UNIVERSITEIT VAN AMSTERDAM

# Tweedejaarsproject A.E.S.I.

## ARTIFICIAL EVOLUTION AND SWARM INTELLIGENCE

VERSLAG

Jeroen ROOIJMANS
5887410

Maarten INJA
5872464

Maarten DE WAARD
5894883

June 25, 2010

# Contents

**1 Introduction** **2**

**2 Project goals** **2**

**3 Method** **2**

3.1 Genotype . . . 2

3.2 Phenotype . . . 3

3.3 Meta language . . . 3

3.4 Fitness function . . . 5

3.5 Evolve . . . 5

**4 Results** **5**

**5 Problems** **5**

5.1 JGAP Framework . . . 5

5.2 Leaving JGAP Framework Behind . . . 6

5.3 Creating our own Framework, a Good Decision? . . . 8

**6 Background information** **9**

6.1 RoboCode Game . . . 9

6.2 Robot Anatomy . . . 9

6.3 Robocode rules . . . 9

6.4 Programming the Robot . . . 10

**7 Planning** **11**

7.1 Documentation . . . 11

7.2 Initiation . . . 11

7.3 Genetic programming, single bots . . . 11

7.4 Test phase . . . 11

7.5 Genetic programming, teams . . . 11

**A Planning** **13**

## 1 Introduction

During this project we will attempt to create a program that artificially evolves tank like agents in the RoboCode environment[1]. RoboCode is a simple game, where virtual tanks can fight eachother. More information and rules of the game can be found in section 6.

It would be fairly easy to write a powerful bot ourselves, but using genetic programming[2] we can let a computer evolve these bots. To test our evolved bots, we use the RoboCode environment to generate test data of battles between our evolved bots and various enemy bots. After a certain amount of battles, RoboCode generates results, these results consist of a overall score. The score is calculated using the survival rate, damage done and ranking of the previously fought battles.

There is a similar program that evolves RoboCode bots created by Klaus Meffert, the program is called RoboCode with JGAP[3]. After short research we found out that this program is implemented different from ours. It does not evolve on the code, like we want to, but it enables code segments, making evolving to a good bot a lot easier.

Bots we use as enemies can be bots that come with RoboCode, self written bots or bots that won online tournaments, informations and rankings of these tournaments can be found on RoboRumble[4]. The code for this project will be written in Java and available on our Google Code page [5]. We planned our activities with the Gantt chart that can be found in Appendix A.

## 2 Project goals

The main goal of this project is to create a Java program that evolves RoboCode bots. These bots will evolve based on their performance, this performance needs to be tested. We use the results of multiple battles, acquired by the benchmark tool in RoboCode environment to give a bot a certain "fitness" score. The evolution is done by manipulating the genotype of the bots. Therefor we need to create code that can add, change or remove genes in the genotype and translate the genotype to the phenotype.

## 3 Method

In order to evolve a powerfull agent using genetic programming, we first need to understand the underlying principles. This implies research on two subjects, powerful behavior in RoboCode and genetic programming.

Powerful behavior is documentated on RoboRumble, information about how to design a genetic algoritm is found in the article "Evolving 3D Morphology and Behavior by Competition" [2]. After acquiring enough information, we will start implementing the genetic algorithm, we tried using the JGAP framework[6], but ended up creating our own framework. More information on this is in section 5.

### 3.1 Genotype

The genotype describes a genetic constitution of the bot. The genotype is created in the class *Chomosome.* A chromosome is a set of genes, these are created using the class *Gen.* For evolutionary purposes we need to create a population of bots. These populations are created in the class *Population.* These classes together represent the whole genotype. To implement evolutionary processes, we need to create two different representations of the genotype.

One datastructure holds information about certain method call that controls the bot. These calls need arguments to initiate, these are also captured in the datastructure. These arguments are mathematical formulas.

