Traveling Salesman Problem (TSP)

Approach by Genetic Algorithms

Ernesto Guzmán Saleh

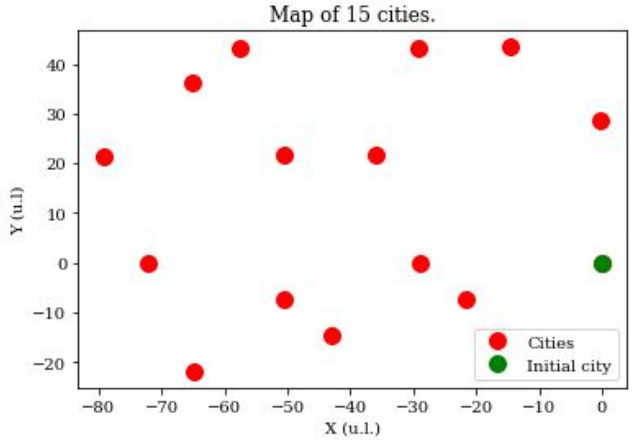
TSP

(See database [2])

The TSP is about finding the optimal (minimum distance) trajectory for a traveler across N places. This is an NP-Hard type of problem.

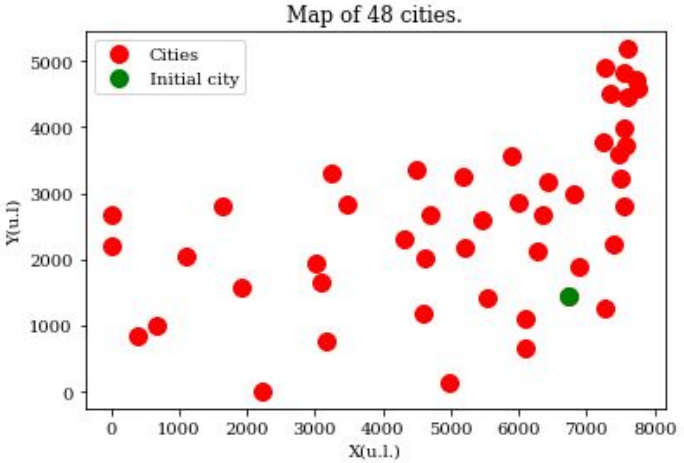
The traveler must return to its origin at the end.

Minimum trip:
D=291 (u.l.)



Minimum trip:
D=33523 (u.l.)

(u.l.): Units of length



Genetics Algorithms Structure

*In GeneticModel02 the crossover is replaced by **clonation**. {cloneMutation} which includes an intrinsic mutation.

*In GeneticModel03 there's both clonation {cloneMutation} and crossover {proCreate00}, {mutation01}.

Step. {Associated function}

Population Generation. {genPopulation}

Aptitude evaluation. {aptitudes}

Selection. {selection}

*Crossover. {proCreate00}

Mutation. {mutation01}

Filter. {filterPopulation}

Results.

Note that the figures included can vary in each run because of the natural randomness of the algorithm

Population Generation

Vector with n cities in traveled order. If is desired to return to the origin then the vector is of size n+1.

0	1	2	…	n-1	n	0

This vector is referred to as an **individual** inside a **population.**

These indexes may be in different orders, but the first index always keeps its first place.

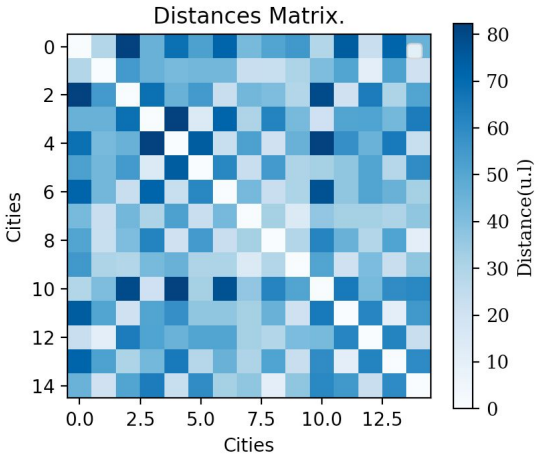
Same as the return point by default.

Aptitude of individuals.

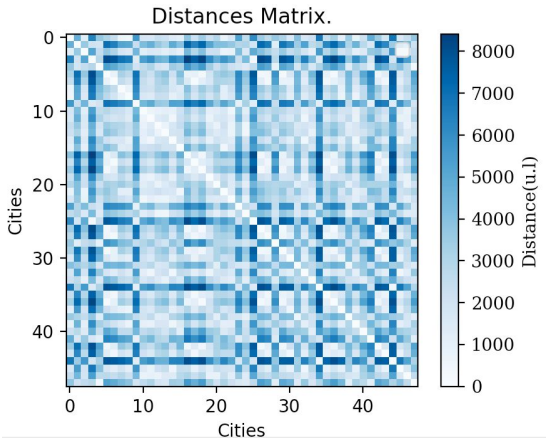
Is defined as the inverse of the traveled distance of an individual. The distance is extracted from a predefined **Distance Matrix (DM).** Returns an aptitudes vector in index order that represents the population.

