Inteligencia Artificial: “Algoritmos genéticos”

Alexander Krasnoshchkov
David Rozas Domingo

Curso : 08/09

Introduction

The following document describes the process of creation of an application which implements a genetic algorithm to minimize the cost of assigning a serial of sports to a set of localizations in a planification of the Olympic Games in Madrid.

The program has been developed in Java 6 using Eclipse IDE under Ubuntu 8.10. The UML diagrams were made using Umbrello 2.1.3. Program has been tested successfully under Linux and Windows XP.

Program design

The following decisions related to program design have been taken:

- Use of an XML configuration file for all parameters. SAX (Simple Api for XML) libraries have been used for reading the file.
- Two different ways of calculating the fitness function have been implemented:
  - F = K-g(x), K>>g(x)
  - F = 1/g(x)

    The first selection function can be chosen writing 1 in the field <fitnessType> of the configuration file, and the second function writing 0.
- The gens crossing process has been developed with a similar idea as the one of the "Traveler problem" explained in a lecture:
  - A random r number is calculated. We copy the first r gens from the first parent.
  - Later we fill the rest of the chromosome copying the gens from the second parent which are not already stored in the new chromosome.
- For the mutation process, we calculate a couple of different random values (r1 and r2) and we exchange the values of the arrays in the position r1 and r2. The percentage can be chosen writing a number between 0.0 and 1.0 in the field <mutationProbability> in the configuration file.
- Two different selection policies have been implemented: Roulette and Boltzmann. We can choose between them by writing "roulette" or "boltzmann" in the configuration file. For Boltzmann it is necessary also to write a float number in the field <temperature>.

  The implementation is quite similar in both cases (differing in the formula): calculate the sumatory

first, and the probability for each gen afterwards.

- ➔ The possibility of using elitism has been implemented as well. The percentage can be chosen writing a number between 0.0 and 1.0 in the field <elitism> in the configuration file.
  Before creating a new population, the best individuals (biggest fitness) from the previous generation are marked. Then we include them in the next generation directly, together with the new chromosomes created in the crossing process.
- ➔ For the output two functionalities have been created:
  - ➔ A chart which is shown just after the algorithm execution has finished, showing the best and the average cost evolution. The SWT (Standard Widget Tool) from eclipse libraries have been used for that purpose.
  - ➔ A log with average and best cost for each generation is stored in the path configured in <logPath>. The values separated by the symbol “;”, and are stored with .csv extension, so can be opened directly with OO Spreadsheet or MS Excel.

Experimentation and discussion

We are going to do some experiments with extreme values in order to analyze the influence of each parameter. Finally, we offer a configuration for getting “good values” (less than 18K).

Comparing different selection policies

Roulette:

```
The best individual is
Genes 14 | 15 | 6 | 13 | 10 | 5 | 7 | 16 | 4 | 11 | 9 | 12 | 17 | 3 | 18 | 8 |
0 | 1 | 2 |
Cost: 18582288
Fitness: 5.38146863292615E-8
The parameters used were :
Selection policy: roulette
Population size : 1000
Generations : 100
Mutation probability : 0.01
The fitness function type is:  1/g(x)
Elitism percentage = 7.0%
```

With roulette members with a high fitness have more probability of being selected, therefore the average and minimum cost converge very quickly.

Boltzmann selection policy

It is difficult to set up what are big and small values for temperature in Boltzmann, but after previous experiments we have determined that a small value is around -1000 and high values are around +1000.

