Graph-based Genetic Programming[1]

Roman Kalkreuth, Léo Françoso Dal Piccol Sotto, Zdeněk Vašíček

TU Dortmund University, Dortmund, Germany

Fraunhofer Institute for Algorithms and Scientific Computing, St. Augustin, Germany

Brno University of Technology, Brno, Czech Republic

roman.kalkreuth@tu-dortmund.de
leo.francoso.dal.piccol.sotto@scai-extern.fraunhofer.de
vasicek@fit.vut.cz

GECCO '22 Companion, Boston, USA

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM. . . $15.00.
doi:10.1145/nnnnnnn.nnnnnnn

[1] The latest version of these slides is available at https://github.com/RomanKalkreuth/graph-based-gp-tutorial

About us

Roman Kalkreuth received a Bachelor and Master Degree in Computer Science from the South Westphalia University of Applied Sciences, Iserlohn, Germany (2010 and 2012), and graduated with a Doctor of Science at TU Dortmund University, Dortmund, Germany (2021). He works as a postdoc within the Computational Intelligence Research Group of Prof. Dr. Günter Rudolph at TU Dortmund University.

Léo Françoso Dal Piccol Sotto works as a research scientist at the Fraunhofer Institute for Algorithms and Scientific Computing, Germany. He received his bachelor (2015) and his Ph.D. (2020) in Computer Science from the Federal University of São Paulo, Brazil.

Zdeněk Vašíček received all his degrees from Brno University of Technology, Czech Republic, where he is currently an Associate professor. He holds a Ph.D. (2012) and an M.S. equivalent (2006) in Computer Science and Engineering. His research interests include formal verification techniques and application of evolutionary approaches in areas related to the design and optimization of complex digital circuits and systems.

Dedication

Dr. Julian Francis Miller (1955 – 2022)
Founder of Cartesian Genetic Programming
Genetic Programming Pioneer

Agenda

- Origin and context of graph-based GP
- Representation models
  - Cartesian Genetic Programming (CGP)
  - Linear Genetic Programming (LGP)
  - Evolving Graphs by Graph Programming (EGGP)
- Practical Applications
  - Digital Circuit Design
  - Artificial Neuroevolution
- Demonstration
  - Java-based Evolutionary Computation Toolkit (ECJ)

Objectives of this Tutorial

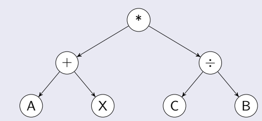
- Illustration of historical background and taxonomy
  - Implications for the development of the field
- Fundamental overview of three representation models
  - Selection and search mechanisms
  - Variation operators
  - Highlighting differences between the models
  - Comparison to tree-based GP
- Getting to know major successful applications
  - Design of digital circuits and neural networks
- Resources for graph-based GP methods implementations.

Introduction and Related Work

General Methodology

Genetic Programming (GP)

- Genetic Programming is a search heuristic.
- Inspired by neo-Darwinian evolution.
- Method for the synthesis of computer programs.
- Traditionally used with parse trees.



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

Introduction and Related Work

General Methodology

Definition (Genetic Programming)

Let Θ be a population of $|\Theta|$ individuals and let Ω be the population of the following generation:

- Each individual is represented with a **genetic program** and a **fitness value**.
- Genetic Programming transforms $\Theta \mapsto \Omega$ by the adaptation of **selection**, **recombination** and **mutation**.

Introduction and Related Work

General Methodology

Definition (Genetic Program)

A genetic programm $\mathcal{P}$ is an element of $\mathcal{T} \times \mathcal{F} \times \mathcal{E}$:

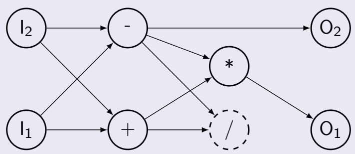
- $\mathcal{F}$ is a finite non-empty set of functions
- $\mathcal{T}$ is a finite non-empty set of terminals
- $\mathcal{E}$ is a finite non-empty set of edges

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

Introduction and Related Work
General Methodology

What we mean by graph-based

- Genetic Programming → traditionally tree representation, which is a **well defined form of graph**.
- Graph-based Genetic Programming → use of graph representation models that **extend GP beyond trees**.
- In the methods we consider, **Directed Acyclic Graphs** (DAGs), introducing:
  - Reuse of intermediate results.
  - Active and inactive nodes.



Introduction and Related Work
Historical Background

Major motivation for graph-based GP development

- Application of GP to **evolvable hardware**.
  - Digital circuit design
- Evolution of programs with high degree of **parallelism and distributedness**.
- Discovery of **symbolic**, **neuro-symbolic** and **neural networks**.
- Direct evolution of **machine code**.
- Advantages from the DAG representation features.

Introduction and Related Work
Historical Background

Graph-based GP Timeline

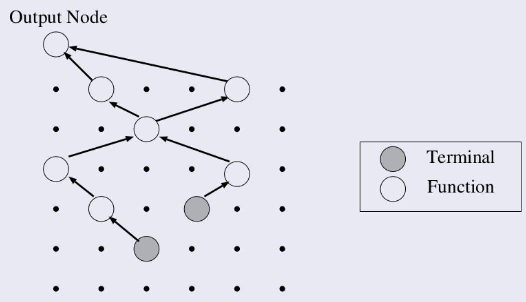
1990	Koza: Genetic Programming on LISP Syntax Trees
1993	Banzhaf: Precursor to Linear Genetic Programming
1994	Nordin: Compiling Genetic Programming System
1996	Poli: Parallel Distributed Genetic Programming
1996	Teller: Parallel Algorithm Discovery and Orchestration (PADO)
1997	Miller, Thompson, Kalganova, Fogarty: Steps forward Cartesian Genetic Programming
1998	Nordin: Automatic Induction of Machine Code with Genetic Programming
1999	Angeline: Multiple Interacting Programs (MIPs)
1999	Miller. Thompson: **Cartesian Genetic Programming**
2001	Brameier, Banzhaf: **Linear Genetic Programming**
2002	Kantschik, Banzhaf: Linear-Graph Genetic Programming
2008	Galvan-Lopez: Multiple Interactive Outputs in a Single Tree (MIOST)
2011	Downey, Zhang: Parallel Linear Genetic Programming
2017	Kelly and Keywood: Tangled Program Graphs (TGP)
2018	Atkinson, Plump, Stepney: **Evolving Graphs by Graph Programming**

Introduction and Related Work
Parallel Distributed Genetic Programming

Parallel Distributed Genetic Programming (PDGP)

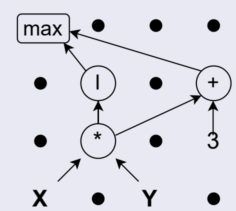
- Introduced by Poli [53, 54, 55, 56]
- Uses a **direct grid-based** graph representation model
- Each node in the graph is located in a **multi-dimensional** and **evenly spaced** grid
- Prefixed **regular** or **irregular** grid **shape**
- Connections between nodes are limited to be upwards (**feed-forward**)

Parallel Distributed Genetic Programming



Parallel Distributed Genetic Programming

Grid representation



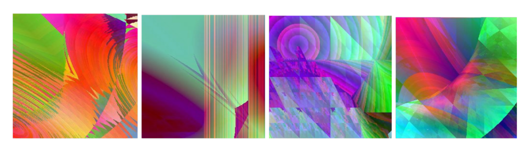
List representation

```
max    (0,0)   +1  +3

I      (0,1)    0
+      (3,1)   -2   0

*      (1,2)   -1  +1
3      (3,2)

X      (0,3)
Y      (2,3)
```


Parallel Distributed Genetic Programming

- Nodes can be deactivated with a **identity function**
- Implementation of **passing-through nodes**
- Genetic variation → Subgraph crossover and mutation

- First work towards CGP was done by Miller, Thompson, Kalganova, and Fogarty [22, 35, 37]
- Represents genetic programs as a two dimensional grid of program primitives.
- Loosely inspired by the architecture of digital circuits.



Automatically evolved images with CGP (Ashmore and Miller [1])

Cartesian Genetic Programming
Historical Background

Cartesian GP Timeline

1997	Miller, Thompson, Kalganova, Fogarty: Work towards CGP
1999	Miller, Thompson: Standard CGP
2004	Walker, Miller: Modular CGP
2005	Smith, Leggett, Tyrrell: Implicit Context Representation of CGP
2007	Clegg, Walker, Miller: Real-valued representation of CGP
2008	Walker, Miller: Embedded CGP
2011	Harding, Miller, Banzhaf: Self-Modifying CGP
2011	Harding, Graziano, Leitner, Schmidhuber: Multi-type CGP
2013	Turner, Miller: CGP encoded artificial neural networks
2014	Turner, Miller: Recurrent CGP
2016	Ryser-Welch, Miller, Swan and Trefzer: Iterative CGP
2018	Wilson, Miller, Cussat-Blanc, Luga: Positional CGP

Cartesian Genetic Programming (CGP)
Representation Model

- Program representation $\rightarrow$ acyclic and directed graph.
- Genotype-phenotype mapping $\rightarrow$ encoding-decoding of the graph
- Predominantly used without recombination $\rightarrow$ mutational $(1+\lambda)$ evolutionary algorithm.

