Technical Report

Project Speech Synthesis Using a GA V1.0

By Pierre-Yves Hervo

Polytech Nantes High school of Engineering

18/09/2013

Introduction

This report contains the technical explanations of the work I have realised for the ICCMR laboratory during my four month internship on the subject of speech synthesis using a Genetic Algorithm.

This report won't explain the background of the project, it define the architecture I have designed and its evolution during the project. If you want to learn about the background, please read the report I have written about it.

I will speak a little about my Java code but I won't explain the implementation in detail. If you want to look at it, the sources are fully documented with Javadoc and my own comments.

Both the Java code and the reports are available at the following address: https://github.com/phervo/ProjetEte2013.

There is two important points to understand if you want to reproduce or improve the project. The first one is how I designed the communication architecture and the second is how my GA works.

I will start this report with these two points, then I will tell a few words about the code, speak about the evolutions I bring during the project and I will conclude.

Contents

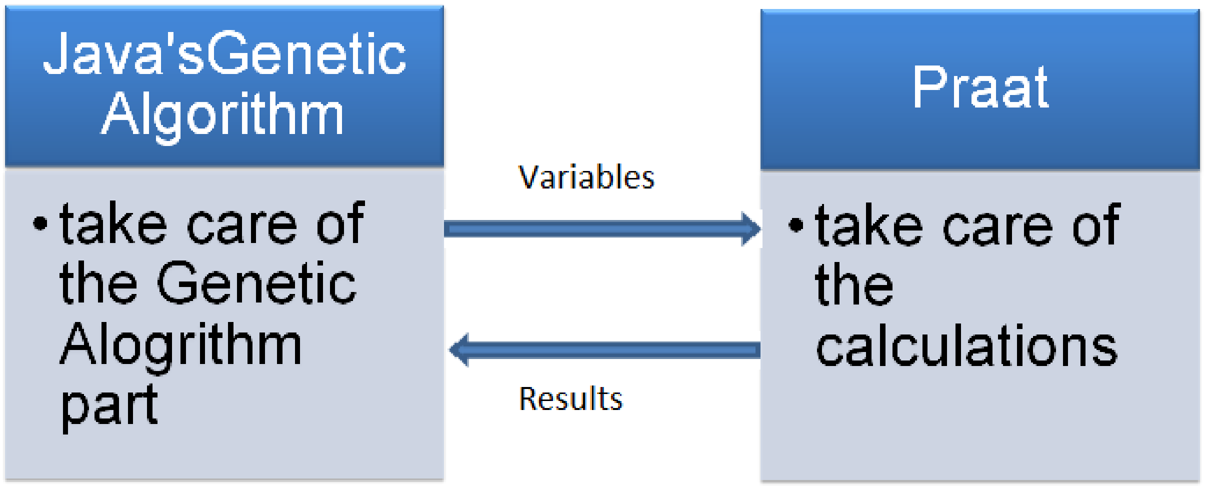
The architecture

This part deals with the architecture I had to develop to make Praat and Java communicated. In this part we won't question when to use it, it will be explain in the second part, here I will just explain how it works.

Quick recap

The goal of the project is to use a genetic algorithm to found the Praat's variables values for a specific vowel. For this, I use the formants of the sounds to know if the sound produced by the current individual of the run is close or not to my target. Praat allows me to generate the sound and to get the formants values, so I use this software to delegate it the calculations.

Here is the general idea of the architecture:


Java'sGenetic Algorithm
• take care of the Genetic Alogrithm part
Variables
Results
Praat
• take care of the calculations

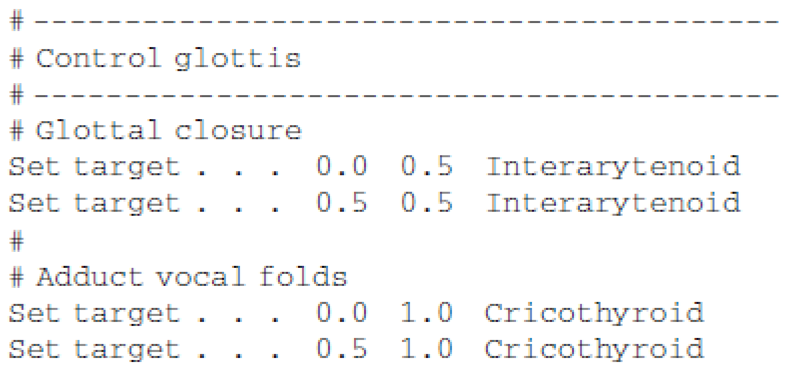
But in practise, the synchronisation of these two software is not so simple.

