UNIVERSIDADE FEDERAL DE SANTA CATARINA
CAMPUS REITOR JOÃO DAVID FERREIRA LIMA
UNDERGRADUATE PROGRAM IN COMPUTER SCIENCE

Bryan Martins Lima

ADAPTIVE BATCH SIZE CGP: IMPROVING CGP LOGIC OPTIMIZATION FLOW

Florianópolis, Santa Catarina – Brazil
2023

Bryan Martins Lima

ADAPTIVE BATCH SIZE CGP:
IMPROVING CGP LOGIC OPTIMIZATION FLOW

Bachelor's Thesis submitted to the Undergraduate Program in Computer Science of Universidade Federal de Santa Catarina for degree acquirement in Bachelor of Science degree in Computer Science.
Supervisor: Jônata Tyska Carvalho, Phd.
Co-supervisor: Augusto André Souza Berndt, MSc.
Co-supervisor: Cristina Meinhardt, Phd.

Florianópolis, Santa Catarina – Brazil
2023

Ficha de identificação da obra elaborada pelo autor,
através do Programa de Geração Automática da Biblioteca Universitária da UFSC.

Lima, Bryan Martins
   Adaptive Batch Size CGP : Improving CGP logic optimization flow / Bryan Martins Lima ; orientador, Jônata Tyska Carvalho, coorientadora, Cristina Meinhardt, coorientador, Augusto André Souza Berndt, 2023.
   70 p.

   Trabalho de Conclusão de Curso (graduação) - Universidade Federal de Santa Catarina, Centro Tecnológico, Graduação em Ciências da Computação, Florianópolis, 2023.

   Inclui referências.

   1. Ciências da Computação. 2. Computação aproximada. 3. Cartesian Genetic Programming. 4. Síntese lógica. 5. Algoritmos evolutivos. I. Carvalho, Jônata Tyska. II. Meinhardt, Cristina. III. Berndt, Augusto André Souza IV. Universidade Federal de Santa Catarina. Graduação em Ciências da Computação. V. Título.

Bryan Martins Lima

ADAPTIVE BATCH SIZE CGP:
IMPROVING CGP LOGIC OPTIMIZATION FLOW

This Bachelor's Thesis was considered appropriate to get the Bachelor of Science degree in Computer Science, and it was approved by the Undergraduate Program in Computer Science of INE – Departamento de Informática e Estatística, CTC – Centro Tecnológico of Universidade Federal de Santa Catarina.

Florianópolis, Santa Catarina – Brazil, 04 of july of 2023.

ACKNOWLEDGEMENTS

Primeiramente, gostaria de agradecer minha família que me apoiou desde o início, principalmente meus avós paternos Ivonete e Adair, minha avó materna Alzira e aos meus pais, Simone e Christopher. Se não fossem vocês, eu não teria chegado até aqui.

Aos meus amigos que me aguentaram durante tanto tempo e continuam presentes até hoje, principalmente Flora, Victor, Gustavo, Natália, Julia, Beatriz e tantos outros. Mesmo que alguns tenham se distanciado, sempre que nos reencontramos é como se não nunca tivéssemos deixado de nos falar. Também gostaria de agradecer aos colegas do curso pela parceria nos diversos trabalhos em conjunto.

A minha parceira Bianca, que me acompanhou durante esse período e me aturou durante esse processo.

E por fim, aos professores, Dr. Jônata Tyska Carvalho e Dra. Cristina Meinherdt e coorientador Augusto Berndt pela orientação durante todo esse processo, mas principalmente pela paciência e colaboração durante o tempo de elaboração deste trabalho. Também gostaria de agradecer à minha colega do grupo de pesquisa Naiara que acompanhou e colaborou nos artigos e apresentações produzidas durante esse período.

RESUMO

Síntese lógica é responsável por produzir um modelo detalhado a nível lógico a partir da abstração de um circuito digital realizada por um *designer* de circuitos. Ela permite a automação na produção de circuitos lógicos e é fundamental na automação de projetos eletrônicos, transformando especificações de circuitos de alto nível em uma descrição de conexões de um circuito, visando manter o número de componentes lógicos o menor possível. Além disso, o principal objetivo da síntese lógica é implementar o "melhor" circuito possível. No entanto, o melhor circuito é composto entre o equilíbrio entre área, atraso e consumo de energia. O setor industrial precisa criar novas técnicas que acompanhem a crescente complexidade dos circuitos integrados. Considerando que técnicas de *Machine Learning* (ML) estão sendo utilizadas em diversas áreas do conhecimento devido a suas habilidades de reconhecimento de padrões, classificação e previsão, era uma questão de tempo até que elas fossem utilizadas no processo de síntese lógica. Essas técnicas têm sido utilizadas antes mesmo da construção física de circuitos lógicos, prevendo o consumo de energia e diretamente no processo síntese lógica. Uma dessas técnicas é o *Cartesian Genetic Programming* (CGP). O CGP é uma forma de programação genética, que se enquadra no campo dos algoritmos evolutivos, em que uma aplicação tem a capacidade de evoluir automaticamente. Esse algoritmo utiliza grafos para representar programas sendo chamado de "cartesiano" porque representa esses programas com uma matriz bidimensional. As abordagens baseadas em CGP apresentam vantagens nos processos de aprendizagem e otimização lógica. No entanto, o principal desafio dos fluxos baseados em CGP é o tempo de execução extenso quando comparado com outras estratégias de síntese lógica. Este trabalho propõe uma técnica, chamada *Adaptive Batch Size* CGP, que visa reduzir o tempo de execução necessário para o algoritmo aprender circuitos lógicos. A estratégia proposta aumenta dinamicamente o número de termos selecionados da tabela-verdade usada para avaliar os indivíduos durante o processo evolutivo. Ao avaliar a estratégia proposta aplicando-a a um *benchmark* com exemplares dos domínios de Aritmética, Lógica Aleatória e *Machine Learning*, os resultados indicam uma redução média de 53,65% no número de avaliações necessárias em comparação com o fluxo CGP padrão; e, apresentaram um aumento geral de 2,66% na precisão, enquanto nenhum circuito sintetizado apresentou uma diminuição estatisticamente significativa na precisão.

Palavras-chaves: Algoritmo evolutivo. Cartesian Genetic Programming (CGP). Síntese Lógica.

ABSTRACT

The main goal of logic synthesis of integrated circuits is to create a detailed logic-level model of a designers' abstraction of a circuit. It allows automation in the production of logic circuits and is crucial in electronic design automation, transforming high-level circuit specifications into a description of connections of a circuit, keeping the number of logical components as small as possible. Furthermore, the main purpose of logic synthesis is to implement the "best" circuit possible. However, the best circuit is composed of a trade-off between area, delay, and power consumption. There is a need by the industry to create novel techniques that can keep up with the growing complexity of integrated circuits. Considering that Machine Learning (ML) techniques are being used in a varied number of human and technological domains due to the pattern recognition, classification, and prediction capabilities, it was a matter of time before the techniques would be used in the logic synthesis process. ML techniques have been used even before the physical construction of logic circuits, predicting the power consumption and directly in the logic synthesis process. One such technique is *Cartesian Genetic Programming* (CGP). CGP is a form of genetic programming, which falls into the field of evolutionary algorithms, in which a computer program has the ability to evolve automatically. This algorithm uses graphs to represent programs and is called "Cartesian" because it represents these programs with a two-dimensional matrix. CGP-based approaches show advantages in the logic learning and logic optimization processes. However, the main challenge of CGP-based flows is the extensive runtime compared to other logic synthesis strategies. This work proposes a strategy, called Adaptive Batch Size CGP, that aims to reduce the runtime required for the algorithm to learn logic circuits. The proposed strategy dynamically increases the number of selected terms of the Truth Table used for evaluating the individuals during the evolutionary process. By evaluating the proposed strategy on a benchmark with exemplars from the Arithmetic, Random Logic, and Machine Learning domains, the results indicate an average reduction of 53.65% in the number of evaluations needed compared to the standard CGP flow; Furthermore, they presented an overall increase of 2.66% in accuracy, while no synthesized circuit presented a statistically significant accuracy decrease.

Keywords: Evolutionary algorithm. Cartesian Genetic Programming (CGP). Logic Synthesis.

LIST OF FIGURES

LIST OF TABLES

LIST OF ABBREVIATIONS AND ACRONYMS

ML	Machine Learning
CGP	Cartesian Genetic Programming
ABS	Adaptive Batch Size
EDA	Electronic Design Automation
SOP	Sum-Of-Products
POS	Products-Of-Sums
ESOP	Exclusive Sum-Of-Products
AIG	AND-Inverter Graph
DAG	Directed Acyclic Graph
LN	Logic Networks
XAIG	XOR-AND-Inverter Graph
MIG	Majority-Inverter Graph
LUT	LookUp-Table
RF	Random Forests
IWLS	International Workshop on Logic and Synthesis
DT	Decision Tree
GP	Genetic Programming
DG	Decision Graph
BS	Batch Size
CE	Change each
SMA	Simple Moving Average
AI	Adaptive Interval

LIST OF SYMBOLS

$\oplus$	Boolean operation XOR
$\wedge$	Boolean operation AND
μ	Number of parent genotypes
λ	Number of offspring genotypes
β	Initial batch size
σ	Adaptive interval
α	Batch increase factor

CONTENTS

1 INTRODUCTION

The number of chips that are present in everyday devices already surpasses thousands (HASSAN; HUMAIRA; ASGHAR, 2010), and the demand for new devices keep growing each day. Every year a new batch of smartphones and laptops is announced by the industry, and every year, designers promise better devices, with better performance and longer battery life. This promise has been constantly being fulfilled in the last decades.

Some of the challenges that integrated circuit designers are thrown upon are the development of circuits confined in small form factor, while being both performative and efficient - factors that are usually competitors. These high expectations are achievable, though are not exclusive, due to the process of synthesis of these devices. Usually, the design functionality is described in a high-level language such as programming languages (C and C++), or as specialized hardware descriptive languages, such as VHDL (TESTA; SOEKEN, et al., 2018). These high-level languages are important for describing the overall functionality of the circuit. However, they do not describe how each of the registers and logic gates should be arranged in order to minimize the circuit's area and delay, which impact its costs and performance. This step is done during the integrated circuit synthesis. The synthesis of a circuit involves some different steps before being able to be manufactured, such as architectural-level, logic-level and geometrical-level syntheses. From the view of the synthesis, we can abstract many of the complexities in the design of a sophisticate integrated circuit and consider that the first step in the synthesis of an integrated circuit is the logic synthesis.

The main goal of logic synthesis of integrated circuits is to create a detailed logic-level model of a designers' abstraction of a circuit. One of the steps of logic synthesis is the optimization process, which focuses on improving the overall circuit, i.e. performance and power efficiency. Some of the traditional logic optimization methods simplify a Boolean function by exploring exact logic minimization techniques, using the Algebraic methods and Boolean methods such as the Karnaugh map technique (KARNAUGH, 1953) and Quine-McCluskey method (QUINE, 1955). The Boolean methods manage to achieve better optimizations when compared to the Algebraic methods, however, they are more computationally intensive, and they are not as scalable in terms of the number of inputs. When considering the fast simplification of circuits with many inputs, Espresso is widely used for optimizing circuits by exploring suboptimal heuristic methods (RUDELL; SANGIOVANNI-VINCENTELLI, 1987)(BRAYTON et al., 1984), which provides a trade-off between computing performance and output quality.

There is an industry need to create novel techniques that can keep up with the growing complexity of integrated circuits (AMARÚ et al., 2017). Considering that Machine Learning (ML) is being used in a varied number of human and technological domains due to its pattern recognition, classification, and prediction capabilities (BURGES, 1998; JOACHIMS, 2005; BREIMAN, 1996), it was a matter of time before it would be used in the logic synthesis process. Along with this, the rising of green energy and environmental discussions pushes circuits into being more power-efficient. Thus, ML can be used in the design flow of digital circuits and in the EDA process (BEEREL; PEDRAM, 2018) to tackle these challenges and high expectations for integrated circuits. For example, decision trees and decision graphs are being used for fast logic optimization of circuits (ABREU et al., 2021; HUANG; JIANG, 2023). These new logic optimization flows are a promising alternative to the traditional methods, however, they do not scale well with the complexity of logic, as decision trees, usually, must expand to all input combinations. Neural networks are also being used in the logic optimization process (MIYASAKA et al., 2021), and evolutionary algorithms such as Cartesian Genetic Programming (BERNDT, A. et al., 2021; BERNDT, A. A. S. et al., 2022) have been recently studied; tough, they require a larger runtime when compared to other optimization techniques.

The evolutionary algorithm CGP was used in a logic optimization flow in the works of Augusto Berndt et al. (2021) and Augusto André Souza Berndt et al. (2022). The work explored the optimization process of logic circuits with two flows: (i) a pure CGP flow, where the algorithm searches for solutions based on random initial circuits; and, (ii) optimize solutions given by another optimization flow. The (ii) flow is very interesting as it can be used as an independent technological step in the logic optimization process, and addresses both accuracy and circuit size optimization. However, the main concern regarding the CGP search is the runtime required to learn these logic circuits when compared to other traditional and novel techniques.

The CGP search algorithm used by Augusto André Souza Berndt et al. (2022) is the first step as a CGP-based tool for the optimization process of digital circuits. The CGP algorithm utilizes Graph representation in order to encode computer programs. By using a graph representation, CGP already has advantages when considering other data structures such as trees. Each node of the graph can have multiple inputs and outputs, and they can be reused. These properties of Graph representation allow for solutions of linear size (HUANG; JIANG, 2023).

When considering error-tolerant applications, the approximate computing paradigm is a very attractive approach in order to speed up the design flow and take advantage of the intrinsic properties of these applications. By using this paradigm, it is possible to synthesize power-efficient solutions, as these applications have lenient accuracy requirements (BARUA; MONDAL, 2019). Furthermore, it is possible to use approximate computing to produce a circuit able to generalize based only on a few samples of an incompletely specified Truth Table.

In this context, using ML algorithms in the logic synthesis process can be a good alternative to traditional techniques, especially when considering approximate circuits.

CGP is one of these promising techniques that are still being researched. However, the long runtime required is a dealbreaker when considering the synthesis of everyday circuits. Thus, this work investigates strategies for reducing the runtime required for a CGP-based logic optimization flow. It is proposed a new technique called Adaptive Batch Size (ABS) CGP which aims to drastically reduce the number of evaluations required for synthesizing and optimizing circuits - therefore, reducing the runtime - while maintaining or improving the accuracy compared to the standard CGP. The proposed strategy dynamically increases the number of selected terms of the Truth Table used for evaluating the individuals during the evolutionary process. By evaluating the proposed strategy on a benchmark with exemplars from the Arithmetic, Random Logic, and Machine Learning domains, the results indicate a significant reduction in the number of evaluations needed compared to the standard CGP flow. Furthermore, they presented an overall increase in accuracy, while no synthesized circuit presented a statistically significant accuracy decrease.

1.1 OBJECTIVES

The main goal of this work is to propose and develop a strategy for speed-up a CGP-based logic optimization tool. The strategy, entitled Adaptive Batch Size CGP, explores the dynamic increase in the number of lines of the truth table used in the evaluation of the intermediate circuits to reduce the runtime required for learning logic circuits. The specific objectives of this work are as follows:

1.1.1 Specific objectives

1. Implement and analyze the proposed Adaptive Batch Size CGP strategy, which monitors the accuracy of the circuits during the search and exposes the algorithm to more lines of the Truth Table whenever the search is stagnant, broadening the search.

2. Evaluate the implemented strategy using a known benchmark and compare the results with the standard optimization process of the CGP.

1.2 SCOPE OF WORK

The scope of work consists of (i) a review of the state-of-the-art of logic synthesis, especially the optimization process, focusing on approximate computing and relating to Machine Learning, (ii) the development of a strategy to improve the CGP runtime, (iii) using a public benchmark to compare the developed strategy upon the standard CGP, (iv) present the results made in the Google Colab environment, and (v) discuss upon this work findings.

2 BASIC CONCEPTS AND RELATED WORK

In this chapter, some core definitions and basic concepts are presented for a better understanding of this work. Section 2.1 presents a concise review of the state-of-the-art in the logic synthesis domain, briefly detailing the logic synthesis history and future directions. Section 2.2 presents the paradigm of Approximate Computing and its emerging novel techniques used in the logic synthesis domain. And, in Section 2.3 it is presented the core concepts regarding Cartesian Genetic Programming, its applications, and its current state.

2.1 LOGIC SYNTHESIS AND THE OPTIMIZATION PROCESS

The main goal of logic synthesis is to create a detailed logic-level model of a designers' abstraction of a circuit, usually described in *hardware description language* (HDL) (MICHELI, 1994). Thus, logic synthesis will have as output the structural behavior of the circuits in terms of the logic-level. In this process, the circuit is analyzed and optimizations are made to reduce the number of gates, the logical depth, and the number of interconnections. These reductions will reflect on the power, performance, and area (PPA) characteristics of the implemented design. Thus, the main focus of logic synthesis is to generate the "best" implementation of the circuit. However, this is a difficult challenge that is bound by delay, power consumption, and the circuits' complexity, besides each specific application constraint. As such, logic synthesis is a crucial part in *electronic design automation* (EDA) helping the automation process of logic circuits (JIANG; DEVADAS, 2009). Considering that everyday integrated circuits are becoming more complex, there is a need to create innovative solutions to maintain the exponential growth of computer hardware (AMARÚ et al., 2017).

The work of Testa, Soeken, et al. (2018) reviewed the state-of-the-art logic synthesis flows and emerging strategies. One of the first logic synthesis algorithms was the Quine-McCluskey (QUINE, 1955) method, which solves the logic minimization of a function exactly. It used the Sum-Of-Products (SOP) as data structure, in which the algorithm focuses on reducing the number of product terms and literals of the function. The ESPRESSO program (RUDELL; SANGIOVANNI-VINCENTELLI, 1987; BRAYTON et al., 1984) also implemented the algorithm in the exact form and utilizing heuristics, specifically two-level forms heuristics. This second approach, with improvements upon the algorithm and suitable data structures, resulted in better-synthesized circuits, considering the computing time needed. With this, the ESPRESSO program managed to greatly impact the EDA industry. Meanwhile, ESPRESSO has its limitations, for example, the number of logic gates can be high in complex functions, which increases the circuit area. Hence, two-level optimization algorithms are mainly used in between steps of the logic synthesis flow, focusing on reducing the circuits' logic complexity. This strategy is a common design choice in EDA, which pushes novel techniques not limited by technological implementations - this will be further detailed in Section 2.2.

