CGP++: a modern C++ implementation of cartesian genetic programming

Kalkreuth, R.T.; Bäck, T.H.W.

CGP++ : A Modern C++ Implementation of Cartesian Genetic Programming

Roman Kalkreuth
CNRS, LIP6, Sorbonne Université
Paris, France
roman.kalkreuth@lip6.fr

Thomas Bäck
LIACS, Leiden University
Leiden, The Netherlands
t.h.w.baeck@liacs.leidenuniv.nl

ABSTRACT

The reference implementation of Cartesian Genetic Programming (CGP) was written in the C programming language. C inherently follows a procedural programming paradigm, which entails challenges in providing a reusable and scalable implementation model for complex structures and methods. Moreover, due to the limiting factors of C, the reference implementation of CGP does not provide a generic framework and is therefore restricted to a set of predefined evaluation types. Besides the reference implementation, we also observe that other existing implementations are limited with respect to the features provided. In this work, we therefore propose the first version of a modern C++ implementation of CGP that pursues object-oriented design and generic programming paradigm to provide an efficient implementation model that can facilitate the discovery of new problem domains and the implementation of complex advanced methods that have been proposed for CGP over time. With the proposal of our new implementation, we aim to generally promote interpretability, accessibility and reproducibility in the field of CGP.

CCS CONCEPTS

• **Computing methodologies** → **Discrete space search**; **Genetic programming**; • **Software and its engineering** → **Object oriented languages**; **Concurrent programming languages**.

KEYWORDS

Cartesian Genetic Programming, Implementation, C++

ACM Reference Format:
Roman Kalkreuth and Thomas Bäck. 2024. CGP++ : A Modern C++ Implementation of Cartesian Genetic Programming. In *Genetic and Evolutionary Computation Conference (GECCO '24), July 14–18, 2024, Melbourne, VIC, Australia.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3638529.3654092

GECCO '24, July 14 - 18, 2024, Melbourne, Australia

ACM ISBN 979-8-4007-0494-9/24/07…$15.00
https://doi.org/10.1145/3638529.3654092

1 INTRODUCTION

Cartesian Genetic Programming (CGP) can be considered a well-established graph-based representation model of GP. The first pioneering work towards CGP was done by Miller, Thompson, Kalganova, and Fogarty [9, 28, 33] by the introduction of a graph encoding model based on a two-dimensional array of functional nodes. CGP can be considered an extension to the traditional tree-based representation model of GP since it enables many applications in problem domains that are well-suited for graph-based representations such as circuit design [4, 41], neural architecture search [45, 45] and image processing [2]. Miller officially introduced CGP over 20 years ago [32] and provided a reference implementation written in the C programming language. Since then, several implementations have been provided in other popular programming languages such as Java or Python, which follow modern programming paradigms. Besides implementations, several sophisticated methods for CGP have been proposed over time, and the significance of various developments has been recently surveyed and discussed in the context of the status and future of CGP [30]. Miller's reference implementation is based on the procedural programming paradigm, which naturally entails challenges and limitations to provide a flexible, reusable and generic architecture that can facilitate the implementation of complex methods and their corresponding structures. Moreover, Julian F. Miller passed away in 2022 [43][1] and his website[2] which served as a resource for his original implementation, disappeared shortly after his death for unknown reason to the authors of this paper. The above-described points and circumstances motivates our work, in which we present a modern implementation of CGP written in C++ called `CGP++`. Our implementation builds upon paradigms and methodologies commonly associated with the modern interpretation of the C++ programming language, such as generic programming. Since C++ has a reputation for providing excellent performance while representing a high-level object-oriented language that offers many features for generic programming, we feel that C++ is a suitable choice for a modern and contemporary implementation of CGP. This paper is structured as follows: In Section 2 we describe GP and CGP and address major problem domains in these fields. Section 3 surveys existing implementations of CGP that have been proposed for various programming languages. In Section 4 we introduce our new implementation by presenting key features and addressing relevant implementation details. Section 5 gives an overview of the architecture and workflow of `CGP++`. In Section 6 we compare our implementation to the implementations that have been addressed in this paper. Section 7 discusses the potential role of `CGP++` in the

[1] http://www.evostar.org/2022/julian-francis-miller/
[2] http://www.cartesiangp.co.uk/

ecosystem of CGP implementations and addresses prospects as well as challenges of enhancements that will be considered by future work. Finally, Section 8 concludes our work.

2 RELATED WORK

2.1 Genetic Programming

In the wider taxonomy of heuristics, Genetic Programming (GP) can be considered an evolutionary-inspired search heuristic that enables the synthesis of computer programs for problem-solving. The fundamental paradigm of GP aims at evolving a population of candidate *computer programs* towards an algorithmic solution of a predefined problem. GP transforms candidate genetic programs (Definition 2.1), that are traditionally represented as parse-trees, iteratively from generation to generation into new populations of programs with (hopefully) better fitness. However, since GP is a stochastic optimization process, obtaining the optimal solution is consequently not guaranteed. A formal definition of GP is provided in Definition 2.2. GP traditionally uses a parse-tree representation model that is inspired by LISP S-expressions. An example of a parse tree is illustrated in Figure 1. In addition to the conventional (tree-based) GP, GP is also used with linear sequence representations [35, 36], graph-based representation models [28, 37], or grammar-based representations [38].

Definition 2.1 (Genetic Program)**.** A genetic program $\mathfrak{P}$ is an element of $\mathfrak{T} \times \mathfrak{F} \times \mathfrak{E}$:

- $\mathfrak{F}$ is a finite non-empty set of functions
- $\mathfrak{T}$ is a finite non-empty set of terminals
- $\mathfrak{E}$ is a finite non-empty set of edges

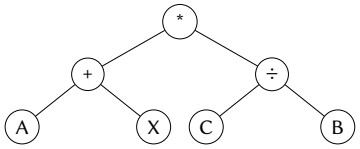
Let ϕ: $\mathcal{P} \mapsto \Psi$ be a decode function which maps $\mathcal{P}$ to a phenotype Ψ

Definition 2.2 (Genetic Programming)**.** Genetic Programming is an evolutionary algorithm-based method for the automatic derivation of computer programs. Let $\mathfrak{B}_\mu^{(g)}$ be a population of μ individuals and let $\mathfrak{B}_\mu^{(g+1)}$ be the population of the following generation:

- Each individual is represented with a genetic program and a fitness value.
- Genetic Programming transforms $\mathfrak{B}_\mu^{(g)} \mapsto \mathfrak{B}_\mu^{(g+1)}$ by the adaptation of selection, recombination and mutation.

2.2 Cartesian Genetic Programming

CGP can be considered an extension of conventional tree-based GP since it represents a genetic program as an acyclic and directed graph, and trees as data structures naturally entail combinational limitations. A genetic program is encoded in the genotype of an individual and is decoded to its corresponding phenotype before evaluation. Originally, the programs were represented with a rectangular $n_\mathrm{r} \times n_\mathrm{c}$ grid of nodes. However, later work focused on a representation with merely one row. A formal definition of a cartesian genetic program (CP) is given in Definition 2.3. In CGP, *function nodes* are used to execute functions, defined in the function set,



$$\mathfrak{F} = \{ +, -, *, \div \}$$
$$\mathfrak{T} = \{ A, X, C, B \}$$
$$\Psi = (A + X) * (C \div B)$$

Figure 1: Example of a parse tree as used in conventional GP. A parse tree can be considered a composition of elements taken from the function set $\mathcal{F}$ and terminal set $\mathcal{T}$. The decoding of the parse tree in the example leads to the symbolic expression Ψ.

on the input values. The decoding routine distinguishes between groups of genes, and each group represents a node of the graph, except the last one, which refers to the outputs. Two types of genes are used to encode a node: 1) the function gene, that indexes the function number in the function set and 2) the connection genes, which index the inputs of the node. The number of connection genes varies based on the on the predefined maximum arity n_a of the function set. The decoding of function nodes is embedded in a backward search that is performed for all output genes.