Giving the Praat's API, we need to use two different ways to connect Java and Praat. We need to consider one way to communicate between Java and Praat and another way to communicate from Praat to Java. The first one is use to send and execute a Praat script and the second to send the answer from Praat to Java. They both use different techniques to be executed and so a particular treatment must be done for each.

2.1 From Java to Praat

Praat allows writing scripts in its own syntax. I use this way to synthesis the sound and calculate the formant.

Here is a short example of a script for speech synthesis:



```
# ----------------------------------------
# Control glottis
# ----------------------------------------
# Glottal closure
Set target . . .  0.0  0.5  Interarytenoid
Set target . . .  0.5  0.5  Interarytenoid
#
# Adduct vocal folds
Set target . . .  0.0  1.0  Cricothyroid
Set target . . .  0.5  1.0  Cricothyroid
```

I generated these scripts dynamically in Java with the good values. Once it is done, I need to send them to Praat in order they get executed.

The way I choose is the software SendPraat. It is a program developed by the same authors as Praat and available at this address: http://www.fon.hum.uva.nl/praat/sendpraat.html. It allows sending orders to a running instance of Praat.

It means we need two programs :

1. A normal Praat software already launched.

2. SendPraat which will give it orders. No need to launch this one, it only works in command line.

If you give SendPraat the name of a script, it will made Praat launch and execute it. The only thing remaining is to make Java executed SendPraat. For this, I used the Java Runtime Environment which can use the command line of windows.

Note: I use a SendPraat.exe as I am a windows user but you can compile the source code yourself to use it in your own operating system. If you want to make Praat communicate with a C program, you can use the SendPraat directive, no need to compile source code. For more information, look at the Praat's API, section Praat scripting. As I was working in Java, the solution I presented is currently the best.

2.2 From Praat to Java

There is only one way to make Praat communicate with another program, whenever the language is: the sockets.

Sockets are tools used in computer science to make two different program communicated. For this, they will use the network principles and send network packets to a computer on a specified port.

It is not necessary that the target is running on another computer, it can be the same and in that case, we use a local network call localhost. The first program will send a message to the other specifying the port and the second one will listen will listen to the port and get the message when it arrived.

Praat allows to send sockets by the directive "sendsocket" but it can’t receive sockets from another program. That is why we got to use the SendPraat program in the other side. If Praat send a socket then our GA will need a functionality which always listen to this port and will take the message. Such functionality is basically called a Server. In that purpose, I implemented a Java server that listens to a specific port to get the message from Praat. It get the message as a string and transmit it to the GA which will cast it in the objects needed.

2.3 Sequencing

There is a problem of sequencing to take care to synchronise Java and Praat. The problem came from the fact that they are two different thread(program) running. They both have a different execution's speed. The Java's GA work very fast, each generation take a few seconds while each sound synthesis take a few seconds in Praat. For example, it takes approximately 12 second to Praat to generate a 2.0 seconds sound. So there is a problem of speed and synchronisation.

This is why I should have establish a sequencing order between the two programs to force the Ga to wait for Praat's answer before going to the next individual. More precisely, to wait for the server to get the answer from Praat and store it into the GA. The GA could do the comparison of formants while it is done.

The only solution was to use semaphores. It is a computing technique for sequencing tasks. It work on the principle of token. You have a token in a box, if someone want to do an action he took the token and it released it when finished. The others wait for the token to be free before doing their action.

