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.

Alberto Bujalance Muñoz - Melchor Vázquez Martínez

07/09/2017

Tabla de contenido

[INTRODUCTION 2](#_Toc492927293)

[Motivation 2](#_Toc492927294)

[Software 2](#_Toc492927295)

[Hardware 2](#_Toc492927296)

[USER GUIDE 3](#_Toc492927297)

[THE ALGORITHM 5](#_Toc492927298)

[Introduction 5](#_Toc492927299)

[Solution Format 5](#_Toc492927300)

[Methods 6](#_Toc492927301)

[The Code 8](#_Toc492927302)

[Data Set 1 – AMD FX-6300 (6C/6T @ 3.5 Ghz) 12](#_Toc492927303)

[Data Sets– Intel i7-4790k (4C/8T @ 4.0 Ghz) 13](#_Toc492927304)

[Data Set 2 – Performance Comparison 15](#_Toc492927305)

[Behaviour study of the mutations 17](#_Toc492927306)

[CONCLUSIONS 20](#_Toc492927307)

[Time 20](#_Toc492927308)

[Results 20](#_Toc492927309)

[Format 21](#_Toc492927310)

[BIBLIOGRAPHY 22](#_Toc492927311)

# 

# INTRODUCTION

## Motivation

This practical assignment requires to develop, using Python, an implementation of genetic algorithms for solving the [Travelling Salesman Problem](https://en.wikipedia.org/wiki/Travelling_salesman_problem) -- TSP.

For this task, we had to find and choose amongst different genetic operators, Implement a variant over a classic GA (Cellular Genetic Algorithm) and made diverse experimentation complemented with some graphics to illustrate our results.

## 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>

For the documentation, we used [Microsoft Office 365](https://products.office.com/es-ES/compare-all-microsoft-office-products?tab=1&WT.srch=1&wt.mc_id=AID623582_SEM_FEWBTNWH).

## Hardware

We mainly used two computers for experimentation.

**Computer1**

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

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

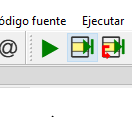
**Computer 2**

**CPU:** Intel i7 4790k 4.0 Ghz x 4 cores

**RAM:** G.Skill Ripjaws Z 16GB (4GBx4)DDR3 2400Mhz.

# 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:

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.

*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']}*

# THE ALGORITHM

## Introduction

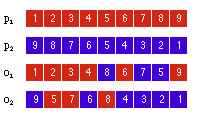
The algorithm is formed by a set of methods which are used in a main method called **genetic\_prob.** They can be classified in two types: **statistical** and **functional**. The main ones are functional, although some statistical methods are used at the end to help us gather the necessary experimentation 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)](#individual)**:** Receives an integer number and returns a list of numbers from 1 to **length** ordered randomly.
* [population(length, count)](#population)**:** creates a list of individuals of given **length**. The size of the population is **count.**
* [City(name, reach)](#city)**:** 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)](#decode)**:** Receives an **individual** and returns a list of cities.
* [fitness\_traveler(individual)](#fitness)**:** Decodes an **individual** and return the fitness of the solution given by that individual.
* [grade\_traveler(population)](#grade)**:** Returns the mean of the fitnesses of an entire **population**.
* [selection(population)](#selection)**:** 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)](#selectindividual)**:** 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)](#mutation)**:** performs the order-mutation operator over an **individual.** It randomly selects two cities, putting one before the other.
* [order\_crossover(individual1, individual2)](#crossover)**:** 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)](#mutatepop)**:** 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)](#evolve)**:** 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)](#prob)**:** 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.

## The Code

***import*** *random*

***import*** *time*

***import*** *math*

***def****individual****(****length****):***

***return*** *random****.****sample****(****range****(****0****,****length****),*** *length****)***

***def****population****(****length****,*** *count****):***

*pop* ***=******[*** *individual****(****length****)******for*** *x* ***in*** *range****(****count****)******]***