The backward search is illustrated in Figure 2 for the commonly used single-row integer representation, which starts from the program output and processes all linked nodes in the genotype until the inputs are reached. Consequently, only active nodes are processed during evaluation. The genotype illustrated in Figure 2 is grouped whereby the first (underlined) gene of each group refers to the function number in the function set. In contrast, the non-underlined genes which refer to the respective input connections of the node. Function nodes, that are not linked in the genotype, remain inactive and are visualized in gray color as well as dashed lines. A parameter called levels-back l is commonly used to control the connectivity of the graph by constraining the node index from which a function or output node can get its inputs.

Definition 2.3 (Cartesian Genetic Program)**.** A cartesian genetic program is an element of the Cartesian product $\mathfrak{N}_\mathrm{i} \times \mathfrak{N}_\mathrm{f} \times \mathfrak{N}_\mathrm{o} \times \mathfrak{F}$:

- $\mathfrak{N}_\mathrm{i}$ is a finite non-empty set of input nodes
- $\mathfrak{N}_\mathrm{f}$ is a finite set of function nodes
- $\mathfrak{N}_\mathrm{o}$ is a finite non-empty set of output nodes
- $\mathfrak{F}$ is a finite non-empty set of functions

The number of inputs n_i, outputs n_o, and the length of the genotype remain static during a run of CGP. Therefore, each candidate program can be represented with $n_\mathrm{r} * n_\mathrm{c} * (n_\mathrm{a} + 1) + n_\mathrm{o}$ integers. However, although the length of the genotype is static, the length of the corresponding phenotype can vary during a run, which enables a certain degree of flexibility of the CGP representation model. CGP is commonly used with a $(1 + \lambda)$ evolutionary algorithm (EA) and a selection strategy called *neutrality*, which is based on the idea that the adaption of a neutral genetic drift mechanism (NGD) can

Algorithm 1 Exemplification of the (1+λ)-EA variant with neutral genetic drift

```
1:  initialize(P)                                   ▷ Initialize parent individual
2:  repeat                                          ▷ Until termination criteria not triggered
3:      Q ← breed(P)                                ▷ Breed λ offspring by mutation
4:      Evaluate(Q)                                 ▷ Evaluate the fitness of the offspring
5:      Q⁺ ← best(Q, P)                             ▷ Get individuals which have better fitness as the parent
6:      Q⁼ ← equal(Q, P)                            ▷ Get individuals which have the same fitness as the parent
7:      ▷ If there exist individuals with better fitness
8:      if |Q⁺| > 0 then
9:          ▷ Choose one individual from Q⁺ uniformly at random
10:         P ← Q⁺[r], r ~ U[0, |Q⁺| − 1]
11:         ▷ Otherwise, if there exist individuals with equal fitness
12:     else if |Q⁼| > 0 then
13:         ▷ Choose one individual from Q⁼ uniformly at random
14:         P ← Q⁼[r], r ~ U[0, |Q⁼| − 1]
15:     end if
16: until P meets termination criterion
17: return P                                        ▷
```

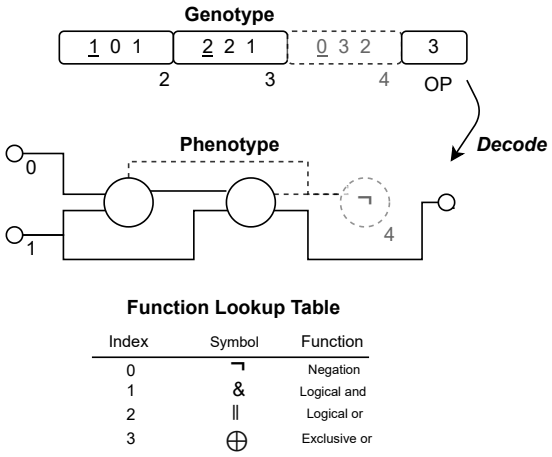


Figure 2: Illustration of the decoding procedure of the CGP genotype to the corresponding phenotype. The identifiers IP1 and IP2 refer to the two input nodes with node index 0 and 1. The identifier OP stands for the output node of the graph.

contribute significantly to the escape from local optima. NGD is implemented by extending the classical selection mechanism in such a way that individuals which have equal fitness as the normally selected parent are first determined, and one equal-fitness individual is then selected uniformly at random. NGD can be therefore considered as a random walk on the neutral neighborhood of equal-fitness offspring. A new population is formed in each generation with the selected parent from the previous population and the λ bred offspring. An exemplification of the $(1 + \lambda)$-EA variant used in CGP is provided in Algorithm 1. For the breeding procedure point mutation is predominantly used to exchange gene values of the genotype within the valid range by chance. The mutations triggered by this operator can alter the functionality of the phenotype as well as the connectivity depending on which type of gene is mutated. Genetic programs are mostly encoded with natural numbers in CGP that is commonly refereed to as integer-based or standard CGP. However, an alternative encoding model that is called real-valued CGP [1] has been proposed. It uses real numbers to encode candidate programs with the intention to adapt intermediate recombination which is commonly used for real-valued representations in Genetic Algorithms [3]. In contrast, integer-based CGP is predominantly used merely with mutation due to its long history of stagnation regarding the question about the effectiveness of recombination [30]. Recent studies, however, have found that various recombination operators such as subgraph [14], block [7] and discrete crossover [11] can be effectively used for various problems [13].

2.3 Problem Domains in GP

GP gained significant popularity when Koza [19, 23, 24] applied his parse tree representation model to practically relevant problem domains, for instance, symbolic regression, algorithm construction, logic synthesis, or classification. In this section, we describe two popular representatives of the GP application scope in more detail which have a reputation for being major real-world application scopes for GP as well as for their relevance for benchmarking GP methods:

2.3.1 Symbolic Regression. Symbolic Regression (SR) is located in the broader taxonomy of regression analysis, where a symbolic search is performed in a space of mathematical compositions to obtain candidate functions that match the ideal input-output mapping of a given data set as closely as possible. Symbolic regression in the context of GP can therefore be considered a black-box problem domain. In general, SR by means of GP relates to the application of GP models to synthesize mathematical expressions that represent input-output mapping of the the unknown function as closely as possible. Quite recently, it has been proved SR to be a NP-hard problem, since it is not always possible to find the best fitting mathematical expression for a given data set in polynomial time [50].

2.3.2 Logic Synthesis. Logic synthesis [5, 6] as tackled with GP comprises the synthesis of Boolean expressions that match input-output mappings of given Boolean functions. Boolean expressions are generally a way of formally expressing Boolean functions. LS as approached with GP predominantly addresses two major tasks located in the scope of this problem domain:

(1) Synthesis of a Boolean expression that matches the correct input-output mapping of a given Boolean function.
(2) Optimization of a Boolean expression (i.e. reduction of complexity).

