Genetic Programming as an approach to the problem of synthesis

Erwin Haasnoot  Niels van Kaam  Gerard Simons
Hans van den Bogert *

September 3, 2012

Abstract

Many problems do not have a direct solution in the form of a known algorithm or program to solve such a problem. These problems include, for example, the designing of electrical circuits and producing robots capable of locomotion. These are all part of a greater problem: the problem of synthesis. How can you make a computer design circuits and produce robots for you? In general, how can you automate the synthesis of anything that normally takes a great deal of human ingenuity?

How a program should look to e.g. make robots walk is not known in advance, however the desired result is known. Genetic Programming provides a method to produce such algorithms or programs using evolutionary techniques applied iteratively to a population of programs to gain newer, more fitting programs which show the correct output.

This paper gives an overview on the history of and developments in Genetic Programming. It describes Genetic Programming itself and where it fits among other Evolutionary Computation methods. Genetic Programming was studied and used by J. Koza which used GP for designing electrical circuits, GP is also used by H. Lipson to design the controller and morphology of robots.

1 Introduction

Letting computers automatically solve problems is one of the main goals in Artificial Intelligence and this intelligence can be created in several ways. As early as 1948 Alan Turing had already outed the idea of a "genetical or evolutionary search" [10] to achieve this intelligence. In the 60’s several pioneers in this field emerged all with different methods but with the same basic Evolutionary Computation(EC) principle in mind.

Algorithms in the field of Evolutionary Computation are commonly known as Evolutionary Algorithms (EAs). Specific to EAs is the use of a set of solutions called the

*Peter Bosman as supervisor
population, rather than operating on a single solution. By means of selection, newer generations of the population can be created. How well a solution fits in the problem domain is quantified by a fitness function. A solution with a high fitness is more likely to be a parent of the next generation of solutions.

The next generation is defined as the offspring of the selected solutions from the current generation. Offspring are created by applying variation operators (VOs) to selected solutions. Common variation operators are recombination, which synthesizes partial or whole solutions into new solutions and mutation which applies random variation to a single solution. Reproduction is a variation operator which simply copies the solution as a whole into the new generation. Using a variety of VOs enables the EAs to more extensively explore the problem domain, instead of converging to the nearest local optimum.

Because little explicit programming is involved, and few constraints are imposed, EAs are capable of producing a wide variety of complex and ingenious solutions.

Various specific EAs exist of which Genetic algorithms (GA) and Genetic Programming (GP) are an example. These EA instances will be explained in the next sections to be followed by examples in which GP plays a major role. One example field is Electrical Circuits, where GP helps design the circuits. Evolutionary Robotics is another field in which GP is applied to enable production of robots capable of locomotion.

2 Genetic Algorithms

In GAs, traditionally a solution or individual is a bitstring of fixed size. An initial population of these individuals is created randomly. To generate a new population, selection is applied. The best solutions so far, ranked on the basis of the problem’s fitness function, are chosen to form the base of a new generation.

As the GA is an EA, it too applies variation operators to the selected individuals. In case of bit strings the prominent VOs are as follows.

**Crossover** To generate an individual for the next generation, GA uses a special case of recombination among 2 or more individuals to create new offspring. So when 2 individuals are represented as bit strings e.g. \( x_1x_2x_3x_4x_5x_6 \) and \( y_1y_2y_3y_4y_5y_6 \), where the amount of elements is chosen arbitrarily, their genetic recombination, which is also called crossover results into new individuals in the following manner: by chance a crossover point is chosen and indicates on which index the bit string should be cut in two. With a crossover point of 2 the two bit strings are now \( x_1x_2|x_3x_4x_5x_6 \) and \( y_1y_2|y_3y_4y_5y_6 \). The crossover is completed when the parts of the individuals are interchanged, resulting in: \( x_1x_2y_3y_4y_5y_6 \) and \( y_1y_2x_3x_4x_5x_6 \). As mentioned before a GA isn’t restricted to using 2 parents, however using more implicates that there should be more than 1 one crossover point for all offspring involved in the crossover to be changed.

