Evolution of Robot Controller Using Cartesian Genetic Programming

Simon Harding and Julian F. Miller

Department Of Electronics
University of York, UK YO10 5DD
{slh, jfm}@evolutioninmaterio.com

Abstract. Cartesian Genetic Programming is a graph based representation that has many benefits over traditional tree based methods, including bloat free evolution and faster evolution through neutral search. Here, an integer based version of the representation is applied to a traditional problem in the field: evolving an obstacle avoiding robot controller. The technique is used to rapidly evolve controllers that work in a complex environment and with a challenging robot design. The generalisation of the robot controllers in different environments is also demonstrated. A novel fitness function based on chemical gradients is presented as a means of improving evolvability in such tasks.

1 Introduction

Cartesian Genetic Programming is a graph based representation that has many benefits over traditional tree based methods, including bloat free evolution and faster evolution through neutral search. In this paper we apply this representation to a robot control task. In this section the representation is discussed - including a comprehensive survey of existing work on Cartesian Genetic Programming(CGP) and we look at previous work in evolving robot controllers using genetic programming. We describe the benefits of this representation over traditional techniques. The algorithm and novel fitness function are discussed in section 2. In section 3 we apply apply CGP to various robot tasks and demonstrate generality in evolved solutions.

1.1 Cartesian Genetic Programming

Cartesian Genetic Programming [13] is a graph based form of Genetic Programming that was developed from a representation for evolving digital circuits [7, 8]. In essence, it is characterized by its encoding of a graph as a string of integers that represent the functions and connections between graph nodes, and program inputs and outputs. This gives it great generality so that it can represent neural networks, programs, circuits, and many other computational structures. Although, in general it is capable of representing directed multigraphs, it has so far only been used to represent directed acyclic graphs. It has a number of features that are distinctive compared with other forms of Genetic Programming.

Foremost among these is that the genotype can encode a non-connected graph (one in which it is not possible to walk between all pairs of nodes by following directed links). This means that it uses a many-to-one genotype-phenotype mapping to produce the graph (or program) that is evaluated. The genetic material that is not utilised in the phenotype is analogous to junk DNA. As we will see, mutations will allow the activation of this redundant code or de-activation of it. Another feature is the ease with which it is able to handle problems involving multiple outputs. Graphs are attractive representations for programs as they are more compact than the more usual tree representation since subgraphs can be used more than once.

CGP has been applied to a growing number of domains and problems: digital circuit design [11, 12], digital filter design [8], image processing [21], artificial life [20], bio-inspired developmental models [9, 14, 10], evolutionary art [1], molecular docking [4] and has been adopted within new evolutionary techniques cell-based Optimization [19] and Social Programming [24]. In addition a more powerful form of CGP with the equivalent of Automatically Defined Functions is also being developed [25].

Figure 1 shows the general form of Cartesian Program for an n input m-output function. There are three user-defined parameters: number of rows (r), number of columns (c) and levels-back (which defined how many columns back a node in a particular column can connect to). Each node has a set of C_i connection genes (according to the arity of the function) and a function gene f_i which defines the nodes's function from a look-up table of available functions. On the far left are seen the program inputs or terminals and on the far right the program output connections O_i

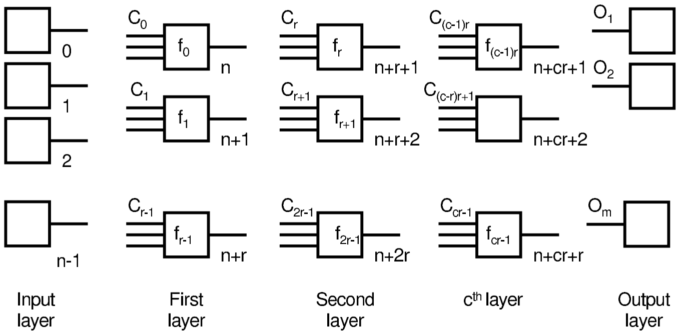

Fig. 1. General form of Cartesian Program

If the graphs encoded by the Cartesian genotype are directed (as in this work) then the range of allowed alleles for C_i are restricted so that nodes can only have their inputs connected to either program inputs or nodes from a previous (left) column. Function values are chosen from the set of available functions. Point mutation consists of choosing genes at random and altering the allele to another value provided it conforms to the above restrictions. Although the use of crossover is not ruled out, most implementations of CGP (including this one) only use point mutation.

