Visualisation and Analysis of Genetic Records Produced by Cartesian Genetic Programming

Lukas Sekanina
Brno University of Technology
Faculty of Information Technology
IT4Innovations Centre of Excellence
Brno, Czech Republic
sekanina@fit.vutbr.cz

Vlastimil Kapusta
Brno University of Technology
Faculty of Information Technology
Brno, Czech Republic
v.kapusta@email.cz

ABSTRACT

Cartesian genetic programming (CGP) is a branch of genetic programming in which candidate designs are represented using directed acyclic graphs. Evolutionary circuit design is the most typical application of CGP. This paper presents a new software tool—CGPAnalyzer—developed to analyse and visualise a genetic record (i.e. a log file) generated by CGP-based circuit design software. CGPAnalyzer automatically finds key genetic improvements in the genetic record and presents relevant phenotypes. The comparison module of CGPAnalyzer allows the user to select two phenotypes and compare their structure, history and functionality. It thus enables to reconstruct the process of discovering new circuit designs. This feature is demonstrated by means of the analysis of the genetic record from a 9-parity circuit evolution. The CGPAnalyzer tool is a desktop application with a graphical user interface created using Java v.8 and Swing library.

CCS Concepts

•Human-centered computing → *Visualization toolkits;*

Keywords

Cartesian genetic programming, Digital circuit, Visualisation

1. INTRODUCTION

Visualisation methods have been adopted for understanding of the search process conducted by evolutionary algorithms (EA) since the introduction of EAs several decades ago. The role of visualisation has grown in importance with introducing genetic programming (GP) which is characterized by complex genotype-phenotype mappings, difficult search spaces and complex and often unusual phenotypes.

The most popular version of GP is tree-based GP. Various software tools capable of visualising the GP search and GP trees are available (see [6]). This paper deals with Cartesian GP (CGP) which is concerned with the automatic evolution of computational structures (such as mathematical equations, computer programs, digital circuits, etc.) encoded as graph structures [13]. For CGP, however, only a few design or visualisation software tools have been developed (see Section 3).

Evolutionary design of combinational circuits (i.e. digital circuits in which the output values only depend on current input values) is the most typical application of CGP. Circuits evolved and optimized by CGP showed in many cases a significant improvement (e.g. in the number of gates) in comparison with circuits optimized by the state of the art logic synthesis and optimization tools [19]. It is important to analyse and discover how these highly optimized circuits are constructed by CGP in order to improve heuristic algorithms used in conventional logic synthesis and optimization tools.

The goal of this paper is to present a new software tool—CGPAnalyzer—developed to analyse and visualise a genetic record (i.e. a log file) generated by a CGP-based circuit design software. CGPAnalyzer automatically finds key genetic improvements in the genetic record and presents phenotypes (circuits) constructed on the basis of the genotypes stored in the genetic record. The most important feature of CGPAnalyzer is a comparison module which allows the user to select two phenotypes and compare their structure, history and functionality. The truth table can be computed not only for the whole circuit but also for selected subcircuits which undergo a detailed analysis.

As the tool is devoted to small combinational circuits, it is assumed that the circuit response is analysed for all possible input combinations in the fitness function. The specification for CGP is thus given as a truth table of a multi-output Boolean function F:

$$F : \{0, 1\}^n \rightarrow \{0, 1\}^m, \quad (1)$$

where n is the number of primary inputs and m is the number of primary outputs.

The rest of the paper is organized as follows. Section 2 introduces the principles of CGP and the utilization of CGP in digital circuit design. A brief survey of available tools devoted to the evolutionary design and visualisation using GP and CGP is given in Section 3. The proposed CGPAnalyzer is presented in Section 4. Its functionality is demonstrated in Section 5, where a genetic record obtained from the evolu-

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

GECCO'16 Companion, July 20-24, 2016, Denver, CO, USA
© 2016 ACM. ISBN 978-1-4503-4323-7/16/07...$15.00
DOI: http://dx.doi.org/10.1145/2908961.2931740

tionary design of a 9-parity circuit is analysed. Conclusions are given in Section 6.

2. CARTESIAN GP

The standard CGP is a branch of GP in which candidate designs are represented using directed acyclic graphs. CGP is characterized by [11]:

- a simple encoding system in which a phenotype is encoded in a constant-size array of integers;
- a simple search method based on the $(1 + \lambda)$ evolution strategy;
- a single genetic operator – a point mutation.

2.1 Representation

A candidate circuit is modeled by means of a directed acyclic graph whose nodes are organized in n_c columns and n_r rows. Depending on a particular application, the nodes can be elementary logic functions, transistors or high-level components such as adders or multipliers. As we will deal with gate-level designs, the nodes will be n_a-input gates operating with 1 bit signals. The set of available gates Γ has to be defined for a given problem instance. The circuit utilizes n_i primary inputs and n_o primary outputs. Each node input can be connected either to the output of a node placed in previous l columns or to one of the primary circuit inputs, where l is one of CGP parameters.

The primary inputs and the outputs of the nodes are labeled $0, 1, \ldots, n_c \cdot n_r + n_i - 1$ and considered as the addresses which the node inputs can be connected to. A candidate solution consisting of two-input nodes (i.e. $n_a = 2$) is represented in the chromosome by $n_r \cdot n_c$ triplets (a_1, a_2, ψ) determining for each processing node its function ψ ($\psi \in \Gamma$), and the addresses of nodes a_1 and a_2 which its inputs are connected to. The last part of the chromosome contains n_o integers specifying the nodes where the primary outputs are connected to. While the chromosome size s is constant for a given product $n_r \cdot n_c$, n_a and n_o, i.e.

$$s = n_c n_r (n_a + 1) + n_o, \tag{2}$$

the phenotype (circuit) size is variable and measured as the number of active (i.e. used) nodes. See an example in Fig. 1, in which nodes 6, 8, 9 and 10 are not used.