I used in fact two levels of semaphore in the fitness function:

The first on is when the GA start the fitness function, it took the token and it released it when it had finished the comparison and calculated a matching mark. It prevent the Ga to launch the fitness function with another candidate while still running the previous one. For example if Praat is still running a synthesis, it won’t be able to do the comparison with an empty values and switch to the next candidate.

The second level is in the function fitness itself, it allows to be sure that the server had store the message into the GA. As the server is a different thread, it is a obligation. It forbids the fitness function to do the comparison with the reference formant until a value was set by the server. It avoid errors.

Here is the representation in the form of an algorithm. The operations concerning the semaphores are written with letters whereas the other actions are written with numbers

```
Fitness function{
    A) launch of the function, took the token "function" to avoid the GA to launch
        1) create a script with the candidates values
        2) send it to praat
        3) praat execute the script and calculate the formant values
        4) the server get the socket and store the result
        => release the "praat result stored" token
    B) If the token "praat result stored" is available we can continue, else we
        5) compare the formants values
        6) give a matching result
    C) Release the token "function".
}
```

The Praat Object

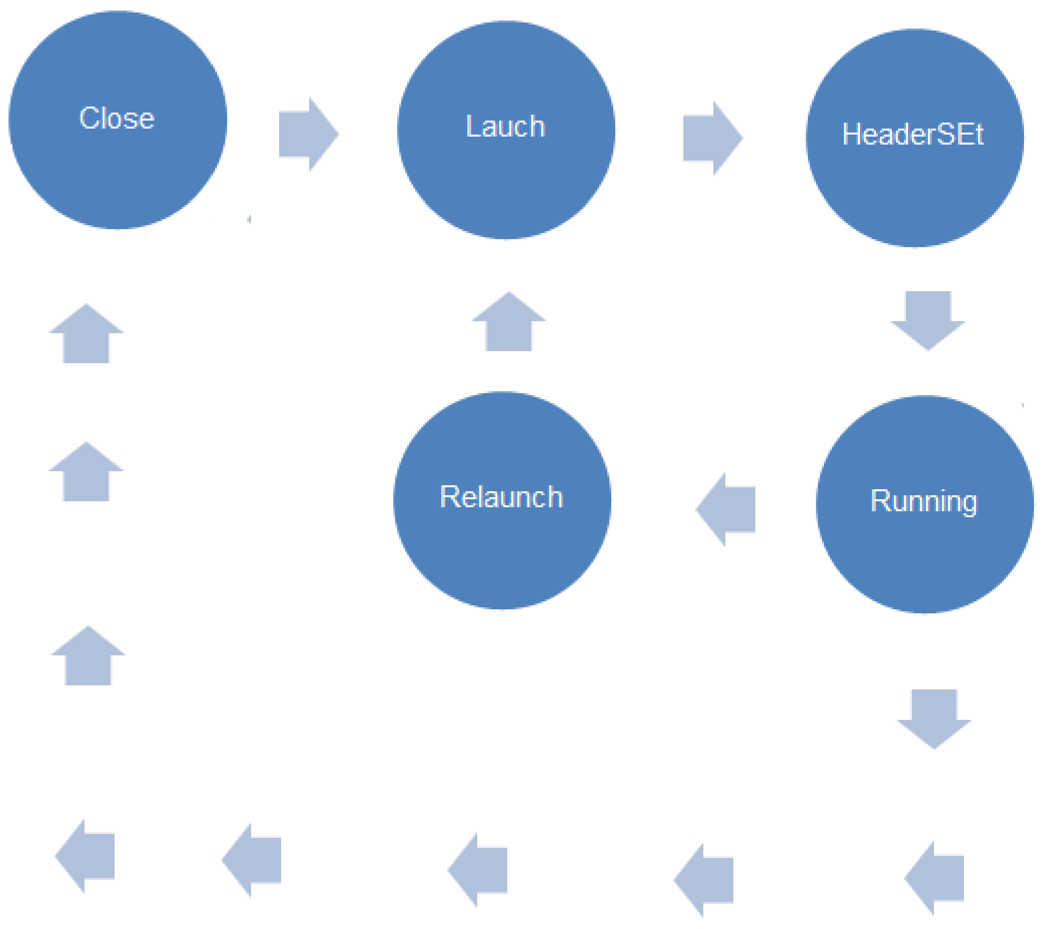
I presented in the previous point how to synchronise the two software with semaphore for the global run but there is two particulars times that need a specific attention and specific controls. It is due to Praat who get two specific times where you need to do a specific action.

