Predicting Prime Numbers Using Cartesian Genetic Programming

Conference Paper · April 2007

DOI: 10.1007/978-3-540-71605-1_19 · Source: DBLP

2 authors:

James Alfred Walker
The University of York

Julian Francis Miller
The University of York

Predicting Prime Numbers Using Cartesian Genetic Programming

James Alfred Walker and Julian Francis Miller

Intelligent Systems Group, Department of Electronics, University of York,
Heslington, York, YO10 5DD, UK
{jaw500,jfm7}@ohm.york.ac.uk

Abstract. Prime generating polynomial functions are known that can produce sequences of prime numbers (e.g. Euler polynomials). However, polynomials which produce consecutive prime numbers are much more difficult to obtain. In this paper, we propose approaches for both these problems. The first uses Cartesian Genetic Programming (CGP) to directly evolve integer based prime-prediction mathematical formulae. The second uses multi-chromosome CGP to evolve a digital circuit, which represents a polynomial. We evolved polynomials that can generate 43 primes in a row. We also found functions capable of producing the first 40 consecutive prime numbers, and a number of digital circuits capable of predicting up to 208 consecutive prime numbers, given consecutive input values. Many of the formulae have been previously unknown.

1 Introduction

There are many questions relating to properties of primes numbers that have fascinated mathematicians for hundreds of years [1]. It is well known that no formulae have ever been produced that can map the sequence of natural numbers into the sequence of primes. However there exists many simple polynomials that can map quite long sequences of natural numbers into a sequence of distinct primes (long is generally measured with respect to the so-called Euler's polynomial $x^2 - x + 41$ [2] which produces distinct primes for values of x from $1 \leq x \leq 40$). Euler's polynomial continues to produce many primes for larger values of x.

Legendre found a similar polynomial $x^2 + x + 41$ which produces prime numbers for $0 \leq x \leq 39$, and it is this polynomial which, oddly, is referred to as *Euler's polynomial* [3,1]. In this paper we present evolved polynomials that can generate long sequences of primes (including re-discovering Euler's and Legendre's polynomials). Recently there has been renewed interest in the mathematics of prime-producing polynomials [4]. In evaluating the quality of prime-producing polynomials we must observe that there can be many criteria for deciding on the fecundity of prime-producing polynomials. Since polynomials can produce positive or negative quantities, some discovered polynomials are particularly fecund at generating positive or negative primes. Other polynomials can produce long sequences of primes, however prime values may be repeated. The most sought

M. Ebner et al. (Eds.): EuroGP 2007, LNCS 4445, pp. 205–216, 2007.

after quality of prime-producing polynomials appears to be the longest sequence of positive distinct primes [5]. Last year there were many computational attempts at producing prime-producing polynomials and a polynomial of degree five was found to be particularly good (though not at producing distinct, positive primes) [6].

Ulam discovered that there are many integer coefficients, b and c, such that $4x^2 + bx + c$ generates a large number of prime numbers [1]. The polynomial $41x^2 + 33x + 43321$ has also been shown to produce prime numbers for ninety value of x, when $0 \leq x \leq 99$, but only twenty six of the primes are consecutive [1]. A high asymptotic density of primes is often considered to be an important criterion of the fecundity of prime-producing polynomials [7]. Gilbert Fung announced his discovery of two polynomials $103x^2 - 3945x + 34381$ and $47x^2 - 2247x + 21647$ which produces prime numbers for $0 \leq x \leq 43$. However, the best polynomial found so far is $36x^2 - 810x + 2753$, which was discovered by Ruby (immediately after hearing Fung's announcement), and produces primes numbers for $0 \leq x \leq 44$ [5]. The interested reader may consult [8] and [9] for more recent mathematical findings on the subject. Since polynomials of fixed order can be easily transformed by translation operations, there are in fact infinitely many quadratics that have 'Euler-fecundity'. The most important mathematical quantity characterising the essential behaviour of prime-producing polynomials is the polynomial discriminant which for a quadratic of form $ax^2 + bx + c$ is $b^2 - 4ac$. Mollin gives tables of polynomials with particular discriminants that produce long sequences of primes [5].