***return*** *pop*

*cities* ***=[]***

***class******City(****object****):***

***def*** *\_\_init\_\_****(****self****,*** *name****,*** *reach****):***

*self****.****name* ***=*** *name*

*self****.****reach****=*** *reach*

*cities****.****append****(****self****)***

***def****decode\_traveler****(****individual****):***

*dec* ***=******[None]\*****len****(****cities****)***

***for*** *i* ***in*** *range****(****len****(****individual****)):***

*dec****[****i****]******=*** *cities****[****individual****[****i****]]***

***return*** *dec*

***def*** *fitness\_traveler****(****individual****):***

*dec* ***=*** *decode\_traveler****(****individual****)***

*sol****=****0*

***for*** *i* ***in*** *range****(****len****(****dec****)):***

***if(****i****==****len****(****dec****)-****1****):***

***if(****dec****[****0****].****name* ***not******in*** *dec****[****i****].****reach****):***

*sol* ***=*** *sol* ***+*** *1000000*

***else:***

*sol* ***=*** *sol* ***+*** *dec****[****i****].****reach****[****dec****[****0****].****name****]***

***elif(****dec****[****i****+****1****].****name* ***not******in*** *dec****[****i****].****reach****):***

*sol* ***=*** *sol* ***+*** *1000000*

***else:***

*sol* ***=*** *sol* ***+*** *dec****[****i****].****reach****[****dec****[****i****+****1****].****name****]***

***return*** *sol*

***def****grade\_traveler****(****population****):***

*sum* ***=*** *0*

***for*** *i* ***in*** *range****(****len****(****population****)):***

*sum* ***=*** *sum* ***+*** *fitness\_traveler****(****population* ***[****i****])***

*sum* ***=*** *sum****/****len****(****population****)***

***return*** *sum*

***def****selection****(****population****):***

*pop\_len****=****len****(****population****)***

*cut* ***=*** *int****(****round****(****pop\_len****/****5****))***

***if(****cut****<=****1****):***

*cut****=****2*

*part* ***=******[None]\*****cut*

*fitnesses* ***=******[None]\*****pop\_len*

***for*** *i* ***in*** *range****(****pop\_len****):***

*fitnesses****[****i****]******=*** *fitness\_traveler****(****population****[****i****])***

*all* ***=*** *list****(****zip****(****fitnesses****,*** *population****))***

*all****.****sort****(****key****=lambda*** *tup****:*** *tup****[****0****],*** *reverse****=False)***

*sort\_pop* ***=******[****x****[****1****]******for*** *x* ***in*** *all****]***

***for*** *x* ***in*** *range****(****cut****):***

***if(****x****<****cut****):***

*part****[****x****]******=*** *sort\_pop****[****x****]***

***return*** *part*

***def****select\_individual\_crossover****(****population****):***

*rand1* ***=*** *random****.****randint****(****0****,****len****(****population****)-****1****)***

*rand2* ***=*** *random****.****randint****(****0****,****1****)***

*ind1* ***=*** *population****[****rand1****]***

*ind2* ***=******None***

***if*** *rand1* ***==*** *0****:***

*ind2* ***=****population****[****1****]***

***elif*** *rand1* ***==******(****len****(****population****)-****1****):***

*ind2* ***=*** *population****[****rand1****-****1****]***

***else:***

***if*** *rand2* ***==*** *0****:***

*ind2* ***=*** *population****[****rand1****-****1****]***

***elif*** *rand2****==****1****:***

*ind2* ***=*** *population****[****rand1****+****1****]***

*inds* ***=******(****ind1****,*** *ind2****)***

***return*** *inds*

***def****order\_mutation****(****individual****):***

*condition* ***=******True***

***while*** *condition****:***

*a* ***=*** *random****.****randint* ***(****0****,*** *len****(****individual****)-****1****)***

*b* ***=*** *random****.****randint* ***(****0****,*** *len****(****individual****)-****1****)***

*condition* ***=******(****a* ***==*** *b* ***or*** *a****>****b****)***

*part1* ***=*** *individual****[****0****:****a****]***

*part2* ***=*** *individual****[****b****]***

*part3* ***=*** *individual****[****a****]***

*part4* ***=*** *individual****[****a****+****1****:****b****]***

*part5* ***=*** *individual****[****b****+****1****:****len****(****individual****)]***

