Using Cyclic Genetic Algorithms to Evolve Multi-Loop Control Programs

Gary B. Parker
Computer Science
Connecticut College
New London, CT 06320, USA
parker@conncoll.edu

Ramona A. Georgescu
Electrical and Computer Engineering
Boston University
Boston, MA 02215, USA
rageo@bu.edu

***Abstract* - Cyclic genetic algorithms were developed to evolve single loop control programs for robots. These programs have been used for three levels of control: individual leg movement, gait generation, and area search path finding. In all of these applications the cyclic genetic algorithm learned the cycle of actuator activations that could be continually repeated to produce the desired behavior. Although very successful for these applications, it was not applicable to control problems that required different behaviors in response to sensor inputs. Control programs for this type of behavior require multiple loops with conditional statements to regulate the branching. In this paper, we present modifications to the standard cyclic genetic algorithm that allow it to learn multi-loop control programs that can react to sensor input.**

Index Terms - Evolutionary robotics, learning, control, genetic algorithms.

I. Introduction

Cyclic genetic algorithms (CGAs) [1] have been successfully used to evolve control programs for differing levels of robot control. They are capable of learning the sequence of instructions needed to produce a desired behavior. In addition, they can be used to learn a cycle of instructions to produce repeated behavior such as a gait cycle. This method is distinct from other evolutionary robotics approaches. Cyclic genetic algorithms are a means of generating code in the form of a single loop program. Although very successful in doing this and in generating controllers for individual leg movement, gait cycles, and learning the sequence of turns and straights to produce a good search pattern, they have been limited to control programs requiring only a single loop. This makes their use for learning control programs that process sensor input very limited. In order to process sensor input, the control program must have branching. Although the instructions in a single loop control program can be conditionals, without other possible loops, the result of sensor input can only be to execute one sequence of a selection of instructions. This limitation does not allow the robot to switch into another cyclic behavior in response to sensor input. What is needed is a means for cyclic genetic algorithms to generate multi-loop control programs with conditionals that allow the control to jump from one loop to another. In this paper, we address the task of learning obstacle avoidance while moving toward a light.

Mondada and Floreano developed Khepera, a miniature mobile robot, to study the evolution of control structures and had the robot perform, among other tasks, navigation and obstacle avoidance [2]. The controller consisted of an artificial neural network. Its weights were evolved using a combination of neural networks and standard genetic algorithms with fitness scaling and "biased mutations" [3]. Tuci, Quinn, and Harvey used a Khepera robot that was placed in an arena with the task to navigate towards/search for a target placed at one end of the arena [4]. No obstacle avoidance was implemented to help during the navigation; if the robot crashed into a wall, the trial was terminated. They used a neural network controller with fixed-connection weights and "leaky integrator" neurons, and a simple genetic algorithm for learning. At the National University of Singapore, controller evolution was studied using an incremental approach on a Khepera robot doing navigation and obstacle avoidance [5]. The goal was to test this incremental approach by first creating a neural controller for the mobile robot to perform straight navigation while avoiding obstacles and then later extend it to a wall following behavior.

Ram, Arkin, Boone, and Pearce applied genetic algorithms to the learning of robot navigation behaviors for reactive control systems [6]. The task to be performed was navigation of dynamic environments. Three schemas were implemented: move to goal, avoid static obstacle and noise. The parameters controlling the behavior of these schemas were determined autonomously using a GA. The GA was used to tune schema-based reactive control systems by learning parameter settings that optimized performance metrics of interest in various kinds of environments. Thus, the GA optimized reactive control by optimizing the individual reactive behaviors. Only simulation results were obtained.

For our research we used a LEGO Mindstorms robot. This was mainly because this portion of research was part of a larger research project that involved the co-evolution of the morphology and control of LEGO Mindstorms robots. Lund explored the concept of development of LEGO robot control systems without programming by children [7]. Neural networks were used as robot controllers for LEGO robots and an interactive GA was applied in combination with reinforcement learning, so that the development time could be reduced. Both simulation and real world tests were performed. The LEGO robots were equipped with light and IR sensors, motors, wheels, etc. After the child had seen the evolved behaviors of all the robots in the population, the child's preferred robots were chosen for reproduction. Mutation was applied to the selected robots. The loop continued until the child was satis-

fied with the evolved behavior of a robot. Obstacle avoidance was implemented.

