EVOLVING PHENOTYPICALLY PLASTIC DIGITAL ORGANISMS

By

Alexander Lalejini

A DISSERTATION

Submitted to
Michigan State University
in partial fulfillment of the requirements
for the degree of

Computer Science - Doctor of Philosophy
Ecology, Evolutionary Biology, and Behavior - Dual Major

2021

ABSTRACT

EVOLVING PHENOTYPICALLY PLASTIC DIGITAL ORGANISMS

By

Alexander Lalejini

The ability to dynamically respond to cues from the environment is a fundamental feature of most adaptive systems. In biological systems, changes to an organism based on environmental cues is called phenotypic plasticity. Indeed, phenotypic plasticity underlies many of the adaptive traits and developmental patterns found in nature and serves as a key mechanism for responding to spatially or temporally variable environments. Most computer programs require phenotypic plasticity, as they must respond dynamically to stimuli such as user input, sensor data, *et cetera*. As such, phenotypic plasticity also has practical applications in genetic programming, wherein we apply the natural principles of evolution to automatically synthesize computer programs rather than writing them by hand.

In this dissertation, I achieve two synergistic aims: (1) I use populations of self-replicating computer programs (digital organisms) to empirically study the conditions under which adaptive phenotypic plasticity evolves and how its evolution shapes subsequent evolutionary outcomes; and (2) I transfer insights from biology to develop novel genetic programming techniques in order to evolve more responsive (*i.e.*, phenotypically plastic) computer programs. First, I illustrate the importance of mutation rate, environmental change, and partially-plastic building blocks for the evolution of adaptive plasticity. Next, I show that adaptive phenotypic plasticity stabilizes populations against environmental change, allowing them to more easily retain novel adaptive traits. Finally, I improve our ability to evolve phenotypically plastic computer programs with three novel genetic programming techniques: (1) SignalGP, which provides mechanisms to control code expression based on environmental cues, (2) tag-based genetic regulation to adjust code expression based on current context, and (3) tag-accessed memory to provide more dynamic mechanisms for storing data.

For Alexa.

ACKNOWLEDGEMENTS

As I complete this dissertation, I stand on the shoulders of many people who have inspired and supported me over the course of my Ph.D. First and foremost, I thank Alexa Lalejini, who (by request) gets their very own paragraph, and without whom, this dissertation would not have been possible. Thank you for moving to Michigan with me and making sure that I eat and sleep. Thank you for putting up with an apartment where the brick walls are absurdly cold in the winter and leak during rain storms because I thought it was an "interesting" place to live. Thank you for helping me to take breaks by whisking me away on adventures. Thank you for making sure that I eat and sleep. Most of all, thank you for keeping me company.

Next, I thank my friends and family for their support over the years. I thank my parents, David and Penny Lalejini, for fostering my academic interests. I especially want to acknowledge my dad who I could always count on for help with math homework or for someone to babble to about research projects. I thank Brandon Odom for always being there for Alexa and me. I also thank Josh Nahum for being my dependable exercise partner for the entire duration of my Ph.D. and for all of his friendship, kindness, and generosity.

I have also had wonderful mentors, without whom I could not have bumbled my way through graduate school. As an undergraduate, I would not have been aware of graduate school as a realistic path for myself if not for Dr. Cindy Bethel at Mississippi State University who trusted me with research projects, funded trips to conferences, let me use her lab space after hours as a safe space to study for exams and do homework, and continues to be a font of advice and encouragement.

It has been a great privilege to have had Dr. Charles Ofria as my Ph.D. advisor and mentor. Charles' immense excitement for research, teaching, and mentoring is infectious. Charles is one of my most valued role models in science, academia, and life. I am also deeply appreciative of the supportive culture that Charles actively cultivates in the Digital

Evolution Lab. I want to specifically thank Dr. Anya Vostinar and Dr. Emily Dolson, from each of whom I have learned so much about being a good researcher, academic, and person. I also want to thank my committee—Dr. Christoph Adami, Dr. Wolfgang Banzhaf, and Dr. Richard Lenski—for their guidance and insightful feedback on my work.

Finally, I want to acknowledge all of the members of the Digital Evolution Lab, past and present. Our constructive discussions, collaborations, and social gatherings have enriched my work. I especially want to thank Dr. Mike Wiser, Dr. Anya Vostinar, Dr. Rosangela Canino-Koning, Dr. Emily Dolson, Anselmo Pontes, Matthew Andres Moreno, Jose Guadalupe Hernandez, Austin J. Ferguson, Acacia Ackles, Kate Skocelas, and Clifford Bohm.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

Chapter 1

Introduction

This dissertation straddles basic research using computational systems for experimental evolution and more applied research for evolutionary computation. These two disciplines have divergent goals, but are unified by our ability to implement, observe, and exploit the constructive process of evolution *in silico*. Experimental evolution allows us to test general hypotheses about evolutionary processes by studying real-time evolutionary changes occurring in experimental populations in response to conditions imposed by the experimenter (Kawecki et al., 2012). Conventionally, evolution experiments are performed under laboratory conditions using populations of biological organisms that are tractable to observe and experimentally manipulate (*e.g.*, *Escherichia coli*, *Pseudomonas*, *Saccharomyces cerevisiae*, *Drosophila melanogaster*, and a variety of phage-bacteria systems). For example, over 70,000 generations of evolution have elapsed the ongoing long-term evolution experiment with *E. coli* (Barrick et al., 2020), which has yielded analyses on a wide range of topics, including long-term evolutionary dynamics (Wiser et al., 2013; Good et al., 2017), historical contingency (Travisano et al., 1995; Card et al., 2019), the evolution of mutation rates (Sniegowski et al., 1997), the origins of novel traits (Blount et al., 2008), and the maintenance of phenotypic plasticity under relaxed selection (Grant et al., 2020). In my work, I conduct experimental evolution studies using populations of *digital* organisms, which are self-replicating computer programs that compete, mutate, and evolve in computational environments.

Insights gained from experimental evolution studies can also be useful for more applied

goals. Evolutionary computation exploits the natural principles of evolution as a general purpose search algorithm in order to solve challenging computational problems. These evolutionary algorithms begin with an initial population of individuals, be they computer programs, neural networks, robot body plans, or potential solutions to some other kind of a well-defined problem. Each generation, candidate solutions are evaluated on one or more criteria to determine their quality. After evaluating the population, promising individuals are selected as parents to contribute genetic material to produce the next generation of individuals. Evolutionary algorithms direct populations through a problem's search space via repeated evaluation, selection, and variation (*i.e.*, replicating promising individuals with random mutations) until a sufficiently good solution is found. In my dissertation work, I focus on *genetic programming* (GP) wherein we apply evolutionary algorithms to automatically synthesize computer programs rather than writing them by hand.

Advances in genetic programming and digital evolution research are synergistic. Both genetic programming and digital evolution systems evolve computer programs albeit with different goals in mind; as such, similar methods for representing and interpreting computer programs can be shared across disciplines. Further, digital evolution studies contribute to a deeper understanding of the open-ended evolutionary processes that continue to generate adaptive biological complexity. We can exploit this understanding to improve existing evolutionary computing techniques or to inspire new evolutionary algorithms altogether (*e.g.*, Goldberg and Richardson 1987; Spector 2011; Goings et al. 2012). Likewise, advances in evolutionary computing can improve our ability to model evolutionary processes *in silico* by providing new ways of representing digital organisms, data analysis techniques, and visualizations.

In my first two research chapters (Chapters 2 and 3), I focus on phenotypic plasticity, which is the capacity for a single genotype to express different phenotypes in response to a change in its environment (West-Eberhard, 2003). Phenotypic plasticity underlies many complex traits and developmental patterns found in nature and serves as a key mechanism for

responding to spatially and temporally variable environments (Bradshaw, 1965). For example, genetically homogeneous cells in a developing multicellular organism require phenotypic plasticity to coordinate their expression patterns through environmental signals (Schlichting, 2003). Indeed, biologists have long been interested in understanding how adaptive phenotypic plasticity evolves, the mechanisms underpinning plasticity in natural organisms, and how the evolution of plasticity influences subsequent evolutionary outcomes (Gibert et al., 2019). In Chapter 2, I investigate how mutation rate and environmental change rate affect the evolution of adaptive phenotypic plasticity, and I additionally identify intermediate evolutionary stepping stones along the lineages of adaptively plastic digital organisms. In Chapter 3, I shift my focus from the evolutionary origins of adaptive plasticity to its evolutionary consequences. Specifically, I explore how the evolution of plasticity affects the rate of subsequent evolutionary change and the evolution and maintenance of novel adaptive traits.

Phenotypic plasticity also has practical applications in evolutionary computing, which I explore in Chapters 4, 5, and 6. In many realistic problem domains, conditions are noisy or cyclically change. As in biological organisms, phenotypic plasticity can allow generated solutions to be robust to noise and capable of dynamically responding to changing problem conditions (*e.g.*, Soltoggio et al. 2018).

Synthesizing computer programs capable of complex forms of adaptive plasticity is especially relevant to genetic programming. Automating software development is a long-standing goal in the genetic programming community (Koza, 1989; O'Neill and Spector, 2019). All useful software applications require some degree of phenotypic plasticity in order to conditionally respond to inputs. Indeed, most software applications require even more advanced forms of phenotypic plasticity, as they must *regulate* responses to inputs based on prior context. For example, the computations that must occur on a calculator after pressing the "equals" button depend on the set of inputs previously provided. I argue that we can draw on our understanding of biological mechanisms of adaptive plasticity and their evolution to evolve more dynamically responsive computer programs.

If conventionally written software commonly contains complex forms of adaptive plasticity, why not evolve programs constructed from conventional programming languages? The programming languages used by human software developers are not easily evolvable (Rasmussen et al., 1990). Software written with a conventional programming language is not robust to minor perturbations (*e.g.*, mutations). Yet, many of the mechanisms required for adaptive plasticity in existing genetic programming representations (*e.g.*, conditional logic and jump-based flow control) are the same mechanisms used by traditional programming languages. Can genetic programming do better than using such conventional mechanisms? In my work, I look to the evolved mechanisms of plasticity in biological organisms to improve the way in which we represent computer programs for evolution. In Chapter 4, I introduce SignalGP, a novel genetic programming technique for evolving event-driven programs that handle signals from the environment or from other agents in a more biologically inspired way than traditional GP approaches. Next in Chapter 5, I introduce tag-based module regulation for genetic programming, which allows us to more easily evolve programs capable of dynamically regulating responses to inputs over time. Finally, in Chapter 6, I briefly introduce tag-accessed memory, a more flexible approach to labeling and accessing memory than traditional direct-addressed memory schemes.

1.1 Digital Evolution

Digital evolution experiments have emerged as a powerful research framework from which evolution can be studied. In digital evolution, self-replicating computer programs (digital organisms) compete for resources, mutate, and evolve in a computational environment (Wilke and Adami, 2002). In my work, a digital organism comprises a linear sequence of program instructions (a genome) and a set of virtual hardware components used to interpret and express those instructions. To reproduce, a digital organism must execute instructions that allow it to copy its genome instruction-by-instruction and then divide (producing an offspring). However, self-replication is imperfect and can result in mutated offspring. The

combination of heritable variation due to imperfect self-replication and competition for limited resources (*e.g.*, space, CPU time, *etc.*) results in evolution by natural selection.

Digital organisms live, interact, and evolve in entirely artificial environments constructed by the experimenters. One potential drawback to digital evolution is that the conclusions drawn from an experiment have the potential to be artifacts of the constructed artificial environment (Wilke and Adami, 2002). This drawback, however, can also be applied to most microbial experimental evolution where organisms are extracted from their natural environment and placed in an artificial environment constructed in a laboratory.

Microbial model organisms at least have natural ancestry and can often be used to infer historic evolutionary events. Digital evolution studies, however, are not grounded in the same evolutionary history and biochemical compounds as carbon-based life on Earth. This limitation makes it more challenging to use digital evolution studies to illuminate idiosyncrasies and contingencies associated with the history of life on our planet. However, these drawbacks are also digital evolution's strength as a research framework, since we are not limited to studying only one particular instance of evolution or locked in to using nucleic-acid, amino acid, and protein based representations. Furthermore, we can fully observe and control digital environments at rapid speeds, allowing us to perform experiments and analyses that would otherwise be challenging or even impossible to perform in biological systems. Additionally, by reproducing results across biological and digital systems, we can disentangle general principles from effects specific to a particular model organism or planetary body (Wilke and Adami, 2002).

In the remainder of this section, I provide historical context for digital evolution research, discuss the benefits of experimental digital evolution, and highlight prior digital evolution research related to phenotypic plasticity.

1.1.1 Historical context

> *Two computer programs in their native habitat—the memory chips of a digital computer—stalk each other from address to address.* (Dewdney, 1984)

Modern digital evolution systems can be traced back to the 1984 computer game "Core War" (Dewdney, 1984). In Core War, human competitors use a simplified assembly language (called Redcode) to write "gladiatorial" computer programs that compete for space in the simulated core memory of a computer. To win a bout of Core War, a program must shut down all of the processes associated with its competitor programs. The most successful programs all engaged in self-replication. Such replicator programs repeatedly created copies of themselves, each of which repeatedly copied themselves *ad infinitum*. Thus, if one copy were to be destroyed by an adversary, other copies would still persist to continue replicating. Replicators could grow exponentially in memory, rapidly outcompeting other programs and taking over core memory. Despite having populations of self-replicating programs and competition, evolution did not occur in Core War because replicators always created perfect copies of themselves.

Inspired by Core War, Rasmussen et al. created Core World (Rasmussen et al., 1989). Core World used the same Redcode language to represent programs, and programs competed in the same computational environment as in Core War. However, Core World introduced the possibility for random mutations when a program copied itself (Rasmussen et al., 1989, 1990). That is, the command used by replicator programs to copy themselves was imperfect, sometimes writing a random instruction instead of copying the intended instruction. Indeed, Core World succeeded in facilitating the evolution of populations of computer programs. However, the Core World system proved to be ill-suited for studying evolution. Programs written in Redcode were not designed to survive mutations, and as such, accumulated deleterious mutations often drove the populations to extinction.

Thomas Ray's Tierra system (Ray, 1991) innovated on the design of Core World and facilitated some of the first successful evolution experiments with self-replicating computer

programs. The programming language used to construct the genomes of evolving programs in Tierra was more syntactically robust than Redcode. As such, genomes in Tierra were more evolvable than those in Core World because mutated daughter programs were less often broken. Furthermore, in contrast to Core World, Tierra protected "living" programs from being overwritten by their competitors, requiring programs to explicitly request a protected block of memory into which they could copy themselves. When the population grew to the environment's carrying capacity, Tierra removed the oldest programs from the population to make room for new programs to be born.

In initial experiments using Tierra, Ray founded populations with an ancestral program capable only of self-replication (Ray, 1991). Competition for space dominated these early studies, resulting in a strong selection pressure for organisms to increase their replication rate. Indeed, Ray observed organisms with shorter genomes evolve and outcompete organisms with longer genomes, as shorter genomes could be copied faster because they contained fewer instructions that needed to be copied to produce an offspring. Ray unexpectedly observed the evolution of obligate parasites—programs that co-opted the copy machinery of their competitors to copy themselves[1]. An evolutionary arms race ensued. Would-be "host" programs evolved mechanisms for resisting parasites, and in turn, parasites evolved to penetrate those defensive mechanisms. The richness of observed evolutionary dynamics in Tierra was initially surprising given the simplicity of Tierra's environment. These initial experiments positioned digital evolution as a promising endeavor for studying evolutionary processes.

The Avida Digital Evolution Platform expanded on the design of Tierra but added the ability to configure complex environments and sophisticated data tracking tools (Adami and Brown, 1994; Ofria and Wilke, 2004; Ofria et al., 2009). In Avida, digital organisms compete for space on a lattice of cells (Ofria et al., 2009). When an organism reproduces, its offspring is placed in a nearby cell (or in a random cell if the population is well-mixed),

[1]Ray labeled these programs as parasites, but they are more accurately described as cheaters because they did not directly harm the programs whose replication machinery they co-opted.

replacing any previous occupant of that cell. As in Tierra, improvements to the speed of self-replication are advantageous in the competition for space in the environment, and organisms in Avida can improve their replication rates by improving genome efficiency (*e.g.*, using a more compact encoding). Avida, however, introduced the concept of resources that can be "metabolized" by a digital organism to accelerate the rate at which it expresses its genome (*i.e.*, its "metabolic rate"). Resources in Avida are associated with completing designated tasks, such as computing Boolean logic functions on inputs from the environment. Avida gives experimenters fine-grained control over how resources are configured, including their abundance (Cooper and Ofria, 2002), spatial distribution (Dolson et al., 2017), and their metabolic effects (Canino-Koning et al., 2016, 2019).

The Avida system is perhaps the most widely used digital evolution system to date and is often credited with advancing digital evolution as a model system for conducting scientifically rigorous evolution experiments. Experimental evolution studies using Avida have been well received, and topics such as the evolution of complexity (Adami et al., 2000; Lenski et al., 2003), sexual recombination (Misevic et al., 2010), modularity (Misevic et al., 2006), robustness (Lenski et al., 1999; Elena et al., 2007), and division of labor (Goldsby et al., 2012a, 2014) have been published in top evolutionary biology venues. Given Avida's track record, I used it to conduct the studies presented in my first two research chapters (Chapters 2 and 3).

1.1.2 The Benefits of Digital Evolution

Evolution experiments using digital organisms balance the speed and transparency of mathematical and computational simulations with the open-ended realism of laboratory experiments. Here, I overview four properties of digital evolution systems that make them valuable complements to traditional carbon-based model organisms for studying evolutionary processes, providing exemplars of each:

Generality

Digital evolution systems offer researchers the unique opportunity to study evolution in organisms that share no ancestry with carbon-based life (Wilke and Adami, 2002). As biologist John Maynard Smith made the case, "So far, we have been able to study only one evolving system and we cannot wait for interstellar flight to provide us with a second. If we want to discover generalizations about evolving systems, we will have to look to artificial ones" (Maynard Smith, 1992). Indeed, studies of carbon-based lifeforms that all share common ancestry dominate evolutionary biology. On their own, these studies can provide deeper insights into life on Earth. However, such studies provide a limited lens with which to make generalizations about evolutionary processes, as they are biased by the particular history of life on our planet. By testing hypotheses across biological and digital model systems, we can disentangle general principles from the effects of specific model organisms.

For example, what is the relative importance of adaptation, chance, and history in explaining diversity in evolved populations? Using experimental populations of *Escherichia coli*, Travisano et al. disentangled the relative contributions of adaptation, chance, and history in the evolution of fitness and cell size (a trait weakly correlated with fitness) (Travisano et al., 1995). Travisano et al. found that fitness gains were most strongly influenced by adaptive processes, and variance in cell size were most explained by chance and history. Wagenaar and Adami replicated this study with Avida (Wagenaar and Adami, 2004), finding that the overall patterns observed in *E. coli* and in digital organisms were broadly similar. Ongoing studies in digital organisms are extending these concepts further, using more restarts at different time points and across different environments, allowing us to explore more of the nuances at play.

Transparency

Digital evolution systems allow for perfect, non-invasive data tracking. Experimenters can save the complete details of evolving populations for further analysis, including every

mutation that occurs, every genotype that exists, every phenotype that is expressed, every environmental state that occurs, every time an organism interacts with another organism or with the environment, *et cetera*. By tracking parent-offspring relationships, we can analyze complete evolutionary histories within an experiment, which circumvents the historical problem of drawing evolutionary inferences using incomplete records (from frozen samples or fossils) and extant genetic sequences.

Many digital evolution studies inspect the complete lineages of evolved digital organisms to tease apart the mutation-by-mutation evolution of novel traits (Lenski et al., 2003; Dolson et al., 2017; Grabowski et al., 2013; Goldsby et al., 2014; Pontes et al., 2020). In an exemplary analytical undertaking, Dolson and Ofria identified spatial hotspots of evolutionary potential in heterogeneous environments (*i.e.*, positions where novel traits disproportionately evolved). They found evidence that the particular *paths* traversed by lineages through space might explain the locations of these evolutionary hotspots (Dolson and Ofria, 2017). Recently, Dolson et al. reviewed a breadth of ancestry-based metrics and analyses that operate on lineages and phylogenies in an effort to improve our capacity to quantitatively explore evolutionary histories in digital evolution experiments (Dolson et al., 2020).

Recording organism relationships and interactions can be valuable for many other goals as well. For example, by tracking phenotypes over time, Cooper and Ofria were able to observe the real-time evolution of stable ecosystems under resource-limited conditions (Cooper and Ofria, 2002). In a similar vein, Fortuna et al. tracked host-parasite interactions to investigate how the structure of infection networks is shaped by antagonistic coevolution (Fortuna et al., 2019).

Control

Digital evolution systems facilitate experimental manipulations that go beyond what is possible in laboratory or field experiments. These capabilities allow researchers to empirically test hypotheses that would otherwise be relegated to theoretical analyses. For example,

digital evolution systems allow experimenters to precisely control basic parameters such as population size and mutation rate. By comparing populations evolving under different mutation rates, Wilke et al. discovered the "survival of the flattest" effect where high mutation rate environments selected for genomes with slower replication rates but that were more robust to mutations (Wilke et al., 2001).

Digital evolution experiments also allow for fine-grained control over other aspects of an environment. For example, Dolson et al. used Avida to experimentally manipulate the spatial distribution of resource availability, finding that phenotypic diversity was positively correlated with spatial entropy and that spatially heterogeneous environments exhibited increased evolutionary potential relative to more homogeneous environments (Dolson et al., 2017). By experimentally controlling how environments changed temporally, Nahum et al. demonstrated that a single temporary environmental change can improve fitness landscape exploration and exploitation in evolving populations of digital organisms (Nahum et al., 2017).

Digital evolution systems also allow experimenters to monitor and manipulate mutational effects in real-time. Covert et al. performed real-time reversions of all deleterious mutations as they occurred to isolate their long-term effects on evolutionary outcomes (Covert et al., 2013). Lalejini et al. implemented a range of slip duplication mutation operators (each designed to isolate a single effect of duplication mutations) in order to tease apart why such mutations can promote the evolution of complex traits (Lalejini et al., 2017).

For an individual digital organism, we can perform systematic knockout analyses to identify which instructions are responsible for producing a given phenotypic outcome. This sort of analysis has been applied along lineages to identify how information accumulates (Ofria et al., 2008) or to investigate how environmental change shapes the evolution of genetic architectures in digital organisms (Canino-Koning et al., 2016). Mutational landscaping analyses go a step further than knockout analyses, allowing experimenters to fully characterize a local mutational landscape by evaluating all possible one- and two-step mutants. Such analyses

have been used to quantify epistasis (Lenski et al., 1999) and mutational robustness (Elena et al., 2007) and to investigate the evolution of evolvability (Canino-Koning et al., 2019).

Scale

Modern computers allow us to observe many generations of digital evolution at tractable time scales; thousands of generations can take mere minutes as opposed to months, years, or centuries. For example, populations of digital organisms have been used to test theoretical predictions about the expected rate of adaptation over hundreds of thousands of generations (Wiser, 2015; Wiser et al., 2018).

Additionally, digital evolution experiments allow researchers to enact complex experimental protocols with minimal extra effort. That is, unlike in wet-lab experiments, computational experiment protocols can easily be automated using modern scripting tools.

With the increasing accessibility of high performance computing systems, it can be trivial to evolve hundreds of replicate populations for a given experimental treatment. Evolution is an inherently stochastic process, so increased replication provides a clearer picture of the distribution of possible treatment effects. Further, a high degree of replication increases the odds that experimenters will be able to observe and study rare events. For example, Pontes et al. (2020) evolved 900 replicate populations of digital organisms in order to observe 10 examples of reversal learning behavior (*i.e.*, the ability to relearn associations between cues and responses when cues are swapped) to further analyze.

Even the fastest computing systems, however, lack the parallelism of the real world. That is, digital evolution systems cannot yet rival bacterial systems in their ability to scale to large population sizes. A typical population of digital organisms contains thousands to tens of thousands of organisms; however, microbial populations used in laboratory experiments often contain several orders of magnitude more individuals.

1.1.3 Phenotypically plastic digital organisms

Phenotypic plasticity has been the subject of many computational evolution studies. Here, I focus on previous work using self-replicating computer programs. Clune et al. demonstrated that adaptively plastic digital organisms can evolve in Avida under the following conditions: a fluctuating environment where conditions are differentiable by reliable cues (sensory instructions), and each condition favors different phenotypic traits (performing different computational tasks) such that no single phenotype exhibits high fitness across all conditions. Clune et al. also characterized two mechanisms by which digital organisms tended to achieve phenotypic plasticity. First, *dynamic-execution-flow* plasticity uses conditional logic statements (*e.g.*, if statements) to modify which instructions are executed based on environmental conditions. Second, *static-execution-flow* plasticity integrates sensory information into internal "metabolic" pathways such that the same sequence of program instructions is always executed, but produces different behaviors in different environmental conditions.

Genetically homogeneous groups of individuals (*e.g.*, cells in a multicellular organism or members of a eusocial insect colony) require phenotypic plasticity to differentiate and coordinate their behavior (Schlichting, 2003). Indeed, digital evolution studies have demonstrated the *de novo* evolution of adaptive phenotypic plasticity that allows "multicellular" collectives of digital organisms to coordinate their behavior. Goldsby et al. showed that direct selection for task specialization in clonal groups of digital organisms promotes the evolution of differentiation and division of labor (Goldsby et al., 2010). In addition, Goldsby et al. demonstrated that task-switching costs can promote the evolution of division of labor (Goldsby et al., 2010, 2012a). Digital organisms have also been used to study the evolution of synchronization and desynchronization (Knoester and McKinley, 2011) and the evolution of consensus (Knoester et al., 2013) in groups of genetically homogeneous individuals.

Digital organisms have also been used to study the evolutionary conditions that give rise to temporal polyethism, a form of behavioral plasticity exhibited by many eusocial insect

species whereby the tasks that an individual attempts to perform are correlated with the individual's age. Goldsby et al. demonstrated that differential task-riskiness is sufficient to promote the evolution of temporal polyethism in genetically homogeneous groups (Goldsby et al., 2012b). Individuals within a group used control-flow instructions to regulate task performance, performing low or no risk tasks early in life and then switching to performing higher risk tasks later in life.

The dirty work hypothesis predicts that the mutagenic effects associated with metabolism can promote the evolution of plasticity in the form of germ–soma differentiation in multicellular organisms (Goldsby et al., 2014). Goldsby et al. tested the dirty work hypothesis using digital organisms, finding that individuals within a multicellular group used phenotypic plasticity both to differentiate between germ and soma and to efficiently divide mutagenic tasks amongst somatic cells.

Quorum sensing is a form of communication used for plasticity in many species of bacteria, allowing individuals to regulate their actions depending on the density of the surrounding population (Miller and Bassler, 2001). Beckmann et al. demonstrated the evolution of quorum sensing in digital organisms whereby individuals adaptively suppress self-replication based on their local population density (Beckmann and McKinley, 2009; Beckmann et al., 2012). Johnson et al. expanded this work, showing that the evolution of quorum sensing can improve the efficacy of adaptive suicidal altruism—a strategy where an altruistic organism dies to increase the fitness of kin—by helping altruistic individuals regulate when to die (Johnson et al., 2014).

Using digital organisms, Wagner et al. investigated how predator-prey coevolution influences the subsequent evolution of behavioral plasticity in predator and prey species (Wagner et al., 2014, 2020). Wagner et al. found increased sensor reliance and behavioral plasticity in prey that coevolved with predators than in prey that evolved without predators. Indeed, prey seemed to exapt genetic components of evolved sense-and-flee predator avoidance strategies for sense-and-retrieve foraging strategies.