[1]RoboCode home page: http://robocode.sourceforge.net/
[2]Genetic Programming wiki: http://en.wikipedia.org/wiki/Genetic_programming
[3]RoboCode with JGAP home page: http://jgap.sourceforge.net/doc/robocode/robocode.html
[4]RoboRumble home page: http://robowiki.net/wiki/RoboRumble
[5]Google Code page: http://code.google.com/p/aesi/
[6]JPAG home page: http://jgap.sourceforge.net/

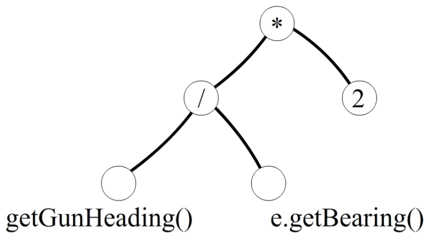


Figure 1: An expression tree.

The second datastructure was needed because we value the potential of our program to evolve impressive mathematical functions highly. An *ExpressionTree* is a binary tree holding both operators and values. Figure 1 shows the expression tree that represents equation 1. This representation allows us to easily mutate expressions. To add, remove or alter a node one only has to recursively walk down a tree to the preferred node or leaf that needs to be altered.

$$\frac{getGunHeading()}{e.getBearing()} * 2 \tag{1}$$

These datastructures are implemented in *AESIGene*. We have decided that each relevant event such as *onHitByBullet* or *onHitWall* can be filled with Java code so each such an event represents a gene. Genes are filled with the previously mentioned datastructures and are mutated during evolution. Representing information on such a gene is currently not yet implemented but should be so in the future when we need to safe data in order to benchmark.

## 3.2 Phenotype

The phenome is the resulting behaviour of the bot. Currently no real powerful behaviour has been generated. This is because not every important aspect is implemented. More of this can be found in the section *result*.

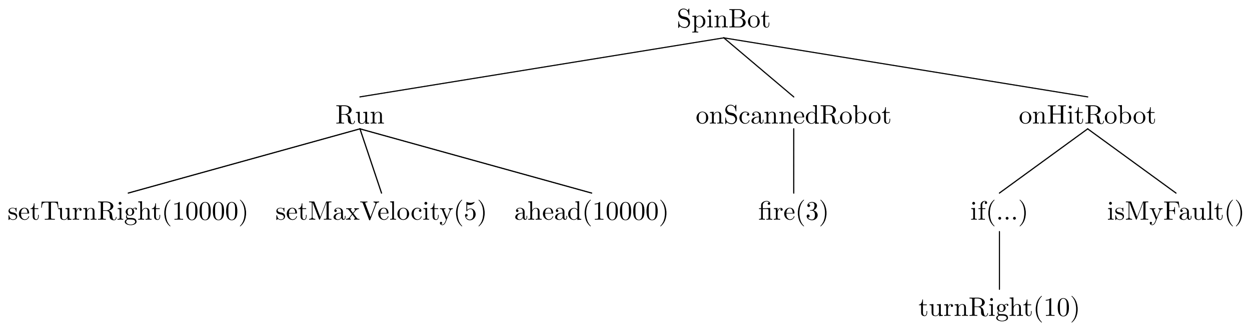
## 3.3 Meta language

We need to design a meta language that creates java code from a genetic code. The genetic code is a string of digits, the location of the digit tells what gene is described, the value describes what this specific gene does. It is important to keep the genetic code meaningful and understandable so we can use the genetic code to “see” what kind of behaviour is evolved. This prevents our evolutional process to become a black box we use to evolve bots but where it is hard to really understand what is going on.

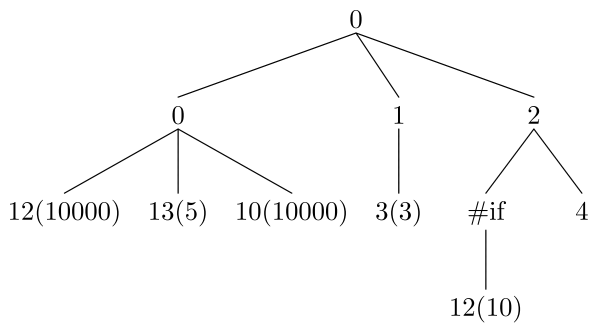
To properly describe how this works, we will show how java code of a simple bot, provided by RoboCode, can be represented in our genes. This is the code of the simple bot:

```
package sample;


import robocode.AdvancedRobot;
import robocode.HitRobotEvent;
import robocode.ScannedRobotEvent;

import java.awt.*;


public class SpinBot extends AdvancedRobot {

        public void run() {
                // Set colors
                setBodyColor(Color.blue);
                setGunColor(Color.blue);
                setRadarColor(Color.black);
                setScanColor(Color.yellow);

                // Loop forever
                while (true) {
                        setTurnRight(10000);
                        setMaxVelocity(5);
                        ahead(10000);
                }
        }

        public void onScannedRobot(ScannedRobotEvent e) {
                fire(3);
        }

        public void onHitRobot(HitRobotEvent e) {
                if (e.getBearing() > -10 && e.getBearing() < 10) {
                        fire(3);
                }
                if (e.isMyFault()) {
                        turnRight(10);
                }
        }
}
```

