Tag-based regulation of modules in genetic programming improves context-dependent problem solving

Alexander Lalejini · Matthew Andres Moreno · Charles Ofria

Abstract We introduce and experimentally demonstrate the utility of tag-based genetic regulation, a new genetic programming (GP) technique that allows programs to dynamically adjust which code modules to express. Tags are evolvable labels that provide a flexible mechanism for referencing code modules. Tag-based genetic regulation extends existing tag-based naming schemes to allow programs to "promote" and "repress" code modules in order to alter expression patterns. This extension allows evolution to structure a program as a gene regulatory network where modules are regulated based on instruction executions. We demonstrate the functionality of tag-based regulation on a range of program synthesis problems. We find that tag-based regulation improves problem-solving performance on context-dependent problems; that is, problems where programs must adjust how they respond to current inputs based on prior inputs. Indeed, the system could not evolve solutions to some context-dependent problems until regulation was added. Our implementation of tag-based genetic regulation is not universally beneficial, however. We identify scenarios where the correct response to a particular input never changes, rendering tag-based regulation an unneeded functionality that can sometimes impede adaptive evolution. Tag-based genetic regulation broadens our repertoire of techniques for evolving more dynamic genetic programs and can easily be incorporated into existing tag-enabled GP systems.

Keywords tag-based referencing · gene regulation · linear genetic programming · automatic program synthesis · SignalGP

Alexander Lalejini[1,2,3]
E-mail: amlalejini@gmail.com
Matthew Andres Moreno[1,2,3]
E-mail: mmore500@msu.edu
Charles Ofria[1,2,3]
E-mail: ofria@msu.edu
[1]BEACON Center for the Study of Evolution in Action, Michigan State University
[2]Department of Computer Science and Engineering, Michigan State University
[3]Ecology, Evolution, and Behavior Program, Michigan State University

1 Introduction

Genetic programming (GP) applies the natural principles of evolution to automatically synthesize programs rather than writing them by hand. Indeed, the promise of automating computer programming has motivated advances in GP since its early successes in the 1980s [11, 18, 33]. Just as human software developers have access to a dazzling array of programming languages, each specialized for solving different types of problems, GP features many ways to represent evolvable programs. Each representation features different programmatic elements that vary in their syntax, organization, interpretation, and evolution. These differences can dramatically influence the types of computer programs that can be evolved, and as such, influence a representation's problem-solving range [25, 77]. Here, we introduce and experimentally demonstrate tag-based module regulation for genetic programming, allowing us to more easily evolve programs capable of dynamically regulating responses to inputs over time.

Nearly all software applications are capable of conditionally responding to inputs. For example, each input button on a calculator triggers a different software response; or, in the Small or Large problem from the Helmuth and Spector's automatic program synthesis benchmark suite [22], programs must output different classifications ("small", "large", or "neither") depending on a numeric input value. Just like such conditional logic is inherent in any non-trivial software, so to is it ubiquitous in biological organisms where it is referred to as "plastic" behavior or "phenotypic plasticity."

Modular software design—that is, designs that promote the partitioning and reusability of functional units—is fundamental to good software development practices; this principle is all the more true in producing programs capable of complex "plasticity." By modularizing code (*e.g.*, into functions, classes, libraries, *etc.*), software developers can craft customized responses to inputs by composing relevant modules. These modules can each contain segments of code whose functionality would otherwise need to be reinvented for each response. Likewise, modularity appears to be critical in natural genomes [69] as well as artificial evolving systems [28]. Moreover, evidence in these evolving systems suggests that modularity can improve the capacity for effective plasticity to arise [17, 45].

Developing GP systems that facilitate the evolution of modular program architectures has long captured the attention of the genetic programming community. Koza introduced Automatically Defined Functions (ADFs) where callable functions can evolve as separate branches of GP syntax trees [34, 35]. Angeline and Pollack developed compression and expansion genetic operators to automatically modularize existing code into libraries of parameterized subroutines [4]. Since these foundational advances, significant efforts have been made to allow GP representations to incorporate internal modules (*e.g.*, [63, 53, 9, 70, 66, 64, 41]), to measure (and select for) modularity in evolving programs (*e.g.*, [36, 59, 60]), and to build "libraries" of reusable code modules accessible to evolving populations of programs (*e.g.*, [5, 31, 32, 58]).

These innovations have improved the ability of GP systems to link modules together to solve problems, thus improving their prospects as general-purpose tools for automatic program synthesis. In existing GP work, links between modules, however, are typically hard coded and static during program execution. Less is known for how to evolve programs that can adjust module associations on the fly. For

many types of problems, the appropriate set of modules to execute in response to a particular input changes over time. This requires programs to continuously adjust associations between inputs and modular responses based on context. For example, the computations that occur on a calculator after pressing the "equals" button are *context-dependent*; that is, they depend on the set of operators and operands (*i.e.*, inputs) previously provided. To achieve this design pattern, programs must internally track contextual information and typically regulate responses using explicit flow control directives (such as if-statements). Our goal is to evolve programs that dynamically regulate modules during execution to more effectively solve context-dependent problems. To reach this goal, we draw inspiration from gene regulatory networks (both natural and artificial) to augment how program modules are called in GP.

Here, we propose to facilitate dynamic module composition by introducing tag-based module regulation for genetic programming. We extend existing tag-based naming schemes to allow programs to dynamically adjust associations between references and code modules. We experimentally demonstrate our implementation of tag-based genetic regulation in the context of SignalGP [41]; however, our approach is immediately applicable to any existing tag-enabled GP system, such as tag-addressed Run Transferable Libraries [31] or PushGP [66]. We add "regulation" instructions to SignalGP that can adjust (*i.e.*, promote or repress) which code modules respond to input signals and internal calls. This extension allows evolution to structure a program as a gene regulatory network where genes are program modules and program instructions mediate regulation. We show that module regulation improves problem-solving performance on problems where responses to particular inputs change depending on prior context (*e.g.*, prior inputs). We also observe that our implementation of tag-based regulation can sometimes impede adaptive evolution when outputs are not context-dependent.

2 Specifying Modules with Tag-based Referencing

All programming representations that support modularizing code into functions or libraries define mechanisms for labeling and subsequently referencing modules. In traditional software development, programmers hand label modules and reference a particular module using its assigned label. Programmers must precisely name the module they intend to reference; imprecision typically results in incorrect outputs or a syntax error. This mechanism for referencing modules allows for an arbitrarily large space of possible module names and is intentionally brittle, ensuring programs are either interpreted by a computer exactly as written or not interpreted at all. Requiring genetic programming systems to adhere to these traditional approaches to module referencing is not ideal. Mutation operators must either ensure that mutated labels are syntactically valid, or else cope with an abundance of broken code. These choices result in either a search space that is overly constrained or one that is rugged and difficult to navigate [56].

Inspired by Holland's use of "tags" to facilitate binding and aggregation in complex adaptive systems [26, 27], Spector *et al.* generalized the use of tags to label and refer to program modules in GP [65, 66]. Tags are evolvable labels that can be mutated, and the similarity (or dissimilarity) between any two tags can be quantified. Tags are most commonly represented as floating point or integer

numeric values [31, 66] or as bit strings [41]. Like traditional naming schemes, tags can provide an arbitrarily large address space. Unlike traditional naming schemes, however, tags allow for *inexact* addressing. A referring tag targets the tagged entity (*e.g.*, a module) with the *closest matching* tag; this ensures that all possible tags are valid references. Further, mutations to tags do not necessarily invalidate existing references. For example, mutating a referring tag will have no phenotypic effect if those mutations do not change which target tag is matched. As such, mutating tag-based names is not necessarily catastrophic to program functionality, allowing the labeling and use of modularized code fragments to incrementally co-evolve [66].

Tag-based referencing has long been used to expand the capabilities of genetic programming systems. Keijzer *et al.* created run transferable libraries of tag-addressable functions using successful code segments evolved in previous GP runs [31, 32]. Evolving programs (represented as program trees) contained dynamically-linked nodes that used tag-based referencing to call library functions. These tag-addressed libraries were updated between runs and did not co-evolve with programs.

Spector *et al.* augmented PushGP with tag-based referencing, allowing tag-addressable code modules to evolve *within* a program [66]. Spector *et al.* found that tags provided a flexible mechanism for modularization that allowed tag-enabled programs to better scale with problem size. Additionally, Spector *et al.* expanded tag-based modules beyond PushGP, successfully applying the technique to tree-based GP [64].

Lalejini and Ofria further extended tag-based naming to linear GP. Their SignalGP system broadens the application of tags to facilitate the evolution of event-driven programs [41, 43]. In SignalGP, tagged modules are called internally or triggered in response to tagged events (*e.g.*, events generated by other agents or the environment). More recently, Lalejini and Ofria demonstrated the use of tags to label memory positions in GP, enabling programs to define and use evolvable variable names [42]. This tag-based memory implementation did not substantively affect problem-solving performance; however, tag-based addressing features a larger addressable memory space than more traditional register-based memory approaches in GP.

3 Tag-based Genetic Regulation

Here, we allow programs to use tag-based referencing to dynamically regulate module execution. To achieve this, we draw inspiration from both natural and artificial gene regulatory networks. We demonstrate that this approach promotes more effective solutions for context-dependent problems.

Gene regulatory networks represent the complex interactions among genes, transcription factors, and signals from the environment that, together, control gene expression [7]. Gene regulation allows for feedback loops so that prior events can continue to influence future expression in flexible and nuanced ways. Gene regulation underlies most important biological processes, including cell differentiation, metabolism, the cell cycle, and signal transduction [30]. The role of gene regulatory networks in sustaining complex life has inspired varied and abundant computational models of these networks [14, 30].

Artificial gene regulatory networks have been used to study how natural gene regulation evolves [1, 12, 16] and as a tool in evolutionary computation to solve challenging control problems (as reviewed by [14]). Evolved artificial gene regulatory networks have even been used as indirect encoders, providing a developmental phase to translate genomes into programs [6, 49] or neural networks [78]. La Cava *et al.* demonstrated a form of *epigenetic* regulation for genetic programming where "gene" activation and silencing is learned each generation [37, 38]; however, the programs themselves did not have direct control over these regulatory elements. Inspired by chromatin remodeling in biological cells, Turner *et al.* introduced artificial epigenetic networks that allow for the regulation (*i.e.*, the addition or removal) of internal network components [67]; such topological self-modification improved problem-solving success for dynamical control problems.

We aim to incorporate gene regulatory network-inspired methodology to allow programs to dynamically adjust which module is triggered by a particular call based on not just current inputs, but also prior inputs. We achieved this goal by instantiating gene regulatory networks using tag-based referencing. Specifically, we implemented tag-based genetic regulation in the context of the linear GP system SignalGP [41], which is described in further detail in Section 4.1. Here, we describe tag-based genetic regulation in terms of our SignalGP-based implementation; however, our overall approach is immediately applicable to each of the tag-enabled systems described in Section 2 and can be easily incorporated into any genetic programming representation.