Grabowski et al. tested whether digital organisms could evolve to plastically use information about past experiences for optimal decision making (Grabowski et al., 2010). Specifically, an organism's reproductive success was tied to its ability to traverse a nutrient trail. To follow a trail, organisms needed to sense and react appropriately to environmental cues that indicated how to remain on the trail. Grabowski et al. found that memory usage evolved only when it provided a substantial advantage; otherwise, organisms tended to adopt reflexive strategies that did not require memory. Some memory-based strategies relied on an evolved odometry sensor wherein organisms tracked the number of steps taken and their orientation. Expanding on this work, Grabowski et al. used lineage analyses to disentangle the step-by-step evolution of such odometry-based strategies (Grabowski et al., 2013).

Building on Grabowski et al.'s work, Pontes et al. used digital organisms to investigate the selective pressures that promote more complex forms of plasticity such as the evolution of associative learning (Pontes et al., 2017, 2020). Pontes et al. evolved organisms capable of associating novel environmental cues with their meaning in different contexts. Environments that were stable across generations promoted the evolution of purely reflexive behavior, and environments that varied across generations (but remained stable during an organism's lifetime) promoted the evolution of learning. Pontes et al. found evidence that reflexive behaviors were a necessary building block for the evolution of learning, indicating that both types of environments were important.

As reviewed above, the majority of prior work investigates diverse forms of phenotypically plastic behaviors in digital organisms, with a focus on the selective pressures that promote their evolution. A smaller subset of prior work used lineage analyses to illuminate the step-by-step process by which phenotypic plasticity tended to evolve (*e.g.*, Grabowski et al. 2013; Goldsby et al. 2014; Pontes et al. 2020). Each of these prior studies have focused on the evolution of adaptive plasticity, but have not emphasized its influence on subsequent evolutionary outcomes. In the chapters below, I further examine the origins of phenotypic plasticity, extend these analyses to explore the consequences of plasticity of future evolution,

and further investigate how to harness plasticity for more applied goals.

1.2 Genetic programming

Both digital evolution and genetic programming (GP) systems evolve populations of computer programs, but each does so with a different objective. Digital evolution aims to use computer programs as model organisms for evolution experiments, whereas GP aims to synthesize computer programs to solve computational problems. As such, GP systems often ignore much of the biological realism that is present in digital evolution systems in order to increase problem-solving efficiency by actively steering populations toward promising regions of the search space.

Most GP systems follow the same overarching recipe for synthesizing computer programs (Ofria et al., 2009):

1. **Initialize** a population of programs (usually with randomly generated programs or hand-designed programs).
2. **Evaluate** each program's quality relative to one or more criteria.
3. **Select** promising programs to contribute genetic material to the next generation based on their quality.
4. **Vary** selected programs by mutating or recombining them to produce the next generation of programs.
5. **Repeat** this process from step two until a sufficiently good program is generated.

Of course, the details of each of these components—initialization, evaluation, selection, and variation—vary dramatically across GP systems (Poli et al., 2008), as different techniques are more or less effective depending on the problem domain. In my final three research chapters (Chapters 4, 5, and 6), I focus on another fundamental aspect of genetic

programming: the substrate being evolved. A major challenge with any problem being solved with GP is determining how to represent and interpret the computer programs that we evolve. More specifically, I ask how we can better represent computer programs such that we can more easily evolve programs capable of complex forms of adaptive phenotypic plasticity. I want GP to be able to produce programs that can dynamically respond to external conditions (including user input) while modifying their behavior based on prior events.

Given that software developers commonly write highly responsive, "phenotypically plastic" programs, perhaps the obvious choice would be to evolve programs with one of the many modern programming languages used by professional software developers. However, as early digital evolution studies revealed, conventional programming languages are ill-suited for evolving computer programs (Rasmussen et al., 1989). For example, any professional software developer will attest that random perturbations (mutations) to a conventionally written program is likely to fatally break its functionality. Even experiments that preserve syntax, however, still find it challenging to cope with the complexity and brittleness of human programming languages, though progress has been made in automatically repairing bugs in human-written code (Le Goues et al., 2012b,a; Yuan and Banzhaf, 2020). As a result, a substantial amount of research in the GP community revolves around developing and analyzing new languages and techniques for representing evolvable computer programs.

Just as human software developers have access to an enormous variety of specialized programming languages, 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 (Hintze et al., 2019; Wilson and Banzhaf, 2008).

The earliest examples of successfully evolving computer programs used tree-based representations (Forsyth, 1981; Koza, 1989). In tree-based GP, programs are organized as abstract syntax trees (Poli et al., 2008). The leaves of a tree are variable inputs or constants (*i.e.*,

terminals) and the internal nodes are typically arithmetic operations (*e.g.*, addition, multiplication, *etc.*). Trees are conventionally executed using preorder traversal. That is, tree execution begins at the root, which immediately request the return value of its first sub-tree, triggering a recursive execution pattern. In practice, results from a tree are produced in a bottom-up fashion; that is, the bottom-most operations are resolved first, and the results of lower operations are propagated up the tree (as inputs to operations higher in the tree) until the root can finish being executed to produce the program's final output. Tree-based programs typically describe multivariate mathematical functions. As such, tree-based GP is often applied to symbolic regression problems (Orzechowski et al., 2018).

Since the early success of tree-based GP, a wide range of other GP representations have been developed, including graph-based GP (Miller, 1999; Kelly and Heywood, 2017), stack-based GP (Perkis, 1994; Spector, 2001), and linear GP (Brameier and Banzhaf, 2007). In particular, linear GP represents programs as linear sequences of instructions and is the technique used for most conventional digital organism research. Linear genetic programs follow an imperative paradigm where computation is procedural: execution often starts at the top of the program and proceeds instruction-by-instruction, jumping or branching as dictated by executed instructions. Indeed, due to linear GP's similarity with conventional digital organism genetic representations, the techniques that I propose in Chapters 4, 5, and 6 are in the context of linear GP, facilitating more direct knowledge transfer between my digital evolution and GP research. Within each of these chapters, I provide a more targeted literature review of the specific types of GP techniques that I am examining.

1.3 Thesis Statement

Adaptive systems require phenotypic plasticity to dynamically respond to complex and ever-changing environments. We must study digital evolution and genetic programming systems if we are to understand how plasticity evolves, how it shapes subsequent evolutionary outcomes, and how to harness it to synthesize adaptive computer programs.

1.4 Contributions

This dissertation can be divided into two parts. In part one (Chapters 2 and 3), I conducted digital evolution studies to investigate the evolutionary origins and consequences of adaptive phenotypic plasticity in cyclic environments. In part two (Chapters 4, 5, and 6), I introduce and experimentally demonstrate three novel genetic programming techniques for representing and evolving more responsive and adaptive (*i.e.*, plastic) computer programs: signal-driven genetic programs (SignalGP), tag-based genetic regulation, and tag-accessed memory.

1.4.1 Part 1. Understanding the evolutionary origins and consequences of adaptive phenotypic plasticity in fluctuating environments

Chapter 2 focuses on the step-by-step process by which adaptive phenotypic plasticity evolves in a fluctuating environment. Many effective and innovative survival mechanisms used by natural organisms rely on the capacity for phenotypic plasticity. Understanding the evolution of phenotypic plasticity is an important step toward understanding the origins of many types of biological complexity, as well as to meeting challenges in evolutionary computation where dynamic solutions are required. In Chapter 2, I used the Avida Digital Evolution Platform to experimentally explore the selective pressures and evolutionary pathways that lead to phenotypic plasticity. I present evolved lineages wherein unconditionally expressed (non-plastic) traits tend to evolve first. Next, imprecise forms of phenotypic plasticity often appear before optimal forms finally evolve. I experimentally disallowed each of these intermediate phenotypes to test their importance. I found that phenotypic plasticity is most likely to evolve when both unconditional trait expression and sub-optimal forms of plasticity are allowed to evolve first. I also show that both mutation rate and environmental change rate influence the evolution and maintenance of adaptive phenotypic plasticity.

In **Chapter 3**, I used Avida to investigate how the evolution of adaptive phenotypic

plasticity alters evolutionary dynamics and influences evolutionary outcomes in cyclically changing environments. Specifically, I (1) examined the evolutionary histories of plastic and non-plastic populations to test whether the evolution of adaptive plasticity promotes or constrains subsequent evolutionary change; (2) evaluated how adaptive plasticity influences fitness landscape exploration and exploitation by testing whether plastic populations are better able to evolve and then maintain novel traits; and (3) tested if the evolution of adaptive plasticity increases the potential for deleterious mutations to accumulate in evolving genomes. I found that populations with adaptive phenotypic plasticity evolve more slowly than non-plastic populations, which rely on genetic variation from *de novo* mutations to continuously re-adapt to the environment. The non-plastic populations undergo more frequent selective sweeps and accumulate many more genetic changes. I find that phenotypic plasticity stabilizes populations against environmental fluctuations; whereas the repeated selective sweeps in non-plastic populations drive the loss of beneficial traits and accumulation of deleterious mutations via genetic hitchhiking. As such, plastic populations are more likely to retain novel adaptive traits than their non-plastic counterparts. My findings suggest that the stabilizing effect of adaptive phenotypic plasticity plays an important role in subsequent adaptive evolution. Indeed, evolutionary dynamics in adaptively plastic populations was more similar to that of populations evolving in a static environment than to that of non-plastic populations evolving in an identical fluctuating environment.

1.4.2 Part 2. Building more responsive program representations

In traditional digital evolution systems (*e.g.*, Tierra and Avida), genetic programs—linear sequences of program instructions—are expressed procedurally: actions are performed sequentially, and programs must explicitly check for new sensory information before they can react. These linear program representations are convenient for their simplicity to analyze, but do not easily support the evolution of modularized responses to environmental signals that can be dynamically regulated over the organism's lifetime. This shortcoming

holds conventional digital organisms back as model systems for studying the evolution of complex forms of phenotypic plasticity. Likewise, it also limits conventional linear genetic programming systems from evolving modular programs capable of dynamically regulating responses to inputs over time.

In Chapters 4, 5, and 6, we introduce novel genetic programming techniques that both improve the problem-solving potential of genetic programming systems and provide new forms of model digital organisms for *in silico* experimental evolution. This work helps both digital evolution and genetic programming systems realize a richer spectrum of evolutionary outcomes that more closely rivals that of biological evolution.

In **Chapter 4**, I present SignalGP, a new genetic programming technique designed to incorporate the event-driven programming paradigm into computational evolution's toolbox. Event-driven programming is a software design philosophy that simplifies the development of reactive programs by automatically triggering program modules (event-handlers) in response to external events, such as signals from the environment or messages from other programs. I demonstrate the value of the event-driven paradigm using two distinct test problems (an environment coordination problem and a distributed leader election problem) by comparing SignalGP to variants that are otherwise identical, but must actively query sensors to process events or messages. In each of these problems, responsiveness to the environment or other agents is critical for maximizing fitness. I also discuss ways in which SignalGP can be generalized beyond a linear GP context.

In **Chapter 5**, I introduce and experimentally demonstrate tag-based genetic regulation, a new genetic programming technique that allows programs to dynamically adjust which code modules to express. This extension allows evolution to structure a program as a gene regulatory network where modules can be made more or less accessible based on instruction executions. I 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 (*i.e.*, current context). Indeed,

some context-dependent problems were unable to be solved by the system until regulation was added. I also identify scenarios where the correct response to a particular input never changes, rendering tag-based regulation an unneeded functionality that can 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.

Finally, in **Chapter 6**, I briefly demonstrate the use of tags to label memory positions in GP, enabling programs to define and use evolvable variable names (Lalejini and Ofria, 2019a). My tag-based memory implementation did not substantively affect problem-solving performance across several program synthesis benchmark problems. However, tag-based addressing features a larger addressable memory space than more traditional register-based memory approaches in GP. Further, in combination with tag-based regulation, tag-accessed memory has the potential to enable more dynamic, context-dependent memory storage in GP.

Chapter 2

The evolutionary origins of phenotypic plasticity

Authors: Alexander Lalejini and Charles Ofria

This chapter is adapted from (Lalejini and Ofria, 2016), which underwent peer review and appeared in the proceedings of the 2016 Artificial Life Conference.

2.1 Introduction

Phenotypic plasticity is the capacity for a genotype to express different phenotypes in response to different environmental conditions (Ghalambor et al., 2010) and is ubiquitous throughout nature. Phenotypic plasticity is central to many complex traits and developmental patterns found in nature and often serves as a key strategy employed by organisms to respond to spatially and temporally variable environments (Bradshaw, 1965; Murren et al., 2015). For example, *Daphnia pulex* use plasticity to differentially invest in morphological defenses during development depending on the presence of predators in their local environment (Black and Dodson, 1990). Genetically homogeneous cells in a developing multicellular organism rely on phenotypic plasticity to coordinate their expression patterns through environmental signals (Schlichting, 2003). Indeed, understanding the evolution of plasticity is an important step toward a deeper understanding of biological complexity.

Phenotypic plasticity also has practical applications in the field of evolutionary computa-

tion where evolution by natural selection is harnessed to solve challenging computational and engineering problems. In many realistic problem domains, conditions are noisy or cyclically change. Plasticity enables solutions to dynamically respond to changing problem conditions and be robust to noise. Both the biological and evolutionary computation domains motivate the following questions: (1) Under what conditions does phenotypic plasticity evolve? And (2), what are the evolutionary stepping stones for phenotypic plasticity?

Ghalambor *et al.* identify four conditions that are necessary for phenotypic plasticity to evolve: (1) populations are exposed to temporally or spatially varying environments, (2) the environments are differentiable by reliable signals, (3) different environments favor different phenotypes, and (4) no single phenotype can exhibit high fitness across all environments (Ghalambor et al., 2010). Theoretical and empirical findings support that phenotypic plasticity can evolve under these conditions in both natural and artificial systems (Clune et al., 2007; Goldsby et al., 2010, 2014; Hallsson and Björklund, 2012; Nolfi et al., 1994).

In addition to exploring the conditions that facilitate the evolutionary origin of phenotypic plasticity, it is also important to explore the step-by-step process by which plasticity actually evolves. What are the reoccurring themes as evolution progresses toward more plastic strategies? Are there genotypic or phenotypic patterns present in lineages leading to phenotypically plastic organisms? These types of questions are especially difficult to address in laboratory systems due to the slow pace of natural evolution, imperfections in lineage tracking, and the difficulty of acquiring high-resolution data on genotypes and phenotypes. Artificial life systems, however, are well-suited for observing and analyzing the process by which phenotypic plasticity evolves.

Here, we use the Avida Digital Evolution Platform (Ofria et al., 2009) to explore the process by which phenotypic plasticity evolves in a fluctuating environment. First, we investigate how environmental factors impact the evolution of phenotypic plasticity. Specifically, we evaluate how mutation rate and environment fluctuation rate affect the evolution of adaptive plasticity. Next, we identify the evolutionary stepping stones in the evolution of

adaptive phenotypic plasticity: do digital organisms evolve to express traits unconditionally before evolving to conditionally express them as a function of their environment, and do sub-optimal forms of plasticity evolve before more optimal forms of plasticity? We empirically test whether such intermediate evolutionary stepping stones are important to the evolution of adaptive plasticity by experimentally disallowing each stepping stone from evolving. Finally, we examine alternative evolutionary strategies to phenotypic plasticity in fluctuating environments, and we find evidence for bet-hedging strategies that use mutationally induced phenotype switching as a substitute for sensory-dependent plasticity.

2.2 Methods

2.2.1 The Avida Digital Evolution Platform

The Avida software platform provides a computational instance of evolution and enables researchers to experimentally test hypotheses about evolution that would otherwise be difficult or impossible to test in natural systems (Ofria et al., 2009). Here, we provide a brief overview of Avida as it is relevant to this work. For a more detailed description of the Avida software platform, see (Ofria et al., 2009).

Digital Organisms

Populations in Avida are made up of self-replicating computer programs (digital organisms) that compete for space in a finite toroidal grid. Each digital organism is defined by a sequence of instructions (*i.e.*, its genotype), virtual hardware to execute the instructions, and a position on the grid. The instruction set of Avida is Turing-Complete and enables organisms to perform basic computations, control their own execution flow, and replicate. An organism's virtual hardware (Figure 2.1) includes components such as a central processing unit (CPU), registers used for computation, input and output buffers, and memory stacks. Organisms replicate asexually by copying themselves line-by-line and dividing; however, an organism's copy instruction is imperfect, which can result in mutated offspring.

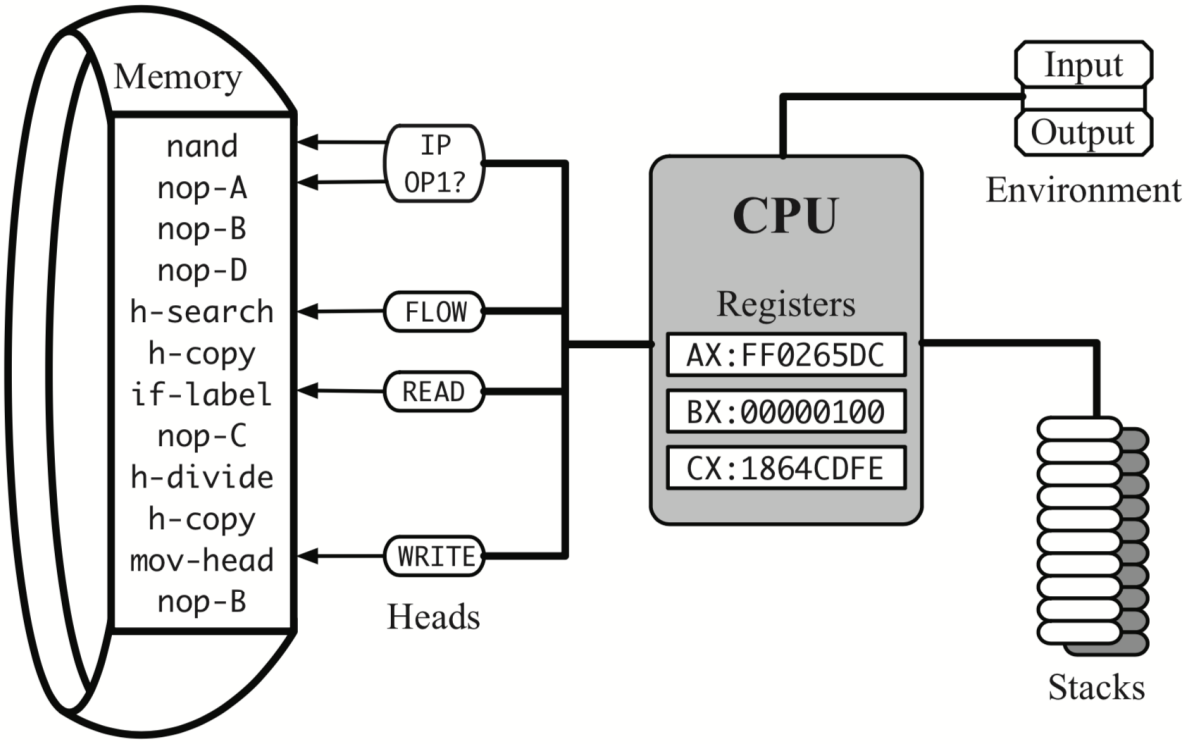


Figure 2.1: **A visual representation of the default virtual hardware used by organisms in Avida.** Original figure from (Ofria et al., 2009).

Organisms can gain additional CPU cycles by performing tasks—such as mathematical computations—to improve their metabolic rate. An organism's metabolic rate determines how rapidly it can execute its genome; a higher metabolic rate allows an organism to replicate faster. Initially, an organism's metabolic rate is roughly proportional to its genome length; however, when an organism completes designated tasks, the organism's metabolic rate can be adjusted. In this way, we can differentially reward or punish the performance of different tasks.

When an organism successfully replicates, its offspring is placed in a random location in the world, replacing the organism formerly occupying that location. In this way, becoming a more efficient replicator in Avida is advantageous in the competition for space. The combination of competition for replication efficiency and heritable variation due to imperfect copying during the replication process results in evolution by natural selection.

Sensing in Avida

In a typical Avida run, organisms must execute an instruction called `IO` to output the result of a computation. That output is analyzed to determine if any tasks have been performed, and if so, the organism is appropriately rewarded or punished. However, in this default scenario, organisms cannot sense the result, even after the task has been performed. To provide organisms with a mechanism to sense their environment, we added an `IO-Sense` instruction to the set of available instructions[1].

The `IO-Sense` instruction simulates `IO` and provides the organism with feedback on what would have happened if the organism had executed an `IO` instruction instead. This separation of `IO` performance and sensing allows organisms to determine whether or not a particular task is being punished without the risk of punishment, lowering the potential cost of sensing. If an `IO` operation would have resulted in a punishment, a -1 is added to the top of the organism's stack memory; if it would have resulted in a reward, a 1 is placed there. If an `IO` operation would have resulted in neither a reward nor a punishment, a 0 is placed on the organism's stack memory. In this way, organisms can sense whether or not a particular task is being rewarded or punished in their current environment and then react accordingly.

Identifying Phenotypic Plasticity in Avida

We define a phenotypically plastic organism in Avida as an organism that leverages sensory information to alter their phenotype based on the environment. We restrict the definition of an organism's phenotype to the set of unique tasks it performs in the given environment. We do not consider how many times an organism performs a particular task in a given environment, but only whether the organism does the task at all. Thus, to be phenotypically plastic, an organism must express a different task profile (*i.e.*, perform different tasks) in different environments.

[1] `IO-Sense` is based on the `IO-Feedback` instruction implemented in (Clune et al., 2007), which worked exactly as the default `IO` instruction, but provided the organism with feedback on the result. As such, an organism must first do a particular task once—and potentially get punished—to sense whether or not the task is beneficial with the `IO-Feedback` instruction.

2.2.2 Experimental Design

To explore the evolutionary history of phenotypically plastic organisms, we used an experimental design based on (Clune et al., 2007).

Environments

We constructed two experimental environments named ENV-NAND and ENV-NOT. In ENV-NAND, organisms were rewarded for performing the NAND logical task but were punished for performing the NOT logical task. Conversely, in ENV-NOT, organisms were rewarded for performing the NOT logical task but were punished for performing the NAND logical task. In each of our experimental treatments, we cycled between these two environmental conditions. In this way, genotypes with the capacity to sense the current environment and express the appropriate task had a competitive advantage over non-plastic organisms.

Phenotypes

Given our simple definition of a phenotype, there are only four possible phenotypes in each of the two previously described environments: (1) perform only NAND, (2) perform only NOT, (3) perform both NAND and NOT, and (4) perform neither NAND nor NOT. When considering an organism's phenotype across both ENV-NAND and ENV-NOT, there are sixteen possible combinations. We enumerate these phenotypes in Figure 2.2. Of these sixteen possible phenotypes, only four express the identical task profile in both environments; the other twelve profiles all exhibit some form of plasticity. The optimal form of plasticity is to perform only the NAND task in ENV-NAND and to perform only the NOT task in ENV-NOT; any other form of plasticity is sub-optimal. There are five possible phenotypes that leverage plasticity to perform punished tasks instead of rewarded tasks in a given environment; we did not expect these maladaptive forms of phenotypic plasticity to be successful.

#	Task Profile				Color Code	Type of Plasticity
	ENV-NAND		ENV-NOT			
	NAND	NOT	NAND	NOT		
1	–	–	–	–		Non-plastic
2	X	–	X	–		
3	–	X	–	X		
4	X	X	X	X		
5	–	–	–	X		Actively Beneficial
6	X	–	–	–		
7	X	X	–	X		
8	X	–	X	X		
9	X	–	–	X		Optimal
10	X	X	–	–		Neutral
11	–	–	X	X		
12	–	X	–	–		Actively Harmful
13	–	–	X	–		
14	–	X	X	X		
15	X	X	X	–		
16	–	X	X	–		

Figure 2.2: **Enumeration of all possible complete phenotypes.** Each row represents a distinct phenotype. An 'X' indicates that the associated task is performed in the specified environment, while a '–' indicates that the task is not performed. For each environment, the column of the rewarded task is highlighted in light purple, and the column of the punished task is highlighted in light orange. An 'X' in a reward column or a '–' in the punished column is optimal. Each phenotype has a color code, which is used in our lineage visualizations. Note that the first four rows are non-plastic phenotypes, rows 5–8 exhibit partially beneficial plasticity, and row 9 is optimally beneficial. Rows 10–11 are neutral non-adaptive plasticity, while rows 12–16 are detrimental forms of plasticity.

Treatments

Our experimental design consisted of five treatments and a control: (1) a baseline treatment with a moderate point-mutation rate and environmental-cycle length, (2) a low-mutation-rate treatment, (3) a high-mutation-rate treatment, (4) a short-environment-cycle-length treatment, (5) a long-environment-cycle-length treatment, and (6) a control where both NAND and NOT were rewarded and the environment did not fluctuate. See Table 2.1 for treatment details.

Treatment	Point-mutation Rate	Environment Cycle Length
Baseline	0.0075	100 updates
Low Mutation Rate	0.0025	100 updates
High Mutation Rate	0.0125	100 updates
Short Environment Cycle Length	0.0075	50 updates
Long Environment Cycle Length	0.0075	200 updates

Table 2.1: **Differences among the five experimental treatments.** Point-mutation rate is given as mutations per instruction copied. Environment cycle length describes the length of time (in updates) an environment is active before toggling to the alternative environment.

We created the baseline treatment to produce phenotypically plastic organisms for lineage analysis. We limited the population size to 3600 organisms and seeded the world with an ancestral genotype capable only of self-replication. We then evolved populations for 100,000 updates[2] in Avida. We imposed a 0.0075 probability of point-mutation per instruction copied, as well as a 0.05 probability for each of single-instruction insertion and deletion per genome copied. We fluctuated the current environment between ENV-NAND and ENV-NOT every 100 updates in the baseline treatment. We ran 50 replicates of each treatment, including the control.

Lineage Visualization

To explore evolutionary strategies evolved in fluctuating environments, we visualized the lineages of evolved genotypes as vertical bars where time (in updates) proceeds from top to bottom beginning with the lineage's original ancestor genotype. Any given genotype on the lineage must express one of the sixteen possible phenotypes enumerated in Figure 2.2. At each point in time, the color of the visualized lineage corresponds to the color representing the phenotype expressed by the lineage at that point in time. For example, because the

[2]An update in Avida is an experimental length of time. One update is defined as the amount of time it takes for the average organism to execute 30 instructions (see (Ofria et al., 2009) for more details).

ancestral organism is capable only of self-replication, all visualized lineages should show that the original ancestor's phenotype performed neither the NAND task nor the NOT task. In addition to the visualized lineages, we indicate the actual environmental conditions experienced by the evolving populations at each point in time by the color of the vertical axis. This type of visualization allows us to display the phenotypic states traversed by any given lineage.

2.3 Results and Discussion

2.3.1 What conditions promote the evolution of phenotypic plasticity?

Ghalambor *et al.* identified four environmentally-dependent requirements for the evolution of phenotypic plasticity (Ghalambor et al., 2010). Our experimental design conforms to these conditions, enabling us to test their validity and relative importance. The oscillation between ENV-NAND and ENV-NOT provides temporal variation. The IO-Sense instruction reliably indicates the current environment. The two environments favor opposing phenotypic traits, and the only way for an individual organism to achieve a high fitness in both environments is to alter its phenotypic expression. Given the existing theoretical and empirical support for these conditions, we expected to see the evolution of phenotypic plasticity in each of our experimental treatments. However, we were unsure of the impact of altering environmental factors such as mutation rate and environment fluctuation rate.