DM.

DM[***i,j***] represents the distance from city ***i*** to city ***j***



15 cities



48 cities

Selection of individual

Selects the individuals with most aptitude by a **roulette wheel selection**.

A cumulative sum vector is created from the aptitudes vector that represents the population.

A random float is generated from an uniform distribution in range [0, sum of aptitudes].

The random float selects the bigger closest number to itself in the cumulative vector and extracts an index.

That index is the one of the selected individual of a population.

GeneticModel01: Crossover.

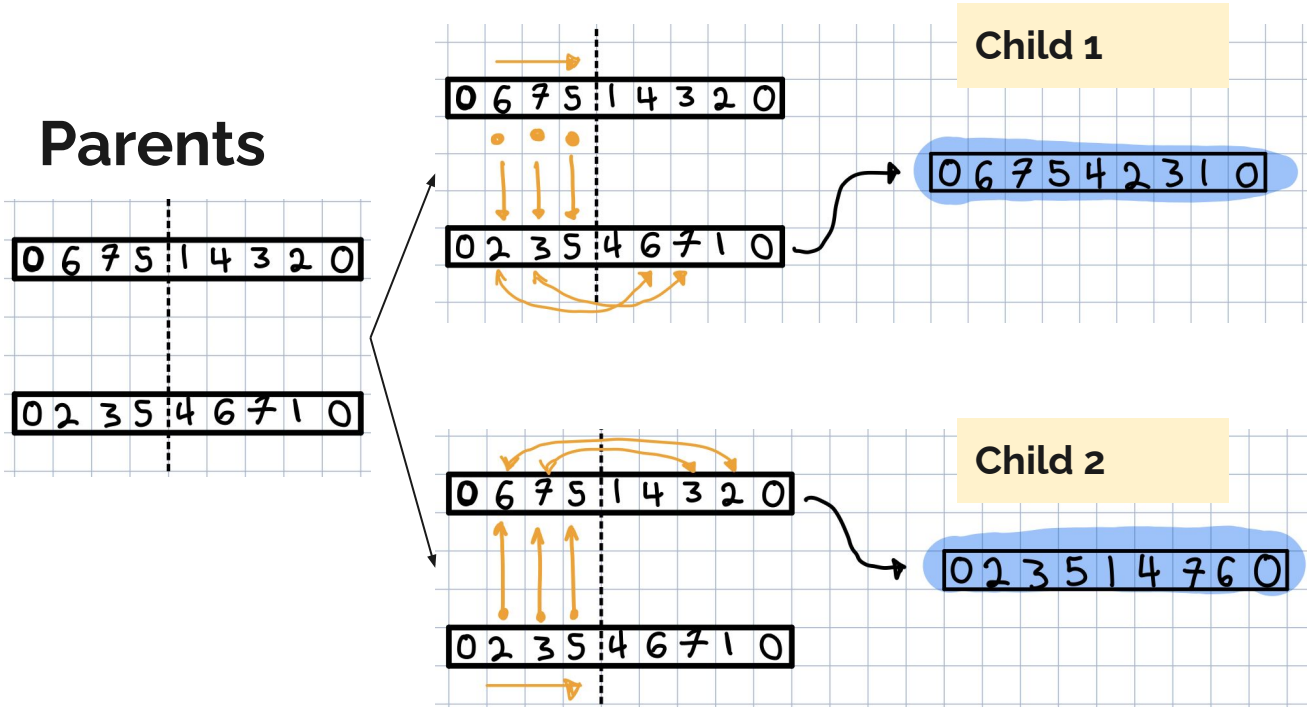
Crossover{proCreate00}

Crosses two different selected individuals of the population **(pop).** Generates either 2 or 4 **children** depending of the parameter **dbKids. Crossover process in next slide.**

Mutation {mutation01}

Shuffles 2 random values in an individual with a probability **P** of occurring.

Crossover



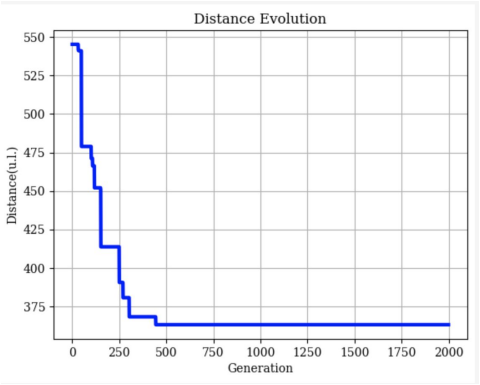
If dbKids=True.
When the diagram process is finish, the second half shuffles to the first one and the process is repeated.
Generates 4 children instead of 2.