**mutation** Besides the recombination variation, a GA can use mutation to variate an individual by flipping a bit in the string, e.g. if \( y_1y_2x_3x_4x_5x_6 \) represents the string 000110.
The mutation of this string, if chosen for a single mutation, and chance decides to flip the third element, is 001110.

Using encodings similar to this enables the VOs to be readily applied to solutions, which makes GA a useful instance of EA and here there is an understanding of how the genetic behaviour can produce new solutions to evolve to a satisfying solution. In the next section, Genetic Programming takes this approach one dimension further where a solution is not simply a parameter for a fitness function, but a function or program itself to be matched against a fitness function.

Coevolution Sometimes it is necessary for an Evolutionary Algorithm to evolve two or more populations simultaneously. This is called coevolution of populations. An analogy from the real world would be evolving populations of foxes and rabbits, predators and prey, solutions and problems. In this analogy the foxes would be the solution population. Trying to approach the optimal solution of the problem as much as possible. The rabbits would be the problem population, trying to avoid the solutions, not allowing ‘them’ to approach the best solution. This approach has been shown to increase the speed of evolution towards the best solution significantly. Coevolution is mostly applied in this way, but it is not limited to solution/problem populations. Solution/solution (Symbiotic or Parasitic) populations are often used aswell.

3 Genetic Programming

Genetic Programming(GP) can be viewed as a variation of GA, in which the individuals in the population are not of fixed length. Furthermore the individuals are not encoded candidate solutions to a problem, but they represent programs that, when executed, are candidate solutions to the problem. Individuals in GP can traditionally be seen as syntax trees representing a program.

Figure 1 is such a syntax tree and provides a fundamental example in this section, representing the mathematical function $x^2 + x + 1$. How to produce such a tree is easy
once the mentioned function is converted to prefix notation resulting in: \(+(1, +(x, \times(x, x)))\)

An initial population is made randomly. Individuals consist of i) terminals, like constants and variables in the example function and ii) functions, like substraction and multiplication, which operate on terminals, or other functions. To continue the numerical example, one might get an intial population of the following prefix notated functions:

1. \(-((x, 1), 0)\)
2. \(+((1, \times(x, x))\)
3. \(+((2, 0))\)
4. \(\times((x, -(-1, -2)))\)

As with other EAs a method of quantifying the fitness of an individual is needed to compare individuals so they can be selected on their fitness. With GAs the fitness function takes an individual as input and computes the function’s corresponding output. However with GP an individual is a program or function, which itself has input and output and is thus not a mere encoded value as is the case with GAs. It is common in GP to test the fitness of programs in the population by means of fitness cases. These cases may test programs i.e. simulate, by means of different inputs which are relevant to the problem domain. Comparing the initial example population with the wanted function \(x^2 + x + 1\) can be defined as the area between an individual’s represented curve like, \(x + 1\) and \(x^2 + x + 1\). The larger the area is, the less an individual behaves like the wanted function and thus, the less fit an individual is. Note that in this example the fitness cases were every input \(x\) on the interval \((-2, 1.5)\) (Figure 3) and the fitness function is defined as the area between the curves where a smaller area is more fit than a larger area. Figure 3 displays the error between the individual \(x + 1\) and the function we’d like to attain. The shaded parts of the diagram show the error of the individual \(x + 1\)

To produce more fitting solutions, VOs are applied to the selected subset of the population. Crossover and mutation are much used VOs which can be applied to the syntax tree which will be shown using the initial population. By chance the mutation operator is applied to the \(x + 1\) individual. Traditionally mutation is implemented by choosing an

![Figure 2: 4 random prefix trees for the initial population](image-url)
edge in a syntax tree where a new randomly created tree is inserted, producing \( x + 1 + x \) for example.