This code can be described in a tree, like this:



The computer will see the tree like this:



Here the upper number is an index of the gene, and the numbers on the second row are numbers indicating the index of the gene array. Under that we have array indices of the methods and their arguments, that are positioned in arrays with all the methods we can call and the arguments we can give them.

## 3.4 Fitness function

Fitness is crucial to determining the trajectory of the evolutionary process. Based on the fitness score, the program decides which chromosomes are transferred to the next generation of bots.

With the fitness function, we translate battle results into a appropriate fitness value. Our basic scoring measure is the fractional score F, which is computed using the score gained by our bot $S_b$ and the score gained by its adversary $S_a$.

Calculating the fitness score this way, we encourage our bot not only to maximize its own score, but to do so at the expense of its enemy.

$$F = 100 \times (1 + \frac{S_p - S_a}{S_p + S_a})$$

Equation 2. The fitness function.

In early stages of the evolutionary process, bots obtain no points at all. To prevent the program dividing with zero we created a catch in the code. If both scores are zero, both bots will be given a zero as fitness value.

## 3.5 Evolve

Because we still need to finish some major functions we have not started evolving as of yet. We do have some results which can be found in the section *results*.

# 4 Results

Currently our program has evolved a single bot, which is capable of doing something. The results are, as discussed in previous sections, minimal, but prove promising. The robot shows action such as moving forward and rotating both it's body and gun but not yet a change in behaviour when events are caught. This means we completed the basis but need to focus on the evolutionary process that will create bots with more interesting behaviour.

We need to be able to compare our results to other bots to give the results a meaning. During this project we will create some bots ourselves and download bots that have proven to be powerfull in online tournaments. These bots will be battling against our bots. This way we can find out if the bots evolved with genetic programming can stand against human made bots. There is a paper “GP-Robocode: Using Genetic Programming to Evolve Robocode Players” [1] about RoboCode bots evolved with a genetic framework that participated in a online tournament with 27 participants. Other bots were all human written. The bot (“GPBot”) ended third.

# 5 Problems

## 5.1 JGAP Framework

Our decision to program in the JGAP Framework has turned out to be quite a 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.

These results were very dissapointing. Our bots did evolved, but they did not get any high fitness values. We found out that after a few hunderds of rounds the bots devolved, bringing us back where we started.

We first thought this was because our fitness function was incorrect. We updated the function but this did not change the results.
We then did a lot of simulations to test our parameters. Using a different population size and different amount of battles to calculate the fitness value did have influence, but did not solve the devolution.

Our next hypothesis was that 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 could not figure out how JGAP exactly works. The framework is quite big, but its options are not adequately documented.

To demonstrate our troubles with this framework, we describe the *Natural Selector* object and its implementation by JGAP might demonstrate our troubles with this framework. At one point in time we tried to simply safe the chromosomes with a high fitness value. 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.

1. Learning to set up the framework and basic functioning. JAR files, buildpaths, classpaths, Eclipse. It's all very confusing.
2. The JGAP Framework worked like a black box. With only a known input and output, it is difficult to locate problems inside the program.
3. Bad and incomplete documentation. A function needs input, but what format is this input? What does this function even do? These are only two examples of many problems we had with JGAP and where the documentation did not help us at all.
4. Unnecessary functions. Our design for the evolutional process is quite simple and elegant. We found that using complicated JGAP code, these simple things turned out to be very difficult. For example, if we want to select the 10 fittest chromosomes of our population we had to go through a chain of registred selectors or find out how or why the configuration is locked or reset.
5. Dismissable results. De-evolving at times without any reason.
6. Unexpected behaviour. Some poorly chosen engineering decisions result in forced use of classes we known nothing about. For example *IChromosome*, because this is an interface, it has certain restrictions. We wanted to store addiontal information in such a class, for example a method to create code from a chromosome. This is not possible in an interface. JGAP works around this problem with a slot for the ApplicationData, which can be any object. This is object is however sometimes, seemingly randomly reset to *null*.