Cartesian Genetic Programming (CGP)
Representation Model

Definition (Cartesian Genetic Program (CP))

A cartesian genetic program $\mathcal{P}$ is an element of $\mathcal{N}_i \times \mathcal{N}_f \times \mathcal{N}_o \times \mathcal{F}$:

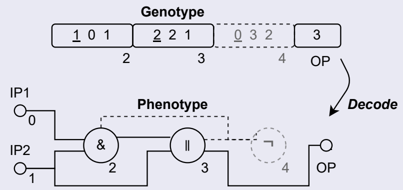
- $\mathcal{N}_i$ is a finite non-empty set of input nodes
- $\mathcal{N}_f$ is a finite set of function nodes
- $\mathcal{N}_o$ is a finite non-empty set of output nodes
- $\mathcal{F}$ is a finite non-empty set of functions

Cartesian Genetic Programming (CGP)
Representation Model

- Nodes of a cartesian genetic program are **continuously indexed**
- Indexing starts with the a value of 0 at the **first input node** and ends at the **last output node**.
- At **each node**, the node number is **increased by one**.
- Let $N = |N_i| + |N_f| + |N_o|$ be the number of nodes of a CP.

Cartesian Genetic Programming
Representation Model

Cartesian Genetic Programming



Function Lookup Table

Index	Symbol	Function
0	¬	Negation
1	&	Logical and
2	‖	Logical or
3	⊕	Exclusive or

Cartesian Genetic Programming (CGP)
Search Algorithm

- CGP is commonly used with a variant of the $(1+\lambda)$-EA $\rightarrow$ $(1+\lambda)$-CGP
- Implements a modified selection strategy called **neutrality**.
- Adapts a **genetic drift** to provide diversity during the evolutionary run.
- Individuals that have the same fitness are determined, and **one** of these **same-fitness individuals** is returned **uniformly at random**.

Cartesian Genetic Programming (CGP)
Search Algorithm

Algorithm 1 $(1+\lambda)$-EA variant used in CGP

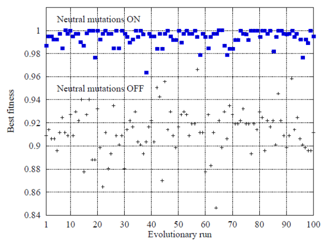
```
initialize(𝒫)                          ▷ Initialize parent individual
repeat                                  ▷ Until termination criteria not triggered
    𝒬 ← breed(𝒫)                        ▷ Breed λ offspring by mutation
    Evaluate(𝒬)                         ▷ Evaluate the fitness of the offspring
    𝒬⁺ ← best(𝒬, 𝒫)                     ▷ Get individuals which have better fitness as the parent
    𝒬⁼ ← equal(𝒬, 𝒫)                    ▷ Get individuals which have the same fitness as the parent
    ▷ If there exist individuals with better fitness
    if |𝒬⁺| > 0 then
        ▷ Choose one individual from 𝒬⁺ uniformly at random
        𝒫 ← 𝒬⁺[r], r ∼ U[0, |𝒬⁺| − 1]
        ▷ Otherwise, if there exist individuals with equal fitness
    else if |𝒬⁼| > 0 then
        ▷ Choose one individual from 𝒬⁼ uniformly at random
        𝒫 ← 𝒬⁼[r], r ∼ U[0, |𝒬⁼| − 1]
    end if
until 𝒫 meets termination criterion
return 𝒫                                ▷
```


Cartesian Genetic Programming (CGP)
Mutation

- Standard genetic operator $\rightarrow$ probabilistic **point mutation**.
- Genes are **selected uniformly at random** in the genotype.
- **Swaps gene values** in the valid range by chance.
- Genetic **variation** of **functionality** and **connectivity**.

Cartesian Genetic Programming (CGP)
Mutation

- A number of studies [71, 70, 39] have demonstrated the importance of **neutral mutations** that are based on **functional redundancy**
- Functional redundancy refers to many different programs representing the same function



Evolution of three-bit Boolean multiplier: Obtained fitness is show for each of 100 runs of 10 million generations with neutral mutations enabled (blue) compared with disabled neutral mutations (+)

Cartesian Genetic Programming (CGP)
Mutation

Algorithm 2 Probabilistic point mutation

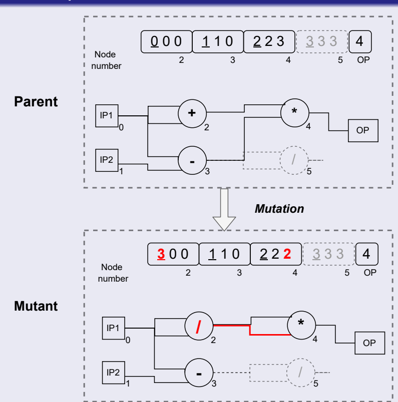
Input: Genome $\mathcal{G}$, Function set $\mathcal{F}$, Number of function nodes $\mathcal{N}_f$, Number of input nodes $\mathcal{N}_i$, Mutation rate $\mathcal{P}$

Output: Mutated Genome $\widetilde{\mathcal{G}}$

foreach $g \in \mathcal{G}$ **do** ▷ *Iterate over the genome*
  $X \sim \mathrm{Ber}(\mathcal{P})$ ▷ *Bernoulli random variable*
  if $X = 1$ **then** ▷ *Control the mutation strength with X*
    if g is a connection gene **then**
      ▷ *Determine the node number of the gene*
      $n \leftarrow \mathrm{NodeNumber}(g)$
      ▷ *Select g in the range of previous node indexes by chance*
      $g \leftarrow r,\ r \sim U[0,\ n-1]$
    else if g is a function gene **then**
      ▷ *Select g in the range of the function indexes by chance*
      $g \leftarrow r,\ r \sim U[0,\ |F|-1]$
    else ▷ *g is a output gene*
      ▷ *Select g in the range of the function and input nodes by chance*
      $g \leftarrow r,\ r \sim U[0,\ |N_f| + |N_i| - 1]$
    end if
  end if
end foreach
return $\mathcal{G}$ ▷ *Return the mutated genome*

Cartesian Genetic Programming (CGP)
Mutation

Example of standard point mutation in CGP



Cartesian Genetic Programming (CGP)
Mutation

- **Single active gene mutation (SAM)** is a variant of standard point mutation introduced by Goldman and Punch [17].
- SAM mutates the genome until an active gene is hit.
- Major advantage → **no parametrization** of the mutation rate is **needed**.
- Let $g_a = n_a * (1 + a) + n_o$ be the number of active gees, the active gene which will be mutated is selected uniformly at random with probability $\frac{1}{g_a}$.
  - n_a is the number of active nodes.
  - n_o is the number of outputs.
  - a is the maximum arity of a function node.

Algorithm 3 Single active gene mutation

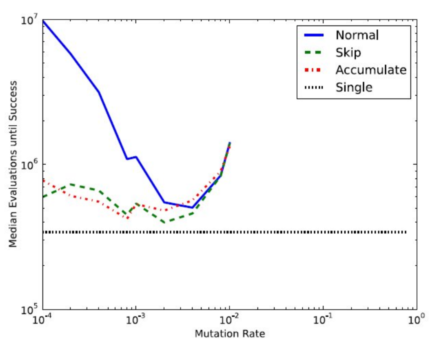
Input: Genome $\mathcal{G}$, Function set $\mathcal{F}$, Number of function nodes $\mathcal{N}_f$, Number of input nodes $\mathcal{N}_i$

Output: Mutated Genome $\widetilde{\mathcal{G}}$

active $\leftarrow$ **false**
repeat
  $g \leftarrow \mathcal{G}[r]$, $r \sim U[0, |\mathcal{G}| - 1]$ ▷ *Select a gene by chance*
  $n \leftarrow$ NodeNumber(g) ▷ *Determine the node number of the gene*
  active $\leftarrow$ NodeActive(n) ▷ *Check whether gene is active or not*
  if g is a connection gene **then**
    ▷ *Select g in the range of previous node indexes by chance*
    $g \leftarrow r$, $r \sim U[0, n - 1]$
  else if g is a function gene **then**
    ▷ *Select g in the range of the function indexes by chance*
    $g \leftarrow r$, $r \sim U[0, |F| - 1]$
  else ▷ *g is a output gene*
    ▷ *Select g in the range of the function and input nodes by chance*
    $g \leftarrow r$, $r \sim U[0, |N_f| + |N_i| - 1]$
  end if
until active is **true**

- Goldman and Punch [17] compared several mutation strategies on a set of Boolean functions → Standard, Skip, Accumulate and SAM.
- **Standard** is the standard (probabilistic) point mutation.
- **Skip** sets the offspring fitness to parent if phenotypes are identical.
- **Accumulate** mutates multiple genes until some active genes are changed.

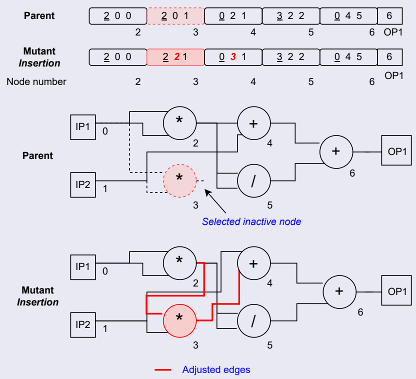
- SAM was close to the best performance of the other strategies on the majority of the tested problems.
- Showed good performance on the 3 bit parallel multiplier problem.



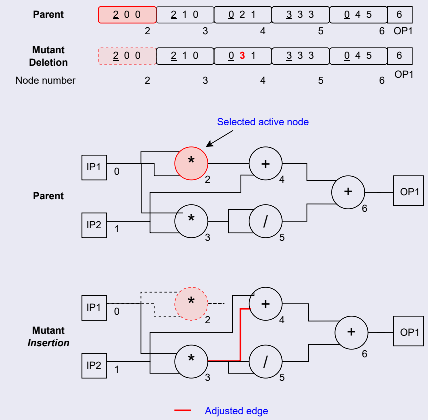
- Kalkreuth [23, 25, 26] recently adapted a set of chromosomal mutations: **insertion**, **deletion**, **inversion** and **duplication**.
- **Insertion** selects an **inactive** node and changes one or more connection genes in the genotype to make it **active**.
- **Deletion** alters connections to an active node so that the node becomes inactive.
- **Inversion*** permutates the function gene values of a set of successive active function nodes.
- **Duplication*** adapts chromosomal duplication by replacing of active function genes.