Both tasks are carried out with respect to Boolean logic and algebra. Truth tables are a common way to represent Boolean functions and to describe their input-output mapping besides to representing them with algebraic expressions. Synthesis of Boolean expressions is typically approached by defining one or multiple respective optimization objectives. LS as an GP application area was greatly popularized by Koza when he started addressing LS by using his parse tree representation model [20–22]. Moreover, Koza utilized his approach to evolve expressions for Boolean functions such as digital multiplexers and parity since these functions can be represented as LISP S-expressions. However, digital circuits are often

characterized by Boolean functions with multiple outputs such as digital adders or multipliers. This resulted in the predominant use of CGP for LS since its graph encoding model is well-equipped to represent such functions [4, 41]. A real-world application of the LS domain is the automatic design of digital circuits.

2.4 Modern C++

C++ as a versatile and powerful programming language, has evolved significantly over the course of the last decade. Starting with the release of `C++11` [8] and the subsequent versions C++14, C++17, and C++20, various new features and corresponding best practices have been introduced, allowing developers to write more efficient and maintainable programs. Moreover, features that are associated with modern C++ have noticeably changed the way code is written in C++ remarkably improved the safety and expressiveness of C++ and are provided in the C++ Standard Library. Some of the language features that shaped modern C++ are:

- **Template type deduction**
  Templates are a feature that enables the use of generic types for functions and classes. Template type deduction therefore allows the creation of functions or classes that can be adapted to more than one type without re-implementing the code constructs for each type. In C++ this can be achieved using template parameters.

- **Smart Pointers**
  Smart Pointers provide a wrapper class around a raw pointer that have overloaded access operators such as `*` and `->`. Smart pointer managed objects have similar appearance as regular (raw) pointers. However, smart pointers can be deallocated automatically, in contrast to raw pointers. Smart pointers are therefore used to ensure that programs are free of memory leaks and, in this way, simplify the dynamic memory allocation in C++ while maintaining efficiency.

- **Lambda Expressions:**
  A concise method for defining inline functions or function objects is to use lambda expressions, especially when working with algorithms or when a function is used as parameter. Lambdas can make the code more readable by allowing more direct expressions of intentions, since they do not require explicit function declarations. Lambdas are, therefore, also called anonymous functions.

- **Constexpr**
  The primary intention behind constant expression is to enable performance improvement of programs by doing computations at compile time rather than runtime. C++11 introduced the keyword `constexpr`, which declares that it is possible to evaluate the value of a certain function or variable at compile time.

- **Concurrency**
  Concurrency support was initially introduced in `C++11` in order to boost program efficiency and allow multitasking. The Concurrency Support Library of C++ provides support for threads, atomic operations, mutual exclusion, and condition variables. Although concurrency enables multitasking, it does not necessarily mean that the desired tasks are executed simultaneously but are more approached by efficient switching between tasks.

3 EXISTING CGP IMPLEMENTATIONS

This section reviews existing implementations of CGP that are later considered for a comparison to `CGP++`. Some implementations have already been addressed in Miller's review on the state and future of CGP [30]. However, with the intention to complete the picture further and to allow a more comprehensive comparison, we consider additional implementations and address their key features and purpose briefly. Despite the fact that the resource for Julian Miller's C reference implementation went down a while ago, a modified version still exists and is publicly available[3]. The modified version has been adapted for hyperparameter tuning experiments and search performance evaluations across several methods suitable for combinatorial optimization and combinational synthesis [17, 42]. The implementation has therefore been additionally equipped with search algorithms such as simulated annealing. Another C implementation is the `CGP-Library`[4] published by Turner and Miller [48]. It supports standard CGP as well as the recurrent CGP [47] variant and provides the functionality for evolving artificial neural networks [18, 46]. The popular Java-based Evolutionary Computation Research System (ECJ) [25, 40] provides a CGP contrib package that supports integer-based CGP as well as real-valued CGP. Moreover, the ECJ CGP contrib package covers functionality, data and benchmarks for applications such as logic synthesis, classification and symbolic regression. Recently, a set of implementations of various advanced genetic operators has been added to the repository. The `CGP Toolbox`[5] is a framework that primarily focuses on LS addressed with CGP and has been proposed by Vasicek and Sekanina [49]. It is shipped in four different versions that, in each case, support LS or SR for either 32 or 64 bit architecture and enables efficient phenotype evaluation based on machine code vectorization. A CGP toolbox for Matlab that focuses on audio and image processing called `CGP4Matlab` has been proposed by Miragaia et al. [34] which was used to apply CGP to the problem of pitch estimation. `HAL-CGP`[6] is a pure Python implementation of CGP designed to target applications that are characterized by computationally expensive fitness evaluations [39]. `CartesianGP.jl`[7] is a library for using CGP in Julia. However, according to the authors, the code should be considered *pre-alpha* at the moment.

4 THE PROPOSED IMPLEMENTATION

4.1 General Motivation and Philosophy

Since Miller officially proposed CGP, increasing development has taken place over the course of the past two decades in the relatively young field of graph-based GP by proposing new representation

[3] http://github.com/paul-kaufmann/cgp/
[4] http://www.cgplibrary.co.uk/
[5] http://www.fit.vutbr.cz/~vasicek/cgp/
[6] http://github.com/Happy-Algorithms-League/hal-cgp
[7] http://github.com/um-tech-evolution/CartesianGP.jl

variants, promising forms of crossover, mutation and search algorithms, as well as benchmarks. With the proposal of CGP++, we think that our proposed implementation can address the following aspects to enhance the following points in the field of CGP:

- Maintaining accessibility for the use of CGP by extending the ecosystem of existing CGP implementations
- Improving the interpretability of sophisticated methods by providing a comprehensible architecture.
- Facilitating reproducibility of existing results by supporting benchmarking frameworks.

The fundamental philosophy behind CGP++ is to utilize aspects of modern C++ that have been described in Section 2 to implement features and properties that are provided by state-of-the-art (SOTA) heuristic frameworks. In the following subsections, we will address the key features and properties of CGP++ and share some details about the respective implementation details of that we used from modern C++.

4.2 Key Features and Properties

4.2.1 Object-oriented and Generic Design. CGP++ pursues an object-oriented and generic design to maintain an interpretable and reusable architecture for fundamental as well as sophisticated functionality, with the intention to assist further implementations of new techniques and the corresponding extension of the underlying architecture.

4.2.2 Advanced Genetic Variation. Since CGP has been predominantly used without recombination in the past, most implementations only support CGP in the standard mutation-only fashion. However, since recent work proposed new recombination operators and demonstrated the effectiveness for various problems [13], block [7] and discrete recombination [11] have been integrated into CGP++. Besides to the (1+λ)-EA variant used in CGP that has been exemplified in Algorithm 1, an implementation of a (μ+λ)-EA is provided to allow the recombination-based use of CGP. Furthermore, since recent work demonstrated that the consecutive execution of multiple mutation operators can benefit the search performance of CGP [10, 12], CGP++ therefore supports mutation pipelining and provides advanced mutations such as *inversion* and *duplication*.

4.2.3 Benchmarking. CGP++ provides an interface to the benchmarks that have been recently proposed for the General Boolean Function Benchmark Suite (GBFS) which provides a diverse set of LS problems for GP [15]. The provided PLU files contain compressed truth tables that can be used to set up the corresponding black-box problem. Moreover, CGP++ also provides a dataset generator and a set of objective functions for the SR benchmarks that have been proposed by McDermott et al. [27] in the framework of the first review on benchmarking standards in GP.