In our approach, controller evolution is achieved with a multi-loop cyclic genetic algorithm. Training is done in simulation; tests are done both in a simulated environment and with the actual robot. The GA is not being used to learn weights for a neural network or parameters for a reactive control system. The controller is not executing a prewritten program using the learned values to guide the computation of an output from the input. The CGA is learning a control program that can be interpreted and directly executed on the controller.

II. CYCLIC GENETIC ALGORITHM

A *cyclic genetic algorithm* (CGA) [1] is much like a regular genetic algorithm [8] except that the gene groupings of the chromosome represent tasks to be completed as opposed to traits of the solution. These tasks can be anything from a single action to a sub-cycle of tasks. Using this method of representation, it is possible to break up a chromosome into multiple genes with each gene acting as a cycle. Each gene or sub-cycle contains two distinct sections, one part representing an action or set of actions, and the second part representing the number of times that action is to be repeated. The entire set of genes in the chromosome can also be executed repetitively, in which case the whole chromosome becomes a cycle.

Parker used CGAs to evolve single-loop programs for robotic control of individual leg cycles, gait cycles for hexapod robots, and area coverage patterns [9]. The CGA was well suited for these problems because the solutions are cyclic in nature and required a single loop for control. Problems that require dynamic changes in behavior depending on sensor input call for multi-loop control programs for which a system of conditional branching must be implemented in the CGA. Robotic control presents an interesting problem for learning algorithms since it usually requires sequential solutions where a series of actions is continually repeated. The Cyclic Genetic Algorithm (CGA) has proven to be an effective method for evolving single loop control programs such as the ones used for gait generation. The current limitation of the CGA is that it does not allow for conditional branching or a multi-loop program, which is required to integrate sensor input.

Parker, Parashkevov, Blumenthal, and Guldimann extended the use of CGAs to multi-loop programs that required sensor input [9]. The problem solved was the development of a search program for a predator robot to find a stationary prey. Their chromosome was 128 bits long and was designed for four different states, thus it had four segments, each of which represented a control loop, a cycle that the robot repeated as long as the sensors' inputs stayed the same. Each segment was linked to all of the other segments; there was one segment for each of the possible combinations of sensor inputs. Each segment consisted of four genes. The genes consisted of a pair of integers. The first integer of the gene determined which action was to be taken and the other dictated the number of repetitions of that action. After performing one action the specified number of repetitions, the robot checked the state of the sensors. If the sensor states were the same as the last time they were checked, the robot went on with the next gene in the same segment. If the last gene in the segment was reached, the cycle continued at the beginning again with the execution of the first gene in the segment. If the sensor inputs were different than the last ones, the robot halted the current cycle and jumped to the first gene of the segment that corresponded to the new sensor inputs. This worked well for the problem being solved, but it is not reasonable for the obstacle avoidance while moving toward a light problem. The drawbacks of this approach for this problem are: a segment was needed for every possible combination of sensor inputs and the multi-loop program didn't work with continuous sensor values.

In the work reported in this paper, we continued to expand the use of CGAs in evolving multi-loop programs by devising a new method to deal with a more complicated problem. The capabilities of the CGA were extended to evolve the program for a controller that incorporated sensors. As opposed to the research described above, for which the chromosome length grew exponentially with the number of sensors in the system, our implementation is more flexible: as many as desired sensors can be easily incorporated into the simulation by adding instructions to the system, while the total number of instructions depending on sensor input can remain constant.

In order for our robot to react properly to sensor input, the controller had to be running a multi-loop program, which is only possible if a system of conditional branching can be implemented. The gene structure of the CGA chromosome was modified so that the implementation of a system of conditional branching was possible. The evolved behavior enabled the robot to properly interpret sensor input to avoid walls and efficiently locate the desired stationary target (light source).

III. PROBLEM DESCRIPTION

The goal of the research reported in this paper was to evolve a multi-loop controller for a robot with sensors. The task chosen for investigation was navigation through an obstacle maze towards a light source.

A. The Robot and Colony Space

The robot, named Amsterdam, was constructed out of LEGO pieces. It was a combination / modification of the Roverbot with Single Bumper and Light Sensors [10] and the Bugbot [11]. The robot was assigned two tasks: navigation through an obstacle maze until reaching a light source and wall following. The RCX of Amsterdam, i.e. the programmable, microcontroller-based brick in the Lego Mindstorms Set, which can simultaneously operate three motors, three sensors, and an infrared serial communications interface, was programmed in NotQuiteC (NQC).