We emphasize that there is no requirement in CGP that all nodes defined in the genotype are actually used (i.e. have their output used in the path from program output to input). Although a particular genotype may have a number of such redundant nodes they cannot be regarded as non-coding genes, since mutation may alter genes "downstream" of their position that causes them to be activated and code for something in the phenotype, similarly, formerly active genes can be deactivated by mutation. We refer genotypes with the same fitness as being neutral with respect to each other. A number of studies (mainly on Boolean problems) have shown that the constant genetic change that happens while the best population fitness remains fixed is very advantageous for search ([13, 23, 26]).

1.2 Obstacle Avoiding Robots

A common test problem in genetic programming involves generating a program that can control a robot. Typically, the task is to travel around a closed environment avoiding the walls and any obstacles [6, 2, 15, 3, 16, 17, 22, 18]. The task may be extended to getting the robot to cover as much floor space as possible or to follow the wall. In this scenario the robot has to navigate around an unknown environment avoiding contact with the walls. The control system is able to use the distance sensors on the robot, perform some form of signal processing and in turn control the motion of the robot. Two common robotic platforms are the Khepera miniature robot or a gantry style robot, such as those from iRobot. Both of these types of robots have an array of distance sensors. The Khepera has 8 short range infra red sensors, the gantry robot because of its larger size can accommodate 24 sonar sensors. Generally evolution is performed in simulation. Solutions based on genetic programming and neural network architectures can be run in faster than real time in simulation, as they can ignore (to a degree) the physical properties of the robot and its hardware. A simple obstacle avoiding robot was evolved by Koza in [5]. In this task, a robot had to move around an 8 by 8 grid and avoid obstacles - in order to mop the floor of a room. The obstacles were cells in the grid that the robot was not allowed to enter. The robot is able to move forward one cell, leap forward several cells, turn left, check to see if an obstacle is directly in front and addition (with modulo 8 integer arithmetic). The fitness was calculated as the number of squares visited in a fixed period of time. Without the use of automatically defined functions (ADFs), it took fewer than 50 generations to evolve a successful behaviour, with ADFs this result was lowered to 29 generations. However, Koza uses relatively large populations - in this example each population contained 500 individuals. The function set available to the system was quite basic. Lazarus and Hu also used a grid based system to evolve robot controllers. In [6] Lazarus and Hu evolved wall following robots, that could find, then move around the edge of a room. The room was a square room, with a number of extrusions in the walls. The most complex map had four extrusions - where each wall was interrupted by a section projecting into the centre of the room. The program could move the robot to adjoining cells and sense the state of those cells. To process the sensor information, the function set included IF, AND, OR and NOT. Using a population of size 1000, solutions for the most complex room were evolved in 90 generations. In [16] Reynolds uses genetic programming (GP) to evolve a program that can take visual information and control a simulated robot that avoids obstacles. The input to the program was a simple sensor that measures how strongly an object is seen in a particular direction. This information is then processed using basic mathematical operations and conditionals. The program output allowed the robot to drive forward and turn. In this example, the robot was simulated, however, in [15] this style of control was used to control a real robot. The input terminals were readings from the distance sensors of the Khepera robot. In [3], Ebner evolves a controller for a gantry style robot using genetic programming. The distance readings from 24 sensors were mapped into 6 virtual sensors and these were used as terminals in the GP program. Output terminals for movement are restricted to go forward, stop and turn. To allow for the evolution of a hierarchical control mechanism the program could make use of conditional statements, which allows for greater program complexity. The task was to navigate around a short stretch of a straight corridor without colliding with the walls. With the robot running in simulation, evolution was performed for 50 generations on a population of 75 individuals. When the process was moved into a physical robot, evolution took a long time (197 hours) and produced similar results to the simulated work. In [15], Nordin and Banzhaf apply genetic programming to evolving robot controllers in a real environment. The program was encoded as a binary string and evolved using a standard genetic algorithm The function set comprised of ADD, SUB, MUL, SHL, SHR, AND, OR, XOR and integer terminals. A population size of 30 was used, and successful individuals were found within 200 to 300 generations. A pleasure-pain fitness function was used. The robot received pleasure from going straight and fast, and pain from coming close to the obstacles. The scores from this pleasure/pain reward are then weighted and summed to produce an overall fitness. In this work, we present a robot, with fine grained control, several environments of greater complexity to those described above. We show that CGP is suitable for controlling the robot, and that the results are competitive to previous techniques.