Briefly, programs in SignalGP are composed of tag-addressed modules (*i.e.*, functions), each of which contain a linear sequence of instructions. Each instruction has arguments, including an evolvable tag that can be used to identify and call a tag-addressed module. When a referring tag (*e.g.*, from an instruction) is used to look up a tag-addressed module, all modules in that program are ranked according to a tag-matching score. A tag-matching score quantifies the quality of the reference between a referring tag and a module's tag; we always select the module with best reference quality (*i.e.*, the highest tag-match score with the referring tag). When a module is called, it is executed procedurally, instruction-by-instruction, in the same way as in a conventional linear GP system.

We modified SignalGP in two ways to implement tag-based genetic regulation:

1. We added a "regulatory modifier" value (represented as a floating point value) to all tag-addressed modules. A module's regulatory modifier adjusts how well that module will match to referring tags, and thus, modifies the likelihood it will be referenced.
2. We supplemented the instruction set with promoter and repressor instructions that, when executed, adjust a target module's regulatory modifier.

When a program begins execution, each internal module initially has no regulatory modification.[1] When a promoter or repressor instruction is executed, its associated tag identifies which module should be regulated using tag-based referencing. Promoter instructions increase a target module's regulatory modifier, which increases the module's tag-match score with subsequent references (according to equation 1 below) and thus increases the module's chances of being referenced. Repressor instructions have the opposite effect. Regulatory modifiers can

[1] Alternatively, allowing programs to inherit their parent's regulatory modifiers can provide a simple model of epigenetics.

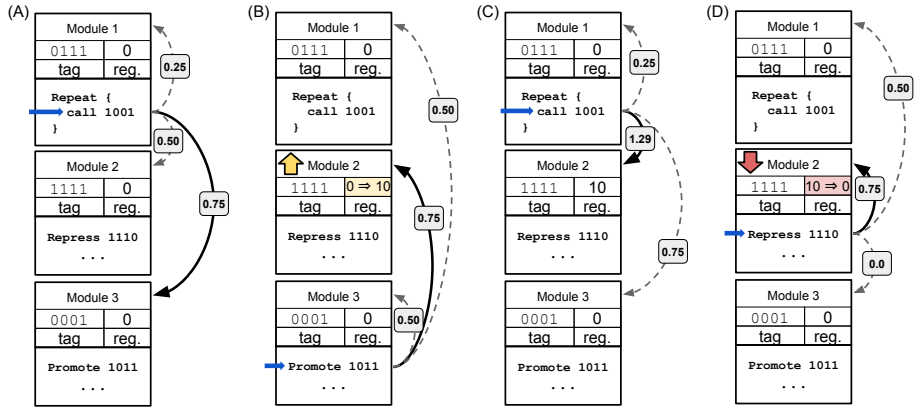


Fig. 1: **Tag-based genetic regulation example.** This example depicts a simple oscillating regulatory network instantiated using tag-based regulation. In this example, tags are length-4 bit strings. The "raw" match score between two tags equals the number of matching bits between them. Regulation (reg.) modifies match scores for "`call`" instructions according to Equation 1. First (A), the `call 1001` in Module 1 executes, triggering Module 3. Next (B), Module 3 is executed, promoting Module 2. After Module 3 returns, the `call 1001` in Module 1 executes again (C); however, Module 2's promotion causes it to be triggered instead of Module 3. Finally (D), Module 2 executes and represses itself, resetting its regulatory modifier to 0.

be configured to persist over a program's entire execution or passively decay over time.

When determining which module to call at runtime, each module's tag-match score is a function of how well the module's tag matches the call instruction's tag as modified by the module's regulatory value. If a module's regulatory modifier has been sufficiently decreased by repressor instructions, it is possible that the module will no longer be able to be referenced, as its regulated tag-match score will always be lower than at least one other program module. We must ensure that this situation does not create an unrecoverable regulatory state and that such a fully repressed module can always be restored. As such, promoter and repressor instructions use *unregulated* tag-based referencing to identify which modules they regulate; that is, we do not apply regulatory modifiers to tag-based references made by promoter and repressor instructions. This ensures that no matter how much a particular module has been repressed, subsequent promoter instructions can increase its regulatory modifier. Figure 1 gives a simplified example of how promoter and repressor instructions can dynamically adjust module execution over time.

We have implemented a toolbox of interchangeable methods for applying regulation to tag-matching scores in the Empirical library [52]. Here, we use a simple exponential function to apply a module's regulation modifier to its tag-match score calculations:

$$M_r(t_q, t_m, R_m) = M(t_q, t_m) \times b^{R_m} \tag{1}$$

R_m specifies the module's regulation modifier, which is under the direct control of the evolving programs. M_r is the regulation-adjusted match score between a querying tag (t_q) and the module's tag (t_m). M is a function that gives the baseline,

unadjusted match score between the querying tag and module tag. If tags are represented as floating point values, M can be as simple as the absolute difference between the two tags. The strength of regulation is determined by the constant, b (set to 1.1 in this work).

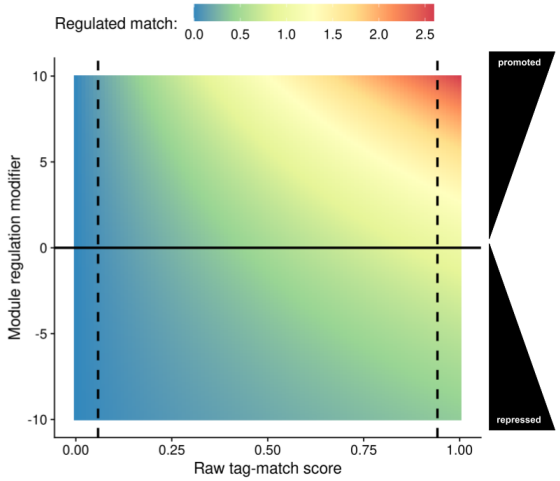


Fig. 2: **Regulated tag-match score as a function of raw tag-match score and regulatory modifier values according to Equation 1.** The horizontal black line indicates a neutral regulatory state; repressed states are below the line, and promoted states are above the line. We expect the raw tag-match score (calculated using the Streak similarity metric, which is described later in Section 4.1) of 90% of random pairs of tags to fall between the two dashed vertical lines; to compute the location of these lines, we generated 10^5 pairs of random tags and found the region that contained the middle 90% of raw tag-matching scores.

When determining which module to reference, each candidate module's M_r is computed, and the module with the highest M_r value is chosen. Intuitively, modules with $R_m < 0$ are down-regulated (*i.e.*, in a repressed state), modules with $R_m > 0$ are up-regulated (*i.e.*, in a promoted state), and modules with $R_m = 0$ are unmodified by regulation. That is, down-regulated modules have lower tag-match scores than they otherwise would without regulation, and up-regulated modules have higher tag-match scores than they otherwise would without regulation. Figure 2 gives a visual representation of Equation 1.

In preliminary experiments, we tested several different methods of implementing regulation (including additive, multiplicative, and the current exponential techniques). We found no evidence for any one method performing substantially better than the others. Future work will more thoroughly explore the potential effects of different regulation mechanisms.

4 Methods

We evaluated how tag-based genetic regulation faculties contribute to, and potentially detract from, the functionality of evolved genetic programs in the context of

SignalGP. First, we assessed the evolvability of our implementation of tag-based genetic regulation: can we evolve programs that rely on regulation to dynamically adjust their response to environmental conditions over time? Additionally, can tag-based genetic regulation improve problem-solving success on context-dependent problems? We addressed these questions using the signal-counting and contextual-signal problems, diagnostic tasks that require context-dependent responses to an input signal.

Next, we assessed tag-based genetic regulation on the Boolean-logic calculator problem, a more challenging program synthesis problem that requires programs to perform Boolean logic computations in response to a sequence of input events that represent button presses on a simple calculator.

Finally, we used the independent-signal problem to investigate the potential for genetic regulation to impede adaptive evolution by producing maladaptive plasticity. The independent-signal problem is a diagnostic that requires programs to associate distinct responses with each type of input; as such, programs do not need to change their response to particular input signals based on prior context. Additionally, fitness evaluation in the independent-signal problem is imperfect: programs receive input signals in a random order, providing ample opportunity for erroneous regulation to impede adaptive evolution.

4.1 SignalGP

Here, we provide a general overview of SignalGP; see [41] for a more in-depth description. SignalGP defines a scheme for organizing and interpreting genetic programs to afford computational evolution access to the event-driven programming paradigm [10]. In event-driven programs, software execution focuses on processing events (often in the form of messages from other processes, sensor alerts, or user actions). In SignalGP, events (signals) trigger the execution of program modules (functions), facilitating efficient reactions to exogeneously- or endogeneously-generated signals. For this work, program modules are represented as sequences of instructions; however, the SignalGP framework generalizes across a variety of program representations [43].

Programs in SignalGP are explicitly modular, comprising a set of functions, each associating a tag with an instruction sequence. SignalGP makes explicit the concept of events or *signals*. All signals contain a tag and any associated signal-specific data (*e.g.*, numeric input values). Because both signals and program functions are tagged, SignalGP determines the most appropriate function to process a signal using tag-based referencing: signals trigger the function with the closest matching tag.

In this work, we represent tags as 256-bit strings, and we quantify the similarity between any two tags using the Streak metric. The Streak metric was originally proposed by Downing [15] and measures similarity between two bit strings in terms of the relationship between the lengths of the longest contiguously-matching and longest contiguously-mismatching substrings.[2] Specifically, we XOR the two bit strings and count the longest substring of all 0's in the first case or of all 1's in

[2] We make a slight modification to Downing's matching procedure due to an error in its mathematical derivation, as detailed in the supplement [40].

the second. The equation below overviews how the Streak metric computes the similarity (S) between two tags (t_q and t_m):

$$S(t_q, t_m) = \frac{p_{\text{mismatch}}(t_q, t_m)}{p_{\text{mismatch}}(t_q, t_m) + p_{\text{match}}(t_q, t_m)}$$

where p_{match} returns the probability of observing the measured length of the longest contiguously-matching substring between t_q and t_m by chance, and p_{mismatch} returns the probability of observing the measured length of the longest contiguously-mismatching substring between t_q and t_m by chance. Both our implementation and the mathematical equations for computing the Streak similarity between two bit strings can be found in supplemental material Section 5 [40].

When a signal triggers a function, the function executes with the signal's associated data as input. SignalGP programs can handle many signals simultaneously by processing and responding to each in parallel threads of execution. Threads each contain local memory registers for performing computations. Additionally, concurrently executing threads may interact by writing to and reading from a shared global memory buffer. For this work, we guaranteed deterministic thread execution using a round robin scheduler to step each thread forward one step (*i.e.*, one instruction) synchronously.

The SignalGP instruction set allows programs to generate internal signals, broadcast external signals, and otherwise work in a tag-based context. In this work, each instruction contains one tag and three integer arguments. Arguments may modify the effect of an instruction, often specifying memory locations or fixed values. For example, instructions may refer to and call internal program modules using tag-based referencing; when an instruction generates a signal (*e.g.*, to be used internally or broadcast), the instruction's tag is used as the signal's tag.