The first one is during its launch. Praat is a stand-alone program, you can't have the control on its execution. You can launch it in a new process. This way you initiate the process but you can't know when it launches and ready to get a script and executed it. The main programme will launch Praat in a new process and continue its execution sequentially. So you will try to send commands to Praat but nothing says that the programme have finished to load in the new process. It is a question of milliseconds but you had to take care of this.

The second one is that it keeps the result in an internal memory and you can't delete them. The remove command in the Praat GUI only clean the list but don't remove the object from your computer memory so you can quickly reach the computer memory limit and Praat will stop generating sound. The only way to clean its memory is to quit Praat and launch it again.

To avoid those two problems, I had to create a state diagram to know in with state Praat is in order to be able to know if I can use it and re-launch.

Here is the diagram:



Now the explanation:

We start with the Close state. Then we passed to the state launch. At this moment we have launched Praat in a new process but we can't be sure it is finished to launch. So if we send a script via sendpraat, it won't do anything. The solution I designed is to put a while loop and send message regularly to Praat. Once it will be launch, it will receive the script, execute it and return a statement to say that it is launch. Once it is done, it passed to the HeaderSet state. In this state, it will launch a special script to create the two objects that will allow to synthesis the sounds. We only need to declare them once and is necessary to declare them before trying to synthesis the sound, that's why I have create this special state.

Then you go to the running state with the normal execution of the programme.

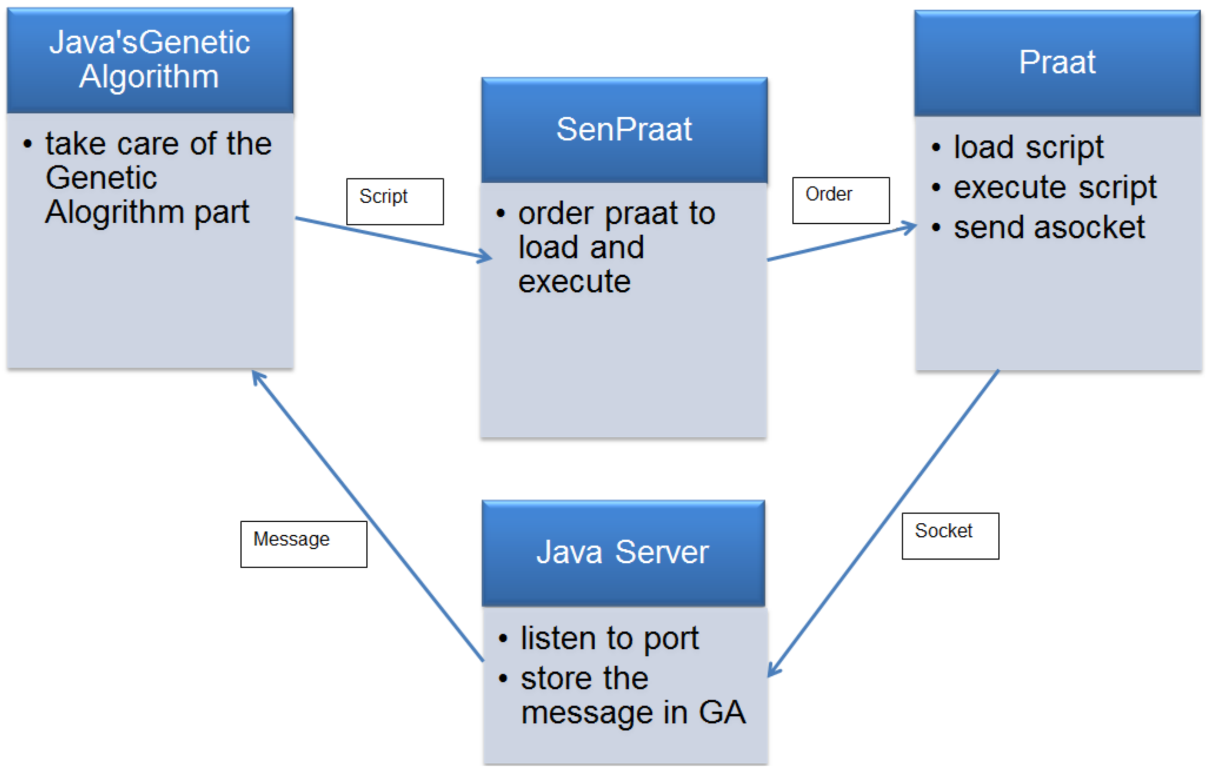
From time to time, Praat consume all the computer memory by keeping its results. When it reach this point, It stop generating sound and stopped. So from time to time I had to close Praat the only solution to clear the memory and relaunch it. I determine with some tests that it corresponds to each 40 generation with my GA. That why I create a special state that will close Praat and then go back to the state and run the state path again. It finished when we decided to close.