Euler's polynomial was the inspiration behind one of the GECCO competitions in 2006. The aim of the GECCO Prime Prediction competition (and some of the work in this paper) was to produce a polynomial $f(i)$ with integer coefficients, such that given an integer input, i, it produces the i^{th} prime number, $p(i)$, for the largest possible value of i. For example, $f(1) = 2$, $f(2) = 3$, $f(3) = 5$, $f(4) = 7$. Therefore, the function $f(i)$ must produce consecutive prime numbers for consecutive values of i. The requirement that the polynomial must not only produce primes for consecutive input values, but also that the primes themselves must be consecutive, makes the problem considerably more challenging than mathematicians have previously considered. The two approaches described in Section 4.2 were entered in the GECCO Prime Prediction competition and were ranked second overall. The winning entry evolved floating point co-efficients of a polynomial using a Genetic Algorithm (GA), where the output of the polynomial was rounded to produce the prime numbers for consecutive values of i. However, the winning entry was only able to predict correctly a few consecutive prime numbers (9 in total). Unfortunately, the details regarding this have not been published.

So far, it seems that no integer polynomial exists, which is capable of producing sequences of consecutive prime numbers. In this paper, we are proposing two approaches to evolve a formula (in one case strictly a polynomial) capable of producing prime numbers. The first approach treats the consecutive prime number producing formula as a symbolic regression problem. The technique used for these approaches is Cartesian Genetic Programming (CGP)[10]. The second approach evolves a digital circuit, which can produce consecutive prime numbers for consecutive input values. Any digital circuit can be represented as a polynomial expression, as any logic function can be expressed using only addition, subtraction or multiplication. The technique used to evolve the consecutive prime generating digital circuit is an extension of the CGP technique, known as multi-chromosome CGP [11]. Multi-chromosome CGP has been shown to significantly improve performance on difficult, multiple-output, digital circuit problems, when compared with the conventional form of CGP [11].

The discovery of new prime producing formulae (consecutive, or otherwise) would be of interest to mathematicians, as it is unknown whether such formulae currently exist. Even if such formulae do exist, they may be too complex for a human mathematician to discover. Therefore, this paper once again highlights the use of evolutionary computation as a tool for discovery and design. Also, we propose that the evolution of prime producing formulae would make an interesting and challenging benchmark for comparing evolutionary computation techniques, as it proved clear by empirical tests that it is a harder and more complex problem to solve than many existing GP benchmarks.

The plan for the paper is as follows: section 2 gives an overview of the CGP technique, followed in section 3 by a description of the multi-chromosome extension to the CGP technique. The details of our experiments on evolving sequences of prime numbers are shown in section 4, followed by the results in section 5. Section 6 gives conclusions and some suggestions for future work.

2 Cartesian Genetic Programming (CGP)

Cartesian Genetic Programming is a form of Genetic Programming (GP) invented by Miller and Thomson [10], for the purpose of evolving digital circuits. However, unlike the conventional tree-based GP [12], CGP represents a program as a directed graph (that for feed-forward functions is acyclic). The benefit of this type of representation is that it allows the implicit re-use of nodes in the directed graph. CGP is also similar another technique called Parallel Distributed GP, which was independently developed by Poli [13]. Originally CGP used a program topology defined by a rectangular grid of nodes with a user defined number of rows and columns. However, later work on CGP showed that it was more effective when the number of rows is chosen to be one [14]. This one-dimensional topology is used throughout the work we report in this paper.

In CGP, the genotype is a fixed length representation and consists of a list of integers which encode the function and connections of each node in the directed graph. However, the number of nodes in the program (phenotype) can vary but is bounded, as not all of the nodes encoded in the genotype have to be connected. This allows areas of the genotype to be inactive and have no influence on the phenotype, leading to a neutral effect on genotype fitness called neutrality. This unique type of neutrality has been investigated in detail and found to be extremely beneficial to the evolutionary process on the problems studied [10,15,14].