*new\_individual* ***=******[]***

***if(****len****(****part1****)!=****0****):***

***if(****len****(****part1****)==****1****):***

*new\_individual****.****append****(****part1****[****0****])***

***else:***

*new\_individual****.****extend****(****part1****)***

*new\_individual****.****append****(****part2****)***

*new\_individual****.****append****(****part3****)***

***if(****len****(****part4****)!=****0****):***

***if(****len****(****part4****)==****1****):***

*new\_individual****.****append****(****part4****[****0****])***

***else:***

*new\_individual****.****extend****(****part4****)***

***if(****len****(****part5****)!=****0****):***

***if(****len****(****part5****)==****1****):***

*new\_individual****.****append****(****part5****[****0****])***

***else:***

*new\_individual****.****extend****(****part5****)***

***return*** *new\_individual*

***def****order\_crossover****(****ind1****,*** *ind2****):***

*repeat* ***=******True***

***while*** *repeat****:***

*crosspoint* ***=*** *random****.****randint****(****0****,*** *len****(****ind1****)-****1****)***

*child* ***=*** *ind2****[:****crosspoint****]+****ind1****[****crosspoint****:]***

***if*** *set****(****ind1****)******==*** *set****(****child****):***

*repeat* ***=******False***

***return*** *child*

***def****mutate\_population****(****population****,*** *chance****):***

*new\_population* ***=******[]***

***for*** *i* ***in*** *population****:***

***if*** *chance* ***>*** *random****.****random****():***

***if*** *fitness\_traveler****(****i****)******!=*** *0****:***

*new\_population****.****append****(****order\_mutation****(****i****))***

*# new\_population.append(insert\_mutation[i])*

***else:***

*new\_population****.****append****(****i****)***

***return*** *new\_population*

***def****evolve****(****population****,*** *chance****):***

*pop\_len* ***=*** *len****(****population****)***

*part* ***=*** *selection****(****population****)***

*new\_size* ***=*** *pop\_len* ***-*** *len****(****part****)***

*new\_part* ***=******[None]\*****new\_size*

***for*** *i* ***in*** *range****(****new\_size****):***

*individuals* ***=*** *select\_individual\_crossover****(****part****)***

*p1* ***=*** *individuals****[****0****]***

*p2* ***=*** *individuals****[****1****]***

*individual* ***=*** *order\_crossover****(****p1****,*** *p2****)***

*new\_part****[****i****]******=*** *individual*

*res* ***=*** *part* ***+*** *new\_part*

*res* ***=*** *mutate\_population****(****res****,*** *chance****)***

***return*** *res*

***def*** *most\_suited****(****population****):***

*pop\_len* ***=*** *len****(****population****)***

*fitnesses* ***=******[None]\*****pop\_len*

***for*** *i* ***in*** *range****(****pop\_len****):***

*fitnesses****[****i****]******=*** *fitness\_traveler****(****population****[****i****])***

*all* ***=*** *list****(****zip****(****fitnesses****,*** *population****))***

*all****.****sort****(****key****=lambda*** *tup****:*** *tup****[****0****],*** *reverse****=False)***

*sort\_pop* ***=******[****x****[****1****]******for*** *x* ***in*** *all****]***

*suited* ***=*** *sort\_pop****[****0****]***

***return*** *suited*

***def*** *genetic\_prob****(****ages****,*** *pop\_size****,*** *mut\_chance****,*** *cities****):***

*time\_start* ***=*** *time****.****time****()***

***if(****pop\_size****>****1****):***

*c\_len* ***=*** *len****(****cities****)***

*pop* ***=*** *population****(****c\_len****,*** *pop\_size****)***

*grade* ***=******[None]\*****ages*

***for*** *i* ***in*** *range****(****ages****):***

*grade****[****i****]******=*** *grade\_traveler****(****pop****)***

*pop* ***=*** *evolve****(****pop****,*** *mut\_chance****)***

*suited* ***=*** *most\_suited****(****pop****)***

*result* ***=******[None]\*****c\_len*

*s\_dec* ***=*** *decode\_traveler****(****suited****)***

***for*** *i* ***in*** *range****(****c\_len****):***

*result****[****i****]******=*** *s\_dec****[****i****].****name*

*timer* ***=*** *time****.****time****()******-*** *time\_start*

*hours* ***=*** *math****.****floor****(****timer****/****3600****)***

*minutes* ***=*** *math****.****floor****(****timer****/****60****)******-*** *hours****\*****60*

*seconds* ***=*** *timer* ***-*** *minutes****\*****60* ***-*** *hours****\*****3600*

*timeStr* ***=*** *str****(****hours****)******+*** *'h '* ***+*** *str****(****minutes****)+*** *'m '* ***+*** *str****(****seconds****)******+*** *'s '*