Recap

So in conclusion we had to consider two different sides for the communications:

- The message from Java to Praat. On this side, we had to use a specific Praat's like program call SendPraat.
- And a message and the message from praat to Java where we had to use a Java's Server to listen to Praat's sockets.

You have a recap of the architecture in the figure below:



The Genetic algorithm

Now that we defined how the two software interact, we can define how the GA works and when it needs to contact Praat.

I will assume in that part that the reader is aware of how a generic Genetic algorithm works.

I will in this part explain how I designed each of the different part to get a specific GA that is available to deal with the speech synthesis under Praat.

The population

To synthesise a sound, I use Praat variables values. There are 29 of them but we can only defined 19 for a vowel, the 10 others will take values by default. It is this combination of variable that would produce a particular sound, so it is the thing we should make evolve. I design a structure as a list of doubles. The length of this structure is variable, you can define it. It allows using sequence of length 8 or 16, I will came back to that point in the III.

I use a population of 10 individuals. As a Praat's calculation is very long, we can't afford to get more individuals. Anyway, it is sufficient to get results.

I generated the initial population at random by picking each variable in a specific dictionary. Each of the values if the sequence corresponds to a Praat variable and each of these variables get an interval of possible values. So I defined an alphabet of possible values for each. I picked each variable in this alphabets to be sure that the variables don't go out of range.

The fitness function

In my implementation, the lower the value of the fitness score is, the better a sound is.

To know if a sound is close or not to the target, I used the difference between the formant sound and the formant target. So the sound with a formant value closer that another one will get a better score. Then I make the sum of the three results. This is what I call the "basis".

Now let's speak about the formants. It is said that a formant value could vary to +/-10% of its value. So I have a limit around the formant and I consider that if the formant sound is in this interval then the formant is "found".

Now the problem with the basis is that it don't evaluate a sound with the number of formants founds. Let's take two different sounds. One has one formants found, F1, with low difference and the difference for the second and third one is low. The second one has two formants found, each just in the limit (high values) and very far from the third. Then the first sound could get a better result because the three formant differences are low, so the sum will be low whereas the second will get a high score. So this basis is not representative of the quality of the sound depending of the number of formants.

So I decided to put a penalty on the sound regarding there number of formants.

For each formant not found I add the difference to the formant to the basis. Let’s take an example. If I have a formant with F1 found and F2, F3 not found, I calculate the basis and I add the difference for F2 and F3. If I have a formant with F1, F2 found and F3 not found, I add the difference to f3 to the basis. So in the worst case, no formant found, I double the score. The objective of this penalty is to penalise the sound regarding there number of formant. So a sound with more formant will get a lower result than those with less formant.