Amsterdam was equipped with two LEGO light sensors which could read light from 0.6 Lux through 760 Lux. Then, the RCX scaled this to a 0-100 percent measure. In our measurements, the source was considered to have a luminosity of 100%. The robot could see light coming directly from the source or light emitted by the source that had been reflected by the walls of the experiment area.

The actual experimental area set up in the lab for real world testing was an 8 x 8 foot area, having wooden walls, a powerful light source placed in the lower left corner and five obstacles whose placement depended on the configuration analyzed (see Section 4b). The 1 x 1 foot obstacles were placed between the robot and the light source. In order to be able to sense obstacles, Amsterdam was equipped with one bump sensor placed in the front.

B. The Simulation

The simulation occurred within a 300 x 300 (arbitrary units) area. All individuals started at position (285, 285) with an angle of 225, i.e. directly facing the light source. Also, this position had the advantages that the initial luminosities to the left and to the right were equal and the robot was not biased to move in a certain direction.

The experiment area was modeled as closely as possible in the simulation, special attention being paid to the light distribution over the experiment area. Each point in this area had been assigned values for luminosity in a way that would best mimic reality. Thus, the corners had been given fixed luminosity values. Along the walls, luminosity was assumed to decrease linearly with distance; when closer to the light, however, the change was less. For the inner points of the experiment area, the intersection point of the beam line coming from the left and right light sensors of the robot with each wall was computed. Depending on the angle of the beam, the wall the robot was facing could be determined. Therefore, the luminosity of the point of the beam projection on the wall could be computed. Then, the luminosity decreased linearly with the distance from the wall.

The obstacle locations were fixed throughout each test. The obstacles could have been randomly placed for the computation of each chromosome's fitness but then the comparison would have been inconsistent. Within a generation, one chromosome might have faced a configuration almost impossible to navigate through, while in another generation, the same chromosome might have been in a very easy to solve configuration. Three configurations were developed and used in the test runs; each for 5 tests making a total of 15 tests.

In the real world tests, Amsterdam was used, while in the simulation, the size of the robot was assumed to be a point. Each time the robot made a move, after the end of the move, the algorithm checked to see if the robot hadn't bump into an obstacle. If this was the case, the coordinates were adjusted to the point that the robot encountered the obstacle.

IV. THE EVOLUTION OF CONTROL

In order to use a CGA to learn the control programs the required NQC instructions were converted into machine code. A chromosome was developed that would have a sufficient number of loops possible and a sufficient number of instructions in each loop to solve the problem. A population of random individuals was created and involved for 350 generations using a simulation of the robot and its environment. The resultant multi-loop control programs were tested on the actual robot.

A. Machine Code

Using the NQC programming language a program was written to control Amsterdam in performing the task of navigating towards a light source while performing obstacle avoidance. This was done to identify all of the commands in NQC that were needed to perform the task. Some of the important commands needed for the operation of the robot were: OnFwd(OUT_X), S1<S3 and Wait(x). OUT_X stood for either A or C, indicating the left or right motor, respectively. OnFwd(OUT_X) turned on the specified motor and started rotating the axle of the motor counterclockwise, so that the robot started moving forward. S1<S3 asked for a comparison between the value of the light intensity measured by the left light sensor (S1) and the value of the light intensity measured by the right light sensor (S3). For example, if the left sensor, S1, measured a larger light intensity. The direction of the infrared beam of S1 would become the target line of movement of the robot. Thus, the robot would turn so that its symmetry line would overlap the direction of the beam of S1 at the time S1 took the measurement for the intensity of the light, and the robot would start advancing on this line. Wait(x) took as a parameter an integer, which represented, in hundredths of a second, the time during which the robot should keep executing the instructions currently on the queue. Section 3.3 describes in more detail in how the queue was created and executed.

In order to allow the CGA to generate code, the individual instructions needed to be represented in binary. Using backward engineering, we generated machine code for each of the possible instructions needed for the task. While creating this code, we fixed the maximum number of loops in our multi-loop program to eight.

The NQC instructions in the program were used in the design a machine code that assigned each possible instruction a 9 bit binary number. For example, the machine code equivalent of the instruction OnFwd(OUT_A), i.e. the left motor of the robot was turned on, resulting in the axle of the motor rotating counterclockwise and the robot moving forward, is 000000101.