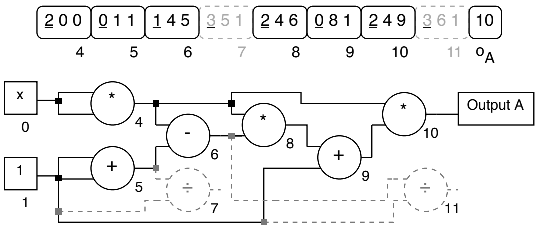

Fig. 1. A CGP genotype and corresponding phenotype for the function $x^6 - 2x^4 + x^2$. The underlined genes in the genotype encode the function of each node, the remaining genes encode the node inputs. The function lookup table is: $+(0)$, $-(1)$, $*(2)$, $\div(3)$. The index labels are shown underneath each program input and node. The inactive areas of the genotype and phenotype are shown in grey dashes.

Each node is encoded by a number of genes. The first gene encodes the node function, whilst the remaining genes encode where the node takes its inputs from. The nodes take their inputs in a feed forward manner from either the output of a previous node or from the program inputs (terminals). Also, the number of inputs that a node has is dictated by the arity of its function. The program inputs are labelled from 0 to n-1, where n is the number of program inputs. The nodes encoded in the genotype are also labelled sequentially from n to n+m-1, where m is the user-defined bound for the number of nodes. If the problem requires k program outputs, then k integers are added to the end of the genotype, each encoding a node output in the graph where the program output is taken from. These k integers are initially set as the outputs of the last k nodes in the genotype. Fig. 1 shows a CGP genotype and corresponding phenotype for the function $x^6 - 2x^4 + x^2$ and Fig. 2 shows the decoding procedure.

3 Multi-chromosome Cartesian Genetic Programming

3.1 Multi-chromosome Representation

The difference between a CGP genotype (described earlier in section 2) and a Multi-chromosome CGP genotype, is that the Multi-chromosome CGP genotype is divided into a number of equal length sections called chromosomes. The number of chromosomes present in a genotype is dictated by the number of program outputs required by the problem, as each chromosome is connected to a *single* program output. This allows large problems with multiple-outputs (normally encoded in a single genotype), to be broken down into many smaller problems (each encoded by a chromosome) with a single output. This approach should make the larger problems easier to solve. By allowing each of the smaller problems to be encoded in a chromosome, the whole problem is still encoded in a single genotype but the interconnectivity between the smaller problems (which can cause obstacles in the fitness landscape) has been removed.

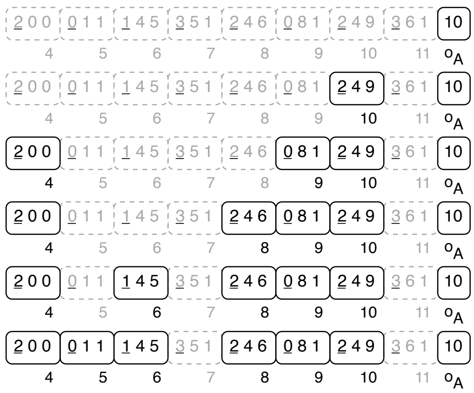

Fig. 2. The decoding procedure of a CGP genotype for the function $x^6 - 2x^4 + x^2$. a) Output A (o_A) connects to the output of node 10, move to node 10. b) Node 10 connects to the output of nodes 4 and 9, move to nodes 4 and 9. c) Nodes 4 and 9 connect to the output of node 8 and program inputs 0 and 1, move to node 8. d) Node 8 connects to the output of nodes 4 and 6, move to node 6, as node 4 has already been decoded. e) Nodes 6 connects to the output of nodes 4 and 5, move to node 5. f) Node 5 connects to program input 1. When the recursive process has finished, the genotype is fully decoded.

Each chromosome contains an equal number of nodes, and is treated as a genotype of an individual with a single program output. The inputs of each node encoded in a chromosome are only allowed to connect to the output of earlier nodes encoded in the same chromosome or any program input (terminals). This creates a form of compartmentalization in the genotype which supports the idea of removing the interconnectivity between the smaller problems encoded in each chromosome. An example of a Multi-chromosome CGP genotype is shown in Fig. 3.