The script

In order to generate the sound and calculate the formant, I use a Praat script. The values of the Praat parameters are automatically fill in by the Java program. Some are statics like the lungs that needs to get specific values or the sound produce will be a white one. To calculate the formant, a Praat function exists, the function “to formant”. It creates a new object with all the formants information. This object allow to do some calculations like the mean, get a value at a time, … I return the mean value of a formant to the GA as the value of the formant. The problem is that some time, you can see on the speckle that there is no values on the speckle plot but the query function will give you a value which is too high or too low. I had to add some verification to be sure that the result I return is correct.

First, there had to be the good number of formants, if it is too low, then it directly goes to the trash and I return 0 0 0. Then there is to be a value all along the sound duration. I query the values for 0.1, 0.25, 0.50 and 0.99 seconds to very it. If there is no value then the sound is not complete and it goes to the trash. The last verification is the value for the mean. I have look for the general interval of value for each formant. If the mean value for a formant is out of this interval, then there is a problem and probably something missing so I return 0 0 0.

The selection operator

I create my own selection operator. It works in accordance with my fitness function. I picked values two by two. The goal is to force a cross over between specific individuals.

I designed it as a selection by level. At lvl0 (no formants found) and lvl1 (1 formant found) I cross individuals between with another of this level to get a better one (more formants found). At lvl2, I try to cross an individual with two formants with an individual with the missing formant.

The general purpose is to get as fast as possible to a sound with 3 formants.

To avoid writing a very dense thing, I skip some parts of the verifications. You might suppose that each time I checked that I didn’t grab twice the same individual. It would be useless to cross an individual with itself, you would get the exact same result. You can manage it with a loop, each time it grab randomly another individual in the list of this level while it is not different than the first selected.

Here is the algorithm:

1) I create lists containing the sounds for a specific formant found value.
   I create a list containing all the sound with no formants founds called none.
   I create a list containing all the sound with the formant F1 only called F1.
   I create a list containing all the sound with the formant F2 only called F2.
   I create a list containing all the sound with the formant F3 only called F3.
   I create a list containing all the sound with F1F2 called F1F2.
   I create a list containing all the sound with F2F3 called F2F3.
   I create a list containing all the sound with F1F3 called F1F3.

2) I create three lists, one for each level of formants

   I create a list containing all the FX called lvl1.

   I create a list containing all the FXFY called lvl2.

3) If level1 is empty and lvl2 is empty then
   a. I select at random between the four best individuals of the list none. I don't use the others; I consider they are not good enough.

4) Else if lvl2 is empty then
   a. Lvl1 is not empty so I picked an individual of lvl1 at random. Now, we need to see if there is only one individual in the list or more.
   b. If size of lvl1 =1 then
      i. There is only one type of formant FX, I need to see if there is more than a single individual in it, I can't allow the individual to cross with itself.
      ii. If size of the FX in lvl1=1 then there is only one individual that I have already picked above. So I picked another one in the 4 bests candidates of none(lvl0).
      iii. Else I picked another at random in the list FX.
   c. Else there is more than one type of FX, so there is at least another individual of the same level, I picked it at random in the list lvl1.
5) Else
   a. Then lvl2 is not empty, I picked a first individual at random
   b. If lvl1=0 et lvl0=0
      i. Then I picked another individuals at random in lvl2
   c. Else
      i. If the list with the remaining formant exist, I pick an individual in it.
      ii. Else I picked another individual at random in lvl2
      iii. Else I picked an individual at random in lvl1
      iv. Else I picked the best lvl0, not below

The evolutions operators

I used two classical evolutions operators: The cross over two points and the mutation with a probability of 0.05.

The cross over two points allow to modify a portion of the sequence between two points selected at random. As we don't know which portion corresponds to which formant, it is better than modify all the sequence after one point (single point cross over). We can hope that it will change between the rights, small portion.

The mutation is an operator that modifies randomly a value of the sequence. The little difference between my mutation and the classical one is that when a mutation occurs, I picked a new value at random in the corresponding alphabet. It avoids having a variable that goes out of range.

The termination

I have two termination conditions. The first one is when I reach a score lower than the sum of the three margin I authorise **AND** the three formants are found.