2 Algorithm

2.1 The Robot

The simulated robot used in this set of experiments is similar to a Kephera robot. There are two wheels driven by motors, these motors can turn in either direction and are variable speed. Driving both motors in the same direction (and speed) moves the robot forward or backward in a straight line. By using different speeds the robot can be made to turn, turning motors in opposite directions increases the turning speed. The robot is equipped with two distance sensors mounted on the front, which are separated by an angle of 20 degrees. The sensors return the distance to the nearest wall in the direction they are pointed. There is no grid representation used in this simulation (other than for the fitness function), and the robot can move anywhere to within the resolution of double precision floating point numbers. The robot's orientation is also defined by a real number.

2.2 Function Set

For these experiments we use an integer version of CGP, in which the nodes operate on signed integer values and output signed integer values. The levels back parameter has been set for the whole width of the graph, and multiple rows are used. The nodes in the CGP graph can use the following functions: Add, Subtract, Multiply, Divide, Compare, Min, Max, Fixed integer and Input node. Add, subtract, multiply and divide are all two input nodes that perform integer arithmetic. The divide function performs integer division and is safe: dividing by 0 returns 0. Min and max respectively output the minimum and maximum of the two inputs to that node. Compare returns -1 if the first input is less than the second, 0 if the inputs are equal and +1 if the first input is larger than the second. Some nodes can store a fixed integer in the range -100 to +100. These are terminal nodes, i.e. nodes with no inputs. The first column in the CGP graph is made of input nodes. The speed used to drive the motors was taken from 2 nodes on the last column in the graph. The integer value for each node was found, truncated to fall between -100 and 100 and then scaled to a value between -1 and +1.

2.3 Operators, Parameters and Fitness Function

The evolutionary algorithm used for these experiments is very simple. The population size was set to 40 individuals. Elitism was used, with the best 5 individuals retained for the next generation. Tournament selection was used, with a tournament size of 5. No crossover was used when generating subsequent populations. Mutation was set to 5 percent of the node count, with entire nodes being mutated in each operation. Evolutionary runs were limited to 1000 generations, with each run being aborted when a solution was found i.e. when an individual's fitness was greater than 9995. For these experiments, the CGP graph was set to 20 nodes wide by 2 nodes tall. The first column of the graph is used for input nodes, leaving 38 nodes to perform processing. Typically for obstacle avoiding robots, fitness values are computed based on factors such as time spent moving forward, total path length and Euclidian distance travelled. For example, Thompson[22] uses the following calculation:

$$fitness = \frac{1}{T}\int_0^T \left(e^{-k_x c_x(t)^2} + e^{-k_y c_y(t)^2} - s(t)\right) \ where\ s(t) =^{1\ when\ stationary}_{0\ otherwise}$$

where the distance of the robot from the centre of the room in the x and y directions at time t was c_x and c_y, for an evaluation for T seconds. However, during initial experiments it appeared that this method for calculating fitness had many drawbacks including local minima which resulted in poor evolutionary characteristics (e.g. mean fitness did not increase smoothly). In environments with obstacles this style of fitness function fails to capture the difficulty of getting to hard to reach locations in the map. To address this a fitness map of the environment was calculated, where each area in the map had an absolute measure of the difficulty in reaching that point. The fitness for the robot was calculated as the highest fitness measure seen during the robot's movement around the map. The calculation for this fitness map was performed by modeling chemical diffusion within the environment. If we imagine the room to be filled with a fluid such as water, and then add some coloured ink in a particular location, the colour would diffuse through the water. Near the point where the dye was added, the concentration would be greatest, the further away from the source the lower the concentration. The actual concentration (after a period of time) at a point is related to the shortest possible path to the point where the dye was added. Using a model of diffusion, we can easily approximate the shortest path required to reach any point in the map. This model automatically takes into account any shape of environment and the obstacles within it. By adding a "dye" at the starting position of the robot, and allowing it to diffuse until the chemical level at all points in the map is above 0 percent, an absolute fitness can be calculated for the map.

