Evolving RoboCode Tanks

Alex Goldhoorn

Abstract—**In this paper we discuss an experiment in which we try to evolve the variables of a robot. A game named RoboCode, made by IBM Alphaworks, has been used as a simulator. In this simulator robot tanks fight against each other. The robot is composed of behaviours which output a vector representing the direction. This is based on the subsumption architecture and the potential fields method. For the games we played battles in teams of robots against other teams. All these behaviours contain several variables. In the experiments these variables were evolved. The goal of this research is to find out if the evolved robots perform better than the robots which use fixed values for the variables in the behaviours. Tournament selection was used for selecting new generations and parents for crossover.**

I. Introduction

Evolution is something used by nature to improve the organisms and adapt them to their environment, which Charles Darwin wrote down in his work The Origin Of Species. Evolution can also be used in robots. In robots the genes can be seen as variables which we want to evolve.

In our experiment we want to find out whether evolution can improve the robot compared to the use of a non-evolved robot. More precise our research question is if the robots perform better when their variable values are evolved compared to fixed variables.

To do this experiment we used a simulation game of robot tanks: RoboCode. In this game one or more robots can fight against each other by shooting. In our experiments we used robot teams and the battles were one team to another team. The goal of the game is to get the highest score. This score depends on, among others, how much bullets were a hit and how many team members survived.

II. Evolution

The idea that populations of organisms evolve by natural selection was introduced in 1859 by Charles Darwin in his work The Origin of Species. The concept of natural selection is that nature selects individuals that are more likely to survive in the future. Individuals that are less likely to survive in the environment have a smaller chance of reproduction. Therefore the fitter individuals survive.

In the 1960s artificial evolution was started by Holland and Fogel in the USA and by Rechenberg in Germany (Pfeifer and Scheier 1999). They created different evolution systems, but in this paper we use Genetic Algorithms (GAs).

In the experiment we tried to optimize the variable values of robot behaviours (modules which operate certain parts of the robot like firing). Evolution is used to search for an optimal in a search space. When this space is too big then it will take a long time to reach an optimum, if it is too small it might not get to an optimum.

A. Genetic Algorithm

Evolution works with populations of individuals. These individuals contain properties stored in genes. In our experiments the genes contain variable values.

New individuals are generated by mutation and by the combining of individuals of the current population, this is called crossover. The best individuals can be chosen for ‘reproduction’, but there are more ways of selecting, see the section selection.

To store the properties of individuals (e.g. the values of variables) genes can be used. These genes can be represented as a row of bits. By changing the values of the bits, the individual changes. This is done by the crossover and mutation steps in the evolution process as will be discussed later in this section.

The genetic algorithm is listed in Figure 1 (Mitchell 1997). The population of individuals is initialized with random individuals. Then they need to be evaluated. This is done by a fitness function. A fitness function can be different for every type of problem. In this experiment the score returned by RoboCode will be used as fitness function for an individual.

In the next step the evolution will continue until there is an individual with a fitness value of at least the fitness threshold.

B. Selection

For the next generation (i.e. the new population) individuals of the current generation need to be selected. Goldberg and Kalyanmoy (1991) mention four commonly used selection schemes:

1. proportionate reproduction;
2. roulette wheel selection (or ranking selection);
3. tournament selection;
4. Genitor (or “steady state”) selection.

Selection mechanism 2 and 3 will be shortly explained.

1) Roulette wheel selection

Holland introduced a method in which the individual’s probability of being selected is proportional to its fitness (Pfeifer and Scheier 1999).

Alex Goldhoorn is a master student of the department of Artificial Intelligence, University of Groningen, The Netherlands (e-mail: a.goldhoorn@student.rug.nl).

GA(*Fitness*, *Fitness_threshold*, *p*, *r*, *m*)
Fitness: A function that calculates the fitness value of an individual.
Fitness_threshold: A threshold specifying the termination criterion.
p: The population size.
r: The fraction of the population to be replaced by crossover offspring.
m: The mutation rate.