Crossover can be implemented in a syntax tree by swapping subtrees from 2 or more individuals. By chance this is applied to the trees (a) and (b) (Figure 2), marking the + node and an \( x \) node and their subtrees respectively. In prefix notation the to be crossed over parts are marked between pipe signs

- (a) \(-([+\,(x,1)],0)\)
- (b) \(+((1,*(\,|\,x\,|,x))\)

One offspring would be \(+((1,*(\,(x,1),x))\). When this new individual is tested with the fitness function it would have a fitness of 100%. This not without reason because the new offspring is mathematically equivalent to our are wanted equation \( x^2 + x + 1 \).

This example is fairly optimistic because the example happens to have chosen the ideal pair of subtrees to produce a highly fit offspring. However the example shows that is not inconceivable to gain such an ideal offspring. Also keeping in mind that in practical situations the fitness is not merely simulated on such a small set and can be computed in parallel to quantify thousands of individuals’ fitness, making the odds of getting better individuals higher.

The next section shows how to synthesize parts of an electrical circuit using GP, instead of synthesizing parts of mathematical functions.

### 4 Automated Design of Analog Electrical Circuits

One of the most successful applications of Genetic Programming is the Automated design of analog electrical circuits (ECs). A few years before the millennium, before Genetic Programming, there was no general automated technique for the design of analog electrical circuits based on a high level description. Most analog circuits at that time were hand-crafted by experts. The design process of analog circuits is a multiphase iterative task, and
usually is very time-consuming. These tasks had to be performed by very skilled and experienced engineers, resulting in a shortage of engineers capable of designing these circuits. Thus, a practical method to take work away from them was needed. At that time there were some methods to design ECs, but those methods did not prove useful because it took way too much time to find a solution. They managed to design very simple ECs through these methods, but as the complexity of the circuit (and in turn the description) increases, the search time exploded to multiple human lifetimes. Pioneered by John R. Koza, Genetic Programming would give very interesting results in the design of EC’s. In this chapter we will explain the basics of how EC’s can be designed by using Genetic Programming, and provide an overview of the promising results Koza has accomplished.

4.1 The Method

Circuits are developed using Genetic Programming. As described earlier, the procedure breeds a population of point-labeled trees, which are acyclic graphs. However electrical circuits are line-labeled cyclic graphs. See Figure 4. In order to use GP to design a circuit, there needs to be a mapping from a Program Tree(A) to a Circuit(B).

![Figure 4: a function tree (A) and an electrical circuit (B)](image)

This is done by making use of an "Embryonic circuit". An embryonic circuit is a circuit that contains all the inputs and outputs for a specific problem. Furthermore the embryonic circuit contains some fixed components that are not modified in the process, and wires that are to be modified in the process. The embryonic circuit does not produce any interesting output. See Figure 5 for an example of an Embryonic Circuit Koza used for a source identification circuit. [6] In the example Rsource and Rload are fixed components. Vsource is the input to the circuit, and Vout the output. Z0 is a wire to be modified in the process. In this example there is only one modifiable wire.

This embryonic circuit is modified by using circuit-constructing functions. In Figure 6 it is shown what happens if we apply a component-creating function to the example1. In this case we used a resistor-creating function. The modifyable wire is now replaced by a resistor.

It is possible to construct a program tree using these circuit constructing functions. Thus, the electrical circuit that is to be produced, is defined by a program tree of functions

---

1Todo: Afbeelding van Circuit-constructing tree
Figure 5: An embryonic circuit used for a source identification circuit.

Figure 6: The example after applying a resistor-creating function.

to be applied to the embryonic circuit. These functions grow the embryonic circuit into a larger, more complex possible solution to the problem. Because now we have an electrical circuit defined by a program tree, it is possible to use GP to design an EC.

The Component-Creating Functions alone however are not sufficient to produce complex EC’s. Next to these CCF’s that insert components into the circuit there also are Connection-modifying functions (CMF’s) that alter the circuit topolog. Furthermore there are Arithmetic-performing functions that appear in sub trees as argument(s) to the CCF’s and specify the numerical value of the component (APS = Arithmetic-Performing Subtree). These APS’s have a different meaning for each CCF. In the example with the resistor, the
APS defines the resistance of the resistor.