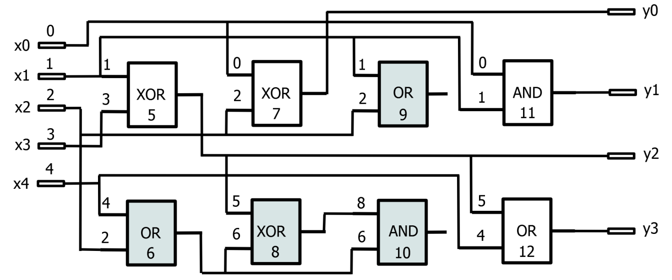

Figure 1: A candidate circuit represented by CGP with parameters: $n_i = 5, n_o = 4$, $n_c = 4$, $n_r = 2$, $l = 4$, $\Gamma = \{0^{\mathbf{and}}, 1^{\mathbf{or}}, 2^{\mathbf{xor}}\}$. **Chromosome: 1, 3, $\underline{2}$; 4, 2, $\underline{1}$; 0, 2, $\underline{2}$; 5, 6, $\underline{2}$; 1, 2, $\underline{1}$; 8, 6, $\underline{0}$; 0, 1, $\underline{0}$; 5, 4, $\underline{1}$; 7, 11, 5, 12. The nodes shown in grey are not used. The circuit implements logic functions:** $y_0 = x_0$ **xor** x_2**;** $y_1 = x_0$ **and** x_1**;** $y_2 = x_1$ **xor** x_3**;** $y_3 = (x_1$ **xor** $x_3)$ **or** x_4**.**

Algorithm 1: CGP

Input: CGP parameters, fitness function
Output: The highest scored individual p and its fitness

1 randomly generate parent p;
2 $Q \leftarrow \{p\} \cup \{\lambda$ offsprings created by mutation of $p\}$;
3 EvaluatePopulation(Q);
4 **while** $\langle$*terminating condition not satisfied*$\rangle$ **do**
5 $\quad \alpha \leftarrow$ highest-scored-individual(Q);
6 $\quad$ **if** *fitness*(α) $\geq$ *fitness*(p) **then**
7 $\quad\quad p \leftarrow \alpha$;
8 $\quad Q \leftarrow \{p\} \cup \{\lambda$ offsprings created by mutation of $p\}$;
9 $\quad$ EvaluatePopulation(Q);
10 **return** p, fitness(p);

2.2 Search method

CGP employs a $(1 + \lambda)$ search method whose pseudocode is given in Algorithm 1. The initial population Q of CGP is created either randomly or seeded by means of existing circuits. The next step consists of the evaluation of candidate circuits using the fitness function. Each member of Q then receives one fitness value and the highest-scored individual becomes a new parent of the next population. However, if two or more individuals can serve as the parent, the individual which has not served as the parent in the previous generation will be selected. This strategy is important because it ensures the diversity of population. From this parent, λ candidate solutions are generated using mutation. The termination criterion depends on a particular application.

Despite many attempts to propose a suitable crossover operator to CGP, the mutation is still used as the crucial genetic operator. The mutation operator modifies up to h randomly chosen genes (integers) of the chromosome. Their new values are generated randomly, but it is checked whether the new values are valid. One mutation can affect either the gate function, gate input connection, or primary output connection. A mutation is called neutral if it does not affect the circuit's fitness. If a mutation hits a non-used part of the chromosome, it is detected and the circuit is not evaluated in terms of functionality because it has the same fitness as its parent.

The encoding used in CGP is highly redundant as many nodes and node inputs can be disconnected from the phenotype. Moreover, there are usually many ways to implement a given logic function in a given CGP instance, for example,

$$y = not(a) = not(not(not(a))) = nand(a, a). \tag{3}$$

This redundancy together with a relatively powerful mutation operator is considered as a key feature of CGP allowing for an efficient circuit evolution. The role of redundancy and mutation have been analysed by many authors, for example, see [12, 5].

2.3 Fitness function

In the simplest case, the circuit responses are calculated for a set of test vectors. The goal is to minimize the difference between the obtained vectors and desired vectors. If the objective is to evolve a desired logic function from

scratch and minimize the number of gates, the fitness value of a candidate combinational circuit is defined as

$$f = \begin{cases} b & \text{when} \quad b < n_o 2^{n_i}, \\ b + (n_c \cdot n_r - z) & \text{otherwise,} \end{cases} \quad (4)$$

where b is the number of correct output bits obtained as response for all possible assignments to the inputs (with respect fo F), z denotes the number of gates utilized in a particular candidate circuit and $n_c \cdot n_r$ is the total number of available gates. It can be seen that the last term $n_c \cdot n_r - z$ is considered only if the circuit behavior is perfect, i.e. $b = b_{max} = n_o 2^{n_i}$. The second term can be modified to optimize other circuit parameters.