At the end of the experiment, we extracted the dominant (most abundant) genotype from the population of each replicate. We tested these genotypes in both ENV-NAND and ENV-NOT and recorded each genotype's expressed phenotype across both environments. In Table 2.2, we report the number of replicates in which the dominant genotype at the end of the experiment was plastic and the number of replicates in which the dominant genotype was optimally plastic. Note that for these results we only evaluated the most abundant genotype at the end of the experiment. An ancestor of the evaluated genotype may have been plastic,

Treatment	Plastic Replicates		Unconditional Precedes Conditional		Sub-optimal Precedes Optimal
	Total	Optimal*	NAND Task	NOT Task	
Baseline	31 (62%)	17 (34%)	31 (100%)	28 (90.3%)	16 (94.1%)
Low Mutation Rate	38 (76%)	30 (60%)	34 (89.5%)	35 (92.1%)	30 (100%)
High Mutation Rate	25 (50%)	11 (22%)	25 (100%)	24 (96%)	10 (90.9%)
Short Environment Cycle Length	36 (72%)	18 (36%)	33 (91.7%)	28 (77.8%)	18 (100%)
Long Environment Cycle Length	16 (32%)	10 (20%)	14 (87.5%)	16 (100%)	9 (90%)
Control	0 (0%)	0 (0%)	–	–	–

*Optimal is defined as the complete phenotype that only performs the rewarded task in each environment.

Table 2.2: **A summary of evolutionary outcomes across all five experimental treatments and control.** "Plastic Replicates" indicates the number of replicates (out of 50 per treatment) in which the final dominant genotype was plastic at all ("Total") and perfectly plastic ("Optimal"). "Unconditional Precedes Conditional" indicates the number of times the NAND task and NOT task were expressed unconditionally before eventually evolving to be express conditionally (out of total plastic). Finally, "Sub-optimal Precedes Optimal" indicates how many runs had an imperfect form of plasticity before eventually evolving to be optimally plastic (out of total optimally plastic).

but if that plasticity was not maintained in the lineage, we did not count it in Table 2.2.

As expected, the capacity for phenotypic plasticity evolved in each experimental treatment; in 31 of the 50 baseline treatment replicates, phenotypic plasticity was present in the final dominant organism. None of the final dominant genotypes from the control replicates were phenotypically plastic. In all control replicates, the dominant genotype performed both the NAND and NOT tasks unconditionally. Our results are consistent with existing theoretical and empirical work, supporting the validity of the conditions likely to facilitate the evolution of phenotypic plasticity (Clune et al., 2007; Ghalambor et al., 2010; Hallsson and Björklund, 2012; Nolfi et al., 1994).

2.3.2 How do environmental factors impact the evolution of phenotypic plasticity?

While our results show phenotypic plasticity can evolve under the conditions identified in (Ghalambor et al., 2010), how do mutation rate and fluctuation rate affect the evolution of phenotypic plasticity under these conditions? We found compelling results for both mutation rate and environmental cycle length.

Mutation Rate

While only of borderline statistical significance ($p = 0.058$ using Fisher's Exact Test with Bonferroni corrections for multiple comparisons; all statistics were done in R version 3.2.2 (R Core Team, 2016)), our results trend such that populations at lower mutation rates appear more likely to evolve phenotypic plasticity than do populations at higher mutation rates. The most abundant genotypes exhibited some plasticity in 38/50 runs at a low mutation rate, 31/50 at the baseline mutation rate, and 25/50 and the high mutation rate. While higher mutation rates increase genetic variation from one generation to the next, most mutations that have phenotypic effects are deleterious (Sniegowski et al., 2000). Thus, at higher mutation rates, the elevated influx of deleterious mutations could increase the difficulty of maintaining the necessary genetic machinery for phenotypic plasticity. Qualitative evidence for this effect can be seen in the time-sliced visualized lineages of final dominant, non-plastic genotypes from the high-mutation-rate treatment (Figure 2.3) where lineages traverse states of plasticity for some time before reverting back to states of non-plasticity[3]. Furthermore, more phenotypic shifts in general increase the probability of quickly finding an appropriate non-plastic phenotype after each environmental change.

Environment Fluctuation Rate

We found a significant difference ($p = 0.00028$ using Fisher's Exact Test with Bonferroni corrections for multiple comparisons) as we varied the cycle length for environmental

[3]For fully interactive visualizations of evolved lineages from all treatments, see `https://lalejini.com/evo-origins-of-phenotypic-plasticity-web/`

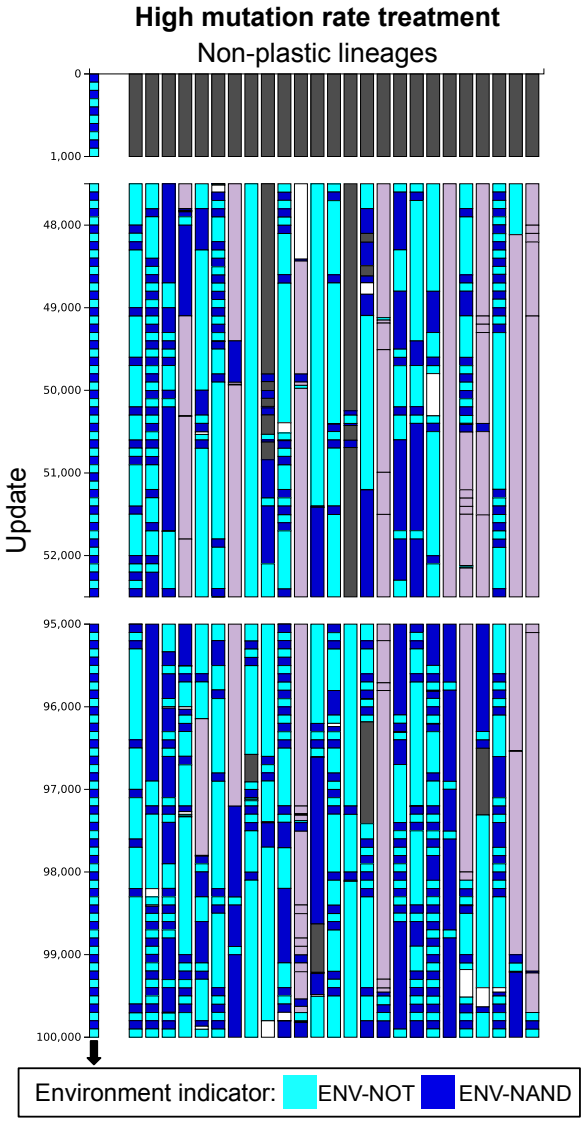


Figure 2.3: **Time-sliced visualization of lineages for non-plastic, dominant genotypes from the high-mutation-rate treatment.** Abbreviated color reference: cyan represents unconditional NOT task performance, dark blue represents unconditional NAND task performance, and light purple represents sub-optimal forms of plasticity. Refer to Figure 2.2 for a full legend of phenotype colors.

switching. Specifically, in the long-environment-cycle-length, only 16/50 runs ended with a final dominant genotype that was phenotypically plastic, while the baseline and short-environment-cycle-length produced 31 and 36 plastic outcomes, respectively.

We expect that the short-environment-cycle-length treatment is biased toward the evolution of phenotypic plasticity because of the rapid environment fluctuations relative to

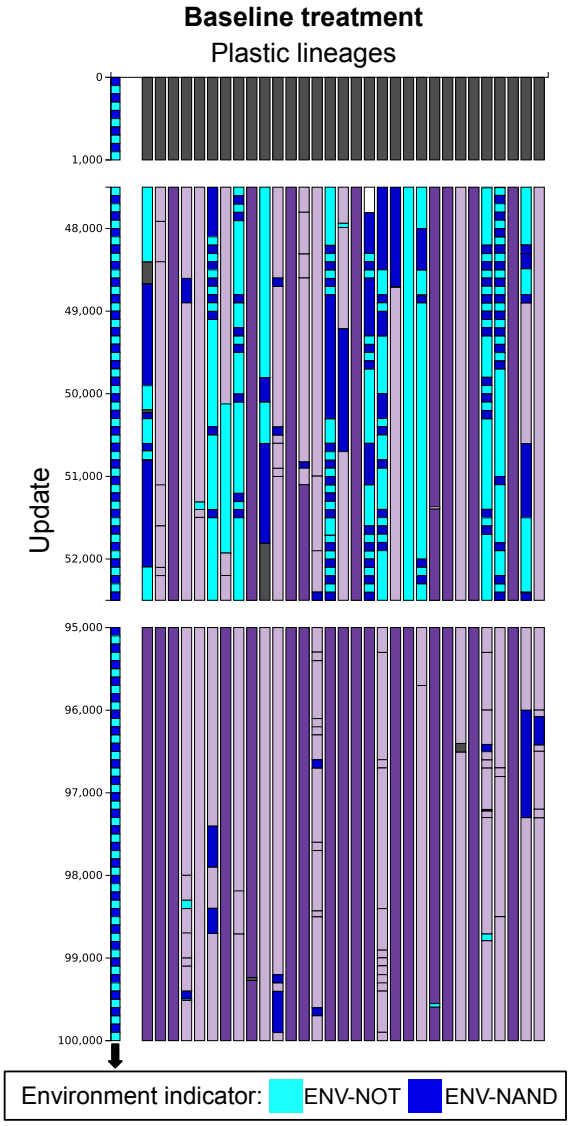


Figure 2.4: **Time-sliced lineage visualization of dominant, plastic genotypes from the baseline treatment.** Abbreviated color reference: cyan represents unconditional NOT task performance, dark blue represents unconditional NAND task performance, light purple represents sub-optimal forms of plasticity, and dark purple represents optimal plasticity. Refer to Figure 2.2 for a full legend of phenotype colors.

other experimental treatments. Rapid fluctuations cause lineages to be less able to rely on mutational input for adaptation. In the long-environment-cycle-length treatment, environmental fluctuations may not occur rapidly enough to produce a sufficient selective pressure for phenotypic plasticity, allowing alternative adaptive strategies to evolve instead.

2.3.3 What are the evolutionary stepping stones for phenotypic plasticity?

In an attempt to identify patterns frequently encountered during the evolution of phenotypically plastic organisms, we extracted and analyzed the full lineages from our experiments. We tested each ancestor genotype in both ENV-NAND and ENV-NOT and classified their phenotype across both environments. In addition to a quantitative analysis, we also visualized the lineages of the dominant, plastic genotypes; see Figure 2.4 for the visualization of the baseline treatment. Using our visualizations and ancestor phenotype classifications, we addressed the following two questions: (1) Do the lineages of phenotypically plastic organisms first evolve to perform tasks unconditionally before evolving to perform them conditionally as a function of their current environment? And (2), do imperfect forms of phenotypic plasticity tend to precede optimal forms?

Unconditional task performance precedes plasticity

To explore whether or not unconditional task performance was an evolutionary stepping stone for conditional task performance (*i.e.*, phenotypic plasticity), we determined whether a task was performed unconditionally prior to being performed conditionally by the ancestors of plastic genotypes. We analyzed both tasks (NAND and NOT) separately. These results are reported in Table 2.2. Across all experimental treatments, non-plastic ancestors generally preceded plastic ancestors. In other words, unconditional task performance of the NAND and NOT tasks generally preceded the conditional performance of either task. Examples of this can be seen in time-sliced plastic lineages from the baseline treatment (Figure 2.4) where many lineages maintain states of unconditional task expression prior to entering states of conditional task expression. These results suggest that, in fluctuating environments similar to those in our experiment, the evolutionary path to phenotypic plasticity usually traverses states of unconditional trait expression prior to entering states of conditional trait expression. This result should be unsurprising. In order to evolve a regulated function, the capacity for

both the regulation and the function must exist. In our experiment, the function can be selected for without regulation; however, regulation of the function is unlikely to be selected for without the prior capacity for the function.

Sub-optimal plasticity precedes optimal plasticity

To investigate sub-optimal phenotypic plasticity as an evolutionary stepping stone for optimal phenotypic plasticity in our experiment, we analyzed lineages of optimally plastic genotypes. Again, we consider only complete phenotypes that exclusively perform the rewarded task in each environment to be optimal. For each optimally plastic genotype's lineage, we determined whether or not the evolution of optimal plasticity was preceded by the evolution of sub-optimal phenotypic plasticity. The results of this analysis are reported in Table 2.2.

Across all experimental treatments, the evolution of sub-optimal plasticity did, indeed, generally precede the evolution of optimal phenotypic plasticity. Examples of sub-optimal plasticity preceding more optimal forms of plasticity can be seen in some of the time-sliced lineages from the baseline treatment visualized in Figure 2.4. These results suggest that, in fluctuating environments similar to those in our experiment, sub-optimal forms of phenotypic plasticity tend to arise before the evolution of optimal forms of phenotypic plasticity.

Unconditional trait expression tends to evolve first; then, sub-optimal forms of plasticity appear before optimal forms finally evolve. While challenging to verify, we expect our results to be applicable to biological systems. The evolution of complex functions (*e.g.*, optimal phenotypic plasticity) build on simpler, previously evolved functions (*e.g.*, unregulated or sub-optimally regulated functions) (Lenski et al., 2003). These results, however, are particularly useful for applied evolutionary computation. If an evolved problem solution must respond dynamically to environmental variables, it is likely that the solution will need to be able to traverse through states of rigidity and sub-optimal plasticity prior to reaching a state of optimal plasticity. Thus, first evolving rigid solutions in fixed environments and

then gradually starting to fluctuate more aspects of the environment over time could provide a scaffolding for the evolution of optimally plastic solutions.

2.3.4 Does plasticity still evolve when evolutionary stepping stones are disallowed?

We conducted a series of followup experiments to investigate the importance of unconditional trait expression and sub-optimal plasticity as evolutionary stepping stones. We evolved 200 replicate populations under baseline treatment conditions (described in Section 2.2.2) and 200 replicate populations in each of three experimental conditions where we disallowed particular phenotypic profiles from evolving: (1) we disallowed phenotypes that expressed NAND and/or NOT unconditionally (*i.e.*, task profiles 2, 3, and 4 from Figure 2.2); (2) we disallowed sub-optimally plastic phenotypes (*i.e.*, task profiles 5 through 8 and 10 through 16 from Figure 2.2); and, (3) we disallowed phenotypes that exhibited unconditional trait expression *or* phenotypes that were sub-optimally plastic (*i.e.*, all task profiles except 1 and 9 from Figure 2.2). Note that in each of these experimental treatments, we always allowed phenotypes that expressed no tasks or were optimally plastic across both environments. In treatments where particular phenotypes were disallowed, we tested all offspring in both ENV-NAND and ENV-NOT; if the phenotype of an organism's offspring was among the disallowed phenotypes, we prevented its birth. As in our previous experiments, we counted the number of replicates of each treatment where the dominant genotype at the end of the experiment exhibited a plastic phenotype.

Figure 2.5 gives the number of plastic replicates that evolved in each experimental condition. We compared each of the three treatments that disallowed stepping stone phenotypes to our unmodified baseline treatment (Fisher's exact test with a significance level of 0.05 and a Bonferonni correction for multiple comparisons). Each of the three treatments where we disallowed offspring with particular phenotypes had significantly fewer replicates with a plastic dominant genotype at the end of the experiment (unconditional trait expression

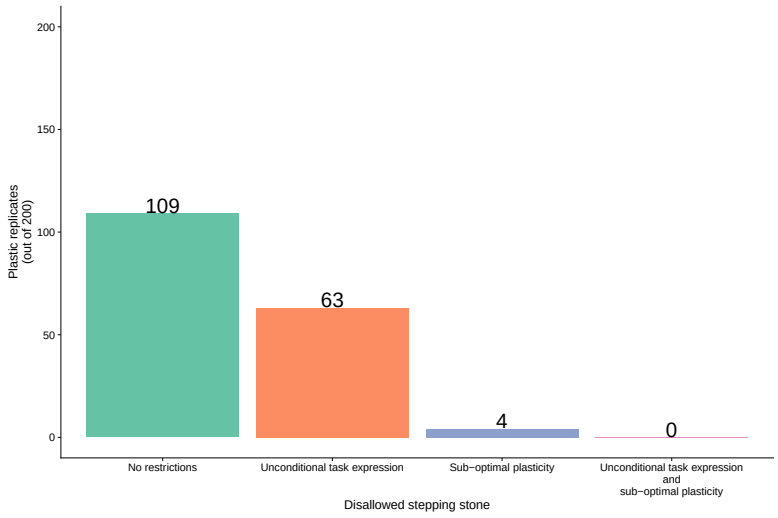


Figure 2.5: **Blocked stepping stone evolutionary outcomes.** For each condition, the bar plot indicates the number of replicates (out of 200 per condition) where the final dominant genotype was plastic.

disallowed: $p < 10^{-4}$; sub-optimal plasticity disallowed: $p < 10^{-4}$; both unconditional trait expression and sub-optimal plasticity disallowed: $p < 10^{-4}$).

No phenotypically plastic genotypes evolved when we disallowed all intermediate phenotypes; when all intermediate phenotypes were disallowed, only two phenotypes were possible: (1) performing neither the NOT nor NAND tasks across environments and (2) optimally regulating between the NOT and NAND tasks across environments. For plasticity to evolve without allowing evolution to traverse intermediate stepping stones, optimal plasticity would need arise in a single mutational step from a genotype that performed neither the NAND nor NOT tasks. This result demonstrates that, together, these intermediate phenotypes represent crucial building blocks for the evolution of phenotypic plasticity.

When we prevented genotypes that perform tasks unconditionally across environments, plasticity evolved less frequently than in treatments where we placed no restrictions on phenotypes; this supports our previous observation that unconditional task expression is a stepping stone toward plastic task expression. However, many replicates (63 out of 200) where unconditional task expression was disallowed still yielded plastic organisms. Thus, while unconditional trait expression is likely a valuable building block in the evolution of

phenotypic plasticity, it is not necessary. Our results indicate that sub-optimal plasticity is a more important building block for optimal plasticity than unconditional trait expression. Only 4 out of 200 replicates where we disallowed sub-optimally plastic phenotypes yielded optimal plasticity. This is not unsurprising, as a lineage would need to move from a state of unconditional task expression to perfect task regulation in a single mutational step.

2.3.5 Are stochastic strategies evolving as an alternative to phenotypic plasticity?

Stochastic phenotype switching, a form of bet hedging (Seger and Brockmann, 1987), is a common strategy leveraged by bacteria in fluctuating environments (Rainey et al., 2011). Some forms of stochastic phenotype switching rely on mutational input to induce phenotypic changes. This strategy is thought to be a viable alternative to phenotypic plasticity in the absence of reliable environmental signals or when the processing of sensory information is costly (Rainey et al., 2011). Strategic stochastic phenotype switching often relies on contingency loci, which are hypermutable regions of the genome that can induce phenotype switching via mutational input (Moxon et al., 2006).

We hypothesized that stochastic phenotype switching was an alternative evolutionary strategy to phenotypic plasticity because of its commonality in bacteria. We most expected to see stochastic phenotype switching in our experimental treatments where the fewest number of replicates produced phenotypically plastic final dominant genotypes.

Lineage Visualization

It can be difficult to intuitively understand evolutionary strategies leveraged by a lineage without a visual aid. To explore evolutionary strategies alternative to phenotypic plasticity in fluctuating environments, we visualized the lineages of dominant, non-plastic genotypes from our experimental treatments.

If a lineage relied on stochastic phenotype switching, we would expect it to switch between phenotypic states of unconditional NAND task performance and unconditional NOT

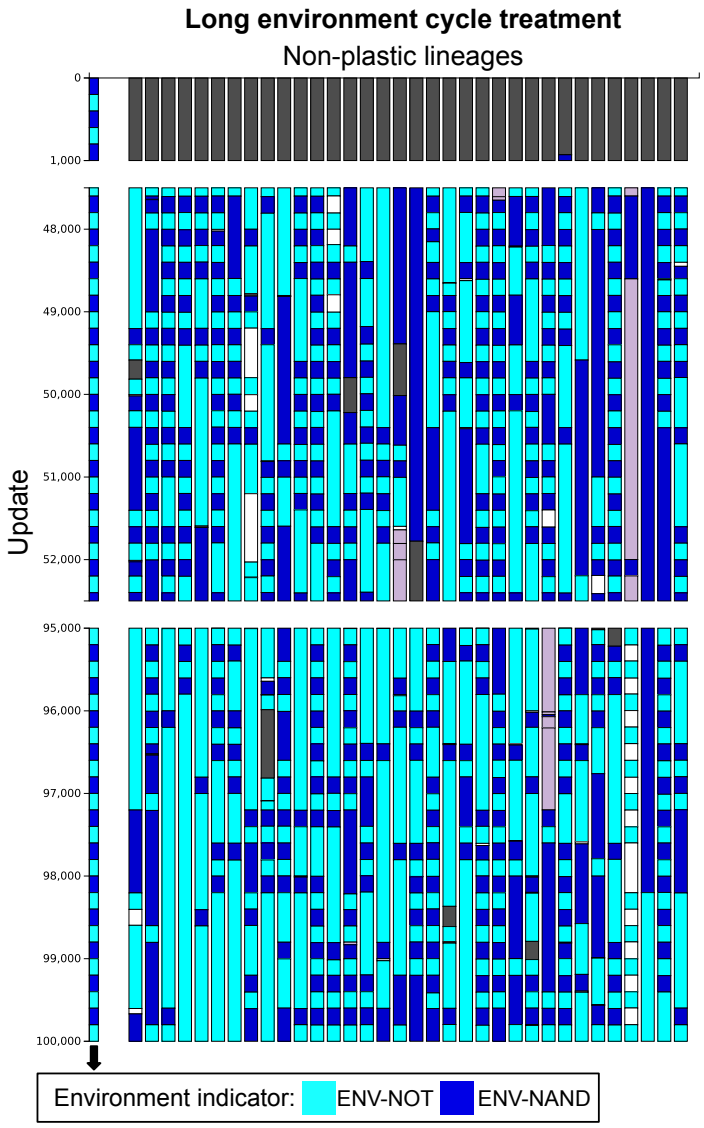


Figure 2.6: **Time-sliced lineage visualization of non-plastic, dominant genotypes from the long environment cycle treatment.** Abbreviated color reference: cyan represents unconditional NOT task performance, dark blue represents unconditional NAND task performance, light purple represents sub-optimal forms of plasticity, and dark purple represents optimal plasticity. Refer to Figure 2.2 for a full legend of phenotype colors.

task performance in approximate synchronization with the changing environment. Specifically, we should see ancestors along a lineage perform NAND unconditionally during periods of ENV-NAND and see ancestors performing NOT unconditionally during periods of ENV-NOT. We show a time-sliced lineage visualization of dominant, non-plastic genotypes at the end of our experiment for the long-environment-cycle-length treatment (Figure 2.6).

From Figure 2.6, we see what appear to be cases of stochastic phenotype switching. That is, we observe lineages switching between phenotypic states of unconditional NAND task performance and unconditional NOT task performance in approximate synchronization with the environment. Many of the lineages in the long-environment-cycle treatment seem to be undergoing stochastic phenotype switching. A few examples of what appear to be stochastic phenotype switching can even be seen in Figure 2.4 (the plastic lineages from our baseline treatment) between updates 47,500 and 52,500 (the middle time-slice), prompting the following open question: in addition to being an alternative strategy to plasticity in fluctuating environments, could stochastic phenotype switching also act as a precursor or building block toward plasticity?

Our visualizations only provide an exploratory method for understanding evolutionary strategies employed by a lineage. Further analysis would be required to confirm or reject our hypothesis that stochastic phenotype switching is evolving as an alternative strategy to phenotypic plasticity in our system. This hypothesis is particularly worthwhile to explore because our mutation rate was fixed across the genome, preventing the evolution of contingency loci. Furthermore, because sensing mechanisms were perfectly accurate, phenotypic plasticity was a reliable strategy. We hypothesize that genotypes are moving to a region of the mutational landscape that straddles the boundary between expressing unconditional NAND task performance and unconditional NOT task performance such that minimal mutational input is required to switch phenotypes. This type of evolutionary trajectory has been demonstrated by Crombach and Hogeweg in evolutionary simulations of simple, genome-encoded gene regulatory network models (Crombach and Hogeweg, 2008). In their simulations, Crombach and Hogeweg found that networks evolved in an oscillating environment possessed genotype to phenotype mappings that were mutationally more efficient at generating adaptive phenotypes in alternative environments.

2.4 Conclusion

In this work, we evolved populations of phenotypically plastic organisms at varied rates of environmental fluctuation and mutation using the Avida Digital Evolution Platform. We analyzed the lineages of evolved genotypes for clues about the evolutionary stepping stones toward phenotypic plasticity. We found that the capacity for phenotypic plasticity evolved under conditions identified by previous research (Clune et al., 2007; Ghalambor et al., 2010). We found evidence that traits are generally expressed unconditionally prior to the evolution of conditional trait expression and that sub-optimal forms of phenotypic plasticity generally evolve before optimal forms of phenotypic plasticity. Both of these results are examples of evolution's use of simpler functions as building blocks for more complex functions as in Lenski et al. (2003).

Visual inspection of the evolutionary histories leading to phenotypically plastic organisms suggests that under certain conditions stochastic phenotype switching evolves as an alternative strategy to phenotypic plasticity, just as it does in many bacteria (Moxon et al., 2006; Rainey et al., 2011). Of course, in these bacterial cases, hypermutable sites tend to appear in the genomes (called "contingency loci") that facilitate such task switching.

Given these promising results, we plan to explore whether stochastic phenotype switching can be a viable evolutionary strategy in the absence of the ability to evolve hypermutable regions of the genome. Given the potential difficulty in maintaining the necessary genetic machinery associated with phenotypic plasticity, are there cases in which stochastic phenotype switching is more robust than phenotypic plasticity? And, does this contribute to the evolution of stochastic phenotype switching as an evolutionary strategy? Metrics are clearly needed for quantifying stochastic phenotype switching in digital systems and for evaluating the mutational landscapes of genotypes along a lineage.

Chapter 3

The Evolutionary Consequences of Adaptive Phenotypic Plasticity

Authors: Alexander Lalejini, Austin J. Ferguson, Nkrumah A. Grant, and Charles Ofria
This chapter is adapted from a manuscript to be submitted for peer review to Frontiers Ecology and Evolution.

3.1 Introduction

Fluctuating environmental conditions are ubiquitous in natural systems. Organisms have evolved a wide range of evolved strategies for coping with environmental change, such as phenotypic plasticity (Ghalambor et al., 2007), bet hedging (Beaumont et al., 2009), periodic migration (Winger et al., 2019), and adaptive tracking (Barrett and Schluter, 2008). The particular coping mechanisms that evolve in fluctuating environments shift the course of subsequent evolution (Wennersten and Forsman, 2012; Schaum and Collins, 2014). Identifying the mechanisms most likely to evolve and examining both the evolutionary constraints and opportunities associated with each is critical for us to understand and predict evolutionary outcomes in changing environments.

In this work, we focus on phenotypic plasticity, which can be defined as the capacity for a single genotype to alter phenotypic expression in response to a change in its environment (West-Eberhard, 2003). Phenotypic plasticity is controlled by genes whose expression

is coupled to one or more abiotic or biotic environmental signals. For example, the sex ratio of the crustacean *Gammarus duebeni* is modulated by changes in photoperiod and temperature (Dunn et al., 2005), and the reproductive output of some invertebrate species is heightened when infected with parasites to compensate for offspring loss (Chadwick and Little, 2005). In this study, we conducted digital evolution experiments to investigate how the evolution of adaptive phenotypic plasticity shifts the course of evolution in a cyclically changing environment. Specifically, we examined the effects of adaptive plasticity on subsequent genomic and phenotypic change, the capacity to evolve and then maintain novel traits, and the accumulation of deleterious alleles.

Evolutionary biologists have long been interested in how evolutionary change across generations is influenced by phenotypic plasticity because of its role in generating phenotypic variance (Gibert et al., 2019). The effects of phenotypic plasticity on adaptive evolution have been disputed, as few studies have been able to observe both the initial patterns of plasticity and the subsequent divergence of traits in natural populations (Ghalambor et al., 2007; Wund, 2012; Forsman, 2015; Ghalambor et al., 2015; Hendry, 2016). In a changing environment, adaptive phenotypic plasticity provides a mechanism for organisms to regulate trait expression within their lifetime, which can stabilize populations through changes (Gibert et al., 2019). In this context, the stabilizing effect of adaptive plasticity has been hypothesized to constrain the rate of adaptive evolution (Gupta and Lewontin, 1982; Ancel, 2000; Huey et al., 2003; Price et al., 2003; Paenke et al., 2007). That is, directional selection may be weak if environmentally-induced phenotypes are close to the optimum; as such, adaptively plastic populations may evolve slowly (relative to non-plastic populations) unless there is a substantial fitness cost to plasticity.