## 5.2 Leaving JGAP Framework Behind

With more and more experience with genetic programming, it became more and more clear that we did not need such a complicated framework. 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.

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 by implementing algorithm 1.

Initialize Population true Evolve Bottom 50% Evaluate Bottom 50% Sort Population Replace Bottom 50% With Top 50%  Main Evolution Algorithm

After the first simulations it became clear that the evolutionary process was not going as expected. While the robots did not seem to devolve, new or mutated chromosomes did not seem to have any improvements. New generations had similar fitness values, but sometimes there was a very big decrease or increase in the fitness value.
This happened even though new chromosomes should be only slightly different from the other chromosomes. Even with our newly gained debug capabilities and advantages of not using JGAP but our own code we could not figure out what was causing these results.
For further testing purpusses we created a graphical representation of the fitness values throughout the evolution process so we can easily find out when in time what fitness value were achieved. This helped u a lot, especially while debugging or evaluating performance of various parameters.

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, shown in figure 2 clearly explain what is going on.

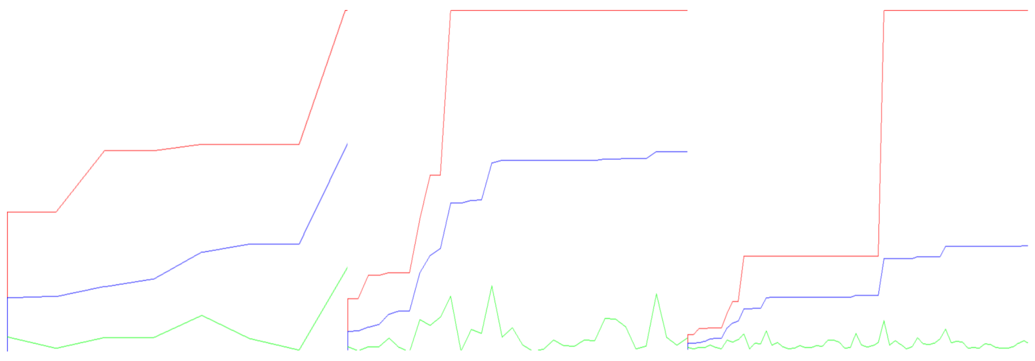

(a) 3.7 After 7 rounds (b) 9.5 After 33 rounds (c) 35 After 65 rounds

Figure 2: Graphical representation of fitness values throughout generations at different times

The results clearly show that the highest fitness value remains in the population. It also indicates that the top fitness values remain in the population as the average does not decrease. However, the average of newly mutated chromosomes is always dissapointingly low. This was thought to be extremely unlikely as one simple mutation should not have such a major impact on the whole generation causing the fitness value drops to zero. In the mean time, the best robots showed promising results!

For a while we had no idea what was going on. When we discovered that sometimes the best robot the program wrote to a file wasn't actually the best robot in the program that we build in a check that reevalutes the complete population every 10 evolution rounds. The result is shown in figure 3.

It seemed the fitness value was constant but the chromosomes continued to evolve. After examining our algorithm even further we were still unable to find any bugs.
When we almost gave up we remembered how Java does not always stores a copy of an object but actually a copy of the reference to the object. This would explain everything and with the Java/Eclipse debugger we could validate this. The easy fix was to clone our chromosomes.

