén', {'Sevilla': 241.4,'Córdoba': 108.1, 'Málaga':203.7, 'Huelva':336.3,'Cádiz':324.8, 'Almería':224.5,'Granada': 92.0})*

* **Execute the main method.** The method **genetic\_prob** is used to solve the problem. It can be stored in a variable (sol=genetic\_prob)if you want to access the properties separately. The method receives 4 parameters: **ages**, **size of the population** (at least 2), **chance of mutation** (from 0 to 1) and the **set of cities** to be used. When you add the cities in the previous step, a set of cities called “cities” is already created so it should be always left to that value. An example: ***genetic\_prob(10, 3, 0.05, cities)*** .

If you want to access the properties of a solution, you need to use: ***Name\_of\_the\_variable[‘property\_name’]***

There are 3 properties: **population** (which gives the final population), **grades** (gives the grade of each population over the ages to see its evolution) and **result** (The final route in order). The total run time of the algorithm is also displayed at the end of the execution.

When executing the example above, we get the next result:

*algorithm finished in: 0h 0m 0.0009999275207519531s*

*{****'grades'****: [2023.4333333333334, 1999.5666666666666, 1916.6333333333332, 1833.7, 1833.7, 1833.7, 1833.7, 1833.7, 1820.533333333333, 1807.3666666666666],*

***'population'****: [[4, 7, 0, 2, 5, 3, 6, 1], [4, 7, 0, 5, 3, 2, 6, 1],*

*[4, 7, 0, 5, 3, 2, 6, 1]],*

***'result'****: ['Málaga', 'Jaén', 'Almería', 'Cádiz', 'Sevilla','Huelva','Granada','Córdoba']}*

Improvement of the grades can be seen from age to age but as the population is small, it doesn’t vary too much. With further experimentation, we will see how the grades and the results behave.

# THE ALGORITHM

## Introduction

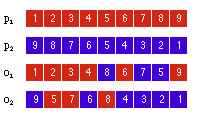
The algorithm is formed by a set of methods which are used in a main method called **genetic\_prob.** They can be classified in two types: **statistical** and **functional**. The main ones are functional, although some statistical methods are used at the end to help us gather the necessary experimentation information.

## Solution Format

Before getting an in-depth view of the algorithm, we believe that a formal explanation of our solution is in order.

**Genes:** Numbers from 0 to the size of the chromosome. Cannot be repeated in the same chromosome.

**Chromosome:** List of numbers from 0 to its size, not necessarily ordered. The **size** of each chromosome or **individual** is given by the number of cities.

**Mutation**: We considered order-mutation to be the best one to solve this problem. Since our main problem is the order in which the salesman goes from one city to another, changing it seems like the most reasonable course of action.

**Crossover:** The same explanation given for mutation goes for the crossover. We used order-crossover for the algorithm.

**Selection:** The selection method is simple: when a result is better than another one, it is highly probable that just changing a little bit the order of this one will eventually result into a better solution. Therefore our selection method is the elite-selection. This method just takes the best individuals of each generation (if they are the best, they are fitted to survival) and brings them to the next one in addition to breed them randomly to generate another population of same size than the previous one. Note than our algorithm is not entirely elitist, since it actually allows mutation and crossover between our individuals after going to the next generation so a solution can be worse than the solution in the previous age.

**Cities:** A set of cities must be given to solve the problem. For this task, a new object, City, has been declared. Whenever a city is created, it’s added automatically to a list of cities called ‘cities’ which can be used later in the main method.

**Decode(Phenotype):** The chromosome will be decoded in order to perform certain functions into a list of cities. Each gene will be equivalent to the position of the city in the ‘cities’ list.

**Fitness:** The fitness function will be the sum of the distances of the cities following the order in which they are, including from the last on the list to the first one (Initial city has to be the last city as well). The method will penalize solutions in which two adjacent cities are not joint by any road (including last and first city).