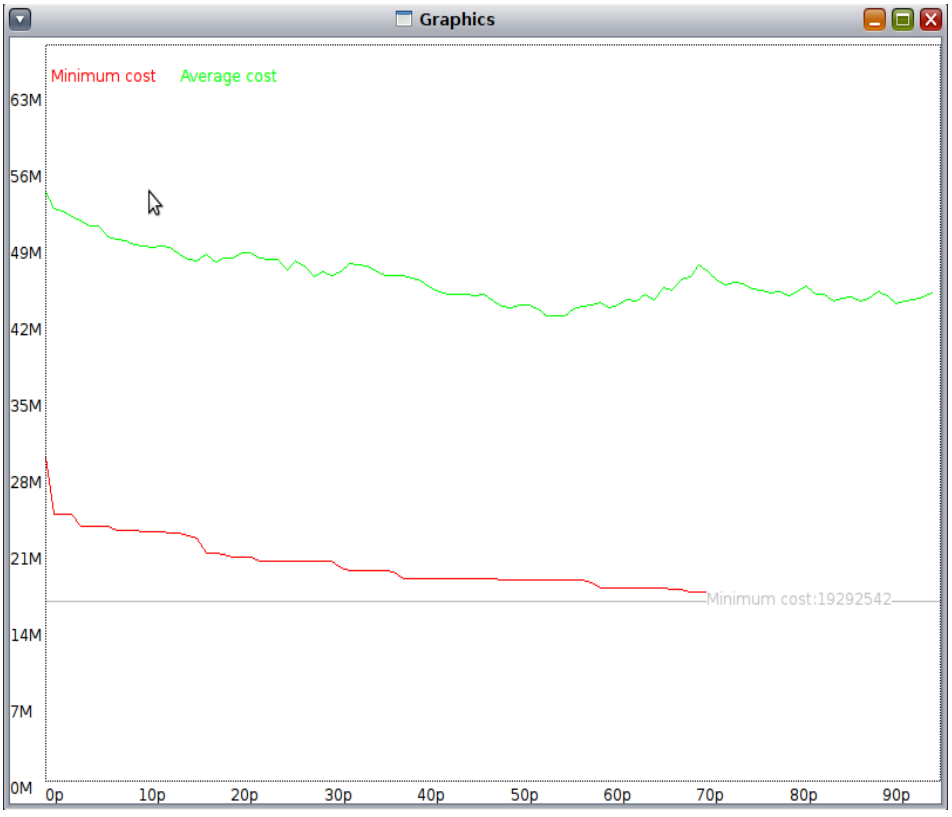
Boltzman with temperature =-1000

```
The best individual is
Genes 14 | 15 | 7 | 5 | 13 | 17 | 3 | 2 | 6 | 18 | 9 | 12 | 8 | 4 | 10 | 11 | 1
| 0 | 16 |
Cost: 19450228
Fitness: 5.141327906284698E-8
The parameters used were :
Selection policy: boltzmann
Population size : 1000
Generations : 100
Mutation probability : 0.01
The fitness function type is:  1/g(x)
Temperature   -1000.0
Elitism percentage = 7.0%
```

Setting Boltzmann with a low temperature, the best chromosomes have a bigger probability of being selected, therefore the average cost decreases quickly and converges.

Boltzmann with temperature=1000

```
The best individual is
Genes 10 | 11 | 4 | 5 | 13 | 18 | 3 | 12 | 7 | 17 | 2 | 16 | 8 | 6 | 14 | 15 |
9 | 0 | 1 |
Cost: 19292542
Fitness: 5.183350125659957E-8
The parameters used were :
Selection policy: boltzmann
Population size : 1000
Generations : 100
Mutation probability : 0.01
The fitness function type is:  1/g(x)
Temperature  = 1000.0
Elitism percentage = 7.0%
```