Previous work has demonstrated that SignalGP facilitates the evolution of event-driven programs capable of identifying and responding to many *distinct* signals [43]. However, without access to regulation, SignalGP requires programs to track context in memory and use procedural mechanisms (*e.g.*, if statements) to adjust how they respond to a particular signal over time based on stored context. Here, we apply tag-based genetic regulation to SignalGP (as described in Section 3). We supplemented the instruction set with regulatory instructions (Table 1) that use tag-based referencing to target internal functions. In this work, we apply regulation to function references using Equation 1. Our full instruction set, including descriptions of each instruction, can be found in our supplemental material [40].

4.1.1 Evolution

In this work, we propagated programs asexually, and we applied mutations to offspring. The parent-selection method varied across experiments. Programs were variable-length: each program contained up to 256 modules, and each module contained up to 128 instructions.

We applied single-instruction substitution, insertion, and deletion mutations each at a per-instruction rate of 0.001. Additionally, we applied a 'slip' mutation operator [44] that could duplicate or delete entire sequences of instructions at a per-module rate of 0.05. We mutated numeric instruction arguments at a per-argument rate of 0.001, and we limited numeric arguments to values between -4 and

Instruction	Description
`SetRegulator+`	Set the regulatory modifier of a target module to the value stored in an argument-specified memory register.
`SetRegulator-`	Set the regulatory modifier of a target module to the negation of the value stored in an argument-specified memory register.
`SetOwnRegulator+`	Set the regulatory modifier of the currently executing module to the value stored in an argument-specified memory register.
`SetOwnRegulator-`	Set the regulatory modifier of the currently executing module to the negation of the value stored in an argument-specified memory register.
`AdjRegulator+`	Add the value stored in an argument-specified memory register to the regulatory modifier of a target module.
`AdjRegulator-`	Subtract the value stored in an argument-specified memory register to the regulatory modifier of a target module.
`AdjOwnRegulator+`	Add the value stored in an argument-specified memory register to the regulatory modifier of the currently executing module.
`AdjOwnRegulator-`	Subtract the value stored in an argument-specified memory-register to the regulatory modifier of the currently executing module.
`ClearRegulator`	Reset the regulatory modifier of a target module.
`ClearOwnRegulator`	Reset the regulatory modifier of the currently executing module.
`SenseRegulator`	Load the value of a target module's regulatory modifier into an argument-specified memory register.
`SenseOwnRegulator`	Load the value of the currently executing module's regulatory modifier into an argument-specified memory register.
`IncRegulator`	Add one to the regulatory modifier of a target module.
`IncOwnRegulator`	Add one to the regulatory modifier of the currently executing module.
`DecRegulator`	Subtract one from the regulatory modifier of a target module.
`DecOwnRegulator`	Subtract one from the regulatory modifier of the currently executing module.

Table 1: **Regulatory instructions used in this work.** We include (+) and (-) instruction variants to ensure that positive and negative regulation values are equally probable.

4. When a numeric argument mutated, we randomized the argument's value to a valid integer between -4 and 4. We mutated instruction- and module-tags at a per-bit rate of 0.0001. We applied whole-module duplication and deletion operators at a per-module rate of 0.05, allowing the number of modules in a program to evolve.

4.2 Signal-counting Problem

The signal-counting problem requires programs to continually change their response to an environmental signal, producing the appropriate output each of the K times that signal is repeated. Programs output responses by executing one of K response instructions. For example, if a program receives two signals from the environment during evaluation (*i.e.*, $K = 2$), the program should execute `Response-1` after the first signal and `Response-2` after the second signal; aside from executing the correct response instruction, no other output is necessary after receiving an environmental signal.

We provide programs 128 time steps to respond to each environmental signal. During each time step, each of a program's active threads execute a single instruction. Once the allotted time expires or the program outputs a response, the program's threads of execution reset, resulting in a loss of all thread-local memory; *only* the contents of the global memory buffer and each program module's regulatory state persist. The environment then produces the next signal (identical to each previous environmental signal) to which the program may respond. A program *must* use the global memory buffer or genetic regulation to correctly shift its response to each subsequent environmental signal. Evaluation continues in this way until the program correctly responds to each of the K environmental signals or until the program executes an incorrect response. A program's fitness equals the number of consecutive correct responses given during evaluation, and a program is considered a solution if it correctly responds to all K environmental signals.

4.2.1 Experimental Design

The signal-counting problem is explicitly designed to (1) evaluate if tag-based genetic regulation can be evolved to dynamically adjust which modules execute in response to a *repeated* input type and (2) assess the problem-solving success of a regulation-enabled GP system relative to an otherwise identical GP system with regulation disabled. We compared programs evolved in a regulation-on treatment to those evolved in a regulation-off control. In the control treatment, we used an identical instruction set where regulation instructions were altered to behave as no-operation instructions. As such, programs must use global memory (in combination with procedural flow-control mechanisms) to correctly respond to environmental signals.

For each experimental condition, we evolved 200 replicate populations of 1000 programs for 10,000 generations at four levels of problem difficulty: $K = 2, 4, 8$, and 16. For each replicate, we randomly generated a unique tag for each environmental signal, and we initialized populations with randomly generated programs. Each generation, we evaluated programs independently, and we selected programs using size-eight tournament selection.

4.3 Contextual-signal Problem

The contextual-signal problem is inspired by Skocelas and DeVries' method for verifying the functionality of recurrent neural network implementations [61]. In

the contextual-signal problem, programs must respond appropriately to a pair of input signals. The order of these signals does not matter, but the first signal must be remembered (as "context") in order to produce the correct response to the second signal. In this work, there are a total of four possible input signals and four possible outputs. Programs output a particular response by executing one of four response instructions. Table 2 gives the correct output type for each pairing of input signals.

Test case ID	Input Sequence	Correct Response
0	S-0, S-0	Response-A
1	S-0, S-1	Response-B
2	S-0, S-2	Response-C
3	S-0, S-3	Response-D
4	S-1, S-0	Response-B
5	S-1, S-1	Response-C
6	S-1, S-2	Response-D
7	S-1, S-3	Response-A
8	S-2, S-0	Response-C
9	S-2, S-1	Response-D
10	S-2, S-2	Response-A
11	S-2, S-3	Response-B
12	S-3, S-0	Response-D
13	S-3, S-1	Response-A
14	S-3, S-2	Response-B
15	S-3, S-3	Response-C

Table 2: **Input signal sequences for the contextual-signal problem.**

We evaluate programs on each of the 16 possible sequences of input signals (Table 2); we consider each of these input sequences as a single test case. For each test case evaluation, we give programs 128 time steps to process each signal. After the first input signal, a program must update internal state information to ensure that the second input signal induces the correct response. Once the allotted time expires after the first input signal, the program's threads of execution are reset, resulting in a loss of all thread-local memory; *only* the contents of global memory and each function's regulatory state persist. The program then receives the second input signal and must execute the correct response instruction within 128 time steps. A program is considered a solution if it produces the correct response for all 16 possible sequences of input signals.

4.3.1 Experimental Design

We use the contextual-signal problem to (1) assess the capacity of tag-based genetic regulation to perform context-dependent module execution based on *distinct* input types and (2) evaluate the problem-solving success of a regulation-enabled GP system relative to an otherwise identical GP system with regulation disabled. As in the signal-counting problem, we compared the problem-solving success of regulation-on and regulation-off GP systems.

For each experimental condition, we evolved 200 replicate populations of 1000 programs for 10,000 generations. For each replicate, we randomly generated the

tags associated with each type of input signal, and we initialized populations with randomly generated programs. Instead of selecting programs to propagate based on an aggregate fitness measure, we used the lexicase parent selection algorithm [23] in which each combination of input signals (*i.e.*, row in Table 2) constituted a single test case.

4.4 Boolean-logic Calculator Problem

Inspired by Yeboah-Antwi's PushCalc system [80], the Boolean-logic calculator problem requires programs to implement a push-button calculator capable of performing each of the following 10 bitwise logic operations: ECHO, NOT, NAND, AND, OR-NOT, OR, AND-NOT, NOR, XOR, and EQUALS. Table 3 gives a brief overview of each of these operations. In this problem, there are 11 distinct types of input signals: one for each of the 10 possible operators and one for numeric inputs. Each distinct signal type is associated with a unique tag (randomly generated per-replicate) and is meant to recreate the context that must be maintained on a physical calculator. Programs receive a sequence of input signals in prefix notation, starting with an operator signal and followed by the appropriate number of numeric input signals (that each contain an operand to use in the computation). After receiving the appropriate input signals, programs must output the correct result of the requested computation.

Operation	# Inputs	NAND gates
ECHO	1	0
NOT	1	1
NAND	2	1
AND	2	2
OR-NOT	2	2
OR	2	3
AND-NOT	2	3
NOR	2	4
XOR	2	4
EQUALS	2	5

Table 3: **Bitwise Boolean logic operations used in the Boolean-logic calculator problem.** Programs are given a `nand` instruction and must construct each of the other operations (aside from ECHO) out of `nand` operations. As such, we measure the difficulty of each operation as the minimum number of NAND gates required to construct the given operation.

Programs are evaluated on a set of test cases (*i.e.*, input/output examples) where each test case comprises a particular operator, the requisite number of operands, and the expected numeric output. Test cases are evaluated on a pass/fail basis, and a program is classified as a solution if it passes all test cases in a training and testing set[3]. The training and testing sets used in this work are included in our supplemental material [40] and contained 442 and 5810 test cases, respectively. Each generation, we sample 20 test cases from the training set, and

[3] We use the testing set only to determine if a program can be categorized as a solution. The testing set is never used by the parent-selection algorithm to determine reproductive success.

we independently evaluate each program in the population on the sampled test cases.

When evaluating a program on a test case, we provide 128 time steps to process each input signal. After time expires, the program's threads of execution are reset, resulting in a loss of all thread-local memory; only the contents of global memory and each function's regulatory state persist. Because input signals are given in prefix notation, programs must adjust their internal state to ensure that the program performs and outputs the result of the appropriate computation after receiving the requisite number of operand input signals.

4.4.1 Experimental Design

We use the Boolean-logic calculator problem to assess the utility of tag-based regulation on a challenging program synthesis problem. The signal-counting and contextual-signal problems each require programs to perform different computations in response to input signals, but those computations are abstracted as 'response' instructions. The Boolean-logic calculator problem requires programs to both dynamically adjust which modules are executed in response to input signals and perform non-trivial computations on numeric inputs.

We compared the problem-solving success of programs evolved in regulation-on and regulation-off conditions. For each condition, we evolved 200 replicate populations of 1000 programs for 10,000 generations. For each replicate, we randomly generated the tags associated with each type of input signal, and we initialized populations with randomly generated programs. We selected parents using a variant of the down-sampled lexicase algorithm [24], guaranteeing that at least one of each type of test case (*i.e.*, at least one of each type of operator) was used during evaluation.