15 Cities

```
popSize=30; iters=2*10**3
```

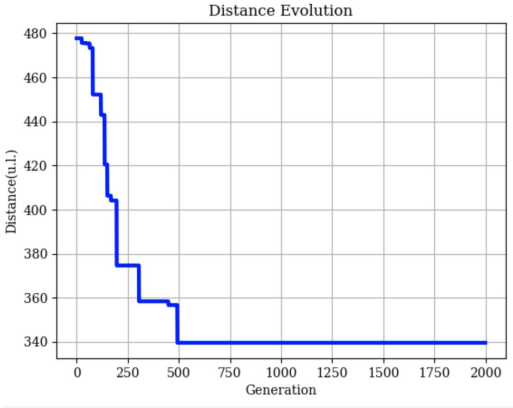
P=0.5
dbKids=False

D=363.123 u.l.
time=1.1s



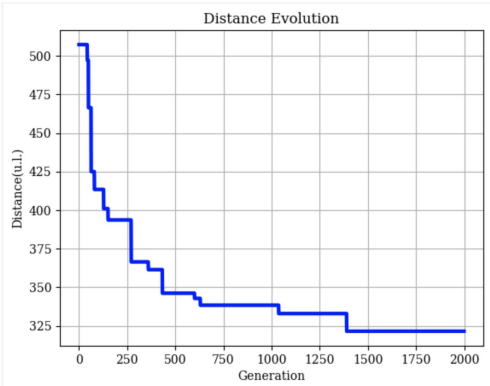
P=0.5
dbKids=True

D=339.52 u.l.
time=1.1s



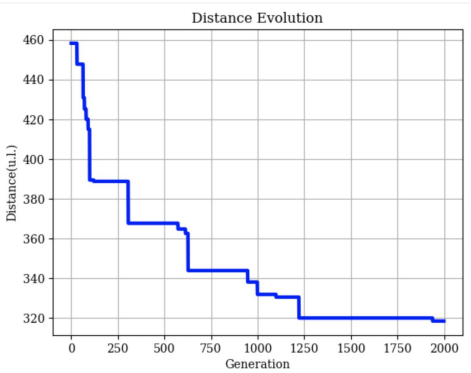
P=1
dbKids=False

D=321.584 u.l.
time=0.7s



P=1
dbKids=True

D=318.517 u.l.
time=1.2s

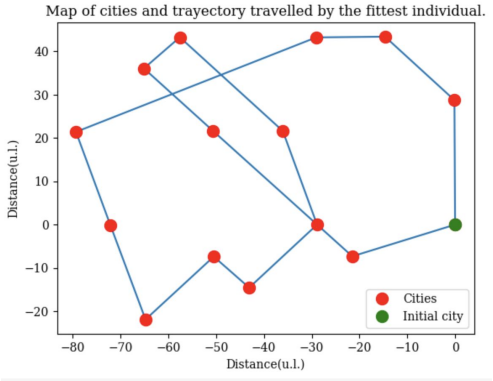


15 Cities

```
popSize=30; iters=2*10**3
```

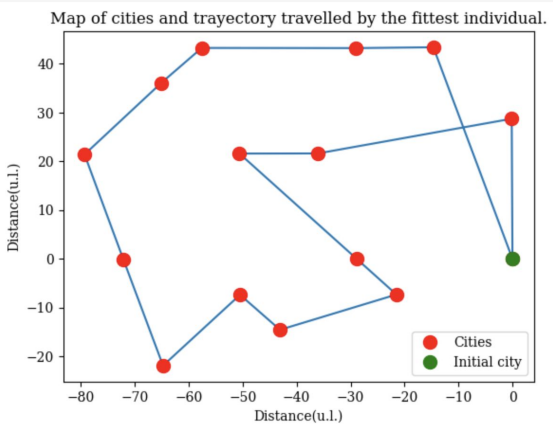
P=0.5
dbKids=False

D=363.123 u.l.