The electrical circuit is constructed by executing the circuit-constructing program tree on the embryonic circuit. The program tree consists of the circuit-constructing functions mentioned above. We shall give a short description of CCF’s and CMF’s.

4.1.1 Component-Creating Functions

Component-Creating Functions insert a component into the developing EC. In the example above we also used a CCF. More specifically, the Resistor Creating Function. Each CCF has a writing head that points to a component that will be modified. Note that wires in the circuit are also components that can be modified. In Figure 6 we modified the wire from Figure 5. The number of input arguments to the function differs per CCF. For example a transistor has 2 input wires and 1 output wire. The APS for a CCF also is different per CCF. The purpose of the APS is to specify a constant value for specific components created by the CCF. The APS for a CCF contains arithmetic operations (subtraction and addition) and random values between -1.000 and 1.000. This way the APS defines a floating point value. For the example with the resistor creating function, the APS defines the resistance of the resistor. Per CCF a unit of measure is used to convert the value to a value that is appropriate for the specific component. For the resistor it would convert the value between -1.000 and 1.000 to a positive amount of kilo Ohms.

Finally, we look at what a the resistor creating function looks like. The function has 2 inputs in the program tree. The first input is the construction-continuing sub tree. This means the component can still be subject to modification, see figure 7. The second input is the APS to produce the value for its resistance.

4.1.2 Connection-Modifying Functions

The purpose of the CMF’s is to modify the circuit’s topology. Just as CCF’s there are a lot of different CMF’s. In this review we will not discuss every CMF. We will give an example
Figure 8: The example after applying the "SERIES" operation on the resistor from figure 6.

Figure 9: The program tree with the "SERIES" operation, as an extension on the program tree in figure 7.

of the SERIES CMF. The series operation places 2 components in a series topology. See Figure 8 where the series operation is applied to the resistor from Figure 6. The series operation has 3 inputs. All the inputs are construction-continuing sub trees. argument 1 is the 1st resistor, argument 2 is the 2nd resistor and argument 3 is the wire in between. All these component are subject to modification in future generations. See figure 9 for the resulting program tree.

4.2 Fitness

The determination of the fitness is a very important part of an EA. To determine the fitness, at first the program tree has to be converted to a circuit. This is done by applying
the functions in the program tree to the embryonic circuit. This way a so called ”netlist” [5] is created. This netlist has to be simulated in some software. For this Koza used a modified version of a program called ”SPICE”. Other tools to simulate EC’s should be possible as well.

The fitness measure is a number that determines how ”fit” a specific solution is. Solutions with a better fitness (a number closer to 0) have a larger chance to be used for a future generation. The fitness measure is determined by a high-level description of the circuit. This description is of course different for each problem. The description must be well-measurable (in the simulation) and guide the evolution[5]. ”guide the evolution” means you’ll want the fitness measure to determine how close solution is to the ideal solution as good as possible. For example, what would the fitness function for a lowpass filter look like:

An easy way would be to map the desired output and the output of a specific solution and determine the total difference, which is the area. This is an easy calculation (you could estimate the area by taking samples at specific frequencies) and it guides the evolution because samples that work better have a smaller area. For some generated circuits (In early generations many [5]) might not be possible to be simulated in the chosen program. These circuits remain in the population but get a very high value in their fitness.

This example shows the idea of the fitness determination. To get better results a slightly more complex fitness function is needed, to guide the process even better. For the lowpass filter Koza made a difference in the differences between the desired output and the given output. Small acceptable differences were given much smaller penalties in its fitness than larger differences. If the difference was larger than the acceptable limit, the area penalty was multiplied by ten.