4.2.4 Hyperparameter Configuration. Hyperparameters related to the CGP functionality can be configured by using either a provided command-line interface or a parameter file, offering a flexible approach that can be used to apply CGP++ to contemporary frameworks for hyperparameter tuning such as `irace` [26].

4.2.5 Checkpointing. To ensure caching of intermediate search results that have been obtained over the course of the search process, CGP++ supports the creation of checkpoints that are automatically written during a run. The created checkpoint file can be used to resume a run in the case that it has been disrupted.

4.3 Implementation Details and Challenges

4.3.1 Generic Template. The generic template of CGP++ can be formally described as a tuple $\mathcal{T} = (\mathcal{E}, \mathcal{G}, \mathcal{F})$ where $\mathcal{E}$ defines the evaluation type, $\mathcal{G}$ the type of the CGP genome and $\mathcal{F}$ the type of fitness. CGP++'s generic approach is achieved by using C++ class templates. The respective data types can be configured via `typedef`. To restrict the data type of certain template classes such as the type of the genome, we use `constexpr` to evaluate the defined template type at compile time.

4.3.2 Smart Memory Management. CGP++ utilizes two types of smart pointers: `std::unique_ptr` and `std::shared_ptr` to provide safe memory allocation as well as efficient passing of objects, containers and data to functions and classes. In our implementation, `std::shared_ptr` is used for shared ownership of objects among instances of different classes. In contrast, `std::unique_ptr` is used in cases where single or exclusive ownership of a resource is desired.

4.3.3 Memorization. Memorization is used to speed up genotype-phenotype decoding by caching the immediate results of function nodes and consequently preventing reevaluating already computed results. The node-value mapping are stored by using `std::map` during the decoding routine.

4.3.4 Concurrency. Besides to consecutive evaluation of individuals, CGP++ takes the first step towards concurrency by providing concurrent evaluation of the population. For this purpose, the population is divided into chunks of individuals whose number is defined by the number of evaluation threads that can be set in the configuration. The chunks are then evaluated within several instances of `std::thread`. CGP++ supports deep cloning of problem instances to create the corresponding thread pool. The pool of threads are synchronized after evaluation via `std::thread::join`. However, the genotype-phenotype mapping of CGP and the corresponding requirement of a decoding procedure poses a bottleneck for the use of concurrency. At this time, we have to limit the concurrency feature in CGP++ for parts of the decoding and evaluation procedure of the genotype by using mutual exclusion via `std::mutex`. We will address potential solutions for this issue in the discussion.

4.4 Resources

The source code of CGP++ and a user guide are publicly available in our GitHub repository[8].

5 ARCHITECTURE AND WORKFLOW OVERVIEW

5.1 Fundamental Architecture

Abstraction and inheritance depict fundamental pillars of the top level architecture of-CGP++ to enable a high degree of reusability of its core functionality. Figure 3 provides a high-level class diagram

[8] http://github.com/RomanKalkreuth/cgp-plusplus

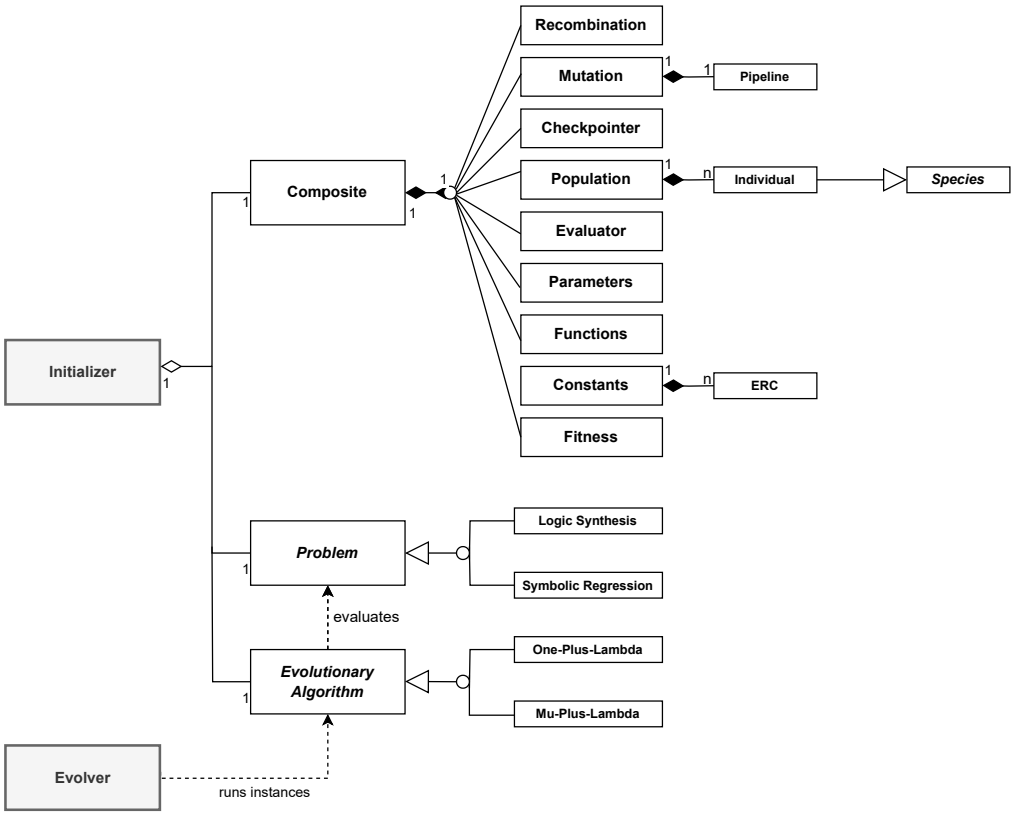


Figure 3: Illustration of the high-level architecture of CGP++ which uses abstraction and inheritance to facilitates reusability and flexibility for further extensions. Smart pointers are used to provide safe and efficient access to elements across the framework.

that covers its key architecture elements. The *Initializer* is designed to instantiate the core elements for the heuristic search performed by CGP, such as the selected search algorithm and defined problem. To bundle essential sub-elements for the heuristic search process, a *composite* is initialized, which can be accessed by other core elements such as the problem and algorithm instances. The composite includes crucial elements and features for the GP search process, such as the population, breeding execution frameworks for crossover and mutation, function and terminal (constants) sets but also backbone elements such as hyperparameter interfaces as well as checkpointing. CGP++ supports the generation of ephemeral random constants (ERC) to create the terminal set, which, together with the function set, is an integral part of the *Composite*. After initialization, the *Evolver* executes the considered number of instances (jobs) and reports final as well as immediate results via command line and output file. The high-level architecture of CGP++ is fundamentally inspired by ECJ [25, 40].

5.2 Top-level Workflow

The top-level workflow of CGP++ is shown in Figure 4 that illustrates the interplay between the elements that have been described related to the fundamental architecture. CGP++ can be used to run experiments that require several instances to ensure statistical validity. The *Evolver* therefore supports the execution of consecutive jobs, whose numbers can be configured via the parameter interface. The workflow within the framework of a job instance maps the typical workflow of the adapted (1+λ) and (μ+λ) strategies. To facilitate the integration of other types of evolutionary algorithm, we provide an abstract base class that can be used as a design pattern.