Some instructions, if encountered, broke the execution of the current loop and started the execution of another loop in the same chromosome, the next loop to be executed being specified by the instruction. For example, if 001 010 000 was encountered, the program identified the first three bits (001) as the instruction S1<S3. Then, the intensity of the light measured by the left light sensor (S1) was compared to the intensity of the light measured by the right sensor (S3). If S1<S3 was true, the next loop to be executed was indicated by bits 4, 5 and 6 of the instruction, in this example, 010, which read, using binary, that the next loop to be executed was loop number 2. On the other hand, if S1<S3 was false, then the next gene to be executed was indicated by bits 7, 8 and 9 of the instruction, in our case, 000, which read, using binary, that the next loop to be executed was loop number 0.

Measurements of how a robot would move when given each instruction separately or in combination with other in-

structions were taken to increase the accuracy of the simulation. For example, if the robot was given the series of instructions: OnRev(OUT_A), OnFwd(OUT_C), Wait(45), the robot would move 0 cm in the x direction, 0 cm in the y direction and 45 degrees in a counter clockwise fashion. The wait time had a linear effect on the motion of the robot: if the robot was given the series of instructions: OnRev(OUT_A), OnFwd(OUT_C), Wait(90), the robot would move 0 cm in the x direction, 0 cm in the y direction and 90 degrees in a counter clockwise fashion. The coordinates of the robot were updated in the following manner: Current X = Previous X + ΔX.

B. Cyclic Genetic Algorithm Setup

A population of 64 chromosomes was used, each chromosome consisting of 7 genes, each gene consisting of a 2 bit number followed by six 9 bit numbers. The gene represented a "for" loop with the two bit number specifying how many times the loop should be executed; the possible values being 01 (once), 10 (twice), 11 (three times) and 00 (infinite). The six 9 bit numbers represented the instructions in the loop. These numbers were determined to be large enough for the problem, but not so large that the GA could not converge on a good solution. An example of a chromosome is given in Figure 1. The quotes indicate that we used a Scheme string format in order not to automatically erase the leading zeros.

```
(("11" "000000101" "000011000" "000000010" "100000110" "000000101" "000000110")
 ("10" "000000001" "000000101" "000000011" "000000111" "000011001" "000011111")
 ("00" "000111101" "000011111" "000011010" "000011001" "101000001" "000011010")
 ("11" "000000110" "000011111" "000011000" "101101101" "000000101" "000000011")
 ("10" "000011101" "000111011" "000011000" "101011011" "000000110" "001000010")
 ("00" "000111001" "000011010" "000000111" "000000010" "010000011" "000011000")
 ("01" "000011001" "001010100" "000000110" "000000011" "000011010" "000000111"))
```

Figure 1: Sample chromosome written in Scheme.

In the initial population, the 2 bit numbers in the beginning of each gene were randomly generated while the 9 bit numbers that followed were randomly picked by the computer with equal probability (1 in 19) from the pool of implemented instructions. For the instructions that stopped the execution of the current gene and started the execution of another gene specified in the instruction, the computer selected the fixed 3 or 6 bit part of the instruction with probability 1 in 19 and randomly generated the remaining number of bits.

Each test was run for 350 generations. The computation of the fitness of a chromosome is shown in Equation (1). The position (x, y) represents the final position of the robot. The light source location is at position (0, 0). The farthest possible point from the light source in the experimental area is (300, 300).

$$\text{fitness} = ((300-0)^2 + (300-0)^2) - ((x-0)^2 + (y-0)^2) \qquad (1)$$

The selection of two chromosomes for crossover was made in a roulette wheel fashion. Then a random index between 0 and the chromosome length was chosen and the resulting chromosome was made up of the [0, index] genes of the first chromosome and the (index, chromosome length] genes of the second chromosome.

After crossover, the new chromosome was subject to two types of mutations. The first type of mutation occurred more often, with the probability of 1 in 300 for each bit in the chromosome to be flipped. The second type of mutation occurred less often, with probability 1 in 5000 for each 9 bit number to be replaced with another 9 bit number randomly picked with equal probability (1 in 19) from the pool of implemented instructions.

The best chromosome from each generation was automatically included in the next generation. However, the same chromosome, when run a second time, would most probably not have the same fitness due to the randomness associated with the instruction Wait (random (50)).

After the fitness of all chromosomes in a generation was evaluated, the best chromosome was identified and printed to file. In addition, the trajectory of its movement was recorded so that it could be displayed in a plot made with Matlab 6.5.