4.5 Independent-signal Problem

The independent-signal problem requires programs to execute a unique response for each of 16 distinct input signals. Because signals are distinct, programs need not alter their response to any particular signal over time. Instead, programs may "hardwire" each of the 16 possible responses to the appropriate input signal. However, input signals are presented in a random order; thus, the correct *order* of responses cannot be hardcoded. Otherwise, evaluation (and fitness assignment) on the independent-signal task mirrors that of the signal-counting task (Section 4.2). A program is considered a solution if it responds correctly to all 16 input signals during evaluation.

4.5.1 Experimental Design

We deliberately configured fitness evaluation and solution identification in the independent-signal problem to be noisy and thus unreliable: each program is evaluated once on a single random ordering of input signals, and we label a program as a solution if it performs optimally during a single evaluation. Because programs receive input signals in a random order, erroneous genetic regulation can manifest as cryptic variation (*i.e.*, behavioral variation that is not expressed and selected

on). For example, non-adaptive down-regulation of a particular response function may be neutral given one sequence of input signals, but may be deleterious in another. Indeed, this form of non-adaptive cryptic variation can also result from erroneous flow control structures.

The independent-signal problem allows us to test whether genetic regulation can impede adaptive evolution in scenarios where outputs are not context-dependent and where fitness evaluation does not reliably differentiate between generalizing and non-generalizing candidate solutions. Fitness evaluation for the independent-signal problem is computationally inexpensive, so we could easily increase the reliability of evaluation by testing programs on multiple orderings of input sequences. However, our goal is not to demonstrate that we can *solve* this diagnostic problem. Rather, we aim to determine if this diagnostic represents a general scenario where unnecessary tag-based regulation can impede adaptive evolution relative to not having regulation.

As in each of the previous experiments, we compared programs evolved in regulation-on and regulation-off conditions. Specifically, we compared initial problem-solving success and how well solutions generalized to a sample of 5000 input sequences (of $\sim 2.1 \times 10^{13}$ possible sequences). We deemed programs as having generalized only if they responded correctly in all 5000 tests.

We evolved 200 replicate populations of 1000 programs for 10,000 generations under each condition. For each replicate, we randomly generated 16 unique input signal tags. All other experimental procedures were identical to that of the signal-counting task.

4.6 Data Analysis and Reproducibility

For each replicate in a given experiment, we extracted and analyzed the first evolved program that was classified as a solution. We compared the number of successful replicates (*i.e.*, replicates that yielded a solution) across experimental conditions using Fisher's exact test. We conducted knockout experiments on successful programs to identify the mechanisms underlying their behavior. In all knockout experiments, we re-evaluated programs with a target functionality (*e.g.*, regulation instructions) replaced with no-operation instructions. Specifically, we independently knocked out (1) all regulatory instructions, (2) all instructions that access a program's global memory buffer, and (3) both regulatory instructions and global memory access instructions. We classify a program as reliant on a particular functionality if, when knocked out, fitness decreases. In addition to knockout experiments, we tracked the distribution of instruction types (*e.g.*, flow control, mathematical operations, *etc.*) executed by successful programs. For each successful replicate, we extracted the proportion of flow control instructions (*i.e.*, conditional logic instructions such as "if" or "while" statements) executed by the evolved solution. We compared the proportions of flow control instructions executed by regulation-on solutions and regulation-off solutions, allowing us to assess the relative importance of conditional logic across experimental treatments.

For programs reliant on genetic regulation, we abstracted regulatory networks as directed graphs by monitoring program execution. Vertices represent program functions, and directed edges (each categorized as promoting or repressing) show the regulatory interactions between two functions. For example, a repressing edge

from function A to function B indicates that B was repressed when A was executing.

We implemented our experiments using the Empirical scientific software library [52], and we conducted all statistical analyses using R version 4.0.2 [55]. We used the reshape2 [72] R package and the tidyverse [73] collection of R packages to wrangle data. We used the following R packages for graphing and visualization: ggplot2 [74], cowplot [75], viridis [19], Color Brewer [21, 51], and igraph [13]. We used R markdown [2] and bookdown [79] to generate web-enabled supplemental material. Our source code for experiments and analyses, along with guides for replication, can be found in supplemental material [40], which is hosted on GitHub. Additionally, we have made all of our experimental data available on the Open Science Framework (see Section 2 in supplement [40]).

5 Results and Discussion

5.1 Tag-based regulation improves problem-solving performance on context-dependent tasks

We found that tag-based regulation improves performance on each of the three problems that require context-dependent behavior: the signal-counting problem (Section 5.1.1), contextual-signal problem (Section 5.1.2), and Boolean-logic calculator problem (Section 5.1.3). Additionally, we conducted knockout experiments that confirmed that evolved tag-based regulation allows solutions to dynamically adjust module execution over time. We also found that, across all three context-dependent problems, regulation-off solutions (*i.e.*, solutions evolved using regulation-disabled SignalGP) executed a larger proportion of conditional logic instructions than regulation-on solutions (*i.e.*, solutions evolved using regulation-enabled SignalGP). This result suggests that without regulation, programs must evolve larger, more complex conditional logic structures.

5.1.1 Signal-counting Problem

	Regulation-off condition	Regulation-on condition
Two-signal	137	200
Four-signal	8	200
Eight-signal	0	198
Sixteen-signal	0	74

Table 4: **Signal-counting problem-solving success.** This table gives the number of successful replicates (*i.e.*, in which a perfect solution evolved) out of 200 on the signal-counting problem across four problem difficulties and two experimental conditions. For each problem difficulty, the regulation-off condition was less successful than the regulation-on condition (Fisher's exact test; all difficulties: $p < 10^{-15}$).

Table 4 shows the results from the signal-counting problem for each experimental condition across all four levels of problem difficulty. Regulation-on conditions

consistently yielded a larger number of successful replicates than regulation-off conditions where programs relied on their global memory buffer in combination with procedural flow control for success. Although global memory is technically sufficient to solve each version of the signal-counting problem[4], in practice such solutions evolved in only the two- and four-signal variants. Tag-based regulation, in contrast, appears more readily adaptive, as regulation-based solutions arose across all problem difficulties, implying that access to tag-based regulation can drive increased problem-solving success. Further, we found that, in the two- and four-signal tasks, solutions arose after significantly fewer generations in the regulation-on conditions than in the regulation-off controls (Figure 3).

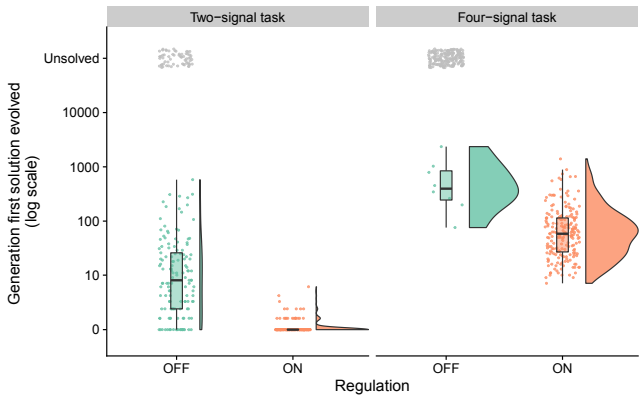


Fig. 3: **Generation at which first solution evolved (log scale) in each successful replicate for the signal-counting problem (Raincloud plot [3]).** We show data from only those problem difficulties in which solutions evolved (two- and four-signal problems). Gray points indicate the number of unsuccessful replicates for each condition. For both problem difficulties, regulation-on solutions typically required fewer generations than regulation-off solutions to arise (Wilcoxon rank sum test; two-signal: $p < 10^{-15}$, four-signal: $p < 9 \times 10^{-05}$).

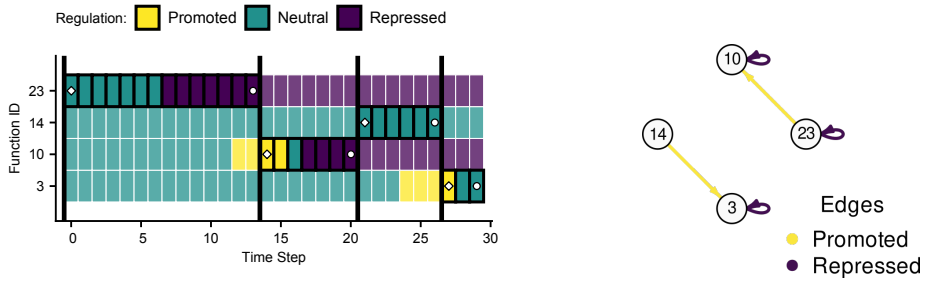
Tag-based regulation renders the two-signal task trivial: all solutions evolved in under 10 generations. In fact, the majority of regulation-on solutions (178 out of 200) were found in the initial randomly generated population. However, not all replicates without access to tag-based regulation even found a solution to the two-signal task.

We conducted knockout experiments to investigate the mechanisms underlying successful programs. Indeed, all solutions evolved without access to tag-based regulation relied exclusively on their global memory buffer to differentiate their behavior (see supplemental Section 7 [40]). Table 5 shows the strategies used by programs evolved with regulation-enabled SignalGP. Our knockout experiments confirm that the majority of solutions evolved with access to tag-based regulation do indeed rely on regulation to dynamically adjust their responses to signals over time.

[4] We verified this claim by hand-coding solutions that rely on global memory and flow-control instructions (supplemental Section 12 [40]).

	No regulation required	Regulation required	Unsolved
Two-signal	11	189	0
Four-signal	0	200	0
Eight-signal	0	198	2
Sixteen-signal	0	74	126

Table 5: **Mechanisms underlying solutions from the regulation-on condition for the signal-counting problem.** To determine a successful program's underlying strategy, we re-evaluated the program with global memory access instructions knocked out (*i.e.*, replaced with no-operation instructions) and with regulation instructions knocked out. This table shows the number of regulation-on solutions that actually rely on regulation to solve the signal-counting problem.



(a) Module regulation over time.

(b) Regulatory network.