3.2 Fitness Function and Multi-chromosome Evolutionary Strategy

The fitness function used in multi-chromosome approach is identical to the fitness function used in single chromosome approach, except for one small change. The output of each chromosome in multi-chromosome approach is calculated and assigned a fitness value based on the hamming distance from the perfect solution of a single output, whereas in CGP a fitness values is assigned to the whole genotype based on the hamming distance from the perfect solution over all the outputs (a perfect solution has a fitness of zero). Therefore, the multi-chromosome approach has n fitness values, where n is the number of program outputs, per genotype. This allows each chromosome in a genotype to be compared with the corresponding chromosome in other genotypes, by using a (1 + 4) multi-chromosome evolutionary strategy.

The (1 + 4) multi-chromosome evolutionary strategy selects the best chromosome at each position from all of the genotypes and generates a new best

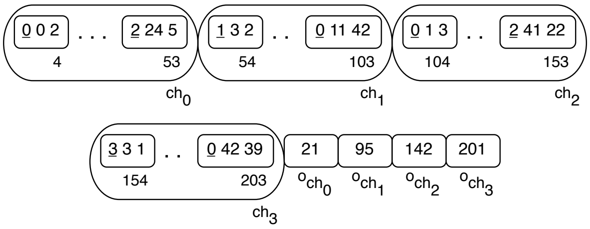

Fig. 3. A Multi-chromosome CGP genotype with four inputs, four outputs ($o_{c_0} - o_{c_3}$) and four chromosomes ($c_0 - c_3$), each consisting of fifty nodes

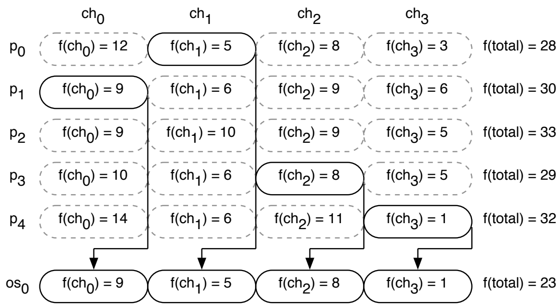

Fig. 4. The (1 + 4) multi-chromosome evolutionary strategy used in Multi-chromosome CGP. $p_{x,g}$ - parent x at generation g, c_y - chromosome y, $f(p_{x,g}, c_y)$ - fitness of chromosome y in parent x at generation g, $f(p_{x,g})$ - fitness of parent x at generation g.

of generation genotype containing the fittest chromosome at each position. The new best of generation genotype may not have existed in the population, as it is a combination of the best chromosomes from all the genotypes, so it could be thought of as a "super" genotype. The multi-chromosome version of the (1 + 4) evolutionary strategy therefore behaves as an intelligent multi-chromosome crossover operator, as it selects the best parts from all the genotypes. The overall fitness of the new genotype will also be better than or equal to the fitness of any genotype in the population from which it was generated. An example of the multi-chromosome evolutionary strategy is shown in Fig. 4.

4 Evolving a Prime Producing Formulae

4.1 Non-consecutive Prime Producing Formulae

The approach chosen for attempting to evolve integer coefficient polynomials (e.g. Euler's) was to assume that the polynomial was quadratic in the index value with a CGP genotype corresponding to each coefficient. Each genotype took the index value i as the only input. The primitive functions used were integer addition, subtraction, multiplication, protected division, and protected modulus. The CGP genotype was 300 primitives. One percent of all genes were mutated to create the offspring genotypes in a 1+4 evolutionary strategy (in which if any offspring were as fit at the best and there were no fitter genotypes, the offspring was always chosen). One hundred runs of 20,000 generations were carried out. The fitness of the polynomial encoded in the genotype was calculated by adding one for every true prime generated (for index values 0 to 49) that was bigger than the previous prime generated.