2.1.1 Main Data Structures for Logic Synthesis

The work Testa, Soeken, et al. (2018) also presented the main data structures used in the logic optimization process, which will be briefly commented upon in this work - for further detail, the work Testa, Soeken, et al. (2018) also gave implementation suggestions for each data structure presented. Truth Tables are one of the first data structures learned by a computer science undergraduate. It is a formal representation of a logic function that maps the output for all possible combinations of inputs. They are useful to introduce basic concepts to students; however, they are not a scalable data structure as the number of inputs of the logic function grows. Figure 1 presents the Truth Table for an XOR of 3 inputs, the inputs are highlighted in blue, and the output is highlighted in green. In any logic function, the total number of combinations is given by 2^n, where n is the number of inputs. For this simple function, the number of combinations for 3 inputs is 8. Considering that modern circuits can have hundreds or even thousands of inputs, Truth Tables are not a viable representation. Hence, they are mainly used for the simple verification of small logic functions or are converted to a more computationally friendly data structure in the logic synthesis process.

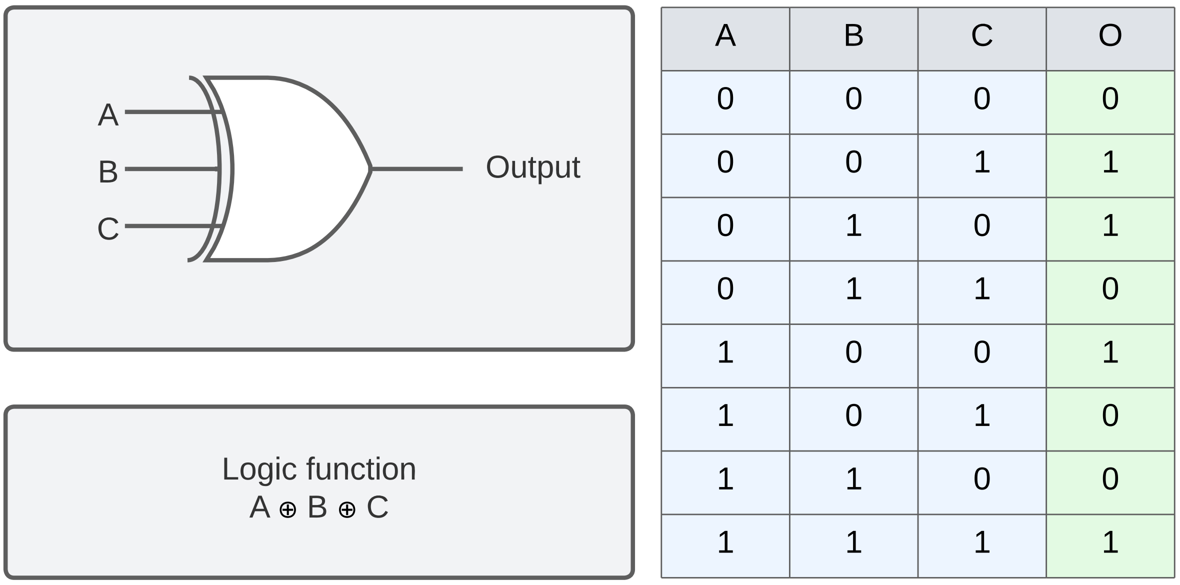
A Boolean function can be defined as $f : \mathbb{B}^n \rightarrow \mathbb{B}$, where n is the number of inputs and $\mathbb{B}$ denotes the Boolean domain (also written as the set $\{0, 1\}$). In the design flow of digital circuits, this function can be completely specified - i.e. the complete Truth Table is known, or only the *care-set*. The care-set represents only the specified output of a given combination of inputs. Conversely, the don't care-set corresponds to the combination of inputs for which the function outputs is either unimportant or not specified.

Another possible data structure is two-level representations, one of which is the sum-of-products (SOP). Equation 1 presents the logic function of an XNOR gate, which is already in the SOP format. Likewise, it is possible to describe the same logic function in products-of-sums (POS), detailed in Equation 2. Alternatively, Equation 3 presents the same function in the exclusive sum-of-products (ESOP) format.

$$f(A, B) = (A * B) + (!A*!B) \tag{1}$$

$$f(A, B) = (A+!B) * (!A + B) \tag{2}$$

$$f(A, B) = (A * B) \oplus (!A*!B) \tag{3}$$



A	B	C	O
0	0	0	0
0	0	1	1
0	1	0	1
0	1	1	0
1	0	0	1
1	0	1	0
1	1	0	0
1	1	1	1

Figure 1 – Truth Table for an XOR of 3 inputs.

Source: Bryan M. Lima, 2023

It is noteworthy that all these forms represent the same logic function. Therefore, depending on the application, each form can be more useful than the other. For example, in a CMOS circuit - which mainly uses NAND and NOR gates - XOR gates are more complex to implement. Thus, ESOP may not be a good design choice. This is not always the case, as that may be an ESOP form of a logic function that uses fewer logic gates which can result in less circuit area, thus, reducing the circuit costs. This simple scenario illustrates some of the challenges of the designer when implementing a logic circuit.

AND-Inverter Graphs (AIGs) (MISHCHENKO; CHATTERJEE; BRAYTON, 2006) can also be used as data structure. They are a type of Directed Acyclic Graph (DAG) in which nodes represent AND gates, and their connections can be directed or inverted. Figure 2 presents the AIG representation with the same XNOR logic function; Equation 4 represents this logic function written with only ANDs and Inverters.

$$f(A, B) = !(!(A * B) * !(!A * !B)) \tag{4}$$

AIGs are a form of representation that categorizes Logic Networks (LN), there are other types of LN representations, such as XOR-AND-Inverter Graphs (XAIG) (HÁLEČEK; FIŠER; SCHMIDT, 2018), Majority-Inverter Graphs (MIGs) (AMARU; GAILLARDON; DE MICHELI, 2015) and lookup-tables (LUTs) (RIENER et al., 2019). Binary decision diagrams (BDD) (BRYANT, 1986) can also be used as a data structure, likewise, it is a type of DAG with the main advantage of avoiding redundancy for representing the logic function.

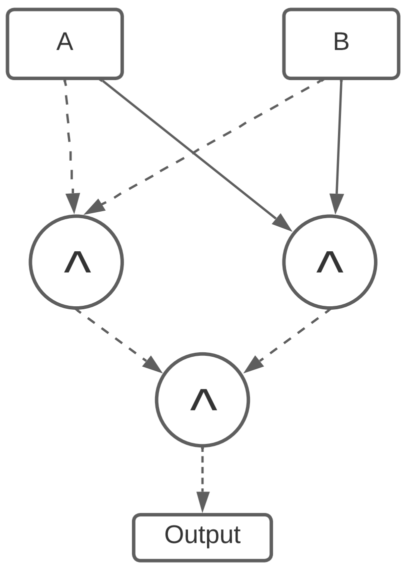


Figure 2 – XNOR function represented with AIG.

Source: Bryan M. Lima, 2023

Considering the mentioned data structures, AIG is the state-of-the-art data structure (RIENER et al., 2019) and is widely used in the logic synthesis flow. Moreover, it is used in the popular synthesis and verification tool ABC (BERKELEY LOGIC SYNTHESIS AND VERIFICATION GROUP, 2019) and it can be easily converted from Truth Tables. Furthermore, it is compatible with CGP; hence, this work utilizes AIG as the data structure in the implementation process.

2.1.2 Optimization Process in Logic Synthesis

There are two main types in the optimization process in logic synthesis: the Algebraic methods and Boolean methods (TESTA; SOEKEN, et al., 2018). Algebraic methods utilize two-level representations, such as SOP, POS, and ESOP, and treat them as a polynomial. Some of the applied techniques are:

1. Extraction: it rewrites the expression by searching for common variables in order to simplify the complete expression (BRAYTON, 1982);
2. (Re)substitution: it reuses the output of a sub-expression as input for another sub-expression (BRAYTON, 1982);
3. Algebraic rewriting: it uses algebraic axioms in order to rewrite a part of the expression with an equivalent one in order to improve a specific goal, such as power, area, and delay (RIENER et al., 2019).

As for the Boolean methods, they use the properties of Boolean algebra and the *don't cares* for modifying the original function in order to improve the overall circuit

(TESTA; AMARÚ, et al., 2020). Some of them are:

1. Resubstitution: similar to the Algebraic method, which aims to reuse the output of an expression into another - in this case, this is more computationally expensive due to the inherited properties of Boolean algebra (TESTA; SOEKEN, et al., 2018);
2. Rewriting: it replaces part of the expressions with smaller ones focusing on reducing the number of nodes of the circuit (RIENER et al., 2019);
3. Balancing: which optimizes the logic representation by reducing the number of logic levels, thus, reducing the circuit's delay (MISHCHENKO et al., 2011).

Considering both methods, the Algebraic methods are faster and can be used with a high number of inputs. However, they do not produce the best optimizations when compared to Boolean methods, which can achieve better results, as they work with the intrinsic properties of Boolean algebra. In spite of that, they are more computationally intensive compared to the Algebraic methods, and they cannot deal with a higher number of inputs.

2.2 MACHINE LEARNING AND APPROXIMATE COMPUTING APPLIED TO LOGIC SYNTHESIS

With Machine Learning (ML) trending in numerous technological areas due to pattern recognition, classification, and its prediction capabilities (BURGES, 1998; JOACHIMS, 2005; BREIMAN, 1996), it was a matter of time until ML was applied to the logic synthesis domain. Besides this, the discussion upon green energy and the environment (BARUA; MONDAL, 2019) attached to the increasing complexity of digital circuits and error-tolerant applications calls for novel techniques in logic synthesis for generating better overall circuits - more efficient and cheaper to produce as well as more powerful. These high expectations of the future of digital circuits push new techniques to meet these beliefs. When considering error-tolerant applications - such as media processing (audio, video, image, etc), sensors for the Internet of Things, and Neural Networks (VENKATESAN et al., 2011; HAN; ORSHANSKY, 2013) - there are intrinsic factors that to produce an exact circuit is unattainable. These factors include but are not limited to (i) the limitations of human perceptions - e.g. vision and hearing, (ii) an impossible perfect (golden) output - e.g. web search and recommendation algorithms - and (iii) noisy inputs (VENKATESAN et al., 2011; HAN; ORSHANSKY, 2013).

In this context, ML can be used in the design flow of digital circuits and in the EDA process (BEEREL; PEDRAM, 2018). There are numerous forms to apply ML to the flow of digital circuits, some of them are: recognizing the functionality of the circuit, which can be used for formal verification, reverse engineering, and analysis of the circuit's security (FAYYAZI et al., 2019); using ML to identify the best classical logic optimizer for a given portion of the Boolean function (NETO et al., 2019); and, more related to this work, when using directly in the logic synthesis process while synthesizing approximate circuits (SCARABOTTOLO et al., 2020).

Currently, the main challenges regarding approximate computing are (i) how much approximation can be used in a given circuit - e.g. approximation in a plane engine system is probably not a good idea, but a few wrong degrees on the output of the plane weather forecast system could be made, and (ii) how multiple approximate circuits behave together when they depend on each other (BARUA; MONDAL, 2019). For example, the MACACO methodology (VENKATESAN et al., 2011) is an initial step to integrate approximate computing to the current industry as it provides analysis and comparisons when it is given the correct implementation - e.g. error distribution, error probability, and average case error.

However, there are still some challenges when it comes to the automation of approximate circuits, as it still requires manual interventions, requiring the programmer to manually annotate sections of the code that can be approximated (BARUA; MONDAL, 2019). For example, Axilog (YAZDANBAKHSH et al., 2015) is a framework for Verilog HDL that introduces new annotations to the language, as well as analysis of the circuit, in an effort to abstract the approximate computing paradigm from the programmer and decreases manual intervention. The framework FlexJava(PARK; ESMAEILZADEH, et al., 2015) aims to reduce the number of annotations when compared to the state-of-the-art EnerJ (SAMPSON et al., 2011) framework for the Java programming language. It is able to achieve this by using its own compiler with approximation analysis. Another approach for this problem is the ExpAX (PARK; ZHANG, et al., 2014) programming framework, in which the programmer sets how much approximation is tolerable and the framework infers which operations can be approximated. These works indicate that approximate computing is not only useful for hardware but for software as well. Nevertheless, there is a need to automate this process for further adherence by the industry.

When it comes to approximate computing, there are five domains according to (BARUA; MONDAL, 2019): i) approximate instruction processing, ii) approximate communication, iii) approximate hardware systems and circuits, iv) approximate cloud computing, and v) approximate mobile computing. The approximate instruction processing domain consists of compilers and toolchains, similar to the frameworks mentioned above. For the present work, we will focus on the iii) domain: the approximate hardware systems and circuits domain, which is related to the logic synthesis process. For a review of the state-of-the-art for the other domains, see the work of Barua and Mondal (2019).

In this work, when referring to approximate computing, it is referring to the approximate hardware systems and circuits category. Considering this, approximate computing can be defined as a paradigm to speed up the design flow of digital circuits and take advantage of the intrinsic factors of error-tolerant applications to produce power-efficient ICs. In this context, can ML be used in logic synthesis for its pattern recognition, classification, and prediction capabilities? This was the final question upon the review of the results of the International Workshop on Logic and Synthesis (IWLS) in 2020 (RAI; AL., 2021). The authors proposed a competition for participants to provide solutions for 100 incompletely specified logic functions. As it was only given a portion of the care-set, the participants needed to find a solution that was able to generalize upon unknown outputs for a given combination of inputs.

The previously presented definition of care-set and don't care-set can be extended to treat this problem as an ML problem, where the care-set is split into the validation set and the training set. By training an algorithm to learn the training set, it is possible to analyze the generalization capability of the synthesized circuit when evaluating its performance with the validation set. It is presumed that if the circuit can generalize upon the validation set, then it should also achieve a decent accuracy upon unknown input combinations; which is vital for approximate circuits. Thus, now, a Boolean function can be described by the care-set, the don't care-set, and the unknown-set.

The main findings of the IWLS 2020 contest are reported in the work by Rai and al. (2021): there is no single solution pointed out as the best for learning logic. It is noteworthy that the benchmark set composed of 100 exemplars is made up of three different problem domains: Arithmetic, Random Logic, and Machine Learning. Hence, there was not a dominant solution that achieved the best results, considering logic learning accuracy, across all domains. For example, the winning team used a combination of strategies - ESPRESSO, LUT networks, and Random Forests (RF) - using the best approach for each function. RF and Decision Trees (DTs) were explored among eight of the ten teams, mainly because this approach presented a strong generalization capability. Furthermore, the results presented in (RAI; AL., 2021) indicate that RF performs better while increasing the number of AND nodes, which is not desirable for logic circuits. Beyond this, there were indications that by compromising a little accuracy, the solutions could drastically reduce the number of AND nodes.

Differently from the other teams, one of the approaches in the contest was the use of Cartesian Genetic Programming (CGP) by team 9 (BERNDT, A. et al., 2021). They proposed a logic optimization flow with CGP with two flows (i) using CGP as a search algorithm using random individuals, or (ii) using a bootstrapped solution to improve its accuracy and/or size. This technique is interesting as it can be used as a technology independent stage in the logic synthesis process, as it can have a previous logic synthesis output in an AIG format as input and improve it, or receive a Truth Table description and find a solution. Furthermore, the proposed flow can modify the output according to the designers' intentions - improving accuracy or size.

The next section will explain the basic concepts of Cartesian Genetic Programming, its current status, and its challenges; moreover, it will be further discussed its impact in the logic synthesis field.

2.3 CARTESIAN GENETIC PROGRAMMING

The general form of Cartesian Genetic Programming (CGP) was proposed in 2000 by Julian F. Miller and Peter Thomson (MILLER; THOMSON, 2000). It is a form of Genetic Programming (GP) that utilizes Graph representations as opposed to the standard GP proposed by (KOZA, 1994) which uses a tree as a data structure. By using a graph-based representation, CGP has advantages when compared to the traditional GP tree-based form. Graphs implicit allow reusing nodes and each node can have multiple inputs and outputs (MIMO), which results in solutions of linear size, compared to exponential size solutions by using tree-based representations (HUANG; JIANG, 2023).

Furthermore, CGP does not suffer bloat, i.e., a phenomenon in which the result does not improve while the solutions keep growing in size when compared to the traditional GP. This is mainly because CGP has a fixed value for the number of nodes, whereas, in the traditional GP, the solutions can keep growing until the computer runs out of memory. Though, currently, there is no formal explanation as to why CGP is not affected by this phenomenon (MILLER, 2019).

The graph representation of CGP is a type of Directed Acyclic Graph (DAG) and it can be used to represent math equations, circuits, and computer programs. CGP is inspired by concepts from the genetic field of Biology - such as mutation, genotypes, fitness, and generations - and applies them to programs capable of improving themselves through time. It is called 'Cartesian' as the formal definition of CGP represents the graphs using a two-dimensional grid.

There are many applications where CGP is used; it has been used in image filters and image processing (HARDING; LEITNER; SCHMIDHUBER, 2013; SEKANINA et al., 2011; HARDING, 2008), it can be used to encode Artificial Neural Networks (KHAN; AHMAD, et al., 2013; KHAN; KHAN; MILLER, 2010) and to optimize the learning of Convolutional Neural Networks (SUGANUMA et al., 2020), as well as in cryptography field (PICEK; JAKOBOVIC, et al., 2016; PICEK; CARLET, et al., 2016)[1]. More related to this work, one of the first usages of CGP for approximate logic synthesis was proposed by Vasicek and Sekanina (2014) where it was a novel technique at the time, and showed promising results, though the execution time was already a concern. Recently, CGP is being researched and applied to the logic optimization flow (BERNDT, A. et al., 2021; BERNDT, A. A. S. et al., 2022), which is used between the optimization steps. However, the main disadvantage when compared to other approximate techniques is still the runtime (BERNDT, A. A. S. et al., 2022).

[1] For further discussion on its applications see the works of Miller (2019) and Manazir and Raza (2019).