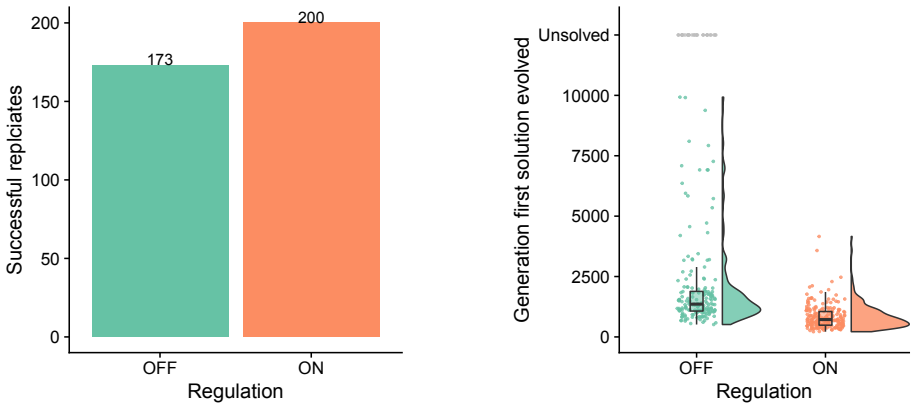
Fig. 4: **Execution trace of a SignalGP program solving the four-signal version of the signal-counting task.** Color denotes each function's regulatory state (yellow: promoted, purple: repressed) during evaluation; functions not regulated or executed are omitted. Functions that are actively executing are annotated with a black outline. Black vertical lines denote input signals, and a diamond (white with black outline) indicates which function was triggered by the input signal. A circle (white with black outline) indicates which function executed a response. (b) shows the directed graph representing the regulatory network associated with trace (a). Vertices depict functions that either ran during evaluation or were regulated. Each directed edge shows a regulatory relationship between two functions where the edge's source acted on (promoted in yellow or repressed in purple) the edge's destination. Note that in the case presented here all repressing relationships are self-referential.

We further assessed the functionality of tag-based regulation by analyzing the execution traces of evolved solutions. We visualized the gene regulatory networks that manifest as a result of programs executing promoter and repressor instructions. Figure 4 overviews the execution of a representative evolved program on the four-signal instance of the signal-counting problem. We found that successful programs tend to operate via a succession of self-repressing events where modules express the appropriate response then disable themselves so that the next best-matching function—expressing the appropriate next response—will activate instead. This behavioral pattern continues for each subsequent environmental signal. Indeed, across all problem difficulties, we observed that successful regulatory networks generally contained more repression relationships than promotion relationships between functions (supplemental Section 7 [40]). Independent knockouts of up-regulation and down-regulation confirm that the majority of successful regulatory networks rely on down-regulation: of the 661 successful regulatory networks

evolved across all problem difficulties, 392 rely exclusively on down-regulation, 7 rely exclusively on up-regulation, 259 rely on both up- and down-regulation, and 3 rely on *either* up- or down-regulation (*i.e.*, they required regulation but were robust to independent knockouts of up- and down-regulation).

Our experimental data highlights the benefit of tag-based genetic regulation in addition to traditional, register-based means of dynamically adjusting responses to a repeated input signal over time. However, our data may also indicate a deficiency in the design of SignalGP's current global memory model. An improved memory model may also enhance the capacity for programs to dynamically adjust their responses to inputs over time; however, any memory-based solution will still suffer from the need to incorporate flow-control structures to implement this functionality, inherently creating a larger evolutionary hurdle to overcome. Indeed, we found that the memory-based solutions that evolved in our experiments executed a larger proportion of flow-control instructions than regulation-based solutions (Wilcoxon rank sum test; two-signal: $p < 10^{-10}$, four-signal: $p = 0.004$; supplemental Section 7 [40]).

5.1.2 Contextual-signal Problem



(a) Successful replicates.

(b) Generations elapsed before solution.

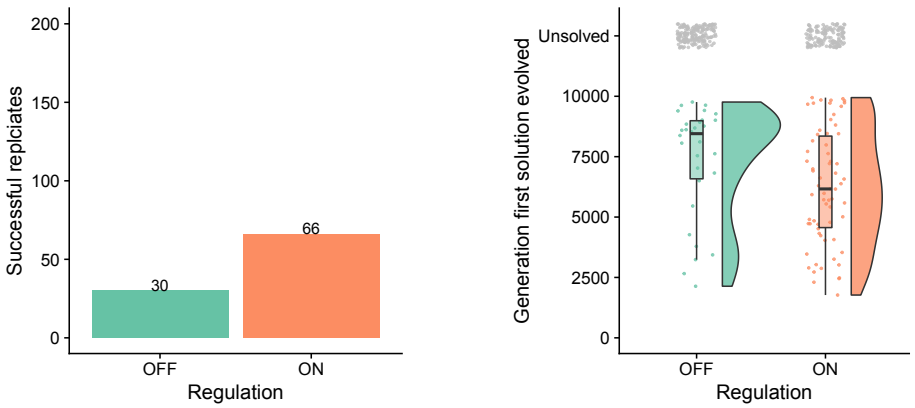
Fig. 5: **Contextual-signal problem-solving performance.** (a) shows the number of successful replicates for the regulation-off and regulation-on conditions on the contextual-signal problem. The regulation-off condition was less successful than the regulation-on condition (Fisher's exact test: $p < 6 \times 10^{-9}$). (b) is a Raincloud plot showing the generation at which the first solution evolved in each successful replicate. Gray points indicate the number of unsuccessful replicates for each condition. Regulation-on solutions typically required fewer generations than regulation-off solutions to arise (Wilcoxon rank sum test: $p < 10^{-15}$).

Figure 5a shows the number of successful replicates on the contextual-signal problem for both the regulation-on and regulation-off conditions. While both conditions were often successful, we found that access to tag-based regulation significantly improved problem-solving success. Further, regulation-on solutions typically required fewer generations to evolve than regulation-off solutions (Figure 5b).

We used knockout experiments to identify the mechanisms underlying each solution's strategy. As expected, all 173 solutions evolved without access to tag-based regulation relied on their global memory buffer to track contextual information and used control flow mechanisms to differentiate their responses based on stored context. Indeed, we found that regulation-off solutions executed a larger proportion of flow-control instructions than regulation-on solutions (Wilcoxon rank sum test: $p < 10^{-15}$; supplement Section 8 [40]). We also found that all 200 regulation-on solutions relied on tag-based regulation for response differentiation: 105 relied only on tag-based regulation and 95 relied on a combination of both tag-based regulation and global memory.

In contrast to the signal-counting problem, we did not find that successful regulatory networks used primarily self-repressing modules. Instead, we found that networks were more balanced between repressing and promoting edges; indeed, we found that successful networks generally contained more promoting edges than repressing edges (supplement Section 8 [40]). This result suggests that we should expect different problems to select for different forms of gene regulatory networks.

5.1.3 Boolean-logic Calculator Problem



(a) Successful replicates.

(b) Generations elapsed before solution.

Fig. 6: **Boolean-logic calculator problem-solving performance.** (a) shows the number of successful replicates for the regulation-off and regulation-on conditions on the Boolean-logic calculator problem. The regulation-off condition was less successful than the regulation-on condition (Fisher's exact test: $p < 4 \times 10^{-05}$). (b) is a Raincloud plot showing the generation at which the first solution evolved in each successful replicate. Gray points indicate the number of unsuccessful replicates for each condition. Regulation-on solutions typically required fewer generations than regulation-off solutions to arise (Wilcoxon rank sum test: $p < 0.042$).

Figure 6a shows the number of successful replicates on the Boolean-logic calculator problem for both the regulation-on and regulation-off conditions. While both regulation-on and regulation-off solutions evolved, we again found that access to genetic regulation significantly improved problem-solving success. Further, as in

the signal-counting and contextual-signal problems, regulation-on solutions typically required fewer generations to evolve than regulation-off solutions (Figure 6b).

As in previous experiments, we conducted knockout experiments to identify the mechanisms underlying each solution's strategy. To compute any of the Boolean logic operations, programs *must* make use of the global memory buffer to store numeric inputs (operands) to be used when performing the computation specified by the final operator signal. Indeed, all solutions evolved across all conditions relied on their global memory buffer to solve this problem. All 66 regulation-on solutions, however, also relied on tag-based regulation to perform the appropriate computation for each test case. Consistent with results from each other context-dependent problem, we found that regulation-off solutions executed a larger proportion of flow-control instructions than regulation-on solutions (Wilcoxon rank sum test: $p < 2 \times 10^{-05}$; supplement Section 9 [40]).

As in the signal-counting problem, we visualized the gene regulatory networks that manifest as a result of programs executing promoting and repressing instructions. Figure 7 overviews the execution of a representative program evolved to solve the Boolean-logic calculator problem. Specifically, Figure 7 shows a program computing NAND and the same program computing NOR. The networks expressed on each of these operations are distinct despite originating from the same code. These visualizations confirm that tag-based regulation allows programs to dynamically adjust their responses based on context (in this case, an initial operator signal).

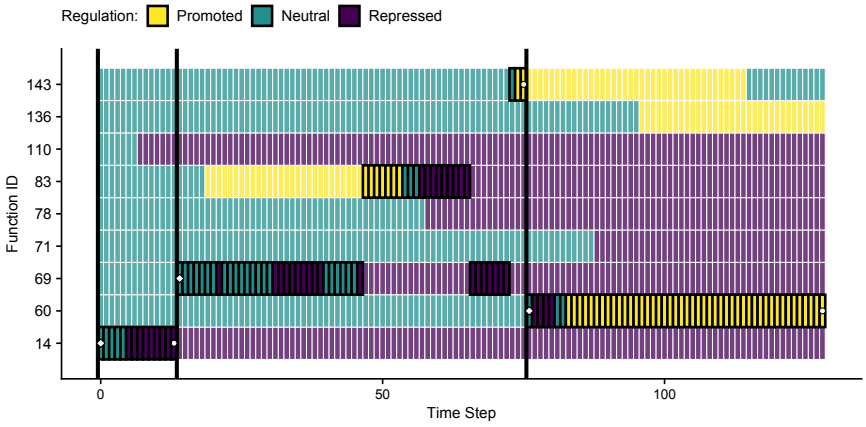
5.2 Erroneous regulation can hinder task generalization

In the signal-counting, contextual-signal, and Boolean-logic calculator problems, programs must adjust their behavior depending on the particular sequence of received signals. The independent-signal problem, however, requires no signal-response plasticity; programs maximize fitness by statically associating K distinct responses each with one of K distinct input signals. For this task, re-wiring signal-response associations within-lifetime is maladaptive. As such, does the capacity for regulation impede adaptation to the independent-signal task?

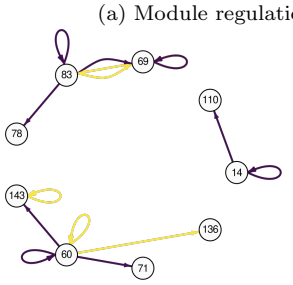
We compared 200 replicate populations evolved with regulation-enabled SignalGP ("regulation-on") and 200 populations evolved with regulation-disabled SignalGP ("regulation-off"). All replicates produced a SignalGP program capable of achieving a perfect score during evaluation. We found no evidence that the availability of regulation affected the number of generations required to produce these solutions.

Next, we investigated how well evolved solutions *generalized* across random permutations of input sequences. Selection was deliberately based on a single stochastic ordering of environmental signals, so a "perfect" score may not generalize across all signal orderings. We expect that programs evolved with access to regulation will more often exhibit non-adaptive plasticity that hinders generalization.