4.2 Consecutive Prime Producing Formulae

The aim of this experiment is to evolve a function $f(i)$, which is capable of producing consecutive prime numbers $p(i)$ for consecutive values of i. For example, $f(1) = 2$, $f(2) = 3$, $f(3) = 5$, $f(4) = 7$, etc. In this paper, we propose two approaches to evolving the polynomial $f(i)$; one treats $f(i)$ as an integer based function, while the other treats $f(i)$ as a binary based function.

An Integer Based Approach to the Prime Producing Polynomial. The first approach discussed uses CGP in a similar manner found in any symbolic regression approach [16]. The input of the CGP program is the i value, in the form of an integer, and the program output is the predicted prime number, $p(i)$, in the form of an integer. The function set used is similar to that used in many symbolic regression problems, comprising of addition, subtraction, multiplication, protected division and protected modulus. The CGP genotype is allowed 200 nodes, which can represent any of the functions in the function set. The fitness function used awards a point for every number produced which is a prime number and is in the correct consecutive position for the first 40 consecutive prime numbers.

A Binary Based Approach to the Prime Producing Polynomial. The second approach treats the polynomial $f(i)$, as a digital circuit problem, and uses multi-chromosome CGP to evolve a solution. Technically, the evolved solution will still be a polynomial, as any logical expression can be expressed in terms of a number of variables and the operators addition, subtraction and multiplication. Also, any input value i, when represented as a binary number, also forms a polynomial, $i = \sum a_j 2^j = a_n 2^n + a_{n-1} 2^{n-1} + ... + a_0 2^0$, where $0 \leq j \leq n$. Likewise, any prime number, $p(i)$, produced can also be represented as a binary number, and also forms a polynomial, $p(i) = \sum b_k 2^k = b_m 2^m + b_{m-1} 2^{m-1} + ... + b_0 2^0$, where $0 \leq k \leq m$. Therefore, we are trying to evolve a function $f(i)$, which given the coefficients of the binary number representing i, $a_0, ..., a_n$, produces the coefficients of the binary number representing $p(i)$, $b_0, ..., b_m$, where n does not have to equal m. An illustration of the process is shown in Fig. 5.

The function $f(i)$, which maps the coefficients of the input i to the output $p(i)$, is evolved using multi-chromosome CGP. The evolved program has n program inputs and m program outputs. In this case, $n = 14$, as this is the minimum number of inputs required to accept the number 10,000 in binary format and $m = 17$, as this is the minimum number of outputs required to produce the

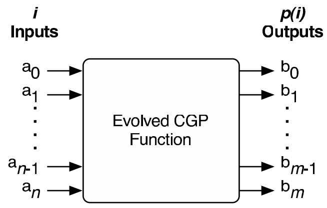

Fig. 5. The function mapping between the coefficients of the binary number representing the input, i, and the coefficients of the binary number representing the prime number output, $p(i)$

10,000th prime number. Each program output is taken from a separate chromosome in the genotype, therefore the genotype consists of m chromosomes. Each chromosome is an equal length and contains 300 nodes. The function set for the experiment simply contains a multiplexer which can choose either input in_0 or input in_1, as its output. The mutation rate used was 3% per chromosome.

As the set of test cases supplied for the GECCO competition was very large (10,000), and there was no guarantee a solution exists for all 10,000 test cases, or how much computational power would be required to find a solution, an incremental form of evolution was used. The evolved program starts off trying to find a solution to the first 16 test cases. If a solution is found, the run continues but the number of test cases is increased to 32. This evolutionary process continues, incrementing the number of test cases by 16 each time a solution is found, until a solution is found for all 10,000 test cases (a total of 625 increments).

In this paper, we are not actually benchmarking the performance of any of the techniques but we are using them for exploratory purposes, to see if any function can be discovered that is capable of predicting consecutive prime numbers.