Phenotypic plasticity allows for the accumulation of genetic variation in genomic regions that are unexpressed under current environmental conditions. Such cryptic ("hidden") genetic variation can serve as a source of diversity in the population, upon which selection can act when the environment changes (Schlichting, 2008; Levis and Pfennig, 2016). It remains

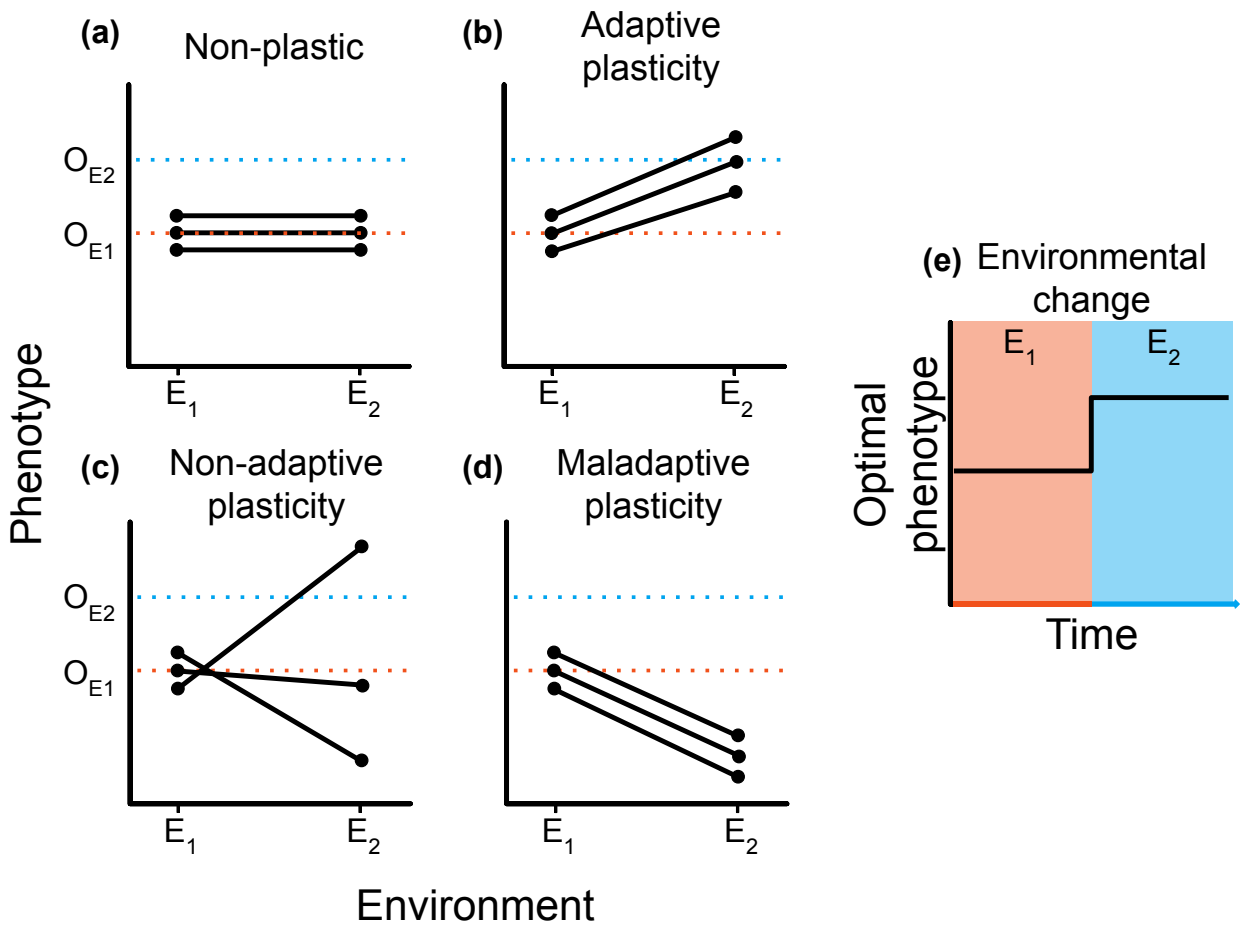


Figure 3.1: **Hypothetical reaction norms for genotypes that exhibit phenotypic variation.** (a) through (d) show four hypothetical reaction norm scenarios for the environmental change described in (e). In (e), the environment changes from E_1 (in red) to E_2 (in blue), and the optimal phenotypes for environments E_1 and E_2 are different (O_{E1} and O_{E2}, respectively). In each of the four reaction norm scenarios, populations are well-adapted to E_1. In (a), genotypes in the population are non-plastic, and as such, we would expect strong directional selection on mutations that move phenotypes toward O_{E2} after the environment changes. In (b), genotypes in the population are adaptively plastic. That is, phenotypic changes induced by the environment change to E_2 are already near the optimum, and as such, we would expect this population to remain relatively stable after the environment changes. In (c), the population exhibits non-adaptive plasticity with substantial variation in how individuals respond to the environmental change. In this case, we expect plasticity to result in a rapid evolutionary response to the change in environment. In (d), the population exhibits maladaptive plasticity relative to the given environmental change. When the environment changes, there is little variation for selection to act on, and without beneficial mutations, this population may be at risk of extinction due to their maladaptive plastic response.

unclear to what extent and under what circumstances this cryptic variation caches adaptive potential or merely accumulates deleterious alleles (Gibson and Dworkin, 2004; Paaby and Rockman, 2014; Zheng et al., 2019).

The "genes as followers" hypothesis (also known as the "plasticity first" hypothesis) predicts that phenotypic plasticity may facilitate adaptive evolutionary change by producing variants with enhanced fitness under stressful or novel conditions (West-Eberhard, 2003; Schwander and Leimar, 2011; Levis and Pfennig, 2016). Environmentally-induced trait changes can be refined through selection over time (*i.e.*, genetic accommodation). Further, selection may drive plastic phenotypes to lose their environmental dependence over time in a process known as genetic assimilation (West-Eberhard, 2005; Pigliucci, 2006; Crispo, 2007; Schlichting and Wund, 2014; Levis and Pfennig, 2016). In this way, environmentally-induced phenotypic changes can precede an evolutionary response.

Phenotypic plasticity may also "rescue" populations from extinction under changing environmental conditions by buffering populations against novel stressors. This buffer promotes stability and persistence and grants populations time to further adapt to rapidly changing environmental conditions (West-Eberhard, 2003; Chevin and Lande, 2010).

Disparate predictions about how phenotypic plasticity may shift the course of subsequent evolution are not necessarily mutually exclusive. Genetic and environmental contexts determine if and to what extent phenotypic plasticity promotes or constrains subsequent evolution. Figure 3.1 overviews how we might expect different forms of phenotypic plasticity to result in different evolutionary responses after an environmental change.

Experimental studies investigating the relationship between phenotypic plasticity and evolutionary outcomes can be challenging to conduct in natural systems. Such experiments would require the ability to irreversibly toggle plasticity followed by long periods of evolution during which detailed phenotypic data would need to be collected. Digital evolution experiments have emerged as a powerful research framework from which evolution can be studied. In digital evolution, self-replicating computer programs (digital organisms) compete for resources, mutate, and evolve following Darwinian dynamics (Wilke and Adami, 2002). Digital evolution studies balance the speed and transparency of mathematical and computational simulations with the open-ended realism of laboratory experiments. Modern computers al-

low us to observe many generations of digital evolution at tractable time scales; thousands of generations can take mere minutes as opposed to months, years, or centuries. Digital evolution systems also allow for perfect, non-invasive data tracking. Such transparency permits the tracking of complete evolutionary histories within an experiment, which circumvents the historical problem of drawing evolutionary inferences using incomplete records (from frozen samples or fossils) and extant genetic sequences. Additionally, digital evolution systems allow for experimental manipulations and analyses that go beyond what is possible in wet-lab experiments. Such analyses have included exhaustive knockouts of every site in a genome to identify the functionality of each (Lenski et al., 2003), comprehensive characterization of local mutational landscapes (Lenski et al., 1999; Canino-Koning et al., 2019), and the real-time reversion of all deleterious mutations as they occur to isolate their long-term effects on evolutionary outcomes (Covert et al., 2013). Digital evolution studies allow us to directly toggle the possibility for adaptive plastic responses to evolve, which enables us to empirically test hypotheses that were previously relegated to theoretical analyses.

In this work, we use the Avida Digital Evolution Platform (Ofria et al., 2009). Avida is an open-source system that has been used to conduct a wide range of well-regarded studies on evolutionary dynamics, including the origins of complex features (Lenski et al., 2003), the survival of the flattest effect (Wilke et al., 2001), and the origins of reproductive division of labor (Goldsby et al., 2014). Our experiments build directly on previous studies in Avida that characterized the *de novo* evolution of adaptive phenotypic plasticity (Clune et al., 2007; Lalejini and Ofria, 2016) as well as previous work investigating the evolutionary consequences of fluctuating environments for populations of non-plastic digital organisms (Li and Wilke, 2004; Canino-Koning et al., 2019). Of particular relevance, Clune et al. (2007) and Lalejini and Ofria (2016) experimentally demonstrated that adaptive phenotypic plasticity can evolve given the following four conditions (as identified by Ghalambor et al. 2010): (1) populations experience temporal environmental variation, (2) these environments are differentiable by reliable cues, (3) each environment favors different phenotypic traits, and (4) no

single phenotype exhibits high fitness across all environments. We build on this previous work, but we shift our focus from the evolutionary causes of adaptive phenotypic plasticity to investigate its evolutionary consequences in a fluctuating environment.

Each of our experiments are divided into two phases: in phase one, we precondition sets of founder organisms with differing plastic or non-plastic adaptations; in phase two, we examine the subsequent evolution of populations founded with organisms from phase one under specific environmental conditions. First, we examine the evolutionary histories of phase two populations to test whether adaptive plasticity constrained subsequent genomic and phenotypic changes. Next, we evaluate how adaptive plasticity influences exploration and exploitation by identifying how well populations produced by each type of founder are able to evolve and retain novel adaptive traits. Finally, we examine lineages to determine whether adaptive plasticity facilitated the accumulation of cryptic genetic variation that would prove deleterious when the environment changed.

We found that the evolution of adaptive plasticity reduced subsequent rates of evolutionary change in a cyclic environment. The non-plastic populations underwent more frequent selective sweeps and accumulated many more genetic changes over time, as non-plastic populations relied on genetic variation from *de novo* mutations to continuously readapt to environmental changes. We found that the evolution of adaptive phenotypic plasticity buffers populations against environmental fluctuations, whereas repeated selective sweeps in non-plastic populations drive the accumulation of deleterious mutations and the loss of secondary beneficial traits via deleterious hitchhiking. As such, adaptively plastic populations were better able to retain novel traits than their non-plastic counterparts.

3.2 Materials and Methods

3.2.1 The Avida Digital Evolution Platform

Avida is a study system wherein self-replicating computer programs (digital organisms) compete for space on a finite toroidal grid (Ofria et al., 2009). Each digital organism is

defined by a linear sequence of program instructions (its genome) and a set of virtual hardware components used to interpret and express those instructions. Genomes are expressed sequentially except when the execution of one instruction deterministically changes which instruction should be executed next (*e.g.*, a "jump" instruction). Genomes are built using an instruction set that is both robust (*i.e.*, any ordering of instructions is syntactically valid, though not necessarily meaningful) and Turing Complete (*i.e.*, able to represent any computable function, though not necessarily in an efficient manner). The instruction set includes operations for basic computations, flow control (*e.g.*, conditional logic and looping), input, output, and self-replication.

Organisms in Avida reproduce asexually by copying their genome instruction-by-instruction and then dividing. However, copy operations are imperfect and can result in single-instruction substitution mutations in an offspring's genome. For this work, we configured copy operations to err at a rate of one expected mutation for every 400 instructions copied (*i.e.*, a per-instruction error rate of 0.0025). We held individual genomes at a fixed length of 100 instructions; that is, we did not include insertion and deletion mutations. We used fixed-length genomes to control for treatment-specific conditions resulting in the evolution of substantially different genome sizes (Lalejini and Ferguson, 2021a)[1], which could, on its own, drive differences in evolutionary outcomes among experimental treatments. When an organism divides in Avida, its offspring is placed in a random location on the toroidal grid, replacing any previous occupant. For this work, we used the default 60 by 60 grid size, which limits the maximum population size to 3600 organisms. As such, improvements to the speed of self-replication are advantageous in the competition for space.

During evolution, organism replication rates improve in two ways: by improving genome efficiency (*e.g.*, using a more compact encoding) or by accelerating the rate at which the genome is expressed (their "metabolic rate"). An organism's metabolic rate determines the speed at which it executes instructions in its genome. Initially, an organism's metabolic

[1]We repeated our experiments without genome size restrictions and observed qualitatively similar results (see supplemental material, Lalejini and Ferguson 2021a).

rate is proportional to the length of its genome, but that rate is adjusted as it completes designated tasks, such as performing Boolean logic computations (Ofria et al., 2009). In this way, we can reward or punish particular phenotypic traits.

Phenotypic plasticity in Avida

In this work, we measure a digital organism's phenotype as the set of Boolean logic functions that it performs in a given environment. Sensory instructions in the Avida instruction set allow organisms to detect how performing a particular logic function would affect their metabolic rate (see supplemental material for more details, Lalejini and Ferguson 2021a). We define a phenotypically plastic organism as one that uses sensory information to alter which logic functions it performs based on the environment.

Phenotypic plasticity in Avida can be adaptive or non-adaptive for a given set of environments. Adaptive plasticity shifts net task expression closer to the optimum for the given environments. Non-adaptive plasticity changes task expression in either a neutral or deleterious way. Optimal plasticity toggles tasks to always perfectly match the set of rewarded tasks for the given set of environments.

3.2.2 Experimental design

We conducted three independent experiments using Avida to investigate how the evolution of adaptive plasticity influences evolutionary outcomes in fluctuating environments. For each experiment, we compared the evolutionary outcomes of populations evolved under three treatments (Figure 3.2): (1) a **PLASTIC** treatment where the environment fluctuates, and digital organisms can use sensory instructions to differentiate between environmental states; (2) a **NON-PLASTIC** treatment with identical environment fluctuations, but where sensory instructions are disabled; and (3) a **STATIC** control where organisms evolve in a constant environment.

Each experiment was divided into two phases that each lasted for 200,000 updates[2] of

[2]One update in Avida is the amount of time required for the average organism to execute 30 instructions.

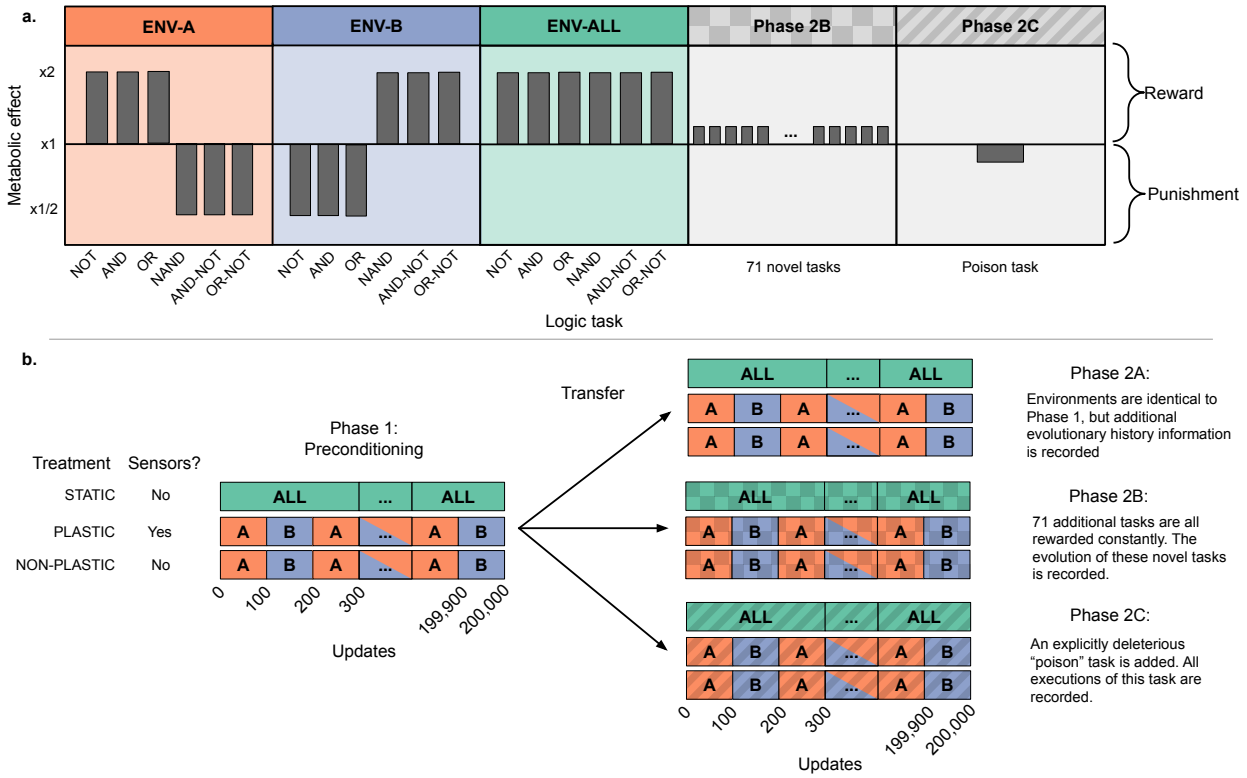


Figure 3.2: **Overview of experimental design.** The first three plots in panel (a) show the environments used in every experiment and whether they reward or punish each base task. Additionally, the last two subplots in (a) show the additional tasks added in phases 2B and 2C. All novel tasks confer a 10% metabolic reward, while executing the poisonous task causes a 10% metabolic punishment (bars not drawn to size). Panel (b) shows treatment differences and experiment phases. Treatments are listed on the left, with each treatment consisting of an environment timeline and whether sensors are functional. We conducted three independent two-phase experiments, each described on the right. Phases 2B and 2C are textured to match their task definitions in panel (a). Phase one is repeated for *each* experiment with 100 replicate populations per treatment per experiment. For each replicate at the end of phase one, we used an organism of the abundant genotype to found the second phase population. All STATIC and NON-PLASTIC populations move on to phase two, but PLASTIC populations only continue to the second phase if their most abundant genotype exhibits optimal plasticity. Metrics are recorded only in phase two.

evolution (Figure 3.2), which is approximately 30,000 to 40,000 generations. In phase one of each experiment, we preconditioned populations to their treatment-specific conditions. In phase two, we founded new populations with the evolved organisms from phase one and examined their subsequent evolution under given combinations of treatment and experimental conditions. During phase two, we tracked each population's evolutionary history as well as

See (Ofria et al., 2009) for more details.

saving the full final population. Phase one was for pre-conditioning only; all comparisons between treatments were performed on phase two data.

Environments

We constructed three experimental environments, abbreviated hereafter as "ENV-A", "ENV-B", and "ENV-ALL". Figure 3.2 describes these environments based on whether each of six Boolean logic tasks (NOT, NAND, AND, OR-NOT, OR, and AND-NOT) is rewarded or punished. A rewarded task performed by an organism doubles their metabolic rate, allowing them to execute twice as many instructions in the same amount of time. A punished task halves an organism's metabolic rate.

In both the PLASTIC and NON-PLASTIC conditions, the environment cycles between equal-length periods of ENV-A and ENV-B. Each of these periods persist for 100 updates (approximately 15 to 20 generations). Thus, populations experience a total of 1,000 full periods of ENV-A interlaced with 1,000 full periods of ENV-B during each experimental phase.

Organisms in the PLASTIC treatments differentiate between ENV-A and ENV-B by executing one of six sensory instructions, each associated with a particular logical task; these sensory instructions detect whether their associated task is currently rewarded or punished. By using sensory information in combination with execution flow-control instructions, organisms can conditionally perform different logic tasks depending on the current environmental conditions.

Experiment Phase 1 – Environment preconditioning

For each treatment, we founded 100 independent populations from a common ancestral strain capable only of self-replication. At the end of phase one, we identified the most abundant (*i.e.*, dominant) genotype and extracted an organism with that genotype from each replicate population to found a new population for phase two.

For the PLASTIC treatment, we measure plasticity by independently testing a given

genotype in each of ENV-A and ENV-B. We discard phase one populations if the dominant genotype does not exhibit optimal plasticity. This approach ensures that measurements taken on PLASTIC-treatment populations during the second phase of each experiment reflect the evolutionary consequences of adaptive plasticity.

Experiment Phase 2A – Evolutionary change rate

Phase 2A continued exactly as phase one, except we tracked the rates of evolutionary change in each of the PLASTIC-, NON-PLASTIC-, and STATIC-treatment populations. Specifically, we quantified evolutionary change rates using four metrics (each described in Table 3.1): (1) coalescence event count, (2) mutation count, (3) phenotypic volatility, and (4) mutational stability. We additionally used knockout experiments to examine how the genetic architectures of organisms and their ancestors changed over time, measuring the architectural volatility (Table 3.1) of evolved lineages.

Experiment Phase 2B – Novel task evolution

Phase 2B extended the conditions of phase one by adding 71 novel Boolean logic tasks, which were always rewarded in all treatments (Ofria et al., 2009). The original six phase one tasks (NOT, NAND, AND, OR-NOT, OR, and AND-NOT; hereafter called "base" tasks) continued to be rewarded or punished according to the particular treatment conditions. An organism's metabolic rate was increased by 10% for each novel task that it performed (limited to one reward per task). This reward provided a selective pressure to evolve these tasks, but their benefits did not overwhelm the 100% metabolic rate increase conferred by rewarded base tasks. As such, populations in the PLASTIC and NON-PLASTIC treatments could not easily escape environmental fluctuations by abandoning the fluctuating base tasks.

During this experiment, we tracked the extent to which populations evolving under each treatment were capable of acquiring and retaining novel tasks. Specifically, we used three metrics (each described in Table 3.1): (1) final novel task count, (2) novel task discovery, and (3) novel task loss.

Experiment Phase 2C – Deleterious instruction accumulation

Phase 2C extended the instruction set of phase one with a `poison` instruction. When an organism executes a `poison` instruction, it performs a "poisonous" task, which reduces the organism's metabolic rate (and thus reproductive success) but does not otherwise alter the organism's function. We imposed a 10% penalty each time an organism performed the poisonous task, making the `poison` instruction explicitly deleterious to execute. We did not limit the number of times that an organism could perform the poisonous task, and as such, organisms could perform the poisonous task as many times as they executed the `poison` instruction.

We tracked the number of times each organism along the dominant lineage performed the poisonous task. Specifically, we used two metrics (each described in Table 3.1): (1) final poisonous task count, and (2) poisonous task acquisition.

3.2.3 Experimental analyses

For each of our experiments, we tracked and analyzed the phylogenetic histories of evolving populations during phase two. For each replicate, we identified an organism with the most abundant genotype in the final evolved population, and we used it as a representative organism for further analysis. We then isolated the lineage from the founding organism to the representative organism, which we used as the representative lineage for further analysis. We manually inspected evolved phylogenies and found no evidence that any of our experimental treatments supported long-term coexistence. As such, each of the representative lineages reflect the majority of evolutionary history from a given population at the end of our experiment.

Some of our metrics required us to measure genotype-by-environment interactions. Importantly, in the fluctuating environments, we needed to differentiate phenotypic changes that were caused by mutations from those that were caused by environmental changes. To accomplish this, we produced organisms with the focal genotype, measured their phenotype

Metric	Description
Coalescence event count	Number of coalescence events that have occurred, which indicates the frequency of selective sweeps in the population.
Mutation count	Sum of all mutations that have occurred along a lineage.
Phenotypic volatility	Number of instances where parent and offspring phenotypic profiles do not match along a lineage. Phenotypic volatility as defined here indicates the rate at which accumulated genetic changes actually change the phenotype along a lineage.
Mutational stability	Proportion of mutated offspring along a lineage whose phenotypic profile matches that of their parent.
Architectural volatility	The average number of loci in the genome that change function per mutation along a lineage.
Final novel task count	Count of unique novel tasks performed by the representative organism in a final population from experiment phase 2B. This metric can range from 0 to 71 and measures the level of exploitation of the fitness landscape (*i.e.*, the mapping between genetic space and phenotype space) at a given point in time.
Novel task discovery	Number of unique novel tasks ever performed along a given lineage in experimental phase 2B, even if a task is later lost. This metric can range from 0 to 71 and measures a given lineage's level of exploration of the fitness landscape.
Novel task loss	Number of instances along a given lineage from experimental phase 2B where a novel task is performed by a parent but not its offspring. This metric measures how often a given lineage fails to retain evolved traits over time.
Final poisonous task count	Number of times the poisonous task is performed by the representative organism from a final population from experiment phase 2C.
Poisonous task acquisition count	Number of instances along a given lineage where a mutation causes an offspring perform the poisonous task more times than its parent.

Table 3.1: **Metric descriptions.**

in each environment, and aggregated the resulting phenotypes to create a *phenotypic profile*. Although organisms with different genotypes may express the same set of tasks across environments, their phenotypic profiles may not necessarily be the same. For example, an organism that expresses NOT in ENV-A and NAND in ENV-B has a distinct phenotypic profile from one that expresses NAND in ENV-A and NOT in ENV-B.

For an individual organism, we can perform knockout experiments to identify which instructions are responsible for producing a given phenotypic outcome. To perform a knockout, we duplicate the organism, replacing a single instruction with an inert "no-operation" instruction. We then identify any phenotypic changes by contrasting the execution results of the "knockout" organism and the original. Such changes provide evidence of the role that the original instruction must have played in the genome. For example, when an organism performs the NAND task but loses it when an instruction is knocked out, we categorize that instruction as part of the NAND task machinery. We use knockout experiments to characterize the role of each instruction in the genomes of every organism along all study lineages, revealing how genetic architectures change over time.

3.2.4 Statistical analyses

Across all of our experiments, we differentiated between sample distributions using non-parametric statistical tests. For each major analysis, we first performed a Kruskal-Wallis test (Kruskal and Wallis, 1952) to determine if there were significant differences in results from the PLASTIC, NON-PLASTIC, and STATIC treatments (significance level $\alpha = 0.05$). If so, we applied a Wilcoxon rank-sum test (Wilcoxon, 1992) to distinguish between pairs of treatments. We applied Bonferroni corrections for multiple comparisons (Rice, 1989) where appropriate.

3.2.5 Software availability

We conducted our experiments using a modified version of the Avida software, which is open source and freely available on GitHub (Lalejini and Ferguson, 2021a). We used Python for data processing, and we conducted all statistical analyses using R version 4 (R Core Team, 2021). We used the tidyverse collection of R packages (Wickham et al., 2019) to wrangle data, and we used the following R packages for analysis, graphing, and visualization: ggplot2 (Wickham et al., 2020), cowplot (Wilke, 2020), Color Brewer (Harrower and Brewer, 2003; Neuwirth, 2014), rstatix (Kassambara, 2021), ggsignif (Ahlmann-Eltze and Patil, 2021), scales (Wickham and Seidel, 2020), Hmisc (Harrell Jr et al., 2020), fmsb (Nakazawa, 2019), and boot (Canty and Ripley, 2019). We used R markdown (Allaire et al., 2020) and bookdown (Xie, 2020) to generate web-enabled supplemental material. All of the source code for our experiments and analyses, including guides for replication and configuration files, can be found in our supplemental material, which is hosted on GitHub (Lalejini and Ferguson, 2021a). Additionally, our experimental data is available on the Open Science Framework at `https://osf.io/sav2c/` (Lalejini and Ferguson, 2021b).