time=1.1s



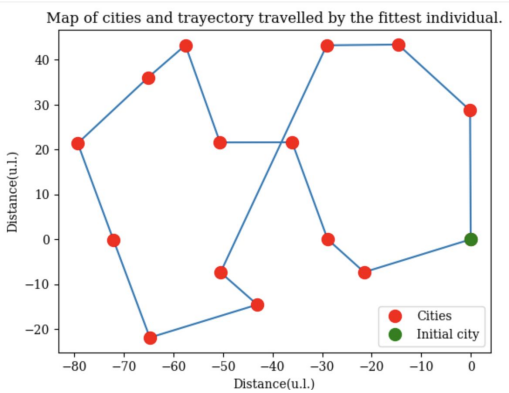
P=0.5
dbKids=True

D=339.52 u.l.
time=1.1s



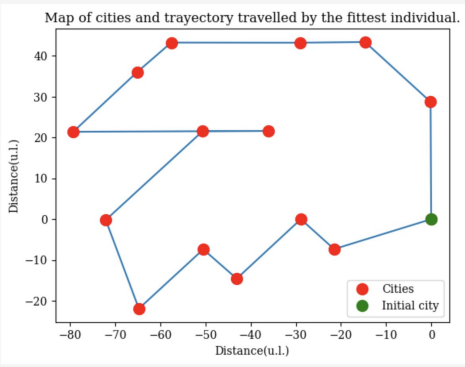
P=1
dbKids=False

D=321.584 u.l.
time=0.7s



P=1
dbKids=True

D=318.517 u.l.
time=1.2s

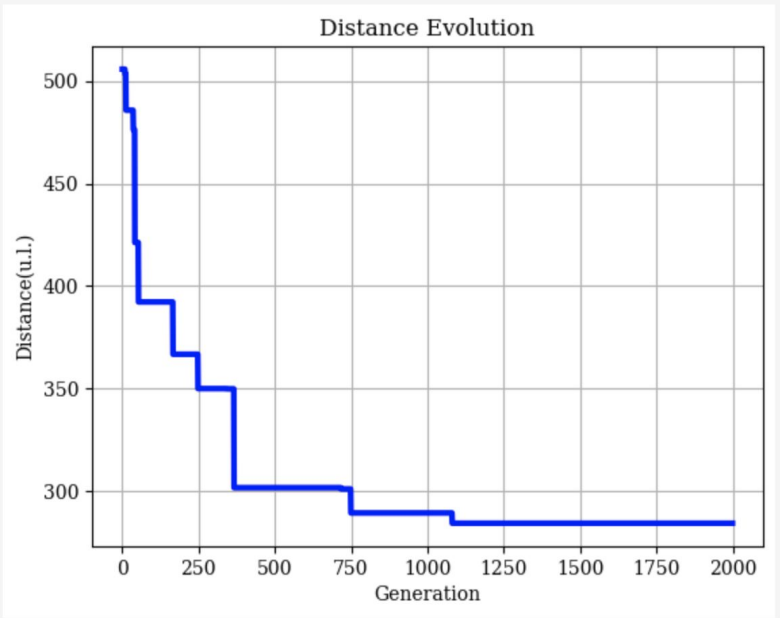


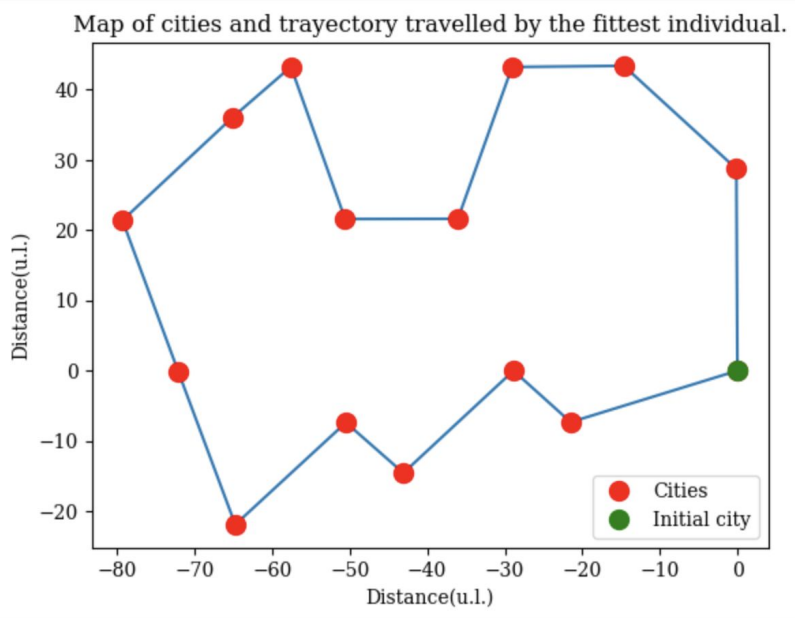
15 Cities

```
popSize=30; iters=2*10**3
```

Params: P=1, dbKids=True **time=1.2s**

This is the optimal solution. A way to ensure it out of randomness is to increase the number of iters.