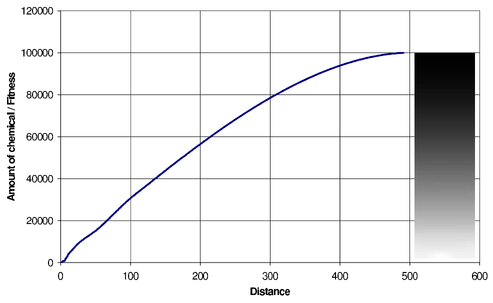

Fig. 2. Plot of fitness against distance from start position. The gradient on the right can be used a key for figures 4 and 6

The diffusion algorithm works by breaking the map area into a grid (in this experiment of size 500 x 500) which stores the amount of chemical in each part of the map, initially all cells are set to 0. The diffusion is calculated using a simple cellular automata style technique. For each cell, if the average amount of chemical in its eight neighbouring cells is greater than the amount in that cell then the amount of chemical in the cell is increased. When all the cells contain some chemical, the map is normalised so that the fitness score falls between 0 and 10000, with 10000 being the maximum fitness. Figure 2 shows the relationship between fitness and distance from the starting position. The gray-scale gradient on the right shows the colour corresponding to the fitness value, and is used in the fitness maps throughout this paper. Figures 4 and 6 show the fitness maps for the two environments used. The darker colours show the areas of greatest difficulty to reach from the robots starting position in the top left corner. Robots that can successfully navigate around the obstacles, and explore large amounts of the map will pass through areas marked as having high fitness. Solutions where the robot does not move far or travels in a circle will obtain low fitness. In these experiments the robots were allowed to travel until they collided with a wall, or a timeout situation occurs i.e. the simulation has been updated 10000 times.

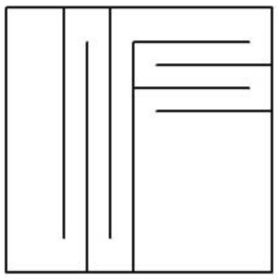

Fig. 3. Map 1

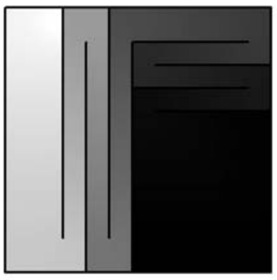

Fig. 4. Fitness values for map 1

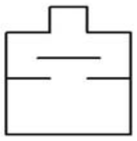

Fig. 5. Escape Map

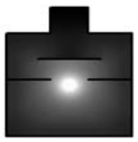

Fig. 6. Fitness values for the escape map

3 Experiments

3.1 Escaping a Room

The first problem involves the robot escaping from a small room. Figure 5 shows the layout of the room, with the robot starting in the centre of the map. It should be noted that all the map images are drawn to the same scale.

Results. 140 runs of the algorithm were performed, with 81% of evolutionary runs providing a solution within 1000 generations. We found the average number of evaluations required for a solution to be 8515, however the standard deviation was high (8091) - the minimum time to discover a solution was 8 generations. On reviewing some of the paths taken by the robot, it was seen that some became stuck in the bottom part of the maze. As the population converged, it would have become harder to escape this local optimum. With the neutrality in CGP, the represenation is more robust to this type of situation - even in converged populations, small mutations can produce large changes in phenotypic behaviour.

In this problem scenario, it is expected that there are many local optima, and with the high success rate it is clear that CGP is capable of escaping these.

The following is a sample program for a good solution. Motor1 and motor2 are the motor speeds of the left and right motors. INPUT_0 and INPUT_1 are the two distance sensor readings. The remaining functionality is described previously. An interesting observation from this is the reuse of the node -25. A non-graph based representation would not easily allow for this and would have to replicate the node. But here GCP can reuse nodes, and share subgraphs that are integral to the behaviour of both motor controllers. In the motor1 control program, it appears evolution has used this node to manufacture a 0 rather than evolve a integer node with a value of 0.

```
motor1= ADD(MINUS(-25 , -25), MULT(ADD(INPUT_0, -25),
        MIN(INPUT_1, INPUT_0)))
motor2=ADD(MINUS(ADD(INPUT_0, MAX(MIN(MIN(INPUT_1, INPUT_0),
 MIN(INPUT_1, INPUT_0)), -25 ), 18), MINUS(MIN(INPUT_1, INPUT_0),
 MIN(ADD(MINUS(INPUT_0, INPUT_0), MULT(MAX(MIN(MIN(INPUT_1, INPUT_0),
 MIN(INPUT_1, INPUT_0)), -25), MAX(MIN(MIN(INPUT_1, INPUT_0),
 MIN(INPUT_1, INPUT_0)), -25))), ADD(INPUT_0, -25))))
```