4.3 Evolving the circuits

With the technique described above, Koza evolved different solutions for many different problems different problems. The program was run multiple times for the same problems. At different runs, the circuit-constructing functions were (slightly) modified. This produced very interesting results.

On for example the lowpass filter from the previous section, the program produced recognizable design patterns: In one of the runs the evolved circuit contained a ladder topology specific for a Butterworth or Chebychev filter. In another run the circuit contained the topology of the elliptic invented and patented by Cauer.[5]. That means GP has rediscovered the ladder in Butterworth and Cheychbev filters and the elliptic Cauer topology.

The computation time required to evolve a satisfying solution for a problem cannot be told exactly in advance. The process was monitored, and stopped whenever a solution good
enough was produced. For some problems a second run was needed to evolve a satisfying solution. Koza ran the program on a computer consisting of 64 80-MHz processors, on which each run took about 2 days on average. On most currently available supercomputers this would take up to a few seconds. Determining the fitness took most computation time, because it requires to simulate the circuit. The process of determining the fitness could still be optimized, by optimizing SPICE. ³

To produce designs for EC's to be used in the industry, some more requirements would have to be added. For example, power consumption was not included in the fitness function. Adding the extra requirements would make the fitness determination more complex and thus slower, which results in a longer computation time.

All results were produced with as input only a very basic description of the problem, the fitness function. To provide such high-level description much less experience and knowledge is required. Previous to this there was no such way to do this automatic, and the design of new circuits had always been a task of educated and experienced engineers. This technique looks very promising at changing that.

5 Robust and super-sensitive Analog Circuits

5.1 Introduction

Recently, there has been increasing interest in the generation of robust and, its inverse, sensitive (or tamper-evident) analog circuits. Robustness of a circuit is defined as the ability to continue its designed function under changes in the environment or its parameters. Protecting against a loss of functionality through (accidental) removal of components and failures of systems, it is essential for creating a reliable system that can continue operating under harsh conditions. Sensitivity of a circuit is defined as the ability to stop its designed function under minimal changes in the environment, effectively giving the circuit a passive protection against Reverse-Engineering.

Traditionally, Sensitivity and Robustness were treated as "after-thoughts" [1]. After a circuit had been designed, parameters would be tweaked in order to maximize robustness of the circuit. Circuits created this way were often sub-optimal when it came to their robustness. For this reason, research has been done to incorporate robustness of a circuit into the design stage, instead of tweaking parameters after the system had already been designed.

GP solutions have been successfully applied to robust circuit design. Hu [3] and Kim [4] showed that synthesizing robust circuits through the use of GP is possible and produces very good results. By including robustness as part of the fitness of a synthesized circuit, they were able to create passively robust circuits, while adding only a minimal amount of circuitry. Kim also showed that by using the inverse of the robustness fitness function, it was possible to create passively super-sensitive circuits.

³A electrical circuit simulation program used to simulate the EC’s
5.2 Adding Robustness

The ‘robustness’ of a circuit is in both Hu’s and Kim’s research defined as the degradation of the circuit’s function resulting in a (complete) failure in the system. Kim called this robustness ’factor’ R. The circuit synthesized in this case was a low-pass filter, a filter that lets low frequency signals pass and blocks high frequency signals. A narrow ’don’t care’-band was in between, where it doesn’t matter if the signal passes or gets blocked. This type of circuit has been researched much in the past, making it a logical pick.

To every component in the synthesized circuit, Kim added 2 resistors. One resistor in parallel, so he could simulate (partial) circuit shorts (1/R), where 0 resistance on the resistor resulted in a full circuit short. The other resistor was put serially with the component, simulating (partial) circuit damage (R), where infinite resistance would simulate a full circuit disconnect. The domain of R is [0,2.0] where R = 2.0 means infinite resistance in the resistors and R = 0 means zero resistance. A full robustness evaluation ’activated’ resistors randomly, so resistance could be applied to them, and then calculating the circuit performance on 101 points in the domain of R. Kim would do this an unspecified amount of times, taking the worst evaluation (the simulation with the worst performance) as the evaluation’s outcome.