The first results proved promising but still not what we expected. After more research we learned we had to clone every object, and not just the chromosomes. Our program stores objects in objects using arrays. We assumed incorrectly cloning the first layer (chromosome) would be sufficient, instead we had to clone up untill the deepest stored objects in our program. Every layer (see figure 4) of our program had to be cloned in a specific manner. It took us a lot of time to figure out that this actually had to be done for each layer. Especially the last layer, the *ExpressionTree*

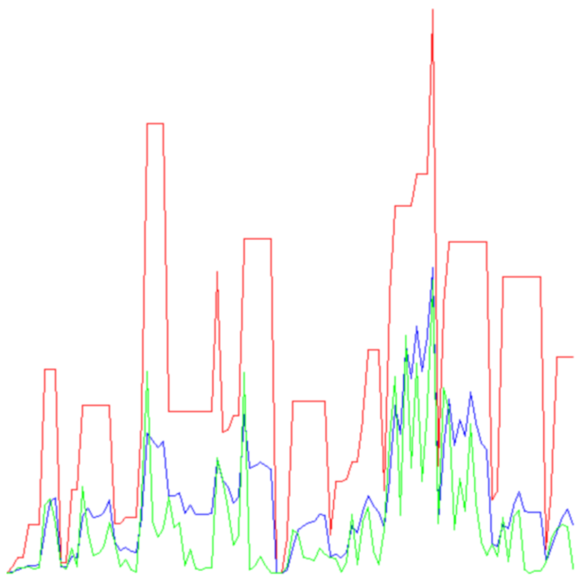


Figure 3: The results with a reevaluation every 10 rounds

which is recursively stored was tricky.

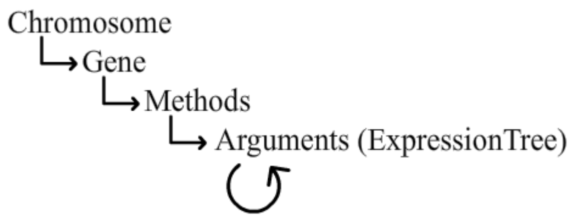


Figure 4: The depth of our program depends on the depth of an *ExpressionTree*

After this ephiphany implementation was easy and we could finally start evolving. As we had expected in the beginning, robots did not devolve and when they did only minimally. The results in figure 5 clearly shows a rising average fitness value! The ninetieth generation was able to deliver robots that could beat our default opponent *Spinbot*. We were very happy with this!

## 5.3 Creating our own Framework, a Good Decision?

Creating our own framework definitely helped us find, identify and solve the problems. Our well structured code now allows us to easily change parameters and even the basic functioning of the program. However, we currently do not know if the unexpected results produced by JGAP were produced by the same kind of programming mistakes. If this is the case some of the problems might not have been caused by JGAP.

While it is easy to change the parameters and basic functioning of the program, changing something more advanced such as the manner on which the program selects the best genes costs significantly more time. Perhaps even more time in our framework then in it would have had in JGAPs.
For example, there are multiple efficient methods to gather information. The robot could scan the entire room by turning the radar continuously or the robot could keep the radar fixed on the target. It is generally accepted that the latter method results in a higher fitness value. However, bots in earlier generations cannot have such complicated behaviour and will almost always perform simple scan behaviour that evolves into a less powerfull scan method. Once a bot has a certain

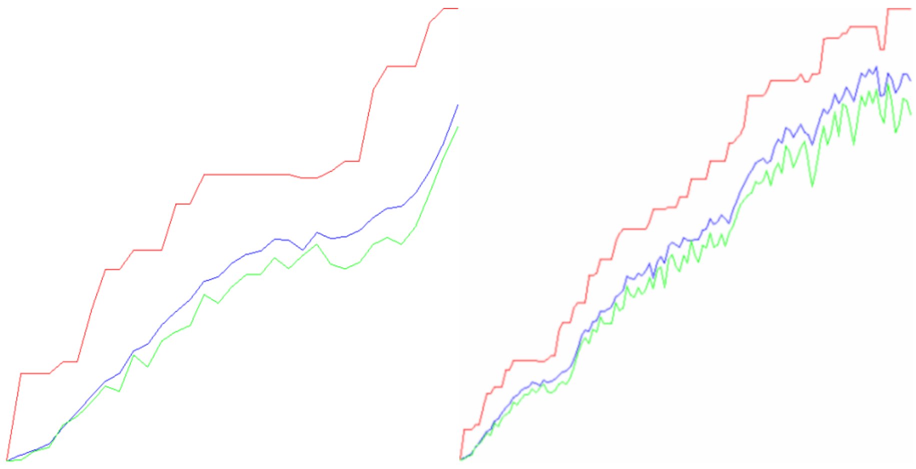

(a) 41 After 41 rounds (b) 120 After 102 rounds

Figure 5: Graphical representation of fitness values throughout generations at different times showing correct results.

method for scanning, it is almost impossible to evolve the different, better method as there are multiple evolution cycles needed to evolve to the different information gathering method. In the mean time, while evolving, it has no information gathering method making it extremely unlikely that the robot achieves an high enough fitness value to survive and evolve further.