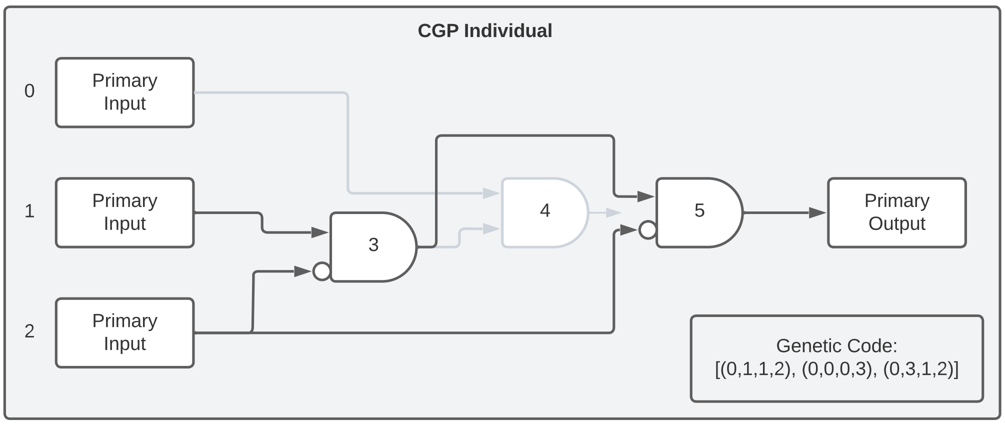


Figure 3 – CGP individual representation.

Source: Bryan M. Lima, 2023

2.3.1 Representation

In this subsection, it is presented the CGP representation used in the optimization of logic functions to illustrate the search algorithm used in this work. Figure 3 shows the representation of a CGP individual as a 1-line array, instead of the traditional 2-dimensional grid. The 1-line array was selected as it has faster convergence in the evolutionary search as demonstrated by Milano, Pagliuca, and Nolfi (2019); though, both representations are equivalent for digital circuits, as they can be represented both ways. The 1-line array representation is an array of 4-tuples composed of two pairs, one pair for each input of the logic gate. The inputs and the individual nodes are indexed; hence, there are 6 indexes representing 3 primary inputs and 3 nodes. The first pair (a, b), represents the first input, in which a represents if the input is inverted or not, and b represents from which index the input is from; similarly, the second pair represents the second input.

Taking node 3 as an example, the first input is not inverted, and it comes from the primary input 1, which is indexed as 1; thus, the first pair is $(0, 1)$. For its second pair, the input is inverted, and it comes from the primary input 2, indexed as 2; therefore, the second pair is $(1, 2)$. Combining both pairs, the complete genetic code for node 3 is $(0, 1, 1, 2)$. There are some considerations about this representation: (i) the input of a given node can be a primary input, i.e. (0, 1, 2), or an output of another node, i.e. (3, 4, 5); (ii) backward connections are not allowed, e.g. node 4 can receive inputs from node 3, but not from node 5. Moreover, there is an important concept regarding the CGP representation presented in Figure 3, that is the term functional node. Note that, in this simple example, node 4 is represented in gray color in Figure 3. This is because it is not a functional node, i.e. it does not have a connection to the output of the CGP individual. The functional part of the CGP individual is called the *phenotype*, and the genetic code, is called *genotype*. The number of primary inputs, primary outputs, and the number of

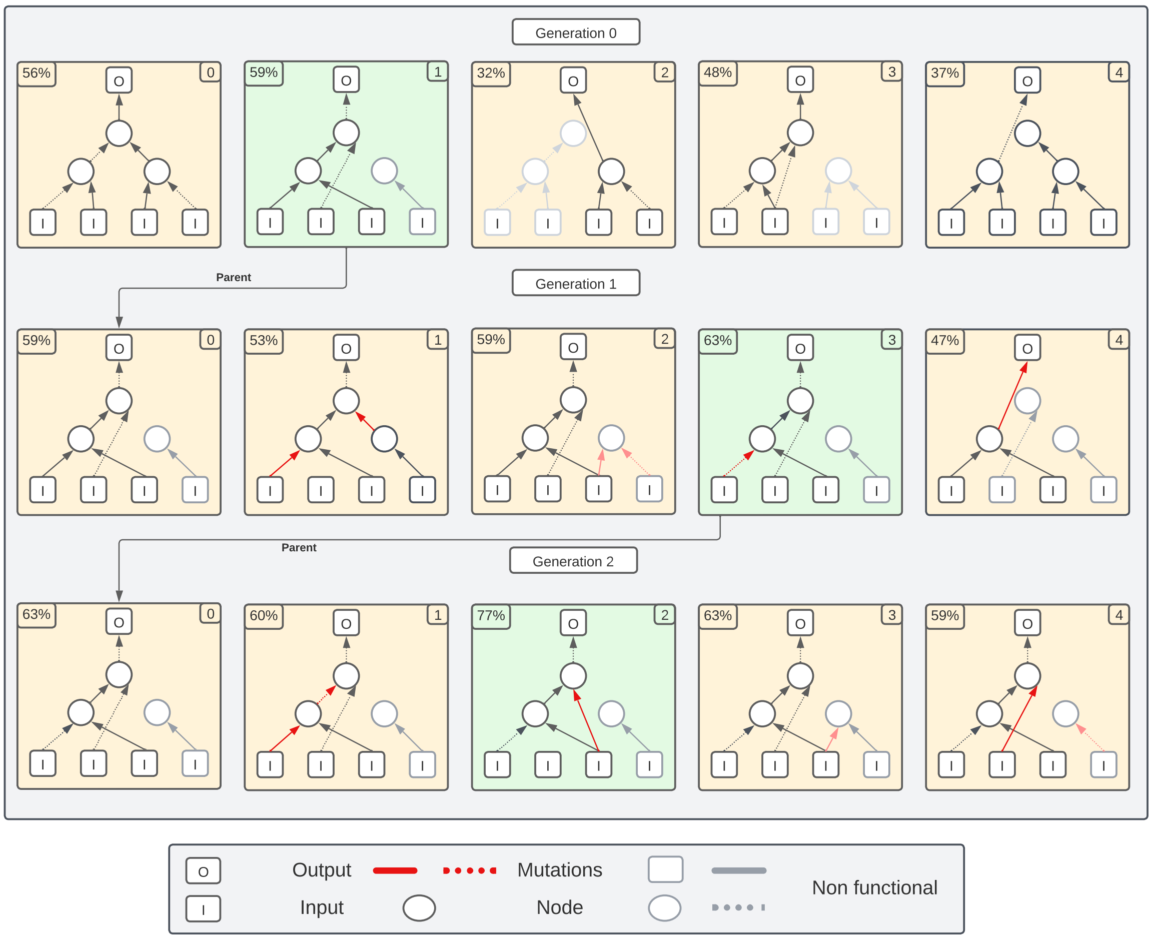


Figure 4 – Example of an evolutionary process of the CGP search.

Source: Bryan M. Lima, 2023

nodes is fixed throughout the whole evolutionary process. As AIG is the state-of-the-art data structure for logic synthesis, and CGP is a form of DAG as well, this work used the AIG data structure already implemented by Augusto André Souza Berndt et al. (2022).

2.3.2 Evolutionary process

To illustrate the evolutionary process of CGP, Figure 4 presents a hypothetical CGP search that utilizes the widely used evolutionary approach $(\mu + \lambda)$, where $\mu = 1$ and $\lambda = 4$ (MILLER, 2019). μ represents the number of parent genotypes to maintain through the next generation. Likewise, λ represents the number of genotypes of mutated offspring derived from the parents' circuits. Thus, for the given values, at each generation, one parent circuit generates four offspring. These offspring circuits are a copy of the parents' circuits that had some of their connections mutated by chance according to a user-defined probability, the mutation rate.

In Figure 4, direct connections are represented by a straight line and inverted connections are represented by dotted lines. Individuals are randomly generated in generation 0 and each circuit has a different accuracy (fitness value). The parent circuit of the next generation is highlighted in green. The circuit with the highest accuracy is selected as the parent of the next generation. Therefore, regarding generation 0, the next parent is the circuit with 59% of accuracy. In generation 1, this parent generates one unmodified copy and four mutated copies, i.e., copies with a chance of having different connections. Also, note that in Figure 4 the mutations are highlighted in red. This process repeats for generation 2 and so on, until reaching the number of generations defined by the user. In this example, for the sake of clarity, it is used a simpler representation of a CGP individual when compared to 3. Thus, we can focus on the learning of the search rather than focus on the individuals. Moreover, in this example, each individual had 3 nodes. However, this value is generally hundreds or even thousands of nodes in a real scenario.

Regarding the functional part of the CGP individual, it has been observed that during the evolutionary process, the majority of the genotypes are redundant - reaching even 95% of inactive nodes (MILLER; SMITH, 2006). This behavior is commonly called neutral drift, and it is important to the performance of CGP (MILLER, 2019), also contributing to the robustness of the system positively.

The work of Milano, Pagliuca, and Nolfi (2019) investigated the impacts of robustness during the evolutionary search of CGP for digital circuits. Previous to understanding the factors that impact robustness, it is relevant to state the concept of evolvability (MILANO; PAGLIUCA; NOLFI, 2019): Evolvability is the capacity of the system to generate solutions in which the variations on the phenotype is heritable for the next individuals. The main factors that impact the CGP robustness are: (i) robustness to mutation can have a positive impact on the performance of the evolvability of the system when it is achieved by selecting individuals with a larger phenotype; thus, having more redundancy in the generated individuals; (ii) whereas robustness negatively impacts the evolutionary search when it is achieved through selecting simpler phenotypical individuals.

The work of Milano, Pagliuca, and Nolfi (2019) demonstrated that the CGP search tends to more easily escape from local-optima regions, so avoiding premature convergence if phenotypically larger solutions are considered preferred candidates when analyzing the individual's fitness accuracy due to higher genetic variation. This happens since artificial evolution tends to select genotypes that are robust to mutations, i.e., that are less likely to produce maladaptive mutations. This, in turn, pushes in the direction of genotypes with tiny functional circuits, since the genotype becomes robust to any mutation that does not affect the functional part of the circuit, and this characteristic leads to premature convergence. The authors demonstrate that this limitation could be eliminated by selecting individuals with larger functional sizes when they have the same fitness. Therefore, in order to achieve positive robustness to mutation, this work utilizes this strategy. Thus, individuals with larger functional sizes are selected when a tie in fitness values happens.

3 ADAPTIVE BATCH SIZE STRATEGY FOR RUNTIME IMPROVEMENT OF CGP-BASED LOGIC OPTIMIZATION

Considering all of this, CGP is a promising technique in the context of logic synthesis for approximate computing with its capability for generalization; though, DTs, RFs, and, more recently, Decision Graphs (DGs) (HUANG; JIANG, 2023) have been outperforming CGP in a pure logic optimization flow. In this section, it is presented the proposed flow entitled Adaptive Batch Size CGP, which aims to reduce the runtime of the search algorithm while maintaining a comparable accuracy when compared to the classical CGP - and even improve it, in some cases, as we will see in section 4.2.

3.1 RESEARCH METHODOLOGY

The following steps were done in the development of this work:

1. Review of the state-of-the-art logic synthesis tools and the optimization process, focusing on approximate computing.
2. Present the basic concepts and current status of Cartesian Genetic Programming (CGP).
3. Present the proposed strategy for improving the CGP runtime in the logic optimization process.
4. Implement the proposed strategy by modifying the base CGP C++ implementation by Augusto André Souza Berndt et al. (2022).
5. Evaluate the proposed strategy in the 100 benchmarks of the International Workshop on Logic and Synthesis (IWLS) contest in 2020 considering the runtime and accuracy of the synthesized circuits.
6. Discuss the results by comparing the proposed strategy against the standard CGP version; the results were statistically tested (significance tests).
7. Summarize this work's findings and discuss future works.

3.2 BASE CGP IMPLEMENTATION

This work is an extension of the initial proposal of a CGP-based logic optimization flow (BERNDT, A. A. S. et al., 2022), keeping some structural parts and modifying others to improve the runtime by adopting the Adaptive Batch Size strategy. This section details the basic operation of the original flow.

The flow receives the circuit specification - i.e. the Truth Table as input in the format of a PLA file - and produces a circuit in the format of an And-Inverter Graph

(AIG). The CGP-based flow can be used to synthesize circuits based on initial random individuals (circuit), called Pure CGP flow, or it can be used to optimize an already synthesized circuit by another approach, called Fine-tuning flow. Figure 5 presents the original flowchart of the CGP-based flow.

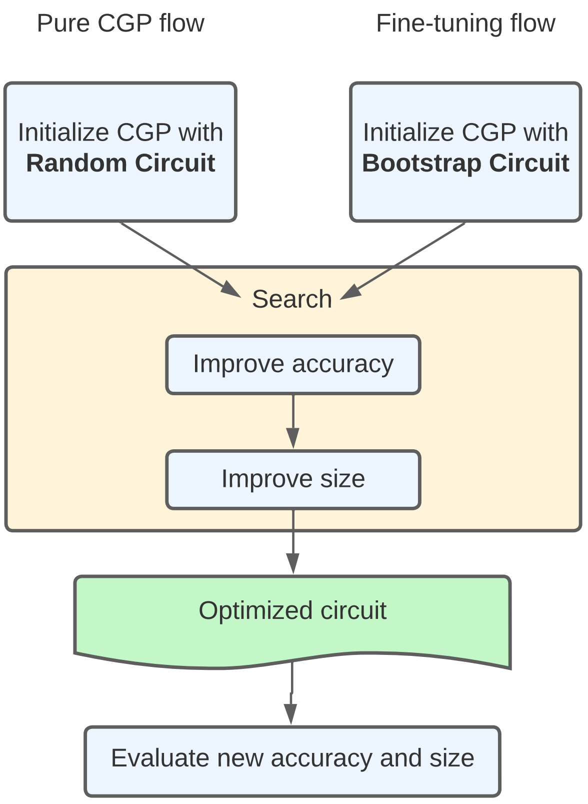

Pure CGP flow
Fine-tuning flow
Initialize CGP with **Random Circuit**
Initialize CGP with **Bootstrap Circuit**
Search
Improve accuracy
Improve size
Optimized circuit
Evaluate new accuracy and size

Figure 5 – CGP-based logic flow proposed by (BERNDT, A. A. S. et al., 2022)

Source: Adapted from (BERNDT, A. A. S. et al., 2022)

In the pure CGP flow, the authors utilize mini-batches instead of using the whole lines of the Truth Table available. The number of lines that these mini-batches use is called *batch size* (BS) and this definition is also used in this work. Furthermore, the authors utilized a strategy proposed by Carvalho, Milano, and Nolfi (2018) which maintains these mini-batches for a defined number of generations - called *change each* (CE). It was demonstrated by Carvalho, Milano, and Nolfi (2018) that this strategy produced better robust individuals. The BS and CE are a part of the hyperparameters of the CGP implementation, along with the number of generations. In the development of this work, CE optimization was also explored in the experiments. However, the ABS CGP was the first strategy explored by the research group and, therefore, CE was left out of this work. There is a promising strategy regarding the CE hyperparameter that will be further discussed in Section 5 that are interesting for future work.

Beyond this, the proposed flow by Augusto André Souza Berndt et al. (2022) also utilizes the strategy of selecting phenotypically complex individuals mentioned in Section 2.3.2. This is the **Improve accuracy** step presented in Figure 5. However, as larger circuits are not desirable in the logic synthesis domain, the authors also introduced the **Improve size** step in the flow. Thus, when the algorithm reaches a local maximum accuracy, it changes to selecting smaller circuits when a tie in fitness happens. Hence, the CGP-based flow first optimizes the circuit's accuracy, and then it optimizes the circuit's area.

This work utilizes parts of the CGP-based flow proposed by Augusto Berndt et al. (2021) and its C++ implementation[1]. The main goal of this work is to improve the CGP runtime, therefore, it focused on the Pure CGP flow. Furthermore, we focused on reducing the runtime while maintaining a reasonable accuracy of the synthesized circuits. Thus, it was only used the **Improve accuracy** step during the search. In the next section, the theoretical specification of the Adaptive Batch Size CGP will be detailed, along with some implementation details.

3.3 ADAPTIVE BATCH STRATEGY

Before detailing the proposed flow, the *batch size* definition is formalized below for a better understanding.

Definition 3.1 (Batch Size)**.** To evaluate the individuals generated at each generation, it is used a Truth Table, and the fitness value of the circuit is equal to how many lines of the Truth Table the individual managed to get it right. When training the algorithm, it is possible to use a portion of the Truth Table, instead of all the lines; the number of lines used to verify the circuits' fitness is called batch size (BS).

3.3.1 Proposed flow

Traditionally, CGP evaluates all the individuals every generation after they are created by the parent mutation. The evaluation process for each of them requires processing all the training batch inputs. This is done with a depth-first search along the AIG. Thus, the evaluation has a time complexity of $O(i * n * b)$, with i being the number of individuals in the CGP population, n the size of each individual, and b the BS used for training.

Figure 6 presents a flowchart of a simplified version of the Adaptive Batch Size (ABS) CGP technique. First, ABS CGP starts with an initial batch size of β terms. The next step is choosing a criterion for defining and detecting evolution stagnation, and by doing so, increasing the batch size when it happens. The initially chosen criterion was a Simple Moving Average (SMA) from the accuracy of the synthesized circuits, considering a window of σ generations. In other words, the SMA gathers the accuracy of the best individual for each generation, for a period of σ generations, and then takes

[1] Source code available in `https://gitlab.com/gudeh/cgpv3`

the average of this value.

For the next step, there is a verification that compares the current SMA to the previous SMA. By always saving the previous SMA calculated, we can compare this value with the SMA of the current generation. The comparison of both values provides us with the information that CGP is actually learning the function at hand. Therefore, if these values are the same, there were no improvements in accuracy in the last $2 * \sigma$ generations. This is the core of the strategy, if CGP is not improving the circuit's accuracy, i.e. the estimation quality is stagnant, exposing the algorithm to more terms could help broaden its search. By using a BS smaller than the whole Truth Table, stochasticity in the evaluation process is inherently added. The smaller the BS value, the faster the evaluation process and the higher the stochasticity. Therefore, there is a moment in which the individuals' fitness reaches a point that, due to stochasticity, the algorithm cannot differ which are the best CGP individuals anymore. The proposed strategy tries to solve this by increasing the number of terms when it detects that this situation occurs, reducing the stochasticity in the evaluation, and by doing so, improving the quality of the fitness estimation.

Before increasing the batch size, there is another verification checking if increasing the batch will not surpass the size of the whole data available. If this verification yields true, i.e. increasing the batch does not surpass the complete data, then the batch is increased by α terms and the SMA is calculated again after σ generations. If the previously mentioned verification yields false, that means the algorithm is almost or already using the complete data available. Thus, increasing the batch would surpass all data available. In this case, the whole data available is chosen as the batch size of the next generation, and there is no need for any more steps, as the CGP search already uses the maximum amount of data available. After this step, the CGP continues its search as the standard version.

