Labo MLG - Genetic Algorithms

A tool for optimization

David Kunzmann and Toni Dias

HEIG-VD

19.06.2016

6.1 - Problem (TPS)

The travelling salesman problem (TSP) asks the following question: Given a list of cities and the distances between each pair of cities, what is the shortest possible route that visits each city exactly once and returns to the origin city? It is an NP-hard problem in combinatorial optimization, important in operations research and theoretical computer science. The objective of this exercise is to solve the TSP problem for a set of 14 cities in Burma, officially the Republic of the Union of Myanmar.

Our Solution

In our solution, we used haversine a pypi library. This library helps us to calculate the distance between two GPS points. Source: https://pypi.python.org/pypi/haversine

We initialize our genome with 14 number form 0 to 13 (each value represents one GPS point, one city). What we are looking for is the order of the number. When we initialize our genome we execute a swap function so that every time we start with a random order. We calculate the fitness value with the total distance of the travelling. If the chromosome contains all cities points we keep a good fitness. But if the chromosome contains one city multiple times and don't have all the cities we put a big penalty.

6.2 - Results

For all our results we used a seed of 123. Our better travelling is the GPS points in this order with the link between the last and the first point.

```
[10, 8, 9, 0, 1, 13, 2, 3, 4, 5, 11, 6, 12, 7]
```

the travel distance is:

```
3354.52313748 km

We believe this is the best solution there are 14! possible solutions. Of
course the goal of our genetic algorithm is not to test all these
possibilities but to keep only the best one. So what makes us think it's the
best solution ? It's because with more generations and with a bigger
population still we obtain 3354.52 as the best result. What we know tough is
that a genetic algorithm doesn't give us the perfect solution it only gives
us a good solution.  So it's only by testing that we think it's the best
solution of course we could be wrong and our solution is just a local minima
and not the global we are searching for.
```

6.3 - Fitness function

For the fitness function, we decide calculate the total distance between points. This value is our fitness value. If all GPS points are on the chromosome (the travel goes through each city) we calculate the total distance. And return a big value (in our code 15000) and subtract the total distance. This method return a big number if the total distance is smaller. This number 15000 was found after several tests, the fitness could not be bigger than this number. So we used it to return a better fitness score for a smaller distance.

But if the chromosome contains duplicate GPS points (and so goes through a city more than once) we put a penalty. We calculate it like that:

```
1 - (length of a chromosome - number of different values in the chromosome)
* 0.05
```

The fitness is in the range]0:1[. This is so big penalty and we wish the chromosome aren't selected for the next generations. We this calculation if a city is more then once in our array then the penalty is bigger. This force our algorithm to generate child with less and less duplicate cities.

6.4 - Explain the way you encoded the solution

Chromosome example

A good chromosome, he has all cities: **[0,1,2,3,4,5,6,7,8,9,10,11,12,13]**

A bad chromosome, he hasn't all cities: **[0,1,2,0,4,5,6,7,8,9,0,11,12,13]**

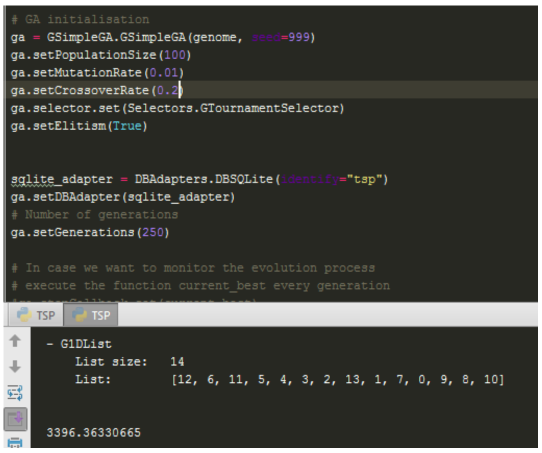
6.5 - Provide the configuration of the GA

To find our better solution, we used the parameters:

- mutation: 0.01
    - 0.01 because if we augment the mutation, we lose the good chromosome

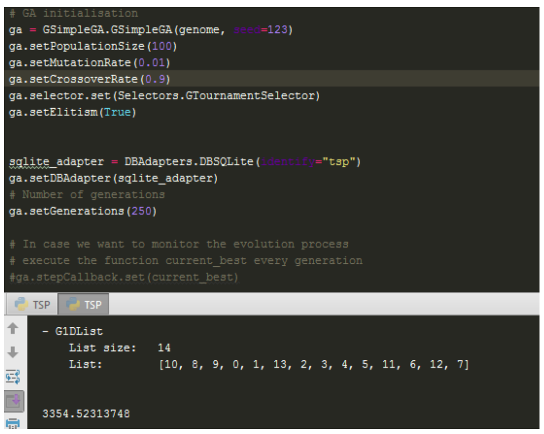
after the mutation. In example, we have a chromosome with all cities if it mutate, this chromosome isn't more a good chromosome.