*Will be presented at GECCO'22. Looking forward to meet you at my poster!

Insertion Mutation



Deletion Mutation



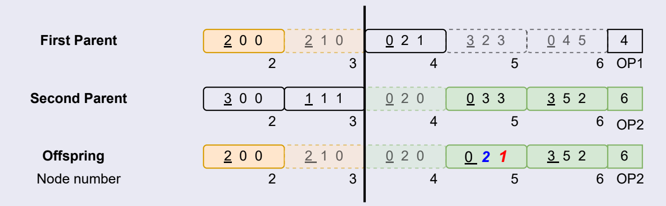
- Evaluation on a diverse set of Boolean function problems.
- Search performance improved significantly on the majority of the tested problems.
- Insertion and deletion **increase** the mean **active node range** of the evolutionary search.

- Miller [37] investigated **genotypic recombination** but observed that **it does not seem to add anything**.
- Recombination on **multiple chromosomes** with **independent fitness assessment**has been found **highly beneficial**.
- Clegg et al. [11] proposed a **floating point representation** of CGP and suggested that **intermediate recombination** might be useful for **symbolic regression**.
- **Cone-based crossover** has been proposed for modular CGP by Kaufmann et al. [32]
- da Silva et al. [12] proposed **path recombination** for Boolean **multiple-output problems** and obtained better search performance on the **single-chromosome representation**.
- Recently, Kalkreuth [28, 24] found that **subgraph** [27] and **function gene recombination** [20] can improve the search performance on various **symbolic regression problems**.

Cartesian Genetic Programming (CGP)

Recombination or not? A long and ongoing story...

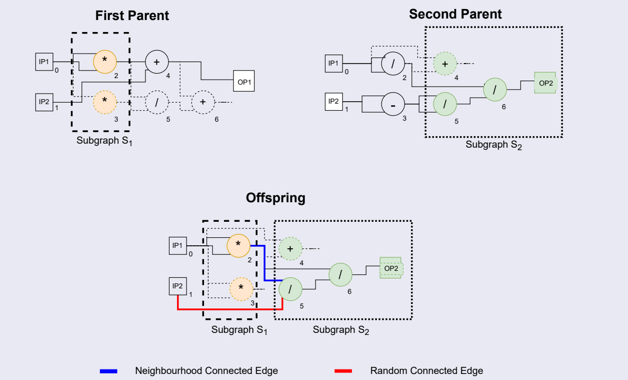
Subgraph Crossover



Cartesian Genetic Programming (CGP)

Recombination or not? A long and ongoing story...

Subgraph Crossover



Cartesian Genetic Programming (CGP)

Parametrization: Genotype Length

- The effectiveness of the search in relation to the number of nodes and mutation rate was investigated on a couple of **Boolean functions** [39].
- **Efficiency** of the search appeared to **continuously improve** as the **number of nodes increased**.
  - Least number of evaluations required for success when 95 % of the genes were inactive.
- However, Kalkreuth [28] demonstrated that small genotypes can perform effectively in the **symbolic regression** domain.

Cartesian Genetic Programming (CGP)

Parametrization: Population size

- Kaufmann and Kalkreuth [30, 31] and Kalkreuth [28] extensively investigated various parameters in CGP for effectiveness.
- Evaluating Boolean function and symbolic regression problems they found:
  - $\lambda = 1$ was often the most effective setting for the tested Boolean function problems.
  - Higher settings (greater than 4) of λ performed effective on various symbolic regression problems.

Linear Genetic Programming

Historical Background

- Linear Genetic Programming (LGP) represents programs as a list of instructions that manipulate registers and has a DAG dataflow.
- Earlier forms proposed by Nordin [42] aimed at directly evolving programs written in machine code.
- Other motivations include a controlled structure to study introns (Nordin and Banzhaf [44], Nordin et al. [47]).
- The advantage of some properties of the representation led to the proposal of current LGP (Brameier and Banzhaf [6, 7, 8]).

Linear Genetic Programming

Historical Background

Linear Genetic Programming Timeline

1993	Banzhaf: Precursor to Linear Genetic Programming
1994	Nordin: Compiling Genetic Programming System (CGPS)
1995	Nordin, Banzhaf: Extension to CGPS
1998	Nordin: Automatic Induction of Machine Code with Genetic Programming (AIM-GP)
1999	Nordin: Extension to AIM-GP for other architectures
2000	Heywood, Zincir-Heywood: Page-based Linear Genetic Programming
2001	Brameier, Banzhaf: **Linear Genetic Programming (LGP)**
2002	Kantschik, Banzhaf: Linear-Graph Genetic Programming
2010	Downey et al.: Crossover operators for multiclass classification
2011	Downey, Zhang: Parallel Linear Genetic Programming
2016	Sotto et al.: λ-Linear Genetic Programming

Linear Genetic Programming

Representation Model

LGP Programs

- Sequence of instructions of the form $(func, dest, arg_1, arg_2)$, where:
  - *func* is a function (e.g. plus, OR).
  - *dest* is the index of the destination register, and refers to the registers vector r.
  - arg_1 and arg_2 are indexes for which registers to use as arguments, referring to the same registers vector r.
  - arg_2 can also be a real-valued constant.

Linear Genetic Programming

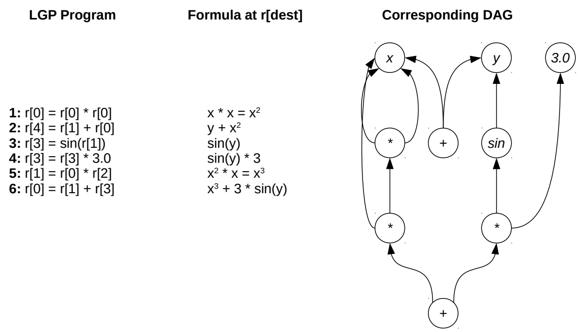
Representation Model

LGP Programs

- The maximal arity can vary, and functions with lower arity use only the first arguments.
- Intermediate and final results are stored in a registers vector r, which is initialized with the inputs to the problem in a cyclic manner ($r[i] = i\%nInputs$, initially).
- The output is the final value in a determined register, normaly $r[0]$ to $r[nOutputs - 1]$.

Linear Genetic Programming

Representation Model - Example program $f(x, y) = x^3 + 3 + sin(y)$



- When 6 registers are allowed, $r[0]$, $r[2]$, and $r[4]$ are initialized with input x and $r[1]$, $r[3]$, and $r[5]$ with y.
- Output is result in $r[0]$ after execution of all instructions.
- **Representation features:** Reuse (instruction 1), non-effective code (instruction 2).

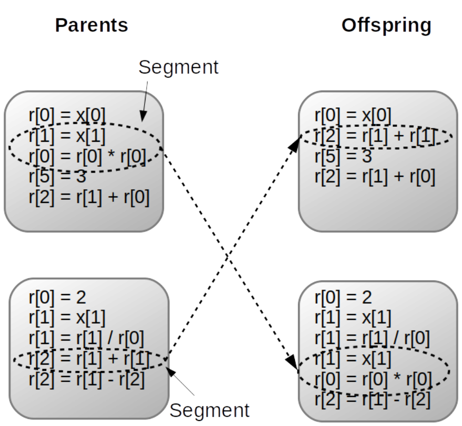
Linear Genetic Programming

Genetic Operators

- **Macro-operators:** Add, replace, or delete entire instructions.
  - **Crossover:** Two segments are exchanges between parents. Program length can be controlled by removing extra instructions after crossover.
  - **Macro-mutation:** Can add a new random instruction in a random position or delete a random instruction. If a maximum length is reached, can replace instructions instead of adding new ones.
- **Micro-operators:** Change individual elements inside instructions.
  - **Micro-mutation:** Can change the function, destination register, or arguments of an instruction. For constant arguments, can apply a perturbation.

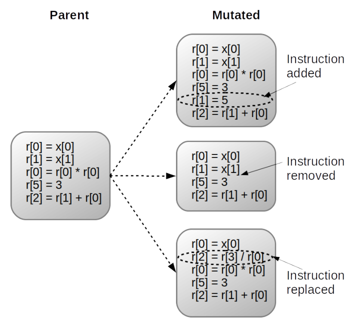
Linear Genetic Programming

Genetic Operators - Example Crossover

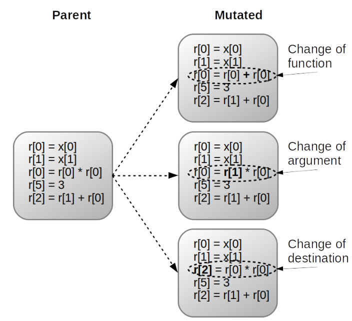


Linear Genetic Programming

Genetic Operators - Example Mutations



(a) Macro-mutation



(b) Micro-mutation

Linear Genetic Programming
Genetic Operators

- Effective genetic operators forcing changes to the effective code generally leads to better results for LGP (Brameier and Banzhaf [8]).
- There are some alternative forms of crossover that were proposed to LGP:
  - Maximum Homologous Crossover ([51]
  - Special crossover techniques for multiclass classification ([14]).
  - Headless chicken and homologous crossover ([34]).
- However, similarly to CGP, crossover is not often used by LGP. Furthermore, recente results by Sotto et al. [59] suggest that using only micro-mutations on a fixed-size genome should be preferred.

Linear Genetic Programming
Search Algorithm

- Traditionally, LGP employs a steady-state search algorithm that replaces individuals in-place.