It is important to note that the value of the BS directly impacts the runtime, as the search algorithm will utilize this value to measure the synthesized circuit's accuracy. Therefore, evaluating a circuit's accuracy with a BS of 128 is computationally cheaper

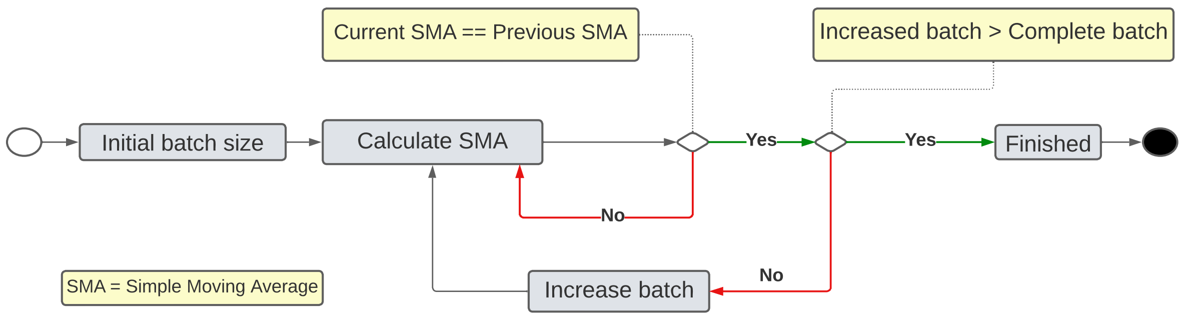

Current SMA == Previous SMA
Increased batch > Complete batch
Initial batch size
Calculate SMA
Yes
Yes
Finished
No
No
Increase batch
SMA = Simple Moving Average

Figure 6 – Adaptive Batch Size CGP flowchart

Source: Bryan M. Lima, 2023

than evaluating with a BS of 1024.

Furthermore, this work performed an improvement regarding the evaluation process using the CGP C++ implementation used in Augusto André Souza Berndt et al. (2022). The main difference in this version is when to evaluate the nodes' accuracy, as this process is computationally costly. For this, we used the proposed strategy "Skip" by Goldman and Punch (2014) which means skipping the evaluation of individuals in which the functional nodes of their parent were not mutated. This change was not only better for performance, but it was necessary to be possible to evenly compare the multiples hyperparameters used in CGP, by saving the number of selected terms of the Truth Table used for evaluating the individuals of each experiment. The "Skip" strategy does not affect by any means the performance of the circuits synthesized by the algorithm (GOLDMAN; PUNCH, 2014), it only prevents unnecessarily evaluating nodes that certainly have the same accuracy as their parent, thus, it does not waste time evaluating an already known accuracy.

Hence, by avoiding unnecessary evaluations and dynamically increasing the number of terms for evaluating the candidate solutions during the CGP search, we reduce the number of evaluations required for synthesizing circuits and consequently reduce runtime.

4 EXPERIMENTS AND RESULTS

4.1 EXPERIMENTAL PROTOCOL

We used the 100 exemplars from the IWLS 2020 contest to compare ABS CGP with the standard CGP. Table 1 presents characteristics of the exemplars chosen for validating the proposed algorithm. Exemplars 00 through 49 are from the Arithmetic domain; exemplars 50 through 79 are from the Random logic domain, and exemplars 80 through 99 are from the ML domain - specifically, exemplars 80 through 89 are from the MNIST (DENG, 2012) dataset, and exemplars 90 through 99 are from the CIFAR-10 (KRIZHEVSKY; HINTON, et al., 2009) dataset. The contest only provided 6,400 Truth Table lines for each exemplar. This amount can only completely describe logic functions below 12 inputs. Most exemplars have an incompletely specified logic function, as shown by the number of inputs in Table 1. Besides this, exemplars from the ML domain are among the largest of the contest and provided lines only representing a tiny portion of all possible combinations for MNIST and CIFAR, 2^{196} and 2^{768} combinations, respectively.

Exemplars	Logic type	# Inputs Min	# Inputs Max	Logic Domain
00-09	Adders	32	512	Arithmetic
10-19	Dividers, Remainders	32	512	Arithmetic
20-29	Multipliers	16	256	Arithmetic
30-39	Comparators	20	200	Arithmetic
40-49	Square-root	10	256	Arithmetic
50-59	PicoJava design	16	394	Random
60-74	MCNC benchmarks	16	52	Random
75-79	Symmetric functions	16	16	Random
80-89	MNIST	196	196	ML
90-99	CIFAR-10	768	768	ML

Table 1 – Exemplars circuit details.

Source: (BERNDT, A. et al., 2021)

Table 2 shows the hyperparameters used in the experiments. We used a value of 50,000 generations to identify the initial impacts on the learning of the circuit. We used a number of nodes of 1,000, which means that the CGP has a maximum of 1,000 functional nodes. Moreover, we replicated each experiment ten times with different random seeds; as this algorithm is impacted by different sources of stochasticity - i.e., its initial randomly generated circuits - ten runs provided enough information for the statistical validation of the results. These variations in the accuracy caused by stochasticity will be noticeable on the box plots presented in Section 4.2.

For the ABS CGP-specific hyperparameters, the Adaptive Interval (AI) parameter was varied between 0.2% and 10% of the total evolutionary process of 50,000 generations to analyze its impact. Due to the high computational cost of optimizing all

Parameter	Value
Number of generations	50,000
Number of nodes	1,000
Number of seeds	10
Evolutionary Strategy	(1 + 4)
ABS CGP-specific	
Adaptive interval (σ)	100, 250, 500, 1000, 2500, 5000
Initial batch size (β)	64
Increase batch factor (α)	64
Standard CGP-specific	
Batch size	6400

Table 2 – Hyperparameters tested

hyperparameters, the initial batch size and the increased batch factor values were fixed to the value of 64. Investigating the impact of varying these values on the performance of the ABS CGP is relevant and could be done in future work. To compare the proposed strategy to the standard CGP version, the value of the BS was fixed with 6,400 terms, as it was the maximum value for the number of terms for all exemplars of the IWLS contest. We also used the same 50,000 generations with 1,000 nodes and ten seeds for the control case.

Finally, instead of using the algorithm runtime as one of the comparison metrics, it was used the number of candidate solutions evaluations performed during the evolutionary process. This is a standard metric within the evolutionary computing community, since it removes hardware-related differences among studies. It will also be provided the runtime for these experiments to demonstrate that the runtime is linked to the number of evaluations. However, these values should be taken lightly, as they were executed in a shared environment, which could impact the runtime of any data collected. Besides this, to measure the total number of evaluations, we use the sum over the generations of the result of the multiplication between the BS and the offspring in each generation. Equation 5 presents the complete equation, where N represents the user-defined number of generations.

$$Evaluations = \sum_{i=1}^{N} BS_i * offspring \tag{5}$$

As we use Equation 5 to calculate the number of evaluations, it is clear that the BS value is the one that impacts the number of evaluations the most; as the number of generations is fixed and the $offspring$ variable is independent of the ABS strategy, the only variable that is dependent on the proposed strategy is the BS value.

4.2 RESULTS

The results presented herein show that the ABS approach led to a significant reduction in the number of evaluations required for evaluating the circuit's accuracy during the evolutionary search. Furthermore, it presented an accuracy increase in some cases. Of the 100 exemplars tested, only 21 achieved a statistically significant accuracy difference when compared to the standard CGP search - the Mann Whitney-U test with $p < .05$ was used. All of these 21 exemplars achieved an accuracy increase. Figure 7 presents the accuracy of the five best circuits' accuracies and the five worst when using all available terms of the Truth Table versus the proposed strategy ABS CGP. The results of ABS consider the best set of hyperparameters for each exemplar, i.e. after running the hyperparameter optimization. Exemplars 27 and 29 achieved the best accuracy increase across all 100 exemplars, they managed to increase the accuracy compared to the standard CGP by 26% and 57.3%, respectively. Furthermore, the worst-case scenarios for ABS CGP were in the results of exemplars 36 and 38, which had a decrease in accuracy by 1.3% and 2.14%, respectively. It is noteworthy that the five worst circuits' performance did not achieve a statistically significant difference.

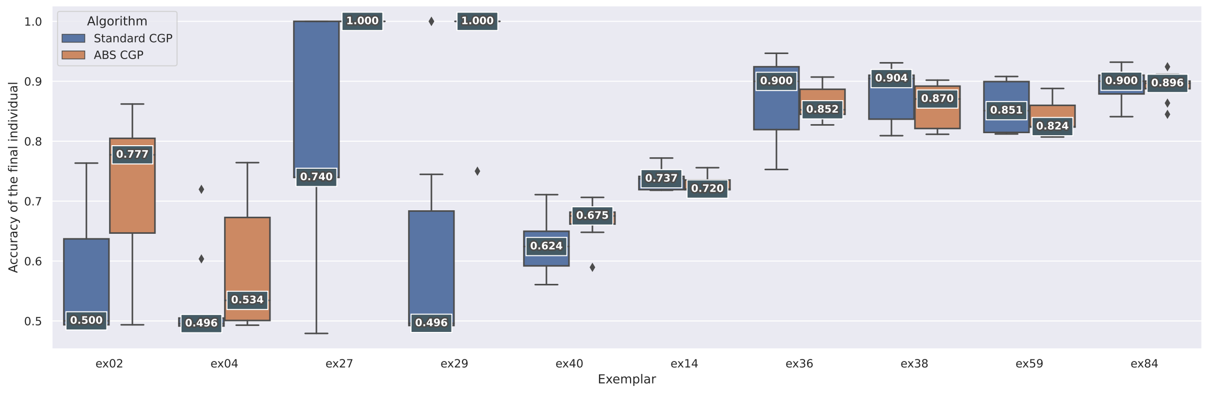


Figure 7 – Accuracy of Standard CGP and ABS CGP

Table 3 presents the comparison of the ABS versus the standard version for the 21 exemplars that achieved a statistically significant accuracy difference. On average, ABS managed to increase the accuracy by 8.69% on these exemplars; though, when considering all 100 exemplars, ABS achieved an average increase in accuracy of 2.66%.

The main advantage of ABS is the reduction in the number of evaluations. Figure 8 presents the comparison of the number of evaluations for the same set of experiments of Figure 7. All results from Figure 8 achieved a statistically significant difference (Mann-Whitney U Test, $p < .05$ for all exemplars). Despite not achieving a significant improvement for all exemplars, the accuracy level obtained by the control case was maintained, while decreasing the evaluations performed by 53.65% on average for these exemplars. From these results, it seems that ABS CGP is capable of synthesizing circuits with better accuracy in the arithmetic domain, considering the highest accuracy

Exemplar	Version	Accuracy Mean	P value	Accuracy Gain
ex02	Standard	0.561 ± 0.100	0.017	27.72%
	ABS	0.717 ± 0.126		
ex04	Standard	0.528 ± 0.072	0.045	11.13%
	ABS	0.587 ± 0.099		
ex10	Standard	0.775 ± 0.027	0.030	3.57%
	ABS	0.803 ± 0.022		
ex19	Standard	0.827 ± 0.008	0.045	1.05%
	ABS	0.835 ± 0.009		
ex20	Standard	0.511 ± 0.006	0.046	1.07%
	ABS	0.516 ± 0.005		
ex24	Standard	0.497 ± 0.004	0.006	1.16%
	ABS	0.503 ± 0.004		
ex27	Standard	0.794 ± 0.192	0.006	25.98%
	ABS	1.000 ± 0.000		
ex29	Standard	0.620 ± 0.204	0.001	57.30%
	ABS	0.975 ± 0.075		
ex30	Standard	0.902 ± 0.039	0.019	4.92%
	ABS	0.947 ± 0.034		
ex40	Standard	0.622 ± 0.044	0.021	7.25%
	ABS	0.667 ± 0.030		
ex41	Standard	0.837 ± 0.039	0.031	5.23%
	ABS	0.881 ± 0.050		
ex43	Standard	0.584 ± 0.025	0.001	7.23%
	ABS	0.626 ± 0.022		
ex47	Standard	0.495 ± 0.004	0.009	1.22%
	ABS	0.501 ± 0.008		
ex50	Standard	0.898 ± 0.061	0.040	5.72%
	ABS	0.950 ± 0.033		
ex53	Standard	0.783 ± 0.018	0.004	3.97%
	ABS	0.814 ± 0.022		
ex55	Standard	0.931 ± 0.030	0.004	3.84%
	ABS	0.966 ± 0.025		
ex60	Standard	0.868 ± 0.022	0.017	3.15%
	ABS	0.895 ± 0.028		
ex63	Standard	0.940 ± 0.030	0.023	2.54%
	ABS	0.964 ± 0.010		
ex69	Standard	0.959 ± 0.017	0.043	2.43%
	ABS	0.982 ± 0.021		
ex72	Standard	0.931 ± 0.034	0.006	3.28%
	ABS	0.961 ± 0.015		
ex83	Standard	0.911 ± 0.038	0.031	2.74%
	ABS	0.936 ± 0.017		

Table 3 – Accuracy gains for the exemplars with significantly different accuracy (Mann Whitney-U test with $p < .05$)

gains are from this domain; although further analysis is needed to better understand why this happens for this particular family of functions.

Figure 9 presents the comparison of runtime between both versions considering the same set of experiments of Figure 7. It is important to note that the experiments were executed in a shared environment, and the time calculation is based on the wall time. Therefore, the data collected could be altered by another computationally intensive process. Even so, these results prove that the number of evaluations impacts the runtime and that ABS CGP managed to decrease the number of evaluations required in the evolutionary process. For the interested reader, Appendix B presents the same set of results considering the comparison of accuracy, number of evaluations, and runtime for all 100 exemplars of the IWLS 2020 contest. This section only maintained the best and worst cases for ABS CGP for a concise comparison between both strategies.

Another noteworthy insight from these experiments is the values of the Adaptive Interval (AI). As we can see in Table 4, most of the AI values are in the lower range of the chosen AI parameters of Table 2. Only the AI values of 100 and 250 managed to reach a point in the evolutionary process in which all available lines were used for evaluating individuals in all the experiments. Furthermore, the results with AI above 500 had worse accuracy compared to the results presented herein. This indicates that more generations could be executed on the search, since with these AI values the CGP search could not reach the complete batch size. Further investigation is required to analyze if by increasing the number of generations the evolutionary processes using these higher AI values present better results with respect to using lower AI values as they would have more time to reach the maximum size. Moreover, using a fixed size of α, which controls by how much the batch will be increased, may be too low. In the set of experiments, we used a value of α representing only 1% of the Truth Table which could be optimal for some exemplars, but not others. Overall, this data indicates that there might be room for improvement if this hyperparameter would be optimized as well.

Adaptive Interval	Count
100	58
250	33
500	9

Table 4 – Number of AI values that achieved the best set of results.

To further investigate the learning curve of the ABS CGP evolutionary search, we checked how the fitness of the best individuals increased through generations. Besides collecting the partial accuracy of the circuit in each generation, we collected its validation accuracy as well, i.e. the accuracy using the validation set. Figure 10 presents

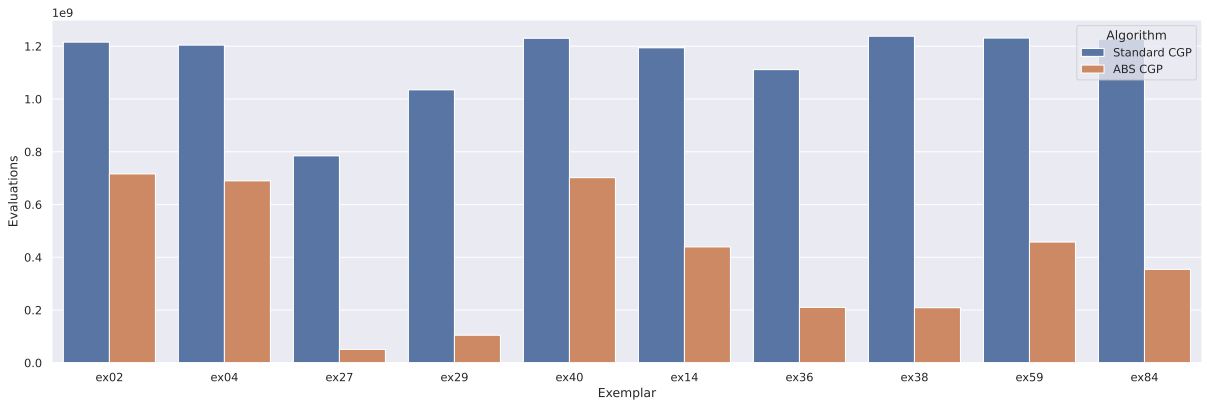


Figure 8 – Number of evaluations of Standard CGP and ABS CGP

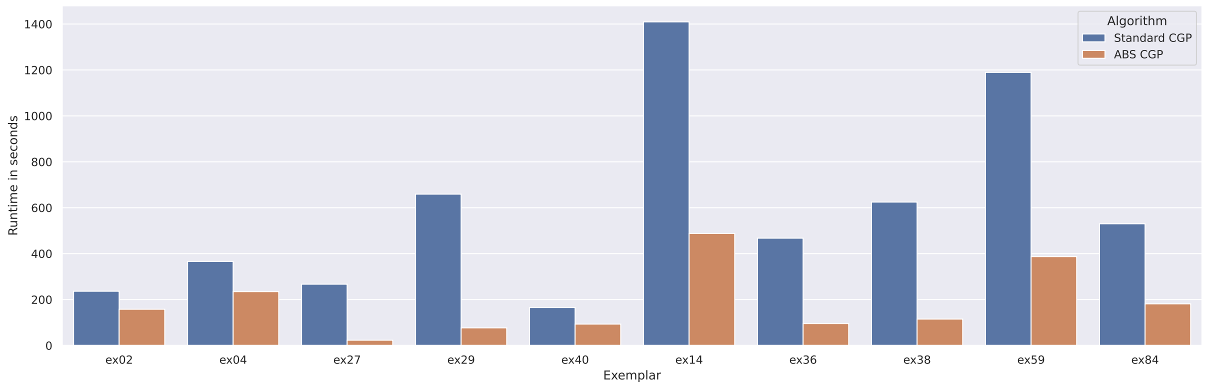


Figure 9 – Runtime of Standard CGP and ABS CGP

this data for the initial evolutionary search for exemplar 41 - the only completely specified function of the contest. The blue line represents the search using all terms available when using the standard CGP. The green line represents the search for the Adaptive version with AI set to 100. Hence, the first generations are using a smaller portion of the Truth Table, and the last ones are using all terms available. The orange line is representing the accuracy of the same Adaptive version of 100 AI while using all terms from the beginning. For this reason, the orange line and green line converge at the end.