***print(****'algorithm finished in: '* ***+*** *timeStr* ***)***

***return******{****'population'* ***:*** *pop****,*** *'grades'* ***:*** *grade****,*** *'result'* ***:*** *result****}***

***else:***

***print(****'THE POPULATION NEEDS TO BE AT LEAST 2.'****)***

## Data Set 1 – AMD FX-6300 (6C/6T @ 3.5 Ghz)

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 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 |

## Data Sets– Intel i7-4790k (4C/8T @ 4.0 Ghz)

|  |  |  |  |
| --- | --- | --- | --- |
| Ages | Population Size | Mutation Chance | Time Elapsed |
| 5 | 5 | 10% | 0.0004999637603759766s |
| 10 | 10 | 20% | 0.002491474151611328s |
| 15 | 10 | 20% | 0.0035178661346435547s |
| 15 | 15 | 30% | 0.0050051212310791016s |
| 15 | 20 | 30% | 0.00700688362121582s |
| 20 | 20 | 30% | 0.008519411087036133s |
| 20 | 20 | 40% | 0.01251530647277832s |
| 25 | 20 | 40% | 0.012025117874145508s |
| 25 | 20 | 50% | 0.01252436637878418s |
| 25 | 25 | 50% | 0.015527963638305664s |
| 25 | 25 | 60% | 0.0200345516204834s |
| 25 | 30 | 60% | 0.024538516998291016s |
| 30 | 30 | 60% | 0.022527217864990234s |
| 35 | 30 | 60% | 0.03002786636352539s |
| 35 | 40 | 60% | 0.03854870796203613s |
| 50 | 60 | 60% | 0.09035468101501465s |
| 100 | 70 | 50% | 0.18924617767333984s |
| 100 | 70 | 70% | 0.2137758731842041s |
| 100 | 30 | 90% | 0.10813474655151367s |

|  |  |  |  |
| --- | --- | --- | --- |
| Ages | Population Size | Mutation Chance | Time Elapsed |
| 10 | 10 | 20% | 0.00650787353515625s |
| 15 | 10 | 20% | 0.0055065155029296875s |
| 15 | 15 | 30% | 0.01001286506652832s |
| 20 | 20 | 30% | 0.014517784118652344s |
| 20 | 20 | 40% | 0.021025657653808594s |
| 25 | 20 | 40% | 0.022027254104614258s |
| 25 | 20 | 50% | 0.027534961700439453s |
| 25 | 25 | 50% | 0.033051490783691406s |
| 25 | 25 | 60% | 0.03103804588317871s |
| 25 | 30 | 60% | 0.047058820724487305s |
| 35 | 30 | 60% | 0.05156421661376953s |
| 35 | 40 | 60% | 0.0680849552154541s |
| 50 | 60 | 60% | 0.15068793296813965s |
| 100 | 70 | 50% | 0.2988746166229248s |
| 100 | 70 | 70% | 0.39694833755493164s |
| 100 | 30 | 90% | 0.1827716827392578s |