5 Results and Discussion

5.1 Non-consecutive Prime Producing Polynomials

In the hundred runs, we obtained 6 Legendre polynomials and 5 Euler polynomials. The most common polynomial found was $2x^2 + 40x + 1$. This was found 57 times. The polynomial produces 47 primes for index values 0 to 49 but 17 is the longest sequence of primes. The most interesting solution obtained was the polynomial $x^2 - 3x + 43$. This produces primes for index values 0 to 42. This is a sequence of primes that is two primes longer than Euler or Legendre's polynomials. However, it has two repeats (the sequence begins 43, 41, 41, 43, 47, for index values 0,1,2,3,4). We could not find this polynomial in the literature (despite its simple form). When the number of generations was increased we found that the technique tended to converge on Euler of Legendre polynomials with much greater frequency (i.e. these polynomials are great 'attractors').

Further work was carried out in which polynomials were rewarded for having as large a sum of coefficients as possible (provided that they were equally good at producing long sequences of primes). We carried out 1000 runs of 40,000 generations with 200 primitives in each coefficient producing program (quadratics). The inputs to the coefficient producing programs were chosen to be 19, 47, 97, 139, and 193 respectively. The Euler polynomial was produced 142 times and the polynomial $2x^2 + 40x + 1$ (second best) was discovered 14 times. This approach was found to produce a much greater variety of polynomials, many of which produced long sequences of primes. Some examples are $8x^2 + 104x + 139$ (25) and $2848x^2 + 73478x + 227951$ (15), where the figures in brackets represent the length of the sequence of primes produced.

5.2 The Integer-Based Approach

The symbolic regression approach, was run independently ten times for 100,000 generations. The results of these runs can be shown in Table 1. ¿From the results, the best individual run was picked with a fitness of 27 out of the first 40 primes correct. This individual was evolved for a further 10 million generations, by which it had reached a fitness of 37 after 3,192,104 generations. Once again, the individual was evolved for a further 20 million generations. This time it had now reached a fitness of 39 after 16,336,784 generations. The individual still had not found all 40 consecutive prime numbers, so it was evolved further until it could correctly produce the first 40 prime numbers consecutively, which took a further 48,755,397 generations. The solution contained 88 active nodes out of the original 200 nodes and required 113,176,917 potential solutions to be evaluated in order to find this solution, indicating the difficulty of this problem.

As an extension to the experiment, the evolved solution was evaluated on the first 100 prime numbers (60 of which it had never been trained on) to see how well the solution generalised. The evolved solution found 21 prime numbers out of the 60 prime numbers it had never seen before. Some of the prime numbers found in the 21 prime numbers were in small groups whilst others were spread out. This indicates that the evolved solution not only found the first 40 consecutive prime numbers but also learnt something about what it means to be a prime number.

Table 1. The results of 10 independent runs of CGP trying to find the first 40 consecutive prime numbers

Run No.	Final Fitness	Generation Achieved	No. Active Nodes
0	16	8257	48
1	17	5666	36
2	13	2331	37
3	16	4234	34
4	17	4955	37
5	19	6261	42
6	16	3447	41
7	27	9944	57
8	18	6383	57
9	15	7305	52

5.3 The Binary-Based Approach

The digital circuit approach was run continuously, incrementing the number of test cases each time a solution was found. Evolved solutions were found for the first 16, 32, 48, 64, 80, 96, 112, 128, 144, 160, 176, 192 and 208 consecutive prime numbers. The evolved solution that produces the first 16 consecutive primes is shown in Equation 1.

$$p(i) = b_5 2^5 + b_4 2^4 + b_3 2^3 + b_2 2^2 + b_1 2^1 + b_0 2^0 \; where \tag{1}$$