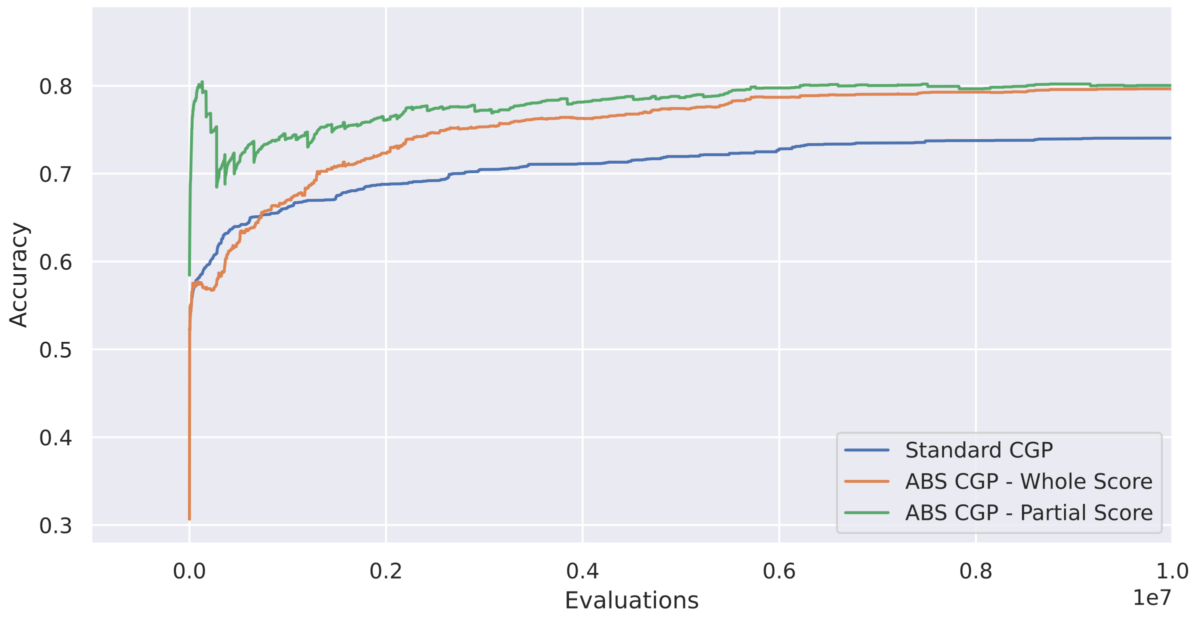


Figure 10 – Accuracy during learning for ex41

It is noteworthy that the first assumption for the ABS CGP was that it could improve faster at the beginning, even when using a rough fitness evaluation of the candidate solutions, i.e., using fewer lines for evaluating each circuit and performing the fine-tuning of circuits at the end, using the whole data available. Indeed, the learning of the ABS CGP (orange line) managed to achieve better accuracy in these initial evaluations compared to the complete version, even with a worse fitness estimation than the version using all data available for evaluating individuals. Furthermore, ABS kept improving the circuits beyond the performance achieved by the standard version, providing a better final result.

5 CONCLUSION

Due to the increasing complexity of digital circuits, novel techniques for logic synthesis are required. The combination of logic synthesis and machine learning has been gaining much attention in the literature due to the promising results regarding circuit area reduction and power consumption, mainly in approximate computing domains. Among these techniques, Cartesian Genetic Programming can synthesize better circuits than traditional techniques in certain domains, but at the cost of demanding a great computational power, which slows down the synthesis.

This work proposes a novel technique called Adaptive Batch Size CGP seeking to alleviate the computational resources required by CGP, and by doing so, improve the runtime and accuracy of the synthesized solutions. The results confirmed that by incrementally improving the fitness estimation along the evolutionary process, we could obtain significant improvements in the CGP convergence speed and in the quality of the synthesized circuits. For all 100 experiments performed, the ABS CGP achieved, on average, an accuracy increase of 2.66%, though it achieved an accuracy increase of up to 57.3% for exemplar 29. Considering all exemplars, only 21 achieved a statistically significant different accuracy compared to the standard CGP search. However, the main benefit of ABS CGP was the reduction of the required evaluations needed to evaluate the individuals. On average, ABS CGP reduced the number of evaluations by 53.65%, and all exemplars achieved a statistically significant difference in the evaluation results.

However, in a number of cases of the presented results, the final BS did not reach a point in the evolutionary process in which all data available was used for evaluating the individuals. These results indicate that some particular combinations of data available, number of generations, and the AI were not optimal for certain cases. Moreover, the current version of ABS CGP has a fixed initial batch size and in all experiments, this value was set to 64 terms. Therefore, as future work, investigating how the initial batch size, the parameter α that controls the batch increase, and the number of generations affect the synthesis for each particular exemplar is an interesting research direction.

In Section 3.2, it was discussed the hyperparameters of CGP, such as *batch size* (BS) and *change each* (CE), though only the BS was explored in this work. Therefore, future work could investigate the impacts of using the CE along the ABS CGP. Furthermore, during the development of this work, there was also a promising direction regarding the optimization of the CE hyperparameter. In the standard CE strategy used by Augusto André Souza Berndt et al. (2022), when the batch size was maintained by the CE value, the mini-batch would randomly select BS lines of the Truth Table that would consist of the next mini-batch. The promising strategy is called partial CE, consisting of only changing a portion of the mini-batch to keep low stochasticity on the algorithm. In a portion of the experiments, changing only 25% of the mini-batch using half the lines of the truth table as the BS produced better accuracy circuits when compared to the standard CGP. Though, further analysis of this strategy is required to confirm its potential, and it could be done in future work.

Beyond this, this work utilizes the AIG data structure; however, there were also experiment sets that utilized XAIG. The usage of AIG limited the circuit accuracy on circuits with many XOR logic gates. However, it was tested using XAIG as a data structure for all exemplars of the IWLS 2020 Contest, and it performed drastically worse than the AIG counterpart. Further analysis in the comparison of the AIG versus XAIG in the CGP-based logic flow is an interesting topic for future work as well.

REFERENCES

ABREU, Brunno A. de et al. Fast Logic Optimization Using Decision Trees. In: 2021 IEEE International Symposium on Circuits and Systems (ISCAS). [S.l.: s.n.], 2021. P. 1–5. Available from: <https://doi.org/10.1109/ISCAS51556.2021.9401664>. Cit. on p. 13.

AMARU, Luca; GAILLARDON, Pierre-Emmanuel; DE MICHELI, Giovanni. Majority-inverter graph: A new paradigm for logic optimization. **IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems**, IEEE, v. 35, n. 5, p. 806–819, 2015. Cit. on p. 17.

AMARÚ, Luca et al. Logic optimization and synthesis: Trends and directions in industry. In: IEEE. DESIGN, Automation & Test in Europe Conference & Exhibition (DATE), 2017. [S.l.: s.n.], 2017. P. 1303–1305. Cit. on pp. 12, 15.

BARUA, Hrishav Bakul; MONDAL, Kartick Chandra. Approximate Computing: A Survey of Recent Trends—Bringing Greenness to Computing and Communication. **Journal of The Institution of Engineers (India): Series B**, Springer, p. 1–8, 2019. Available from: <https://doi.org/10.1007/s40031-019-00418-8>. Cit. on pp. 13, 19, 20.

BEEREL, P. A.; PEDRAM, M. Opportunities for Machine Learning in Electronic Design Automation. In: 2018 IEEE International Symposium on Circuits and Systems (ISCAS). [S.l.: s.n.], 2018. P. 1–5. DOI: 10.1109/ISCAS.2018.8351731. Available from: <https://doi.org/10.1109/ISCAS.2018.8351731>. Cit. on pp. 13, 19.

BERKELEY LOGIC SYNTHESIS AND VERIFICATION GROUP. **ABC: A System for Sequential Synthesis and Verification**. [S.l.: s.n.], 2019. http://www.eecs.berkeley.edu/ alanmi/abc/.html. Visited on: 10 Jan. 2020. Cit. on p. 18.

BERNDT, Augusto et al. Accuracy and Size Trade-off of a Cartesian Genetic Programming Flow for Logic Optimization. In: PROCEEDINGS of the 34th Symposium on Integrated Circuits and Systems Design. Brazil: [s.n.], 2021. (SBCCI '21). Available from: <https://doi.org/10.1109/SBCCI53441.2021.9529968>. Cit. on pp. 13, 21, 22, 29, 32.

BERNDT, Augusto André Souza et al. A CGP-based Logic Flow: Optimizing Accuracy and Size of Approximate Circuits. **Journal of Integrated Circuits and Systems**, v. 17, n. 1, p. 1–12, 2022. DOI: https://doi.org/10.29292/jics.v17i1.546. Cit. on pp. 13, 22, 24, 27, 28, 31, 39.

BRAYTON, Robert K. The decomposition and factorization of Boolean expressions. **ISCA-82**, p. 49–54, 1982. Cit. on p. 18.

BRAYTON, Robert K. et al. Logic Minimization Algorithms for VLSI Synthesis". **The Kluwer International Series in Engineering and Computer Science**, v. 2, p. 1–194, 1984. Available from: <https://doi.org/10.1007/978-1-4613-2821-6>. Cit. on pp. 12, 15.

BREIMAN, Leo. Bagging predictors. **Machine learning**, Springer, v. 24, p. 123–140, 1996. Cit. on pp. 13, 19.

BRYANT, Randal E. Graph-based algorithms for boolean function manipulation. **Computers, IEEE Transactions on**, IEEE, v. 100, n. 8, p. 677–691, 1986. Cit. on p. 17.

BURGES, Christopher JC. A tutorial on support vector machines for pattern recognition. **Data mining and knowledge discovery**, Springer, v. 2, n. 2, p. 121–167, 1998. Cit. on pp. 13, 19.

CARVALHO, Jônata; MILANO, Nicola; NOLFI, Stefano. Evolving Robust Solutions for Stochastically Varying Problems. In: IEEE. 2018 IEEE Congress on Evolutionary Computation (CEC). [S.l.: s.n.], 2018. P. 1–8. Available from: <https://doi.org/10.1109/CEC.2018.8477811>. Cit. on p. 28.

DENG, L. The MNIST Database of Handwritten Digit Images for Machine Learning Research. **IEEE Signal Processing Magazine**, v. 29, n. 6, p. 141–142, Nov. 2012. ISSN 1558-0792. DOI: 10.1109/MSP.2012.2211477. Cit. on p. 32.

FAYYAZI, Arash et al. Deep Learning-Based Circuit Recognition Using Sparse Mapping and Level-Dependent Decaying Sum Circuit Representations. In: IEEE. 2019 Design, Automation & Test in Europe Conference & Exhibition (DATE). [S.l.: s.n.], 2019. P. 638–641. DOI: 10.23919/DATE.2019.8715251. Available from: <https://doi.org/10.23919/DATE.2019.8715251>. Cit. on p. 19.

GOLDMAN, Brian W; PUNCH, William F. Analysis of cartesian genetic programming's evolutionary mechanisms. **IEEE Transactions on Evolutionary Computation**, IEEE, v. 19, n. 3, p. 359–373, 2014. Cit. on p. 31.

HÁLEČEK, Ivo; FIŠER, Petr; SCHMIDT, Jan. Towards AND/XOR balanced synthesis: Logic circuits rewriting with XOR. **Microelectronics Reliability**, v. 81, p. 274–286,

2018. ISSN 0026-2714. DOI: https://doi.org/10.1016/j.microrel.2017.12.031. Available from: <https://www.sciencedirect.com/science/article/pii/S0026271417305899>. Cit. on p. 17.

HAN, Jie; ORSHANSKY, Michael. Approximate computing: An emerging paradigm for energy-efficient design. In: IEEE. 2013 18th IEEE European Test Symposium (ETS). [S.l.: s.n.], 2013. P. 1–6. Cit. on p. 19.

HARDING, Simon. Evolution of image filters on graphics processor units using cartesian genetic programming. In: IEEE. 2008 IEEE Congress on Evolutionary Computation (IEEE World Congress on Computational Intelligence). [S.l.: s.n.], 2008. P. 1921–1928. Cit. on p. 22.

HARDING, Simon; LEITNER, Jürgen; SCHMIDHUBER, Juergen. Cartesian genetic programming for image processing. In: GENETIC programming theory and practice X. [S.l.]: Springer, 2013. P. 31–44. Cit. on p. 22.

HASSAN, Shazia; HUMAIRA; ASGHAR, Mamoona. Limitation of Silicon Based Computation and Future Prospects. In: 2010 Second International Conference on Communication Software and Networks. [S.l.: s.n.], 2010. P. 559–561. DOI: 10.1109/ICCSN.2010.81. Cit. on p. 12.

HUANG, Yu-Shan; JIANG, Jie-Hong R. Circuit Learning: From Decision Trees to Decision Graphs. **IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems**, IEEE, 2023. Cit. on pp. 13, 22, 27.

JIANG, Jie-Hong Roland; DEVADAS, Srinivas. Logic synthesis in a nutshell. In: ELECTRONIC Design Automation. [S.l.]: Elsevier, 2009. P. 299–404. Cit. on p. 15.

JOACHIMS, Thorsten. Text categorization with support vector machines: Learning with many relevant features. In: SPRINGER. MACHINE Learning: ECML-98: 10th European Conference on Machine Learning Chemnitz, Germany, April 21–23, 1998 Proceedings. [S.l.: s.n.], 2005. P. 137–142. Cit. on pp. 13, 19.

KARNAUGH, M. The map method for synthesis of combinational logic circuits. **Transactions of the American Institute of Electrical Engineers, Part I: Communication and Electronics**, v. 72, n. 5, p. 593–599, 1953. DOI: 10.1109/TCE.1953.6371932. Available from: <https://doi.org/10.1109/TCE.1953.6371932>. Cit. on p. 12.

KHAN, Maryam Mahsal; AHMAD, Arbab Masood, et al. Fast learning neural networks using cartesian genetic programming. **Neurocomputing**, Elsevier, v. 121, p. 274–289, 2013. Cit. on p. 22.

KHAN, Maryam Mahsal; KHAN, Gul Muhammad; MILLER, Julian F. Evolution of neural networks using cartesian genetic programming. In: IEEE. IEEE congress on evolutionary computation. [S.l.: s.n.], 2010. P. 1–8. Cit. on p. 22.

KOZA, John R. Genetic programming as a means for programming computers by natural selection. **Statistics and computing**, Springer, v. 4, p. 87–112, 1994. Cit. on p. 22.

KRIZHEVSKY, Alex; HINTON, Geoffrey, et al. Learning multiple layers of features from tiny images. Toronto, ON, Canada, 2009. Cit. on p. 32.

MANAZIR, Abdul; RAZA, Khalid. Recent developments in Cartesian genetic programming and its variants. **ACM Computing Surveys (CSUR)**, ACM New York, NY, USA, v. 51, n. 6, p. 1–29, 2019. Available from: <https://doi.org/10.1145/3275518>. Cit. on p. 22.

MICHELI, Giovanni De. **Synthesis and optimization of digital circuits**. [S.l.]: McGraw-Hill Higher Education, 1994. Cit. on p. 15.

MILANO, Nicola; PAGLIUCA, Paolo; NOLFI, Stefano. Robustness, evolvability and phenotypic complexity: insights from evolving digital circuits. **Evolutionary Intelligence**, Springer, v. 12, n. 1, p. 83–95, 2019. Available from: <https://doi.org/10.1007/s12065-018-00197-z>. Cit. on pp. 23, 25.

MILLER, J; THOMSON, P. **Cartesian Genetic Programming In: Proc. Of the Third European Conference on Genetic Programming, LNCS 1802**. [S.l.]: Springer-Verlag, 2000. Cit. on p. 22.

MILLER, Julian F; SMITH, Stephen L. Redundancy and computational efficiency in cartesian genetic programming. **IEEE Transactions on evolutionary computation**, IEEE, v. 10, n. 2, p. 167–174, 2006. Cit. on p. 25.

MILLER, Julian Francis. Cartesian genetic programming: its status and future. **Genetic Programming and Evolvable Machines**, Springer, p. 1–40, 2019. Available from: <https://doi.org/10.1007/s10710-019-09360-6>. Cit. on pp. 22, 24, 25.

MISHCHENKO, A.; CHATTERJEE, S.; BRAYTON, R. DAG-aware AIG rewriting: a fresh look at combinational logic synthesis. In: 2006 43rd ACM/IEEE Design Automation Conference. [S.l.: s.n.], 2006. P. 532–535. Available from: <https://doi.org/10.1145/1146909.1147048>. Cit. on p. 17.

MISHCHENKO, Alan et al. Delay optimization using SOP balancing. In: IEEE. 2011 IEEE/ACM International Conference on Computer-Aided Design (ICCAD). [S.l.: s.n.], 2011. P. 375–382. Cit. on p. 19.

MIYASAKA, Yukio et al. Logic Synthesis for Generalization and Learning Addition. In: 2021 Design, Automation Test in Europe Conference Exhibition (DATE). [S.l.: s.n.], 2021. P. 1032–1037. DOI: 10.23919/DATE51398.2021.9474169. Available from: <https://doi.org/10.23919/DATE51398.2021.9474169>. Cit. on p. 13.

NETO, Walter Lau et al. LSOracle: a Logic Synthesis Framework Driven by Artificial Intelligence. In: IEEE. 2019 IEEE/ACM International Conference on Computer-Aided Design (ICCAD). [S.l.: s.n.], 2019. P. 1–6. DOI: 10.1109/ICCAD45719.2019.8942145. Available from: <https://doi.org/10.1109/ICCAD45719.2019.8942145>. Cit. on p. 20.

PARK, Jongse; ESMAEILZADEH, Hadi, et al. FlexJava: Language Support for Safe and Modular Approximate Programming. In: PROCEEDINGS of the 2015 10th Joint Meeting on Foundations of Software Engineering. Bergamo, Italy: Association for Computing Machinery, 2015. (ESEC/FSE 2015), p. 745–757. DOI: 10.1145/2786805.2786807. Available from: <https://doi.org/10.1145/2786805.2786807>. Cit. on p. 20.