## Data Set 2 – Performance Comparison

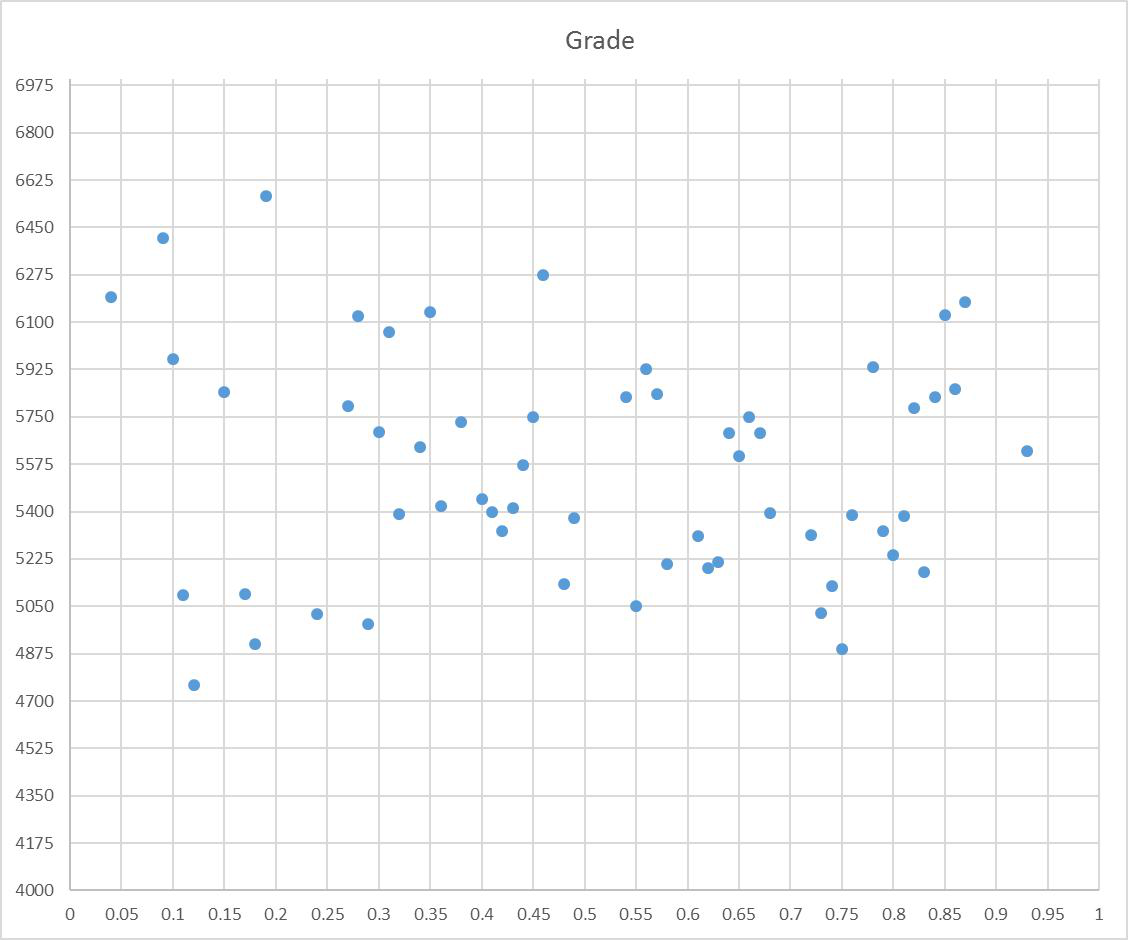
|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Ages | Population Size | Mutation Chance | Time AMD | Time Intel | Cost of Solution AMD | Cost of Solution Intel |
| 5 | 5 | 10% | 0.002s | 0.002003s | 14002732.0 | 11003369 |
| 10 | 5 | 10% | 0.0025s | 0.001501s | 11003786.5 | 8004239.9 |
| 50 | 5 | 10% | 0.015s | 0.007982s | 11002828.0 | 12002198.2 |
| 200 | 5 | 10% | 0.06s | 0.029537s | 5003943.2 | 5005219.9 |
| 1000 | 5 | 10% | 0.3s | 0.151690s | 1005823.1 | 2004574.4 |
| 5 | 10 | 10% | 0.005s | 0.002001s | 12002391.3 | 12003501 |
| 5 | 50 | 10% | 0.027s | 0.012014s | 8004002.8 | 7003464.5 |
| 5 | 200 | 10% | 0.11s | 0.0575714s | 7004448.8 | 7004476.1 |
| 5 | 1000 | 10% | 0.5s | 0.237287s | 6004615.5 | 5004737.3 |
| 10 | 10 | 10% | 0.009s | 0.003494s | 9003986.0 | 10002513.6 |
| 50 | 50 | 10% | 0.16s | 0.079088s | 5004057.9 | 2005575.2 |
| 200 | 200 | 10% | 2.71s | 1.559041s | 5681.6 | 1004875.1 |
| 1000 | 1000 | 10% | 1m 26.31s | 52.873924s | 1005190.5 | 6300.4 |
| 200 | 50 | 10% | 0.6s | 0.308385s | 1005744.2 | 6186 |
| 10 | 50 | 10% | 0.04s | 1.311168s | 9002844.6 | 5097.4 |
| 10 | 50 | 10% | 0.038s | 0.018012s | 7005469.0 | 5004297.7 |
| 10 | 50 | 10% | 0.044s | 0.020535s | 8003898.0 | 6003963.8 |
| 10 | 50 | 10% | 0.04s | 0.018020s | 7005079.3 | 9004002.2 |
| 50 | 10 | 10% | 0.038s | 0.0195245s | 7005419.7 | 7003736.3 |
| 50 | 10 | 10% | 0.038s | 0.016510s | 7004394.3 | 7003591.6 |
| 50 | 10 | 10% | 0.032s | 0.015529s | 5005055.4 | 8002964.6 |
| 50 | 10 | 10% | 0.036s | 0.016010s | 6003539.0 | 6004744.2 |