2.4 Evolutionary circuit design with CGP

Although various new designs have been discovered using CGP [11], the method is not directly applicable for the design of large combinational circuits because the fitness evaluation time grows exponentially with the number of primary inputs. Moreover, the number of evaluations can easily go to the millions, even for small (but nontrivial) circuits such as multipliers. This problem has partially been eliminated by introducing circuit decomposition techniques at the representation level [16] and formal verification methods in the fitness function [21]. CGP is usually capable of improving the best results of conventional optimization tools. For example, a 34% improvement was reported for 100 complex benchmark circuits in [19] if the task is to minimize the number of gates. Other successful applications of CGP have been proposed in domains in which candidate circuits are not evaluated using all possible input combinations (see e.g. hash functions [7]).

3. VISUALISATION IN EA

Visualisation techniques have been adopted for understanding of evolutionary algorithms for decades [15]. Current research in visualisation techniques for evolutionary computing is very active. It includes visualisation methods for complex genotypes, phenotypes, genetic operators, genotype-phenotype mappings, lineages, and dynamics of evolution in single objective EAs (e.g., [2]), parameter and decision spaces in multi-objective EAs (e.g., [18]), and parallel EAs (e.g., [10]).

In the case of genetic programming, a special focus is on visualising of the structure of trees [4] that occur during the evolution considering the fact that several thousands of nodes may be involved in each phenotype [3]. Visualization of genetic lineages and inheritance also helped in assembling genealogical and inheritance graphs of populations [1]. The whole spectrum of approaches can be found in [6].

EAs can evolve complex phenotypes such as programs, circuits, antennas, robotic bodies and artefacts. Analysing the development of phenotypes in the course of evolution and evolved designs at the end of evolution could reveal new design principles in engineering designs as exemplary demonstrated by Koza et al. [9]. An intuitive and interactive system capable of presenting the evolved design and design choices to the user is of great importance. Following this direction, for example, a 3D interactive method for linking EA decisions through time with the design of engineering systems was presented for water distribution network design in [8].

While many tools have been proposed for a fast prototyping of GA/GP-based applications (including visualisation), only a few tools exist to support the development of CGP-based applications. In addition to Miller's initial implementation of CGP [11], Turner and Miller have developed CGP-Library [17] which is a cross platform CGP library designed to be simple to use whilst being highly extendible. The library is written in C and is compatible with Linux, Windows and Mac OS. The CGP-Library supports standard CGP, recurrent CGP, and CGP for artificial neural networks.

Vasicek and Slany have proposed an online generator of accelerated fitness functions for the CGP-based combinational circuit evolution [23]. The method is based on translation of the CGP phenotype to a binary machine code that is consequently executed.

Pedroni developed a Java-based tool for evolutionary design using CGP [14]. All basic CGP parameters can be modified using a simple graphical user interface which also enables to visualize the population and select the active nodes in phenotypes.

Tools4CGP [20] is a set of tools implemented in C with the aim to run experiments with CGP and visualise CGP phenotypes (the cgpviewer tool). If the phenotype is a digital circuit, it can be simulated and exported as a source code in hardware description language (VHDL).

4. CGPANALYZER

The available tools devoted to CGP are capable of visualizing one phenotype and performing its basic analysis such as removing of unused components. In this paper, we present a new tool (CGPAnalyzer) developed to analyse the progress of evolution and the phenotypes created by CGP in the task of evolutionary design of small combinational circuits. CGPAnalyzer enables the user to:

- Read the whole record of CGP evolution, display the progress of the fitness and find important points during the evolution.
- Perform various visualisation functions over combinational circuits (phenotypes) such as active and inactive nodes visualisation, visualisation of the age of circuit components, and truth table visualisation.
- Perform simulation of candidate circuits and compute the truth table of selected subcircuit.
- Compare two phenotypes in terms of structure, history and functionality.

The CGPAnalyzer tool is a desktop application with a graphical user interface created using Java v.8 and Swing library.

When presenting and analysing CGP records in the task of circuit evolution, specific features of CGP have to be taken into account, particularly, the fact that CGP typically operates with a very small population (less than 10 individuals) and the number of generations is very high (in the order of millions) and thus many candidate circuits are produced. As most genetic operations have no effect on the functionality, the role of neutrality and redundancy is important. Functionality of the circuit (as well as its subcircuits) can easily be simulated, but the simulation can be time consuming or even intractable as the simulation time is exponentially depending on the number of inputs. The evolution has two

main phases which are (1) searching for a fully functional solution from a randomly created initial population and (2) optimizing circuits parameters (the number of gates in our case) when a fully functional solution is discovered.

The following subsections introduce the key functionality and features of CGPAnalyzer.

4.1 Input file

CGPAnalyzer accepts a log file generated in the course of the CGP evolution. The log file is in the format supported by Tools4CGP [20]. It contains the following information for every recorded generation:
Generation: k
Chromosome 1
Chromosome 2
...
Chromosome λ,
where k is the generation counter. Each chromosome has the following structure:
$\{f_c, \beta\}\{n_i, n_o, n_c, n_r, n_a, l, \beta\}$
$([n_i]a^{n_i}, b^{n_i}, c^{n_i}) \dots ([g]a^g, b^g, c^g)$
$(o_1, \dots, o_{n_o})$,
where the first part contains the fitness value (f_c), the number of active gates β and the standard CGP parameters. For each gate (with the index $n_i \dots g$, where $g = n_i + n_c.n_r - 1$), the gate index (denoted []), the input connection addresses (a, b) and gate function c are provided. Finally, n_o integers are devoted to the primary outputs.

In another file, functions.txt, the function set for a given log file is defined.