PARK, Jongse; ZHANG, Xin, et al. **Expax: A framework for automating approximate programming**. [S.l.], 2014. Cit. on p. 20.

PICEK, Stjepan; CARLET, Claude, et al. Evolutionary algorithms for boolean functions in diverse domains of cryptography. **Evolutionary computation**, MIT Press, v. 24, n. 4, p. 667–694, 2016. Cit. on p. 22.

PICEK, Stjepan; JAKOBOVIC, Domagoj, et al. Cryptographic Boolean functions: One output, many design criteria. **Applied Soft Computing**, Elsevier, v. 40, p. 635–653, 2016. Cit. on p. 22.

QUINE, W. V. A Way to Simplify Truth Functions. **The American Mathematical Monthly**, Taylor & Francis, v. 62, n. 9, p. 627–631, 1955. Available from: <https://doi.org/10.1080/00029890.1955.11988710>. Cit. on pp. 12, 15.

RAI, Shubham; AL., et. Logic Synthesis Meets Machine Learning: Trading Exactness for Generalization. In: IEEE. 2021 Design, Automation & Test in Europe Conference & Exhibition (DATE). [S.l.: s.n.], 2021. DOI: `10.23919/DATE51398.2021.9473972`. Available from: <`https://doi.org/10.23919/DATE51398.2021.9473972`>. Cit. on p. 21.

RIENER, H. et al. On-the-fly and DAG-aware: Rewriting Boolean Networks with Exact Synthesis. In: 2019 Design, Automation Test in Europe Conference Exhibition (DATE). [S.l.: s.n.], 2019. P. 1649–1654. DOI: `10.23919/DATE.2019.8715185`. Available from: <`https://doi.org/10.23919/DATE.2019.8715185`>. Cit. on pp. 17–19.

RUDELL, R. L.; SANGIOVANNI-VINCENTELLI, A. Multiple-Valued Minimization for PLA Optimization. **IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems**, v. 6, n. 5, p. 727–750, 1987. DOI: `10.1109/TCAD.1987.1270318`. Available from: <`https://doi.org/10.1109/TCAD.1987.1270318`>. Cit. on pp. 12, 15.

SAMPSON, Adrian et al. EnerJ: Approximate Data Types for Safe and General Low-Power Computation. **SIGPLAN Not.**, Association for Computing Machinery, New York, NY, USA, v. 46, n. 6, p. 164–174, June 2011. ISSN 0362-1340. DOI: `10.1145/1993316.1993518`. Available from: <`https://doi.org/10.1145/1993316.1993518`>. Cit. on p. 20.

SCARABOTTOLO, I. et al. Approximate Logic Synthesis: A Survey. **Proceedings of the IEEE**, p. 1–19, 2020. DOI: `10.1109/JPROC.2020.3014430`. Available from: <`https://doi.org/10.1109/JPROC.2020.3014430`>. Cit. on p. 20.

SEKANINA, Lukas et al. Image processing and CGP. In: CARTESIAN genetic programming. [S.l.]: Springer, 2011. P. 181–215. Cit. on p. 22.

SUGANUMA, Masanori et al. Evolution of Deep Convolutional Neural Networks Using Cartesian Genetic Programming. **Evolutionary Computation**, v. 28, n. 1, p. 141–163, Mar. 2020. DOI: `10.1162/evco_a_00253`. eprint: `https://direct.mit.edu/evco/article-pdf/28/1/141/1858844/evco\_a\_00253.pdf`. Available from: <`https://doi.org/10.1162/evco%5C_a%5C_00253`>. Cit. on p. 22.

TESTA, Eleonora; AMARÚ, Luca, et al. Extending Boolean Methods for Scalable Logic Synthesis. **IEEE Access**, IEEE, v. 8, p. 226828–226844, 2020. Cit. on p. 19.

TESTA, Eleonora; SOEKEN, Mathias, et al. Logic synthesis for established and emerging computing. **Proceedings of the IEEE**, IEEE, v. 107, n. 1, p. 165–184, 2018. Cit. on pp. 12, 15, 16, 18, 19.

VASICEK, Zdenek; SEKANINA, Lukas. Evolutionary approach to approximate digital circuits design. **IEEE Transactions on Evolutionary Computation**, IEEE, v. 19, n. 3, p. 432–444, 2014. Cit. on p. 22.

VENKATESAN, Rangharajan et al. MACACO: Modeling and analysis of circuits for approximate computing. In: IEEE. 2011 IEEE/ACM International Conference on Computer-Aided Design (ICCAD). [S.l.: s.n.], 2011. P. 667–673. Cit. on pp. 19, 20.

YAZDANBAKHSH, Amir et al. Axilog: Language support for approximate hardware design. In: IEEE. 2015 Design, Automation & Test in Europe Conference & Exhibition (DATE). [S.l.: s.n.], 2015. P. 812–817. Cit. on p. 20.

Appendix

APPENDIX A – ARTICLE PUBLISHED IN THE THE 26TH EUROPEAN CONFERENCE ON GENETIC PROGRAMMING (EUROGP) 2023

A.1 ADAPTIVE BATCH SIZE CGP: IMPROVING ACCURACY AND RUNTIME FOR CGP LOGIC OPTIMIZATION FLOW

Abstract: With the recent advances in the Machine Learning field, alongside digital circuits becoming more complex each day, machine learning based methods are being used in error-tolerant applications to solve the challenges imposed by large integrated circuits, where the designer can obtain a better overall circuit while relaxing its accuracy requirement. One of these methods is the Cartesian Genetic Programming (CGP), a subclass of Evolutionary Algorithms that uses concepts from biological evolution applied in electronic design automation. CGP-based approaches show advantages in the logic learning and logic optimization processes. However, the main challenge of CGP-based flows is the extensive runtime compared to other logic synthesis strategies. We propose a new strategy to tackle this challenge, called Adaptive Batch Size (ABS) CGP, in which the CGP algorithm incrementally improves the fitness estimation of the candidate solutions by using more terms of the truth table for evaluating them along the evolutionary process. The proposed approach was evaluated in nine exemplars from the IWLS 2020 contest, in which 3 exemplars are from the arithmetic domain, and six are from image recognition domain, specifically three from the CIFAR-10 dataset and three from the MNIST dataset. The results show that ABS presented an accuracy increase of up to 8.19% and decreased the number of candidate solutions evaluations required by up to 84.56%, in which directly affects the runtime of the algorithm. Furthermore, for all circuits, no significant accuracy reduction was observed while a significant reduction in the number of evaluations was achieved.

Adaptive Batch Size CGP: Improving Accuracy and Runtime for CGP Logic Optimization Flow

Bryan Martins Lima(✉), Naiara Sachetti, Augusto Berndt, Cristina Meinhardt, and Jonata Tyska Carvalho

Federal University of Santa Catarina - UFSC, Florianópolis, Brazil
{bryan.l,naiara.sachetti}@grad.ufsc.br, augusto.berndt@posgrad.ufsc.br, {cristina.meinhardt,jonata.tyska}@ufsc.br

Abstract. With the recent advances in the Machine Learning field, alongside digital circuits becoming more complex each day, machine learning based methods are being used in error-tolerant applications to solve the challenges imposed by large integrated circuits, where the designer can obtain a better overall circuit while relaxing its accuracy requirement. One of these methods is the Cartesian Genetic Programming (CGP), a subclass of Evolutionary Algorithms that uses concepts from biological evolution applied in electronic design automation. CGP-based approaches show advantages in the logic learning and logic optimization processes. However, the main challenge of CGP-based flows is the extensive runtime compared to other logic synthesis strategies. We propose a new strategy to tackle this challenge, called Adaptive Batch Size (ABS) CGP, in which the CGP algorithm incrementally improves the fitness estimation of the candidate solutions by using more terms of the truth table for evaluating them along the evolutionary process. The proposed approach was evaluated in nine exemplars from the IWLS 2020 contest, in which 3 exemplars are from the arithmetic domain, and six are from image recognition domain, specifically three from the CIFAR-10 dataset and three from the MNIST dataset. The results show that ABS presented an accuracy increase of up to 8.19% and decreased the number of candidate solutions evaluations required by up to 84.56%, in which directly affects the runtime of the algorithm. Furthermore, for all circuits, no significant accuracy reduction was observed while a significant reduction in the number of evaluations was achieved.

Keywords: Logic synthesis · Cartesian Genetic Programming (CGP) · Evolutionary algorithms · Approximate Computing

1 Introduction

Logic optimization is an initial task when converting an abstract specification of a digital circuit in terms of logic gates. It focus on reducing the number

G. Pappa et al. (Eds.): EuroGP 2023, LNCS 13986, pp. 149–164, 2023.
https://doi.org/10.1007/978-3-031-29573-7_10

of logic elements, including nodes and logic depth, used for producing a specific input-output mapping (function). These optimizations will influence future steps in the design flow, improving circuit area, delay and energy consumption [23]. Traditional logic optimization methods simplify a Boolean function, exploring exact logic minimization techniques like the Algebraic method, the Karnaugh map technique [12], or the Quine-McCluskey method [22]. However, the main limitation of the traditional logic optimization methods is the number of inputs that they can deal with. For instance, the Quine-McCluskey method is limited to functions with up to 15 variables [6]. As for real-world applications, Espresso is used for the simplification of circuits with many inputs reaching faster results by exploring sub-optimal heuristic methods [5,25]. These fast simplification methods provide a trade-off for computing performance at the cost of output quality.

Some new logical optimization flows address fast logic optimization based on machine learning approaches like decision trees [1]. However, many of them fail to deal with scaling the logic function complexity. For example, decision tree solutions, in general, must expand all the input combinations, which becomes prohibitive for large inputs. To deal with the expensive rising of complexity on logic functions, neural networks [19] and evolutionary algorithms such as Cartesian Genetic Programming (CGP) [3,4] have been recently studied. An important drawback of these approaches is the large runtime compared to the traditional logic synthesis approaches.

Most of these new optimization flows are particularly interesting when applied to Approximate Computing. The traditional methods, such as Espresso, struggle when problems have an incomplete specification, as this method was designed to tackle exact minimization. In this scenario, the novel optimization flows are becoming an alternative when specifications for problems do not require an exact circuit. In this case, using CGP with Approximate Computing paradigm can be a good alternative to traditional methods [4].

This paradigm have been mainly used to design power-efficient solutions for error-tolerant applications [26,29]. Examples of error-tolerant applications include those using image and sound data, video processing, sensors for Internet of Things, and neural networks. As these applications have less strict accuracy requirements of the implemented functions, it is possible to focus on a smaller circuit, which can improve power and delay [2].

Furthermore, we can use Approximate Computing to generalize a circuit based on a few selected samples. This was one of the goals of the International Workshop on Logic and Synthesis (IWLS) Contest in 2020 [23], in which multiple teams of different countries competed to generalize logic functions. The teams used multiple strategies, including Espresso [25], multi-layer perceptrons (MLP), random forests, lookup-table (LUT) networks [19] as well as CGP [4]. Overall, the presented results confirmed that sacrificing some accuracy was possible to achieve a significantly smaller circuit, as well as many of them used a combination of these strategies; therefore, none of these strategies dominated the others.

One of the promising flows proposed in the IWLS was CGP [4], which presented reasonable accuracy performance but with a long runtime for synthesizing the circuits. This flow can be used to optimize already synthesized circuits as well as synthesize a new circuit. In this context, we present a new technique called Adaptive Batch Size (ABS) CGP which aims to reduce the number of evaluations required for synthesizing and optimizing circuits, while maintaining or improving the accuracy compared to the standard CGP. The proposed strategy dynamically increases the number of selected terms of the truth table used for evaluating the individuals during the evolutionary process. By evaluating the proposed strategy on a subset of the IWLS contest, the results indicate a reduction of up to 84.6% on the number of evaluations needed compared to the standard CGP flow. Furthermore, some circuits presented increased accuracy of up to 8.2%, while no synthesized circuit presented an accuracy decrease.

This paper is organized as follows. Section 2 presents the Cartesian Genetic Programming algorithm. In Sect. 3 is described our methodology detailing the strategy proposed. In Sect. 4 there is a discussion on the data collected. Finally, Sect. 5 concludes this work and summarizes our findings.

2 Cartesian Genetic Programming

The CGP is a form of genetic programming that uses a graph representation to encode computer programs. It is called ‘Cartesian’ because it represents a program using a two-dimensional grid of nodes [18]. It was created by Julian F. Miller in 1999 to encode digital circuits [18]. It is capable of representing math equations, circuits and computer programs as a directed acyclic graph. The CGP is bio-inspired by concepts from genetics, that are used for building the meta-heuristic responsible for the optimization process and program/solution synthesis. For this, genes are the integers that form genotypes of a node, thus, they represent the input, operation and output of the given node.

There are many applications where CGP is used. It has been used in image filters and image processing [10,11,27], it can be used to encode Artificial Neural Networks [13,14] and to optimize the learning of Convolutional Neural Networks [28], as well as in cryptography field [20,21]. For further discussion on its applications see the works of [16,18].

2.1 Representation

Figure 1 shows the representation of a CGP individual as a 1-line array, instead of a 2-dimensional grid. The 1-line array was selected as it has faster convergence in the evolutionary search as demonstrated in [17]. However, both representations are equivalent for digital circuits, as they can be represented either way.

The 1-line array representation is an array of 4-tuples composed of two pairs, one pair for each input of the logic gate. It is noteworthy that the inputs are indexed as well as the individual nodes. Hence, there are 6 indexes representing 3 primary inputs and 3 nodes. The first pair (a, b), represents the first input, in which a represents if the input is inverted or not, and b representing from which

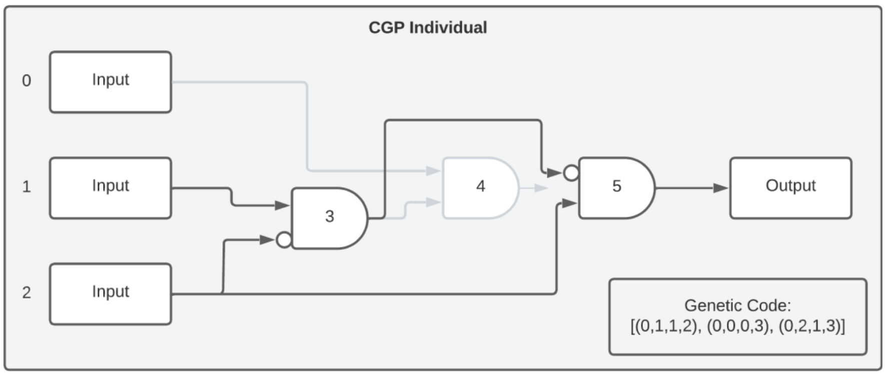


Fig. 1. CGP individual representation. Adapted from: [4]

index to get the input from; similarly, the second pair represents the second input.

Taking node 3 as an example, its first pair, the input is not inverted and it came from the primary input 1, which is indexed as 1; thus, the first pair is $(0, 1)$. For its second pair, the input is inverted and it came from the primary input 2, indexed as 2; therefore, the second pair is $(1, 2)$. Therefore, the complete genetic code for node 3 is $(0, 1, 1, 2)$. There are two main considerations for this representation. The first one is that the input of a given node can be a primary input, i.e. (0, 1, 2), or an output of another node, i.e. (3, 4, 5). It is important to notice that backward connections are not allowed so, for instance, node 4 can receive inputs from node 3, but not from node 5. The second one regards the meaning of the term functional node. Note that, in this simple example, node 4 is represented in gray color in Fig. 1. That is because it is not a functional node, i.e. it does not have a connection to the output of the CGP individual. The functional part of the CGP individual is called the *phenotype*, and the genetic code, is called *genotype*. Is noteworthy, that the number of primary inputs, primary outputs and the number of nodes is fixed through the whole evolutionary process.

It is important to note that we used AND and Inverter gates; thus, it represents a AND-Inverter graph or AIG. It is noteworthy that any logic function can be described with only AND and Inverter logic gates.

2.2 Evolutionary Process

Figure 2 presents a hypothetical CGP search that uses the evolutionary approach (1 + 4), in which, at each generation, a parent circuit generates four offspring. These offspring circuits are a mutated copy of the parent's circuit, i.e., a copy that had some of its connections mutated by chance according to a given probability, the mutation rate. Direct connections are represented by a straight line and inverted connections are represented by dotted lines.

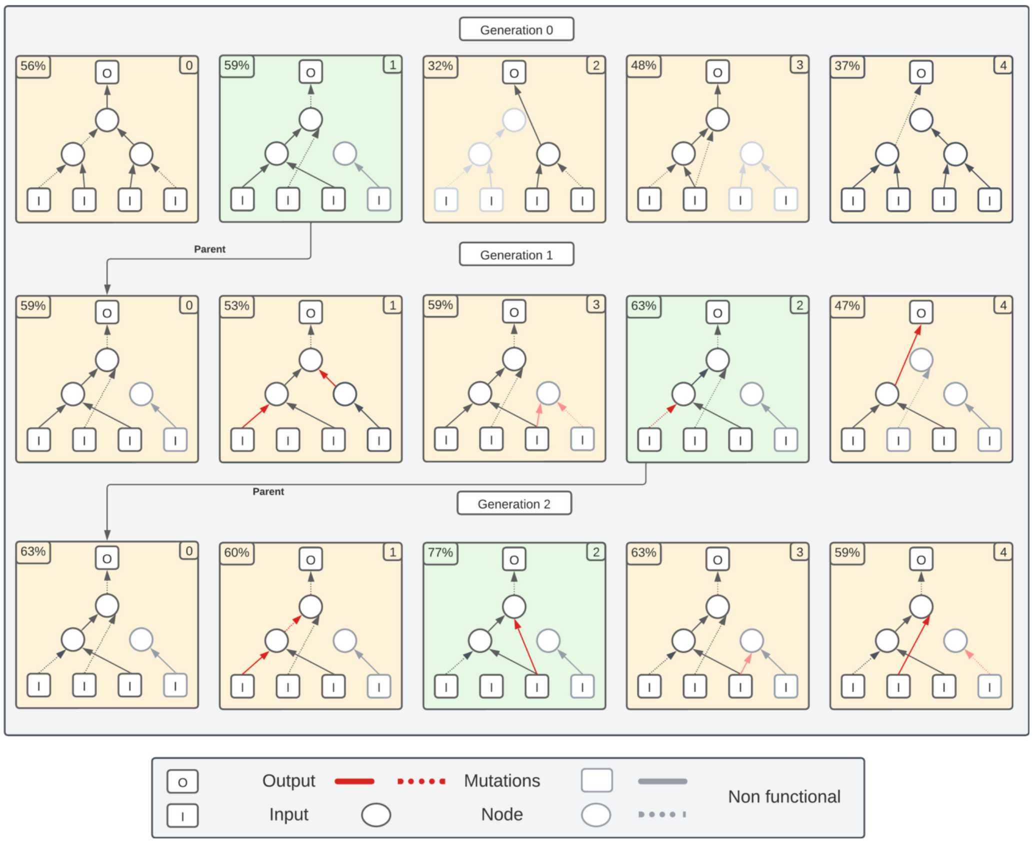


Fig. 2. Example of an evolutionary process of the CGP search.