As this is a rather computationally intensive operation, Kim approximated it by only checking the performance at two points in the domain instead of 101. He used two different evolution strategies here, the first one used the points 0 and 2.0. Effectively checking how the system performed under optimal conditions (R = 0) and the harshest conditions possible (R = 2.0). The second strategy used coevolution to evolve those points. Every 500th generation Kim would run a full robustness evaluation on the circuit, while also evolving a generation of pairs of points. These pairs were ranked on their performance, how closely they approximated the full robustness evaluation. After that the top ranked pair was used as points, in the fitness evaluation, to evolve the next 500 generations of circuits. The other part of the fitness evaluation, for both strategies, was the performance of the circuits. Kim checked the actual performance against the expected performance on 101 points in the frequency domain. The smaller the difference, the higher the fitness. This produces a fitness function that’s based on both robustness and performance of a circuit.

Kim compared the performance against one of Koza’s evolved circuits (through GP) and a circuit from Butterworth (designed in the 'old-fashioned' way).

In Figure 10 you can see the performance mapped against the amount of degradation (R) on each of the circuits tested. Koza’s and Butterworth’s circuits perform better under optimal conditions, but when you introduce circuit deterioration Kim’s circuit starts to outperform Koza’s and Butterworth’s.

It is very much possible to create robust circuits through GP by adding a form of robustness to the fitness evaluation function. Against a traditionally designed circuit (Butterworth) and one human-competitive circuit synthesized using GP, Kim’s circuit did well exactly on that which it was evolved for. Robustness. As robustness evaluation functions get more and more sophisticated, the trade-off between performance (Koza’s, Butterworth’s) and robustness (Kim’s) might become less off an issue. This allows the synthesis of more
robust circuits while maintaining performance levels. However, the greatest advantage GP has over other traditional design methods is that only the fitness evaluation has to be changed in order to add robustness, or any other feature a circuit might need.

6 Evolutionary Robotics

6.1 Introduction

Taking evolvable hardware one step further, one could imagine building a robot capable of exhibiting high-level functionality - not a trivial task. It is called the problem of synthesis and although many best-practice approaches and guidelines do exist, it is still a problem mainly addressed by human ingenuity. A formal model of automatic open-ended synthesis still doesn’t exist. Devising other methods for inventing new robotic configurations remains difficult. This is the driving concept of Evolutionary Robotics. Instead of an intelligent design, we would let a population of robotics be subject to evolution. Those robots who possess skills deemed fit for their environment, like traversing the greatest distance, have greater odds of providing offspring for future generations. Evolving these robots imposes an even greater challenge than evolving circuits. Not only are circuits necessary to constitute a successful robot, but a valid configuration of other hardware, such as legs, wheels and other body parts are needed in addition to actuators and sensors. These parts must then be assembled as such that they act efficiently in an environment and the constituent parts must be correctly controlled.

As shown, the idea of evolving robotics is an exciting and challenging one. Why then all the effort, what encourages us to do so? Robotics have been around for decades, especially industrial ones, automating manufacturing of cars and other products. The
tasks successfully fulfilled by contemporary robotics are very specific and the robotics are specially designed for this task. As manufacturing processes and other robot functions increase in complexity, so will robotics have to increase in their complexity. As robotics become more and more complex, and need to be more adaptable and generic, their design will become too difficult for human creativity and instinct to tackle. Evolutionary robotics is much more scalable and can, in theory, be scaled beyond human engineering skills, creating vastly more complex and efficient robots humans will ever be able to design.

The idea of evolving robotics is an exciting and challenging one. Why then all the effort, what encourages us to do so? Robotics have been around for decades, especially industrial ones, automating manufacturing of cars and other products. The tasks successfully fulfilled by contemporary robotics are very specific and the robotics are specially designed for this task. As manufacturing processes and other robot functions increase in complexity, so will robotics have to increase in their complexity. As robotics become more and more complex, and need to be more adaptable and generic, their design will become too difficult for human creativity and instinct to tackle. Evolutionary robotics is much more scalable and can, in theory, be scaled beyond human engineering skills, creating vastly more complex and efficient robots humans will ever be able to design.