- Initialize population: P Generate p individuals at random
- Evaluate: For each h in P, compute $Fitness(h)$
- While $[\max_h Fitness(h)] < Fitness\_threshold$ do
  Create a new generation, P_s:
  - *Select*: Select $(1 - r)p$ members of P to add to P_s. To be done by a selection mechanism (e.g. roulette wheel selection or tournament selection).
  - *Crossover*: Probabilistically select $rp/2$ pairs of hypotheses from P, according to the selection method. For each pair (h_1, h_2), produce offspring by applying crossover. Add all offspring to P_s.
  - *Mutate*: Choose m percent of the members of P_s with uniform probability. For each, invert one randomly selected bit in its representation.
  - Update $P \leftarrow P_s$.
  - For each h in P, compute Fitness(h).
- Return the hypothesis with the highest fitness from P.

Figure 1 The GA algorithm (Mitchell 1997) which returns a hypothesis with a fitness value of at least *Fitness_threshold*.

The selection can be seen as a roulette wheel where every individual has a part that is proportionate to its chance of being selected. An advantage is that not only the individuals with the highest fitness will be used, but also others have a chance. This is important because the not so good individuals might contain some good properties, but which do not appear in combination with its other properties.

Disadvantages are that it needs to sort the population first (on fitness), which requires computation time. And secondly the selection pressure cannot be controlled.

The time complexity of sorting is $O(n \log n)$ (Goldberg and Kalyanmoy 1991) with standard algorithms. The selection can be done between $O(n)$ and $O(n^2)$. Therefore the roulette wheel selection method is in $O(n \log n)$.

2) Tournament selection

A selection method that has adjustable selection pressure and does not need to be sorted is tournament selection (Goldberg and Kalyanmoy 1991). The algorithm is simple: first choose a number of individuals (the tournament size t) randomly from the population and then select the best individual from this group. And repeat this process as often as individuals are needed. The selected individuals can be used for reproduction. Tournaments are often held between pairs of individuals ($t = 2$), also called binary tournament selection.

The selection pressure can be adjusted by changing the tournament size. Increasing t increases chance of being selected for each individual. Therefore there is a greater chance of selecting an individual with a higher fitness. So in general the 'solution' will converge faster then when a greater t is used, i.e. has a higher selection pressure (Legg et al. 2004). When t is smaller also individuals with a small fitness value will be selected. This improves the diversity, which is important to prevent suboptimal solutions.

The time complexity of tournament selection is $O(n)$ (Goldberg and Kalyanmoy 1991), because each competition requires the random selection of a constant number of individuals of the population. The comparison among the individuals can be done in linear time. Therefore it has time complexity $O(n)$.

Goldberg and Kalyanmoy (1991) found linear ranking (roulette wheel selection) and a variant of binary tournament selection to have identical performance, but tournament selection is preferred due to its lower time complexity.

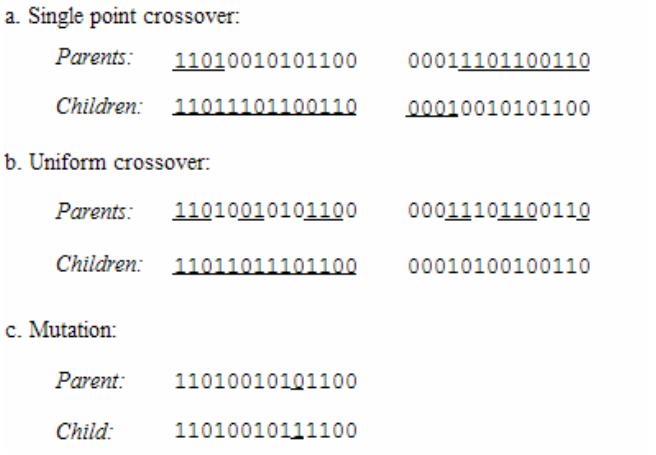

Figure 2 Crossover and mutation on binary gene strings. Single point crossover (a) where the genome is 'cut' at only one part. In uniform crossover (b) the genomes can be cut at more places. And at mutation (c) only one randomly chosen bit is flipped.