C. Fitness Evaluation in Simulation

The algorithm took a chromosome as input to evaluate its fitness. The first gene was analyzed, i.e. the algorithm determined how many times the gene should be executed (once, twice, three times or infinitely many times) and read its six 9 bit numbers to an input queue. See Figure 2 for an example.

Then, the algorithm searched in the input queue for the first occurrence of one of these four types of instructions: a Wait instruction, a touch sensor instruction, a light sensor instruction, and a jump to another gene instruction. These are the types of instruction that would result in the robot moving. In the following discussion, this instruction will be referred to as the main instruction. If the gene had no main instruction, for example ("01" OnFwd OnFwd OnRev null Off null), the robot wouldn't move from its initial position.

```
- gene = ("01"  "000000110"  "000000011"  "001010100"
                "000011010" "000011001" "000000000")
- "01" = the gene will be executed once
- input queue =   OnFwd (OUT_C)
          Wait (50)
          If (S1<S3)
               Start executing gene 010 (gene 2)
                        Else Start executing gene 100 (gene 4)
                    OnRev (OUT_C)
                    OnRev (OUT_A)
               Null
- main instruction = Wait(50)
- partial queue = OnFwd (OUT_C)
- new queue =     If (S1<S3)
               Start executing gene 010 (gene 2)
                        Else Start executing gene 100 (gene 4)
                    OnRev (OUT_C)
                    OnRev (OUT_A)
                    Null, i.e. do nothing
```

Figure 2: An example of how the instructions in a gene are executed.

After the main instruction had been identified, the input queue was split into two queues: the queue consisting of all instructions given prior to the main instruction which was called the "partial queue" and the queue consisting of all in-

structions given after the main instruction which was called the “new queue”.

The instructions in the partial queue and the main instruction were executed in the order they had been added to the input queue and afterwards the process continued with the new queue as the input queue.

The algorithm executed a chromosome in the following manner. It started searching for a main instruction in the first gene (repeating the search if the first number of the gene, which indicates how many times to repeat the “for” loop, was something other than 01). If it didn’t find it, the algorithm went to the second gene, etc, until it found a main instruction. It then executed all of the instructions in the partial queue as explained earlier in this section. As the algorithm finished executing each gene, it went on to the next gene in the chromosome, unless a jump in the gene sent the point of execution to another gene. This was continued until the whole chromosome had been executed, at which time the program would halt. The algorithm was capable of identifying consecutive Wait commands. For example, if the following sequence was encountered: OnFwd(OUT_A), Wait(50), Wait(50), the instruction OnFwd(OUT_A) would be run for a total time of 100. The value of the Wait time was added to a timer that expired at 25000. This timer was needed so that the program would stop when executing an infinite loop.

D. *Testing*

Five tests were performed for each of three obstacle configurations; a total of 15 tests were performed. In two of the configurations, the obstacles were placed with sufficient distance from each other so that the robot could penetrate through the maze, and in the other configuration, the obstacles were placed next to each other so that the robot was forced to turn and find a way around them.

The three obstacle configurations were used for training and testing done in simulation and testing with the actual robot navigating inside the experimental area. The time it took Amsterdam to find the light (execution time) was recorded for each test. The trajectory of the robot in the simulated tests was also recorded. In the real robot tests, the trajectory was observed and sketched to compare it with the simulated track.

V. RESULTS: THE SIMULATION TESTS

Three configurations for the placement and number of the obstacles were used for the tests in order to see how dependent the performance of the algorithm was on the placement of the obstacles. In the end, no evidence of the performance being dependent on the placement or the number of the obstacles was found.

When the obstacles were placed in the experiment area at a distance from each other that would allow the robot to free itself from them rather easily by navigating through them, the CGA produced some robots that went straight toward the light until they reached it at (0, 0) and then wandered in the proximity of the light source. This wandering was the reason why the robot might not have been assigned the maximum fitness 179776 even though it had reached the (0, 0) position at some point in its evolution. Its fitness however would be over 178000 when it wandered in the area close to the light source. The robot never left the area close to the light source once it reached the light source.

When obstacles were next to each other in the experiment area and the robot was forced to go around them, the robot had the tendency to avoid them, run into a wall and then do wall following until the light source was reached. This is explained by the way the luminosity was assigned to each of the points in the experiment area. The luminosity decreased linearly with the distance from the wall and it also decreased linearly along the wall.