If we take the above programs and turn them into a more human readable form, we find the following rules have been evolved.

```
IF INPUT_0 <= INPUT_1 THEN
      motor1 := INPUT_0 ( INPUT_0 - 25 )
   ELSE
      motor1 := INPUT_1 ( INPUT_0 - 25 )
ENDIF

IF INPUT_0 <= INPUT_1 THEN
   motor2 := (2 * INPUT_0) - 43
ELSE
   IF (INPUT_1 ^ 2) <= (INPUT_0 - 25) THEN
      motor2 := INPUT_0 + (2 * INPUT_1) - (INPUT_1 ^ 2) - 18
   ELSE
      motor2 := (2 * INPUT_1) - 43
   ENDIF
ENDIF
```

The program for motor1 is very simple. If INPUT_0 detects a wall, then the speed of the motor is negative - and the robot will start to turn. Otherwise the motor is on in a forward direction. For motor2 the program is slightly more complex. However, it still has the same basic functionality - the motor speed is dependent on the value of sensor that is nearest the wall. However, motor2 appears to have it's speed regulated to ensure that in general it is going forward, and to slow down when INPUT_1 detects a wall. This code demonstrates a sophistication beyond the binary control of Braitenberg type vehicles - in this result the robot speeds up and slows down depending on the current state of its sensors.

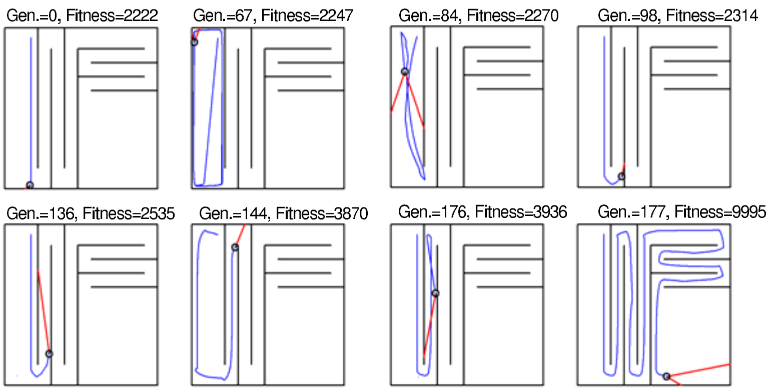

Fig. 7. Evolutionary history of maze solving behaviour

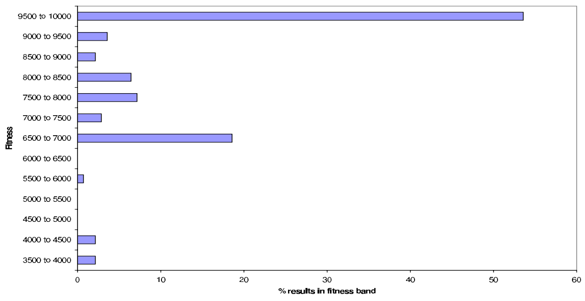

Fig. 8. Distribution of peak fitness scores during evolution, for the initial maze problem (figure 3)

3.2 Solving a Maze

The challenge in this scenario is for robot to solve a complicated maze. The maze, shown in figure 3 has many tight u-bends which are placed at different orientations. Figure 4 shows the fitness scores throughout the map.

Out of 140 runs, 51% of the runs produced individuals with perfect fitness. For each run the maximum fitness obtained and the number of evaluations required to reach that fitness were logged. The average fitness evolved was 8722 (with a standard deviation of 1635). Figure 8 shows the distribution of fitness scores, and from these results we can see that the least successful robots all managed to make their way around the first bend before crashing or looping back on themselves. The range 6500-7000 contains 26 individuals. Based on their average fitness of 6932, we can see that the robots fail to navigate into the second half of the map - where the walls change from vertical to horizontal. The ability to see where the local optima are is a useful feature of the fitness function. Without having to plot all the runs onto a map and observing where the robot becomes stuck, we can use the absolute fitness values on the fitness map to locate any trouble spots in the environment.