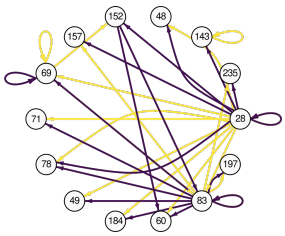
Figure 8 shows the number of evolved solutions from each condition that successfully generalized. All programs that evolved without access to regulation successfully generalized; however, evolved programs from 18 out of 200 successful regulation-on replicates failed to generalize beyond the test cases they experienced during evolution (Fisher's exact test: $p < 6 \times 10^{-6}$). Moreover, 5 of 18



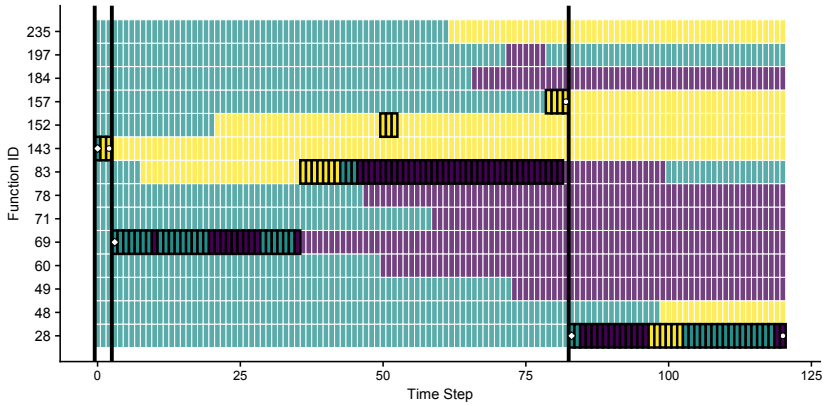
(a) Module regulation over time for a NAND operation.



(b) NAND regulatory network.



(c) NOR regulatory network.



(d) Module regulation over time for a NOR operation.

Fig. 7: **Execution traces of a successful SignalGP program computing a NAND operation (a) and a NOR operation (d).** (b) and (c) show the directed graphs representing the regulatory networks associated with traces (a) and (d), respectively. These visualizations are in the same format as those in Figure 4.

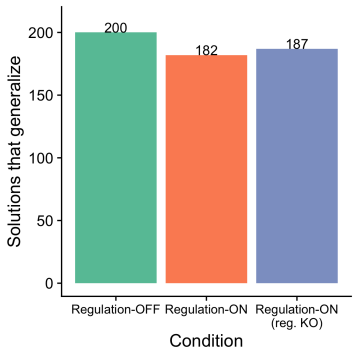


Fig. 8: **The number of evolved solutions that generalize on the independent-signal problem.** The difference in number of solutions that generalize between the regulation-on and regulation-off conditions is statistically significant (Fisher's exact test: $p < 6 \times 10^{-06}$). The "Regulation-ON (reg. KO)" condition comprises the solutions from the Regulation-on condition, except with regulatory instructions knocked out (*i.e.*, replaced with no-operation instructions).

non-generalizing programs generalized when we knocked out tag-based regulation. Upon closer inspection, the other non-general programs relied on tag-based regulation for initial success but failed to generalize to arbitrary environment sequences.

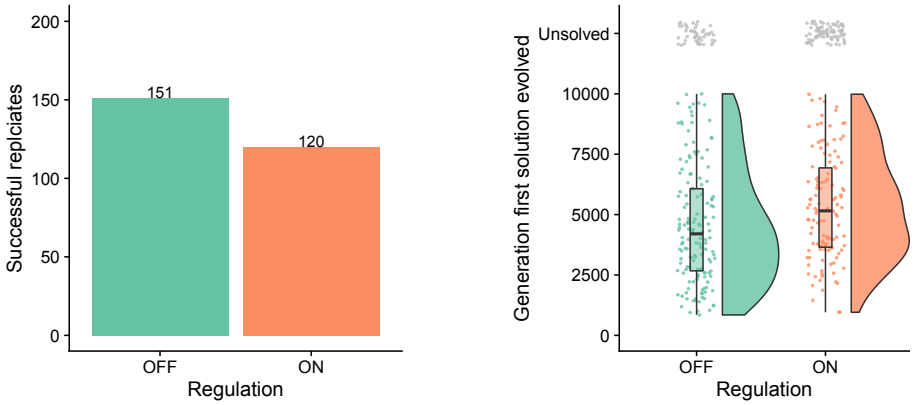
Unexpressed traits that vary in a population (but do not affect fitness) are collectively known as cryptic variation. Cryptic variation is pervasive in nature and thought to play an important role in evolution, potentially acting as a cache of diverse phenotypic effects in novel environments [20, 54]. Such cryptic variation has been shown to help GP systems escape local optima, improving overall problem-solving performance [68]. Cryptic variation arises when environmental conditions that would reveal the variation are not experienced. Access to tag-based regulation appears to make such cryptic variation in evolving programs a stronger possibility than previously. This dynamic can be valuable for performing more realistic studies of evolutionary dynamics with digital organisms (*i.e.*, self-replicating computer programs [76]). However, when using regulation-enabled SignalGP in problem-solving domains, such as automatic program synthesis, non-adaptive plasticity should be accounted for in fitness objectives. In the independent-signal problem, for example, we could have performed more thorough evaluations of programs using multiple random permutations of input sequences instead of one. In more challenging problems, however, more thorough evaluations can come at the cost of substantial computational effort.

5.3 Reducing the context required for the Boolean-logic calculator problem eliminates the benefit of regulation

Experimental results on the independent-signal problem suggest that enabling tag-based regulation is not necessarily beneficial for solving problems that do not require context-dependent responses to input. We use a modified version of the Boolean-logic calculator problem to further investigate the potential for tag-based regulation to impede adaptive evolution. The Boolean-logic calculator problem

as described in Section 4.4 provides inputs in prefix notation: the operator (*e.g.*, AND, OR, XOR, *etc.*) is specified first, followed by the requisite number of numeric operands. As such, the final input signal does not differentiate which type of computation a program is expected to perform. Programs must adjust their response based on the context provided by previous signals, thereby increasing the utility of regulation.

Here, we explore whether the calculator problem's context-dependence is driving the benefit of tag-based regulation that we identified in Section 5.1.3. We can reduce context-dependence of the calculator problem by presenting input sequences in *postfix* notation. In postfix notation, programs receive the requisite numeric operand inputs first and the operator input last. As such, the final signal in an input sequence will always differentiate which bitwise operation should be performed. Successful programs must store the numeric inputs embedded in operand signals, and then, as in the independent-signal problem, a distinct signal will differentiate which of the response types a program should execute.



(a) Successful replicates.

(b) Generations elapsed before solution.

Fig. 9: **Boolean-logic calculator (postfix notation) problem-solving performance.** (a) shows the number of successful replicates for the regulation-off and regulation-on conditions on the postfix Boolean-logic calculator problem. The regulation-on condition was less successful than the regulation-off condition (Fisher's exact test: $p < 0.002$). (b) is a Raincloud plot showing the generation at which the first solution evolved in each successful replicate. Gray points indicate the unsuccessful replicates for each condition. Regulation-off solutions typically required fewer generations than regulation-on solutions to arise (Wilcoxon rank sum test: $p < 0.004$).

We repeated the Boolean-logic calculator experiment (as described in Section 4.4), except we presented inputs in postfix notation instead of prefix notation. Figure 9a shows the number of successful replicates evolved in regulation-on and regulation-off conditions. Postfix notation decreases the overall difficulty of the Boolean-logic calculator problem; more solutions evolved in each condition than evolved with prefix notation (Section 5.1.3). We found that the regulation-on condition resulted in lower problem-solving success than the regulation-off condition. We also found that regulation-off solutions typically required fewer generations

than regulation-on solutions to arise (Figure 9b). Additionally, we did not observe a significant difference in the proportion of flow-control instructions represented in execution traces of regulation-on and regulation-off solutions (supplement Section 11 [40]).

These results, in combination with our previous experimental results, suggest that tag-based regulation is beneficial when prior context dictates behavioral responses to input. On such context-dependent problems, representations without explicit regulation must compensate with additional conditional logic structures.

6 Conclusion

We demonstrated that tag-based genetic regulation allows GP systems to evolve programs with more dynamic plasticity. These evolved programs are better able to solve context-dependent problems where the appropriate software modules to execute in response to a particular input changes over time. Genetic regulation broadens the applicability of SignalGP, both as a representation for problem-solving and as a type of digital organism for studying evolutionary dynamics [39]. Further, this work illustrates an approach for easily incorporating tag-based models of gene regulation into existing GP systems.

Our results also reveal that tag-based regulation is not necessarily beneficial across all problem domains. We observed that the addition of tag-based regulation can impede adaptive evolution on problems where responses to inputs are not context-dependent (*e.g.*, the independent-signal task and postfix version of the Boolean-logic calculator problem). A more thorough examination of what types of context-free problems are most sensitive to tag-based regulation—and how to mitigate any harm—would be potentially fruitful.

Across all problems used in this work, the tag representation and matching scheme that we used was clearly sufficient for success. However, existing tag systems are limited in their capacity to scale up to substantially larger gene regulatory networks. As these networks grow, the specificity required for references to differentiate between modules increases. At some point references become brittle, as any mutation will switch the module that a call triggers. In ongoing work, we are investigating the wide variations in scalability of different metrics for measuring the similarity between tags. Substantial work will also need to be conducted by the community in order to develop more scalable representations for tag-based naming. For example, insights from the indirect referencing mechanisms of artificial biochemical networks and enzyme genetic programming systems may prove to be informative in developing new tag representations [46, 47, 48].

Evolved programs are often more challenging to read and understand than programs written by human developers. In our experience, evolved programs that make use of tag-based regulation were substantially more difficult to read and interpret by hand than evolved programs that do not use tag-based regulation. We found that visualizations of tag-based regulatory networks and program execution traces (*e.g.*, Figures 4 and 7) greatly improved our ability to understand how a given evolved program worked. As we scale up tag-based regulation, the development of interactive visualizations will become increasingly important for understanding evolved programs that use tag-based regulation.

The current investigations have focused on regulation as a problem-solving tool, but with a few extensions these sorts of systems can also help us answer open questions about biological evolution. Our current implementation of tag-based regulation facilitates plasticity only within a program's lifetime; if we extend this capacity across multiple generations, we can study the effects of epigenetic inheritance on evolutionary dynamics. Epigenetic inheritance refers to heritable phenotypic changes that are not directly encoded by the underlying genetic sequence [8, 29]. For example, epigenetics is used in combination with gene regulation for cell-type differentiation in multicellular organisms [50, 62] and caste determination in some species of eusocial insects [71]. SignalGP supports epigenetics with the addition of instructions that mark existing function regulation as heritable. For our next steps, we will apply epigenetics-enabled SignalGP to study fraternal transitions in individuality and the evolution of differentiation before, during, and after a transition occurs [39]. Open-ended experiments with epigenetics and gene regulation will help illuminate the relationship between within-lifetime plastic adaptation and evolutionary adaptation over generational time scales. Additionally, mechanisms for epigenetic inheritance have been shown to potentially improve GP performance [37, 38, 57]; as such, we plan to apply our insights back to automatic program synthesis.