48 cities

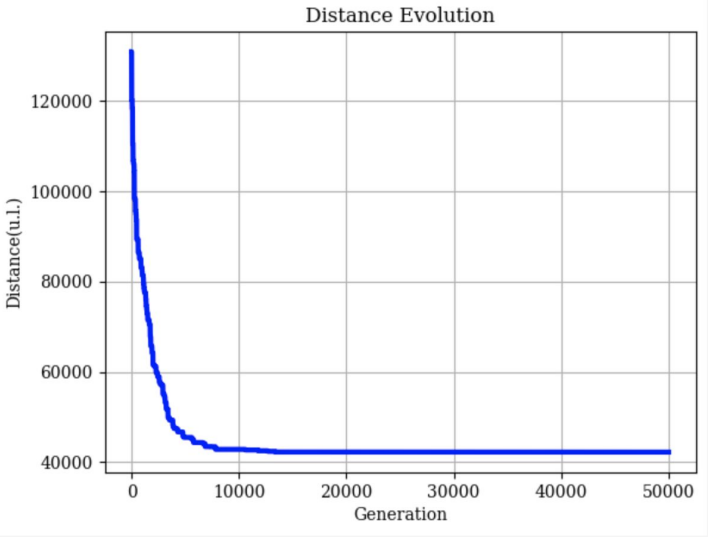
```
popSize=50; iters=5*10**4
```

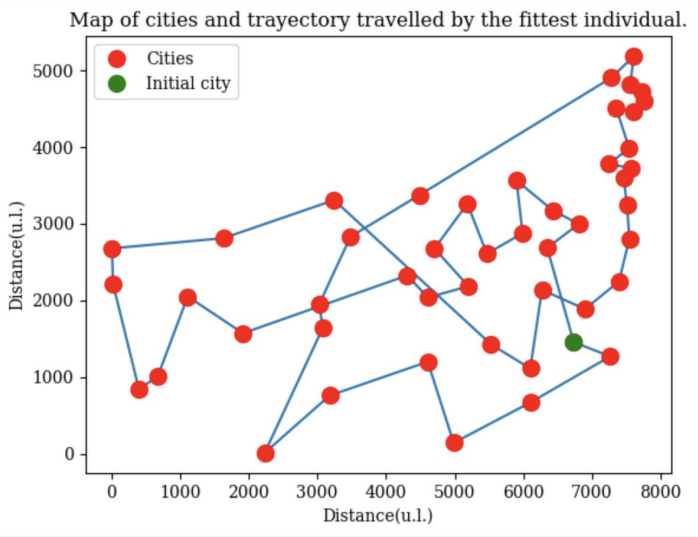
Intento 1: time=50.6s

Aunque todos los parámetros son válidos para diferentes problemas, atacamos este que es mayor con los parámetros anteriores.
Aumentamos la población para más diversidad y las iteraciones para mejor convergencia.

```
Distance:
42204.49627
u.l.
```

Dada una rápida convergencia, hacerlo para más iteraciones no tiene caso.



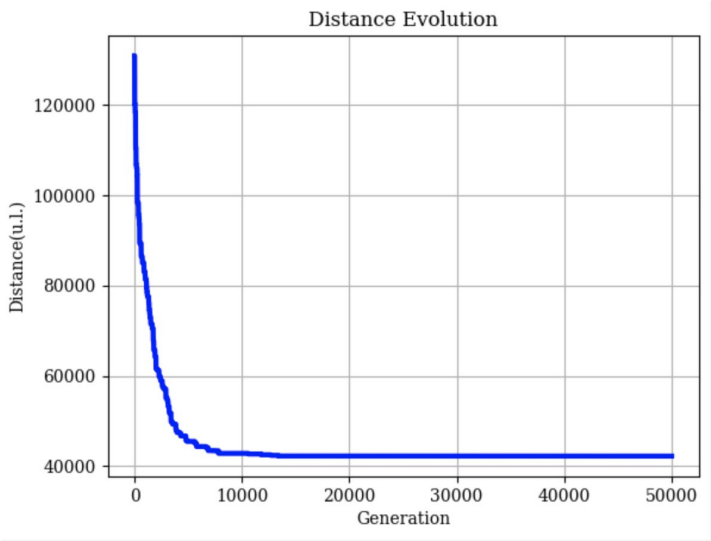


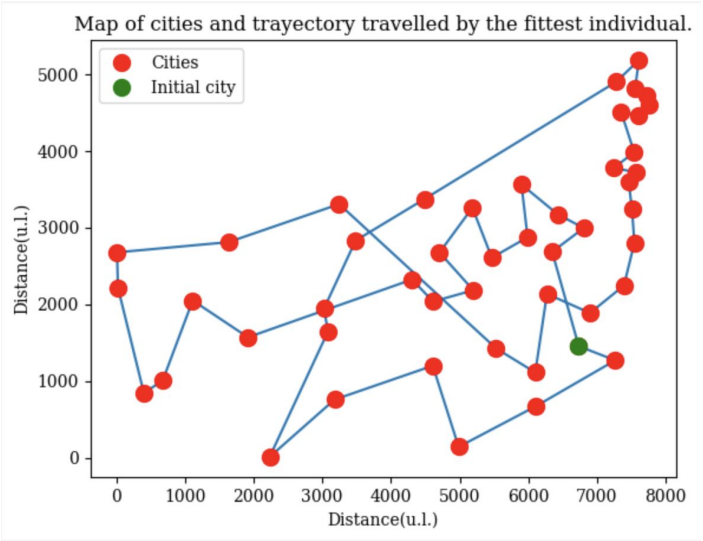
48 cities

```
popSize=50; iters=5*10**4
```