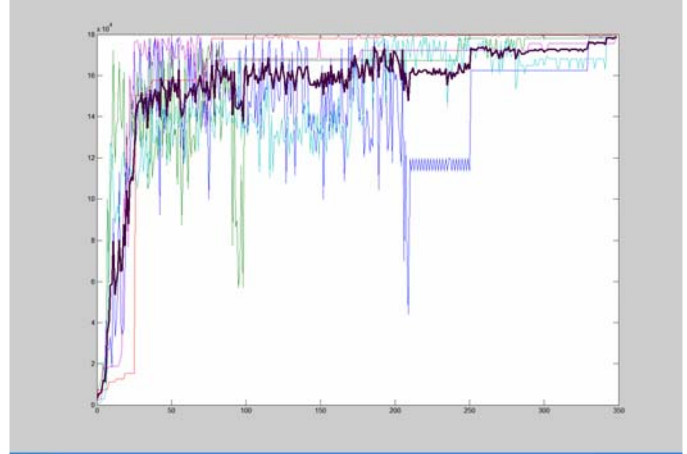

Figure 3: Fitness evolution for the five tests performed on configuration 3 using population sizes of 64 individuals and 350 generations. The x axis (0 to 350) shows the number of generations and the y axis (0 to 18000) shows the best fitness at each generation. The average of the best fitnesses is in bold.

Five tests were made for each of the three obstacle configurations and Figure 3 displays the fitnesses of the best chromosome in the population over the 350 generations for each of the 5 tests performed with configuration 3. For the tests with configurations 1 and 2, similar growth curves were obtained. In all tests, the best chromosome passed through the light source position, i.e. (0, 0).

VI. RESULTS: ACTUAL ROBOT TESTS

The chromosome with the highest fitness at generation 350 obtained in the test runs made in the simulation was translated from the machine code chromosome into NQC code and used for tests made with the real robot. Five tests using the same controller were performed for each of the three obstacle configurations. The time it took the robot to find the light (execution time) was recorded for each (Table 1).

Table 1: Execution times for the tests performed with the real robot.

	Test 1	Test 2	Test 3	Test 4	Test 5
Configuration 1	3min 20sec	2min 40sec	1min 10sec	1min	1min 15sec
Configuration 2	2min 30sec	2min 45sec	1min 45sec	2min 40sec	2min 20sec
Configuration 3	1min	2min	3min	2min 15sec	2min 20sec

Configuration 1 and configuration 3 were very similar regarding the performance of the robot. The only difference between these two configurations was that the obstacles that were closest to the upper and right walls were closer to these walls for configuration 3 then for configuration 1. This difference was responsible for the higher average execution time of the tests made with configuration 3 then the average execution time of the tests made with configuration 1.

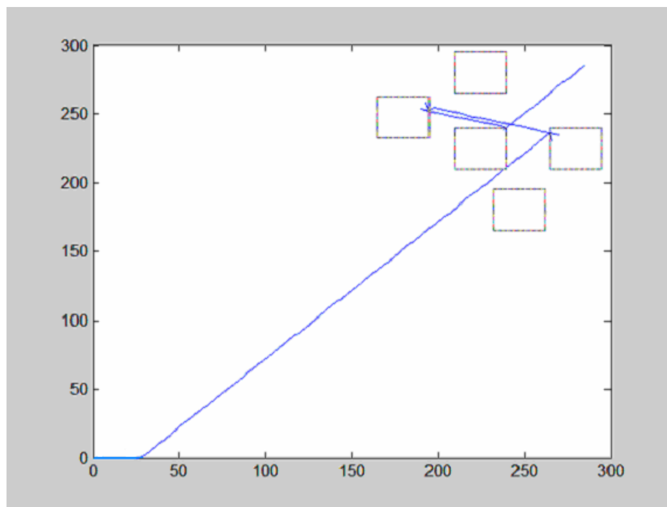

Figure 4: A typical track of the real robot in tests where it finds a way through the set of obstacles.

In tests 3, 4 and 5 with configuration 1 and also in test 1 with configuration 3, the robot bumped into the central obstacle and then went straight for the light (Figure 4), thus the shorter execution times (~1 min) compared to the other execution times for the tests made with configurations 1 and 3.

For configuration 2, the robot could not move through the obstacles so it typically made one attend and then did wall following until it reached the goal. The actual robot moved clockwise three tests and counter clockwise in two tests.