3.3 Generalisation of Evolved Behaviour

It is important to demonstrate that the solutions evolved generalise to different starting conditions. In this experiment we perform two tests for generalisation. In the first, evolution is allowed to solve the maze problem and on success the evolved program is tried from a different starting position. The second starting position is on the right hand side of the map, half way down. The fitness map for this starting condition is shown in figure 9 - the point of highest fitness is now in the top left of the map.

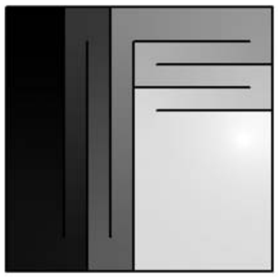
Fig. 9. Fitness values for the maze - starting at different location

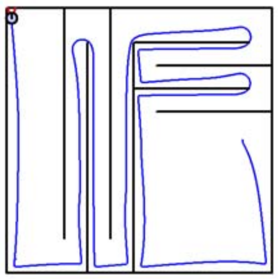
Fig. 10. Solution to the reverse maze problem

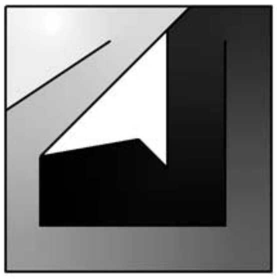
Fig. 11. Fitness values for the unseen environment

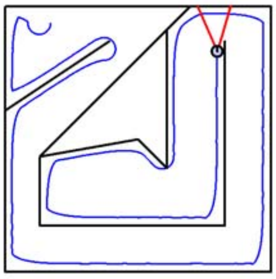
Fig. 12. Example behaviour of robot in unseen enivronment

In the second test the robot is evolved to solve the original maze, and then is put into a different environment, the map with fitness values is shown in figure 11. The maze incorporates features not seen in the previous maps - e.g. walls at angles. In both these scenarios the second test is performed as soon as a solution for the first map has been evolved. We do not allow further evolution to occur.

For the first test of generalisation, from 70 evolutionary runs it was found that 97% of programs that evolved to solve the first map successfully completed the maze when started from a different location. With the new environment, 91% of programs that can complete the first maze can fully explore the second maze. This shows that the evolved programs have achieved a high degree of generalisation.

4 Conclusions

This work has demonstrated the suitability of CGP for control applications. Although it is most often misleading to directly compare results (because of different simulators, environments and methodologies), the results indicate that CGP is highly competitive when compared to previous results using traditional GP. In future work we hope to use CGP to control a physical robot, and to perform the entire evolutionary algorithm in hardware. Currently, we are working on an FPGA implementation of the algorithm, which could be used with the Kephera robots.

References

1. L. Ashmore. An investigation into cartesian genetic programming within the field of evolutionary art. Technical report, Final year project, Department of Computer Science, University of Birmingham, 2000. http://www.gaga.demon.co.uk/evoart.htm.
2. R.A. Dain. Developing mobile robot wall-following algorithms using genetic programming. In *Applied Intelligence*, volume 8, pages 33–41. Kluwer Academic Publishers, 1998.
3. Marc Ebner. Evolution of a control architecture for a mobile robot. In *Proc. of the 2nd International Conference on Evolvable Systems*, pages 303–310. Springer-Verlag, 1998.
4. A. Beatriz Garmendia-Doval, Julian Miller, and S. David Morley. Cartesian genetic programming and the post docking filtering problem. In Una-May O'Reilly and Tina Yu et al., editors, *Genetic Programming Theory and Practice II*, chapter 14. Kluwer, 2004.
5. John R. Koza. *Genetic Programming II: Automatic Discovery of Reusable Programs.* MIT Press, Cambridge Massachusetts, 1994.
6. C. Lazarus and H. Hu. Using genetic programming to evolve robot behaviours. In *Proc. of the 3rd British Workshop on Towards Intelligent Mobile Robots*, 2001.
7. J.F. Miller, P. Thomson, and T. Fogarty. Designing electronic circuits using evolutionary algorithms arithmetic circuits: A case study. In D. Quagliarella and J. Périaux et al., editors, *Genetic Algorithms and Evolution Strategies in Engineering and Computer Science.* John Wiley and Sons, West Sussex, England, 1997.
8. Julian F. Miller. An empirical study of the efficiency of learning boolean functions using a cartesian genetic programming approach. In Wolfgang Banzhaf and Jason Daida et al., editors, *Proc. of GECCO*, volume 2, pages 1135–1142. Morgan Kaufmann, 1999.