attempt 2: time=52.1s

```
Distance: 46421.06u.l.
```





The graph converges at 10000. Considering this proportion: time=**10.42s**

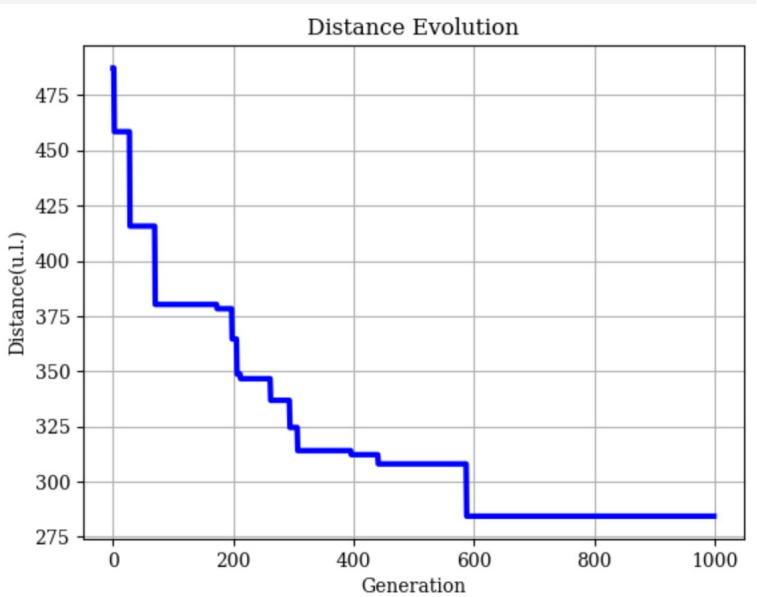
GeneticModel02: Clonation

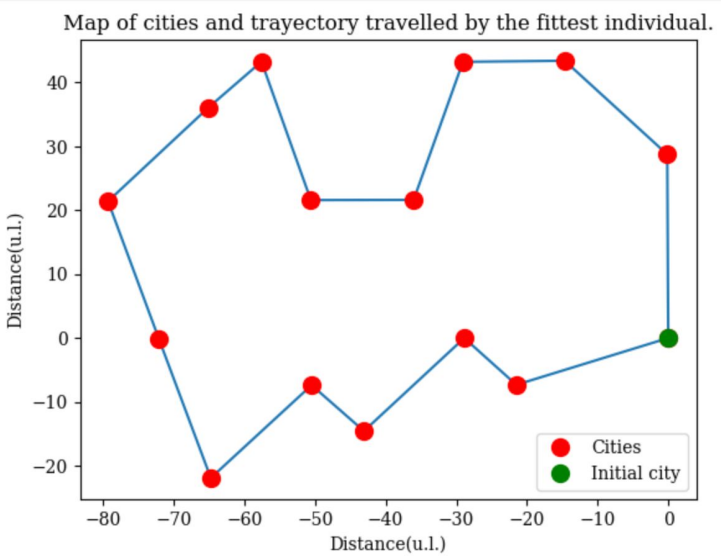
The classic steps of genetic algorithms are complied. With the exception that the crossover is replaced by a clonation of selected individuals (cloneKids) and then mutations {cloneMutation}. This mutations differ from the ones in {mutation01}. The integer mutsPerKid determines how many shuffles an individual suffers.

This idea was first implemented by https://github.com/fernandosangerman in a team class project. However the code was then rewritten by me so it fits with the other functions.

```
popSize=50; iters=10**3
time=0.7s
cloneKids=10; mutsPerKid=2
```

D=284.38086286247795 u.l.