Algorithm 4 Pseudocode for the LGP algorithm. The outer **repeat** loop refers to the number of generations, while the inner **repeat** loop refers to the application of genetic operators inside a generation.

```
Initialize population of size P.
Calculate the population fitness.
repeat
    repeat
        Do two size T tournaments, returning two winners and two losers.
        Replace losers by copies of winners.
        Apply macro-operator (crossover or macro-mutation) on both winners.
        Apply micro-mutation on the two resulting individuals.
        Calculate fitness of the two modified winners.
    until P/2 individuals are processed
until stopping criterion is met.
```


Linear Genetic Programming
Search Algorithm

- Recent results suggest that a $(1+\lambda)$ search algorithm similar to CGP is also recommended for LGP.
  - LGP using a variation of the $(\mu + \lambda)$ algorithm with macro- and micro-mutations was able to produce state-of-the-art results for different trails for the Ant Trail problem, including the difficult Los Altos Hills trail (Sotto et. al. [58]).
  - The $(1+\lambda)$ variation with only micro-mutations also improved over standard LGP in many situations, specially for digital circuits design (Sotto et. al. [59]).

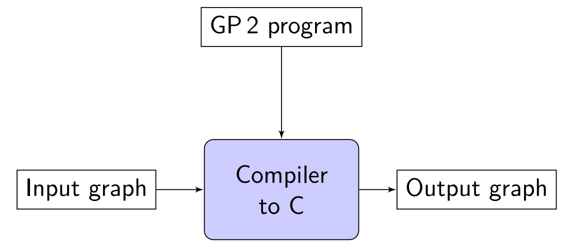
Linear Genetic Programming
Improvements on standard LGP

- **Linear Graph GP [29]:**
  - Nodes are a sequence of instructions followed by a branching node that leads to one or more different nodes.
  - Represent more complex structures in a more natural way, and initial results show a great improvement in performance in comparison to linear GP.
- **Parallel LGP [13]:**
  - Proposed to solve the problems of disruptiveness and high amount of neutral mutations with larger program sizes.
  - Program is an ordered set of blocks of instructions. Each block is evaluated in parallel as an independent program and the final result is the sum of the final register vectors.
  - Especially for longer programs, the technique is able to improve on standard LGP for the multi-class classification task.

Evolving Graphs by Graph Programming (EGGP)
Representation Model

- EGGP as introduced by Atkinson et al. [3, 2] is based on the graph programming language. GP2 [52].
- Uses a **direct graph representation**.
- Extension of GP2 that has **probabilistic elements** to support **EA applications**.

Evolving Graphs by Graph Programming (EGGP)
Graph Programming Language GP 2



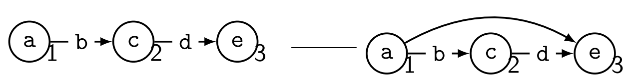
- Experimental language for graphs.
- Rule-based visual manipulation of graphs.
- Computationally complete.
- Non-deterministic.

Evolving Graphs by Graph Programming (EGGP)
Graph Programming Language GP 2

A graph is *transitive* if for every directed path $v_1 \rightsquigarrow v_2$ where $v_1 \neq v_2$ there is an edge $v_1 \rightarrow v_2$.

`Main := link!`

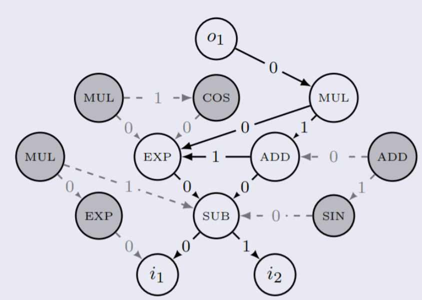
`link(a,b,c,d,e:list)`



`where not edge(1, 3)`

Evolving Graphs by Graph Programming (EGGP)
EGGP Individual

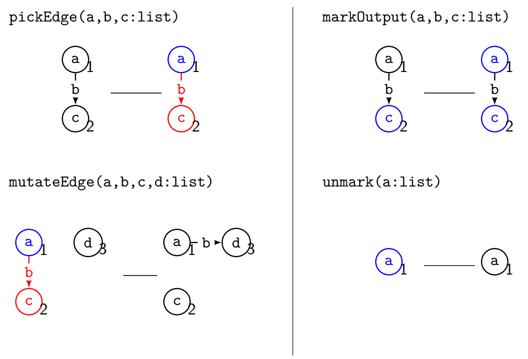
EGGP Individual



An example EGGP individual. The single output computes the function $o_1 = e^{i_1 - i_2} \times ((i_1 - i_2) + e^{i_1 - i_2})$

Evolving Graphs by Graph Programming
Mutation

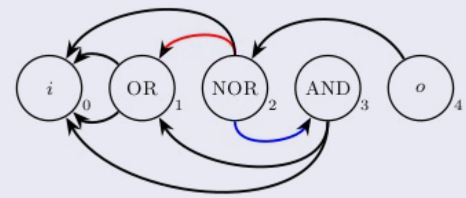
```
Main := pickEdge; markOutput!; mutateEdge; unmark!
```



Mutating an edge of an EGGP individual.

Evolving Graphs by Graph Programming
Mutation

Feed-forward preserving edge mutation

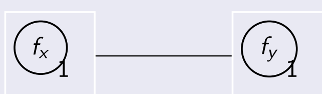


An edge (red) directed from node 2 to node 1 is replaced with an edge (blue) directed to node 3. This mutation produces a valid circuit but is impossible in CGP as it does not preserve order.

Evolving Graphs by Graph Programming
Mutation

Node function mutation

mutateFunction-f_y(f_x:string)



Mutating the function of an EGGP individual's node to some function f_y where f_x is the existing function of the node being mutated. An equivalent rule can be constructed for each function in the function set.

Evolving Graphs by Graph Programming (EGGP)
Recombination

- EGGP is also based on fixed size programs, mutations to nodes or connections, and the $(1+\lambda)$ evolutionary scheme.
- Atkinson proposes a form of recombination called **Horizontal Gene Transfer** (HGT) [4, 5], where active portion of one parent is copied as the inactive portion of the other parent.
- As it requires more than one parent, HGT is used with a $\mu \times \lambda$ strategy, where μ parents each produce λ children that compete only with their own parent.
  - Parallel $1+\lambda$ EAs with genetic information shared horizontally between elite individuals.
- HGT events followed by edge mutations may perform operations very similar to subgraph crossover in CGP and PDGP.

Properties of the DAG Representation
Some Results

- **Modularity through code reuse:**
  - Nordin [46] already observed that his machine code system evolved more compact solutions, and Fogelberg and Zhang [16] show that LGP evolves shorter and simpler solutions.
  - Controlling the degree of intermediate results reuse was critical for CGP and LGP on digital circuits tasks (Sotto et. al. [59]).
- **Neutral search via inactive genes:**
  - Neutral search in CGP has helped escaping from local optima by navigating neutral networks (Turner and Miller [64]).
  - Neutral search in LGP was able to improve search results and enable the evolution of gradually more complex programs when required (Sotto et. al [60]).

Comparison Between Graph-based Methods
Methods Features

- Summarization of work done in Sotto et. at. [59].
- **Motivations:**
  - How does the EA, operator, and representation details of each graph-based approach affect performance?
  - Do graphs present an advantage over trees when the same EA is used?

Technique	Representation	Operators	EA
GP	Tree	Crossover, subtree and point mutation	Generational
LGP	List of instructions (Genotype) / DAG (Phenotype)	Macro and micro-mutations	Steady-state
CGP	List of nodes (Genotype) / Grid, DAG (Phenotype)	Point mutation	$(1+\lambda)$ EA
EGGP	DAG (not feedforward)	Point mutation	$(1+\lambda)$ EA

Comparison Between Graph-based Methods
Search Algorithm

- **Methods:** GP, LGP, LGP-micro (only with micro-mutations, closer to CGP), CGP, EGGP.
- **Search algorithms:** Generational, Steady-state, $(1+\lambda)$.
- **Experiment:** Fix the method and compare the performance of the search algorithms.
- **Main conclusions:**
  - For regression benchmarks and datasets, the best search scheme varies for each combination of method and problem.
  - For digital circuits evolution, the $(1+\lambda)$ scheme is by far the most recommended method (benefit from exploitation).

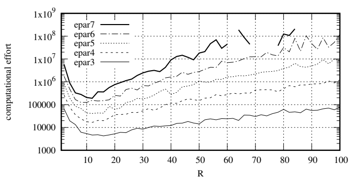
Comparison Between Graph-based Methods
CGP, LGP, and EGGP

- **Methods:** LGP, LGP-micro (only with micro-mutations, closer to CGP), CGP, EGGP.
- **Search algorithms:** Generational, Steady-state, $(1+\lambda)$.
- **Experiment:** Fix the search scheme and compare the performance of the graph-based methods.
- **Main conclusions:**
  - For regression, results are mixed, but there is tendendcy that EGGP with a generational scheme works best for more complex datasets.
  - For digital circuits evolution, EGGP worked the best. However, for parity functions, LGP-micro was better.
  - Fixed-length genomes with point-mutations are recommended, as LGP-micro, CGP, and EGGP.

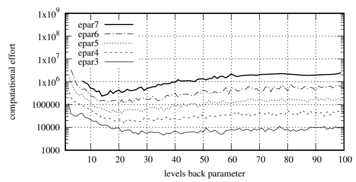
Comparison Between Graph-based Methods

Reuse of Intermediate Results

- Why does LGP-micro work better for parity functions?
- Analysis on the number of registers R (LGP) and levels-back l (CGP).
- Lower values = more reuse of intermediate results.
- From plots[2]: More code reuse is beneficial and can be set so as LGP and CGP perform similarly. It is possible that LGP is less robust due to registers being overwritten.