## Methods

All the methods implemented and their use are as follow:

* **individual(length):** Receives an integer number and returns a list of numbers from 1 to **length** ordered randomly.
* **population(length, count):** creates a list of individuals of given **length**. The size of the population is **count.**
* **City(name, reach):** Creates a new city and adds it to the basic list of cities. The **name** is a string, and the **reach** a dictionary of string (names) and doubles (distances).
* **decode\_traveler(individual):** Receives an **individual** and returns a list of cities.
* **fitness\_traveler(individual):** Decodes an **individual** and return the fitness of the solution given by that individual.
* **grade\_traveler(population):** Returns the mean of the fitnesses of an entire **population**.
* **selection(population):** Returns the fifth best part individuals of the **population**. If there are less than 5 individuals in the population, it takes the first and the second best.
* **select\_individual\_crossover(population):** Returns two individuals inside a given **population**. These individuals will later breed into a new one. The first individual is picked at random. Then, the second one has to be the one before or after the first one to keep up with the ***Cellular Genetic Algorithm*** variation.
* **order\_mutation(individual):** performs the order-mutation operator over an **individual.** It randomly selects two cities, putting one before the other.
* **order\_crossover(individual1, individual2):** Performs the order-crossover operation over two individuals to generate a third one. Note that it only returns one of the children that can be generated through this process although this doesn’t really affect the algorithm performance. This operation takes several genes from the first parent and then fills with the remaining genes of the second parent in the order that they appear.
* **mutate\_population(population, chance):** mutates a whole **population.** Each individual has a given chance of being mutated so, theoretically speaking, we could have an entire population not being mutated.
* **evolve(population, chance):** selects, breeds, and mutates a whole **population** in order to generate a new one. This method uses the previously seen **select, order\_crossover and mutate\_population** to carry on with this task.
* **most\_suited(population):** Returns the individual with the best fitness (the minimun total distance) in a **population**.
* **genetic\_prob(ages, population\_size, chance, cities):** Returns the solved problem. It prints the elapsed time and gives us three attributes. The grade of the populations through the ages, the final population and the final route. This method uses all the previous methods to carry on its task. It first generates an initial population at random and evolves it once every age. When all the ages are passed, it returns the result, being it the most suited individual of the last population.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Ages | Population Size | Mutation Chance | Time Elapsed | Cost of Solution |
| 5 | 5 | 10% | 0.002s | 1576.6 |
| 10 | 5 | 10% | 0.0025s | 1401.0 |
| 50 | 5 | 10% | 0.01s | 1341.8 |
| 200 | 5 | 10% | 0.035s | 1212.0 |
| 1000 | 5 | 10% | 0.2s | 1212.0 |
| 5 | 10 | 10% | 0.002s | 1491.6 |
| 5 | 50 | 10% | 0.014s | 1350.1 |
| 5 | 200 | 10% | 0.046s | 1306.1 |
| 5 | 1000 | 10% | 0.237s | 1212.0 |
| 10 | 10 | 10% | 0.005s | 1425.8 |
| 50 | 50 | 10% | 0.088s | 1212.0 |
| 200 | 200 | 10% | 1.845s | 1212.0 |
| 1000 | 1000 | 10% | 56.083s | 1212.0 |
| 200 | 50 | 10% | 0.35s | 1212.0 |
| 10 | 50 | 10% | 0.025s | 1212.0 |
| 10 | 50 | 10% | 0.023s | 1212.0 |
| 10 | 50 | 10% | 0.023s | 1341.8 |
| 10 | 50 | 10% | 0.023s | 1306.6 |
| 50 | 10 | 10% | 0.024s | 1355.7 |
| 50 | 10 | 10% | 0.023s | 1355.7 |
| 50 | 10 | 10% | 0.024s | 1212.0 |
| 50 | 10 | 10% | 0.024s | 1212.0 |
| 10 | 10 | 20% | 0.0054s | 1212.0 |
| 10 | 10 | 20% | 0.008s | 1476.7 |
| 10 | 10 | 30% | 0.006s | 1476.7 |
| 10 | 10 | 30% | 0.005s | 1341.8 |
| 10 | 10 | 40% | 0.0061s | 1448.2 |
| 10 | 10 | 40% | 0.01s | 1306.6 |
| 10 | 10 | 50% | 0.009s | 1306.6 |
| 10 | 10 | 50% | 0.007s | 1212.0 |
| 10 | 10 | 100% | 0.005s | 1448.2 |
| 10 | 10 | 100% | 0.005s | 1355.7 |