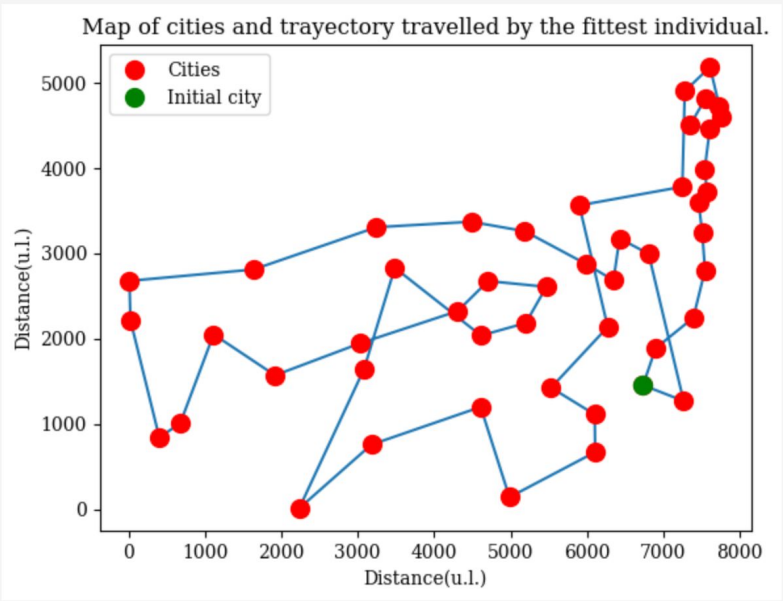


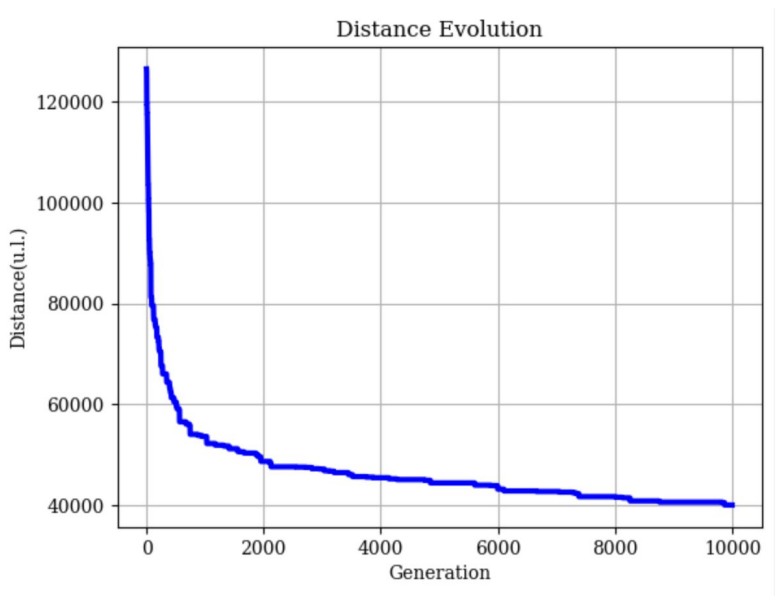
```
popSize=100; iters=10**4
```

```
time=40.2s
```

D=39971.383 u.l.

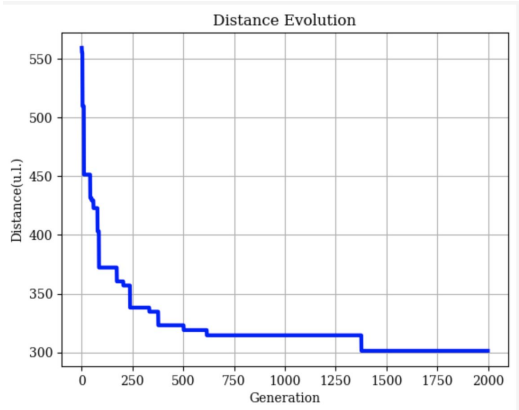
```
cloneKids=10;
mutsPerKid=2;
```

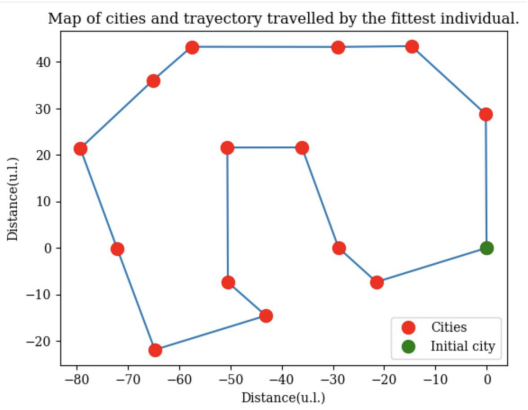


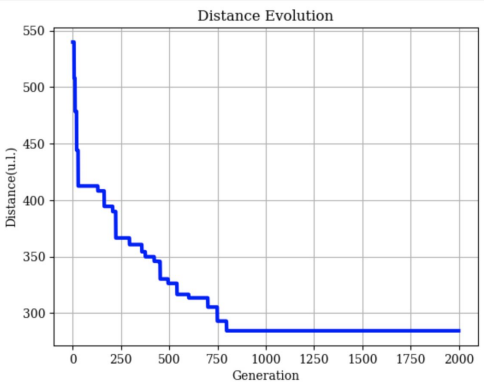


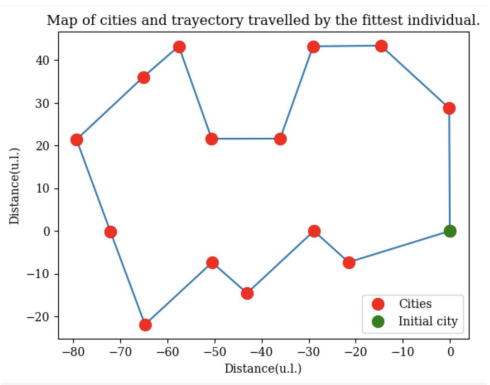
GeneticModel3: Crossover and Clonation

This is a combination of the previous 2 models. A new variable is added: natural couples (natCouples). which represents the number of couples to make offsprings. These couples are formed according to the selection function. If natCouples=2 and the selected individuals are [1,2,3] then 1,2 and 2, 3 will crossover.









15 cities.