Acknowledgements We thank our anonymous reviewers and Clifford Bohm for feedback and suggestions on this manuscript. We also thank the Digital Evolution Laboratory for thoughtful discussions, ideas, and support. This research was supported in part by NSF grants DEB-1655715, DBI-0939454, and DGE-1424871, and by MSU through the computational resources provided by the Institute for Cyber-Enabled Research. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the NSF. This is a post-peer-review, pre-copyedit version of an article published in Genetic Programming and Evolvable Machines. The final authenticated version is available online at: https://doi.org/10.1007/s10710-021-09406-8

Conflict of interest

The authors declare that they have no conflict of interest.

References

1. Maximino Aldana, Enrique Balleza, Stuart Kauffman, and Osbaldo Resendiz. Robustness and evolvability in genetic regulatory networks. *Journal of Theoretical Biology*, 245(3):433–448, April 2007.
2. JJ Allaire, Yihui Xie, Jonathan McPherson, Javier Luraschi, Kevin Ushey, Aron Atkins, Hadley Wickham, Joe Cheng, Winston Chang, and Richard Iannone. *rmarkdown: Dynamic Documents for R*, 2020. R package version 2.6.
3. Micah Allen, Davide Poggiali, Kirstie Whitaker, Tom Rhys Marshall, and Rogier A. Kievit. Raincloud plots: a multi-platform tool for robust data visualization. *Wellcome Open Research*, 4:63, April 2019.
4. Peter J. Angeline and Jordan B. Pollack. The evolutionary induction of subroutines. In *Proceedings of the Fourteenth Annual Conference of the Cognitive Science Society*, pages 236–241, Bloomington, Indiana, USA, 1992. Lawrence Erlbaum.
5. Dirk Banscherus, Wolfgang Banzhaf, and Peter Dittrich. Hierarchical Genetic Programming using Local Modules. Technical report, Universität Dortmund, October 2001. Publication Title: Reihe Computational Intelligence ; 56.

6. W. Banzhaf. Artificial Regulatory Networks and Genetic Programming. In Rick Riolo and Bill Worzel, editors, *Genetic Programming Theory and Practice*, pages 43–61. Springer US, Boston, MA, 2003.
7. Wolfgang Banzhaf and Lidia Yamamoto. *Artificial chemistries*. The MIT Press, Cambridge, MA, 2015.
8. Judith Bender. Plant epigenetics. *Current Biology*, 12(12):R412–R414, June 2002.
9. Franck Binard and Amy Felty. An abstraction-based genetic programming system. In Peter A. N. Bosman, editor, *Late breaking paper at Genetic and Evolutionary Computation Conference (GECCO'2007)*, pages 2415–2422, London, United Kingdom, 7-11 July 2007. ACM Press.
10. Christos G. Cassandras. The event-driven paradigm for control, communication and optimization. *Journal of Control and Decision*, 1(1):3–17, January 2014.
11. Nichael Lynn Cramer. A representation for the adaptive generation of simple sequential programs. In *Proceedings of the 1st International Conference on Genetic Algorithms*, page 183–187, USA, 1985. L. Erlbaum Associates Inc.
12. Anton Crombach and Paulien Hogeweg. Evolution of Evolvability in Gene Regulatory Networks. *PLoS Computational Biology*, 4(7):e1000112, July 2008.
13. Gabor Csardi and Tamas Nepusz. The igraph software package for complex network research. *InterJournal*, Complex Systems:1695, 2006.
14. Sylvain Cussat-Blanc, Kyle Harrington, and Wolfgang Banzhaf. Artificial Gene Regulatory Networks—A Review. *Artificial Life*, 24(4):296–328, March 2019.
15. Keith L. Downing. *Intelligence emerging: adaptivity and search in evolving neural systems*. The MIT Press, Cambridge, Massachusetts, 2015.
16. J. Draghi and G. P. Wagner. The evolutionary dynamics of evolvability in a gene network model. *Journal of Evolutionary Biology*, 22(3):599–611, March 2009.
17. Kai Olav Ellefsen, Jean-Baptiste Mouret, and Jeff Clune. Neural Modularity Helps Organisms Evolve to Learn New Skills without Forgetting Old Skills. *PLOS Computational Biology*, 11(4):e1004128, April 2015.
18. Richard Forsyth. BEAGLE - A Darwinian Approach to Pattern Recognition. *Kybernetes*, 10(3):159–166, March 1981.
19. Simon Garnier. *viridis: Default Color Maps from matplotlib*, 2018. R package version 0.5.1.
20. Greg Gibson and Ian Dworkin. Uncovering cryptic genetic variation. *Nature Reviews Genetics*, 5(9):681–690, September 2004.
21. Mark Harrower and Cynthia A. Brewer. ColorBrewer.org: An Online Tool for Selecting Colour Schemes for Maps. *The Cartographic Journal*, 40(1):27–37, June 2003.
22. Thomas Helmuth and Lee Spector. General Program Synthesis Benchmark Suite. In *Proceedings of the 2015 on Genetic and Evolutionary Computation Conference - GECCO '15*, pages 1039–1046, Madrid, Spain, 2015. ACM Press.
23. Thomas Helmuth, Lee Spector, and James Matheson. Solving Uncompromising Problems With Lexicase Selection. *IEEE Transactions on Evolutionary Computation*, 19(5):630–643, October 2015.
24. Jose Guadalupe Hernandez, Alexander Lalejini, Emily Dolson, and Charles Ofria. Random subsampling improves performance in lexicase selection. In *Proceedings of the Genetic and Evolutionary Computation Conference Companion on - GECCO '19*, pages 2028–2031, Prague, Czech Republic, 2019. ACM Press.
25. Arend Hintze, Jory Schossau, and Clifford Bohm. The Evolutionary Buffet Method. In Wolfgang Banzhaf, Lee Spector, and Leigh Sheneman, editors, *Genetic Programming Theory and Practice XVI*, pages 17–36. Springer International Publishing, Cham, 2019. Series Title: Genetic and Evolutionary Computation.
26. John H. Holland. Concerning the emergence of tag-mediated lookahead in classifier systems. *Physica D: Nonlinear Phenomena*, 42(1-3):188–201, June 1990.
27. John H Holland. The effect of labels (tags) on social interactions. Technical report, Santa Fe Institute Working Paper 93-10-064. Santa Fe, NM, 1993.
28. Joost Huizinga, Jean-Baptiste Mouret, and Jeff Clune. Does Aligning Phenotypic and Genotypic Modularity Improve the Evolution of Neural Networks? In *Proceedings of the Genetic and Evolutionary Computation Conference 2016*, GECCO '16, pages 125–132, New York, NY, USA, 2016. Association for Computing Machinery. event-place: Denver, Colorado, USA.

29. Eva Jablonka and Gal Raz. Transgenerational Epigenetic Inheritance: Prevalence, Mechanisms, and Implications for the Study of Heredity and Evolution. *The Quarterly Review of Biology*, 84(2):131–176, June 2009.
30. Guy Karlebach and Ron Shamir. Modelling and analysis of gene regulatory networks. *Nature Reviews Molecular Cell Biology*, 9(10):770–780, October 2008.
31. Maarten Keijzer, Conor Ryan, and Mike Cattolico. Run Transferable Libraries — Learning Functional Bias in Problem Domains. In Takeo Kanade, Josef Kittler, Jon M. Kleinberg, Friedemann Mattern, John C. Mitchell, Moni Naor, Oscar Nierstrasz, C. Pandu Rangan, Bernhard Steffen, Madhu Sudan, Demetri Terzopoulos, Dough Tygar, Moshe Y. Vardi, Gerhard Weikum, and Kalyanmoy Deb, editors, *Genetic and Evolutionary Computation – GECCO 2004*, volume 3103, pages 531–542. Springer Berlin Heidelberg, Berlin, Heidelberg, 2004. Series Title: Lecture Notes in Computer Science.
32. Maarten Keijzer, Conor Ryan, Gearoid Murphy, and Mike Cattolico. Undirected Training of Run Transferable Libraries. In David Hutchison, Takeo Kanade, Josef Kittler, Jon M. Kleinberg, Friedemann Mattern, John C. Mitchell, Moni Naor, Oscar Nierstrasz, C. Pandu Rangan, Bernhard Steffen, Madhu Sudan, Demetri Terzopoulos, Dough Tygar, Moshe Y. Vardi, Gerhard Weikum, Maarten Keijzer, Andrea Tettamanzi, Pierre Collet, Jano van Hemert, and Marco Tomassini, editors, *Genetic Programming*, volume 3447, pages 361–370. Springer Berlin Heidelberg, Berlin, Heidelberg, 2005. Series Title: Lecture Notes in Computer Science.
33. J. R. Koza. Hierarchical genetic algorithms operating on populations of computer programs. In N. S. Sridharan, editor, *Proceedings of the Eleventh International Joint Conference on Artificial Intelligence IJCAI-89*, volume 1, pages 768–774, Detroit, MI, USA, 1989. Morgan Kaufmann.
34. John R. Koza. *Genetic programming: on the programming of computers by means of natural selection*. Complex adaptive systems. MIT Press, Cambridge, Mass, 1992.
35. John R. Koza. *Genetic programming II: automatic discovery of reusable programs*. Complex adaptive systems. MIT Press, Cambridge, Mass, 1994.
36. Krzysztof Krawiec and Bartosz Wieloch. Functional modularity for genetic programming. In *Proceedings of the 11th Annual conference on Genetic and evolutionary computation - GECCO '09*, page 995, Montreal, Québec, Canada, 2009. ACM Press.
37. William La Cava, Thomas Helmuth, Lee Spector, and Kourosh Danai. Genetic Programming with Epigenetic Local Search. In *Proceedings of the 2015 on Genetic and Evolutionary Computation Conference - GECCO '15*, pages 1055–1062, Madrid, Spain, 2015. ACM Press.
38. William La Cava and Lee Spector. Inheritable Epigenetics in Genetic Programming. In Rick Riolo, William P. Worzel, and Mark Kotanchek, editors, *Genetic Programming Theory and Practice XII*, pages 37–51. Springer International Publishing, Cham, 2015. Series Title: Genetic and Evolutionary Computation.
39. Alexander Lalejini, Matthew A Moreno, and Charles Ofria. Case study of adaptive gene regulation in dishtiny, Jun 2020. OSF. doi: 10.17605/OSF.IO/KQVMN.
40. Alexander Lalejini, Matthew Andres Moreno, and Charles Ofria. *Supplemental material (GitHub Repository)*, 2021. doi: 10.5281/zenodo.4316015. url: https://lalejini.com/Tag-based-Genetic-Regulation-for-LinearGP/.
41. Alexander Lalejini and Charles Ofria. Evolving event-driven programs with SignalGP. In *Proceedings of the Genetic and Evolutionary Computation Conference on - GECCO '18*, pages 1135–1142, Kyoto, Japan, 2018. ACM Press.
42. Alexander Lalejini and Charles Ofria. Tag-accessed memory for genetic programming. In *Proceedings of the Genetic and Evolutionary Computation Conference Companion on - GECCO '19*, pages 346–347, Prague, Czech Republic, 2019. ACM Press.
43. Alexander Lalejini and Charles Ofria. What Else Is in an Evolved Name? Exploring Evolvable Specificity with SignalGP. In Wolfgang Banzhaf, Lee Spector, and Leigh Sheneman, editors, *Genetic Programming Theory and Practice XVI*, pages 103–121. Springer International Publishing, Cham, 2019.
44. Alexander Lalejini, Michael J. Wiser, and Charles Ofria. Gene duplications drive the evolution of complex traits and regulation. In *Proceedings of the 14th European Conference on Artificial Life ECAL 2017*, pages 257–264, Lyon, France, September 2017. MIT Press.
45. Sylvain Londe, Thibaud Monnin, Raphaël Cornette, Vincent Debat, Brian L. Fisher, and Mathieu Molet. Phenotypic plasticity and modularity allow for the production of novel mosaic phenotypes in ants. *EvoDevo*, 6(1):36, December 2015.