(a) R in LGP



(b) l in CGP

[2] Logarithmic scale

Comparison With Tree-based GP

- **Methods:** GP, LGP-micro (only with micro-mutations, closer to CGP), EGGP.
- **Search algorithms:** Generational, Steady-state, $(1 + \lambda)$.
- **Experiment:** Fix the search scheme and compare the performance of the graph-based methods to tree-based GP.
- **Main conclusions:**
  - For regression, graphs presented some disadvantage to trees, but worked better for complex real-world datasets.
  - For digital circuits evolution, graphs worked better regardless of the search scheme, but specially better under the $(1 + \lambda)$ algorithm.
  - Possible features responsible for the improvement in performance are code reuse and intensified neutral search with the $(1 + \lambda)$ EA.

Digital Circuit Design

- Evolution of gate-level and functional-level digital circuits represents a complex multi-objective design problem that can be efficiently addressed using CGP.
- Many competitive results have been achieved since the introduction of the CGP (see ehw.fit.vutbr.cz)
  - Evolutionary design of image filters [57, 66]
  - Evolutionary design of gate-level digital circuits [38, 68]
  - Evolutionary optimization of digital circuits [67, 65]
  - Evolutionary design of approximate digital circuits [69, 41, 10]
  - Evolutionary design of cryptographic Boolean functions [18, 19, 49, 50]

Humies	Subject of research	Key innovation	Evolved circuits
2008 silver	Evolutionary synthesis of benchmark circuits	Non-functional property (testability) in the fitness function	synthetic circuits (2k – 1.2M gates) [48]
2011 silver	Evolutionary optimization of digital circuits	SAT solver in the fitness function	LGSynth93 benchmarks (128 inputs, 1.5k gates) [67]
2015 gold	Evolutionary approximation of digital circuits	Heuristic seeding strategies	4-bit multiplier, 8-bit 25-input median circuit [69]
2018 bronze	Evolutionary design of approximate arithmetic circuits with formal error guarantees	Verifiability driven search	multipliers (up to 32-bit), adders (up to 128-bit) [10]

How Can We Compete with Conventional Synthesis Tools?

- The synthesis tools are mature, but they are known to produce non-optimal (sometimes even far from optimum) circuits for some instances [15].
  - The evolution is not biased towards a particular class of solutions.
- The synthesis tools represent circuits using a directed acyclic graph denoted as and-inverter graph. This representation is simple and scalable, and leads to simple algorithms but it suffers from an inherent bias [33].
  - The evolution can be performed directly at the gate-level representation to avoid inefficiencies of internal representations.
- The synthesis tools have never been constructed to perform the synthesis of incompletely specified or approximate (erroneous) circuits.
  - The acceptance of partially working solutions during the design process is natural for evolution.
- The synthesis tools can't easily handle additional constraints (e.g. accurate multiplication by zero in case of approximate multipliers)
  - It is natural for evolution to include multiple constraints in fitness function.

Problem Classification - Specification Completeness

Completely specified problems

- **Fitness value**: calculated according to the knowledge of all possible valid input-output mappings
- **Problem complexity**: A difficuilt problem, scalability of the evaluation needs to be properly addressed
- **Examples**: Synthesis of gate-level circuits, Evolution of cryptographic functions

Incompletely specified problems

- **Fitness value**: the correctness can only be evaluated using a subset of all possible input vectors (it is hard or even impossible to define correct output values for all possible inputs)
- **Problem complexity**: Substantially less challenging
- **Examples**: Evolution of approximate circuits, classifiers, filters, hash functions, predictors

Problem Classification - Problem Formulation

Evolutionary optimization

- **Initial population**: typically seeded with a known sub-optimal but fully working solution
- **Goal**: improve non-functional parameteres of the existing solution
- **Scalability**: Very complex real-world circuits can be handled.
- **Examples**: Optimization of gate-level circuits

Evolutionary design

- **Initial population**: generated randomly, a heuristic may be used to start with a partially working solution.
- **Goal**: discover a novel implementation
- **Scalability**: Good for functional-level evolution and incompletely specified problems, limited for completely specified problems (hundred CGP nodes)
- **Examples**: Evolution of gate-level circuits, image filters, classifiers

Evolutionary Design of Image Filters and Classifiers

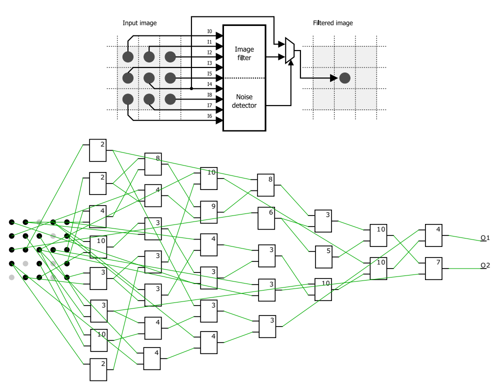
- Evolutionary design of image filters is a popular topic studied since the introduction of CGP.
- **Classification**: incompletely specified problem, evolutionary design
- **CGP node functions**: high-level functions such as 8-bit addition, multiplication, division, absolute difference, conditional assignment, etc.
- **Fitness function**: difference between filtered and original image (MAE, PSNR, ...)

$$MAE = \frac{1}{WH}\sum_{i=1}^{W}\sum_{j=1}^{H}|\text{filtered}[i,j] - \text{original}[i,j]|.$$

- **Scalability**: very good (acceleration of the evaluation in FPGAs is the big advantage of CGP [])

Evolutionary Design of Image Filters and Classifiers

- Evolutionary design allow simultaneous evolution of noise detector and noise filter (only the affected pixels are modified).
- Example of one of the best evolved filters for impulse burst noise operating on 5x5 image kernel [66]:



Evolutionary design of Gate-level Circuits

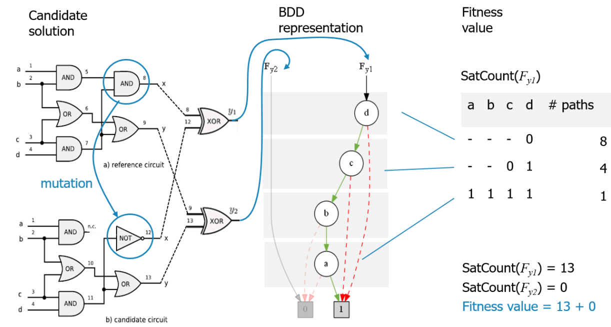
- **Classification**: completely specified problem, evolutionary design
- **CGP node functions**: common 2-input logic gates plus inverter
- **Fitness function**: Hamming distance between the specification (TT) and response of a candidate solution C

$$\mathit{fitness}(C) = \mathrm{HD}(C, \mathrm{TT}) = \sum_{\forall x \in \mathbb{B}^n} \mathit{1's}(C(x) \oplus \mathrm{TT}(x))$$

- **Scalability**: limited due to the complexity of the fitness calculation (#P-complete) and biases in CGP encoding
- **Evolved circuits**: various LGSynth benchmarks; two most complex evolved circuits: frg1 – 28 inputs, 44 gates (57.3% improvement compared to ABC), alu4 – 14 inputs, 70 gates (93.4% improvement)

Evolutionary Design of Gate-level Circuits

- To evolve circuit with more than ~ 24 inputs, scalability of fitness evaluation needs to be addressed.
- Vasicek and Sekanina [68] proposed to use Binary Decision Diagrams (BDDs) instead of Truth tables (speedup 1-2 orders of magnitude on LGSynth).



Evolutionary Optimization of Gate-level Circuits

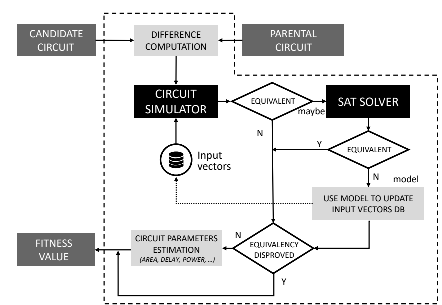
- **Classification**: completely specified problem, evolutionary optimization
- **CGP node functions**: common 2-input logic gates plus inverter
- **Fitness function**: non-functional parameter (e.g. the circuit size) and functional equivalence constraint

$$\mathit{fitness}(C) = \begin{cases} \mathit{cost}(C), & \text{if } f(C) \equiv f(P). \\ \infty, & \text{otherwise,} \end{cases}$$

- **Scalability**: typically very good, depends only on the scalability of the functional equivalence checking
- **Optimized circuits**: hundreds of inputs, thousands of gates directly [65]; more complex real-world instances (millions of gates) can be optimized using windowing [33]

Evolutionary Optimization of Gate-level Circuits

- **Observation**: Each candidate solution created by means of genetic operators must be functionally equivalent with its parent in order to be further evaluated.
- **Key idea**: Apply formal equivalence checking to decide whether a candidate circuit is functionally correct or not [67]. The latest hybrid approach can prove the equivalence in few milliseconds even for circuits with hundred inputs (2^{100} combinations) [33].



Evolutionary Synthesis of Approximate Circuits

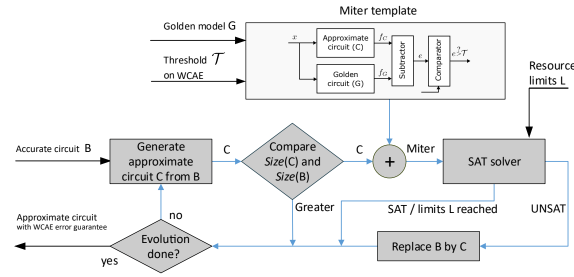
- **Classification**: incompletely specified problem, evolutionary optimization
- **CGP node functions**: common 2-input logic gates plus inverter
- **Fitness function**: non-functional parameter (e.g. the circuit size) and error constraint (e.g. worst-case absolute error)

$$
fitness(C) = \begin{cases} cost(C), & \text{if } error(C) \leq \tau. \\ \infty, & \text{otherwise,} \end{cases}
$$

- **Scalability**: typically very good, depends only on the scalability of the error constraint checking procedure
- **Optimized circuits**: logic circuits (tens of inputs []); arithmetic circuits (up to 32-bit multipliers, up to 128-bit adders [10])

Evolutionary Synthesis of Approximate Circuits

- Formal verification of some circuits (e.g. multipliers) is extremely hard, how to overcome this issue without sascrificing the requirement for a formal error guarantee?
- A verifiability driven search has been introduced in [10] to discover high-quality approximate circuits.