The second one is a timer condition, I stop the algorithm after 3 hours. I considered that after this time the solution is good enough or that it wouldn't converge.

The Evolution of the project

I spend most of my time to synchronise java and Praat, it was a really difficult task which asked a lot of reflexions, tests and debugs to be able to get a stable and final version. Once I have finished I try some things to get a good Genetic algorithm which is perfectly available for the project.

Here is the list of this evolution:

The first thing I have done concerns the alphabets. I was running the GA with the same interval of value from 0 to 1. Then I learn about the values for each variable and try it. The result were better as soon as the change made, the syntheses goes faster.

I started by a fitness function of 4 values corresponding to the number of formants founds. I realised soon that it wasn't working great. The main reason was the lack of diversity of fitness values. If each individual with the same formants gets the same value then it was the same that picking one at random. I didn't grab good individuals.

Then I changed my fitness function to get the version with only the basis. It allows me to discover the errors in the Praat calculations and force me to adapt my script to get better results from the Praat calculation.

Once it was done, I realised that some result with less formants get a better score than those with more formants. I thought to the system of penalty to favour the sound proportionally to their number of formants. It give betters results. I kept this fitness function since.

As it return good results, I decided to kept the best individual as an elite, it allow me not to lost the best one at each generation. Another improvement was to build a cache for the results. I decided to kept the result from the previous generation to avoid to recalculate them I found them again. It is really useful, I save 12seconds of calculation each for each individual that was in the previous generation. It is really common at the end of the GA run.

The last thing I have done is my new selection operator. I implemented it the last week of my internship so I haven't had the time to run a lot of test but the results I get were better than the previous one using one of the default GA selection operators.

Now I think the idea of the level selection is correct but I haven't had the time to test some possibilities. I don't know if it is better to cross a second level individual with a 1st level formant or another second level formant. For example is it better to cross F1f2 with F3 or F1F3? So some test needs to be done and maybe my selection operator can be improved a little, but I think it goes in the good direction.

Java sources

As I said in the introduction, I won't speak in detail about the implementation of the solution in java. I will explain the general architecture of the project in order that you understand it and could be able to found quickly the things you are looking for.

First I used the Java Genetic Algorithm's API: watchmaker Version: **0.7.1**, available here: http://watchmaker.uncommons.org/. This API allows me to design and manipulate my own structures for the candidate, the fitness function and the evolution operators by implementing interfaces. I needed to rewrite some classes to modify the execution as I wanted. As it is an open source project, it was very easy.

I divided the project in different Package with a specific purpose each:

The **application package** contains the main class to launch the program.

The **communication package** contains the classes for the java Server, tools for the communication with it and the way to deal with the message from Praat.

The **controller package** is used by my GUI application according to the MVC design pattern model. It defined all the actions of the GUI components.

The **elements package** contains all the basics elements I manipulated during the run of the Genetic algorithm. You will find there the definitions for a Sequence or a Formant for example.

The **exceptions package** contains the exceptions I raised if necessary for some elements.

The **genetic algorithm package** contains all the classes I had to implement or re-implements to use the watchmaker API, It also contains the GeneticAlgorithmCall class which is the main class of this project. It is the one that use all the others and set the elements of the watchmaker API as I need. It is a very big class but I wanted to keep all the GA's things in one place.

The **message package** contains the definition of all the messages I send to Praat or the way to treat the message that comes from Praat after the server get them.

The **monitoring package** contains all the tools I used to create the CSV file and the results curves at the end of the GA's run.

The **Praat Gestion package** contains all the classes needed for the use of the Praat object. As I explain in a point above, it is an object that simulates the Praat software state using the design pattern "State".

The **test package** contains both the unitary and the integration tests.

The **vue package** contains the definitions of the graphic frames used in the MVC deign pattern for the GUI.

Conclusion

In conclusion, it was a very interesting project and I think I have done all I can for a period of 4 months. A possible improvement would be to recode Praat, which is an open source project to allow the parallelization and reduce the time of run.