4.2 Analysis of the record of evolution

After reading the input file, CGPAnalyzer displays the development of the best fitness and corresponding phenotypes (Fig. 2). The generations in which the fitness score was improved are automatically identified (shown as red squares) and the user can navigate through the corresponding phenotypes, skipping thus (potentially uninteresting) generations. The fitness graph as well as the phenotypes can be zoomed in/out. The phenotype size is limited by the screen size.

4.3 Visualisation of a phenotype

For the phenotype chosen by the user, its array of $n_c \times n_r$ gates is displayed including the primary inputs, outputs, gate functions and gate interconnects. Unused gates and interconnects can be hidden in order to better understand the circuit structure. The histogram of used functions can be also displayed.

CGPAnalyzer computes the last generation in which each gene was modified by the mutation operator. Since there is the one to one mapping between genotypes and phenotypes in CGP, the last modification directly represents the age of each component (gate function, interconnect) of the circuit. Please note that this age is relative with respect to the generation which the chosen circuit belongs to. For example, let us assume that the chosen circuit is the best one from generation 100 and we are interested in one of its gates whose function (gene) is encoded at position 50 of the chromosome. If the last modification of this gene were done in generation 90 then its age would be 10. Now let us assume that the user selected the best circuit from generation 200 and the gene (at position 50) has not been modified since generation 100. Then the age of the gate function is 110.

The age is mapped onto (up to) 20 different colours and the determined colour is used to display a given circuit component. Moreover, the coloured area of the rectangle representing a particular gate is proportional to the age of the gate function. Only 20 different colours are used to be comfortably distinguishable by the user. This method enables to easily identify those parts of the circuits which were fixed many generations ago as well as the newly introduced changes. The visualization of circuit's history will be demonstrated using a concrete example in Section 5, Fig. 5.

4.4 Comparison of phenotypes

The user is allowed to select two phenotypes which can be compared. This unique feature enables to identify how the changes conducted at the level of genotypes are reflected in phenotypes and finally in the circuit behaviour. If the best phenotypes obtained in the course of CGP generations are analyzed and compared, the construction of the final phenotype from the initial population can be reconstructed.

In some cases, the evolution is seeded by a fully functional circuit and the goal is to minimize the number of gates. The proposed comparison utility is useful in revealing the tricks used by the evolution to improve the circuit.

In CGPAnalyzer, both circuits are displayed in such a way that differences in gate functions and interconnects are highlighted. The age of components of the first circuit can be determined with respect to the second circuit.

As all phenotypes are combinational circuits, they can be simulated to compute the truth table. This functionality is implemented in an innovative way. The truth table can be obtained not only for the whole circuit, but also for subcircuits. A subcircuit is defined by a gate (selected by the user) and all the gates directly or indirectly connected to the inputs of the chosen gate. The truth table is shown in a separate window. In order to illustrate this feature, Figure 3 shows the best circuits obtained during the evolution of a 3-input/8-output circuit in generations 19 and 39. The task was to compare subcircuits connected to the second output in both cases (in other words, the output of gate 23). These subcircuits are shown in orange color. Corresponding truth tables are given in a separate window (Fig. 4), where the input combinations for which the subcircuits give different output values are highlighted (red).

a	b	c	Y	Z
0	0	0	1	1
0	0	1	1	1
0	1	0	1	1
0	1	1	1	1
1	0	0	1	1
1	0	1	1	1
1	1	0	0	1
1	1	1	1	0

Figure 4: Comparison of truth tables for selected subcircuits (according to gate 23) in Figure 3.

5. CASE STUDY: 9-BIT PARITY

This section demonstrates the functionality of CGPAnalyzer when one record of the evolution of a 9-bit even parity circuit is analysed. The CGP parameters were $n_i = 9$, $n_o = 1$, $n_c = 6$, $n_r = 6$, $l = 1$, $n_a = 2$, $\Gamma = \{$and, or, xor,