$$\begin{aligned}
b_5 &= a_0 + a_1 a_2 - 2a_0 a_1 a_2 \\
b_4 &= -((a_1(2a_2 - 1) - a_2)(1 + a_2(a_3 a_4 - 1))) \\
&\quad + a_0(1 - 2a_2 + a_2^2(2 - 2a_3 a_4) + 2a_1(2a_2 - 1)(1 + a_2(a_3 a_4 - 1))) \\
b_3 &= a_2(a_3 + a_4 - 2a_3 a_4) + a_1 a_3(1 + a_2(2a_3 a_4 - a_3 - a_4)) \\
b_2 &= 1 - a_3 - a_4 + 2a_3 2a_4 - 2a_2^4(2a_3 - 1)3(a_4 - 1)a_4 \\
&\quad + 2a_3 a_4^2 - 2a_3^2 a_4^2 - a_2^2(2a_3 - 1)(a_3(2 - 6a_4) \\
&\quad + (3 - 2a_4)a_4 + 6a_3^2(a_4 - 1)a_4) + a_2^3(1 - 2a_3)^2(1 - (1 + 6a_3)a_4 \\
&\quad + (6a_3 - 2)a_4^2) + a_2(2a_3^3(a_4 - 1)a_4 + 2a_4^2 + a_3(2 + 5a_4 - 8a_4^2) \\
&\quad + a_3^2(1 - 9a_4 + 6a_4^2) - 1) + a_1(1 - a_2 a_3 + a_2^2(2a_3 - 1))(2a_3 \\
&\quad + 2a_4 - 1 - a_3 a_4 - 2a_3^2 a_4 + 2a_2^4(2a_3 - 1)^3(a_4 - 1)a_4 - 2a_3 a_4^2 \\
&\quad + 2a_3^2 a_4^2 + 2a_2^2(2a_3 - 1)(a_3(2 - 4a_4) - (a_4 - 2)a_4 \\
&\quad + 3a_3^2(a_4 - 1)a_4) - 2a_2^3(1 - 2a_3)^2(1 - (1 + 3a_3)a_4 + (3a_3 - 1)a_4^2) \\
&\quad - a_2(a_4 - 2 + 2a_3^3(a_4 - 1)a_4 + 2a_4^2 + a_3(4 + 4a_4 - 8a_4^2) \\
&\quad + 2a_3^2(1 - 5a_4 + 3a_4^2))) \\
b_1 &= 1 - a_3 + a_3^2 - a_2 a_3^2 - a_1(a_2(1 + a_3^2 + a_3(a_4 - 3)) \\
&\quad + a_3^2(1 - 2a_4) + a_2^2 a_3(2a_3 - 1)(a_4 - 1)) - a_3^2 a_4 + a_2 a_3^2 a_4 \\
&\quad + a_1^2(a_2 - 1)a_2 a_3(2a_3 - 1)(2a_4 - 1) \\
&\quad - a_0(2a_1 a_2 - 1)(a_3 - 1)(1 + (a_2 - 1)a_3(1 - a_4 + a_1(2a_4 - 1))) \\
b_0 &= a_2 - a_0(a_1 - 1)(a_2 - 1)(a_3 - 1) + a_1(a_2 - 1)(a_3 - 1) + a_3 - a_2 a_3
\end{aligned}$$

The solution producing 208 consecutive primes contained 400 active nodes and required 230,881,977 generations. A total of 923,527,909 potential solutions had to be evaluated, which required approximately three weeks of computing time on a PC with a single 1.83GHz processor and 448MB RAM. We believe that with enough computing power it would be possible to find a solution capable of predicting the first 10,000 prime numbers.

On examining the solutions, it can be observed that the more consecutive primes a solution can predict, the more active nodes the solution contains. The majority of the evolved solutions could not be included in this paper, as they were to large to print. Due to the sheer complexity of the solutions, we believe that it is highly unlikely that a human would ever devise such a solution, especially for the solutions producing high numbers of consecutive primes.

As the evolved solution for the first 16 prime numbers was capable of accepting inputs up to 31, we decided to extend the experiment to see how the solution generalised on 15 previously unseen inputs (just as we did with the integer-based approach). From the 15 unseen inputs, 7 of the predicted 15 outputs were prime numbers, which is just below 50%, indicating that the solution had learned something about "primeness" or favoured prime numbers. However, none of the 7 prime numbers produced from the 15 unseen inputs were consecutive.