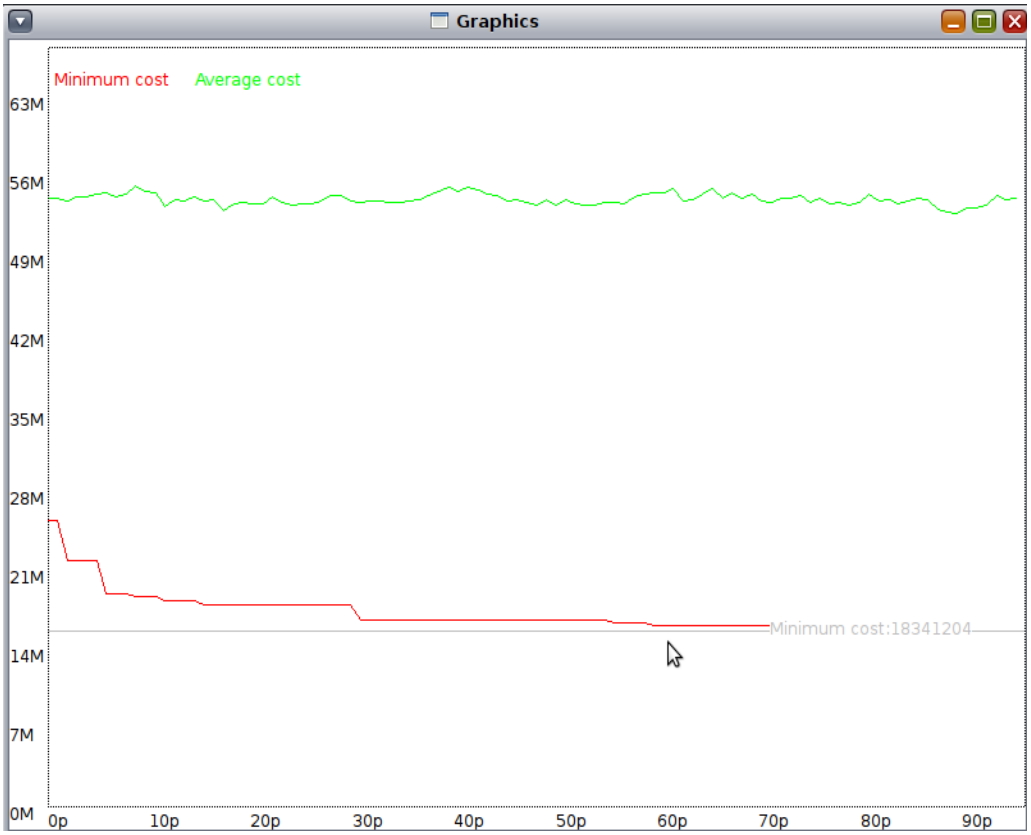


Setting Boltzmann with a high temperature,all the chromosomes have a good probability of being selected, therefore the average and the minimum cost decrease smoothly if we compare with the previous one.

Low mutation percentage vs High mutation percentage

No mutation (0%)

```
The best individual is
Genes 13 | 3 | 8 | 11 | 10 | 18 | 7 | 12 | 9 | 17 | 4 | 16 | 5 | 6 | 14 | 15 | 0 |
1 | 2 |
Cost: 18341204
Fitness: 5.4522047734707056E-8
The parameters used were :
Selection policy: roulette
Population size : 1000
Generations : 100
Mutation probability : 0.45
The fitness function type is:  1/g(x)
Elitism percentage = 1.0%
```



Setting a low percentage of mutation, the minimum cost function converges quicker, but in general terms it is really difficult to get excellent values because tends to be very poor.

Big mutation (45%)

```
The best individual is
Genes 18 | 6 | 13 | 17 | 10 | 7 | 12 | 0 | 3 | 8 | 4 | 9 | 5 | 11 | 15 | 14 | 16 |
2 | 1 |
Cost: 22274702
Fitness: 4.489397882853831E-8
The parameters used were :
Selection policy: roulette
Population size : 1000
Generations : 100
Mutation probability : 0.45
The fitness function type is:  1/g(x)
Elitism percentage = 0.0%
```

Setting a very high percentage of mutation, the minimum cost function does not converge at all, and it is also really difficult to get excellent values. In general, we experimented that a good percentage is around 15%.

Low elitism percentage vs High elitism percentage

High elitism percentage (15%)

```
The best individual is
Genes 14 | 15 | 5 | 6 | 18 | 11 | 9 | 16 | 3 | 10 | 4 | 12 | 17 | 7 | 8 | 13 | 0 |
1 | 2 |
Cost: 18263048
Fitness: 5.475537270668072E-8
The parameters used were :
Selection policy: boltzmann
Population size : 1000
Generations : 100
Mutation probability : 0.25
The fitness function type is:  1/g(x)
Temperature  = 200.0
Elitism percentage = 15.000001%
```

High elitism percentages can make both functions to converge quickly, but in general we experimented that it is important to have high elitism values to get good results (up to 25-30%).

Low elitism percentage (0%)