There are two things that could fix this. We could select the genes that survive based on their fitness value and a chance or we could implement cross mutation.

## 6 Background information

### 6.1 RoboCode Game

Time is represented in ticks. There are several things happening each tick. For example the robot is allowed a certain amount of processing time to calculate and execute actions. The game world is also updated, meaning that objects are relocated according to the physical laws of the game and potential collisions between objects and the results of such collisions are calculated.

### 6.2 Robot Anatomy

A robot consists of three seperate parts, the radar which is mounted on the barrel which is mounted on the body. All parts can be rotated seperately. The barrel fires bullets, the radar scans for other tanks and the body drives around in the arena.

### 6.3 Robocode rules

The Robocode environment has specific rules constraining our bot. A fight starts by placing all participating bots in the arena at random places. All bots have a certain amount of energy. Energy is lost by getting shot by an enemy or ramming walls and enemies. Certain actions also use up energy, for example shooting a bullet. A bot can gain energy by hitting an opponent. When a bot is out of energy, it is disabled and explodes when hit by a bullet.

### 6.4 Programming the Robot

Programming a robot means *extending* a certain robot class from RoboCode, we use Advanced-Robot. This allows us to program methods that catch events that are launched by RoboCode

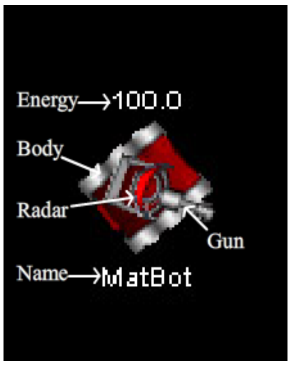


Figure 6: The anatomy of a RoboCode bot

(such as *robotScannedEvent*, this event is launched when a bot scans another bot) and program what we would like the robot to do in such cases (such as *rotateGun*, this method rotates the barrel changing the direction of a fired bullet). The robots behavior relies completely on what events occur during a battle. The robots actions during a certain event is generated by the evolutionary process.

# 7 Planning

## 7.1 Documentation

- create workplan, everybody
- round of halfway report and presentation, M. de Waard
- present halfway presentation, everybody
- finish end report and presentation, J. Rooijmans
- present final product, everybody
- maintain documentation, M. Inja

## 7.2 Initiation

- create planning and set up websites, M. Inja
- present workplan, everybody
- familarization with programs, M. Inja, M. de Waard
- read articles about genetic algoritms, J. Rooijmans

## 7.3 Genetic programming, single bots

- create genotype, J. Rooijmans
- create fitness function, M. de Waard
- evolve, M. Inja

## 7.4 Test phase

- benchmark with RoboResearch, M. de Waard
- compare RoboCode with JPAG, M. Inja
- design and compare with own bot, everybody
- compare with bots from the internet, J. Rooijmans