6 Conclusion and Future Work

In this paper, we have presented an approach for evolving non-consecutive prime generating polynomials and also two different approaches using CGP for evolving a function $f(i)$, which produces consecutive prime numbers $p(i)$, for consecutive input values i. The best non-consecutive prime generating polynomial evolved produced 43 primes in a row (better than Euler's). Of the consecutive prime generating formulae, the symbolic regression approach using CGP, evolved a function capable of producing 40 consecutive prime numbers for input value i, where $1 \leq i \leq 40$. The digital circuit approach using multi-chromosome CGP, evolved multiple functions for consecutive sequences of prime numbers with increasing length, the longest of which produced 208 consecutive prime numbers, for input value i, where $1 \leq i \leq 208$. Although the second approach produced much larger sequences of prime numbers, the size of the solutions were enormous, in comparison with those produced by the first approach. In future work, once a solution is found, we intend to continue the evolutionary process with an altered fitness function, which minimises the number of nodes used. Therefore, making the solutions more compact. The downside of this approach is any generality evolved for solving further test cases could be lost.

The binary approach produced larger numbers of consecutive primes much easier than the integer-based approach, possibly indicating that by altering the search space from $\log_{10}$ to $\log_2$ has discovered a previously unknown relationship between the prime numbers. It is possible that by investigating other bases in the future, such as $\log_8$ or $\log_{16}$ could produce further links between prime numbers and help in discovering a function for prime prediction.

References

1. Wells, D.: Prime Numbers. John Wiley and sons (2005)
2. Euler, L.: Extrait d'un lettre de m. euler le pere a m. bernoulli concernant le memoire imprime parmi ceux de 1771. Nouveaux Mémoires de l'Académie royale des Sciences de Berlin, Histoire (1772) 35–36
3. Legendre, A.M.: Théorie des nombres. 2 edn. Libraire Scientifique A. Herman (1808)
4. Mollin, R.: Quadratics. Boca Raton (1995)
5. Mollin, R.: Prime-producing quadratics. American Mathematical Monthly **104**(6) (1997) 529–544

6. Pegg Jnr., E.: Math games: Prime generating polynomials
7. Fung, G., Williams, H.: Quadratic polynomials which have a high density of prime values. Mathematics of Computation **55** (1990) 345–353
8. Mollin, R.: New prime-producing quadratic polynomials associated with class number one or two. New York Journal of Mathematics **8** (2002) 161–168
9. Harrell, H.: Prime Producing Equations: The Distribution of Primes and Composites Within a Special Number Arrangement. AuthorHouse (2002)
10. Miller, J.F., Thomson, P.: Cartesian genetic programming. In: Proceedings of the 3rd European Conference on Genetic Programming (EuroGP 2000). Volume 1802 of LNCS., Edinburgh, UK, Springer-Verlag (15-16 April 2000) 121–132
11. Walker, J.A., Miller, J.F.: A multi-chromosome approach to standard and embedded cartesian genetic programming. In: Proceedings of the 2006 Genetic and Evolutionary Computation Conference (GECCO 2006). Volume 1., Seattle, Washington, USA, ACM Press (8-12 July 2006) 903–910
12. Koza, J.R.: Genetic Programming: On the Programming of Computers by Means of Natural Selection. MIT Press (1992)
13. Poli, R.: Parallel Distributed Genetic Programming. Technical Report CSRP-96-15, School of Computer Science, University of Birmingham, B15 2TT, UK (September 1996)
14. Yu, T., Miller, J.F.: Neutrality and the evolvability of boolean function landscape. In: Proceedings of the 4th European Conference on Genetic Programming (EuroGP 2001). Volume 2038 of Lecture Notes in Computer Science., Springer-Verlag (2001) 204–217
15. Vassilev, V.K., Miller, J.F.: The advantages of landscape neutrality in digital circuit evolution. In: Proceedings of the 3rd International Conference on Evolvable Systems (ICES 2000). Volume 1801 of Lecture Notes in Computer Science., Springer Verlag (2000) 252–263
16. Walker, J.A., Miller, J.F.: The automatic acquisition, evolution and re-use of modules in cartesian genetic programming. Accepted for publication in IEEE Transactions on Evolutionary Computation