| 200 | 200 | 20% | 2.57s | 0.015510s | 1005041.7 | 8003382 |
| 200 | 200 | 20% | 2.7s | 1.377218s | 2004522.2 | 1004599 |
| 200 | 200 | 30% | 2.8s | 1.359795s | 5608.9 | 5652.9 |
| 200 | 200 | 30% | 2.9s | 1.438868s | 1005285.0 | 5468.2 |
| 200 | 200 | 40% | 3.42s | 1.9049496s | 2004468.5 | 5500.8 |
| 200 | 200 | 40% | 3.2s | 2.0190796s | 1004857.4 | 5879.7 |
| 200 | 200 | 50% | 3.6s | 1.654190s | 5677.1 | 1006537.3 |
| 200 | 200 | 50% | 3.61s | 1.854846s | 1005616.4 | 1005092.2 |
| 200 | 200 | 100% | 6.38s | 2.150208s | 2005449.1 | 6331.6 |
| 200 | 200 | 100% | 6.47s | 3.164473s | 1005971.6 | 1005654.7 |
| 1000 | 2000 | 20% | 2m 34s | 3.124570s | 5447.9 | 1006112.9 |
| 500 | 3000 | 20% | 2m 6s | 2.121737s | 1004799.7 | 1005145 |
| 1000 | 2000 | 50% | 4m 23s | 20.107631s | 5843.9 | 5758.3 |
| 500 | 3000 | 50% | 3m 28s | 1m 17.150226s | 5989.1 | 5931.7 |
| 2000 | 5000 | 30% | 12m 18s | 10m 27s | 5311.8 | 1004417.6 |
| 10000 | 50 | 30% | 34.16s | 1m 30s | 6045.5 | 5407.7 |
| 50 | 10000 | 30% | 36s | 25.392112s | 5281.5 | 5949.8 |
| 50 | 5000 | 30% | 18.7s | 14.852145s | 1005115.0 | 5084.8 |
| 50 | 5000 | 30% | 21.2s | 16.275572s | 4979.7 | 1005139.7 |
| 50 | 5000 | 30% | 22s | 10.234093s | 1005587.0 | 5434.6 |
| 50 | 5000 | 30% | 17.9s | 18.823772s | 5758.1 | 5566.4 |
| 50 | 5000 | 30% | 19.4s | 9.290736s | 1005101.1 | 6024.5 |
| 5000 | 50 | 30% | 19.3s | 9.4287848s | 1005250.3 | 5921 |
| 5000 | 50 | 30% | 19.1s | 9.582973s | 5945.9 | 5176.1 |
| 5000 | 50 | 30% | 21.3s | 9.260580s | 5964.7 | 5561.8 |
| 5000 | 50 | 30% | 19.7s | 9.586987s | 5918.7 | 5896.6 |
| 5000 | 50 | 30% | 20.5s | 8.021030s | 1004290.6 | 5668.1 |

