1 Problems

1.1 JGAP Framework

Our first decision to program in the JGAP Framework was a really bad one. Besides the days of effort we put into setting up JGAP we also lost a lot of time attempting to get JGAP to work. Of course JGAP did something but not at all what we wanted it to do or what we expected.

JGAP uses a *Configuration* object in which it stores, among others the *Fitness Function* object and the *Population*. But also settings and other configurations.. This is immidiately confusing as we do not have an idea what these settings are or what how the population is build up. We also had to implement functions we had no need for and had to use with variables (arguments) we did not want. This was easily handled as we did not use what we did not need but when the results started to role in we were baffled.

The results were seriously dissapointing. Our population hardly evolved into something better and when it did it devolved in a matter of seconds. While the first might not be such a big problem, the second most certainly always is as devolving is the opposite of what we want.

We first thought this was because our fitness function but after we updated this function to something that we had full confidence in results were still horrible.

We then thought this was because of the innacuracy of our fitness function , perhaps we missed the golden mean between time saving and accuracy? It turned out we did not. A lot of simulations and testing proved that even with more thorough fitness functions and more accurate fitness values de-evolving continued.

Our next hypothesis was that strange settings within the JGAP Framework might be the cause for the unexpected reults. After we had did some serious internet research and code inspectations we had to conclude that we had no idea what these settings were. Luckily there were thousands of options we could try but none of these options were adequately documented. For example, an argument suggested it would change something by it's name. But we could not know what format this argument should be, why it would suddenly print errors and more extreme, we could not evaluate the results as results needed hours of processing time for, most likely minimal changes.

An example of a weird and unwanted (by our team) function is the *Natural Selector* object and how they are implemented by JGAP might demonstrate our troubles with this framework. At one point in time we tried to simply safe those chromosomes that work the best. Instead of simply setting this we needed to create an object and add it to the *Configuration* object. One would think this is not a problem but JGAP stores these *Natural Selectors* in an ordered chain of registered selectors. The *Configuration* object can also be locked and not be unlocked once it is unlocked. And lastly, the *Configuration* object sometimes resets data to *null*. Once we took care of all these problems we could start testing, but this took a lot time and results were inconclusive.

A short summary of our problems with JGAP.

Algorithm 1 Main program

```
Initialize Population
while true do
    Evolve Bottom 50%
    Evaluate Bottom 50%
    Sort Population 50%
    Replace Bottom 50% With Top 50%
end while
```

1.2 Leaving JGAP Framework Behind

With more and more experience with genetic programming, it became more and more clear that what we needed did not need to be extremely complicated. In the beginning we were already uncertain if JGAP was the best alternative but after these experiences it became clear that it was not. In one afternoon a simple framework was written that did exactly what we wanted. Another afternoon was used to rewrite the code into something more structured using an OOP approach. Now, with extremely well-ordered and structured code we could start running more definitive simulations, debug and expand the potential capabilities of our robots even though we were already behind schedule a complete week.

Because we were not sure if it was just JGAP or JGAPs settings that were responsible for the previously achieved horrible results we made sure the top performing chromosomes always survived. This resulted in the following algorithm:

1. Initialize the population.
2. Evolve the bottom 50%.
3. Evaluate the bottom 50%.
4. Sort the population.
5. Copy the top 50% to the bottom 50%.
6. Restart x

After the first simulations it became clear that not everything was going as well as expected. The robots still devolved extremely even though this was thought now to be impossible. The textual output of the program now indicated what was happening but not in an easy fashion. We needed a graphical repreentation of the fitness values throughout the evolution so we can easily in, one moment, see when in time the best fitness value was evolved and what the circumstances were. This helped us a lot in later stages of our project, both in debugging and performance evaluations.

We plot the following three values: the biggest fitness value in red, the average fitness value of the total population in blue and the average fitness value of the newly mutated chromosomes in a population in green. The first results explain what is going on.