C. Crossover

Crossover is used to create a new genome (child) from two other genomes, its parents. The idea is to replace one part of the genome of parent 1 with information in the part at the same place of the second parent. This is illustrated in Figure 2.

The most simple case is to replace only one part, this is called single-point crossover (Figure 2a). But two points (two-point crossover) or more (uniform crossover, Figure 2b) can also be replaced.

D. Mutate

Another way to generate new individuals is by mutation. This is simply flipping one (or more) chosen bit (see Figure 2c). The bit or bits to be flipped are chosen randomly with a certain chance. In Figure 1 m percent of the population is chosen to be mutated.

III. Robot Architecture

Robot architectures are: "software systems and specifications that provide languages and tools for the construction of behavior-based systems." (Arkin 1998).

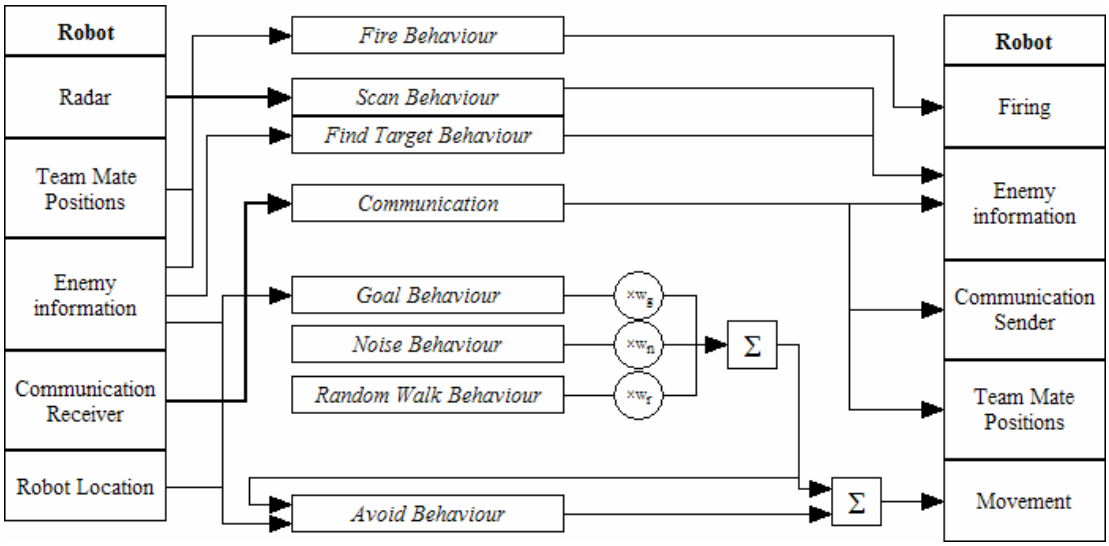

Figure 3 The robot architecture. The robot (left and right) contains several information modules (enemy and team mate information) and operation modules (firing, movement and communication). In the middle (italic) are the behaviours. Not all the robots have all the behaviours. The communication module is not directly implemented as a behaviour, but it is used by several behaviours.

In the robots we used for our experiments we used a mix of subsumption architecture and potential fields.

A. *Behaviour-based robotics*

In the GOFAI (Good Old Fashioned AI) the sense-think-act cycle was used in robot processing, these systems reacted rather slowly and are not (very) biological plausible. The subsumption architecture was introduced by Brooks and started the behaviour-based robotics paradigm.

Subsumption decomposes the robots control architecture in a set of behaviours (Pfeifer and Scheier 1999). Each behaviour can have several inputs (e.g. sensor readings) which can be used by other behaviours. Each behaviour outputs control actions, e.g. the speed and direction of the robot.

The behaviours are layered according to their relative importance. Higher layers can subsume lower layers by suppression of input or inhibition of output.

B. *Potential Fields*

Khatib (1985) and Krogh (1984) developed the potential field method for generating smooth trajectories of the robot. The method generates certain (vector)fields around certain places. Here goals are the attractors and obstacles repulsors. The result, the direction (and speed) of the robot, is calculated by adding the vectors that are present at the robots location.