46. Michael A. Lones, Luis A. Fuente, Alexander P. Turner, Leo S. D. Caves, Susan Stepney, Stephen L. Smith, and Andy M. Tyrrell. Artificial Biochemical Networks: Evolving Dynamical Systems to Control Dynamical Systems. *IEEE Transactions on Evolutionary Computation*, 18(2):145–166, April 2014.
47. Michael A. Lones, Alexander P. Turner, Luis A. Fuente, Susan Stepney, Leo S. D. Caves, and Andy M. Tyrrell. Biochemical connectionism. *Natural Computing*, 12(4):453–472, December 2013.
48. Michael A. Lones and Andy M. Tyrrell. Modelling biological evolvability: implicit context and variation filtering in enzyme genetic programming. *Biosystems*, 76(1-3):229–238, August 2004.
49. Rui L. Lopes and Ernesto Costa. The Regulatory Network Computational Device. *Genetic Programming and Evolvable Machines*, 13(3):339–375, September 2012. Place: USA Publisher: Kluwer Academic Publishers.
50. Fabio Mohn and Dirk Schübeler. Genetics and epigenetics: stability and plasticity during cellular differentiation. *Trends in Genetics*, 25(3):129–136, March 2009.
51. Erich Neuwirth. *RColorBrewer: ColorBrewer Palettes*, 2014. R package version 1.1-2.
52. Charles Ofria, Matthew Andres Moreno, Emily Dolson, Alexander Lalejini, Santiago Rodriguez-Papa, Jake Fenton, Katherine Perry, Steven Jorgensen, Riley Hoffman, Robin Miller, Oliver Baldwin Edwards, Jason Stredwick, Nitash C G, Raheem Clemons, Anya Vostinar, Ryan Moreno, Jory Schossau, Luis Zaman, and Dylan Rainbow. Empirical: A scientific software library for research, education, and public engagement, October 2020. doi: 10.5281/zenodo.4141943.
53. Michael O'Neill and Conor Ryan. Grammar based function definition in Grammatical Evolution. In Darrell Whitley, David Goldberg, Erick Cantu-Paz, Lee Spector, Ian Parmee, and Hans-Georg Beyer, editors, *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO-2000)*, pages 485–490, Las Vegas, Nevada, USA, 2000. Morgan Kaufmann.
54. Annalise B. Paaby and Matthew V. Rockman. Cryptic genetic variation: evolution's hidden substrate. *Nature Reviews Genetics*, 15(4):247–258, April 2014.
55. R Core Team. *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing, Vienna, Austria, 2020.
56. Steen Rasmussen, Carsten Knudsen, Rasmus Feldberg, and Morten Hindsholm. The coreworld: Emergence and evolution of cooperative structures in a computational chemistry. *Physica D: Nonlinear Phenomena*, 42(1):111 – 134, 1990.
57. Esteban Ricalde and Wolfgang Banzhaf. Evolving Adaptive Traffic Signal Controllers for a Real Scenario Using Genetic Programming with an Epigenetic Mechanism. In *2017 16th IEEE International Conference on Machine Learning and Applications (ICMLA)*, pages 897–902, Cancun, December 2017. IEEE.
58. J.P. Rosca and D.H. Ballard. Learning by adapting representations in genetic programming. In *Proceedings of the First IEEE Conference on Evolutionary Computation. IEEE World Congress on Computational Intelligence*, pages 407–412, Orlando, FL, USA, 1994. IEEE.
59. Anil Kumar Saini and Lee Spector. Modularity metrics for genetic programming. In *Proceedings of the Genetic and Evolutionary Computation Conference Companion on - GECCO '19*, pages 2056–2059, Prague, Czech Republic, 2019. ACM Press.
60. Anil Kumar Saini and Lee Spector. Using Modularity Metrics as Design Features to Guide Evolution in Genetic Programming. In Wolfgang Banzhaf, Erik Goodman, Leigh Sheneman, Leonardo Trujillo, and Bill Worzel, editors, *Genetic Programming Theory and Practice XVII*, pages 165–180. Springer International Publishing, Cham, 2020. Series Title: Genetic and Evolutionary Computation.
61. Katherine G. Skocelas and Byron DeVries. Test Data Generation for Recurrent Neural Network Implementations. In *2020 IEEE International Conference on Electro Information Technology (EIT)*, pages 469–474, Chicago, IL, USA, July 2020. IEEE.
62. Zachary D. Smith and Alexander Meissner. DNA methylation: roles in mammalian development. *Nature Reviews Genetics*, 14(3):204–220, March 2013.
63. Lee Spector. Simultaneous evolution of programs and their control structures. In Peter J. Angeline and K. E. Kinnear, Jr., editors, *Advances in Genetic Programming 2*, chapter 7, pages 137–154. MIT Press, Cambridge, MA, USA, 1996.
64. Lee Spector, Kyle Harrington, and Thomas Helmuth. Tag-based modularity in tree-based genetic programming. In *Proceedings of the fourteenth international conference on Ge-*

netic and evolutionary computation conference - GECCO '12, page 815, Philadelphia, Pennsylvania, USA, 2012. ACM Press.
65. Lee Spector, Kyle Harrington, Brian Martin, and Thomas Helmuth. What's in an Evolved Name? The Evolution of Modularity via Tag-Based Reference. In Rick Riolo, Ekaterina Vladislavleva, and Jason H. Moore, editors, *Genetic Programming Theory and Practice IX*, pages 1–16. Springer New York, New York, NY, 2011.
66. Lee Spector, Brian Martin, Kyle Harrington, and Thomas Helmuth. Tag-based modules in genetic programming. In *Proceedings of the 13th annual conference on Genetic and evolutionary computation - GECCO '11*, page 1419, Dublin, Ireland, 2011. ACM Press.
67. Alexander P. Turner, Leo S. D. Caves, Susan Stepney, Andy M. Tyrrell, and Michael A. Lones. Artificial Epigenetic Networks: Automatic Decomposition of Dynamical Control Tasks Using Topological Self-Modification. *IEEE Transactions on Neural Networks and Learning Systems*, 28(1):218–230, January 2017.
68. 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, December 2015.
69. Günter P. Wagner, Mihaela Pavlicev, and James M. Cheverud. The road to modularity. *Nature Reviews Genetics*, 8(12):921–931, December 2007.
70. J.A. Walker and J.F. Miller. The Automatic Acquisition, Evolution and Reuse of Modules in Cartesian Genetic Programming. *IEEE Transactions on Evolutionary Computation*, 12(4):397–417, August 2008.
71. Susan A. Weiner and Amy L. Toth. Epigenetics in Social Insects: A New Direction for Understanding the Evolution of Castes. *Genetics Research International*, 2012:1–11, 2012.
72. Hadley Wickham. *reshape2: Flexibly Reshape Data: A Reboot of the Reshape Package*, 2020. R package version 1.4.4.
73. Hadley Wickham, Mara Averick, Jennifer Bryan, Winston Chang, Lucy D'Agostino McGowan, Romain François, Garrett Grolemund, Alex Hayes, Lionel Henry, Jim Hester, Max Kuhn, Thomas Lin Pedersen, Evan Miller, Stephan Milton Bache, Kirill Müller, Jeroen Ooms, David Robinson, Dana Paige Seidel, Vitalie Spinu, Kohske Takahashi, Davis Vaughan, Claus Wilke, Kara Woo, and Hiroaki Yutani. Welcome to the tidyverse. *Journal of Open Source Software*, 4(43):1686, 2019.
74. Hadley Wickham, Winston Chang, Lionel Henry, Thomas Lin Pedersen, Kohske Takahashi, Claus Wilke, Kara Woo, Hiroaki Yutani, and Dewey Dunnington. *ggplot2: Create Elegant Data Visualisations Using the Grammar of Graphics*, 2020. R package version 3.3.2.
75. Claus O. Wilke. *cowplot: Streamlined Plot Theme and Plot Annotations for ggplot2*, 2020. R package version 1.1.0.
76. Claus O. Wilke and Christoph Adami. The biology of digital organisms. *Trends in Ecology & Evolution*, 17(11):528–532, November 2002.
77. Garnett Wilson and Wolfgang Banzhaf. A Comparison of Cartesian Genetic Programming and Linear Genetic Programming. In David Hutchison, Takeo Kanade, Josef Kittler, Jon M. Kleinberg, Friedemann Mattern, John C. Mitchell, Moni Naor, Oscar Nierstrasz, C. Pandu Rangan, Bernhard Steffen, Madhu Sudan, Demetri Terzopoulos, Doug Tygar, Moshe Y. Vardi, Gerhard Weikum, Michael O'Neill, Leonardo Vanneschi, Steven Gustafson, Anna Isabel Esparcia Alcázar, Ivanoe De Falco, Antonio Della Cioppa, and Ernesto Tarantino, editors, *Genetic Programming*, volume 4971, pages 182–193. Springer Berlin Heidelberg, Berlin, Heidelberg, 2008. Series Title: Lecture Notes in Computer Science.
78. Borys Wróbel and Michał Joachimczak. Using the Genetic Regulatory Evolving Artificial Networks (GReaNs) Platform for Signal Processing, Animat Control, and Artificial Multicellular Development. In Taras Kowaliw, Nicolas Bredeche, and René Doursat, editors, *Growing Adaptive Machines*, volume 557, pages 187–200. Springer Berlin Heidelberg, Berlin, Heidelberg, 2014. Series Title: Studies in Computational Intelligence.
79. Yihui Xie. *bookdown: Authoring Books and Technical Documents with R Markdown*, 2020. R package version 0.21.
80. Kwaku Yeboah-Antwi. Evolving software applications using genetic programming – PushCalc: the evolved calculator. In *Proceedings of the fourteenth international conference on Genetic and evolutionary computation conference companion - GECCO Companion '12*, page 569, Philadelphia, Pennsylvania, USA, 2012. ACM Press.