As can be seen in Fig. 2, in generation 0, individuals are randomly generated, and each circuit has a different accuracy (fitness value). The parent circuit of the next generation is highlighted in green. The circuit with the highest accuracy is selected as the parent of the next generation. Therefore, regarding generation 0, the next parent is the circuit with 59% of accuracy. In generation 1, this parent generates one unmodified copy and four mutated copies, i.e., copies with a chance of having different connections. Mutations are highlighted in red. The same process repeats for generation 2 and so on, until reaching the maximum number of generations defined by the user.

In this example, for sake of clarity, we used a simpler representation of a CGP individual when compared to Fig. 1. We used this simpler representation to focus on the learning of the search rather then focus in the individuals. Moreover, it is important to note that each individual had 3 nodes; however, the number of nodes is generally hundreds or even thousands.

Furthermore, as demonstrated by [17], the CGP search tends more easily to escape from local-optima regions, so avoiding premature convergence, if phenotypically larger solutions are considered preferred candidates when analyzing the individual's fitness accuracy due to higher genetic variation. This happens since artificial evolution tends to select genotypes that are robust to mutations, i.e., that are less likely to produce maladaptive mutations. This, in turn, pushes in the direction of genotypes with tiny functional circuits, since the genotype becomes robust to any mutation that does not affect the functional part of the circuit, and this characteristic leads to premature convergence. The authors demonstrate that this limitation could be eliminated by selecting individuals with larger functional sizes when they have the same fitness. Therefore, in this work, individuals with larger functional sizes are selected when a tie in fitness values happens.

3 Methodology

In our implementation, we explore the utilization of AIGs, which are the state-of-the-art data structure for technology-independent optimizations during logic synthesis [24]. An AIG is a directed acyclic graph composed of nodes representing AND gates and edges representing inverted or directed connections. An AIG node is composed of exactly two inputs and an arbitrary number of outputs. An AIG may represent any logic function. In our CGP implementation, an AIG is represented as an individual from the CGP population, and CGP mutations concern modifying the AIG connections and inversions.

To better explain our proposed flow and experiments, we present four definitions before detailing two important parts of our work. Then, in Sect. 3.3 we present the hyperparameters used to run the CGP implementation and analyze their impact in the CGP synthesis.

3.1 Definitions

The following definitions describe fundamental concepts in our approach.

Definition 1 (Batch size): The actual number of terms used for the individuals' evaluation is called *batch size* (BS).

Definition 2 (Adaptive Interval (σ)): represents the interval, in number of generations, in which ABS CGP will use to monitor the evolution stagnation.

Definition 3 (Mutation): to explore the search space, the circuit needs to change. For this, to add variability to the system, mutations occur in the AIG nodes, as demonstrated by red drawings in Fig. 2. Mutations during generations happen according to the mutation rate value, which is given by the $1/5^{th}$ rule [8]. When a mutation happens, one of the values of the 4-tuple representing the inputs and input-inversions of a node is randomly changed. Notice that backward connections are not allowed, so for a given node i, the mutated value will be drawn from a uniform distribution ranging from 0 to $i-1$.

Definition 4 (Fitness): in this work, fitness means the capability of a circuit to produce the correct output given a set of terms (accuracy). Meaning that Boolean signals are propagated and binary outputs achieved are checked if they match with the expected value.

3.2 Adaptive Batch Size CGP

Traditionally, CGP evaluates all the individuals every generation after they are created by the parent mutation. The evaluation process for each of them requires processing all the training batch inputs. This is done with a depth-first search along the AIG. In other words, the evaluation has a time complexity of $O(i*n*b)$, with i being the number of individuals in the CGP population, n the size of each individual, and b the batch size (BS) used for training.

Figure 3 presents a flowchart of a simplified version of the proposed mechanism for the CGP technique. First, ABS CGP starts with an initial batch size of β terms. The next step is choosing a criterion for defining and detecting evolution stagnation, and by doing so increasing the batch size when it happens. The initially chosen criterion was a Simple Moving Average (SMA) from the accuracy of the synthesized circuits considering a window of σ generations.

For the next step, there is a verification step that compares the current SMA to the previous SMA. By always saving the previous SMA calculated we can compare this value with the SMA of the current generation. The comparison of both values provides us with the information that CGP is actually learning the function at hand. Therefore, if these values are the same, there were no improvements in accuracy in the last $2 * \sigma$ generations. This is the core of our strategy, if the CGP is not improving the circuit's accuracy, i.e. the estimation quality is stagnant, exposing the algorithm to more terms could help broaden its search. By using a BS smaller than the whole truth table, stochasticity in the evaluation process is inherently added. The smaller the BS value, the faster is the evaluation process and higher is the stochasticity. Therefore, there is a moment in which the individuals' fitness reached a point that, due to stochasticity, the algorithm cannot differ which are the best CGP individuals anymore. Our proposed strategy tries to solve this by increasing the number of terms when it detects that this situation occurs, reducing the stochasticity in the evaluation, and by doing so, improving the quality of the fitness estimation.

Before actually increasing the batch size, there is another verification checking if increasing the batch will not surpass the size of the whole data available. If this verification yields true, i.e. increasing the batch does not surpass the complete data, then the batch is increased by α terms and the SMA is calculated again after σ generations. If the previously mentioned verification yields false,

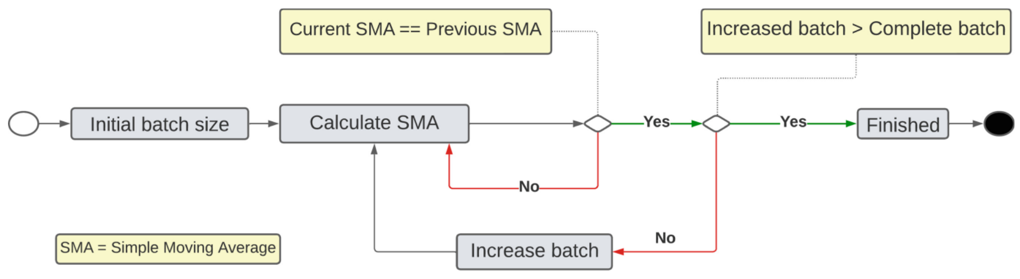

Current SMA == Previous SMA
Increased batch > Complete batch
Initial batch size
Calculate SMA
Yes
Yes
Finished
No
No
SMA = Simple Moving Average
Increase batch

Fig. 3. Adaptive Batch Size CGP flowchart

Table 1. Exemplars details

Exemplar	Logic function	# of inputs
40, 41, 43	LSB of n-square root	16, 10, 18
80, 81, 84	MNIST	196
93, 98, 99	CIFAR-10	768

that means the algorithm is almost or already using the complete data available; thus, increasing the batch would surpass all data available. In this case, the whole data available is chosen as the batch size of the next generation, and there is no need for any more steps, as the CGP search is already using the maximum amount of data available. After this step, the CGP continues its search as the standard version.

It is important to note that the value of the BS directly impacts the runtime. As the search algorithm will utilize this value to measure the synthesized circuit's accuracy. Therefore, evaluating a circuit's accuracy with a BS of 128 is computationally cheaper than evaluating with a BS of 1024.

Furthermore, this work performed an improvement regarding the evaluation process using the CGP C++ implementation[1] used in [4]. The main difference in our version is when to evaluate the nodes accuracy, as this process is computationally costly. For this, we used the proposed strategy "Skip" in [9] which means skipping the evaluation of individuals in which the functional nodes of their parent were not mutated. This change was not only better for performance, but it was necessary to be possible to evenly compare the multiples hyperparameters used in CGP. The "Skip" strategy does not affect by any means the performance of the circuits synthesized by the algorithm, it only prevents unnecessarily evaluating nodes that certainly have the same accuracy as their parent, thus, it does not waste time evaluating an already known accuracy.

Hence, by avoiding unnecessary evaluations and dynamically increasing the number of terms for evaluating the candidate solutions during CGP search, we reduce the number of evaluations required for synthesizing circuits, and consequently reduce runtime.

3.3 Experimental Protocol

To compare ABS CGP with the standard CGP, we chose exemplars from the IWLS 2020 contest. Specifically, we chose those in which CGP had a bigger difference of accuracy when compared to the best teams in the IWLS contest when using different strategies. In other words, those in which CGP had a worse accuracy than the others strategies; therefore, CGP could still have a margin of improvement. Thus, the selected exemplars were: ex40, ex41, ex43, ex80, ex81, ex84, ex93, ex98, ex99. As ex41 was our proof of concept for the ABS strategy, we selected two similar exemplars for comparison.

[1] Source code available in: https://gitlab.com/gudeh/cgpv3.

Table 2. Hyperparameters tested

Parameter	Value
Number of generations	50,000
Number of nodes	1,000
Number of seeds	10
Evolutionary Strategy	(1 + 4)
ABS CGP-specific	
Adaptive interval (σ)	100, 250, 500, 1000, 2500, 5000
Initial batch size (β)	64
Increase batch factor (α)	64
Standard CGP-specific	
Batch size	6400

Table 1 presents more details upon the exemplars chosen for validating the proposed algorithm. The nine exemplars are from arithmetic - ex40, ex41 and ex43 - and image recognition domains - MNIST [7] and CIFAR-10 [15]. It is noteworthy that the image recognition functions are the largest in the contest; furthermore, the contest only provided 6,400 inputs for all exemplars, which only represent a tiny portion of all possible combinations for MNIST and CIFAR, 2^{196} and 2^{768} combinations respectively.

Table 2 shows the hyperparameters used in our experiments. We used 50,000 generations to identify the initial impacts on the learning of the circuit. We used a number of nodes of 1,000, which means that the CGP has a maximum of 1,000 functional nodes. Moreover, we replicated each experiment ten times with different random seeds. As this algorithm is impacted by different sources of stochasticity, for instance, its initial randomly generated circuits, ten runs provided enough information for the statistical validation of the results. These variations in the accuracy caused by stochasticity will be noticeable on the box plots presented in Sect. 4.

For the ABS CGP-specific hyperparameters, we varied the AI parameter between 0.2% and 10% of the total evolutionary process of 50,000 generations to analyze its impact. Due to the high computational cost of optimizing all hyper-parameters, we decided to fix the values for the initial batch size and the increased batch factor, which were fixed to the value of 64. Investigating the impact of varying these values on the performance of the ABS CGP is relevant and planned as future work. To compare the proposed strategy to the standard CGP version the value of the BS was fixed with 6,400 terms, as it was the maximum value for number of terms for all exemplars of the IWLS contest. Along this, we used the same 50,000 generations with 1,000 nodes and ten seeds for the control case.

Finally, instead of using the algorithm runtime as one of the comparison metrics, we use the number of candidate solutions evaluations performed during the evolutionary process. This is a standard metric within the evolutionary computing community since it removes any hardware-related differences among studies. To measure the total number of evaluations we use the sum over the generations of the result of the multiplication between the BS and the offspring in each generation. Equation 1 presents the complete equation.

$$Evaluations = \sum_{i=1}^{i=N} BS_i * offspring \tag{1}$$

As we use Eq. 1 to calculate the number of evaluations, it is clear that the BS value is the one that impacts the number of evaluations the most. As the number of generations is fixed and the $offspring$ variable is independent of the ABS strategy, the only variable that is dependent on the proposed strategy is the BS value.

4 Results

Our results show that in most cases the ABS approach not only increased the accuracy but also led to a significant reduction in the number of evaluations performed during the evolutionary search. Figure 4 presents each circuit accuracy when using all available terms of the truth table versus our approach with ABS CGP. The results of ABS consider the best set of hyperparameters for each exemplar, considering the best AI values, presented in Table 3.

Table 3. Best AI values for each exemplar of Fig. 4 and Fig. 5

Exemplar	Adaptive interval
ex40	100
ex41	500
ex43	100
ex80	250
ex81	100
ex84	250
ex93	250
ex98	100
ex99	250

Figure 5 presents the number of evaluations to learn the circuits and Table 4 presents their average values from the ten different seeds used. Furthermore, Table 5 presents the gain/loss in accuracy and the number of evaluations for each exemplar between ABS CGP and the standard CGP using all data available. Exemplars from the arithmetic domain had the best-improved accuracy

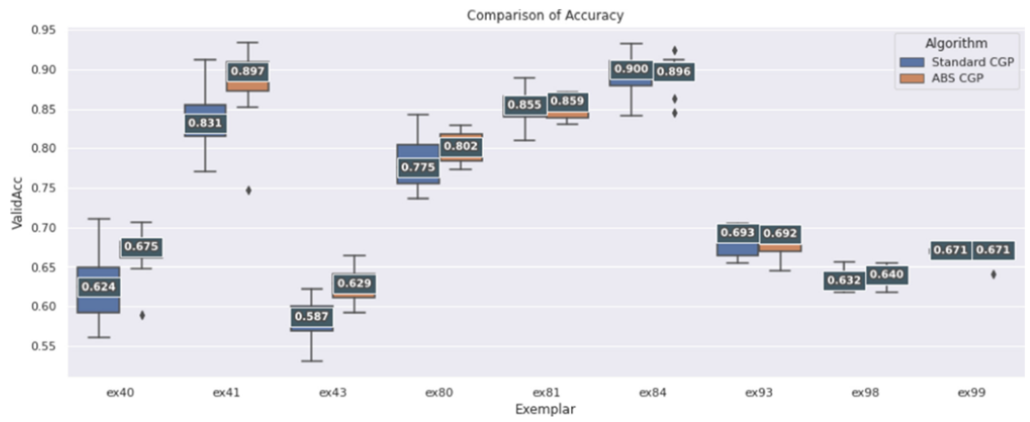


Fig. 4. Accuracy of Standard CGP and ABS CGP

Table 4. Average number of evaluations (in 10^7) of Fig. 5

Exemplar	Complete version	Adaptive version
ex40	123.00 ± 1.99	70.18 ± 10.2
ex41	126.17 ± 1.47	18.80 ± 2.15
ex43	122.10 ± 1.93	62.45 ± 10.4
ex80	121.67 ± 2.70	32.06 ± 3.02
ex81	123.11 ± 3.02	79.38 ± 5.64
ex84	122.67 ± 2.97	35.39 ± 5.95
ex93	119.18 ± 0.08	37.31 ± 3.85
ex98	119.12 ± 0.09	85.56 ± 3.85
ex99	119.15 ± 0.09	35.21 ± 3.12

of all nine exemplars tested, with a statistically significant accuracy increase of 7.74% on average (Mann-Whitney U Test, p-values of .02, .03 and .001 for exemplars ex40, ex41 and ex43, respectively), while decreasing 57.96% on average the number of evaluations performed (Mann-Whitney U Test, $p < .001$ for all exemplars). As for the exemplars from the image recognition domain, the accuracy improvement was 0.792% in average, which was not statistically significant (Mann-Whitney U Test, p-values of .16, .909, .71, .569, .87 and .939, for exemplars ex80, ex81, ex84, ex93, ex98 and ex99, respectively). Despite not achieving a significant improvement for the exemplars ex80 through ex99, the accuracy level obtained by the control case was maintained, while decreasing the evaluations performed by 58.015% on average for these exemplars (Mann-Whitney U Test, $p < .001$ for all exemplars). From these results, it seems that ABS CGP is capable of synthesizing circuits with better accuracy in the arithmetic domain. Although more exemplars of this type and further analysis are needed to better understand why this happens for this particular family of functions.

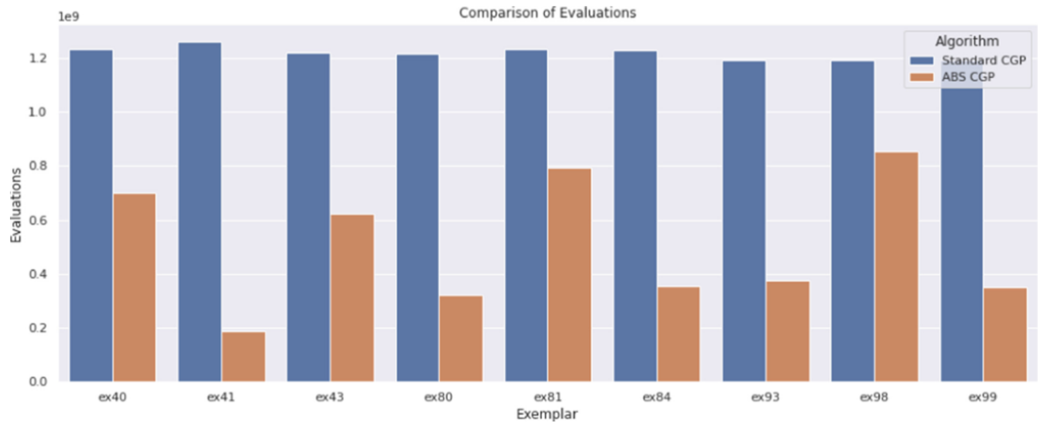


Fig. 5. Number of evaluations between ABS CGP and Standard CGP

Another noteworthy insight from these experiments are the values of AI. As we can see in Table 3, most of the AI values are in the lower range of the chosen AI parameters of Table 2. Only the AI values of 100 and 250 managed to reach a point in the evolutionary process in which all available lines were used for evaluating individuals in all the experiments. Furthermore, the results with AI above 500 had worse accuracy compared to the results presented herein. This indicates that more generations could be executed on the search since with these AI values the CGP search could not reach the complete batch size. Further investigation is required to analyze if by increasing the number of generations the evolutionary processes using these higher AI values present better results with respect to using lower AI values as they would have more time to reach the maximum size. Moreover, using a fixed size of α, which controls by how much the batch will be increased, may be too low. In our set of experiments we used a value of α representing only 1% of the truth table which could be optimal for some exemplars, but not others. Overall, this data indicates that there might be room for improvement if this hyperparameter would be optimized as well.