C. *Behaviours with vector output*

In our experiment the robots have different behaviours, but they do not directly subsume each other. They only share information such as enemies. Figure 3 shows the robot architecture of our experiment robots.

Each behaviour outputs a vector containing a magnitude and angle. The magnitude represents the speed, the angle represents the direction it should turn. Some behaviours do not output a vector, because they are not used for driving, but for shooting or target detection.

The results vector, the speed and direction the robot will move, is the sum of the vectors of all behaviours. But each behaviour output is first multiplied with a weight for each behaviour. This allows us to change the influence of different behaviours on the overall result.

IV. RoboCode

RoboCode (Sing Li 2002) is a virtual battle simulator made in Java originally created by IBM alphaWorks. Robots fight against each other in the arena. This arena is a flat rectangular field (Figure 4) with walls on all sides. There are no obstacles in the field.

A. *The robots*

The robots can move over the field. They have a gun and a radar. The amount of firepower is adjustable. The radar and

Figure 4 RoboCode running a battle of two teams.

turret can move independently from the robot.

The robots have an energy level which indicates their 'health'. An energy level of zero means that the robot is disabled. A robot can increase its energy level by hitting another robot, the higher the firepower, the more energy it will gain. The robot loses energy by being hit by a bullet, another robot or by hitting a wall. Also firing costs energy, and more fire power costs more energy, but it gains energy when the robot hits another robot by shooting it.

The radar detects other robots. In the Java program the radar generates an event which can be used in the robot. The radar provides the following information of another robot: name, velocity, heading, remaining energy, the angular difference between the other and itself and the distance to the robot. The positions in RoboCode are absolute and the origin (0,0) is at the lower left corner. Every robot can always get its own location. Therefore it is easy to calculate the location of another robot using your own position, the distance and the angular difference.

B. Teams

The games can be between one or more robots, but it is also possible to make a team of robots. Some advantages of using a team of robots are: the possibility to cooperate and allocating different goals to different robots (e.g. protecting another robot). The robots in a team must check whether the robot is a team member before the shoot.

The first member of a team in RoboCode starts with a higher energy level (200 instead of the default 100). Also droids start with a higher energy level (120), but they have they do not have a radar. Therefore droids should use communication to receive information about enemies from other team members, which do have a radar.

C. The game

RoboCode uses time ticks in which every robot can do computations for a limit amount of time. After each tick the robots are moved (if their velocity is not zero) and the bullets are moved. Also after every time tick the visual view of the game is updated (if display is enabled).

The main goal of the game is to get the highest score. The score is based on whether the robots survived, the amount of damage it caused by bullets and by ramming (collision) and some other bonuses.

A disadvantage of RoboCode is that it is not very realistic. The radar has no noise and can detect almost everything about the other robot including energy level. The field is perfect rectangular without any obstacles and there is no fuel needed for driving (except for energy). But still there are enough reasons for using such an unrealistic simulation. The simplicity is good for a relative easy robot design, but still there are enough difficulties in defeating opponents, because there are an 'infinite' amount of strategies. This makes it interesting for evolving robots, because there are enough variables which can change the outcome of a game.

```
robocode.battle.rules.gunCoolingRate=0.1
robocode.battle.rules.inactivityTime=100
robocode.battle.numRounds=3
robocode.battleField.width=3500
robocode.battleField.height=2500
robocode.placement.sd=150
robocode.placement.margin=500
robocode.options.maxScanRadius=1200
```

Figure 5 The settings used for the RoboCode battles which are set in the `battle.properties` file. The last three settings are not available in the standard RoboCode version.

D. Other projects

There have been done several projects and experiments using RoboCode. Frøkjær et al. (2004) give a detailed report from their creation of a RoboCode team. They used several learning techniques among which genetic algorithms. Eisenstein (2003) used evolved RoboCode tanks by encoding a simple form of programming (TableRex) into genes, this resulted in rather good robots which beat some good hand made robots. Sipper et al. (2005) discuss their evolved game players of three games, among which RoboCode.

V. Method

In our experiment we evolved the robots to improve their performance in the RoboCode games. To detect the improvement of the robot team by evolution, we need to compare it to a default case. We used the robots with fixed values for the variables to compare the evolved robots to. Some of the fixed values were only slightly optimized by hand.