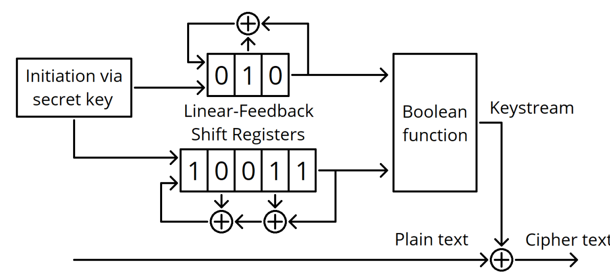
Evolutionary Design of Cryptographic Boolean Functions

- **Boolean functions** with good cryptographic properties form integral part of many cryptographic algorithms.
- Multiple mutually competing properties are required to provide protection against all possible types of attack.
- Uses include: Stream ciphers, block ciphers, hash functions, number generators, and side-channel masking schemes [21, 49].

- **Evolutionary design** is able to create a much wider range of functions than the traditional algebraic construction [9, 50].
- Can be easily adjusted to create functions of various sizes and properties [18, 19, 21].
- Led to the discovery of new functions with previously unknown cryptographic values [49].

Properties of Boolean Functions in Stream Ciphers

- Boolean function combines the outputs of multiple LFSRs to create a keystream used to cipher or decipher messages [9].
- **Balancedness** prevents the detection of statistical bias.
- **Nonlinearity** prevents linear approximation of the output.
- **Correlation immunity** prevents attacks via fixed inputs.



- Algebraic **degree** and **immunity** prevent yet further attacks.

Evolutionary Design of Cryptographic Boolean Functions

- **The graph-based representation** allows significantly outperform other evolutionary approaches such as GA [18, 50].
- All GP variants are competitive, and each is suitable for designing Boolean functions with different properties [18, 19].

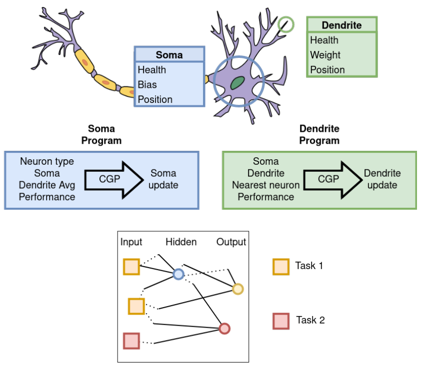
Function	GP	CGP	LGP
Bent-6	485	**331**	535
Bent-12	**1385**	1994	3701
Balanced-12	965	**619**	1191
Resilient-12	**1365**	1509	2184
Siegenthaler-12	214414	233049	**198551**
Masking-6-3	218505	**135585**	356539

Table: The number of evaluations required to find a cryptographic function with the desired properties.

Artificial Neuroevolution

- Turner and Miller investigated the evolution of artificial neural networks with CGP [61, 62, 63].
- CGP has recently been used to evolve neural programs that grow neural networks [40, 36].
- The programs can build a brain-like network that can solve a number of problems simultaneously.
  - Classification.
  - Re-inforcement learning.
- The aim is to evolve programs that can construct neural networks that can learn to solve many problems simultaneously.

Artificial Neuroevolution



Each neuron has a position, health and bias and a variable number of dendrites. Each dendrite has a position, health and weight. The behaviour of a neuron soma is governed by a single evolved program. In addition each dendrite is governed by another single evolved program.

Acknowledgments

We thank **Jakub Husa** from the Brno University of Technology for sharing his slides about his work on cryptographic function synthesis with GP.

We also thank **Paul Kaufmann** from the Westphalian University of Applied Sciences for providing the four bit digital adder and multiplier benchmark.

Julian Miller gave us permission to use the material of his tutorials.

Timothy Atkinson from NNAISENSE provided us his presentation material about Evolving Graphs by Graph Programming.

Thanks to both of you!

Resources I

1. **Graph-based GP Tutorial GitHub Repository:** https://github.com/RomanKalkreuth/graph-based-gp-tutorial
2. **Resources at www.cartesiangp.com:** https://www.cartesiangp.com/resources
3. **ECJ GitHub Repository:** https://github.com/GMUEClab/ecj
4. **Revised ECJ CGP Contrib Package:** https://github.com/RomanKalkreuth/ecj-contrib-cgp-revised
5. **Extension of Julian Miller's C Implementation of CGP:** https://github.com/paul-kaufmann/cgp
6. **Cartesian Genetic Programming Toolbox by Zdenek Vasicek and Lukas Sekanina** https://www.fit.vut.cz/research/product/61/

Resources II

7. **Boolean Benchmark Builder:** https://github.com/RomanKalkreuth/boolean-benchmark-builder
8. **Linear Genetic Programming implementation in Python and C++:** https://github.com/leo-sotto/LGP
9. **Linear Genetic Programming implementation on the JVM using Kotlin:** https://github.com/JedS6391/LGP
10. **Evolving Graphs by Graph Programming C implementation:** https://github.com/timothyatkinson/GraphComparison

Supplementary Material

Supplementary Material Content

1. **LGP:** Historical background and genetic operators
2. **EGGP:** Example of edge mutations

Linear Genetic Programming
Historical Background

Evolving Machine Code

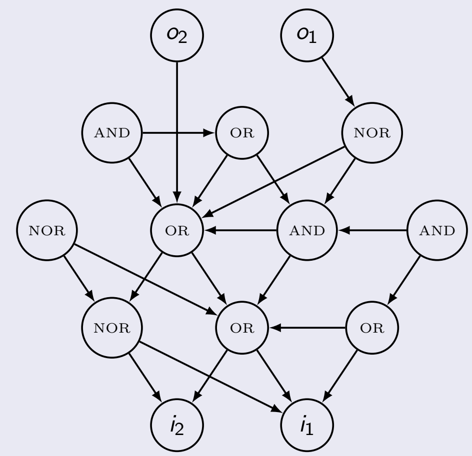
- Proposed by Nordin [42, 45] as CGPS, later called AIM-GP ([43, 46]).
  - **Program:** Sequence of binary numbers that encode syntactically closed instructions (e.g. a:=a+1).
  - **Operators:** Crossover exchanges sequences of genes, mutations change the binary numbers.
  - A system in C directly manipulates the machine code programs.
- Achieved huge speedups in comparison to a complete C (100x faster) or LISP implementation (2000x faster), and to a multi-layer perceptrons (200x faster).

Linear Genetic Programming
c

- Effective genetic operators has proven to lead to better results for LGP (Brameier and Banzhaf [8]).
  - **Effective crossover:** Applies crossover after removing the non-effective code of programs.
  - **Effective macro-mutation:** For adding, it adds a new instruction in a position where one of the following effective instructions uses its destination register. For deleting, it chooses an effective instruction. For replacing, it tries to replace a non-effective instruction by an effective one.
  - **Effective micro-mutation:** The chosen instruction must be effective.

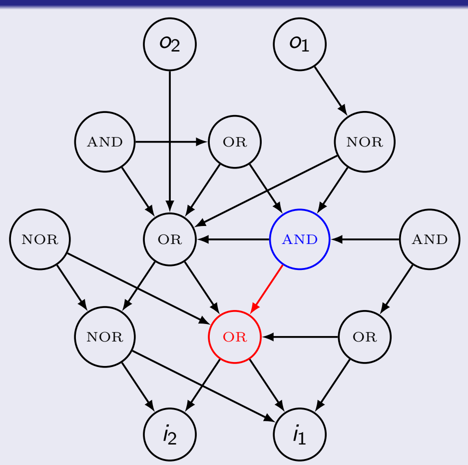
Evolving Graphs by Graph Programming
Mutation

Original EGGP Individual



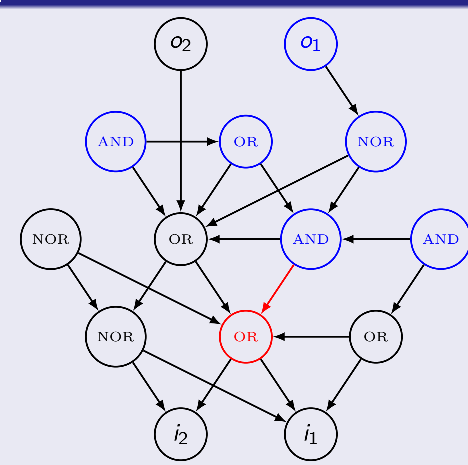
Evolving Graphs by Graph Programming
Mutation

Pick Edge

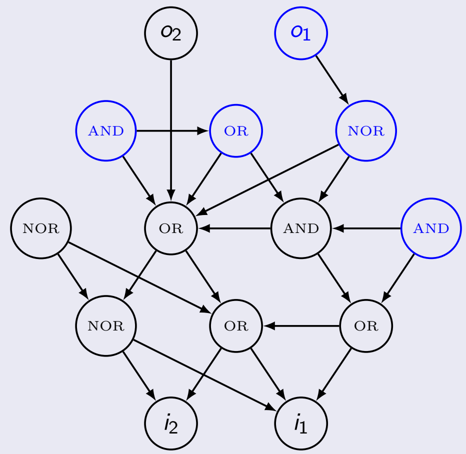


Evolving Graphs by Graph Programming
Mutation

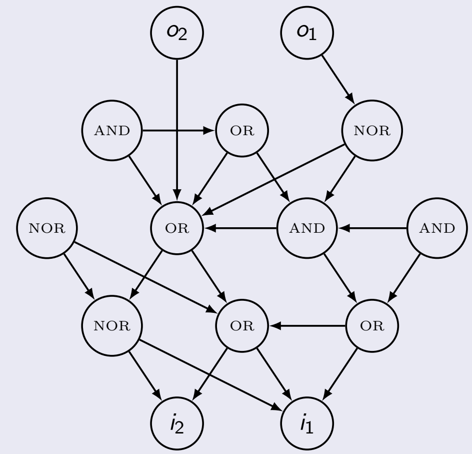
Mark Output



Mutate Edge



Unmark



References I

[1] Laurence Ashmore and J Miller. Evolutionary art with cartesian genetic programming. *Technical Online Report*, 2004.

