Evolution of Multi-Loop Controllers for Fixed Morphology with a Cyclic Genetic Algorithm

Gary Parker
Computer Science Department,
Connecticut College,
New London, CT 06320, USA
1 860 439 5208
parker@conncoll.edu

Ramona Georgescu
Electrical Engineering Department,
Boston University,
Boston, MA 02215, USA
1 617 353 5262
rageo@conncoll.edu

ABSTRACT
Cyclic genetic algorithms can be used to generate single loop control programs for robots. While successful in generating controllers for individual leg movement, gait generation, and area search path finding, cyclic genetic algorithms have had limited use when dealing with control problems that require different behaviors in response to sensor inputs. For such behaviors, there is a need for modifications that will allow the generation of multi-loop control programs, which can properly react to sensor input. In this work, we present modifications to the standard cyclic genetic algorithm that enables it to learn multi-loop control programs with branching that allows the control to jump from one loop to another. Preliminary tests show the success of our modification.

Categories and Subject Descriptors
I.2.9 [**Artificial Intelligence**]: Robotics – *autonomous vehicles.*

General Terms
Algorithms, experimentation.

Keywords
Evolutionary robotics, learning, control, code generation

1. INTRODUCTION
The cyclic genetic algorithm (CGA) was developed to learn single-loop control programs for robot locomotion [1] and has been successfully applied at three levels of control. However, its use for learning control programs that process sensor input has been 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 section 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 work, we provide a solution as we address the task of learning obstacle avoidance while moving toward a light.

GECCO'05, June 25-29, 2005, Washington, DC, USA.
ACM 1-59593-010-8/05/0006.

The robot (Figure 1) was constructed out of LEGO pieces. It was a combination / modification of the Roverbot with Single Bumper and Light Sensors [2] and the Bugbot [3]. The robot was equipped with light and bump sensors. Its RCX, i.e. the programmable, microcontroller-based brick in the LEGO Mindstorms Set, was programmed in NotQuiteC (NQC) [3]. To make a simulation for evolving the controller, the NQC instructions were converted into machine code. A chromosome was developed such that it would have a sufficient number of loops possible and a sufficient number of machine code instructions in each loop for the CGA to be able to solve the problem. A lab area set aside for the experiment was modeled to be as accurate as reasonable in simulation, special attention being paid to the light distribution over the experiment area. The locations of the obstacles placed within the experiment area were fixed throughout five separate tests performed in simulation.

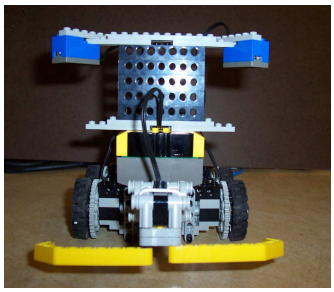

Figure 1. Robot with two light sensors and a bump sensor.

2. MULTI-LOOP CGA
A population of 64 chromosomes was used, each chromosome (Figure 2) 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 machine code instructions within the loop.

```
(("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 2: Sample chromosome written in Scheme.

Execution of the chromosome was done in simulation. 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. 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, or a jump to another gene instruction. These are the types of instructions that would result in the robot moving. In the following discussion, this instruction will be referred to as the main instruction.

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 instructions 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 the algorithm finished executing each gene, it went on to the next gene in the chromosome, unless a jump instruction in the gene sent the point of execution to another gene. This process was continued until the whole chromosome had been executed, at which time the program would halt.

The initial population was randomly generated. Each test was run for 350 generations. The computation of the fitness of a chromosome was based on the position of the robot at the end of the execution of the chromosome. The best individual was passed on to the next generation; the remaining 63 individuals were generated using roulette wheel selection, crossover, and two types of mutation. The best chromosome from each generation was saved to a file for subsequent evaluation. In addition, the trajectory of its movement was recorded so that it could be displayed.

3. RESULTS AND DISCUSSION

Five tests were run using different starting populations. In all 5 tests, the robot reached its goal, i.e. it successfully navigated through a set of obstacles in its search for the light source and after reaching the light source it stayed in its proximity. Figure 3 displays the evolution of the 5 fitnesses of the 5 tests performed on the static obstacle configuration.

The success of these tests leads us to believe that we have developed a plausible method for evolving multi-loop control programs using a cyclic genetic algorithm. The CGA had at its disposal of a set of primitive instructions, including conditionals, that could be interpreted and downloaded to the robot for execution. There was very little *a priori* knowledge needed for the code generation. The only limitations were to the instructions made available for evolution and to the maximums set for the number of loops in the program and the number of instructions in each loop. In further work, we are testing or implementation on additional obstacle configurations and confirming the results on the actual robot.

REFERENCES

[1] Parker, G., and Rawlins, G. *Cyclic Genetic Algorithms for the Locomotion of Hexapod Robots*. Proceedings of the World Automation Congress, Volume 3, Robotic and Manufacturing Systems, 1996.

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

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

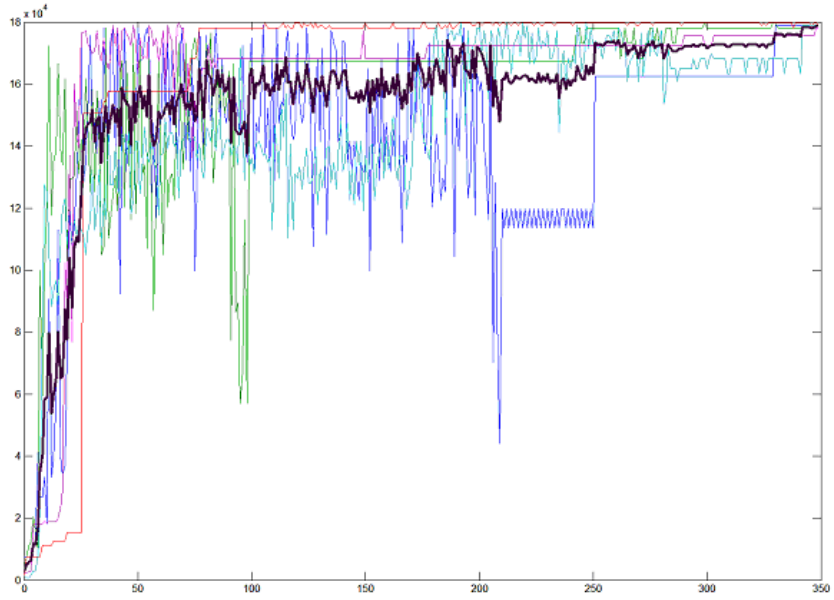

Figure 3: Fitness evolution for 5 tests performed on a static obstacle configuration. The x axis shows the number of generations and the y axis shows the best fitness at each generation. The average of the best fitnesses is in bold.