3.3 Results

3.3.1 The evolution of adaptive phenotypic plasticity slows evolutionary change in fluctuating environments

In experimental phase 2A, we tested whether the evolution of adaptive phenotypic plasticity constrained or promoted subsequent evolutionary change in a fluctuating environment. First, we compared the total amount of evolutionary change in populations evolved under the PLASTIC, NON-PLASTIC, and STATIC treatments as measured by coalescence event count, mutation count, and phenotypic volatility (Figure 3.3). According to each of these metrics, NON-PLASTIC populations experienced a larger magnitude of evolutionary change than either PLASTIC or STATIC populations. We observed significantly higher coalescence

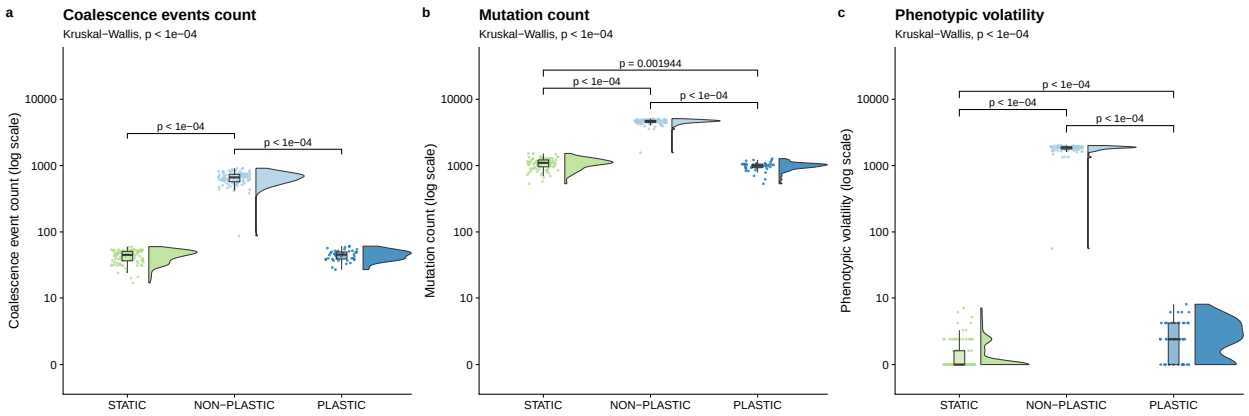


Figure 3.3: **Magnitude of evolutionary change.** Raincloud plots (Allen et al., 2019) of (a) coalescence event count, (b) mutation count, and (c) phenotypic volatility. See Table 3.1 for descriptions of each metric. Each plot is annotated with statistically significant comparisons (Bonferroni-corrected pairwise Wilcoxon rank-sum tests). Note that adaptive phenotypic plasticity evolved in 42 of 100 replicates from the PLASTIC treatment during phase one of this experiment; we used this more limited group to found 42 phase-two PLASTIC replicates from which we report these PLASTIC data.

event counts in NON-PLASTIC populations than in PLASTIC or STATIC populations (Figure 3.3a). NON-PLASTIC lineages had significantly higher mutation counts (Figure 3.3b) and phenotypic volatility than PLASTIC or STATIC lineages (Figure 3.3c).

Changing environments have been shown to increase generational turnover in Avida populations (Canino-Koning et al., 2016), which could explain why we observe a larger magnitude of evolutionary change at the end of 200,000 updates of evolution in NON-PLASTIC populations. Indeed, we found that significantly more generations of evolution elapsed in NON-PLASTIC populations (mean of 41090 ± 2702 std. dev.) than in PLASTIC (mean of 31016 ± 2615 std. dev.) or STATIC (mean of 30002 ± 3011 std. dev.) populations during phase 2A (corrected Wilcoxon rank-sum tests, $p < 10^{-4}$).

To evaluate whether increased generational turnover explains the greater magnitude of evolutionary change in NON-PLASTIC populations, we examined the average number of generations between coalescence events and the mutational stability of lineages (Table 3.1). A coalescence event indicates a selective sweep, which is a hallmark of adaptive evolutionary change. Mutational stability measures the frequency that mutations cause phenotypic

changes along a lineage (Table 3.1). We expect that static conditions should favor fit lineages with high mutational stability that no longer undergo rapid adaptive change and hence do not trigger frequent coalescence events. Under fluctuating conditions, however, lineages must be composed of plastic organisms if they are to maintain high fitness and mutational stability. Without plasticity, we expect these conditions to produce lineages with low stability and frequent coalescence events as populations must continually readapt.

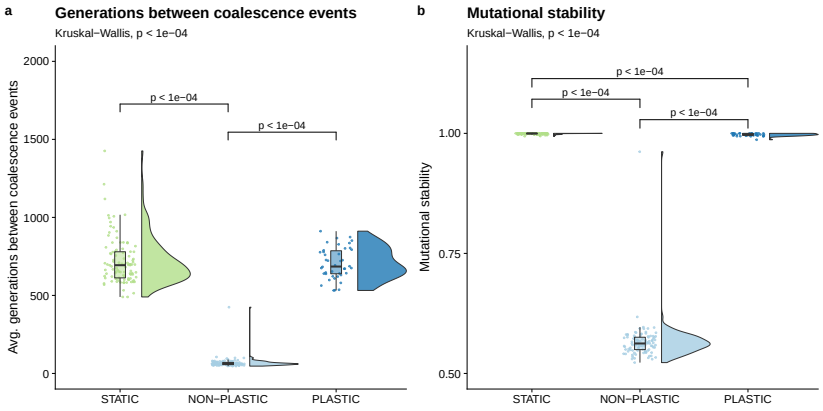


Figure 3.4: **Pace of evolutionary change.** Raincloud plots of (a) average number of generations between coalescence events, and (b) mutational stability (Table 3.1). Each plot is annotated with statistically significant comparisons (Bonferroni-corrected pairwise Wilcoxon rank-sum tests).

On average, significantly fewer generations elapsed between coalescence events in NON-PLASTIC populations than in either PLASTIC or STATIC populations (Figure 3.4a). We also found that both STATIC and PLASTIC lineages exhibited higher mutational stability relative to that of NON-PLASTIC lineages (Figure 3.4b); that is, mutations more often caused phenotypic changes along NON-PLASTIC lineages. Overall, our results indicate that NON-PLASTIC populations underwent more rapid (and thus a greater amount of) evolutionary change than either PLASTIC or STATIC populations.

While both STATIC and PLASTIC lineages exhibited high mutational stability, we found that STATIC lineages exhibited higher mutational stability than PLASTIC lineages (Figure 3.4b). Overall, there were rare instances of mutations that caused a change in phenotypic profile across all PLASTIC lineages. Of these mutations, we found that over 80% (83 out of 102) of changes to phenotypic profiles were cryptic. That is, the mutations

affected traits that would not have been expressed in the environment that the organism was born into, but would have been expressed had the environment changed.

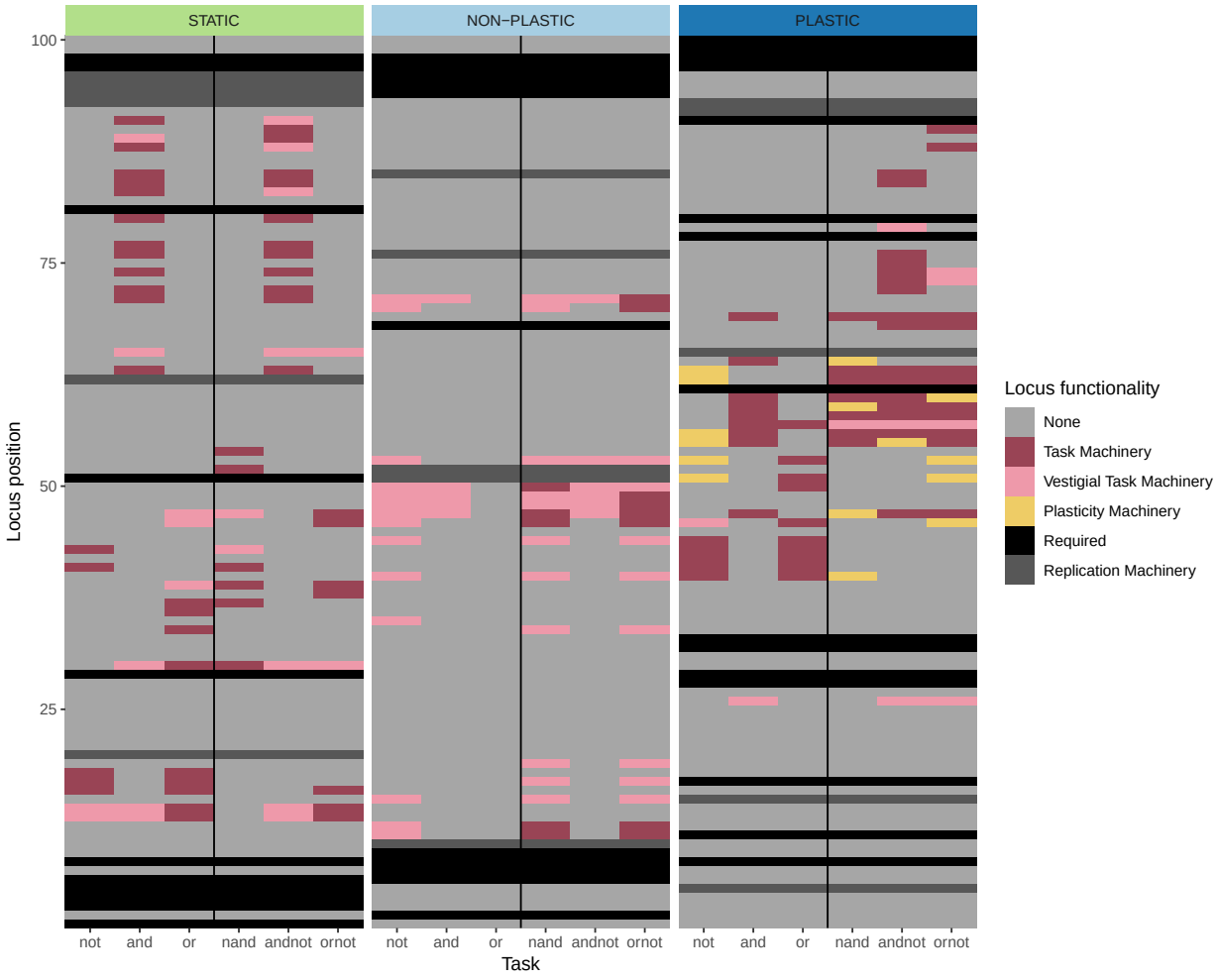


Figure 3.5: **Representative genetic architectures from each treatment.** Each box shows a representative genome from each condition at the end of Phase 2A. The y-axis indicates each site in each genome, and colors indicate the function of each locus with respect to a particular task (given by the x-axis). The vertical black line splits tasks rewarded in ENV-A (left of the line) from those rewarded in ENV-B. Loci colored as "Task Machinery" are actively involved in the performance of that task, while "Vestigial Task Machinery" represents loci that have not mutated, but no longer code for the task (*i.e.*, a change elsewhere in the genome has disabled or modified the task). "Plasticity Machinery" refers to loci that regulate the given task. Knocking out a "Replication Machinery" locus negatively affects replication time, while knocking out a "Required" locus results in a non-viable organism.

Next, we performed knockout experiments to investigate how genetic architectures evolved under the three treatment regimes. Thus far, we have shown that adaptive plasticity slows evolutionary change in fluctuating environments, but we have not determined if

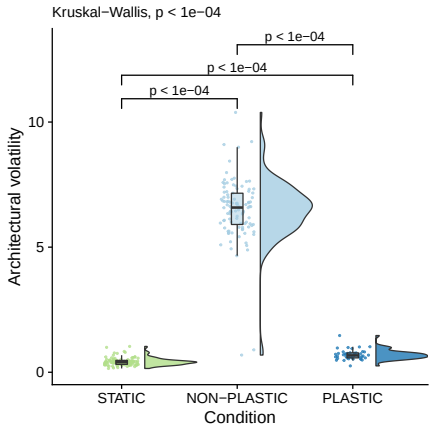


Figure 3.6: **Architectural volatility.** Raincloud plot of architecture stability (Table 3.1). The plot is annotated with statistically significant comparisons (Bonferroni-corrected pairwise Wilcoxon rank-sum tests).

adaptive plasticity also alters how genetic architectures (*i.e.*, how functions are arranged on genomes) change over time. Figure 3.5 shows the function of each locus in a representative genome from each treatment at the end of the experiment, which is also 100 updates after the environment changed from ENV-A to ENV-B in the PLASTIC and NON-PLASTIC treatments. The PLASTIC and STATIC genomes are capable of performing tasks from both environments, while the NON-PLASTIC genome only performs tasks rewarded in ENV-B. We expect instructions associated with ENV-A tasks to be lost in NON-PLASTIC populations during evolution in ENV-B. Indeed, we assign a metabolic cost for performing ENV-A tasks in ENV-B, resulting in negative selection on performing ENV-A tasks. Unused ENV-A tasks should also decay with evolution in ENV-B even without the metabolic cost we impose, as the associated instructions are more likely to accumulate mutations under relaxed selection (Lahti et al., 2009). Alternatively, previous work in Avida has shown that tasks punished in one environment can be maintained in the genome as vestigial loci (Canino-Koning et al., 2016, 2019) where their function is retained and co-opted for use in an alternate environment. Under this scenario we might expect the NON-PLASTIC genomes to retain the instructions needed for ENV-A tasks during their evolution in ENV-B. Consistent with Canino-Koning et al. (2016, 2019)'s work, the NON-PLASTIC genome that we analyzed from ENV-B (Figure

3.5) contains substantial vestigial loci for ENV-A's NOT task; we also found that these vestigial sites were exapted for ENV-B's NAND and OR-NOT tasks. Indeed, visual inspection of locus functions over entire lineages shows that NON-PLASTIC lineages often contain loci that are continuously cycling between coding for ENV-A tasks and ENV-B tasks. Further, NON-PLASTIC lineages exhibited significantly higher architectural volatility than STATIC or PLASTIC lineages (Figure 3.6).

In general, the evolution of adaptive plasticity stabilized PLASTIC treatment populations against environmental fluctuations, and their evolutionary dynamics more closely resembled those of populations evolving in a static environment. We observed no significant difference in the number and frequency of coalescence events in PLASTIC and STATIC populations. We did, however, observe small, but statistically significant, differences in each of the following metrics: elapsed generations, mutation counts, mutational stability, and architectural volatility between PLASTIC and STATIC populations (see supplemental material Lalejini and Ferguson 2021a).

3.3.2 Adaptively plastic populations retain more novel tasks than non-plastic populations in fluctuating environments

We have so far shown that adaptive plasticity constrains the rate of evolutionary change in fluctuating environments. However, it is unclear how this dynamic influences the evolution of novel tasks. Based on their relative rates of evolutionary change, we might expect NON-PLASTIC-treatment populations to evolve more novel tasks than PLASTIC-treatment populations. But, how much of the evolutionary change in NON-PLASTIC populations is useful for exploring novel regions of the fitness landscape versus continually rediscovering the same regions?

To answer this question, we quantified the number of novel tasks performed by a representative organism in the final population of each replicate. We found that both PLASTIC and STATIC populations had significantly higher final task counts than NON-PLASTIC pop-

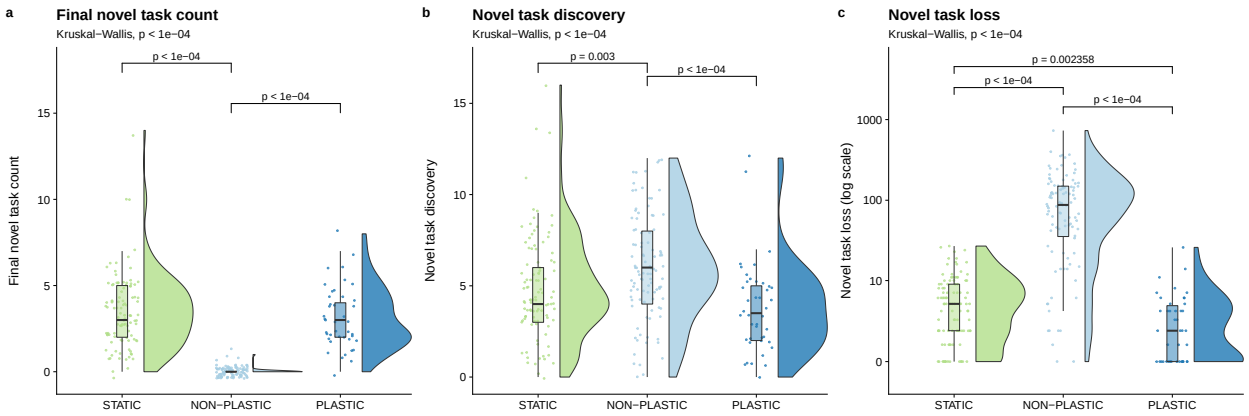


Figure 3.7: **Novel task evolution.** Raincloud plots of (a) final novel task count, (b) novel task discovery, and (c) novel task loss. See Table 3.1 for descriptions of each metric. Each plot is annotated with statistically significant comparisons (Bonferroni-corrected pairwise Wilcoxon rank-sum tests). Note that adaptive phenotypic plasticity evolved in 42 of 100 replicates from the PLASTIC treatment during phase one of this experiment; we used this more limited group to found 42 phase-two PLASTIC replicates from which we report these PLASTIC data.

ulations at the end of the experiment (Figure 3.7a). The final novel task count in PLASTIC and STATIC lineages could be higher than that of the NON-PLASTIC lineages for several non-mutually exclusive reasons. One possibility is that PLASTIC and STATIC lineages could be exploring a larger area of the fitness landscape when compared to NON-PLASTIC lineages. Another possibility is that the propensity of the NON-PLASTIC lineages to maintain novel traits could be significantly lower than PLASTIC and STATIC lineages. When we looked at the total sum of novel tasks discovered by each of the PLASTIC, STATIC, and NON-PLASTIC lineages, we found that the NON-PLASTIC lineages explored a significantly larger area of the fitness landscape (Figure 3.7b). Although the NON-PLASTIC lineages discovered more novel tasks, those lineages also exhibited significantly higher novel task loss when compared to PLASTIC and STATIC lineages (Figure 3.7c).

A larger number of generations elapsed in NON-PLASTIC populations than in PLASTIC or STATIC populations during our experiment (Lalejini and Ferguson, 2021a). Are NON-PLASTIC lineages discovering and losing novel tasks more frequently than PLASTIC or STATIC lineages, or are our observations a result of differences in generational turnover? To answer this question, we converted the metrics of novel task discovery and novel task loss

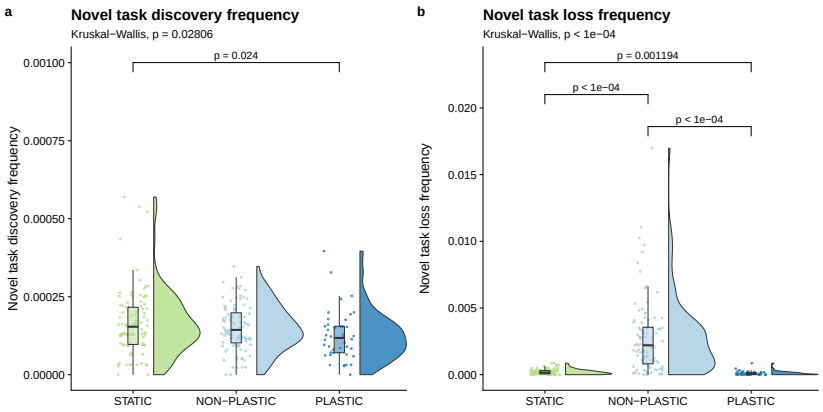


Figure 3.8: **Rates of novel task evolution.** Raincloud plots of (a) novel task discovery frequency and (b) novel task loss frequency. Each plot is annotated with statistically significant comparisons (Bonferroni-corrected pairwise Wilcoxon rank-sum tests).

to rates by dividing each metric by the number of elapsed generations along the associated representative lineages. We found no significant difference in the frequency of novel task discovery between NON-PLASTIC and STATIC lineages, and we found that PLASTIC lineages had a lower frequency of novel task discovery than STATIC lineages (Figure 3.8a). Therefore, we cannot reject the possibility that the larger magnitude of task discovery in NON-PLASTIC lineages was driven by a larger number of elapsed generations. NON-PLASTIC lineages had a higher frequency of task loss than either PLASTIC or STATIC lineages, and PLASTIC lineages tended to have a lower frequency of novel task loss than STATIC lineages (Figure 3.8b).

Next, we examined the frequency at which novel task loss along lineages co-occurred with the loss or gain of any of the six base tasks. Across all NON-PLASTIC representative lineages, over 97% (10998 out of 11229) of instances of novel task loss co-occurred with a simultaneous change in base task profile. In contrast, across all PLASTIC and STATIC dominant lineages, we observed that approximately 20% (29 out of 142) and 2% (13 out of 631), respectively, of instances of novel task loss co-occurred with a simultaneous change in base task profile. As such, the losses of novel tasks in NON-PLASTIC lineages appear to be primarily due to hitchhiking.

3.3.3 Lineages without plasticity that evolve in fluctuating environments express more deleterious tasks

Phenotypic plasticity allows for genetic variation to accumulate in genomic regions that are unexpressed, which could lead to the fixation of deleterious instructions in PLASTIC populations. However, in NON-PLASTIC lineages we observe a higher rate of novel task loss, indicating that they may be more susceptible to deleterious mutations (Figure 3.8).

Therefore, in experiment phase 2C, we tested whether adaptive phenotypic plasticity can increase the incidence of deleterious task performance. Specifically, we added an instruction that triggered a "poisonous" task and measured the number of times it was executed. Each execution of the `poison` instruction reduces an organism's fitness by 10%. At the beginning of phase 2C, the `poison` instruction is not present in the population, as it was not part of the instruction set during phase one of evolution. Accordingly, if a `poison` instruction fixes in a population, it must be the result of evolutionary dynamics during phase 2C, including cryptic variation or hitchhiking.

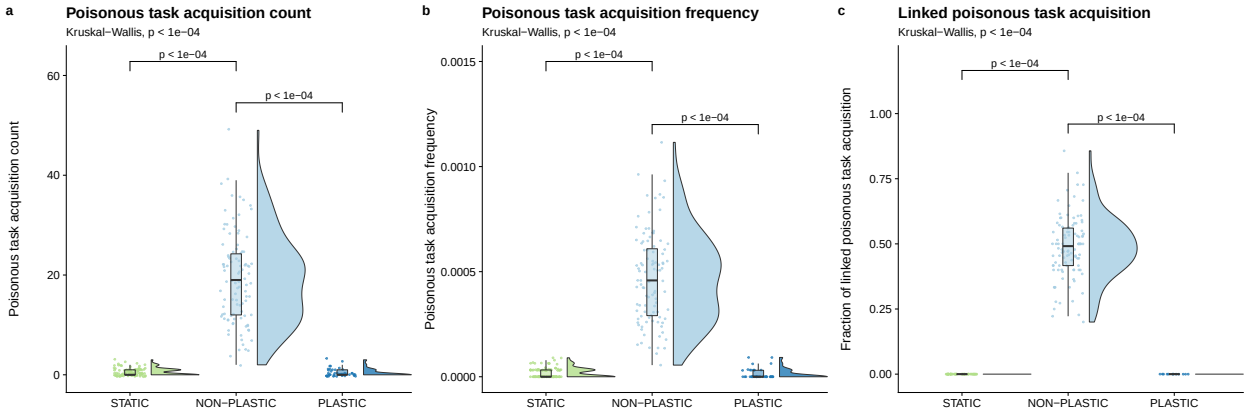


Figure 3.9: **Deleterious instruction accumulation.** Raincloud plots of (a) poisonous task acquisition, (b) poisonous task acquisition frequency, and (c) the proportion of mutations that increase poisonous task performance along a lineage that co-occur with a change in phenotypic profile. Each plot is annotated with statistically significant comparisons (Bonferroni-corrected pairwise Wilcoxon rank-sum tests). Note that adaptive phenotypic plasticity evolved in 43 of 100 replicates from the PLASTIC treatment during phase one of this experiment; we used this more limited group to found 43 phase-two PLASTIC replicates from which we report these PLASTIC data.

At the end of our experiment, no representative organisms from the PLASTIC or STATIC treatments performed the poisonous task under any environmental condition; however, representative organisms in 14% of replicates of the NON-PLASTIC treatment performed the poisonous task at least once. NON-PLASTIC lineages contained significantly more mutations that conferred the poisonous task as compared to PLASTIC or STATIC lineages (Figure 3.9a). This result does not change when we normalize by the number of generations represented in the given lineage (Figure 3.9b).

Next, we measured how often mutations that increased poisonous task performance co-occurred with changes to the base task profile within representative lineages. A poisonous instruction can fix in a lineage by having a beneficial effect that outweighs its inherent cost (*e.g.*, knocking out a punished task) or through linkage with a secondary beneficial mutation at another site within in the genome. Across all NON-PLASTIC representative lineages, we found that approximately 49% (956 out of 1916) of mutations that increased poisonous task expression co-occurred with a change in the base task profile (Figure 3.9c). In all representative lineages from the PLASTIC treatment, only 18 mutations increased poisonous task expression, and none co-occurred with a change in base task profile (Figure 3.9c). Likewise, only 58 mutations increased poisonous task performance in all representative lineages from the STATIC treatment, and none co-occurred with a change in base task profile (Figure 3.9c). We did not find compelling evidence that the few mutations that conferred poisonous task expression in PLASTIC lineages occurred as cryptic variation.

We repeated this experiment with 3% and 30% metabolic rate penalties associated with the poisonous task, which produced results that were consistent with those reported here (Lalejini and Ferguson, 2021a).

3.4 Discussion

In this work, we used evolving populations of digital organisms to determine how adaptive phenotypic plasticity alters subsequent evolutionary dynamics and influences evolution-

ary outcomes in fluctuating environments. First, we examined the evolutionary histories of plastic and non-plastic populations to test whether the evolution of adaptive plasticity promotes or constrains subsequent evolutionary change. Next, we evaluated how adaptive plasticity influences fitness landscape exploration and exploitation by testing whether plastic populations are better able to evolve and then maintain novel traits. Finally, we tested if the evolution of adaptive plasticity increases the potential for deleterious instructions to accumulate in evolving genomes.

Overall, our results indicate that adaptive plasticity can improve evolution's ability to maintain and refine novel traits, though with the tradeoff of reducing evolutionary exploration of the fitness landscape. Additionally, we found no evidence that adaptive plasticity increased the potential for deleterious instructions to accumulate in genomes. Instead, the genomes of non-plastic organisms that evolved in an identical fluctuating environment accumulated more deleterious instructions than that of adaptively plastic genomes. These dynamics appear to be driven by the stabilizing effect that adaptive plasticity had on population dynamics rather than plasticity's effect on genetic architecture or regulation.

3.4.1 The speed of evolutionary change

Adaptively plastic populations experienced fewer selective sweeps and fewer total genetic changes relative to non-plastic populations evolving under the same environmental conditions (Figure 3.3). Plastic populations adapted to the fluctuating environmental regime by evolving to sense environmental changes and regulate their metabolism (task performance) in response to such changes, which also stabilized these populations against fluctuations. Indeed, across all three of our experiments, the evolutionary dynamics of plastic populations were more similar to that of populations evolving in a static environment than to that of non-plastic populations evolving in an identical fluctuating environment.