5.3 Concurrent Evaluation

The concurrent evaluation architecture is illustrated in Figure 5. When concurrent evaluation of the individuals is used, the evaluation procedure forks and joins a thread pool and each thread is equipped with a chunk of individuals as well as a deep copy of the problem instance. The respective functionalities such as deep cloning are provided in the *Population* and *Problem* classes. Since the *Decoder* is the shared resource in this framework, its access is maintained via mutual exclusion, as already mentioned in the previous section.

5.4 Checkpointing

The generation of a checkpoint is shown in Figure 6. For a checkpoint, we consider the random seed, generation number, genomes of the population, and the constants. These attributes are obtained

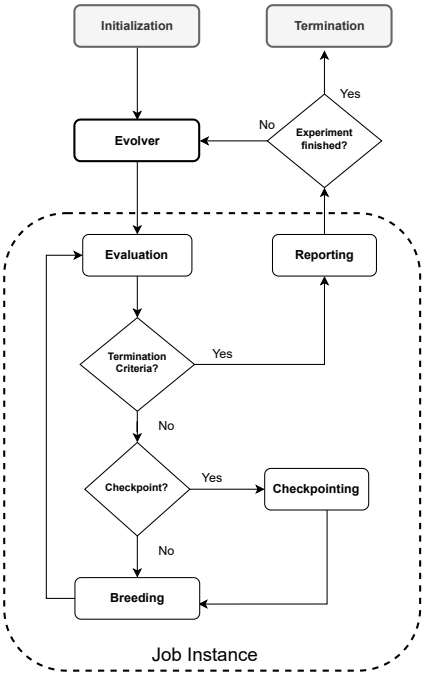


Figure 4: Overview of CGP++'s top-level workflow, addressing the execution of run instances as well as the main workflow that is executed within an instance.

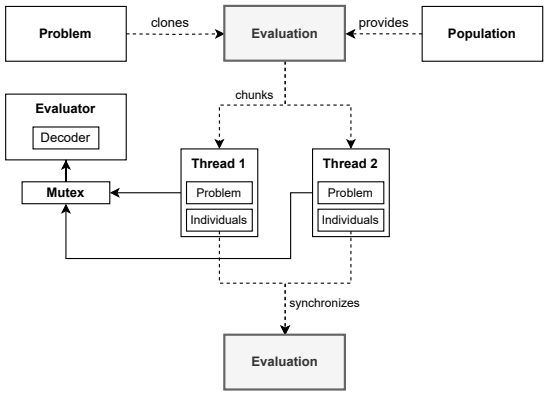


Figure 5: Illustration of the architecture for concurrent evaluation. In this example, the thread pool is simplified with two threads to give a structured overview. However, in general, CGP++ is capable of instantiating and executing more than two threads.

from the respective instances and are then considered as a checkpoint instance that is written to a checkpoint file. Instances can be resumed by using the same configuration as the aborted run and passing the checkpoint file to CGP++. When a checkpoint file is detected, it is loaded by the *Checkpointer* and the run instance is resumed at the given generation number.

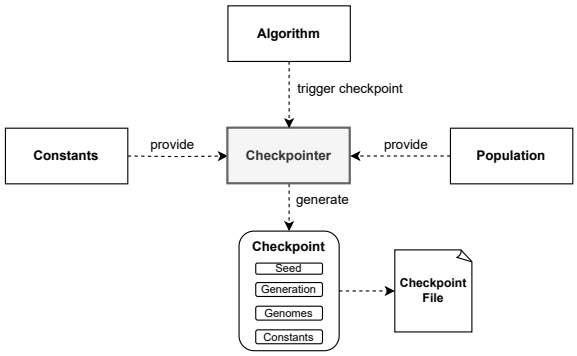


Figure 6: Illustration of the generation of a checkpoint that is generated with the chromosomes obtained from the population, the constants (ERC), the seed of the random generator, and the generation number. The checkpoint is written to an output file that can be used to resume a run.

6 IMPLEMENTATION COMPARISON

We considered various features for our comparison that can be found in modern metaheuristics toolkits. These include programming design, generic properties, checkpointing, variation pipelining, concurrency, and the existence of a parameter interface that can be used for hyperparameter tuning. We consider vectorization as a feature for our comparison, since it can be seen as a competitive feature to concurrency in CGP. With respect to recent findings about the role of crossover in CGP, we also consider this feature in our comparison. Table 1 shows the result of our comparison, and it is visible that the the supported features of our implementation are on the level of a modern metaheuristics toolkit such as ECJ. Please note that the evaluation of Julian Miller's reference implementation is based on its modified version. The ECJ CGP contrib package offers a wide range of features that are derived from the underlying ECJ framework. For the other implementations, we notice that the number of features is quite limited. However, the CGP Toolbox supports vectorization, which is currently not supported by CGP++. A finding that we will address in the following discussion. CGP4Matlab and HAL-CGP use programming languages that pursue dynamic typing concepts. However, we do not consider these concepts as generic, since generic programming aims at enabling data type independence while maintaining compile-time type safety. To our best knowledge, this is not covered by default by the dynamic typing concepts of these languages, and the generic extensions and features are currently not used for the respective implementations.

7 DISCUSSION AND FUTURE WORK

The primary intention of this work is to propose the first version of a modern implementation of CGP in the C++ programming language. Another intention behind our work is to propose and establish a flexible and reusable architecture that can facilitate and simplify the implementation of further extensions. We therefore deliberately chose C++ over Rust which would also have been a suitable option. However, we consider Rust more a procedural and

	Language	Paradigm	Generic Design	Checkpointing	Concurrency	Pipelining	Crossover	Vectorization	Parameter Interface
CGP++	C++	O	✓	✓	✓	✓	✓	-	✓
ECJ (CGP contrib)	Java	O	-	✓	✓	✓	✓	-	✓
Reference (Miller)	C	P	-	-	-	-	-	-	✓
CGP-Library	C++	P	-	-	✓	-	-	-	-
CGP Toolbox	C++	P	-	-	-	-	-	✓	✓
CGP4Matlab	Matlab	O	-	-	-	-	-	-	-
HAL-CGP	Python	O	-	-	-	-	-	-	-
CartesianGP.jl	Python	P	✓	-	-	-	-	-	-

O: object-oriented **P**: procedural

Table 1: Results of our implementation comparison that considered different general features of modern heuristic toolkits but also CGP related aspects.

functional-oriented programming language rather than a strong object-oriented one. Moreover, since we also focus on interpretability of CGP methodologies implemented in CGP++, we find C++ code more approachable than Rust code for this purpose.