## Behaviour study of the mutations

We used the second dataset, with a fixed population of 100, and an age value of 1000. We incremented the values for the mutations by 1% each time, and compared the time and the results.

|  |  |  |
| --- | --- | --- |
| **Mutation** | **Time** | **Grade** |
| 0.01 | 3.137451172 | 1005376.3 |
| 0.02 | 3.136056185 | 1005542.2 |
| 0.03 | 3.357459784 | 2005215.7 |
| 0.04 | 5.109558344 | 6194.4 |
| 0.05 | 3.330438852 | 1005125.6 |
| 0.06 | 3.339533806 | 1005254.9 |
| 0.07 | 3.805482388 | 1005558.6 |
| 0.08 | 4.05217123 | 1005676.6 |
| 0.09 | 3.849071026 | 6409 |
| 0.1 | 3.958406925 | 5965 |
| 0.11 | 3.418223381 | 5091.4 |
| 0.12 | 3.971525192 | 4758.1 |
| 0.13 | 3.637979269 | 1004300.5 |
| 0.14 | 3.547343969 | 1005264.8 |
| 0.15 | 3.556481838 | 5843.1 |
| 0.16 | 3.611516476 | 2004833.5 |
| 0.17 | 4.045490742 | 5094.8 |
| 0.18 | 3.928123474 | 4910.8 |
| 0.19 | 3.770686865 | 6565.1 |
| 0.2 | 4.48905158 | 1005381.4 |
| 0.21 | 3.893435001 | 1005234.5 |
| 0.22 | 4.163539886 | 2004663.9 |
| 0.23 | 4.433093786 | 1005057.7 |
| 0.24 | 3.593697071 | 5020.1 |
| 0.25 | 4.447141886 | 1005461.8 |
| 0.26 | 3.878845215 | 1006251.3 |
| 0.27 | 4.249248028 | 5788.8 |
| 0.28 | 4.09624362 | 6122.1 |
| 0.29 | 4.546109676 | 4983.9 |
| 0.3 | 4.279965401 | 5693 |
| 0.31 | 4.258892775 | 6062.2 |
| 0.32 | 4.005457878 | 5391.8 |
| 0.33 | 3.829885244 | 1005281.9 |
| 0.34 | 4.146991014 | 5639.4 |
| 0.35 | 4.841482639 | 6137.9 |
| 0.36 | 3.836588144 | 5420.8 |
| 0.37 | 3.8582623 | 1005813.9 |
| 0.38 | 4.369506121 | 5729.9 |
| 0.39 | 4.484093666 | 1004535.8 |
| 0.4 | 4.514813662 | 5446.6 |
| 0.41 | 3.954436064 | 5397.6 |
| 0.42 | 4.250844955 | 5329.8 |
| 0.43 | 4.577340364 | 5412.2 |
| 0.44 | 4.45214653 | 5571.4 |
| 0.45 | 4.778818607 | 5749.2 |
| 0.46 | 4.858976126 | 6273.5 |
| 0.47 | 4.033477783 | 2005441.4 |
| 0.48 | 4.690611362 | 5132.3 |
| 0.49 | 5.090416908 | 5375.7 |
| 0.5 | 5.000016689 | 1004636.4 |
| 0.51 | 5.335576296 | 1005658.1 |
| 0.52 | 4.816133022 | 1006021.9 |
| 0.53 | 4.561561823 | 1004730.7 |
| 0.54 | 4.637157917 | 5822.5 |
| 0.55 | 4.727406979 | 5050.3 |
| 0.56 | 4.994752645 | 5927.1 |
| 0.57 | 4.166218281 | 5833.8 |
| 0.58 | 4.969370127 | 5206.6 |
| 0.59 | 5.313681126 | 1005536.9 |
| 0.6 | 4.496293783 | 1005287.6 |
| 0.61 | 5.477864742 | 5311 |
| 0.62 | 4.535001278 | 5190.5 |
| 0.63 | 5.051811695 | 5212.4 |
| 0.64 | 5.532406569 | 5690.5 |
| 0.65 | 5.18422699 | 5606.8 |
| 0.66 | 6.146364927 | 5748.9 |
| 0.67 | 4.73033309 | 5690.7 |
| 0.68 | 5.063166857 | 5393.9 |
| 0.69 | 6.19960022 | 1005772 |
| 0.7 | 5.916298628 | 1005958.7 |
| 0.71 | 6.135194063 | 1005473.3 |
| 0.72 | 4.719962358 | 5311.8 |
| 0.73 | 4.792996645 | 5025.6 |
| 0.74 | 4.952252388 | 5124.2 |
| 0.75 | 5.48132062 | 4891.2 |
| 0.76 | 4.75211525 | 5388.2 |
| 0.77 | 4.688547611 | 1005158 |
| 0.78 | 5.337213516 | 5932.5 |
| 0.79 | 5.705574512 | 5328.1 |
| 0.8 | 5.882567644 | 5238 |
| 0.81 | 5.955734968 | 5383.6 |
| 0.82 | 5.954282999 | 5783 |
| 0.83 | 5.864335775 | 5178 |
| 0.84 | 6.070289612 | 5824 |
| 0.85 | 6.185899019 | 6124.8 |
| 0.86 | 6.364593744 | 5851.2 |
| 0.87 | 6.638770103 | 6173.4 |
| 0.88 | 6.587663889 | 1004626.4 |
| 0.89 | 6.854533672 | 1005203.9 |
| 0.9 | 6.942231894 | 1005929.7 |
| 0.91 | 7.022170305 | 2005350.1 |
| 0.92 | 7.260246515 | 1006441.2 |
| 0.93 | 6.979295492 | 5623.4 |
| 0.94 | 7.35837245 | 2005241.6 |
| 0.95 | 7.379751205 | 2004526.9 |
| 0.96 | 7.595736504 | 2005586.3 |
| 0.97 | 7.420787096 | 1005905.6 |
| 0.98 | 7.456779957 | 1005563.2 |
| 0.99 | 7.508932829 | 1006621.7 |
| 1 | 7.834355593 | 1005659 |