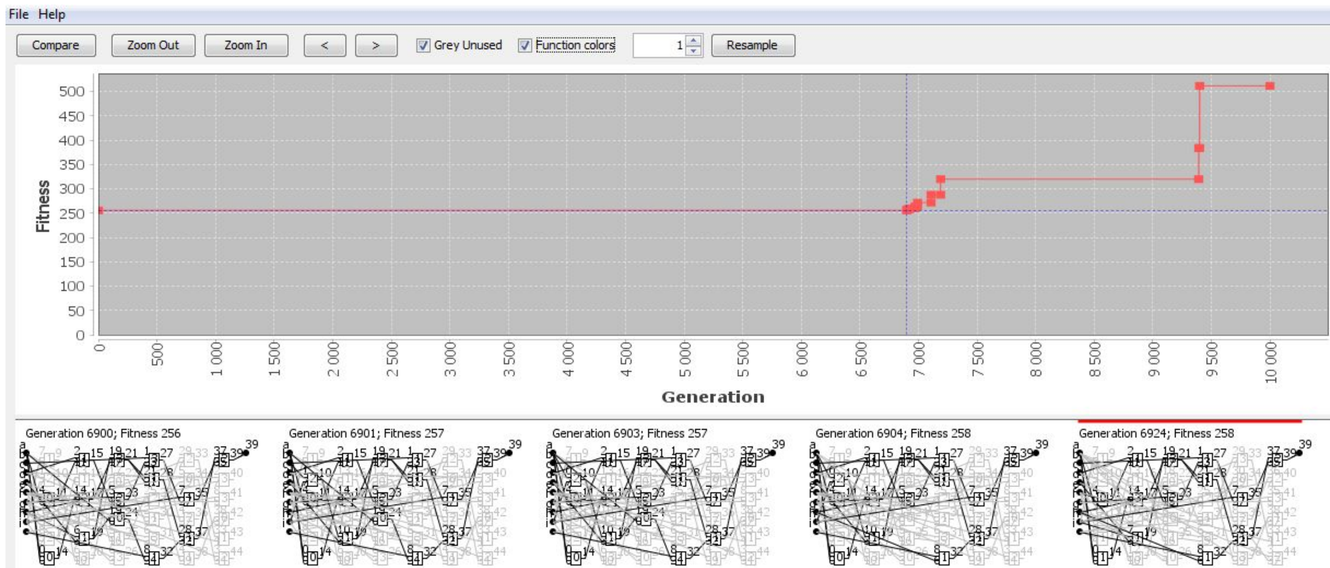

Figure 2: CGPAnalyzer application window showing the progress of the best fitness value (top) and best candidate circuits in selected generations (down).

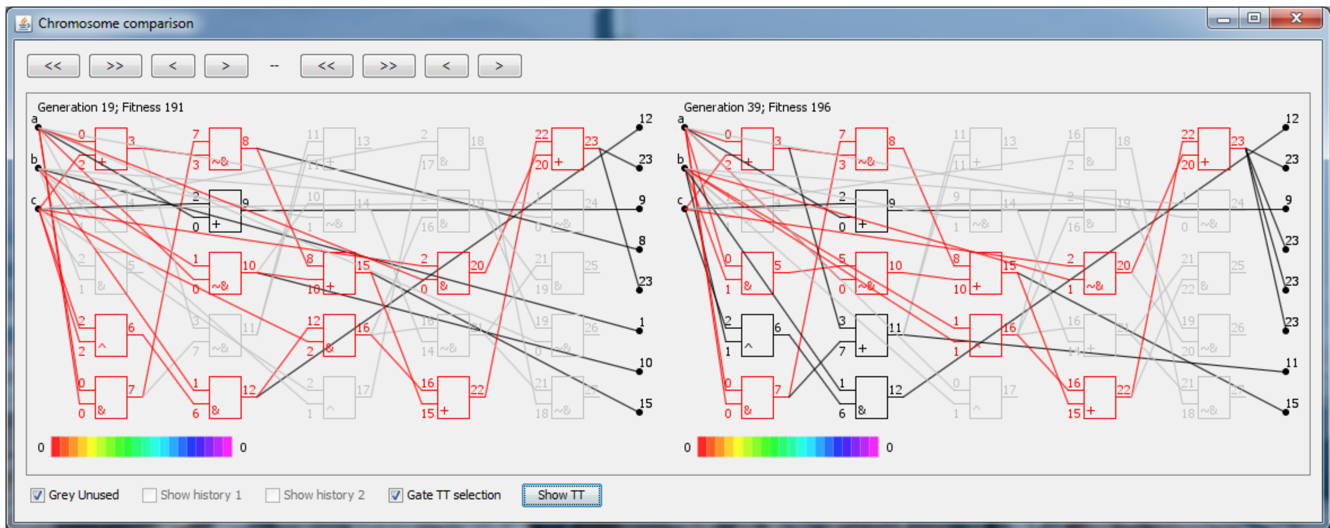

Figure 3: A comparison window: Two selected subcircuits are highlighted and compared according to gate 23 (selected by the user) which determines the particular subcircuits.

identity}, $\lambda = 4$, and 10 000 generations were conducted. The maximum fitness (in terms of the functionality) of a circuit computing the 9-bit parity is $f_{max} = 2^9 = 512$.

The development of the best fitness is shown in Fig. 2. The best circuit randomly created in the first population received the fitness value $f = 256$. There were 9 changes in the best fitness in the course of the evolution before reaching $f_{max} = 512$. The first improvement in the fitness ($f = 257$) from generation 6 901 indicates that the initial population contained circuits far from the desired one and it took thousands of generations, full of neutral or harmful mutations, before introducing a useful mutation which improved the fitness just by 1 point.

The analysis of the circuit with $f = 257$ in Fig. 5 revealed that all its components are no older than 397 generations, most of them were fixed a few generations before the generation 6 901. Please note that unused gates are shown in grey in Fig. 5.

The comparison function of CGPAnalyzer is very useful for understanding how the final phenotype was constructed. In order to demonstrate this feature, we will analyse selected changes of the best circuit in the progress of evolution. Figure 6 compares the best circuit from generations 7 107 ($f = 272$) and 7 108 ($f = 288$). The second input of gate 39 is reconnected from gate 35 to gate 38. By connecting a new subcircuit to (and disconnecting one gate from) the circuit, the fitness was improved by $288 - 272 = 16$ points. This illustrates a typical property of CGP that mutations can accumulate useful logic function in the unused part of the genotype which is then activated, connected to