[2] Timothy Atkinson. *Evolving graphs by graph programming*. PhD thesis, University of York, UK, 2019.

[3] Timothy Atkinson, Detlef Plump, and Susan Stepney. Evolving graphs by graph programming. In Mauro Castelli, Lukás Sekanina, Mengjie Zhang, Stefano Cagnoni, and Pablo García-Sánchez, editors, *Genetic Programming - 21st European Conference, EuroGP 2018, Parma, Italy, April 4-6, 2018, Proceedings*, volume 10781 of *Lecture Notes in Computer Science*, pages 35–51. Springer, 2018.

[4] Timothy Atkinson, Detlef Plump, and Susan Stepney. Evolving graphs with horizontal gene transfer. In Anne Auger and Thomas Stützle, editors, *Proceedings of the Genetic and Evolutionary Computation Conference, GECCO 2019, Prague, Czech Republic, July 13-17, 2019*, pages 968–976. ACM, 2019.

[5] Timothy Atkinson, Detlef Plump, and Susan Stepney. Horizontal gene transfer for recombining graphs. *Genet. Program. Evolvable Mach.*, 21(3):321–347, 2020.

[6] M. Brameier and W. Banzhaf. A comparison of linear genetic programming and neural networks in medical data mining. *IEEE Transactions on Evolutionary Computation*, 5(1):17–26, 2001.

[7] Markus Brameier. *On Linear Genetic Programming*. PhD thesis, Fachbereich Informatik, Universität Dortmund, Germany, 2004.

[8] Markus Brameier and Wolfgang Banzhaf. *Linear Genetic Programming*. Number XVI in Genetic and Evolutionary Computation. Springer, 2007.

[9] Claude Carlet, Yves Crama, and Peter L Hammer. Boolean functions for cryptography and error-correcting codes., 2010.

[10] Milan Ceska, Jiri Matyas, Vojtech Mrazek, Lukas Sekanina, Zdenek Vasicek, and Tomas Vojnar. Approximating complex arithmetic circuits with formal error guarantees: 32-bit multipliers accomplished. In *2017 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. IEEE, November 2017.

[11] Janet Clegg, James Alfred Walker, and Julian Francis Miller. A new crossover technique for cartesian genetic programming. In Hod Lipson, editor, *Genetic and Evolutionary Computation Conference, GECCO 2007, Proceedings, London, England, UK, July 7-11, 2007*, pages 1580–1587. ACM, 2007.

[12] José Eduardo Henriques da Silva and Heder Bernardino. Cartesian genetic programming with crossover for designing combinational logic circuits. In *7th Brazilian Conference on Intelligent Systems, BRACIS 2018, São Paulo, Brazil, October 22-25, 2018*, pages 145–150. IEEE Computer Society, 2018.

[13] Carlton Downey and Mengjie Zhang. Parallel linear genetic programming. In *Proceedings of the 14th European Conference on Genetic Programming, EuroGP 2011*, pages 178–189, Turin, Italy, 2011. Springer Verlag.

References II

[14] Carlton Downey, Mengjie Zhang, and Will N. Browne. New crossover operators in linear genetic programming for multiclass object classification. In *Proceedings of the 12th Annual Conference on Genetic and Evolutionary Computation*, GECCO '10, page 885–892, New York, NY, USA, 2010. Association for Computing Machinery.

[15] Petr Fiser, Jan Schmidt, Zdenek Vasicek, and Lukas Sekanina. On logic synthesis of conventionally hard to synthesize circuits using genetic programming. In *13th IEEE Symposium on Design and Diagnostics of Electronic Circuits and Systems*. IEEE, April 2010.

[16] Christopher Fogelberg and Mengjie Zhang. Linear genetic programming for multi-class object classification. In *AI 2005: Advances in Artificial Intelligence, 18th Australian Joint Conference on Artificial Intelligence, Proceedings*, volume 3809 of *Lecture Notes in Computer Science*, pages 369–379, Sydney, Australia, 2005. Springer.

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

[18] Jakub Husa. Comparison of genetic programming methods on design of cryptographic boolean functions. In *European Conference on Genetic Programming*, pages 228–244. Springer, 2019.

[19] Jakub Husa. Designing correlation immune boolean functions with minimal hamming weight using various genetic programming methods. In *Proceedings of the Genetic and Evolutionary Computation Conference Companion*, pages 342–343, 2019.

[20] Jakub Husa and Roman Kalkreuth. A comparative study on crossover in cartesian genetic programming. In Mauro Castelli, Lukás Sekanina, Mengjie Zhang, Stefano Cagnoni, and Pablo García-Sánchez, editors, *Genetic Programming - 21st European Conference, EuroGP 2018, Parma, Italy, April 4-6, 2018, Proceedings*, volume 10781 of *Lecture Notes in Computer Science*, pages 203–219. Springer, 2018.

[21] Domagoj Jakobovic, Stjepan Picek, Marcella SR Martins, and Markus Wagner. Toward more efficient heuristic construction of boolean functions. *Applied Soft Computing*, 107:107327, 2021.

[22] T. Kalganova. Evolutionary approach to design multiple-valued combinational circuits. In *Proceedings. of the 4th International conference on Applications of Computer Systems (ACS'97)*, pages 333–339, Szczecin, Poland, 1997.

[23] Roman Kalkreuth. Two new mutation techniques for cartesian genetic programming. In Juan Julián Merelo Guervós, Jonathan M. Garibaldi, Alejandro Linares-Barranco, Kurosh Madani, and Kevin Warwick, editors, *Proceedings of the 11th International Joint Conference on Computational Intelligence, IJCCI 2019, Vienna, Austria, September 17-19, 2019*, pages 82–92. ScitePress, 2019.

References III

[24] Roman Kalkreuth. A comprehensive study on subgraph crossover in cartesian genetic programming. In Juan Julián Merelo Guervós, Jonathan M. Garibaldi, Christian Wagner, Thomas Bäck, Kurosh Madani, and Kevin Warwick, editors, *Proceedings of the 12th International Joint Conference on Computational Intelligence, IJCCI 2020, Budapest, Hungary, November 2-4, 2020*, pages 59–70. SCITEPRESS, 2020.

[25] Roman Kalkreuth. *An Empirical Study on Insertion and Deletion Mutation in Cartesian Genetic Programming*, pages 85–114. Springer International Publishing, Cham, 2021.

[26] Roman Kalkreuth. Phenotypic duplication and inversion in cartesian genetic programming applied to boolean function learning. In *Genetic and Evolutionary Computation Conference, Boston, USA, July 9-13, 2022, Companion Material Proceedings*, Genetic and Evolutionary Computation Conference 2022 (GECCO '22), New York, NY, USA, 2022. ACM.

[27] Roman Kalkreuth, Günter Rudolph, and Andre Droschinsky. A new subgraph crossover for cartesian genetic programming. In James McDermott, Mauro Castelli, Lukás Sekanina, Evert Haasdijk, and Pablo García-Sánchez, editors, *Genetic Programming - 20th European Conference, EuroGP 2017, Amsterdam, The Netherlands, April 19-21, 2017, Proceedings*, volume 10196 of *Lecture Notes in Computer Science*, pages 294–310, 2017.

[28] Roman Tobias Kalkreuth. *Reconsideration and Extension of Cartesian Genetic Programming*. PhD thesis, 2021.

[29] Wolfgang Kantschik and Wolfgang Banzhaf. Linear-graph gp - a new gp structure. In *Proceedings of the 5th European Conference on Genetic Programming*, EuroGP '02, page 83–92, Berlin, Heidelberg, 2002. Springer-Verlag.

[30] Paul Kaufmann and Roman Kalkreuth. Parametrizing cartesian genetic programming: An empirical study. In Gabriele Kern-Isberner, Johannes Fürnkranz, and Matthias Thimm, editors, *KI 2017: Advances in Artificial Intelligence - 40th Annual German Conference on AI, Dortmund, Germany, September 25-29, 2017, Proceedings*, volume 10505 of *Lecture Notes in Computer Science*, pages 316–322. Springer, 2017.

[31] Paul Kaufmann and Roman Kalkreuth. On the parameterization of cartesian genetic programming. In *IEEE Congress on Evolutionary Computation, CEC 2020, Glasgow, United Kingdom, July 19-24, 2020*, pages 1–8. IEEE, 2020.

[32] Paul Kaufmann and Marco Platzner. Advanced techniques for the creation and propagation of modules in cartesian genetic programming. In Conor Ryan and Maarten Keijzer, editors, *Genetic and Evolutionary Computation Conference, GECCO 2008, Proceedings, Atlanta, GA, USA, July 12-16, 2008*, pages 1219–1226. ACM, 2008.

[33] Jitka Kocnova and Zdenek Vasicek. EA-based resynthesis: an efficient tool for optimization of digital circuits. *Genetic Programming and Evolvable Machines*, 21(3):287–319, January 2020.