A. The RoboCode Games

In our experiment we use a team of robot which competes against other teams of the same composition. Our team exists of one leader, two normal robots and six droids. The used battle settings are listed in Figure 5. The default settings of RoboCode were used, except for the field size which was made 3500×2500. Besides that we used a slightly changed version of RoboCode. This version contains an option to change the maximum scan radius, this is the maximum distance at which the radar can detect another robot. Another added option is to let the teams start in a different corner and not randomly spread across the field.

B. The Robots

All robots of our team maintain a list of the enemies and of its team mates. From each enemy the position, speed, heading, energy and time of the scan are stored. These can be used to guess the new position of the robot at a later time, where we assume that the robot moves linearly in the same direction with the same speed. This method works best when the time between scanning and using the guessed position (for example when the robot wants to shoot) is short.

The positions of the team mates are used to prevent colliding into the robots. Or to prevent shooting at the robot

when that robot is in the line of sight, i.e. an enemy is behind it.

The enemy information and team mate positions are sent to each team mate at every time tick. This way every robot, including the droids, has information of each team mate and every seen enemy. The advantage of the team is that they can search enemies in a greater area than only one robot can.

All the robots have behaviours, which together make the robot move, shoot and communicate.

C. The Behaviours

The robots have different behaviours which all do a part of the total processing, a total view of all the behaviours can be seen in Figure 3. Every behaviour can output a vector, which has a magnitude and angle (i.e. velocity and direction), but not all do. Several behaviour were made, each behaviour has one or more variables which are evolved. These variables are listed in Table I. Every behaviour which outputs a vector also has a weight, except for the avoid behaviour.

1) Avoid Behaviour

Avoid Behaviour avoids walls and obstacles by returning a repulsive vector perpendicular to the faced on wall or object. The turn distance variable is the distance (in pixels) to the object or wall from which it needs to start turning to avoid it.

2) Fire Behaviour 1

This behaviour fires when the robot has an enemy (set by scan behaviour or find target behaviour). Before it fires it calculates the place where the enemy should be at the arrival time of the bullet. Then it checks whether there is a team mate in front of the enemy to prevent shooting team members. This behaviour does not have any variables to be evolved.

3) Fire Behaviour 2

The second fire behaviour is a more advanced version of the fire behaviour. It makes the fire power depending on the distance of the enemy. As can be see in Table I the closer the enemy the higher the fire power. Both fire behaviours were tried during the evolution runs, but only one fire behaviour was used at a time.

4) Scan Behaviour

This behaviour 'listens' to the radar and selects an enemy or updates the enemy information. The closest enemy is selected. When an enemy is selected it only scans in the direction of the enemy, but when no new enemy information has been received for a longer time it rescans the whole environment (i.e. turns it radar a full circle). The time is defined by the variable full scan time.

5) Find Target Behaviour

The find target behaviour finds a target from the list of enemies, so it does not (directly) use a radar. It selects an enemy based on its distance and its type (leader or normal robot). The variables distance to normal and leader are the maximum distances for the robot to select it as an enemy.

6) Goal Behaviour

This behaviour makes the enemy the goal to drive to. The only evolution variable is the slow down distance, this is the distance from the goal from where the robot should start to slow down, to prevent overshooting the goal.

TABLE I
BEHAVIOUR VARIABLES

Name (unit)	Min.	Max.	Fixed	Evolved
Avoid Behaviour				
Turn distance (pixels)	0	1000	150	695
Fire Behaviour 1 (no genes)				
Fire Behaviour 2				
Close fire power (energy)	0	3.5	3.0	2.6
Medium fire power (energy)	0	3.5	2.0	1.9
Far fire power (energy)	0	3.5	1.0	2.2
Close distance (pixels)	0	500	20	209
Medium distance (pixels)	50	1500	100	550
Far distance (pixels)	100	3500	500	1286
Scan behaviour				
Full scan time (time ticks)	1	10	4	5
Find target behaviour				
Distance to normal (pixels)	0	2000	1000	1478
Distance to leader (pixels)	0	2000	1500	716
Goal behaviour				
Slow down distance (pixels)	0	500	100	255
Weight	0	1	0.4	0.45
Noise behaviour				
Max. velocity (pixels/tick)	0	750	31.3	378
Max. angle (rad)	0	2π	$\pi/16$	4.3
Weight	0	1	0.5	0.73
Random walk behaviour				
Max. angle (rad)	0	2π	π	3.3
Direction duration (time tick)	1	500	20	184
Weight	0	1	0.8	0.73