Adaptive phenotypes in ENV-A were maladaptive in ENV-B and vice versa. As such, non-plastic generalists that performed all tasks or performed no tasks at all did not evolve

in any of the fluctuating environmental regimes. Selection against non-plastic generalists may be attributed to competition with phenotypic specialists that have a much larger fitness advantage for performing environment-specific tasks. In non-plastic populations where plasticity was disallowed, we hypothesize that strong selection on task specialization after each environmental change drove the repeated fixation of beneficial mutations (that alter an organism's phenotypic profile). This hypothesis is supported by the increased frequency of coalescence events in these populations (Figure 3.4) as well as increased rates of genetic and phenotypic changes observed along the lineages of non-plastic organisms.

Analysis of the evolved genetic architectures further supports our hypothesis that the non-plastic populations relied on mutations to continuously readapt to the fluctuating environment. This aligns with previous work, which has shown that, in the absence of plasticity, fluctuating environments steer populations toward genotypes that readily mutate to alternative phenotypes (Lalejini and Ofria, 2016; Canino-Koning et al., 2016). Indeed, (Canino-Koning et al., 2016) also observed that genomes evolved in cyclic environments often contained vestigial fragments of genetic material adapted to prior environments, which we see in the non-plastic populations.

This study is the first in-depth empirical investigation into how the *de novo* evolution of adaptive plasticity shifts the course of subsequent evolution in a cyclic environment. The evolutionary dynamics that we observed in non-plastic populations, however, are consistent with results from previous digital evolution studies. Consistent with our findings, Dolson et al. (2020) showed that non-plastic populations that were evolved in cyclically changing environments exhibited higher phenotypic volatility and accumulated more mutations than that of populations evolved in static conditions.

Our results are also consistent with conventional evolutionary theory. A trait's evolutionary response to selection depends on the strength of directional selection and on the amount of genetic variation for selection to act upon (Lande and Arnold, 1983; Zimmer and Emlen, 2013). In our experiments, non-plastic populations repeatedly experienced strong

directional selection to toggle which tasks were expressed after each environmental change. As such, retrospective analyses of successful lineages revealed rapid evolutionary responses (that is, high rates of genetic and phenotypic changes). Evolved adaptive plasticity shielded populations from strong directional selection when the environment changed by eliminating the need for a rapid evolutionary response to toggle task expression. Indeed, both theoretical and empirical studies have shown that adaptive plasticity can constrain evolutionary change by weakening directional selection on evolving populations (Price et al., 2003; Paenke et al., 2007; Ghalambor et al., 2015).

3.4.2 The evolution and maintenance of novel tasks

In fluctuating environments, non-plastic populations explored a larger area of the fitness landscape than adaptively plastic populations, as measured by novel task discovery (Figure 3.7b). Despite lower overall novel task discovery in adaptively plastic populations, they better exploited the fitness landscape, retaining a greater number of novel tasks than non-plastic populations evolving under identical environmental conditions (Figure 3.7a). Evolution in non-plastic populations was dominated by numerous bouts of strong directional selection driven by repeated environmental change. After each change, the performance of the six base tasks needed to be realigned to the environment. In our experiment, novel tasks were less important to survival than the fluctuating base tasks. In non-plastic populations, mutations that improve an offspring's fitness after an environmental change are extremely beneficial, and as such beneficial mutations fix, they can carry with them co-occurring deleterious mutations that knock out novel tasks. Indeed, we found that mutations associated with novel task loss along representative lineages from non-plastic populations co-occurred with mutations that helped offspring adapt to environmental changes 97% of the time.

Temporary environmental changes can improve fitness landscape exploration and exploitation in evolving populations of non-plastic digital organisms (Nahum et al., 2017). In our system, however, we found that *repeated* fluctuations reduced the ability of non-plastic

populations to maintain and exploit tasks; that said, we did find that repeated fluctuations may improve overall task discovery by increasing generational turnover. Consistent with our findings, Canino-Koning et al. (2019) found that non-plastic populations of digital organisms evolving in a harsh cyclic environment maintained fewer novel traits than populations evolving in static environments.

Our results suggest that adaptive phenotypic plasticity can improve the potential for populations to exploit novel resources by stabilizing them against stressful environmental changes. The stability that we observe may also lend some support to the hypothesis that phenotypic plasticity can rescue populations from extinction under changing environmental conditions (Chevin et al., 2010).

Our data do not necessarily provide evidence for or against the genes as followers hypothesis. The genes as followers hypothesis focuses on contexts where plastic populations experience novel or abnormally stressful environmental change. However, in our system, environmental changes were cyclic (not novel), and the magnitude of changes were consistent for the entirety of the experiment (so none were abnormally stressful). Further, the introduction of novel tasks during the second phase of the experiment merely added additional static opportunities for fitness improvement and did not change the meaning of existing environmental cues.

3.4.3 The accumulation of deleterious instructions

We found that non-plastic lineages that evolved in a fluctuating environment exhibited both larger totals and higher rates of deleterious instruction (`poison`) accumulation than that of adaptively plastic lineages (Figure 3.9). We did not find evidence of `poison` instructions accumulating as cryptic variation in adaptively plastic lineages. We hypothesize that deleterious genetic hitchhiking drove `poison` instruction accumulation along non-plastic lineages in changing environments. In asexual populations without horizontal gene transfer, all co-occurring mutations are linked. As such, deleterious mutations linked with a stronger

beneficial mutation (*i.e.*, a driver) can sometimes "hitchhike" to fixation (Smith and Haigh, 1974; Van den Bergh et al., 2018; Buskirk et al., 2017). Natural selection normally prevents deleterious mutations from reaching high frequencies, as such mutants would be outcompeted. However, when a beneficial mutation sweeps to fixation in a clonal population, it carries along any linked genetic material, including other beneficial, neutral, or deleterious mutations Barton (2000); Smith and Haigh (1974).

Across our experiments, the frequency of selective sweeps in non-plastic populations provided additional opportunities for genetic hitchhiking with each environmental change. Indeed, representative lineages from non-plastic populations in the cyclic environment exhibited higher mutation accumulation (Figure 3.3b), novel trait loss (Figure 3.7c), and deleterious instruction accumulation (Figure 3.9) than their plastic counterparts. In aggregate, we found that many (∼49%; 956 / 1916) mutations that increased `poison` instruction execution in offspring co-occurred with mutations that provided an even stronger benefit by adapting the offspring to an environmental change. This rate of co-occurrence is conservative because we did not analyze mutations that became linked in different generations.

We found that adaptive phenotypic plasticity reduced `poison` instruction accumulation by reducing the rate of evolutionary change, which in turn reduced opportunities for `poison` instructions to fix. We did not find compelling evidence of cryptic variation harboring `poison` instructions in adaptively plastic lineages. We have two hypotheses for why we did not observe the accumulation of `poison` instructions in unexpressed plastic responses. First, the period of time between environmental changes was too fast for variants carrying unexpressed `poison` instructions to reach high frequencies before the environment changed, after which such variants would have been outcompeted. Indeed, we tuned the frequency of environmental fluctuations so that genes that needed to function appropriately in the off environment were able to remain in the population despite relaxed selection. Second, the genetic mechanisms of plasticity that evolve in Avida are typically well-integrated and highly specific; that is, plastic genomes usually adjust their phenotypic expression by toggling

a minimal number of key instructions. As such, there is little genomic space for variation to accumulate in preexisting (but unexpressed) regulated regions.

3.4.4 Limitations and future directions

Our conclusions are limited to *adaptively* plastic populations. We did not explore the effects of non-adaptive plasticity where environmental changes induce phenotypes that are further away from the local optimum (*e.g.*, Leroi et al. 1994). Non-adaptive plasticity can increase a population's extinction risk, especially if the misaligned plastic response is strongly tied to survival or the population is not sufficiently large (Gomulkiewicz and Holt, 1995; Chevin et al., 2010). If the population persists, however, non-adaptive plasticity has been shown to be capable of accelerating evolutionary change by increasing the strength of directional selection. (Ghalambor et al., 2015).

Environmental cues in our experiments were reliable, and environmental changes were consistent over time. That is, sensory instructions perfectly differentiated between ENV-A and ENV-B, and environmental fluctuations never exposed populations to entirely new conditions. Both the reliability of cues and the timescales of environment switching are known to influence evolutionary outcomes (Li and Wilke, 2004; Boyer et al., 2021). For example, Boyer et al. (2021) evolved populations of *Saccharomyces cerevisiae* in an environment that fluctuated between two growth conditions, observing that both environmental predictability and switching rate influenced the rates of evolutionary responses as well as adaptive outcomes. In adaptively plastic populations, environmental switching rate can influence how plastic responses are maintained, including their genetic architecture as well as their likelihood of maintenance. Our work lays the groundwork for using digital evolution experiments to investigate the evolutionary consequences of phenotypic plasticity in a range of contexts, including different forms of plasticity (*e.g.*, adaptive versus non-adaptive), more complex environments with more than two possible states, stochastic environmental changes, and different environment switching rates.

We focused our analyses on the lineages of organisms with the most abundant genotype in the final population. These successful lineages represented the majority of the evolutionary histories of populations at the end of our experiment, as populations did not exhibit long-term coexistence of different clades. Our analyses, therefore, gave us an accurate picture of what fixed in the population. We did not, however, examine the lineages of extinct clades. Future work will extend our analyses to include extinct lineages, giving us a more complete view of evolutionary history, which may allow us to better distinguish adaptively plastic populations from populations evolving in a static environment.

As with any wet-lab experiment, our results are in the context of a particular model organism: "Avidian" self-replicating computer programs. Digital organisms in Avida regulate responses to environmental cues using a combination of sensory instructions and conditional logic instructions (`if` statements). The `if` instructions conditionally execute a single instruction depending on previous computations and the state of memory. As such, plastic genomes typically regulate a small number of key instructions that, when executed, change the expressed phenotype as opposed to large, sequences of co-regulated instructions (Lalejini and Ferguson, 2021a). This bias may limit the accumulation of hidden genetic variation in Avida genomes. However, as there are many model biological organisms, there are many model digital organisms that have different regulatory mechanisms that should be used to test the generality of our results.

Chapter 4

Evolving Event-driven Programs with SignalGP

Authors: Alexander Lalejini and Charles Ofria

This chapter is adapted from (Lalejini and Ofria, 2018), which underwent peer review and appeared in the proceedings of the 2018 Genetic and Evolutionary Computation Conference.

4.1 Introduction

Here, we introduce SignalGP, a new genetic programming (GP) technique designed to provide evolution direct access to the event-driven programming paradigm, allowing evolved programs to handle signals from the environment or from other agents in a more biologically inspired way than traditional GP approaches. In SignalGP, signals (*e.g.*, from the environment or from other agents) direct computation by triggering the execution of program modules (*i.e.*, functions). SignalGP augments the tag-based referencing techniques demonstrated by Spector *et al.* (Spector et al., 2011b,a, 2012) to specify which function is triggered by a signal, allowing the relationships between signals and functions to evolve over time. The SignalGP implementation presented here is demonstrated in the context of linear GP, wherein programs are represented as linear sequences of instructions; however, the ideas underpinning SignalGP are generalizable across a variety of genetic programming representations.

Linear genetic programs generally follow an imperative programming paradigm where computation is driven procedurally. Execution often starts at the top of a program and proceeds in sequence, instruction-by-instruction, jumping or branching as dictated by executed instructions (Brameier and Banzhaf, 2007; McDermott and O'Reilly, 2015). In contrast to the imperative programming paradigm, program execution in event-driven computing is directed primarily by signals (*i.e.*, events), easing the design and development of programs that, much like biological organisms, must react on-the-fly to signals in the environment or from other agents. Is it possible to provide similarly useful abstractions to evolution in genetic programming?

Different types of programs are more or less challenging to evolve depending on how they are represented and interpreted. By capturing the event-driven programming paradigm, SignalGP targets problem domains where agent-agent and agent-environment interactions are crucial, such as in robotics or distributed systems.

In the following sections, we provide a broad overview of the event-driven paradigm, discussing it in the context of an existing event-driven software framework, cell signal transduction, and an evolutionary computation system for evolving robot controllers. Next, we discuss our implementation of SignalGP in detail. Then, we use SignalGP to demonstrate the value of capturing event-driven programming in GP with two test problems: an environment coordination problem and a distributed leader election problem. Finally, we conclude with planned extensions, including how SignalGP can be generalized beyond our linear GP implementation to other forms of GP.

4.2 The event-driven paradigm

The event-driven programming paradigm is a software design philosophy where the central focus of development is the processing of events (Etzion and Niblett, 2010; Heemels et al., 2012; Cassandras, 2014). Events often represent messages from other agents or processes, sensor readings, or user actions in the context of interactive software applications. Events are

processed by callback functions (*i.e.*, event-handlers) where the appropriate event-handler is determined by an identifying characteristic associated with the event, often the event's name or type. In this way, events can act as remote function calls, allowing external signals to direct computation.

Software development environments that support the event-driven paradigm often abstract away the logistics of monitoring for events and triggering event-handlers, simplifying the code that must be designed and implemented by the programmer and easing the development of reactive programs. Thus, the event-driven paradigm is especially useful when developing software where computation is most appropriately directed by external stimuli, which is often the case in domains such as robotics, embedded systems, distributed systems, and web applications.

For any event-driven system, we can address the following three questions: What are events? How are event-handlers represented? And, how does the system determine the most appropriate event-handler to trigger in response to an event? Crosbie and Spafford (Crosbie and Spafford, 1996) have addressed why answering such questions can be challenging in genetic programming; thus, it is useful to look to how existing event-driven systems address them. While many systems that exhibit event-driven characteristics exist, we restrict our attention to three: the Robot Operating System (ROS) (Quigley et al., 2009), the biological process of signal transduction, and Byers *et al.*'s digital enzymes robot controller (Byers et al., 2011, 2012).

ROS is a popular robotics software development framework that provides standardized communication protocols to independently running programs, which are referred to as nodes. While the ROS framework provides a variety of tools and other conveniences to robotics software developers, we focus on ROS's publish-subscribe communication protocol, framing it under the event-driven paradigm. ROS nodes can communicate by passing strictly typed messages over named channels (topics). Nodes send messages by publishing them over topics, and nodes receive messages from a particular topic by subscribing to that topic. A node

subscribes to a topic by registering a callback function that takes the appropriate message type as an argument. Anytime a message is sent over a topic, all callback functions registered with the topic are triggered, allowing subscribed nodes to react to published messages. Topics can have any number of publishers and subscribers, all agnostic to one another (Quigley et al., 2009). In ROS's publish-subscribe system, events are represented as strictly typed messages, event-handlers are callback functions that take event information as input, and named channels (topics) determine which event-handlers an event triggers.

The behavior of many natural systems can be interpreted as using the event-driven paradigm. In cell biology, signal transduction is the process by which a cell transforms an extracellular signal into a response, often in the form of cascading biochemical reactions that alter the cell's behavior. Cells respond to signaling molecules via receptors, which bind specifically to nearby signaling molecules and initiate the cell's response (Alberts et al., 2002). The process of cell signal transduction can be viewed as a form of event-driven computation: signaling molecules are like events, receptors are event-handlers, and the chemical and physical properties of signaling molecules determine with which receptors they are able to bind.

Evolutionary computation researchers have also made use of the event-driven paradigm. For example, Byers *et al.* (Byers et al., 2011, 2012) demonstrated virtual robot controllers that operate using a digital model of signal transduction, and like biological signal transduction, these controllers follow an event-driven paradigm. Byers *et al.*'s virtual robot controllers have digital stimuli receptors, which bind to nearby "signaling molecules" represented as bit strings. Different bit strings represent different signals in the environment (*e.g.*, the presence of nearby obstacles). Once a signaling molecule binds to a digital receptor, a digital enzyme (program) processes the signaling molecule and influences the controller's behavior. In a single controller, there are many digital enzymes (not all of the same type) processing signaling molecules in parallel, all vying to influence the controller's actions; in this way, virtual robot behavior emerges. As in cell signal transduction, signaling molecules are events,

digital stimuli receptors and digital enzymes act as event-handlers, and events are paired with handlers based on signal type and signal location.

4.3 SignalGP

As with other tag-based systems, SignalGP agents (programs) are defined by a set of functions (modules) where each function is referred to using a tag and contains a linear sequence of instructions. To augment this framework, SignalGP also makes explicit the concept of *events* where event-specific data is associated with a tag that agents can use to specify how that event should be handled. In this work, we arbitrarily chose to represent tags as fixed-length bit strings. Agents may both generate internal events and be subjected to events generated by the environment or by other agents. Events trigger functions based on the similarity of their tags. When an event triggers a function, the function is run with the event's associated data as input. SignalGP agents handle many events simultaneously by processing them in parallel. Figure 4.1 shows a high-level overview of SignalGP.

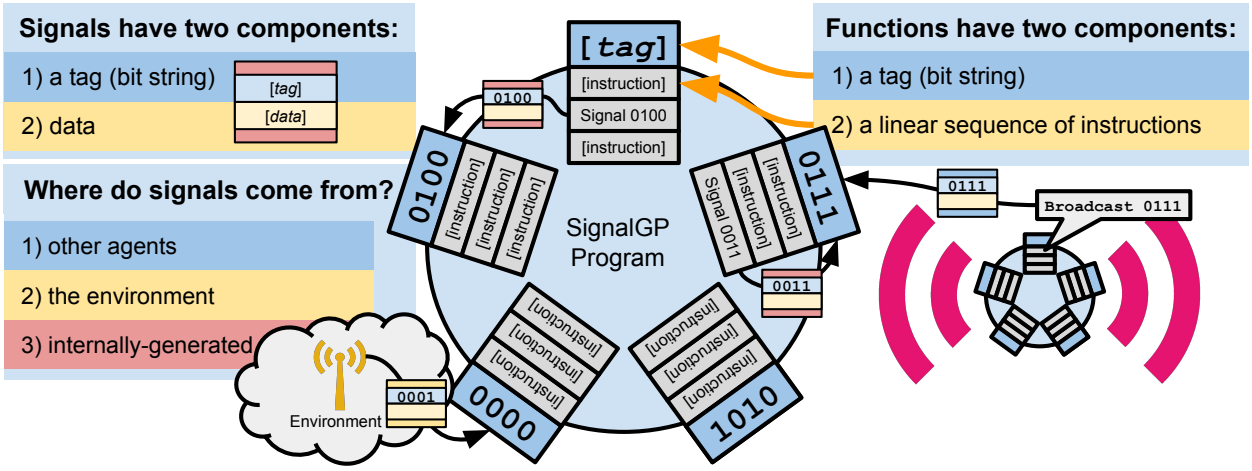


Figure 4.1: **A high-level overview of SignalGP.** SignalGP programs are defined by a set of functions. Events trigger functions with the *closest matching* tag, allowing SignalGP agents to respond to signals. SignalGP agents handle many events simultaneously by processing them in parallel.

4.3.1 Tag-based Referencing

Incorporating modules (*e.g.*, functions, subroutines, macros, *etc.*) into genetic programming has been extensively explored, and the benefits of modules in GP have been well documented (*e.g.*, Koza (1992, 1994); Angeline and Pollack (1992); Keijzer et al. (2005); Walker and Miller (2008); Roberts et al. (2001); Spector (1996)). The main purpose of SignalGP functions are to act as event-handlers—computations triggered in response to signals. However, they have the additional benefit of providing explicit architectural support for program modularity, bestowing the boon of reusable code. As with any reusable code block in GP, the question remains: how should the code be referenced? The answer to this question can be reused to answer the following question: how should we determine which event-handlers are triggered by events?

Inspired by John Holland's concept of a "tag" (Holland, 1993, 1987, 1990, 2006) as a mechanism for matching, binding, and aggregation, Spector *et al.* introduced and demonstrated the value of tag-based referencing in the context of GP (Spector et al., 2011b,a, 2012). In this context, a tag-based reference always links to a tagged entity with its closest match. These tagged entities include instructions and sequences of code (*i.e.*, modules), providing an evolvable mechanism for code referencing.

SignalGP shifts these ideas into a more fully event-driven context. In SignalGP, sets of instructions are modularized into functions that are labeled with tags. Events are made explicit and trigger those functions with whose tags have the closest match. The underlying instruction set is crafted to easily trigger internal events, broadcast external events, and to otherwise work in a tag-based context. Finally, SignalGP can be configured to only match tags that are relatively close (within a threshold) allowing agents to ignore events entirely by avoiding the use of similar tags.

4.3.2 Virtual Hardware

As in many GP representations, linear GP programs are often interpreted in the context of virtual hardware, which typically comprises memory—usually in the form of registers or stacks—and other problem-specific virtual hardware elements, allowing programs to achieve complex functionality (McDermott and O'Reilly, 2015; Poli et al., 2008; Ofria et al., 2009). SignalGP programs are interpreted by virtual hardware consisting of the following four major components: program memory, an event queue, a set of execution threads, and shared memory.

Program memory stores the SignalGP program currently executing on the virtual hardware.

The **event queue** manages recently received events waiting to be dispatched and processed by functions. The event queue dispatches events in the order they are received.

The SignalGP virtual hardware supports an arbitrary number of **execution threads** that run concurrently. Each thread processes a single instruction every time step. Just as Byers *et al.*'s parallel-executing digital enzymes allow robot controllers to process many external stimuli simultaneously (Byers et al., 2011), parallel execution allows SignalGP agents to handle many events at once.

Each thread maintains a call stack that stores state information about the thread's active function calls. The current state for any given thread resides at the top of the thread's call stack. Call states maintain local state information for the function call they represent: a function pointer, an instruction pointer, input memory, working memory, and output memory. The function pointer indicates the current function being run. The instruction pointer indicates the current instruction within that function. Input, working, and output memory serve as local memory.

Working memory is used for performing local operations (*e.g.*, addition, subtraction, multiplication, *etc.*). Input memory is analogous to function arguments (*i.e.*, function input), and output memory is analogous to function return memory (*i.e.*, what is returned when

a function call concludes). By convention, instructions can both read from and write to working memory, input memory is read-only, and output memory is write-only. To use an analogy, working memory, input memory, and output memory are to SignalGP functions as hidden nodes, input nodes, and output nodes are to conventional artificial neural networks. **Shared memory** serves as global memory. Shared memory is accessible (*i.e.*, readable and writable) by all threads, allowing them to store and share information.

4.3.3 Program Evaluation

SignalGP programs are sets of functions where each function associates an evolvable tag with a linear sequence of instructions. In our implementation of SignalGP, instructions are argument-based, and in addition to evolvable arguments, each instruction has an evolvable tag. Arguments modify the effect of the instruction, often specifying memory locations or fixed values. Instruction tags may also modify the effect of an instruction. For example, instructions that refer to functions do so using tag-based referencing. Further, instructions use their tag when generating events, either to be broadcast to other SignalGP agents or to be handled internally for their own use.

Program evaluation can be initialized either actively or passively. During active initialization, the program will begin evaluation by automatically calling a designated main function on a new thread. In passive initialization, computation takes place only in response to external events. In the work presented here, we use active initialization and automatically reset the main thread if it would have otherwise terminated.

While executing, the SignalGP virtual hardware advances on each time step in three phases: (1) All events in the event queue are dispatched, with each triggering a function *via* tag-based referencing. (2) Each thread processes a single instruction. (3) Any threads done processing are removed. Phases occur serially and in order.

Executed instructions may call functions, manipulate local and shared memory, generate events, perform basic computations, control execution flow, *et cetera* (see supplementary

material (Lalejini, 2018) for details on all instructions used in this work). Instructions in SignalGP are guaranteed to always be syntactically valid, but may be functionally useless. Every instruction has three associated arguments and an associated tag. Not all instructions make use of their three arguments or their tag; unused arguments and tags are not under direct selection and may drift until a mutation to the operator reveals them.

Instruction-triggered Function Calls

Functions in SignalGP may be triggered by either instruction calls or events. When a `Call` is executed, the function in program memory with the most similar tag to the `Call` instruction's tag (above a similarity threshold) is triggered; in this work, ties are broken by a random draw (though any tie-breaking procedure could be used). Tag similarity is calculated as the proportion of matching bits between two bit strings (simple matching coefficient).

When a function is triggered by a `Call` instruction, a new call state is created and pushed onto that thread's call stack. The working memory of the caller state is copied as the input memory of the new call state (*i.e.*, the arguments to the called function are the full contents of the previous working memory). The working memory and the output memory of the new call state are initially empty. To prevent unbounded recursion, we place limits on call stack depth; if a function call would cause the call stack to exceed its depth limit, the call instead behaves like a no-operation.

Instruction-triggered functions may return by either executing a `Return` instruction or by reaching the end of the function's instruction sequence. When an instruction-triggered function returns, its call state is popped from its call stack, and anything stored in the output memory of the returning call state is copied to the working memory of the caller state (otherwise leaving the caller state's working memory unchanged). In this way, instruction-triggered function calls can be thought of as operations over the caller's working memory.

Event-triggered Function Calls

Events in SignalGP are analogous to external function calls. When an event is dispatched from the event queue, the virtual hardware chooses the function with the highest tag similarity score (above a similarity threshold) to handle the event. In this work, ties are broken by a random draw (though any tie-breaking procedure could be used).

Once a function is selected to handle an event, it is called on a newly-created execution thread, initializing the thread's call stack with a new call state. The input memory of the new call state is populated with the event's data. In this way, events can pass information to the function that handles them. When the function has been processed (*i.e.*, all of the active calls on the thread's call stack have returned), the thread is removed. To prevent unbounded parallelism, we place a limit on the allowed number of concurrently executing threads; if the creation of a new thread would cause the number of threads to exceed this limit, thread creation is prevented.

4.3.4 Evolution

Evolution in SignalGP proceeds similarly to that of typical linear GP systems. Because function referencing is done *via* tags, changes can be made to program architecture (*e.g.*, inserting new or removing existing functions) while still guaranteeing syntactic correctness. Thus, modular program architectures can evolve dynamically through whole-function duplication and deletion operators or through function-level crossover techniques.

In the studies presented in this paper, we evolve SignalGP programs directly (as opposed to using indirect program encodings), which requires SignalGP-aware mutation operators. We propagated SignalGP programs asexually and applied mutations to offspring. We used whole-function duplication and deletion operators (applied at a per-function rate of 0.05) to allow evolution to tune the number or functions in programs. We mutated tags for instructions and functions at a per-bit mutation rate (0.05). We applied instruction and argument substitutions at a per-instruction rate (0.005). Instruction sequences could be

inserted or deleted via slip-mutation operators (Lalejini et al., 2017), which facilitate the duplication or deletion of sequences of instructions; we applied slip-mutations at a per-function rate (0.05).

4.4 Test Problems

We demonstrate the value of incorporating the event-driven programming paradigm in GP using two distinct test problems: a changing environment problem and a distributed leader-election problem. For both problems, we compared SignalGP performance to variants that are otherwise identical, except for how they handle sensor information. For example, our primary variant GP must actively monitor sensors to process external signals (using the imperative paradigm). For both test problems, a program's capacity to react efficiently to external events is crucial; thus, we hypothesized that SignalGP should perform better than our imperative alternatives.

4.4.1 Changing Environment Problem

This first problem requires agents to coordinate their behavior with a randomly changing environment. The environment can be in one of K possible states; to maximize fitness, agents must match their internal state to the current state of their environment. The environment is initialized to a random state and has a 12.5% chance of changing to a random state at every subsequent time step. Successful agents must adjust their internal state whenever an environmental change occurs.

We evolved agents to solve this problem at K equal to 2, 4, 8, and 16 environmental states. The problem scales in difficulty as the number of possible states that must be monitored increases. Agents adjust their internal state by executing one of K state-altering instructions. For each possible environmental state, there is an associated `SetState` instruction (*i.e.*, for $K = 4$, there are four instructions: `SetState0`, `SetState1`, `SetState2`, and `SetState3`). Being required to execute a distinct instruction for each environment represents

performing a behavior unique to that environment.