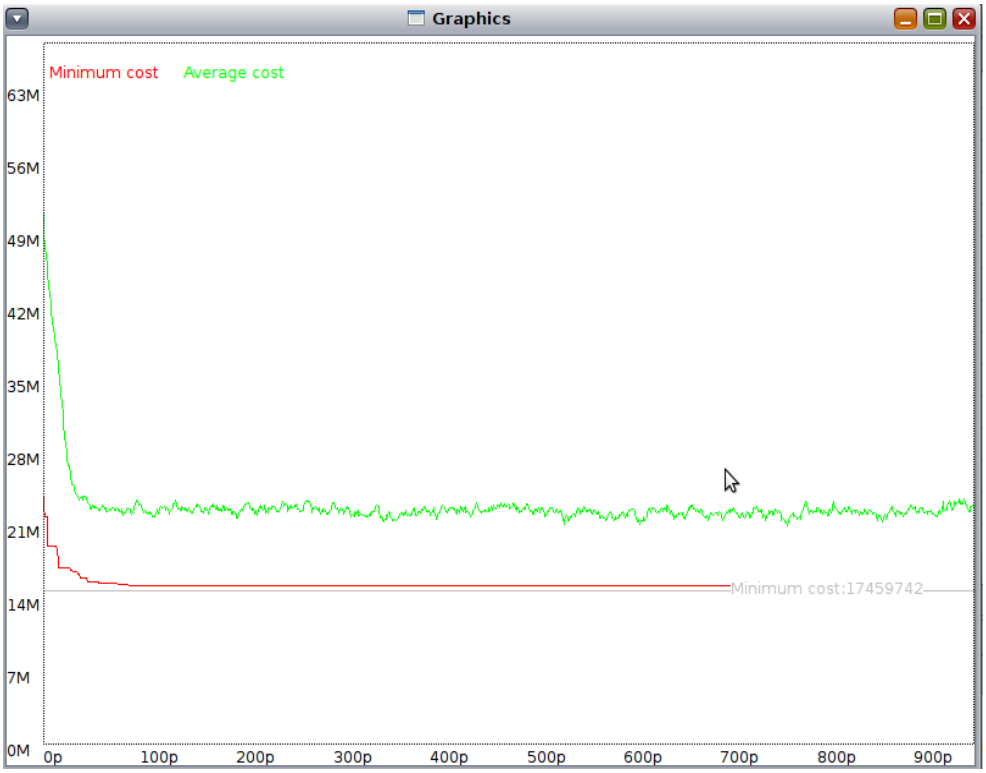
```
The best individual is
Genes 9 | 10 | 5 | 17 | 18 | 11 | 13 | 2 | 8 | 7 | 12 | 3 | 6 | 4 | 14 | 15 | 0 | 1
| 16 |
Cost: 22414182
Fitness: 4.4614610517573203E-8
The parameters used were :
Selection policy: boltzmann
Population size : 1000
Generations : 100
Mutation probability : 0.25
The fitness function type is:  1/g(x)
Temperature  = 200.0
Elitism percentage = 0.0%
```

With a lack of elitism (or very low percentages) the populations are really poor, and in general terms we get bad results.

Parameters for a "good case"

```
The best individual is
Genes 8 | 6 | 7 | 17 | 13 | 18 | 9 | 16 | 3 | 10 | 4 | 0 | 11 | 5 | 15 | 14 | 12 |
1 | 2 |
Cost: 17459742
Fitness: 5.727461493990003E-8
The parameters used were :
Selection policy: boltzmann
```

```
Population size : 2000
Generations : 1000
Mutation probability : 0.15
The fitness function type is:  1/g(x)
Temperature  = 200.0
Elitism percentage = 25.0%
```



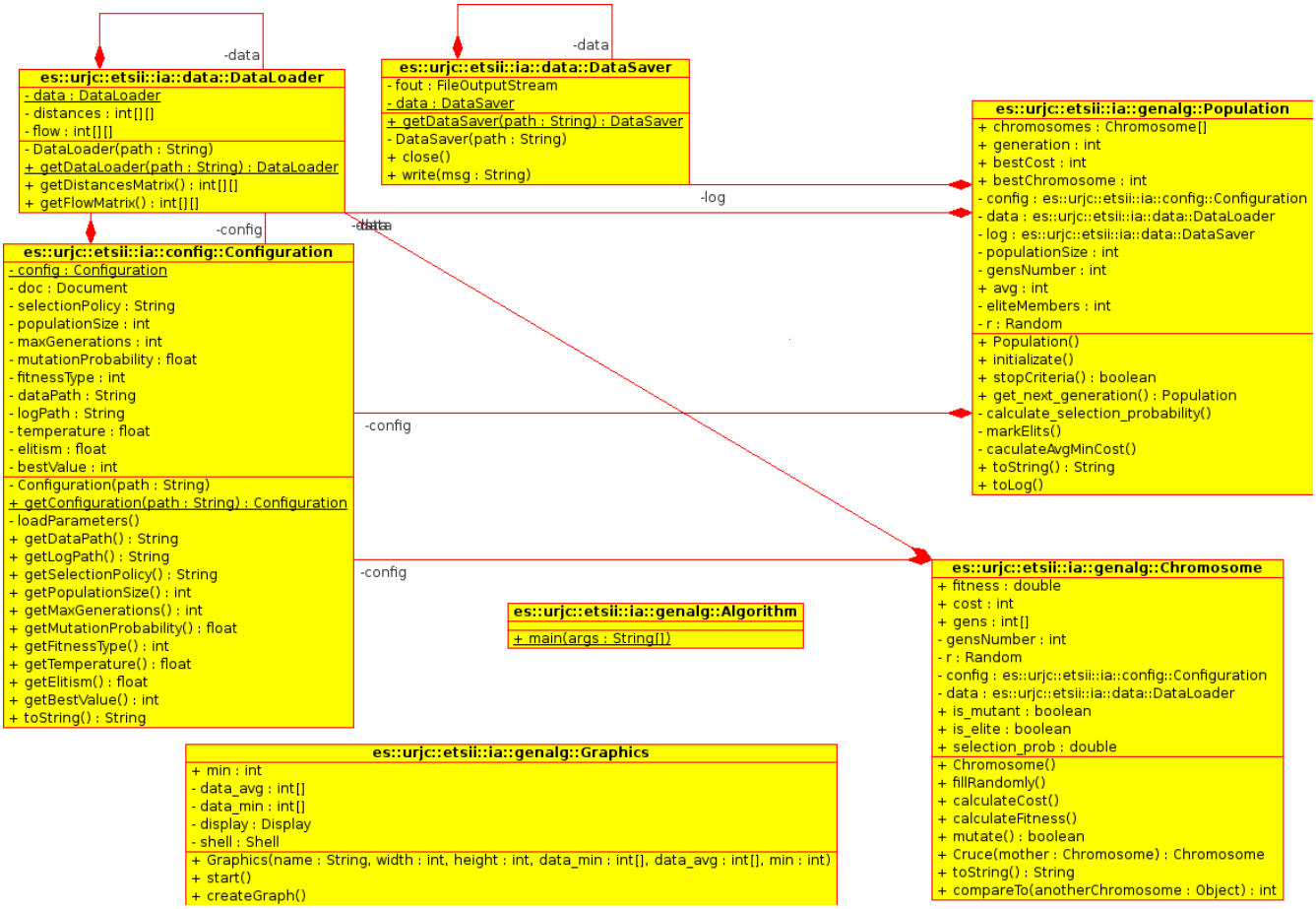
In order to get excellent values, we set a high elitism percentage, a medium mutation percentage, and in general terms we need much more generations. In this example the best value is obtained after more than 700 hundred generations.