This table shows all the variables of the behaviours which were evolved. Min and max are the bounds (minimum and maximum value) of the variables. The evolved column shows the average values of the variables from seven evolution runs.

7) Noise Behaviour

The noise behaviour generates a noise vector, a vector with random angle and magnitude. The maximum angle and velocity for the noise vector are variables.

8) Random Walk Behaviour

This behaviour is comparable to the noise behaviour, but random walk keeps the random vector for a certain time. This time, the direction duration, and the maximum angle are variables which were evolved.

All of the robot types have a slightly different collection of behaviours. But for each robot all of its behaviours are executed every time tick. The result vectors (if the behaviour has one) are summed. But before summing them, the vector magnitude is multiplied by a weight factor for the particular behaviour as can be seen in Figure 3 (w_g, w_n and w_r). In this manner the influence of each behaviour can be changed. These weights are also evolved. This result vector is checked for the maximum speed. This is done to prevent it from speeding too fast and thereby missing its goal or hitting a robot or wall.

The summing of all behavioural vectors caused problems when doing wall and robot avoidance. This occurred because the avoid behaviour does not know the summed output of the

other behaviours. Because of this the avoid behaviour may have too little influence and thereby still colliding into the wall or the robot. To prevent this the vectors of all behaviours are summed, except for the avoid behaviour. The avoid behaviour then calculates the avoid vector based on the summed vector of the other behaviours. In this order it is possible for the avoid behaviour to adapt its magnitude vector to the summed vector of the other behaviours.

D. Team Members

As said before our team exists of one leader robot, two normal robots and six droids. The only difference between these is that the droids do not have a radar and the others do. They have a slightly different collection of behaviours.

All robots have: a noise behaviour, goal behaviour, fire behaviour and random walk behaviour. The droid also has a find target behaviour. The leader and normal robot have a scan target behaviour, because they have a radar. The droid does not have a radar and therefore it can only use the received enemy information. The find target behaviour searches the list of received enemy information for a target.

What might be noticed is that there is no difference between the leader and the normal robot. This was not planned, because we planned to add a coordinate or find strategy behaviour for the leader. This behaviour has not been made because the results were good enough for evolution without such a behaviour.

E. Evolution

For the evolution of the robots, we created genomes for each behaviour. Each of these genomes contained genes which represented some variables in a behaviour, as listed in Table I.

The evolution is done like in Figure 1, except for the use of a fitness threshold. Instead the number of generations to evolve was used. We did this to be able to do several evolutions with different evolution parameters, but the evolution system also allows us to increase or decrease the number of generations during a run (based upon the results).

The number of individuals, the population size, is also adjustable. An individual contains the genomes of all the behaviours. An option was created for the evolution system to either calculate the fitness (i.e. run RoboCode and analyze the scores) for one randomly chosen individual per generation or for all individuals in the generation.

For the battles opponent teams are needed. Therefore we used the teams of the Robotics course of 2005/2006. Four teams were used. In the configuration the opponents were added to be used for evolution. Per opponent a chance of being selected is set. This was done because some opponent teams performed better than others.

The steps of the evolution program:

1) Initialization

At initialization the population is filled with the number of individuals. The added individuals have genomes with random gene values, but within the bounds of the variables as listed in Table I. The fitness values of these individuals are all set to zero.

2) Choose opponent

For every generation an opponent is randomly selected from the list of opponents. The chance of being selected per opponent can be set (in a configuration file).

3) Calculate fitness

The fitness calculation is done by first choosing an individual. This can be done by iterating the population or by randomly choosing an individual.