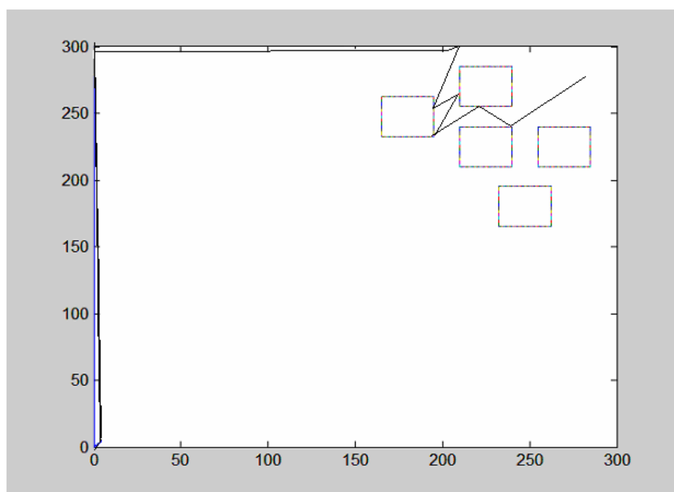

Figure 5: A typical track of the real robot doing wall following after initially attempting to find a way through the obstacles.

A typical track followed by the real robot when it did wall following is shown in Figure 5. This figure is a drawing of the trajectory the real robot followed during the actual test. In approximately half the tests, the robots moved clockwise and in the other half counter clockwise in the case where they did not take a more direct route.

VII. CONCLUSIONS

In this research, we successfully evolved multi-loop control programs for robots with fixed morphology using a cyclic genetic algorithm. The only a priori knowledge that went into the learning system was to limit the machine code instructions to those that were pertinent to the robot configuration and had a possible contribution to the solution and to make judgements on the maximum number of loops that would be required and the maximum number of instructions needed in each. Apart from these decisions in the setup, the learning system generated the needed code that, after interpretation, was directly executed on the controller. In the 15 test runs made with three different obstacle configurations, the robot always reached its goal, i.e. it successfully navigated through an obstacle maze in its search for the light source and after reaching the light source it stayed in its proximity.

In future work, we will continue to develop the multi-loop capabilities of CGAs by comparing them to other learning methods and using them to evolve programs for other applications requiring more than one loop.

REFERENCES

[1] G. Parker and G. Rawlins, "Cyclic Genetic Algorithms for the Locomotion of Hexapod Robots," P*roceedings of the World Automation Congress (WAC '96), Volume 3, Robotic and Manufacturing Systems,* 1996.

[2] F. Mondada and D. Floreano, "Evolution of Neural Control Structures: Some Experiments on Mobile Robots," *Robotics and Autonomous Systems*, 16, 183-195, 1995.

[3] D. Goldberg, *Genetic Algorithms in Search, Optimization and Machine Learning*. Addison-Wesley, Reading, MA, 1989.

[4] E. Tuci, M. Quinn, and I. Harvey, "Evolving Fixed-Weight Networks for Learning Robots," *Proceedings of Congress on Evolutionary Computation (CEC2002)*, 2002.

[5] D. Bajaj and M. Ang, "An Incremental Approach in Evolving Robot Behavior," *Proceedings of the Sixth International Conference on Control, Automation, Robotics and Vision*, 2000.

[6] A. Ram, R. Arkin, G. Boone, and M. Pearce, "Using Genetic Algorithms to Learn Reactive Control Parameters for Autonomous Robotic Navigation," *Adaptive Behavior*, vol. 2, issue 3, 1994.

[7] H. Lund, O. Miglino, L. Pagliarini, A. Billard, and A. Ijspeert, "Evolutionary Robotics - A Children's Game," *Proceedings of IEEE 5th International Conference on Evolutionary Computation*, 1998.

[8] J. Holland, *Adaptation in Natural and Artificial Systems*. Ann Arbor, MI, The University of Michigan Press, 1975.

[9] G. Parker, I. Parashkevov, H. Blumenthal, and T. Guildman, "Cyclic Genetic Algorithms for Evolving Multi-Loop Control Programs," *Proceedings of the 2004 World Automation Congress*, 2004.

[10] Robotics Invention Systems 2.0 Constructopedia. LEGO Mindstorms, 2000.

[11] D. Baum, *Definitive Guide to LEGO MINDSTORMS*. Apress, Berkeley, CA. (2000).