### 6.2 Morphology & Controllers

A robotic system consists of two major parts: The morphology and the controller. The morphology constitutes the tangible real parts of the robots, the physical entities which make up a robot like trusses, wheels and sensors. The controller is the part of the robot responsible for operating these parts. The distinction is very similar to the hardware-software distinction made in IT. Though the division between the two is usually blurry at best, it is still useful for understanding how robotics are evolved. We will now adress a few exciting and prominent experiments of evolutionary robotics, in which the controllers and also the morphologies are adapted, after which we will discuss the current challenges and limitations imposed on Evolutionary Robotics.

### 6.3 Evolving Controllers

We will first describe the process of evolving the controllers of robots. The controller of a robot can be defined in many ways, as a wide array of if-else statements, finite state machines or others. The method most used in contemporary robotics is a neural network. A neural network is an artificial model of a biological neural network. Models of neurons are created and interconnected in a certain architecture. These neural networks map sensory input to actuator outputs, and the connections are given certain weights. The exact layout and mechanisms of these networks can vary widely and sometimes the synthesis algorithm determines the architecture. An interesting example of Bongard[2] explores the concept of evolving controllers for a simulated legged robot (there are other experiments that did so for wheeled robots[8]). The legged robots consisted of a fixed morphology consisting of four legs and rotary actuators, and a few rotary sensors and chemical sensors. The sensors
were then connected to the actuators through a neural network. The goal of the robots was to reach an area with a high concentration of a chemical substance. Those robotics who were able to reach the high-concentration areas by travelling the shortest distance were deemed fittest. Using this concept in an actual physical world is obviously even more exciting.

Similarly, attempts were made to evolve controllers for a dynamical legged robot [9]. The nine-legged machine, composed of two platforms, was powered by pneumatic linear actuators powered by paintball canisters. The controllers were only able to regulate power and duration of the actuators. Eventually a valid on-off pattern was evolved able to sustain the robot.

From these examples we can deduce that the evolution of controllers seems a valid approach in evolving robots exhibiting high-level functionality, even though very pronounced challenges exist. These lie mainly in the accurate yet fast simulation of the evolved controllers. Other approaches evolve both controller and morphology, which we will now consider.

6.4 Coevolving Controllers and Morphology

Coevolution in our case consists of evolving both the morphology and controller of robots simultaneously and thus in providing a more powerful tool in approaching a valid solution more quickly. Lipson[7] experimented with this tactic by allowing robots to be created from bars and linear actuators attached to them, which controls the bars length. The controller was defined as a neural network.

Neural Networks  Artificial Neural Networks can be defined as a model of reasoning based on the biological neural networks which constitute a brain. Neurons are the basic information-processing units and the axons the medium down which information is propagated (analogous to the copper wire). The axons connect to other neurons using synapses. Instead of bits, neural networks mediate so-called action potentials, which are released from neurons after they have been excited beyond a certain threshold and travel down the neural network possibly exciting other neurons possible initiating even more action potentials. The action potentials travel down a neural network in a very distributed ripple-like fashion. It is this distributed processing behaviour that gives neural networks their amazing computational ability. Artificial neural networks tap into this resource by simulating neural networks, they are modeled as follows: synapses are modeled as weights, a numeric value assigned to each connection. Positive weights for connections mean they excite the next neuron, whereas negative weights inhibit the next neuron. The neuron then computes the weighted sum of the inputs, called the linear combination. The activation function then computes the corresponding output by checking the sum with the threshold value, called the activation function (see Figure 11. When multiple layers of these neurons are connected together into an actual neural network, the complexity rises exponentially giving a wide variety of functions that can be fulfilled. Usually input can be given by sensors, in the case of Lipson[7] however, no sensors were used and a repeating pattern was
evolved as network input. The outputs of the network were connected to the actuators. By modifying the neurons and the connectivity the behaviour of the network is modified.