Next RoboCode is run with battles of our team (using the genome from the individual) against the chosen opponent.

When RoboCode is finished, the results of the game are used to calculate the fitness. Our goal is to make a team which has a highest score in every battle. So the fitness function should include this score. First we used a relative score (1), with m the total score of our team and o the opponents score.

$$f(m,o) = \frac{m}{m+o} \tag{1}$$

This fitness had the disadvantage of only showing the distance in score distance. But it is more important to have a higher score, independent of the other team. Therefore we now use fitness function (2).

$$f(m,o) = m - \alpha \cdot o \tag{2}$$

For this fitness function the score of our team is the most important, but there is a penalty for a (high) score of the opponent. This penalty depends on α which we set to 0.1 .

When an individual survives several generations, then it is possible it has been chosen for several fitness calculations. Because we do not want to lose the previous score of the individual by overwriting the fitness of the individual, we use a running average of length three. This is also important because the performance of the opponents differ.

4) Selection

The selection mechanism selects half of the population (n/2) for the new generation. We used a tournament selection, because this is a proven selection mechanism and relatively fast, as discussed in section II.

The tournament size t can be set, but cannot be larger than half of the population size (number of individuals), because then it is not possible to select the last few individuals. To search for the best tournament size, several sizes were tried.

The tournament selection is done by randomly selecting t individuals and from that list selecting the best individual for the new generation. This process is repeated until half of the population of the new generation has been is generated.

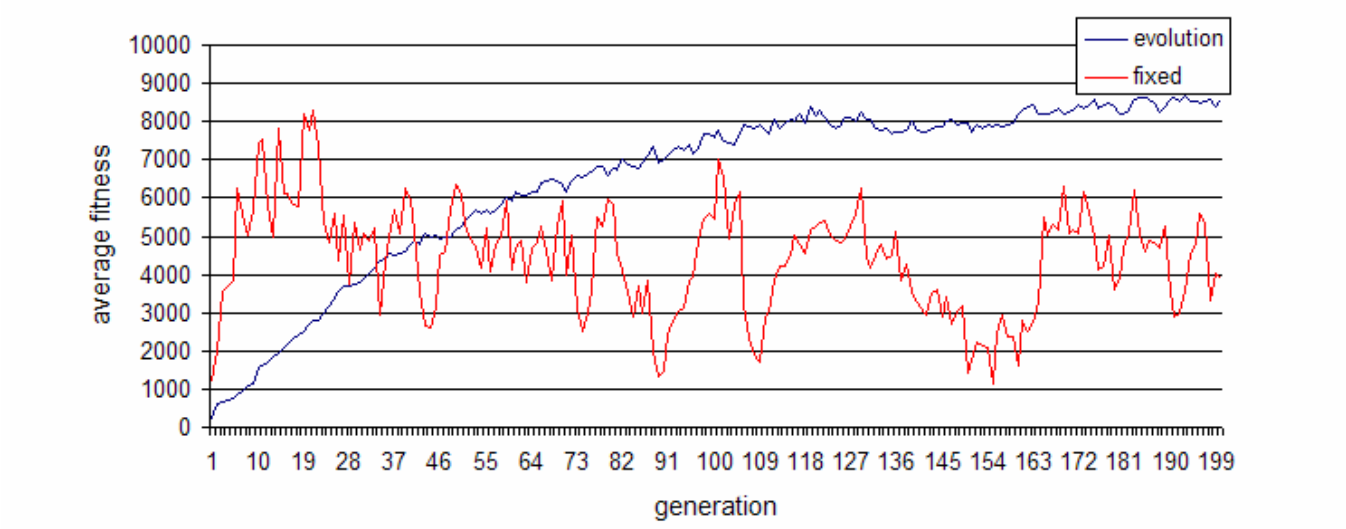

Figure 6 The average fitness of the population during 200 generations. The red line corresponds to the fitness during 200 runs without evolution, i.e. the use of fixed values. The blue line corresponds to the average fitness of 200 runs while evolving.

5) Crossover

From the half of the selected new population the other half is generated by crossover. So *n*/2 times two 'parents' are chosen for crossover.