We want to stress that with CGP++ we do not intend to propose a framework that we generally consider superior over other implementations. With our contribution, we more intend to extend the already-existing ecosystem of CGP implementations. We acknowledge that every programming language has its own specific characteristics, and each implementation has its own purpose and philosophy. However, we feel that most CGP implementations fall short of providing features that can facilitate the discovery of novel applications and the integration of new techniques. Moreover, since the majority of the implementations that we considered for our comparison follow the procedural programming paradigm, we think that there is a need for frameworks that can facilitate the implementation and maintenance of larger and more complex methods that have been proposed for CGP. Another point that should be discussed is related to how CGP is effectively used and how this is related to future work on CGP++. CGP has a reputation for being effectively used with relatively small population sizes due to early experiments with Boolean functions [29, 31]. However, recent studies on the parametrization of CGP demonstrated that CGP can be also effectively used with large population sizes in the SR domain [16]. In contrast, these studies also demonstrated that CGP performs best in the LS domain when a $(1+\lambda)$-ES with a very small population size is used. Moreover, very recent work found the $(1+1)$-ES to be the best choice for the evaluation of the General Boolean Function Benchmark Suite (GBFS) [15] for LS. Based on these findings, we think that at least two modalities in CGP have to be considered for future work. Therefore, we would like to address the following point as natural next steps for CGP++:

7.1 Concurrency

Since we already raised the issue of using concurrency for CGP, we would like to discuss how the corresponding challenges could be tackled in the future. In the first place, this would imply extending the thread pool design by using multiple evaluator instances. However, we have to stress here that related elements such as the CGP decoder have to be multiplied, and this would lead away from the idea of using a lightweight thread pool inside CGP++. Another idea would be to consider an alternative concurrency design pattern that could enable highly concurrent use and implements aspects from parallel dynamic programming [44]. Currently, CGP++ only supports concurrency for the evaluation process. Therefore, another contribution would be to enable breeding concurrency. In general, despite the highlighted challenges, we think it is worthy to explore whether concurrency could be effectively used for problems where CGP seems to work well when large population sizes are being considered.

7.2 Vectorization

The use of vector operations by using related extended SIMD instructions is another feature that we would like to consider for future work. Vectorization with machine code that has been generated from CGP primitives and which contains SIMD instructions has been successfully used to speed up the CGP evaluation procedure [49]. The used SSE/SSE2 SIMD instruction calls operated with 128-bit vectors in that case. However, providing such a feature from today's perspective could also enable support for contemporary instruction sets such as Advanced Vector Extensions (i.e. AVX-2 or AVX-512). In view of the fact that the $(1+1)$-ES has been found to be the best choice on GBFS, vectorization could be used to provide a way to use CGP effectively in a consecutive fashion.

7.3 Towards a Modern General GP Toolkit

Even if this paper proposes the first version of CGP++, we do not only see it as a implementation of CGP but also as a blueprint that can shape the way towards a modern and general GP toolkit that allows the use of multiple GP variants in a flexible and effective way. Therefore, we consider extending CGP++ to GP++ in the future that can benefit the GP domain across different representations with the contributing factors that we intend to achieve with the proposal of our implementation. As a first step towards that goal, we plan to integrate tree-based and linear GP as popular representatives of the GP domain.

8 CONCLUSION

In this paper we presented the first version of a modern C++ implementation of Cartesian Genetic Programming, which closes a major gap in the framework of existing implementations. Our implementation provides key features and characteristics of modern heuristics frameworks. Our proposed implementation offers a genetic design and provides a reusable architecture that can facilitate the discovery of new problem domains and the integration of new methods for CGP. Equipped with interfaces and generators for benchmarking in Logic Synthesis and Symbolic Regression, CGP++, also provides a framework that can be used for the reproducibility of existing results.

Acknowledgments. CGP++ is dedicated to Dr. Julian Francis Miller (1955 - 2022), who as the founder of Cartesian Genetic Programming devoted a large part of his scientific life to its proposal, development and analysis. With the introduction of CGP++ we pay tribute to Julian's pioneering effort in the field of graph-based GP and acknowledge his lifework. The project was financially supported by ANR project HQI ANR-22-PNCQ-0002.

REFERENCES

[1] Janet Clegg, James Alfred Walker, and Julian Francis Miller. 2007. A new crossover technique for Cartesian genetic programming. In *Genetic and Evolutionary Computation Conference, GECCO 2007, Proceedings, London, England, UK, July 7-11, 2007*, Hod Lipson (Ed.). ACM, 1580–1587. https://doi.org/10.1145/1276958.1277276

[2] Kévin Cortacero, Brienne Mckenzie, Sabina Mueller, Roxana Khazen, Fanny Lafouresse, Gaëlle Corsaut, Nathalie Acker, François-Xavier Frenois, Laurence Lamant, Nicolas Meyer, Béatrice Vergier, Dennis Wilson, Hervé Luga, Oskar Staufer, Michael Dustin, Salvatore Valitutti, and Sylvain Cussat-Blanc. 2023. Evolutionary design of explainable algorithms for biomedical image segmentation. *Nature Communications* 14 (11 2023). https://doi.org/10.1038/s41467-023-42664-x

[3] A. E. Eiben and J. E. Smith. 2015. *Representation, Mutation, and Recombination.* Springer Berlin Heidelberg, Berlin, Heidelberg, 49–78. https://doi.org/10.1007/978-3-662-44874-8_4

[4] Petr Fiser, Jan Schmidt, Zdenek Vasícek, and Lukás Sekanina. 2010. On logic synthesis of conventionally hard to synthesize circuits using genetic programming. In *13th IEEE International Symposium on Design and Diagnostics of Electronic Circuits and Systems, DDECS 2010, Vienna, Austria, April 14-16, 2010*, Elena Gramatová, Zdenek Kotásek, Andreas Steininger, Heinrich Theodor Vierhaus, and Horst Zimmermann (Eds.). IEEE Computer Society, 346–351. https://doi.org/10.1109/DDECS.2010.5491755

[5] Gary D. Hachtel and Fabio Somenzi. 1996. *Logic synthesis and verification algorithms.* Kluwer.

[6] Soha Hassoun and Tsutomu Sasao. 2001. *Logic synthesis and verification.* Vol. 654. Springer Science & Business Media.

[7] Jakub Husa and Roman Kalkreuth. 2018. A Comparative Study on Crossover in Cartesian Genetic Programming. In *Genetic Programming - 21st European Conference, EuroGP 2018, Parma, Italy, April 4-6, 2018, Proceedings (Lecture Notes in Computer Science, Vol. 10781)*, Mauro Castelli, Lukás Sekanina, Mengjie Zhang, Stefano Cagnoni, and Pablo García-Sánchez (Eds.). Springer, 203–219. https://doi.org/10.1007/978-3-319-77553-1_13

[8] ISO. 2012. *ISO/IEC 14882:2011 Information technology — Programming languages — C++.* International Organization for Standardization, Geneva, Switzerland. 1338 (est.) pages. http://www.iso.org/iso/iso_catalogue/catalogue_tc/catalogue_detail.htm?csnumber=50372

[9] T. Kalganova. 1997. Evolutionary Approach to Design Multiple-valued Combinational Circuits. In *Proceedings. of the 4th International conference on Applications of Computer Systems (ACS'97)*. Szczecin, Poland, 333–339.