- crossover: 0.9 as expected a high crossover gives better result. As we can see in the following pictures:



```
# GA initialisation
ga = GSimpleGA.GSimpleGA(genome, seed=999)
ga.setPopulationSize(100)
ga.setMutationRate(0.01)
ga.setCrossoverRate(0.2)
ga.selector.set(Selectors.GTournamentSelector)
ga.setElitism(True)


sqlite_adapter = DBAdapters.DBSQLite(identify="tsp")
ga.setDBAdapter(sqlite_adapter)
# Number of generations
ga.setGenerations(250)

# In case we want to monitor the evolution process
# execute the function current_best every generation
```

```
 - G1DList
    List size:   14
    List:        [12, 6, 11, 5, 4, 3, 2, 13, 1, 7, 0, 9, 8, 10]


 3396.36330665
```



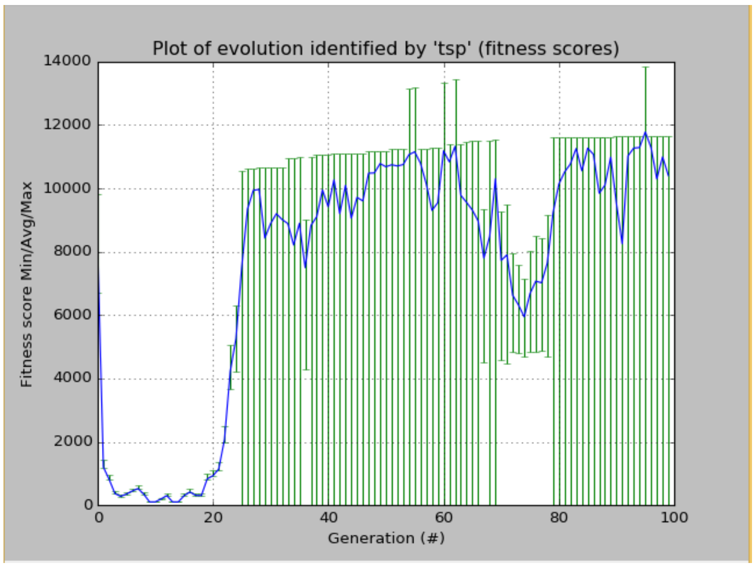
```
# GA initialisation
ga = GSimpleGA.GSimpleGA(genome, seed=123)
ga.setPopulationSize(100)
ga.setMutationRate(0.01)
ga.setCrossoverRate(0.9)
ga.selector.set(Selectors.GTournamentSelector)
ga.setElitism(True)


sqlite_adapter = DBAdapters.DBSQLite(identify="tsp")
ga.setDBAdapter(sqlite_adapter)
# Number of generations
ga.setGenerations(250)

# In case we want to monitor the evolution process
# execute the function current_best every generation
#ga.stepCallback.set(current_best)
```

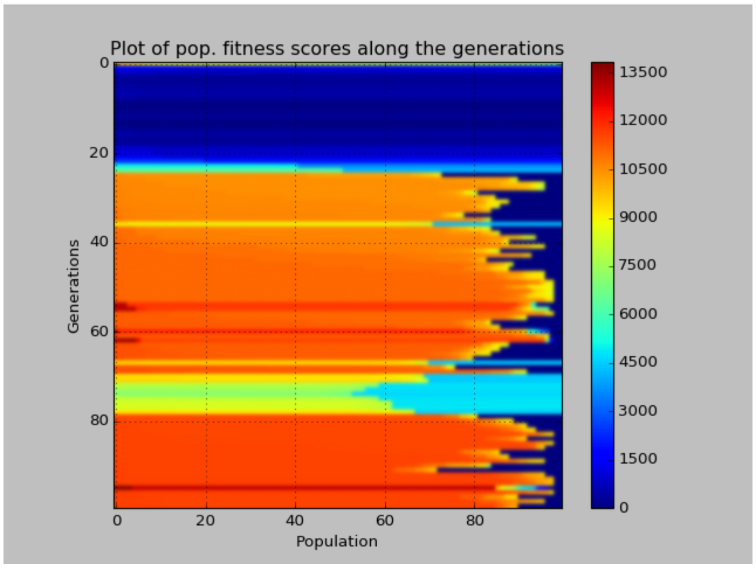
```
 - G1DList
    List size:   14
    List:        [10, 8, 9, 0, 1, 13, 2, 3, 4, 5, 11, 6, 12, 7]


 3354.52313748
```