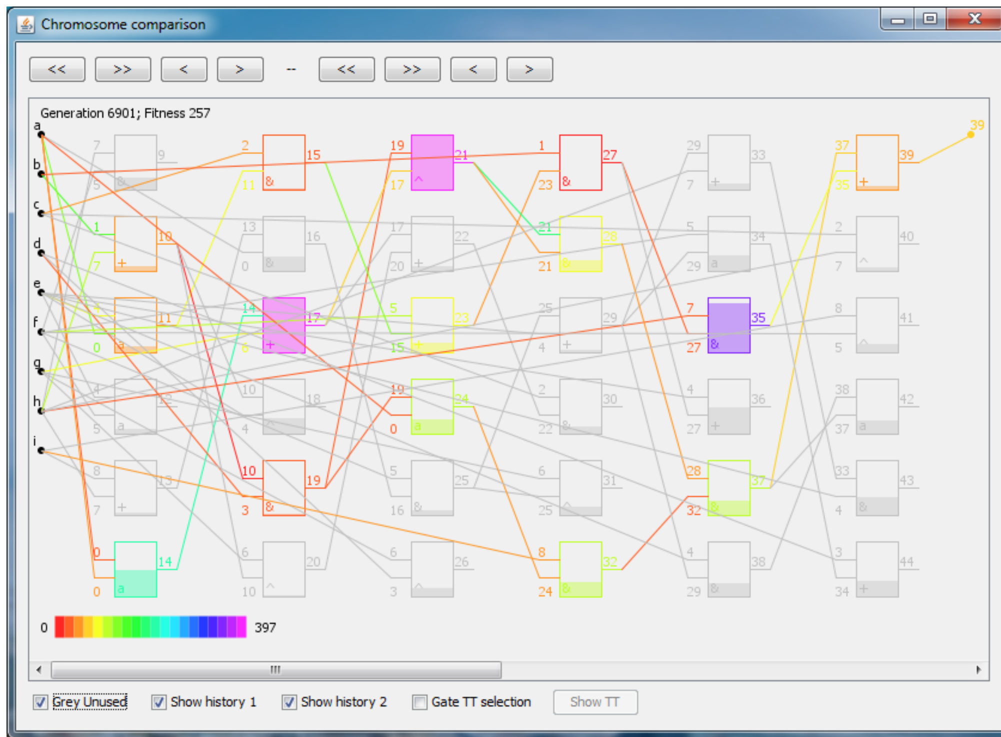

Figure 5: The age (history) of active circuit components shown in color. Red (0) is used for genes (components) created in the current generation. Unused gates are shown in grey.

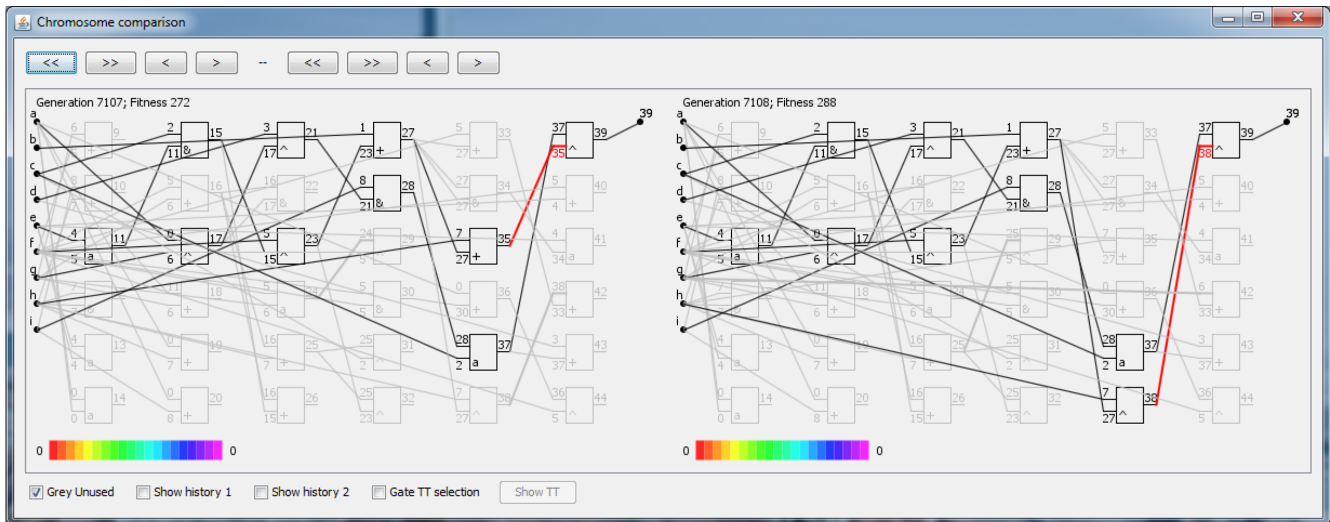

Figure 6: Comparison of the best phenotypes in generations 7 107 and 7 108.

the phenotype and the fitness is improved. Note that the age of components is not shown in Fig. 6.

Figure 7 compares the best circuit from generations 9 392 ($f = 320$) and 9 400 ($f = 512$). In can be seen that the fully functional circuit discovered in generation 9 400 differs in just two gate functions from the best circuit in generation 9 392, but the improvement in the fitness is huge ($521 - 320 = 192$ points).

After reaching the fully functional solution in generation 9 400, CGP used the remaining 600 generations to minimize the number of gates from 11 to 10. Figure 8 shows the age of circuit components in generations 9 400 and 10 000. As the age of active gates (and interconnect) is almost identical in both circuits (consider the difference of 600 generations) we can conclude that the last 600 generations were primarily spent by mutations in the unused part of the best chromosomes. Mutations hitting the active part of parent circuits were harmful or neutral.