[10] Roman Kalkreuth. 2022. Phenotypic duplication and inversion in cartesian genetic programming applied to boolean function learning. In *Proceedings of the Genetic and Evolutionary Computation Conference Companion* (Boston, Massachusetts) *(GECCO '22)*. Association for Computing Machinery, New York, NY, USA, 566–569. https://doi.org/10.1145/3520304.3529065

[11] Roman Kalkreuth. 2022. Towards Discrete Phenotypic Recombination in Cartesian Genetic Programming. In *Parallel Problem Solving from Nature - PPSN XVII - 17th International Conference, PPSN 2022, Dortmund, Germany, September 10-14, 2022, Proceedings, Part II (Lecture Notes in Computer Science, Vol. 13399)*, Günter Rudolph, Anna V. Kononova, Hernán E. Aguirre, Pascal Kerschke, Gabriela Ochoa, and Tea Tusar (Eds.). Springer, 63–77. https://doi.org/10.1007/978-3-031-14721-0_5

[12] Roman Kalkreuth. 2022. Towards Phenotypic Duplication and Inversion in Cartesian Genetic Programming. In *Proceedings of the 14th International Joint Conference on Computational Intelligence, IJCCI 2022, Valletta, Malta, October 24-26, 2022*, Thomas Bäck, Bas van Stein, Christian Wagner, Jonathan M. Garibaldi, H. K. Lam, Marie Cottrell, Faiyaz Doctor, Joaquim Filipe, Kevin Warwick, and Janusz Kacprzyk (Eds.). SCITEPRESS, 50–61. https://doi.org/10.5220/0011551000003332

[13] Roman Kalkreuth. 2023. Crossover in Cartesian Genetic Programming: Evaluation of Two Phenotypic Methods. In *Computational Intelligence*, Jonathan Garibaldi, Christian Wagner, Thomas Bäck, Hak-Keung Lam, Marie Cottrell, Kurosh Madani, and Kevin Warwick (Eds.). Springer International Publishing, Cham, 44–72.

[14] Roman Kalkreuth, Günter Rudolph, and Andre Droschinsky. 2017. A New Subgraph Crossover for Cartesian Genetic Programming. In *Genetic Programming - 20th European Conference, EuroGP 2017, Amsterdam, The Netherlands, April 19-21, 2017, Proceedings (Lecture Notes in Computer Science, Vol. 10196)*, James McDermott, Mauro Castelli, Lukás Sekanina, Evert Haasdijk, and Pablo García-Sánchez (Eds.). 294–310. https://doi.org/10.1007/978-3-319-55696-3_19

[15] Roman Kalkreuth, Zdeněk Vašíček, Jakub Husa, Diederick Vermetten, Furong Ye, and Thomas Bäck. 2023. General Boolean Function Benchmark Suite. In *Proceedings of the 17th ACM/SIGEVO Conference on Foundations of Genetic Algorithms* (Potsdam, Germany) *(FOGA '23)*. Association for Computing Machinery, New York, NY, USA, 84–95. https://doi.org/10.1145/3594805.3607131

[16] Paul Kaufmann and Roman Kalkreuth. 2017. Parametrizing Cartesian Genetic Programming: An Empirical Study. In *KI 2017: Advances in Artificial Intelligence - 40th Annual German Conference on AI, Dortmund, Germany, September 25-29, 2017, Proceedings (Lecture Notes in Computer Science, Vol. 10505)*, Gabriele Kern-Isberner, Johannes Fürnkranz, and Matthias Thimm (Eds.). Springer, 316–322. https://doi.org/10.1007/978-3-319-67190-1_26

[17] Paul Kaufmann and Roman Kalkreuth. 2020. On the Parameterization of Cartesian Genetic Programming. In *IEEE Congress on Evolutionary Computation, CEC 2020, Glasgow, United Kingdom, July 19-24, 2020*. IEEE, 1–8. https://doi.org/10.1109/CEC48606.2020.9185492

[18] Maryam Mahsal Khan, Arbab Masood Ahmad, Gul Muhammad Khan, and Julian F. Miller. 2013. Fast learning neural networks using Cartesian genetic programming. *Neurocomputing* 121 (2013), 274–289. https://doi.org/10.1016/J.NEUCOM.2013.04.005

[19] J. Koza. 1990. *Genetic Programming: A paradigm for genetically breeding populations of computer programs to solve problems.* Technical Report STAN-CS-90-1314. Dept. of Computer Science, Stanford University.

[20] John R. Koza. 1989. Hierarchical Genetic Algorithms Operating on Populations of Computer Programs. In *Proceedings of the 11th International Joint Conference on Artificial Intelligence. Detroit, MI, USA, August 1989*, N. S. Sridharan (Ed.). Morgan Kaufmann, 768–774. http://ijcai.org/Proceedings/89-1/Papers/123.pdf

[21] John R. Koza. 1990. Concept Formation and Decision Tree Induction Using the Genetic Programming Paradigm. In *Parallel Problem Solving from Nature, 1st Workshop, PPSN I, Dortmund, Germany, October 1-3, 1990, Proceedings (Lecture Notes in Computer Science, Vol. 496)*, Hans-Paul Schwefel and Reinhard Männer (Eds.). Springer, 124–128. https://doi.org/10.1007/BFb0029742

[22] John R. Koza. 1990. A Hierarchical Approach to Learning the Boolean Multiplexer Function. In *Proceedings of the First Workshop on Foundations of Genetic Algorithms. Bloomington Campus, Indiana, USA, July 15-18 1990*, Gregory J. E. Rawlins (Ed.). Morgan Kaufmann, 171–192. https://doi.org/10.1016/b978-0-08-050684-5.50014-8

[23] John R. Koza. 1992. *Genetic Programming: On the Programming of Computers by Means of Natural Selection.* MIT Press, Cambridge, MA, USA. http://mitpress.mit.edu/books/genetic-programming

[24] John R. Koza. 1994. *Genetic Programming II: Automatic Discovery of Reusable Programs.* MIT Press, Cambridge Massachusetts. http://www.genetic-programming.org/gpbook2toc.html

[25] Sean Luke. 2017. ECJ then and now. In *Genetic and Evolutionary Computation Conference, Berlin, Germany, July 15-19, 2017, Companion Material Proceedings*, Peter A. N. Bosman (Ed.). ACM, 1223–1230. https://doi.org/10.1145/3067695.3082467

[26] Manuel López-Ibáñez, Jérémie Dubois-Lacoste, Leslie Pérez Cáceres, Mauro Birattari, and Thomas Stützle. 2016. The irace package: Iterated racing for automatic algorithm configuration. *Operations Research Perspectives* 3 (2016), 43–58. https://doi.org/10.1016/j.orp.2016.09.002

[27] James McDermott, David Robert White, Sean Luke, Luca Manzoni, Mauro Castelli, Leonardo Vanneschi, Wojciech Jaskowski, Krzysztof Krawiec, Robin Harper, Kenneth A. De Jong, and Una-May O'Reilly. 2012. Genetic programming needs better benchmarks. In *Genetic and Evolutionary Computation Conference, GECCO '12, Philadelphia, PA, USA, July 7-11, 2012*, Terence Soule and Jason H. Moore (Eds.). ACM, 791–798. https://doi.org/10.1145/2330163.2330273

[28] Julian F. Miller. 1999. An empirical study of the efficiency of learning boolean functions using a Cartesian Genetic Programming approach. In *Proceedings of the Genetic and Evolutionary Computation Conference*, Wolfgang Banzhaf, Jason Daida, Agoston E. Eiben, Max H. Garzon, Vasant Honavar, Mark Jakiela, and Robert E. Smith (Eds.), Vol. 2. Morgan Kaufmann, Orlando, Florida, USA, 1135–1142. http://citeseer.ist.psu.edu/153431.html

[29] Julian F. Miller. 1999. An empirical study of the efficiency of learning boolean functions using a Cartesian Genetic Programming approach. In *Proceedings of the 1st Annual Conference on Genetic and Evolutionary Computation - Volume 2* (Orlando, Florida) *(GECCO'99)*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1135–1142.