We compared the performance of programs with three different mechanisms to sense the environment: (1) an event-driven treatment where environmental changes produce signals that have environment-specific tags and can trigger functions; (2) an imperative control treatment where programs needed to actively poll the environment to determine its current state; and (3) a combined treatment where agents are capable of using either option. Note that in the imperative and combined treatments we added new instructions to test each environmental state (*i.e.*, for $K = 4$, there are four instructions: `SenseEnvState0`, `SenseEnvState1`, `SenseEnvState2`, and `SenseEnvState3`). In preliminary experiments we had provided agents with a single instruction that returned the current environmental state, but this mechanism proved more challenging for them to use effectively when there were too many states (the environment ID returned by the single instruction needed to be thresholded into a true/false value, whereas the individual environment state tests directly returned a true/false value).

Across all treatments, we added a `Fork` instruction to the available instruction set. The `Fork` instruction generates an internally-handled signal when executed, which provides an independent mechanism to spawn parallel-executing threads. The `Fork` instruction ensures that programs in all treatments had trivial access to parallelism. Because the `SenseEnvState` instructions in both the imperative and combined treatments bloated the instruction set relative to the event-driven treatment, we also added an equivalent number of no-operation instructions in the event-driven treatment.

Hypotheses

For low values of K, we expected evolved programs from all treatments to perform similarly. However, as continuously polling the environment is cumbersome at higher values of K, we expected fully event-driven SignalGP programs to drastically outperform programs evolved in the imperative treatment; further, we expected successful programs in the com-

bined treatment to favor the event-driven strategy.

Experimental Parameters

We ran 100 replicates of each condition at $K = 2$, 4, 8, and 16. In all replicates and across all treatments, we evolved populations of 1000 agents for 10,000 generations, starting from a simple ancestor program consisting of a single function with eight no-operation instructions. Each generation, we evaluated all agents in the population individually three times (three trials) where each trial comprised 256 time steps. For a single trial, an agent's fitness was equal to the number of time steps in which its internal state matched the environment state during evaluation. After three trials, an agent's fitness was equal to the minimum fitness value obtained across its three trials. We used a combination of elite and tournament (size four) selection to select which individuals reproduced asexually each generation. We applied mutations to offspring as described in Section 4.3.4. Agents were limited to a maximum of 32 parallel executing threads and a maximum of 32 functions. Functions were limited to a maximum length of 128 instructions. Agents were limited to 128 call states per call stack. The minimum tag reference threshold was 50% (*i.e.*, tags must have at least 50% similarity to successfully reference). All tags were represented as length 16 bit strings.

Statistical Methods

For every run, we extracted the program with the highest fitness after 10,000 generations of evolution. Because the sequence of environmental states experienced by an agent during evaluation are highly variant, we tested each extracted program in 100 trials, using a program's average performance as its fitness in our analyses. For each environment size, we compared the performances of evolved programs across treatments. To determine if any of the treatments were significant ($p < 0.05$) within a set, we performed a Kruskal-Wallis test. For an environment size in which the Kruskal-Wallis test was significant, we performed a post-hoc Dunn's test, applying a Bonferroni correction for multiple comparisons. All statistical analyses were conducted in R 3.3.2 (R Core Team, 2016), and each Dunn's test was

conducted using the FSA package (Ogle, 2017).

4.4.2 Distributed Leader Election Problem

In the distributed leader election problem, a network of agents must unanimously designate a single agent as leader. Agents are each given a unique identifier (UID). Initially, agents are only aware of their own UID and must communicate to resolve the UIDs of other agents. During an election, each agent may vote, and an election is successful if all votes converge to a single, consensus UID. This problem has been used to study the evolution of cooperation in digital systems (Knoester et al., 2007, 2013) and as a benchmark problem to compare the performance of different GP representations in evolving distributed algorithms (Weise and Tang, 2012). A common strategy for successfully electing a leader begins with all agents voting for themselves. Then, agents continuously broadcast their vote, changing it only when they receive a message containing a UID greater than their current vote. This process results in the largest UID propagating through the distributed system as the consensus leader. Alternatively, a similar strategy works for electing the agent with the smallest UID.

We evolved populations of homogeneous distributed systems of SignalGP agents where networks were configured as 5x5 toroidal grids, and agents could only interact with their four neighbors. When evaluating a network, we initialized each agent in the network with a random UID (a number between 1 and 1,000,000). We evaluated distributed systems for 256 time steps. During an evaluation, agents retrieve their UID by executing a `GetUID` instruction. Agents vote with a `SetOpinion` instruction, which sets their opinion (vote) to a value stored in memory, and agents may retrieve their current vote by executing a `GetOpinion` instruction. Agents communicate by exchanging messages, either by sending a message to a single neighbor or by broadcasting a message to all neighboring agents.

After an evaluation, we assigned fitness, F according to Equation 4.1 where V gives the number of valid votes at the end of evaluation, $C_{\max}$ gives the maximum consensus size at

the end of evaluation, $T_{\text{consensus}}$ gives the total number of time steps at full consensus, and S gives the size of the distributed system.

$$F = V + C_{\max} + (T_{\text{consensus}} \times S) \tag{4.1}$$

Distributed systems maximize their fitness by achieving consensus as quickly as possible and maintaining consensus for the duration of their evaluation. Our fitness function rewards partial solutions by taking into account valid votes (*i.e.*, votes that correspond to a UID present in the network) and partial consensus at the end of an evaluation.

We evolved distributed systems in three treatments: one with event-driven messaging and two different imperative messaging treatments. In the event-driven treatment, messages were events that, when received, could trigger a function. In both imperative treatments, messages did not automatically trigger functions; instead, messages were sent to an inbox and needed to be retrieved *via* a `RetrieveMessage` instruction. The difference between the two imperative treatments was in how messages were handled once retrieved. In the fork-on-retrieve imperative treatment, messages act like an internally-generated event when retrieved from an inbox, triggering the function with the closest (above a threshold) matching tag on a new thread. In the copy-on-retrieve imperative treatment, messages are not treated as internal events when retrieved; instead, message contents are loaded into the input memory of the thread that retrieved the message. In the copy-on-retrieve imperative treatment, we augmented the available instruction set with the `Fork` instruction, allowing programs to trivially spawn parallel-executing threads.

Hypothesis

Event-driven SignalGP agents do not need to continuously poll a message inbox to receive messages from neighboring agents, allowing event-driven programs to more efficiently coordinate. Thus, we expected distributed systems evolved in the event-driven treatment to outperform those evolved in the two imperative treatments.

Experimental Parameters

We ran 100 replicates of each treatment. In all replicates of all treatments, we evolved populations of 400 homogeneous distributed systems for 50,000 generations. We initialized populations with a simple ancestor program consisting of a single function with eight no-operation instructions. Selection and reproduction were identical to that of the changing environment problem. Agents were limited to a maximum of 8 parallel executing threads. Agents were limited to a maximum of 4 functions, and function length was limited to a maximum 32 instructions. Agents were limited to 128 call states per call stack. The minimum tag reference threshold was 50%. All tags were represented as length 16 bit strings. The maximum inbox capacity was 8. If a message was received and the inbox was full, the oldest message in the inbox was deleted to make room for the new message.

Statistical Methods

For every replicate across all treatments, we extracted the program that produces the most fit distributed system after 50,000 generations of evolution. As in the changing environment problem, we compared treatments using a Kruskal-Wallis test, and if significant ($p < 0.05$), we performed a post-hoc Dunn's test, applying a Bonferroni correction for multiple comparisons.

4.5 Results and Discussion

4.5.1 Changing Environment Problem

Event-driven strategies outperform imperative strategies

Figure 4.2 shows results for all environment sizes ($K = 2$, 4, 8, and 16). Programs evolved in treatments with fully event-driven SignalGP significantly outperformed those evolved in the imperative treatment across all environments ($p < 10^{-4}$ for all conditions). Across all environments, there was no significant difference in final program performance

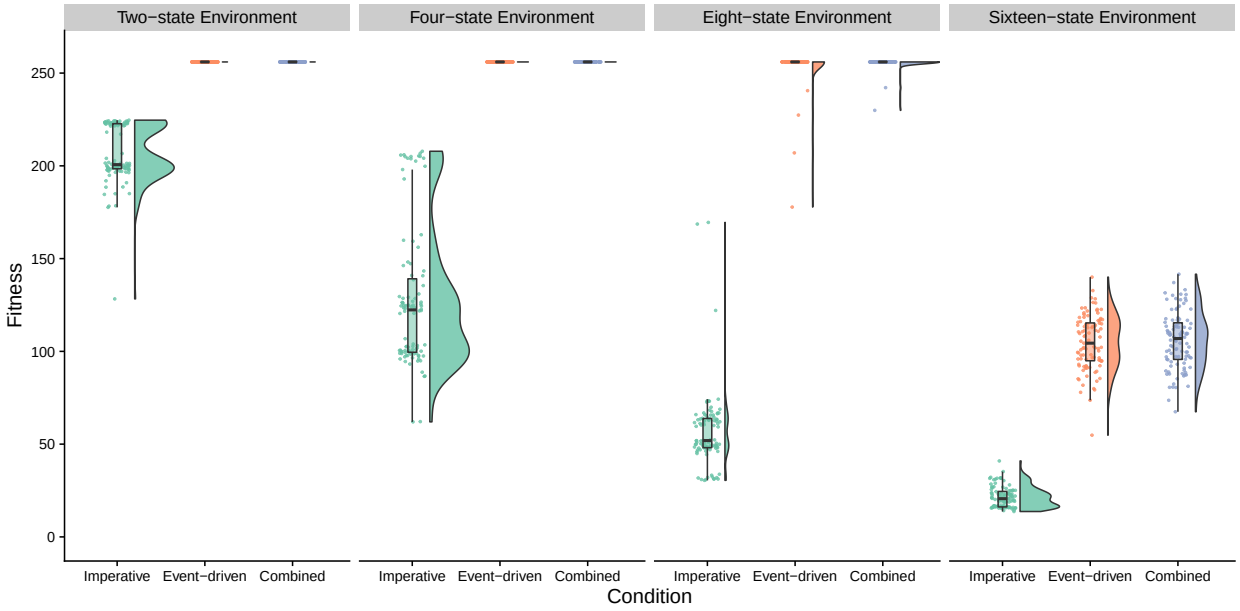


Figure 4.2: **Changing environment problem results across all environments**: two-state environment, four-state environment, eight-state environment, and sixteen-state environment. The raincloud plots (Allen et al., 2019) indicate the fitnesses (each an average over 100 trials) of best performing programs from each replicate.

between the event-driven and combined treatment. See supplementary material for full details on statistical test results (Lalejini, 2018).

Further, only treatments with fully event-driven SignalGP produced programs capable of achieving a perfect fitness of 256. This result is not surprising, as *only* programs that employ an entirely event-driven strategy can achieve a perfect score in multi-state environments. This is because imperative strategies must continuously poll the environment for changes, which decreases the efficiency of their response to an environmental change. This strategy becomes increasingly cumbersome and inefficient as the complexity of the environment increases. In contrast, event-driven responses are triggered automatically *via* the SignalGP virtual hardware, facilitating immediate reactions to environmental changes. This allows event-driven strategies to more effectively scale with environment size than imperative strategies.

Evolution favors event-driven strategies

In the combined treatment, evolution had access to both the event-driven (signal-based) strategy and the imperative (sensor-polling) strategy. As shown in Figure 4.2, performance in the combined treatment did not significantly differ from the event-driven treatment, but significantly exceeded performance in the imperative treatment. However, this result alone does not reveal what strategies were favored in the combined treatment.

To tease this apart, we re-evaluated programs evolved under the combined treatment in two distinct conditions: one in which we deactivated sensors and one in which we deactivated external events. In the deactivated sensors condition, `SenseEnvState` instructions behaved as no-operations, which eliminated the viability of a sensor-based polling strategy. Likewise, the deactivated events re-evaluation condition eliminated the viability of event-driven strategies. Any loss of functionality by programs in these new environments will tease apart the strategies that those programs must have employed.

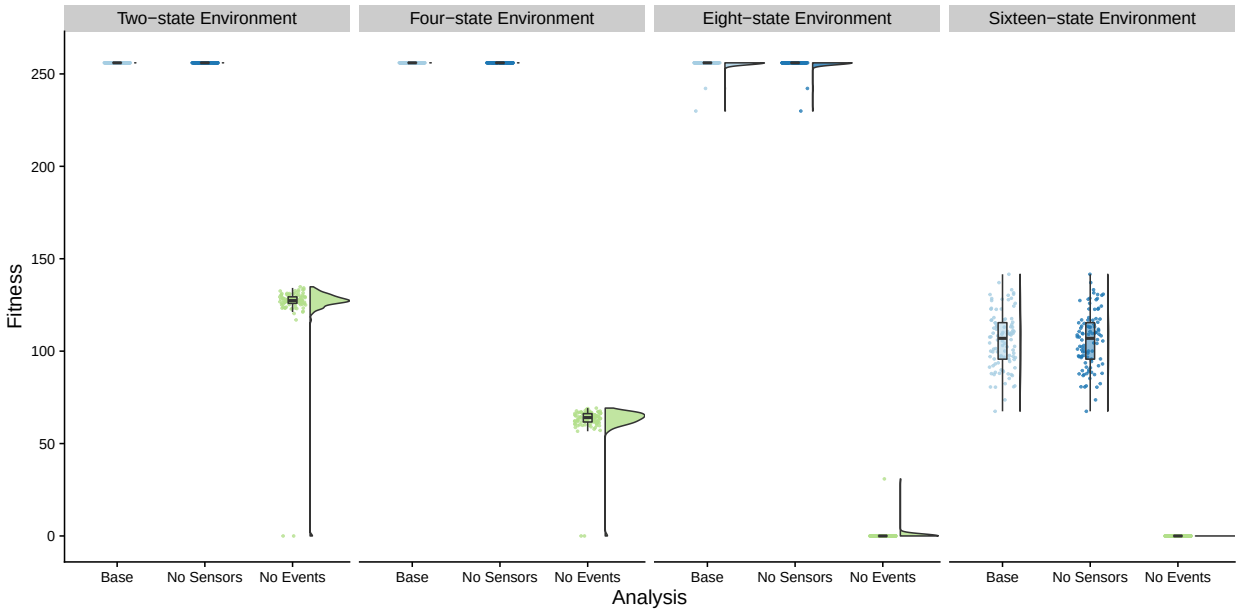


Figure 4.3: **Re-evaluation results for combined condition in the changing environment problem across all environments**: two-state environment, four-state environment, eight-state environment, and sixteen-state environment. The raincloud plots indicate the fitnesses (each an average over 100 trials) of best performing programs from each re-evaluation.

Figure 4.3 shows the results of our re-evaluations. Across all environment sizes, there was no significant difference between program performance in their original combined condition and the deactivated sensors conditions. In contrast, program performances were significantly worse in the deactivated events condition than in the combined condition (all conditions, $p < 10^{-4}$). These data indicate that programs evolved in the combined condition primarily rely on event-driven strategies for the changing environment problem.

4.5.2 Distributed Leader Election Problem

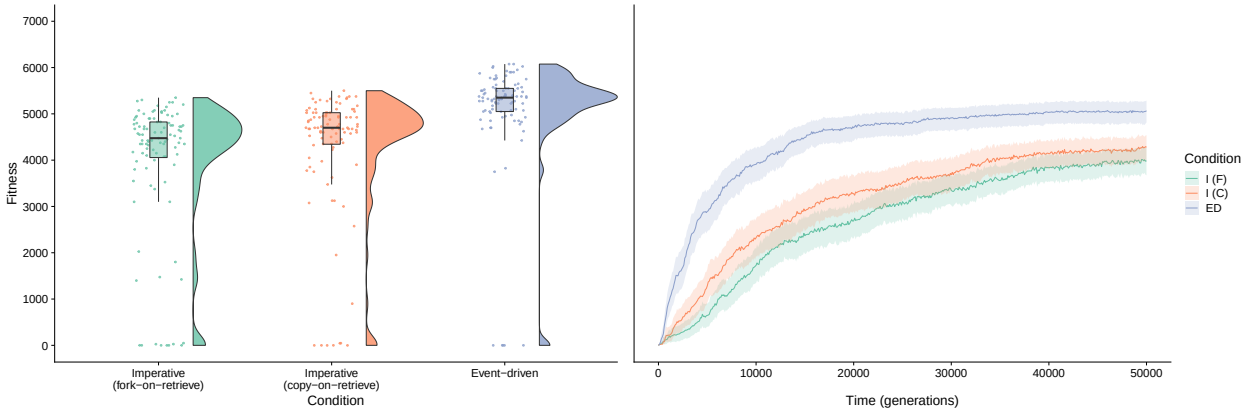


Figure 4.4: **Distributed leader election problem results.** The raincloud plots indicate the fitnesses of best performing distributed systems from each replicate. The time series gives average fitness over time during evolution. The colors in the time series correspond to the colors in the raincloud plots. The shading on fitness trajectories in the time series indicates a bootstrapped 95% confidence interval.

Event-driven networks outperform imperative networks

Figure 4.4 shows the results for the distributed leader election problem. Distributed systems evolved in the event-driven treatment significantly outperformed those evolved in both imperative treatments (fork-on-retrieval and copy-on-retrieval, $p < 10^{-4}$). See supplementary material for full details on statistical test results (Lalejini, 2018).

All three conditions produced distributed systems capable of achieving election consensus. The difference in performances across treatments primarily reflect how quickly consensus is able to be reached within a distributed system. The event-driven programming paradigm

is able to more efficiently encode communication between agents, as it does not require programs to continuously poll for new messages from other agents. Thus, the event-driven paradigm allows signals to propagate more quickly through a distributed system than the imperative paradigm. The time series shown in Figure 4.4 hints that the event-driven SignalGP representation evolves more rapidly for the distributed leader election problem than the imperative variants; however, deeper analyses are required for confirmation.

4.6 Conclusion

We introduced SignalGP, a new type of GP technique designed to provide evolution direct access to the event-driven programming paradigm by augmenting Spector *et al.*'s (Spector et al., 2011b) tag-based modular program framework. We have described and demonstrated SignalGP within the context of linear GP. Additionally, we used SignalGP to explore the value of capturing the event-driven paradigm on two problems where the capacity to react to external signals is critical: the changing environment problem, and the distributed leader election problem. At a minimum, our results show that access to the event-driven programming paradigm allows programs to more efficiently encode agent-agent and agent-environment interactions, resulting in higher performance on both the changing environment and distributed leader election problems. Deeper analyses are needed to tease apart the effects of the event-driven programming paradigm on the evolvability of solutions.

4.6.1 Beyond Linear GP

While this work presents SignalGP in the context of linear GP, the ideas underpinning SignalGP are generalizable across a variety of evolutionary computation systems.

We can imagine SignalGP functions to be black-box input-output machines. Here, we have exclusively put linear sequences of instructions inside these black-boxes, but could have easily put other representations capable of processing inputs (*e.g.*, other forms of GP, Markov brains (Hintze et al., 2017), artificial neural networks, *etc.*). We could even employ

black-boxes with a variety of different contents within the same agent. Encasing a variety of representations within a single agent may complicate the virtual hardware, program evaluation, and mutation operators, but also provides evolution with a toolbox of diverse representations.

As we continue to explore the capabilities of SignalGP, we plan to explore the evolvability of event-driven programs versus imperative programs across a wider set of problems and incorporate comparisons to other GP representations. Further, we plan to extend SignalGP to other representations beyond linear GP and compare their relative capabilities and interactions.

4.7 Software and Data Availability

We implemented SignalGP, the changing environment problem, and the distributed leader election problem using the Empirical scientific software library (Ofria et al., 2020). We conducted all statistical analyses for this work using R version 3.3.2 (R Core Team, 2016). Our source code for test problems, experiment data, and analyses can be found in supplemental material (Lalejini, 2018), which is hosted on GitHub[1].

[1]https://github.com/amlalejini/GECCO-2018-Evolving-Event-driven-Programs-with-SignalGP/

Chapter 5

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

Authors: Alexander Lalejini, Matthew Andres Moreno, and Charles Ofria

This chapter is adapted from (Lalejini et al., 2020b), which has passed peer review and is to appear in Genetic Programming and Evolvable Machines.

5.1 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 (Cramer, 1985; Forsyth, 1981; Koza, 1989). 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 (Hintze et al., 2019; Wilson and Banzhaf, 2008). Here, we introduce and experimen-

tally 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 (Helmuth and Spector, 2015), 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 (Wagner et al., 2007) as well as artificial evolving systems (Huizinga et al., 2016). Moreover, evidence in these evolving systems suggests that modularity can improve the capacity for effective plasticity to arise (Ellefsen et al., 2015; Londe et al., 2015).

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 (Koza, 1992, 1994). Angeline and Pollack developed compression and expansion genetic operators to automatically modularize existing code into libraries of parameterized subroutines (Angeline and Pollack, 1992). Since these foundational advances, significant efforts have been made to allow GP representations to incorporate internal modules (*e.g.*, (Spector, 1996; O'Neill and Ryan, 2000; Binard and Felty, 2007; Walker

and Miller, 2008; Spector et al., 2011b, 2012; Lalejini and Ofria, 2018)), to measure (and select for) modularity in evolving programs (*e.g.*, (Krawiec and Wieloch, 2009; Saini and Spector, 2019, 2020)), and to build "libraries" of reusable code modules accessible to evolving populations of programs (*e.g.*, (Banscherus et al., 2001; Keijzer et al., 2004, 2005; Rosca and Ballard, 1994)).

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 (Lalejini and Ofria, 2018); however, our approach is immediately applicable to any existing tag-enabled GP system, such as tag-addressed Run Transferable Libraries (Keijzer et al., 2004) or PushGP (Spector et al., 2011b). 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.

5.2 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 (Rasmussen et al., 1990).

Inspired by Holland's use of "tags" to facilitate binding and aggregation in complex adaptive systems (Holland, 1990, 1993), Spector *et al.* generalized the use of tags to label and refer to program modules in GP (Spector et al., 2011a,b). 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 (Keijzer et al., 2004; Spector et al., 2011b) or as bit strings (Lalejini and Ofria, 2018). 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 (Spector et al., 2011b).

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 (Keijzer et al., 2004, 2005). 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 (Spector et al., 2011b). 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 (Spector et al., 2012).

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 (Lalejini and Ofria, 2018, 2019b). 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 (Lalejini and Ofria, 2019a). 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.

5.3 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 (Banzhaf and Yamamoto, 2015). 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 (Karlebach and Shamir, 2008). The role of gene regulatory networks in sustaining complex life has inspired varied and abundant computational models of these networks (Cussat-Blanc et al., 2019; Karlebach and Shamir, 2008).

Artificial gene regulatory networks have been used to study how natural gene regulation evolves (Aldana et al., 2007; Crombach and Hogeweg, 2008; Draghi and Wagner, 2009) and as a tool in evolutionary computation to solve challenging control problems (as reviewed by (Cussat-Blanc et al., 2019)). Evolved artificial gene regulatory networks have even been used as indirect encoders, providing a developmental phase to translate genomes into programs (Banzhaf, 2003; Lopes and Costa, 2012) or neural networks (Wróbel and Joachimczak, 2014). La Cava *et al.* demonstrated a form of *epigenetic* regulation for genetic programming where "gene" activation and silencing is learned each generation (La Cava et al., 2015; La Cava and Spector, 2015); 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 (Turner et al., 2017); 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 (Lalejini and Ofria, 2018), which is described in further detail in Section 5.4.1. 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 5.2 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 matches 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 mod-

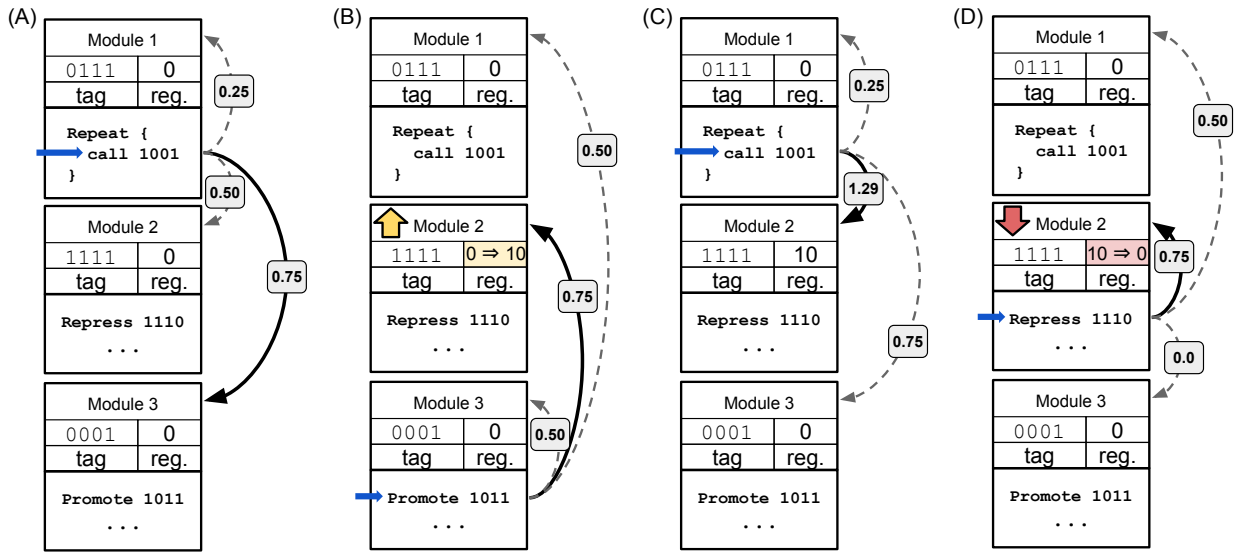


Figure 5.1: **Tag-based genetic regulation example.** This example depicts a simple oscillating regulatory network instantiated using tag-based regulation. In this example, tags are length-4 bit strings. The "raw" match score between two tags equals the number of matching bits between them. Regulation (reg.) modifies match scores for "`call`" instructions according to Equation 5.1. 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.

ification.[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 5.1 below) and thus increases the module's chances of being referenced. Repressor instructions have the opposite effect. Regulatory modifiers can 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,

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

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 5.1 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 (Ofria et al., 2020). Here, we use a simple exponential function to apply a module's regulation modifier to its tag-match score calculations:

$$M_r(t_q, t_m, R_m) = M(t_q, t_m) \times b^{R_m} \tag{5.1}$$

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).

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 5.2 gives a visual representation of Equation 5.1.

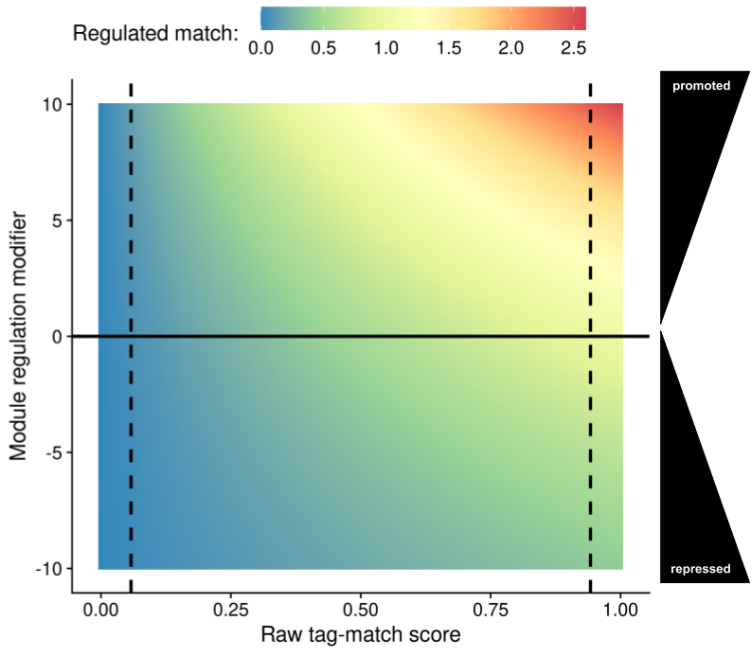


Figure 5.2: **Regulated tag-match score as a function of raw tag-match score and regulatory modifier values according to Equation 5.1.** The horizontal black line indicates a neutral regulatory state; repressed states are below the line, and promoted states are above the line. We expect the raw tag-match score (calculated using the Streak similarity metric, which is described later in Section 5.4.1) 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.