9. Julian F. Miller. Evolving developmental programs for adaptation, morphogenesis, and self-repair. In Wolfgang Banzhaf and Thomas Christaller et al, editors, *Proc. of ECAL*, volume 2801 of *LNAI*, pages 256–265. Springer, 2003.
10. Julian F. Miller and Wolfgang Banzhaf. Evolving the program for a cell: from french flags to boolean circuits. In Sanjeev Kumar and Peter J. Bentley, editors, *On Growth, Form and Computers*, pages 278–301. Academic Press, 2003.
11. Julian F. Miller, Dominic Job, and Vesselin K. Vassilev. Principles in the evolutionary design of digital circuits-part I. *Genetic Programming and Evolvable Machines*, 1(1/2):7–35, 2000.
12. Julian F. Miller, Dominic Job, and Vesselin K. Vassilev. Principles in the evolutionary design of digital circuits-part II. *Genetic Programming and Evolvable Machines*, 1(3):259–288, 2000.
13. Julian F. Miller and Peter Thomson. Cartesian genetic programming. In Riccardo Poli and Wolfgang Banzhaf et al., editors, *Proc. of EuroGP 2000*, volume 1802 of *LNCS*, pages 121–132. Springer-Verlag, 2000.
14. Julian Francis Miller. Evolving a self-repairing, self-regulating, french flag organism. In Kalyanmoy Deb and Riccardo Poli et al., editors, *Proc. of GECCO*, volume 3102 of *LNCS*, pages 129–139. Springer-Verlag, 2004.
15. Peter Nordin and Wolfgang Banzhaf. Genetic programming controlling a miniature robot. In E. V. Siegel and J. R. Koza, editors, *Working Notes for the AAAI Symposium on Genetic Programming*, pages 61–67, MIT, Cambridge, MA, USA, 1995. AAAI.
16. Craig W. Reynolds. An evolved, vision-based behavioral model of obstacle avoidance behaviour. In Christopher G. Langton, editor, *Artificial Life III*, volume 16 of *SFI Studies in the Sciences of Complexity*. Addison-Wesley, 1993.
17. Craig W. Reynolds. Evolution of corridor following behavior in a noisy world. In *Simulation of Adaptive Behaviour (SAB-94)*, 1994.
18. Craig W. Reynolds. Evolution of obstacle avoidance behavior: using noise to promote robust solutions. pages 221–241, 1994.
19. J. Rothermich, F. Wang, and J. F. Miller. Adaptivity in cell based optimization for information ecosystems. In IEEE Press, editor, *Proc. of the CEC2003*, pages 490–497, 2003.
20. Joseph A. Rothermich and Julian F. Miller. Studying the emergence of multicellularity with cartesian genetic programming in artificial life. In Erick Cantú-Paz, editor, *GECCO Late Breaking Papers*, pages 397–403. AAAI, 2002.
21. Lukas Sekanina. *Evolvable Components: From Theory to Hardware Implementations.* SpringerVerlag, 2004.
22. A. Thompson. Evolving electronic robot controllers that exploit hardware resources. In F. Morán and A. Moreno et al., editors, *Proc. 3rd Eur. Conf. on Artificial Life*, volume 929 of *LNAI*, pages 640–656. Springer-Verlag, 1995.
23. Vesselin K. Vassilev and Julian F. Miller. The advantages of landscape neutrality in digital circuit evolution. In *ICES*, pages 252–263, 2000.
24. M. S. Voss and J. Holland. Financial modelling using social programming. In *FEA 2003: Financial Engineering and Applications*, 2003.
25. James Alfred Walker and Julian Francis Miller. Evolution and acquisition of modules in cartesian genetic programming. In Maarten Keijzer and Una-May O'Reilly et al., editors, *Genetic Programming 7th European Conference, EuroGP 2004, Proceedings*, volume 3003 of *LNCS*, pages 187–197. Springer-Verlag, 2004.
26. Tina Yu and Julian Miller. Neutrality and the evolvability of boolean function landscape. In Julian F. Miller and Marco Tomassini et al., editors, *Proc. of EuroGP*, volume 2038 of *LNCS*, pages 204–217. Springer-Verlag, 2001.