## 7.5 Genetic programming, teams

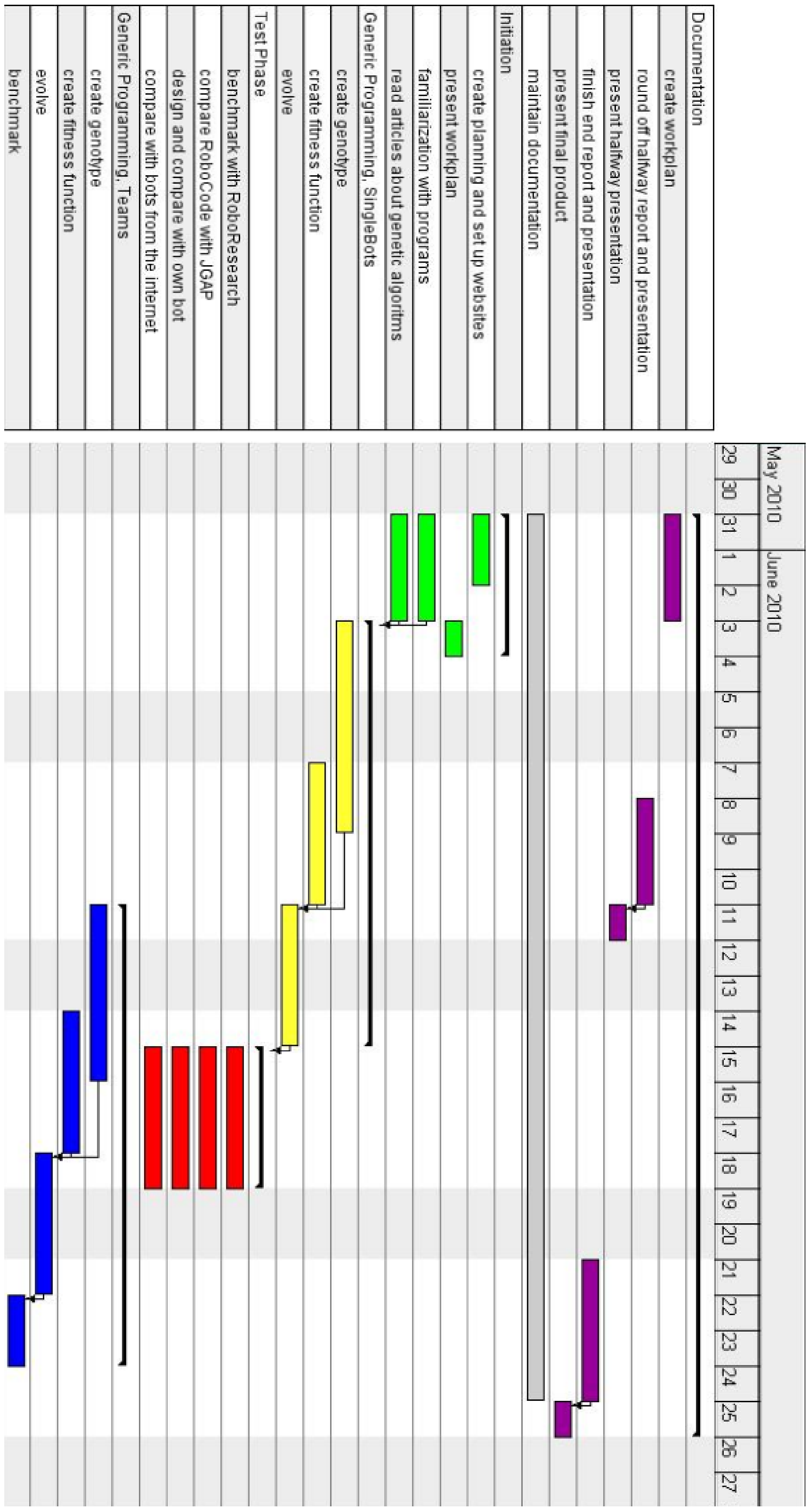
- create genotype, J. Rooijmans
- create fitness function, M. de Waard
- evolve, M. Inja
- benchmark, everybody

## References

[1] Y. Shichel, E. Ziserman, and M. Sipper. GP-robocode: Using genetic programming to evolve robocode players. In M. Keijzer, A. Tettamanzi, P. Collet, J. I. van Hemert, and M. Tomassini, editors, *Proceedings of the 8th European Conference on Genetic Programming*, volume 3447 of *Lecture Notes in Computer Science*, pages 143–154, Lausanne, Switzerland, 30 Mar. - 1 Apr. 2005. Springer.

[2] K. Sims. Evolving 3D morphology and behavior by competition. *Artificial Life*, 1(4):353–372, 1994.

## A Planning


Documentation
create workplan
round off halfway report and presentation
present halfway presentation
finish end report and presentation
present final product
maintain documentation
Initiation
create planning and set up websites
present workplan
familiarization with programs
read articles about genetic algorithms
Generic Programming, SingleBots
create genotype
create fitness function
evolve
Test Phase
benchmark with RoboResearch
compare RoboCode with JGAP
design and compare with own bot
compare with bots from the internet
Generic Programming, Teams
create genotype
create fitness function
evolve
benchmark
May 2010
June 2010
29 30 31 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27