[34] W.B. Langdon and W. Banzhaf. Repeated sequences in linear genetic programming genomes. *Complex Systems*, 15(4):285 – 306, 2005.

References IV

[35] J. F. Miller, P. Thomson, and T. Fogarty. Designing electronic circuits using evolutionary algorithms. arithmetic circuits: A case study. In *Genetic Algorithms and Evolution Strategies in Engineering and Computer Science*, pages 105–131. Wiley, 1997.

[36] Julian Miller. Designing multiple ANNs with evolutionary development: Activity dependence. In Wolfgang Banzhaf, Leonardo Trujillo, Stephan Winkler, and Bill Worzel, editors, *Genetic Programming Theory and Practice XVIII*, Genetic and Evolutionary Computation, East Lansing, USA, 19-21 May 2021. Springer. Forthcoming.

[37] Julian F. Miller. An empirical study of the efficiency of learning boolean functions using a cartesian genetic programming approach. In Wolfgang Banzhaf, Jason Daida, Agoston E. Eiben, Max H. Garzon, Vasant Honavar, Mark Jakiela, and Robert E. Smith, editors, *Proceedings of the Genetic and Evolutionary Computation Conference*, volume 2, pages 1135–1142, Orlando, Florida, USA, 13-17 July 1999. Morgan Kaufmann.

[38] Julian F. Miller, Dominic Job, and Vesselin K. Vassilev. *Genetic Programming and Evolvable Machines*, 1(1/2):7–35, 2000.

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

[40] Julian F. Miller, Dennis G. Wilson, and Sylvain Cussat-Blanc. Evolving developmental programs that build neural networks for solving multiple problems. In Wolfgang Banzhaf, Lee Spector, and Leigh Sheneman, editors, *Genetic Programming Theory and Practice XVI, [GPTP 2018, University of Michigan, Ann Arbor, USA, May 17-20, 2018]*, Genetic and Evolutionary Computation, pages 137–178. Springer, 2018.

[41] V. Mrazek, R. Hrbacek, et al. Evoapprox8b: Library of approximate adders and multipliers for circuit design and benchmarking of approximation methods. In *Proc. of DATE'17*, pages 258–261. EDAA, 2017.

[42] Peter Nordin. A compiling genetic programming system that directly manipulates the machine code. In *Advances in Genetic Programming*, chapter 14, pages 311–331. MIT Press, 1994.

[43] Peter Nordin. Aimgp: A formal description. In *Late Breaking Papers at the Genetic Programming 1998 Conference*, pages 169–175, University of Wisconsin, Madison, Wisconsin, USA, 1998. Stanford University Bookstore.

[44] Peter Nordin and Wolfgang Banzhaf. Complexity compression and evolution. In *Proceedings of the 6th International Conference on Genetic Algorithms*, page 310–317, San Francisco, CA, USA, 1995. Morgan Kaufmann Publishers Inc.

[45] Peter Nordin and Wolfgang Banzhaf. Evolving Turing-complete programs for a register machine with self-modifying code. In *Genetic Algorithms: Proceedings of the Sixth International Conference (ICGA95)*, pages 318–325, Pittsburgh, PA, USA, 1995. Morgan Kaufmann.

References V

[46] Peter Nordin, Wolfgang Banzhaf, and Frank D. Francone. *Efficient Evolution of Machine Code for CISC Architectures Using Instruction Blocks and Homologous Crossover*, page 275–299. MIT Press, Cambridge, MA, USA, 1999.

[47] Peter Nordin, Frank Francone, and Wolfgang Banzhaf. *Explicitly Defined Introns and Destructive Crossover in Genetic Programming*, page 111–134. MIT Press, Cambridge, MA, USA, 1996.

[48] Tomas Pecenka, Lukas Sekanina, and Zdenek Kotasek. Evolution of synthetic RTL benchmark circuits with predefined testability. *ACM Transactions on Design Automation of Electronic Systems*, 13(3):1–21, July 2008.

[49] Stjepan Picek, Claude Carlet, Sylvain Guilley, Julian F Miller, and Domagoj Jakobovic. Evolutionary algorithms for boolean functions in diverse domains of cryptography. *Evolutionary computation*, 24(4):667–694, 2016.

[50] Stjepan Picek, Domagoj Jakobovic, Julian F Miller, Lejla Batina, and Marko Cupic. Cryptographic boolean functions: One output, many design criteria. *Applied Soft Computing*, 40:635–653, 2016.

[51] Michael Defoin Platel, Manuel Clergue, and Philippe Collard. Maximum homologous crossover for linear genetic programming. In *Proceedings of the 6th European Conference on Genetic Programming*, EuroGP'03, page 194–203, Berlin, Heidelberg, 2003. Springer-Verlag.

[52] Detlef Plump. The design of GP 2. In Santiago Escobar, editor, *Proceedings 10th International Workshop on Reduction Strategies in Rewriting and Programming, WRS 2011, Novi Sad, Serbia, 29 May 2011*, volume 82 of *EPTCS*, pages 1–16, 2011.

[53] R. Poli. Some steps towards a form of parallel distributed genetic programming. In *The 1st Online Workshop on Soft Computing (WSC1)*, http://www.bioele.nuee.nagoya-u.ac.jp/wsc1/, 19–30 August 1996. Nagoya University, Japan.

[54] Riccardo Poli. Evolution of graph-like programs with parallel distributed genetic programming. In Thomas Back, editor, *Genetic Algorithms: Proceedings of the Seventh International Conference*, pages 346–353, Michigan State University, East Lansing, MI, USA, 19-23 July 1997. Morgan Kaufmann.

[55] Riccardo Poli. Parallel distributed genetic programming applied to the evolution of natural language recognisers. *Evolutionary Machine Learning and Classifier Systems*, pages 163–177, 1997.

[56] Riccardo Poli. *New Ideas in Optimization*, chapter Parallel Distributed Genetic Programming, pages 403–432. McGraw-Hill Ltd., UK, Maidenhead, UK, England, 1999.

[57] Lukáš Sekanina. Image filter design with evolvable hardware. In *Lecture Notes in Computer Science*, pages 255–266. Springer Berlin Heidelberg, 2002.

[58] Leo Francoso Dal Piccol Sotto, Vinicius Veloso de Melo, and Marcio Porto Basgalupp. λ-lgp: an improved version of linear genetic programming evaluated in the ant trail problem. *Knowledge and Information Systems*, 52(2):445–465, 2017.

References VI

[59] Leo Francoso Dal Piccol Sotto, Paul Kaufmann, Timothy Atkinson, Roman Kalkreuth, and Marcio Porto Basgalupp. Graph representations in genetic programming. *Genetic Programming and Evolvable Machines*, 22(4):607–636, 2021. Special Issue: Highlights of Genetic Programming 2020 Events.

[60] Leo Francoso Dal Piccol Sotto, Franz Rothlauf, Vinicius Veloso de Melo, and Marcio P. Basgalupp. An analysis of the influence of non-effective instructions in linear genetic programming. *Evolutionary Computation*, 30(1):51–74, 2022.

[61] Andrew James Turner and Julian Francis Miller. Cartesian genetic programming encoded artificial neural networks: a comparison using three benchmarks. In Christian Blum and Enrique Alba, editors, *Genetic and Evolutionary Computation Conference, GECCO '13, Amsterdam, The Netherlands, July 6-10, 2013*, pages 1005–1012. ACM, 2013.

[62] Andrew James Turner and Julian Francis Miller. The importance of topology evolution in neuroevolution: A case study using cartesian genetic programming of artificial neural networks. In Max Bramer and Miltos Petridis, editors, *Research and Development in Intelligent Systems XXX, Incorporating Applications and Innovations in Intelligent Systems XXI Proceedings of AI-2013, The Thirty-third SGAI International Conference on Innovative Techniques and Applications of Artificial Intelligence, Cambridge, England, UK, December 10-12, 2013*, pages 213–226. Springer, 2013.

[63] Andrew James Turner and Julian Francis Miller. Neuroevolution: Evolving heterogeneous artificial neural networks. *Evol. Intell.*, 7(3):135–154, 2014.

[64] Andrew James Turner and Julian Francis Miller. Neutral genetic drift: an investigation using cartesian genetic programming. *Genetic Programming and Evolvable Machines*, 16(4):531–558, 2015.

[65] Zdenek Vasicek. Cartesian gp in optimization of combinational circuits with hundreds of inputs and thousands of gates. In *EuroGP'15*, LCNS 9025, pages 139–150. Springer International Publishing, 2015.

[66] Zdenek Vasicek, Michal Bidlo, and Lukas Sekanina. Evolution of efficient real-time non-linear image filters for FPGAs. *Soft Computing*, 17(11):2163–2180, April 2013.

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

[68] Zdenek Vasicek and Lukas Sekanina. How to evolve complex combinational circuits from scratch? In *2014 IEEE International Conference on Evolvable Systems Proceedings*, pages 133–140. IEEE, 2014.

[69] Zdenek Vasicek and Lukas Sekanina. Evolutionary approach to approximate digital circuits design. *IEEE Transactions on Evolutionary Computation*, 19(3):432–444, June 2015.

[70] Vesselin K. Vassilev and Julian F. Miller. The advantages of landscape neutrality in digital circuit evolution. In Julian Miller, Adrian Thompson, Peter Thomson, and Terence C. Fogarty, editors, *Evolvable Systems: From Biology to Hardware*, pages 252–263, Berlin, Heidelberg, 2000. Springer Berlin Heidelberg.

[71] Tina Yu and Julian Miller. Neutrality and the evolvability of boolean function landscape. In Julian Miller, Marco Tomassini, Pier Luca Lanzi, Conor Ryan, Andrea G. B. Tettamanzi, and William B. Langdon, editors, *Genetic Programming*, pages 204–217, Berlin, Heidelberg, 2001. Springer Berlin Heidelberg.