Regarding the computing time, the reading of a 200 MB record of evolution requires approx. 8 seconds on the In-

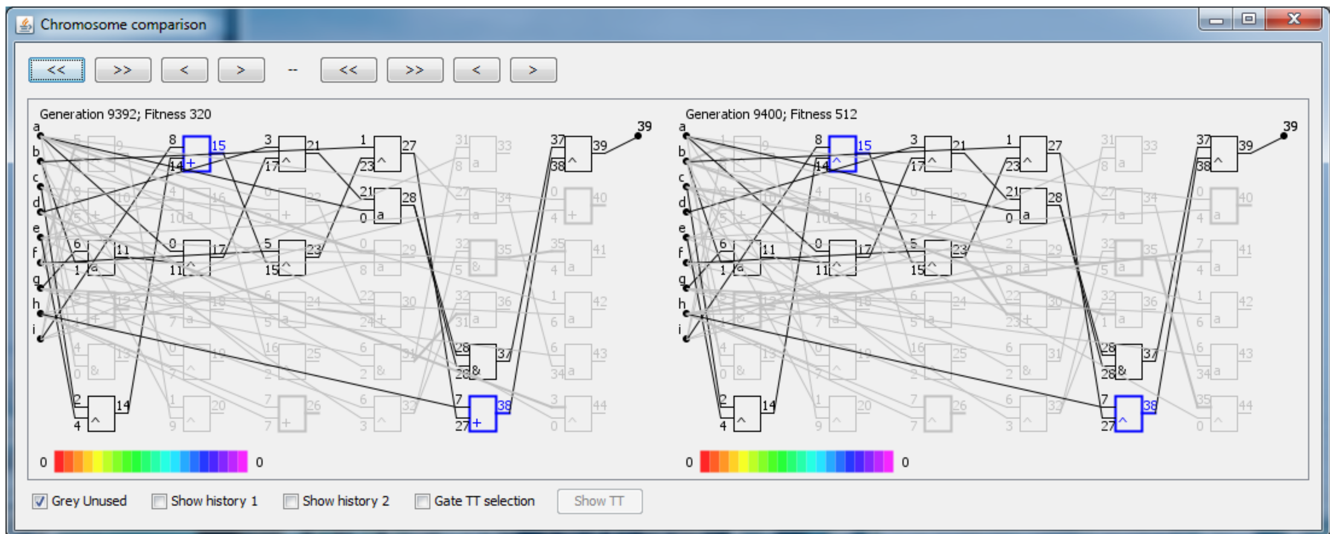

Figure 7: Comparison of the best phenotypes in generations 9 392 and 9 400.

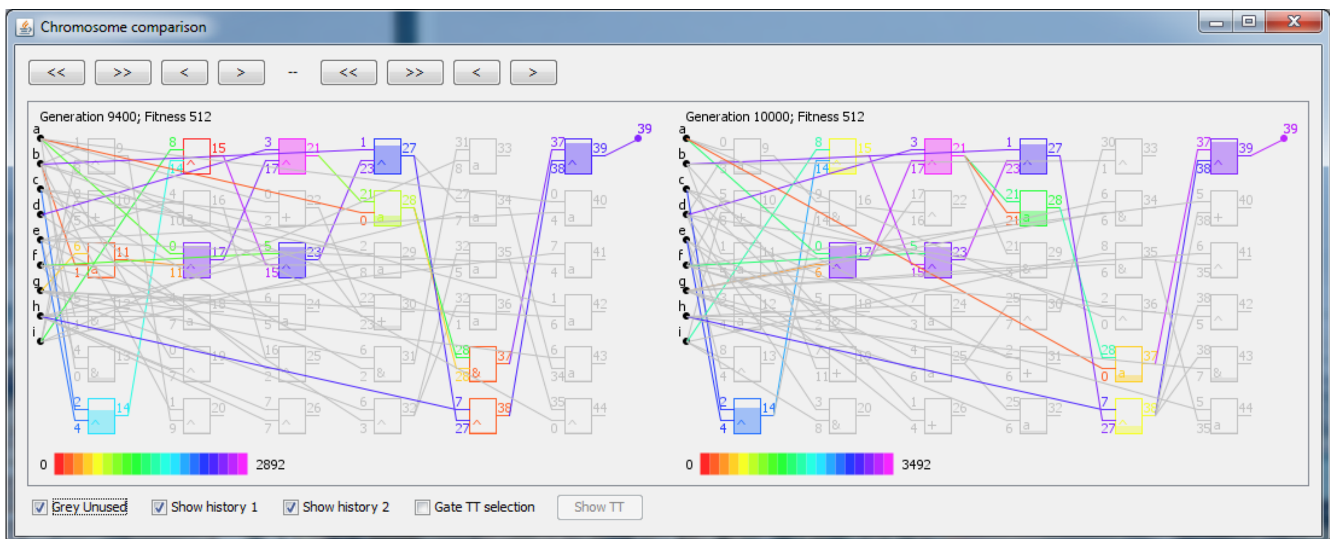

Figure 8: Comparison of the age of components in the best phenotypes in generations 9 400 and 10 000.

tel Core i5-3570 processor running at 3.40 GHz. All other operations of CGPAnalyzer represent no delay for the user.

6. CONCLUSIONS

In this paper, a new tool was presented which is capable of reading and analysing the genetic records generated in the process of evolutionary circuit design using CGP. The key functionality presented in the paper (but unseen in current software tools dealing with CGP) is an automatic identification of interesting genotypes (and thus phenotypes) in the genetic record and a comparison module allowing the user to select two phenotypes and compare their structure, history and functionality.

These features should help the designers to understand the underlying circuit optimization steps and reveal new optimization heuristics that could be applied in conventional circuit design and optimization tools and for (relaxed) Boolean equivalence checking methods, which have been developed in the area of approximate computing [22].