- population size: 100
  - after a lot of tests, we concluded that: with a population of 100 we have the same results as a bigger population, but faster.
- type of selection: GTournamentSelector
  - We choose this selector because after several tests this gave us the best results. We suppose the reason is when the first chromosome with a good fitness is found, the algorithm takes its parents and replace them. At the end, we have a "local" minima and this is not the best.
- number of generations: 250
  - when we make a tests with 500 generations, we see after about 220 generations the best fitness was found. So we choose 250 to have a little margin.

6.6 - Plots, experimentations and explanations

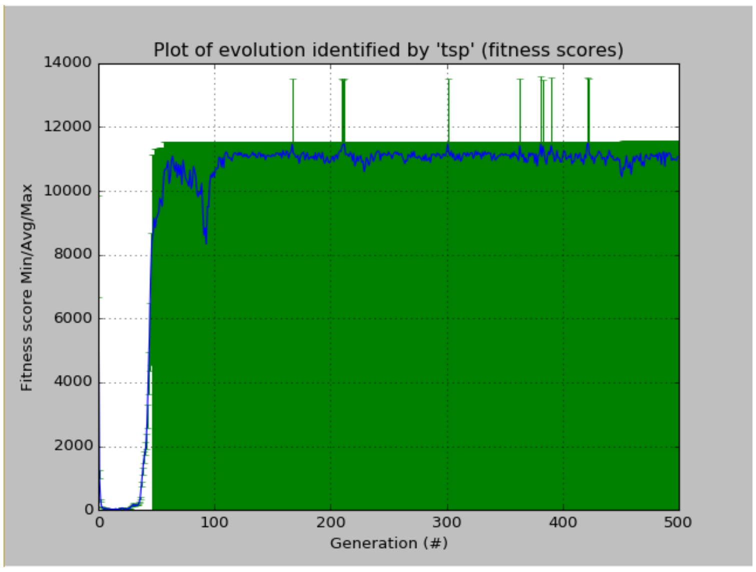
100 Generations and 100 populations The First (for reference)

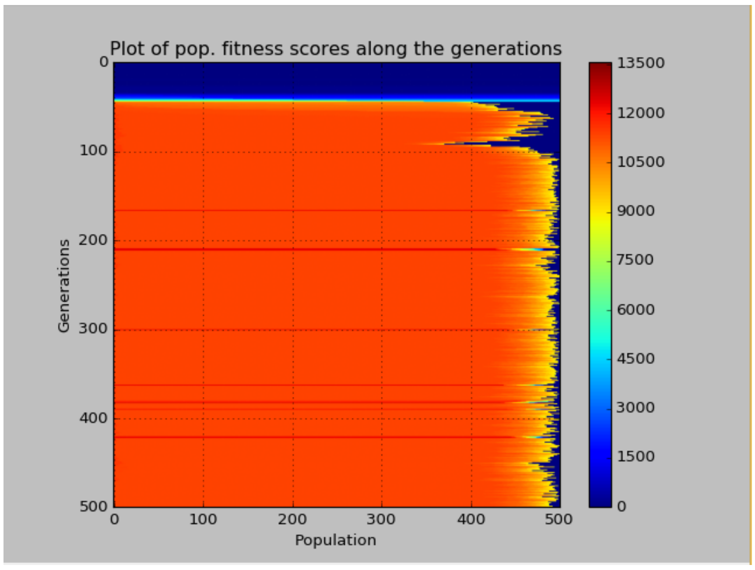




We can see in the first chart the range of fitness is very big. This is because we put a big penalty if the chromosome don't have all cities. In the second chart, we see the first 20 generations don't have only one good chromosome this is because when we generate chromosomes we have a little chance to have a chromosome with 14 GPS points difference.

500 Generations and 500 populations





We can see the fitness score don't evolve a lot after 100 generations, the program found the best solution and don't change there. The same can be observed in the second chart. After a good solution is found (a chromosome with 14 GPS points difference), the evolution don't change a lot after 100-120 generations.

6.7 - Conclusion

We appreciated a lot to work on this project. We could practice the theory view during classes and made a complete program with a result at the end. This was indeed the best project of the semester and also the most interesting.

This course introduces us the machine learning programing and we are happy to have the chance to learn that. It's a discipline that grows up very fast and gives us a useful skill for our professional life. This project is a good example for us in order to grasp the main concept of evolutionary algorithms.