To further investigate the learning curve of the ABS CGP evolutionary search we checked how the fitness of the best individuals increased through generations. Besides collecting the partial accuracy of the circuit in each generation, we collected its validation accuracy as well, i.e. the accuracy using the validation set. Figure 6 presents this data for the initial evolutionary search for exemplar 41. The blue line represents the search using all terms available when using the standard CGP. The green line represents the search for the Adaptive version with AI set to 100. Hence, the first generations are using a smaller portion of the truth table, and the last ones are using all terms available. The orange line is representing the accuracy of the same Adaptive version of 100 AI while using all terms from the beginning. For this reason, the orange line and green line converge at the end.

It is noteworthy that our first assumption for the ABS version was that ABS CGP could improve faster at the beginning, even when using a rough fitness evaluation of the candidate solutions, i.e., using fewer lines for evaluating each

Table 5. Adaptive CGP gains over standard CGP

Exemplar	Accuracy gain/lost	Evaluation reduction
ex40	+8.186%[1]	−44.180%[1]
ex41	+7.890%[1]	−84.563%[1]
ex43	+7.143%[1]	−45.125%[1]
ex80	+3.443 %	−73.617%[1]
ex81	+0.550 %	−35.343%[1]
ex84	−0.399 %	−72.641%[1]
ex93	−0.144 %	−68.823%[1]
ex98	+1.217 %	−26.839%[1]
ex99	+0.029 %	−70.827%[1]
Average	3.102%± 3.344	−57.995% ± 19.110

[1] Statistically significant difference (Mann Whitney-U with $p < .05$).

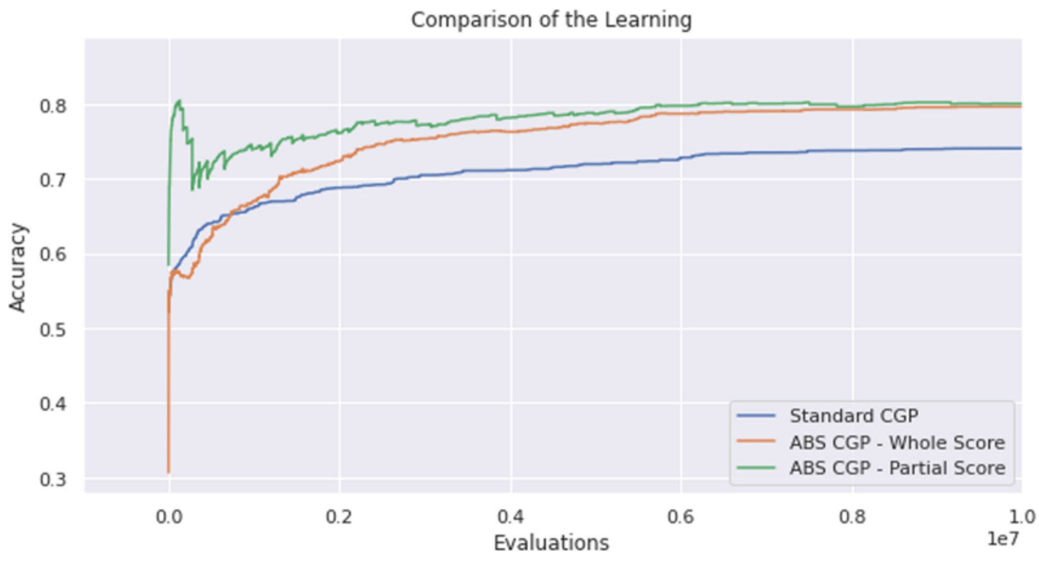


Fig. 6. Accuracy during learning for ex41

circuit, and performing the fine-tuning of circuits at the end, using the whole data available. Indeed, the learning of the ABS CGP (orange line) managed to achieve better accuracy in these initial evaluations compared to the complete version even with a worst fitness estimation than the version using all data available for evaluating individuals. Furthermore, ABS kept improving the circuits beyond the performance achieved by the standard version, providing a better final result.

5 Conclusion

Due to the increasing complexity of digital circuits novel techniques for logic synthesis are required. The combination of logic synthesis and machine learning has been gaining much attention in the literature due to the promising results regarding circuit area reduction and power consumption, mainly in approximate computing domains. Among these techniques, Cartesian Genetic Programming is capable of synthesizing better circuits than traditional techniques in certain domains, but at the cost of demanding a great computational power, which slows down the synthesis. This work proposes a novel technique called Adaptive Batch Size CGP seeking to alleviate the computational resources required by CGP, and by doing so, improve the runtime and accuracy of the synthesized solutions. Our results confirmed that by incrementally improving the fitness estimation along the evolutionary process, we can obtain significant improvements in the CGP convergence speed and in the quality of the synthesized circuits. For all experiments performed, the ABS CGP achieved better, or at least the same, accuracy as the base version of the algorithm while performing significantly fewer evaluations, and being significantly faster. In a number of cases of the presented results, the final BS did not reach a point in the evolutionary process in which all data available was used for evaluating the individuals. These results indicate that some particular combinations of data available, number of generations, and the AI were not optimal for certain cases. Moreover, the current version of ABS CGP has a fixed initial batch size and in all experiments, this value was set to 64 terms. Therefore, as future work, investigating how the initial batch size, the parameter α that controls the batch increase, and the number of generations affect the synthesis for each particular exemplar is an interesting research direction.

Acknowledgments. This work was financed in part by National Council for Scientific and Technological Development - CNPq and the Propesq/UFSC.

References

1. de Abreu, B.A., et al.: Fast logic optimization using decision trees. In: 2021 IEEE International Symposium on Circuits and Systems (ISCAS), pp. 1–5 (2021). https://doi.org/10.1109/ISCAS51556.2021.9401664
2. Barua, H.B., Mondal, K.C.: Approximate computing: a survey of recent trends—bringing greenness to computing and communication. J. Inst. Eng. (India): Series B **100**(6), 619–626 (2019). https://doi.org/10.1007/s40031-019-00418-8
3. Berndt, A., et al.: Accuracy and size trade-off of a cartesian genetic programming flow for logic optimization. In: Proceedings of the 34th Symposium on Integrated Circuits and Systems Design. SBCCI 2021 (2021). https://doi.org/10.1109/SBCCI53441.2021.9529968
4. Berndt, A.A.S., et al.: A CGP-based logic flow: optimizing accuracy and size of approximate circuits. J. Integr. Circ. Syst. **17**(1), 1–12 (2022). https://doi.org/10.29292/jics.v17i1.546
5. Brayton, R.K., Hachtel, G.D., Mcmullen, C.T., Sangiovanni-Vincentelli, A.L.: Logic Minimization Algorithms for VLSI Synthesis. Kluwer Int. Ser. Eng. Comput. Sci. **2**, 1–194 (1984). https://doi.org/10.1007/978-1-4613-2821-6
6. Coudert, O., Sasao, T.: Two-level logic minimization. In: Hassoun, S., Sasao, T. (eds.) Logic Synthesis and Verification. SECS, vol. 654, pp. 1–27. Springer, Boston (2002). https://doi.org/10.1007/978-1-4615-0817-5_1

7. Deng, L.: The MNIST database of handwritten digit images for machine learning research. IEEE Signal Process. Mag. **29**(6), 141–142 (2012). https://doi.org/10.1109/MSP.2012.2211477
8. Doerr, B., Doerr, C.: Optimal parameter choices through self-adjustment: applying the 1/5-th rule in discrete settings. In: Proceedings of the 2015 Annual Conference on Genetic and Evolutionary Computation, pp. 1335–1342 (2015). https://doi.org/10.1145/2739480.2754684
9. Goldman, B.W., Punch, W.F.: Analysis of cartesian genetic programming's evolutionary mechanisms. IEEE Trans. Evol. Comput. **19**(3), 359–373 (2014)
10. Harding, S.: Evolution of image filters on graphics processor units using cartesian genetic programming. In: 2008 IEEE Congress on Evolutionary Computation (IEEE World Congress on Computational Intelligence), pp. 1921–1928. IEEE (2008)
11. Harding, S., Leitner, J., Schmidhuber, J.: Cartesian genetic programming for image processing. In: Riolo, R., Vladislavleva, E., Ritchie, M., Moore, J. (eds.) Genetic programming theory and practice X. (GEVO), pp. 31–44. Springer, Cham (2013). https://doi.org/10.1007/978-1-4614-6846-2_3
12. Karnaugh, M.: The map method for synthesis of combinational logic circuits. Trans. Am. Inst. Electr. Eng. Part I: Commun. Electron. **72**(5), 593–599 (1953). https://doi.org/10.1109/TCE.1953.6371932
13. Khan, M.M., Ahmad, A.M., Khan, G.M., Miller, J.F.: Fast learning neural networks using cartesian genetic programming. Neurocomputing **121**, 274–289 (2013)
14. Khan, M.M., Khan, G.M., Miller, J.F.: Evolution of neural networks using cartesian genetic programming. In: IEEE Congress on Evolutionary Computation, pp. 1–8. IEEE (2010)
15. Krizhevsky, A., Hinton, G., et al.: Learning multiple layers of features from tiny images (2009)
16. Manazir, A., Raza, K.: Recent developments in cartesian genetic programming and its variants. ACM Comput. Surv. (CSUR) **51**(6), 1–29 (2019). https://doi.org/10.1145/3275518
17. Milano, N., Pagliuca, P., Nolfi, S.: Robustness, evolvability and phenotypic complexity: insights from evolving digital circuits. Evol. Intel. **12**(1), 83–95 (2019). https://doi.org/10.1007/s12065-018-00197-z
18. Miller, J.F.: Cartesian genetic programming: its status and future. Genet. Program Evolvable Mach. **21**(1), 129–168 (2019). https://doi.org/10.1007/s10710-019-09360-6
19. Miyasaka, Y., Zhang, X., Yu, M., Yi, Q., Fujita, M.: Logic synthesis for generalization and learning addition. In: 2021 Design, Automation Test in Europe Conference Exhibition (DATE), pp. 1032–1037 (2021). https://doi.org/10.23919/DATE51398.2021.9474169
20. Picek, S., Carlet, C., Guilley, S., Miller, J.F., Jakobovic, D.: Evolutionary algorithms for Boolean functions in diverse domains of cryptography. Evol. Comput. **24**(4), 667–694 (2016)
21. Picek, S., Jakobovic, D., Miller, J.F., Batina, L., Cupic, M.: Cryptographic Boolean functions: one output, many design criteria. Appl. Soft Comput. **40**, 635–653 (2016)
22. Quine, W.V.: A way to simplify truth functions. Am. Math. Mon. **62**(9), 627–631 (1955). https://doi.org/10.1080/00029890.1955.11988710
23. Rai, S., et al.: Logic synthesis meets machine learning: trading exactness for generalization. In: 2021 Design, Automation & Test in Europe Conference & Exhibition (DATE). IEEE (2021). https://doi.org/10.23919/DATE51398.2021.9473972

24. Riener, H., Haaswijk, W., Mishchenko, A., De Micheli, G., Soeken, M.: On-the-fly and DAG-aware: rewriting Boolean networks with exact synthesis. In: 2019 Design, Automation Test in Europe Conference Exhibition (DATE), pp. 1649–1654 (2019). https://doi.org/10.23919/DATE.2019.8715185
25. Rudell, R.L., Sangiovanni-Vincentelli, A.: Multiple-valued minimization for PLA optimization. IEEE Trans. Comput. Aided Des. Integr. Circ. Syst. **6**(5), 727–750 (1987). https://doi.org/10.1109/TCAD.1987.1270318
26. Scarabottolo, I., et al.: Approximate logic synthesis: a survey. In: Proceedings of the IEEE, pp. 1–19 (2020). https://doi.org/10.1109/JPROC.2020.3014430
27. Sekanina, L., Harding, S.L., Banzhaf, W., Kowaliw, T.: Image processing and CGP. In: Miller, J. (ed.) Cartesian genetic programming. NCS, pp. 181–215. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-17310-3_6
28. Suganuma, M., et al.: Evolution of deep convolutional neural networks using cartesian genetic programming. Evol. Comput. **28**(1), 141–163 (2020). https://doi.org/10.1162/evco_a_00253
29. Venkataramani, S., Kozhikkottu, V., Sabne, A., Roy, K., Raghunathan, A.: Logic synthesis of approximate circuits. IEEE Trans. Comput. Aided Des. Integr. Circ. Syst. **39**, 2503–2515 (2019)

APPENDIX B – EXPERIMENTS RESULTS

In this appendix, it is presented the results comparing the standard CGP against ABS CGP for all 100 exemplars of the International Workshop in Logic Synthesis (IWLS) Contest in 2020.

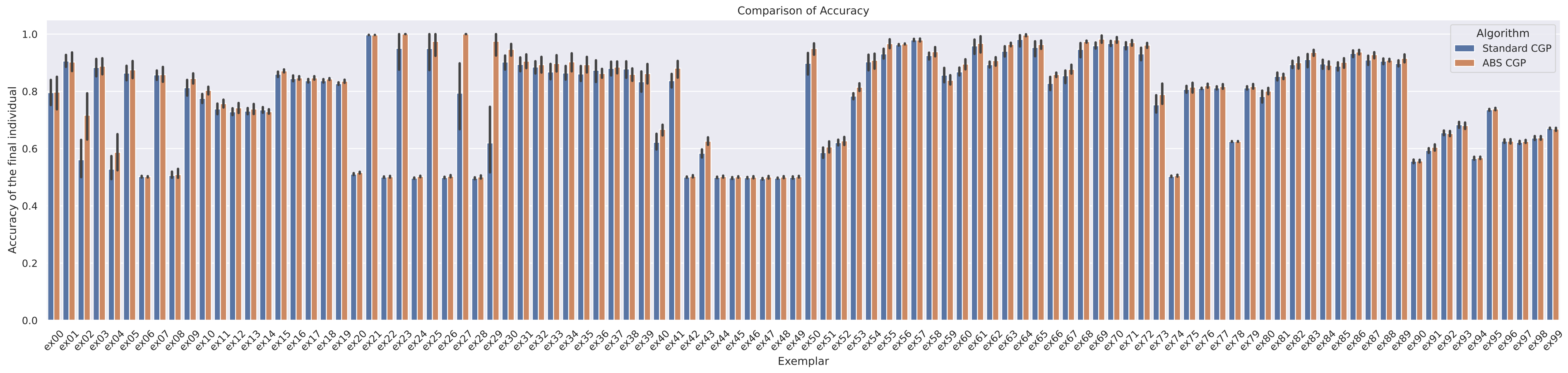


Figure 11 – Accuracy of Standard CGP and ABS CGP

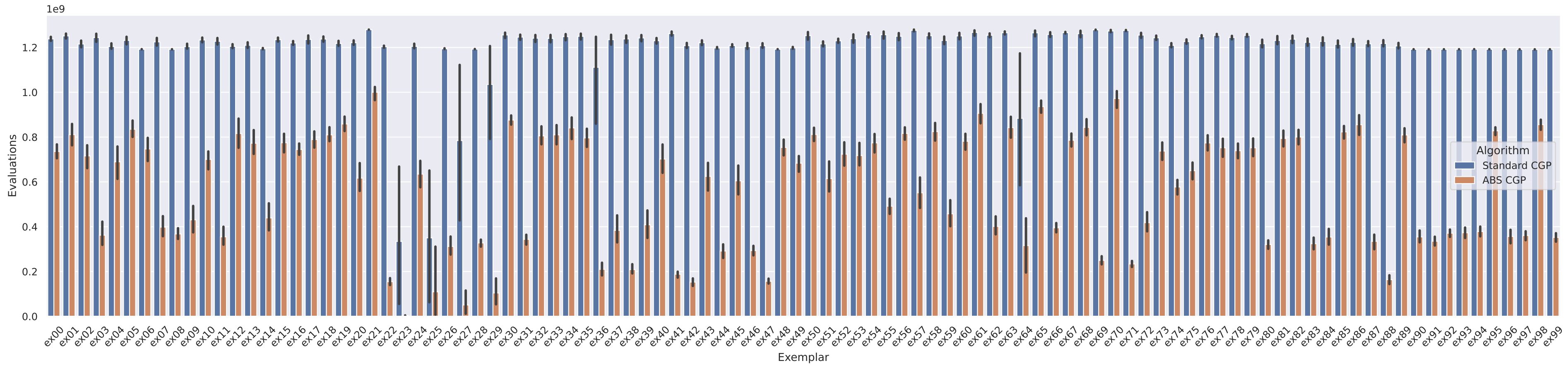


Figure 12 – Number of evaluations of Standard CGP and ABS CGP

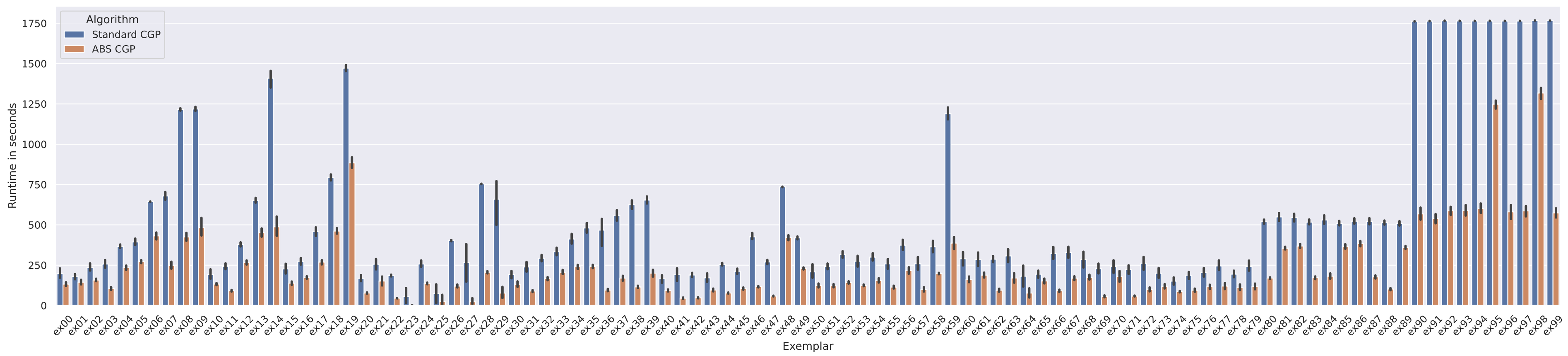


Figure 13 – Runtime of Standard CGP and ABS CGP