One of future directions for developing CGPAnalyzer lies in supporting a multi-objective scenario (with the objectives such as the area on a chip, delay and power consumption) which is typical for conventional circuit optimization tools.

Acknowledgment

This work was supported by the Czech science foundation project GA16-17538S and Brno University of Technology project FIT-S-14-2297.

7. REFERENCES

[1] B. Burlacu, M. Affenzeller, M. Kommenda, S. M. Winkler, and G. Kronberger. Visualization of genetic lineages and inheritance information in genetic programming. In *Genetic and Evolutionary*

Computation Conference, Companion Material Proceedings, pages 1351–1358. ACM, 2013.

[2] A. Cruz, P. Machado, F. Assunção, and A. Leitão. Elicit: Evolutionary computation visualization. In *Proceedings of the Companion Publication of the 2015 Annual Conference on Genetic and Evolutionary Computation*, pages 949–956. ACM, 2015.

[3] J. M. Daida, A. M. Hilss, D. J. Ward, and S. L. Long. Visualizing tree structures in genetic programming. *Genetic Programming and Evolvable Machines*, 6(1):79–110, 2005.

[4] A. Ekart and S. Gustafson. A data structure for improved GP analysis via efficient computation and visualisation of population measures. In *7th European Conference on Genetic Programming, EuroGP 2004, Proceedings*, volume 3003 of *LNCS*, pages 35–46. Springer-Verlag, 2004.

[5] B. W. Goldman and W. F. Punch. Analysis of cartesian genetic programming's evolutionary mechanisms. *IEEE Transactions on Evolutionary Computation*, 19(3):359–373, 2015.

[6] S. M. Gustafson, W. B. Langdon, and J. Koza. Bibliography on genetic programming, 2015. http://liinwww.ira.uka.de/bibliography/Ai/genetic.programming.html.

[7] P. Kaufmann, C. Plessl, and M. Platzner. EvoCaches: Application-specific Adaptation of Cache Mappings. In *Proceedings of the NASA/ESA Conference on Adaptive Hardware and Systems (AHS)*, pages 11–18. IEEE Computer Society, 2009.

[8] E. Keedwell, M. Johns, and D. Savic. Spatial and temporal visualisation of evolutionary algorithm decisions in water distribution network optimisation. In *Proceedings of the Companion Publication of the 2015 Annual Conference on Genetic and Evolutionary Computation*, pages 941–948. ACM, 2015.

[9] J. R. Koza. *Genetic Programming IV: Routine Human-Competitive Machine Intelligence*. Kluwer Academic Publishers, 2003.

[10] E. Lutton, H. Gilbert, W. Cancino, B. Bach, P. Parrend, and P. Collet. Gridvis: Visualisation of island-based parallel genetic algorithms. In *Applications of Evolutionary Computation: 17th European Conference, EvoApplications 2014*, pages 702–713. Springer Berlin Heidelberg, 2014.

[11] J. F. Miller. *Cartesian Genetic Programming*. Springer-Verlag, 2011.

[12] J. F. Miller and S. L. Smith. Redundancy and computational efficiency in cartesian genetic programming. *IEEE Transactions on Evolutionary Computation*, 10(2):167–174, 2006.

[13] J. F. Miller and P. Thomson. Cartesian Genetic Programming. In *Proc. of the 3rd European Conference on Genetic Programming EuroGP2000*, volume 1802 of *LNCS*, pages 121–132. Springer, 2000.

[14] E. Pedroni. JCGP, 2014. https://bitbucket.org/epedroni/jcgp/.

[15] T. Routen. Techniques for the visualisation of genetic algorithms. In *IEEE World Congress on Computational Intelligence. Proceedings of the First IEEE Conference on Evolutionary Computation*, pages 846–851 vol.2. IEEE, 1994.

[16] E. Stomeo, T. Kalganova, and C. Lambert. Generalized disjunction decomposition for evolvable hardware. *IEEE Transaction Systems, Man and Cybernetics, Part B*, 36(5):1024–1043, 2006.

[17] A. J. Turner and J. F. Miller. Introducing a cross platform open source cartesian genetic programming library. *Genetic Programming and Evolvable Machines*, 16(1):83–91, 2014.

[18] T. Tusar and B. Filipic. Visualization of pareto front approximations in evolutionary multiobjective optimization: A critical review and the prosection method. *IEEE Transactions on Evolutionary Computation*, 19(2):225–245, 2015.

[19] Z. Vasicek. Cartesian GP in optimization of combinational circuits with hundreds of inputs and thousands of gates. In *Proceedings of the 18th European Conference on Genetic Programming – EuroGP*, LCNS 9025, pages 139–150. Springer International Publishing, 2015.

[20] Z. Vasicek and L. Sekanina. Tools4CGP – tools for cartesian genetic programming, 2008. http://www.fit.vutbr.cz/˜ vasicek/cgp/tools/.

[21] Z. Vasicek and L. Sekanina. Formal verification of candidate solutions for post-synthesis evolutionary optimization in evolvable hardware. *Genetic Programming and Evolvable Machines*, 12(3):305–327, 2011.

[22] Z. Vasicek and L. Sekanina. Evolutionary design of complex approximate combinational circuits. *Genetic Programming and Evolvable Machines*, 17(2):1–24, 2016.

[23] Z. Vasicek and K. Slany. Efficient phenotype evaluation in cartesian genetic programming. In *Proc. of the 15th European Conference on Genetic Programming*, LNCS 7244, pages 266–278. Springer Verlag, 2012.