Implementation

The code is organized in three different packages:

- es.urjc.etsii.ia.config: Contains the classes responsible to implement all the functionality related to the configuration file.
  - Configuration.java: Singleton class which takes care of reading the configuration file, and offers services to access to the values.
  - Test.java: Test class to test Configuration.java.
- es.urjc.etsii.ia.data: Contains the classes responsible to implement all the functionality related to the process of reading and writing external data.
  - DataLoader.java: Singleton class which takes care of reading the data about the sports and cities, and offers services to access to the values.
  - DataSaver.java: Singleton class which takes care of writing into the log file.
  - Test.java: Test class to test DataLoader.java.
- es.urjc.etsii.ia.genalg: Contains the classes responsible to implement all the functionality related to the genetic algorithm.
  - Algorithm.java: It is the main class, and represents the entry point to the application.
  - Graphics.java: It takes care of the functionality related to draw the chart with the results from the application execution.
  - Population.java: Represents a generation of chromosomes, and implements all the operations related to it: calulate_selection_probability(), calculateAvgMinCost(), get_next_generation(), ...
  - Chromosome.java: Represents a set of gens, and implements all the operations operations related to it: mutate(), cruce(), calculateCost(), calculateFitness(), ...

A class diagram with the main classes is shown below:



User Manual

There is two different distributions of program for Windows OS and for Linux type OS.
The first step in installation process is to copy contents of the folder correspondenting to your OS to your hard disk in some folder, let's call it “<path_in_your_system>”.
Then to follow instructions of “!Readme.txt” which is situated inside the copied folder.

For Windows:
Run <path_in_your_system>\olympics.exe by double-clicking on it
Alternatively you can run <path_in_your_system>\olympics.jar also by double-clicking on it

For Linux:
Execute following script in linux shell

cd <path_in_your_system>
java -jar ./olympics.jar

Program is provided with parameters xml file input and combined output. That means that all program's configuration is situated in <path_in_your_system>/config/params.xml, and output is split into 3 flows:
-console output, that shows progress of task execution and the best obtained result
-log file output, goes into folder <path_in_your_system>/log/ and can be used for further analysis of population evolution during execution of algorithm
-graphical window to show visual information

You need just to set parameters in configuration file and execute the program, how it was described before.
Important:

- You should have proper installed JRE 6 on your machine
- You should make log directory available for writing

Conclusions

Application of genetic algorithms can be very useful due to the flexibility they offer. It would not be difficult to adapt this code to a problem of a very different nature. The most difficult part consists of getting really good values for the problem, because of its “random nature”. Sometimes it is really difficult to set the parameters, and it is a “trial-and-error” experience.

But, in general terms, we are satisfied with the result, and consider that it is a very interesting and effective way of solving AI problems.