Obviously, as we solved this problem using genetic algorithm, and the mutations are random, we only can assure that making the elements mutate more, increments the time of performing the algorithm, and in exchange it doesn’t give us better noticeable performance.



# CONCLUSIONS

## Time

As shown in the [first graphic](#graph1), ages, population and mutation chance all vary the time for finishing the iterations. This is an obvious conclusion since the more ages and population, the more cases must be processed. The same happens with the mutation chance; as it increases, more cities will be mutated resulting in more time for each iteration. The important question is how much does the time vary with each change.

Checking the results for [Computer 1](https://d.docs.live.net/345df0a65d07ae12/Público), which made more ambitious test oriented in the results, we can appreciate that in the first cases the time is slightly changed, due to the base time still being significant. The important data here can be seen in later tests where is easily appreciated that the changes made in population and ages are proportionated with the increase in the elapsed time, except for some error depending on the status of the computer in that moment. This phenomenon is better appreciated in the last test cases.

Something similar happens with the mutation chance. In the tests of Data set 2 – [Computer 1](#computer1), it’s shown in a good way how this percentage alters the time and increases it but not in the same way than ages or population. It depends in a series of factors.

We can say that duplicating ages and population with the same mutation chance amounts to increasing the time x4 times. If we increase the mutation chance as we increase ages the performance is greatly lowered as we are making more iterations which imply more mutations. Same happens with population and mutation changes. As the population grows bigger, more individuals will mutate resulting in increasing the time significantly as well.

## Results

The tests we made are not really concluding in the matter of getting an optimal configuration in order to get an optimal solution or even getting a valid one with data set 2 containing few valid solutions among all its possible combinations (22 cities are equal to 22! combinations, which is an enormous number). Of course, it is appreciated that when any of the factors increases, the chances of getting an optimal or valid solution increases as well, but none of the combinations shed a light about how this exactly works. With further experimentation, with a specifically designed test machine and a couple of days we could maybe get an optimal configuration and estimate which values are more important in order to achieve a faster and better solution. Anyway, we made some extra experimentation with mutation chances, getting some interesting results that show that the first and last intervals (0-10%, 90-100%) in the data set 2 usually don’t get a valid solution. Our guess is that either without mutation or with too much mutation, it’s hard to get or to maintain a valid state and make it improve.

## Format

Although we couldn’t determine well how the results change from the introduced data, we learnt some things due to previous algorithms experiences and this new one.

Genetic algorithm proved to be a valuable tool provided that we have a really large data set and don’t need the best solution, only a good one. The problem comes when constraints are added since the randomness of the problem usually makes it to face up against a wall for some time before reaching a valid solution. In these cases, it would be better to use a search algorithm with some heuristic instead of making an evolutionary algorithm. This choice really depends on the number of constraints that your problem has or if they are totally prohibitive or only a little bit penalizing.

# BIBLIOGRAPHY

* *Artificial Intelligence: A Modern Approach. S. Russell and P. Norvig.*
* [Travelling Salesman - Wikipedia](https://en.wikipedia.org/wiki/Travelling_salesman_problem)
* [Selecting the most efficient genetic algorithm sets in solving unconstrained building optimization problem – Ali alajmi, Jonathan Wright](http://www.sciencedirect.com/science/article/pii/S2212609014000399)
* [Choosing Operators for TSP](http://www.permutationcity.co.uk/projects/mutants/tsp.html)
* [Google My Maps for measures](https://drive.google.com/open?id=17rWyYVO15no_shT7QGI9BrXVrq4&usp=sharing)