[30] Julian Francis Miller. 2020. Cartesian genetic programming: its status and future. *Genet. Program. Evolvable Mach.* 21, 1-2 (2020), 129–168. https://doi.org/10.1007/S10710-019-09360-6

[31] Julian F. Miller and Stephen L. Smith. 2006. Redundancy and Computational Efficiency in Cartesian Genetic Programming. *IEEE Trans. Evol. Comput.* 10, 2 (2006), 167–174. https://doi.org/10.1109/TEVC.2006.871253

[32] Julian F. Miller and Peter Thomson. 2000. Cartesian Genetic Programming. In *Genetic Programming, European Conference, Edinburgh, Scotland, UK, April 15-16, 2000, Proceedings (Lecture Notes in Computer Science, Vol. 1802)*, Riccardo Poli, Wolfgang Banzhaf, William B. Langdon, Julian F. Miller, Peter Nordin, and Terence C. Fogarty (Eds.). Springer, 121–132. https://doi.org/10.1007/978-3-540-46239-2_9

[33] J. F. Miller, P. Thomson, and T. Fogarty. 1997. Designing Electronic Circuits Using Evolutionary Algorithms. Arithmetic Circuits: A Case Study. In *Genetic Algorithms and Evolution Strategies in Engineering and Computer Science*. Wiley, 105–131.

[34] Rolando Miragaia, Gustavo Reis, Francisco Fernandéz, Tiago Inácio, and Carlos Grilo. 2018. CGP4Matlab - A Cartesian Genetic Programming MATLAB Toolbox for Audio and Image Processing. In *Applications of Evolutionary Computation*, Kevin Sim and Paul Kaufmann (Eds.). Springer International Publishing, Cham, 455–471.

[35] S. Openshaw and I. Turton. 1994. Building new spatial interaction models using genetic programming. In *Evolutionary Computing, Lecture Notes in Computer Science*. Springer-Verlag, 11–13.

[36] Tim Perkis. 1994. Stack-Based Genetic Programming. In *Proceedings of the 1994 IEEE World Congress on Computational Intelligence*, Vol. 1. IEEE Press, Orlando, Florida, USA, 148–153. https://doi.org/doi:10.1109/ICEC.1994.350025

[37] Riccardo Poli. 1996. *Parallel Distributed Genetic Programming*. Technical Report CSRP-96-15. School of Computer Science, University of Birmingham, B15 2TT, UK. ftp://ftp.cs.bham.ac.uk/pub/tech-reports/1996/CSRP-96-15.ps.gz

[38] Conor Ryan, J. J. Collins, and Michael O'Neill. 1998. Grammatical Evolution: Evolving Programs for an Arbitrary Language. In *Proceedings of the First European Workshop on Genetic Programming (LNCS, Vol. 1391)*, Wolfgang Banzhaf, Riccardo Poli, Marc Schoenauer, and Terence C. Fogarty (Eds.). Springer-Verlag, Paris, 83–96. https://doi.org/doi:10.1007/BFb0055930

[39] Maximilian Schmidt and Jakob Jordan. 2020. *hal-cgp: Cartesian genetic programming in pure Python*. https://doi.org/10.5281/zenodo.3889163

[40] Eric O. Scott and Sean Luke. 2019. ECJ at 20: toward a general metaheuristics toolkit. In *Proceedings of the Genetic and Evolutionary Computation Conference Companion, GECCO 2019, Prague, Czech Republic, July 13-17, 2019*, Manuel López-Ibáñez, Anne Auger, and Thomas Stützle (Eds.). ACM, 1391–1398. https://doi.org/10.1145/3319619.3326865

[41] Lukas Sekanina, James Alfred Walker, Paul Kaufmann, and Marco Platzner. 2011. *Evolution of Electronic Circuits*. Springer Berlin Heidelberg, Berlin, Heidelberg, 125–179. https://doi.org/10.1007/978-3-642-17310-3_5

[42] Léo Françoso Dal Piccol Sotto, Paul Kaufmann, Timothy Atkinson, Roman Kalkreuth, and Márcio Porto Basgalupp. 2020. A study on graph representations for genetic programming. In *GECCO '20: Genetic and Evolutionary Computation Conference, Cancún Mexico, July 8-12, 2020*, Carlos Artemio Coello Coello (Ed.). ACM, 931–939. https://doi.org/10.1145/3377930.3390234

[43] Susan Stepney and Alan Dorin. 2022. Julian Francis Miller, 1955–2022. *Artificial Life* 28, 1 (06 2022), 154–156. https://doi.org/10.1162/artl_a_00371 arXiv:https://direct.mit.edu/artl/article-pdf/28/1/154/2029198/artl_a_00371.pdf

[44] Alex Stivala, Peter J. Stuckey, Maria Garcia de la Banda, Manuel Hermenegildo, and Anthony Wirth. 2010. Lock-free parallel dynamic programming. *J. Parallel and Distrib. Comput.* 70, 8 (2010), 839–848. https://doi.org/10.1016/j.jpdc.2010.01.004

[45] Masanori Suganuma, Masayuki Kobayashi, Shinichi Shirakawa, and Tomoharu Nagao. 2020. Evolution of Deep Convolutional Neural Networks Using Cartesian Genetic Programming. *Evolutionary Computation* 28, 1 (03 2020), 141–163. https://doi.org/10.1162/evco_a_00253 arXiv:https://direct.mit.edu/evco/article-pdf/28/1/141/2020362/evco_a_00253.pdf

[46] Andrew James Turner and Julian Francis Miller. 2013. Cartesian genetic programming encoded artificial neural networks: a comparison using three benchmarks. In *Genetic and Evolutionary Computation Conference, GECCO '13, Amsterdam, The Netherlands, July 6-10, 2013*, Christian Blum and Enrique Alba (Eds.). ACM, 1005–1012. https://doi.org/10.1145/2463372.2463484

[47] Andrew James Turner and Julian Francis Miller. 2014. Recurrent Cartesian Genetic Programming. In *Parallel Problem Solving from Nature - PPSN XIII - 13th International Conference, Ljubljana, Slovenia, September 13-17, 2014. Proceedings (Lecture Notes in Computer Science, Vol. 8672)*, Thomas Bartz-Beielstein, Jürgen Branke, Bogdan Filipic, and Jim Smith (Eds.). Springer, 476–486. https://doi.org/10.1007/978-3-319-10762-2_47

[48] Andrew James Turner and Julian Francis Miller. 2015. Introducing a cross platform open source Cartesian Genetic Programming library. *Genet. Program. Evolvable Mach.* 16, 1 (2015), 83–91. https://doi.org/10.1007/S10710-014-9233-1

[49] Zdenek Vasicek and Karel Slany. 2012. Efficient Phenotype Evaluation in Cartesian Genetic Programming. In *Proceedings of the 15th European Conference on Genetic Programming, EuroGP 2012 (LNCS, Vol. 7244)*, Alberto Moraglio, Sara Silva, Krzysztof Krawiec, Penousal Machado, and Carlos Cotta (Eds.). Springer Verlag, Malaga, Spain, 266–278. https://doi.org/doi:10.1007/978-3-642-29139-5_23

[50] Marco Virgolin and Solon P. Pissis. 2022. Symbolic Regression is NP-hard. *Trans. Mach. Learn. Res.* 2022 (2022). https://openreview.net/forum?id=LTiaPxqe2e