Figure 11: A model of a neuron.

The controller of the robots is defined, as previously explained as neural networks, the morphology as bars connected with each other through ball-and-socket joints. The goal of the robots was simple; get a net distance as high as possible. Starting with 200 blank machines consisting of no bars and no neurons, evolution in simulation was conducted. Though it took tens of generations for any movement to occur at all, eventually very elaborate and effective mechanisms were developed, some using anti-phase synchronization, ratcheting and dragging. It is interesting to note that despite a lack of explicit declaration in the fitness function, all robots seemed to converge to symmetry, perhaps because symmetry allowed for more straight movement. Also many of the evolved robots contained redundancies, possibly to make small alterations in morphology less catastrophic.

The results of these experiments are very interesting in that they seem to resemble the diversity found in natural life. Even with these rather simple elements, complicated and efficient robots were created in a relatively short time span. The results are encouraging for further research and seem to confirm the usefulness of creating intelligent robotics by coevolving both controller and morphology.
6.5 Main Challenges & Limitations

The results mentioned above seem to indicate a promising future for Evolutionary Robotics. However, to provide a balanced overview of the prospects of Evolutionary Robotics, it is necessary to provide a short overview of the main challenges faced by Evolutionary Robotics as well. Creating controllers for robotics capable of performing a complicated task requires evolving large populations over many generations, which is a very time consuming process. Because not every robot from every population can be built, simulation is needed to evaluate which robots are worth the effort of actually constructing them. Because simulations are but models of reality, they are practically always flawed. To minimize the errors created by this discrepancy, more accurate simulations are required. Requiring fast yet accurate computer simulations is one of the most important limitations imposed on Evolutionary Robotics, today.

In this section we have considered the problem of synthesis: using a finite set of elements, both in terms of morphology as well as controller, to construct a robot capable of exhibiting high-level functionality suitable for its environment and task. By considering some of the more recent research done in the field of evolutionary robotics we have sought to provide good terms for an alternative to intelligent design when it comes to creating robotics. The examples show that evolution of both morphology and controllers were suitable approaches in producing useful robotics. Furthermore that coevolving morphologies and controllers are especially powerful in creating diverse species of robotics with effective capabilities.

7 Conclusion

This article reviews some of the applications of Genetic Programming. We have shown that over ten years ago GP proved promising at taking over tasks that previously only could be executed by educated and experienced engineers. Given a high-level description it was possible to make a computer evolve solutions that approach the quality of those designed by engineers. One of the requirements of industrially more usable circuits is the robustness. In the article we described more recent work on making them robust or very sensitive. Though these features of a circuit are traditionally difficult to design, however it was proven to integrate well into the GP design approach. Merely altering the fitness function to consider robustness was shown to be enough.

Where electrical circuits are reasonably 'low level' applications of GP, creating robots capable of exhibiting high-level functionality could be considered 'high-level' applications. This article shows that, although research in this field is still young, it is possible to evolve robots that, for example, are capable of locomotion. Some research focusing on evolving the controller for existing robots, whereas others evolved both the controller and the robot morphology.

Genetic Programming is a powerful tool towards solving the problem of synthesis. Automating synthesis of at least basic robots and complex electrical circuits, while showing
its potential through easily integrating traditionally hard to design features of electrical circuits. The limitations imposed on applications of GP are almost always due to limited computing power. As computing power rises, we will definitely start seeing more and more applications of GP in fields that noone has even considered yet. Or as Hod Lipson aptly put it ”As a theoretical extreme, if we could use only atoms as building blocks, laws of physics as constraints and nanomanipulation for fabrication, the versatility of the manufacturable design would be maximimized” [7]. Though we are still far away from reaching this level of detail, it clearly shows the potential of GP.

References

[1]