`time=2.6s`

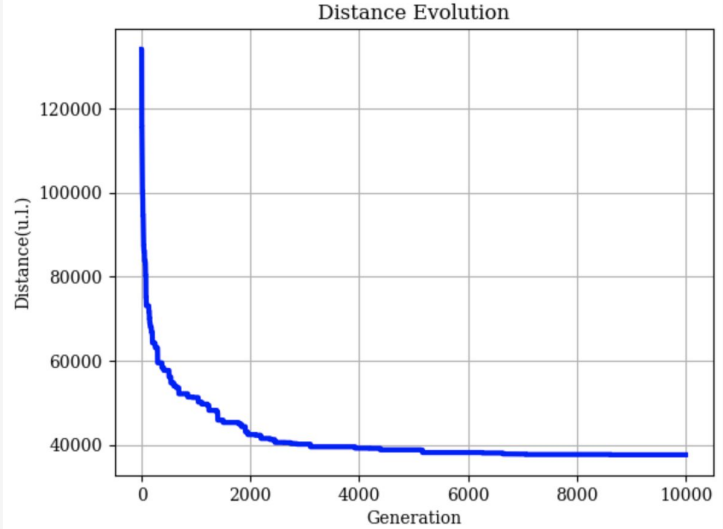
- popSize=50
- iters=2*10^3
- mutsPerKid=2
- dbkids=True
- natCouples=2
- cloneKids=5

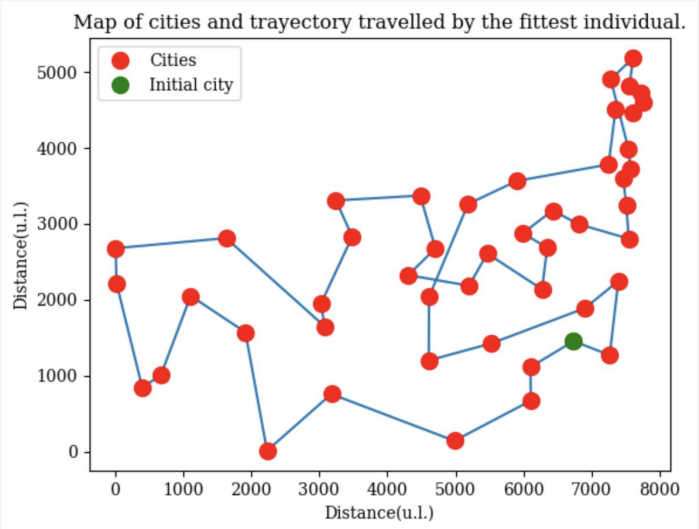
Some samples of running these parameters. Most of the time the optimal is given.

For 18 cities

`time=105.9s`

- popSize=100
- iters=10^4
- mutsPerKid=2
- dbkids=True
- natCouples=7
- cloneKids=50





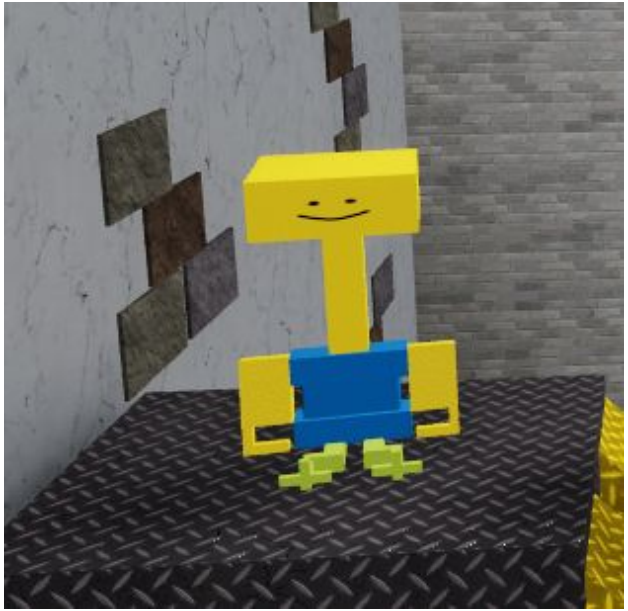
`Distance=37615.240 (u.l.)`

Recommendations and observations

Both models 1 and 2 converge at the same time and same iterations with the proper parameters. They're good for easy problems since the convergence is fast.

Model 3 convergence is slow. The benefit is that the algorithm doesn't get biased in a solution to quick, making it useful for more complex problems.

The offspring solutions:



Sources

An example of TSP by genetic algorithms:

[1] Traveling salesman problem using genetic algorithm. GeeksforGeeks. (2021, July 2). Retrieved December 6, 2021, from https://www.geeksforgeeks.org/traveling-salesman-problem-using-genetic-algorithm/.

The database of the presented data:

[2] TSP data for the traveling salesperson problem. TSP - Data for the Traveling Salesperson Problem. (n.d.). Retrieved December 6, 2021, from https://people.sc.fsu.edu/~jburkardt/datasets/tsp/tsp.html.

The type of problem explanation:

[3] ¿Qué significa 'p vs Np' para el resto de nosotros? MIT Technology Review. (2017, August 17). Retrieved December 6, 2021, from https://www.technologyreview.es//s/1374/que-significa-p-vs-np-para-el-resto-de-nosotros.

Fernando's Github as credits for its apportation in GeneticModel02:

[4] https://github.com/fernandosangerman