The crossover is based on uniform crossover (Figure 2b). It works by copying the genes one by one. The crossover starts with the genes of parent 1, and per gene to be copied there is a chance (0.05) of going to parent 2. So for example gene 1 of parent 1 is copied, then genes 2-5 are copied from parent 2 and finally 6-10 are copied from parent 1. In this case the child has six genes from parent 1 and four from parent 2.

6) Mutation

For the mutation a value is added to every gene. This value is a random value with normal distribution $N(0, \beta(\max - \min)/2)$. Here β is a factor which we set to 0.2, because we do not want big changes of the values. Max and min are the upper and lower bound of the variable.

VI. Results

The evolution has been done several times with different configurations. When running all the individuals per generation a small population size (around 10 individuals) was used, this because it takes a lot of time per generation. We evolved for about 100 up to 200 generations. When only one randomly chosen individual was run per generation, the population size was set to about 100.

An important question is whether the evolution works or not. This can be answered by comparing the average population fitness of an evolution to the average fitness without evolution, but only the use of fixed values. The result of this comparison can be seen in Figure 6. Here clearly can be seen that the average fitness of the fixed values (red line) is changing a lot. In contrast the blue line is much smoother and is increasing until about generation 120. After which it slowly flattens.

The evolution from Figure 6 was done with 100 individuals, 200 generations and a tournament size of 50. This was about the best result we found.

Although the increasing line looks very promising, it must be noticed that it is an average fitness value of the population. Because all individuals start with a fitness of zero, the average after the first generation is very small (only one run per generation was done). Every new generation exists for the half of new individuals, which also have a fitness of zero and thus decreases the average fitness. The fitness of the best individual of the last generation of that particular evolution run was 22254.

The average values of seven evolution runs can be seen in Table I (last column), here can be seen that some evolved variables were quite close to the fixed values.

VII. Conclusion

From the experiments we can conclude that evolution can create better performing robot teams in comparison to the fixed variables. The best proof can be seen in Figure 6, as discussed in the previous section.

Although there were several runs which improved the robot, there were also quite some evolution runs with little improvement of the robot. This is probably caused by the parameter settings of the evolution. It could also be 'luck' when the evolution randomly initializes certain variables to a close optimum.

In future research as well the robot as the evolution system could be improved. Another selection mechanism could be tested. And the robot could be improved by for example a leader behaviour which orders other robots to go somewhere. Another discussion is the RoboCode game. This is an easy to use robot simulation platform, but it has some disadvantages. A big disadvantage is that is quite slow, and because evolution requires a lot of runs, it slows down the evolution process

enormously. Another disadvantage is that it contains several small bugs, but these can be fixed by editing the RoboCode sources, which are open-source.

REFERENCES

Arkin, R. C. (1998) *Behavior-based robotics*. The MIT Press, Cambridge Massachusetts.

Eisenstein (2003) Evolving Robocode Tank Fighters. *Technical Report AIM-2003-023*, AI Lab, Massachusetts Institute of Technology.

Frøkjær et al. (2004). *Robocode; Development of a Robocode team.* Student report of the University of Aalborg, Denmark.

Goldberg, D. E. and Deb, K. (1991) A comparative analysis of selection schemes used in genetic algorithms. *Foundations of Genetic Algorithms*, p. 69–93.

Legg, S., Hutter, M. and Kumar, A. (2004) Tournament versus Fitness Uniform Selection. *Technical Report IDSIA-04-04.*

Mitchell, T. M. (1997) *Machine Learning*. McGraw-Hill International Edition.

Pfeifer, R. and Scheier, C. (1999) *Understanding Intelligence*. The MIT Press, Cambridge Massachusetts.

Sing Li (2002) Rock 'em, sock 'em Robocode!, http://www-128.ibm.com/developerworks/java/library/j-robocode/

Sipper, M., Azaria, Y., Hauptman, A. and Shichel Y. (2005) Designing an evolutionary strategizing machine for game playing and beyond. *IEEE Transactions on Systems, Man, and Cybernetics, Part C: Applications and Reviews*.