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.

5.4 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.

5.4.1 SignalGP

Here, we provide a general overview of SignalGP; see (Lalejini and Ofria, 2018) 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 (Cassandras, 2014). 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 (Lalejini and Ofria, 2019b).

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 (Downing, 2015) 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 the second. The equation below overviews how the Streak metric computes the similarity (S) between two tags (t_q and t_m):

[2]We make a slight modification to Downing's matching procedure due to an error in its mathematical derivation, as detailed in the supplement (Lalejini et al., 2021).

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

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 (Lalejini et al., 2021).

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 (Lalejini and Ofria, 2019b). 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

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

regulation to SignalGP (as described in Section 5.3). We supplemented the instruction set with regulatory instructions (Table 5.1) that use tag-based referencing to target internal functions. In this work, we apply regulation to function references using Equation 5.1. Our full instruction set, including descriptions of each instruction, can be found in our supplemental material (Lalejini et al., 2021).

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 (Lalejini et al., 2017) 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 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.

5.4.2 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.

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.

5.4.3 Contextual-signal Problem

The contextual-signal problem is inspired by Skocelas and DeVries' method for verifying the functionality of recurrent neural network implementations (Skocelas and DeVries, 2020). 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 5.2 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 5.2: **Input signal sequences for the contextual-signal problem.**

We evaluate programs on each of the 16 possible sequences of input signals (Table 5.2); 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.

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 (Helmuth et al., 2015) in which each combination of input signals (*i.e.*, row in Table 5.2) constituted a single test case.

5.4.4 Boolean-logic Calculator Problem

Inspired by Yeboah-Antwi's PushCalc system (Yeboah-Antwi, 2012), 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 5.3 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 5.3: **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 (Lalejini et al., 2021) and contained 442 and 5810 test cases, respectively. Each generation, we sample 20 test cases from the training set, and 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

[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.

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.

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 (Hernandez et al., 2019), 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.

5.4.5 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 5.4.2). A program is considered a solution if it responds correctly to

all 16 input signals during evaluation.

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.

5.4.6 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 (Ofria et al., 2020), and we conducted all statistical analyses using R version 4.0.2 (R Core Team,

2020). We used the reshape2 Wickham (2020) R package and the tidyverse (Wickham et al., 2019) collection of R packages to wrangle data. We used the following R packages for graphing and visualization: ggplot2 (Wickham et al., 2020), cowplot (Wilke, 2020), viridis (Garnier, 2018), Color Brewer (Harrower and Brewer, 2003; Neuwirth, 2014), and igraph (Csardi and Nepusz, 2006). We used R markdown (Allaire et al., 2020) and bookdown (Xie, 2020) to generate web-enabled supplemental material. Our source code for experiments and analyses, along with guides for replication, can be found in supplemental material (Lalejini et al., 2021), 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 (Lalejini et al., 2021)).

5.5 Results and Discussion

5.5.1 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 5.5), contextual-signal problem (Section 5.5), and Boolean-logic calculator problem (Section 5.5). 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.

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

Table 5.4: **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: $p < 10^{-15}$).

Signal-counting Problem

Table 5.4 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 5.3).

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

[4]We verified this claim by hand-coding solutions that rely on global memory and flow-control instructions (supplemental Section 12 (Lalejini et al., 2021)).

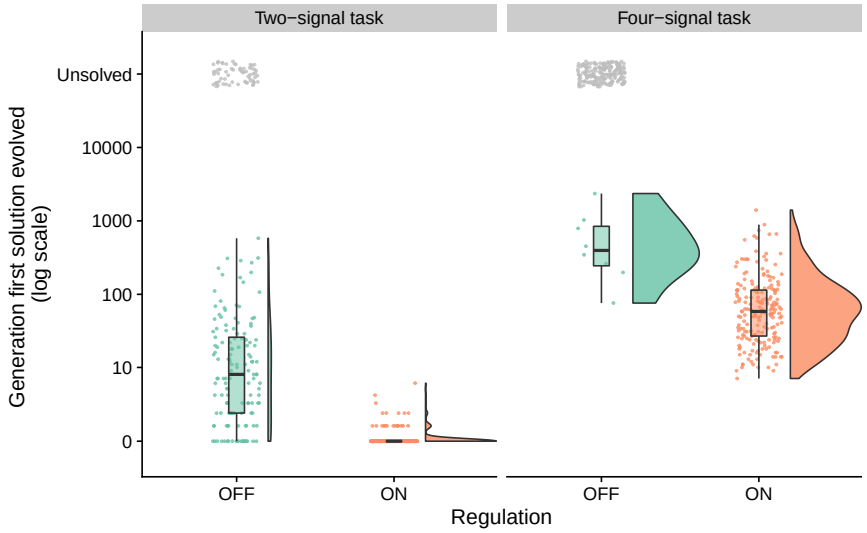
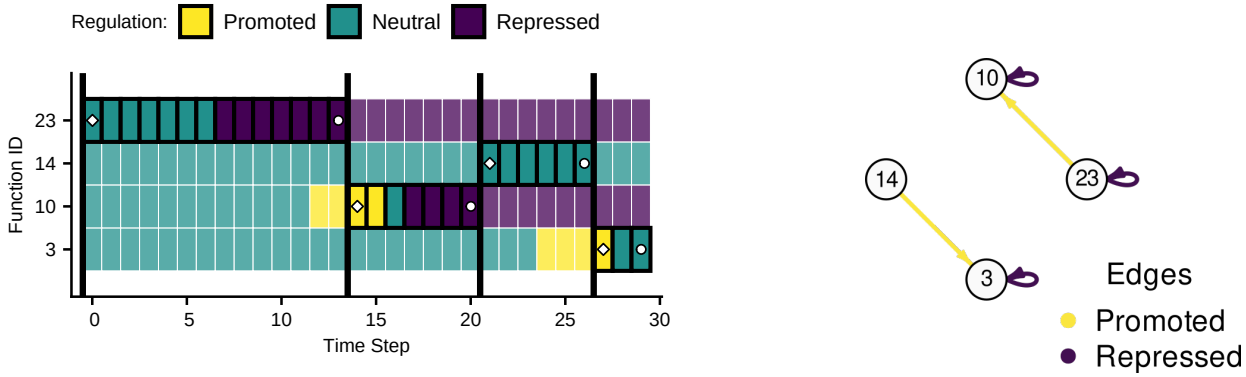


Figure 5.3: **Generation at which first solution evolved (log scale) in each successful replicate for the signal-counting problem (raincloud plot (Allen et al., 2019)).** We show data from only those problem difficulties in which solutions evolved (two- and four-signal problems). Gray points indicate 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: $p < 9 \times 10^{-05}$).

	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 5.5: **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.

exclusively on their global memory buffer to differentiate their behavior (see supplemental Section 7 (Lalejini et al., 2021)). Table 5.5 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.



(a) Module regulation over time.

(b) Regulatory network.

Figure 5.4: **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 5.4 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.

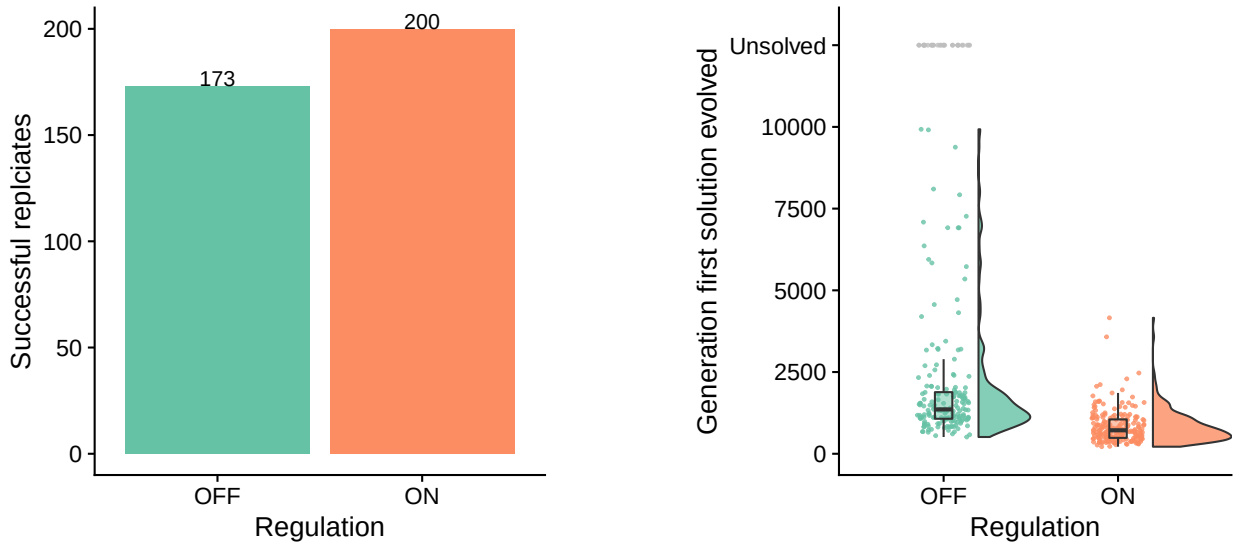
Indeed, across all problem difficulties, we observed that successful regulatory networks generally contained more repression relationships than promotion relationships between functions (supplemental Section 7 (Lalejini et al., 2021)). Independent knockouts of up-regulation and down-regulation confirm 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 (Lalejini et al., 2021)).

Contextual-signal Problem

Figure 5.5a 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 5.5b).

We used knockout experiments to identify the mechanisms underlying each solution's



(a) Successful replicates.

(b) Generations elapsed before solution.

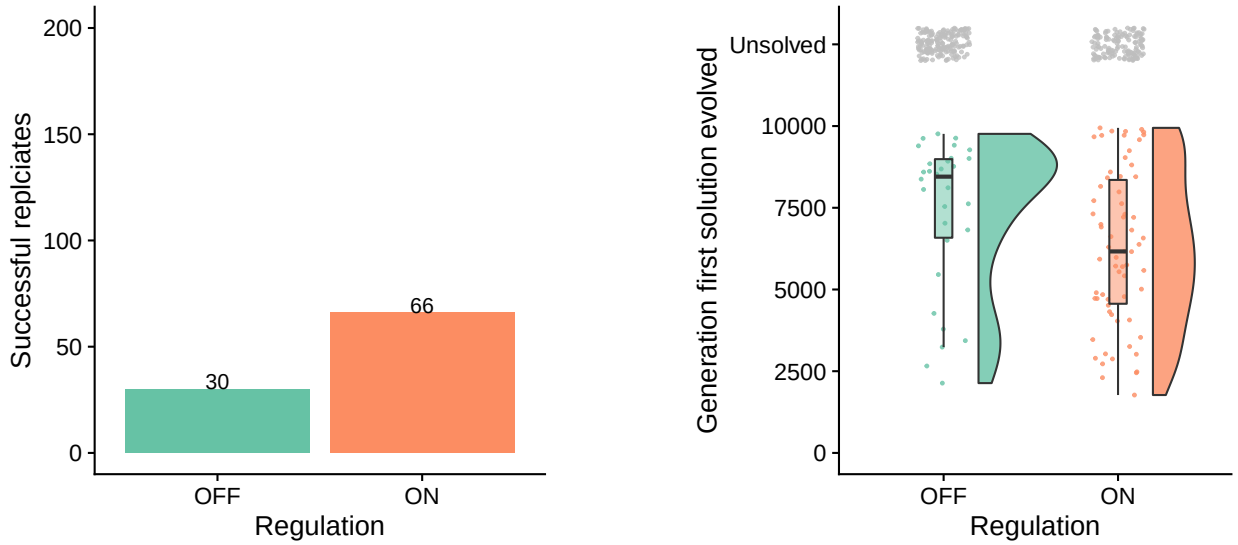
Figure 5.5: **Contextual-signal problem-solving performance.** (a) shows the number of successful replicates for the regulation-off and regulation-on conditions on the contextual-signal problem. The regulation-off condition was less successful than the regulation-on condition (Fisher's exact test: $p < 6 \times 10^{-9}$). (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: $p < 10^{-15}$).

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 (Lalejini et al., 2021)). 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 (Lale-

jini et al., 2021)). This result suggests that we should expect different problems to select for different forms of gene regulatory networks.

Boolean-logic Calculator Problem



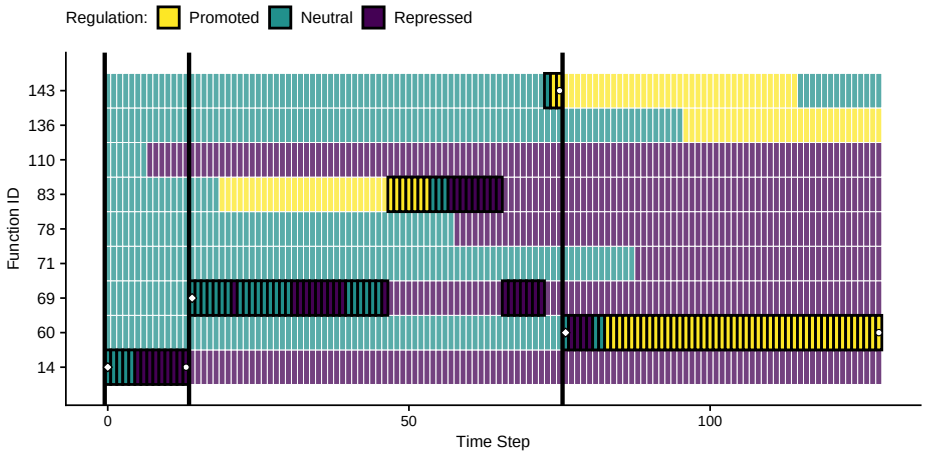
(a) Successful replicates.

(b) Generations elapsed before solution.

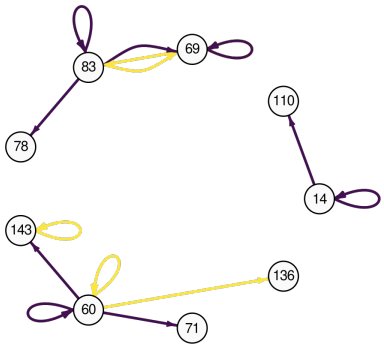
Figure 5.6: **Boolean-logic calculator problem-solving performance.** (a) shows the number of successful replicates for the regulation-off and regulation-on conditions on the Boolean-logic calculator problem. The regulation-off condition was less successful than the regulation-on condition (Fisher's 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 5.6a 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 5.6b).

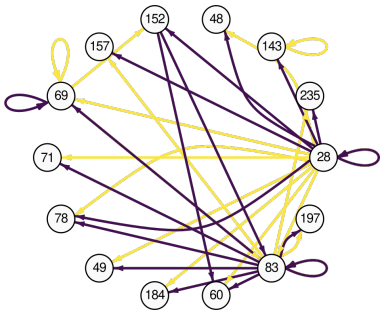
As in previous experiments, we conducted knockout experiments to identify the mechanisms underlying each solution's strategy. To compute any of the Boolean logic operations, programs *must* make use of the global memory buffer to store numeric inputs (operands)



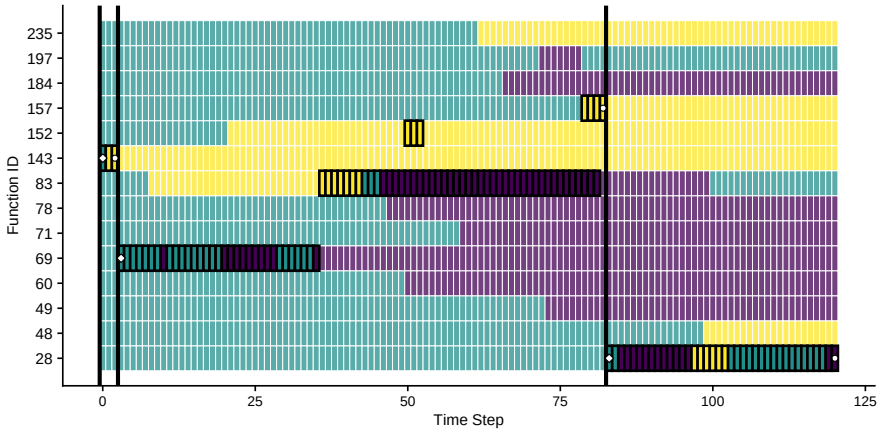
(a) Module regulation over time for a NAND operation.



(b) NAND regulatory network.



(c) NOR regulatory network.



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

Figure 5.7: **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 5.4.

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 (Lalejini et al., 2021)).

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 5.7 overviews the execution of a representative program evolved to solve the Boolean-logic calculator problem. Specifically, Figure 5.7 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).

5.5.2 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 per-

fect 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 5.8 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 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 (Gibson and Dworkin, 2004; Paaby and Rockman, 2014). Such cryptic variation has been shown to help GP systems escape local optima, improving overall problem-solving performance (Turner and Miller, 2015). 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 (Wilke and Adami, 2002)). 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 per-

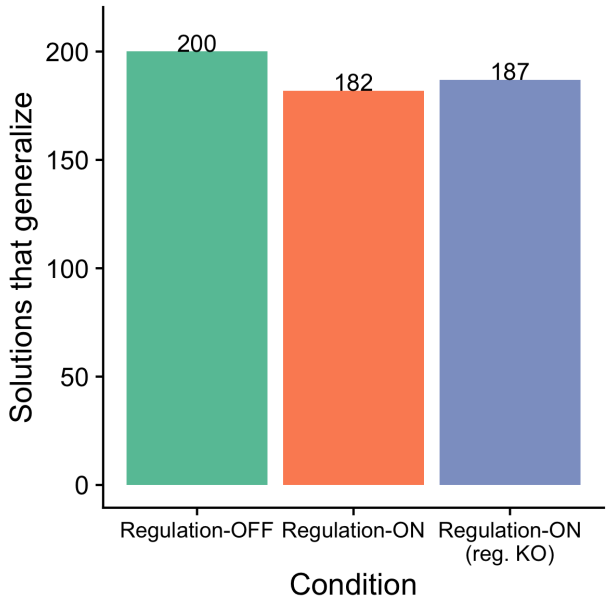


Figure 5.8: **The number of evolved solutions that generalize on the independent-signal problem.** The difference in number of solutions that generalize between the regulation-on and regulation-off conditions is statistically significant (Fisher's exact test: $p < 6 \times 10^{-06}$). The "Regulation-ON (reg. KO)" condition comprises the solutions from the Regulation-on condition, except with regulatory instructions knocked out (*i.e.*, replaced with no-operation instructions).

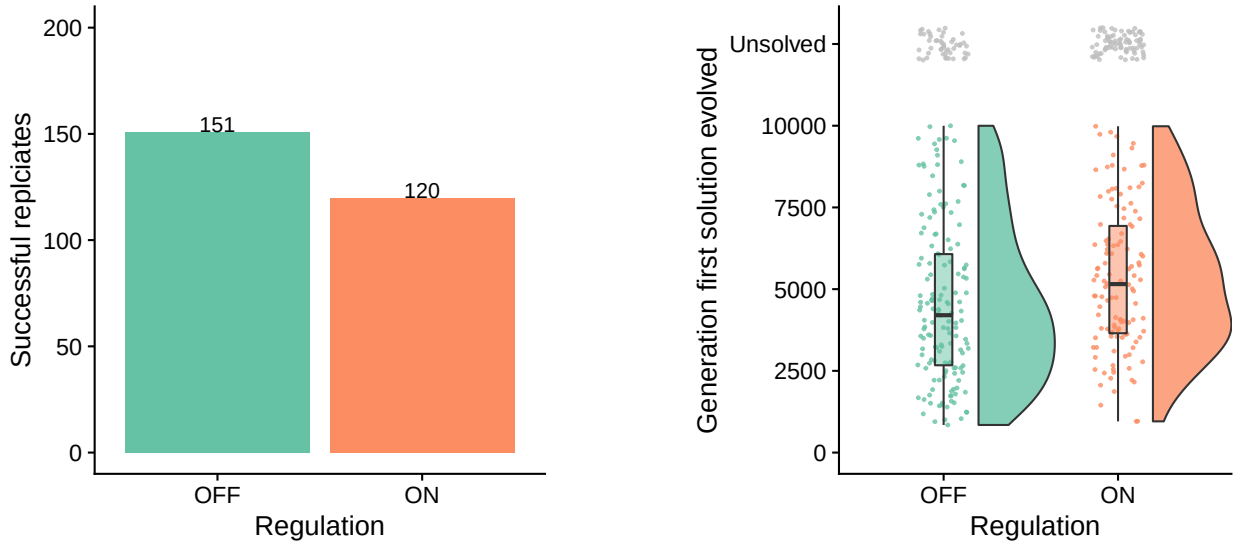
formed more thorough evaluations of programs using multiple random permutations of input sequences instead of one. In more challenging problems, however, more thorough evaluations can come at the cost of substantial computational effort.

5.5.3 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 5.4.4 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 5.5. 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.

Figure 5.9: **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 5.4.4),

except we presented inputs in postfix notation instead of prefix notation. Figure 5.9a 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 5.5). 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 5.9b). 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 (Lalejini et al., 2021)).

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.

5.6 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 (Lalejini et al., 2020a). 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 (Lones et al., 2014, 2013; Lones and Tyrrell, 2004).

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 5.4 and 5.7) 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 (Bender, 2002; Jablonka and Raz, 2009). For example, epigenetics is used in combination with gene regulation for cell-type differentiation in multicellular organisms (Mohn and Schübeler, 2009; Smith and Meissner, 2013) and caste determination in some species of eusocial insects (Weiner and Toth, 2012). 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 (Lalejini et al., 2020a). 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 (La Cava et al., 2015; La Cava and Spector, 2015; Ricalde and Banzhaf, 2017); as such, we plan to apply our insights back to automatic program synthesis.

Chapter 6

Tag-accessed Memory for Genetic Programming

Authors: Alexander Lalejini and Charles Ofria

This chapter is adapted from (Lalejini and Ofria, 2019a), which appeared in the companion proceedings of the 2019 Genetic and Evolutionary Computation Conference.

6.1 Introduction

Here, we demonstrate the use of tags (evolvable labels that can be specified with imperfect matching) to identify memory positions in genetic programming (GP). Specifically, we conducted a series of experiments using simple linear GP representations on five problems from the general program synthesis benchmark suite (Helmuth and Spector, 2015). We show that tag-indexed memory does not substantively affect problem solving success relative to more traditional, direct-indexed memory.

In traditional software engineering, human programmers create variables with unique names to specify data that they are working with. These variables are inherently associated with locations in memory that are accessed by using the variable's name. This technique for referencing values in memory is intentionally rigid, requiring programmers to precisely name the data they want to reference, and imprecision results in syntactic errors. Many traditional GP systems that give genetic programs access to memory (*e.g.*, indexable memory registers)

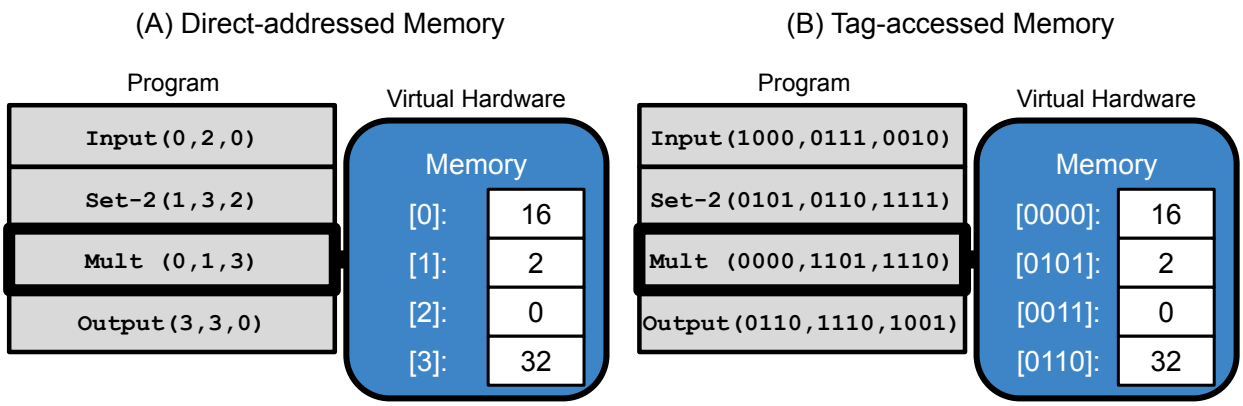


Figure 6.1: **Examples of (A) direct-indexed memory and (B) tag-accessed memory.** The programs in (A) and (B) behave identically: both request input to the first memory register, set the second memory register to the terminal value "2", place the result of multiplying the contents of the first two memory registers into the fourth memory register, and output the contents of the fourth register. Here, we show the state of memory after the `Mult` instruction has been executed. Note that not all instructions use all three arguments.

use similarly rigid naming schemes where memory is numerically indexed, and mutation operators must guarantee the validity of memory-referencing instructions. Interestingly, although exact naming is the most intuitive referencing mechanism for human programmers, evolution in other contexts (such as identifying modules to run (Lalejini and Ofria, 2019b)) has been shown to be more successful when program references are allowed to be inexact. Beyond computer code, robustness to perturbations is also thought to be important in the evolution of complex biological systems (Kitano, 2004).

Tags are evolvable labels that give genetic programs a flexible mechanism for specification, originally used by Holland in genetic algorithms (Holland, 1993) and refined by Spector *et al.* for GP (Spector et al., 2011b). To facilitate *inexact* referencing, the similarity (or dissimilarity) between any two tags must be quantifiable; a referring tag can always reference the closest matching referent tag. Here, we continue to expand the integration of tags into linear GP by allowing instructions to use tags to identify positions in memory (as needed for their function). All instructions have three tag-based arguments, each of which is represented as a length-16 bit string and compared using Hamming distance to measure similarity. Our instruction set allows programs to perform basic computations, manipulate

memory contents, and control execution flow (see supplemental material (Lalejini, 2019) for details). Programs are executed in the context of a virtual CPU that gives them access to 16 statically tagged memory registers used for storing data for performing computations. Figure 6.1 contrasts tag-based memory with direct-indexed memory. Tag-based instruction arguments reference the memory position with the *closest matching* tag; as such, argument tags need not *exactly* match any of the tags associated with memory positions. This inexactness makes program phenotypes more robust to minor genetic perturbations, smoothing the genotype-phenotype mapping relative to more traditional memory-indexing techniques.

6.2 Experimental Results

We compared the performance of our simple linear GP to a variant that replaced the tag-accessed memory with memory indexed with direct arguments (which is more akin to memory access in traditional linear GP (Brameier and Banzhaf, 2007)). We evolved programs using the lexicase parent selection algorithm Helmuth et al. (2015) to solve five problems from Helmuth and Spector's program synthesis benchmark suite (Helmuth and Spector, 2015): number IO, smallest, median, grade, and for loop index. For each problem, we added custom instructions to the instruction set that facilitated loading test case inputs into memory and returning program responses. We used the same training and testing sets when evaluating programs as Helmuth and Spector in (Helmuth and Spector, 2015). We measured performance by counting the number of successful runs (*i.e.*, runs that produced a perfect solution).

For each experimental condition, we evolved 50 replicate populations of 500 individuals (for 100 generations for the number IO problem and 500 generations for all other problems), giving each replicate a unique random number seed. We propagated programs asexually and applied mutations to offspring (single-instruction insertions, deletions, and substitutions at a per-instruction rate of 0.005 each and multi-instruction sequence duplications and deletions at a per-program rate of 0.05). The relative success of these two memory-indexing techniques

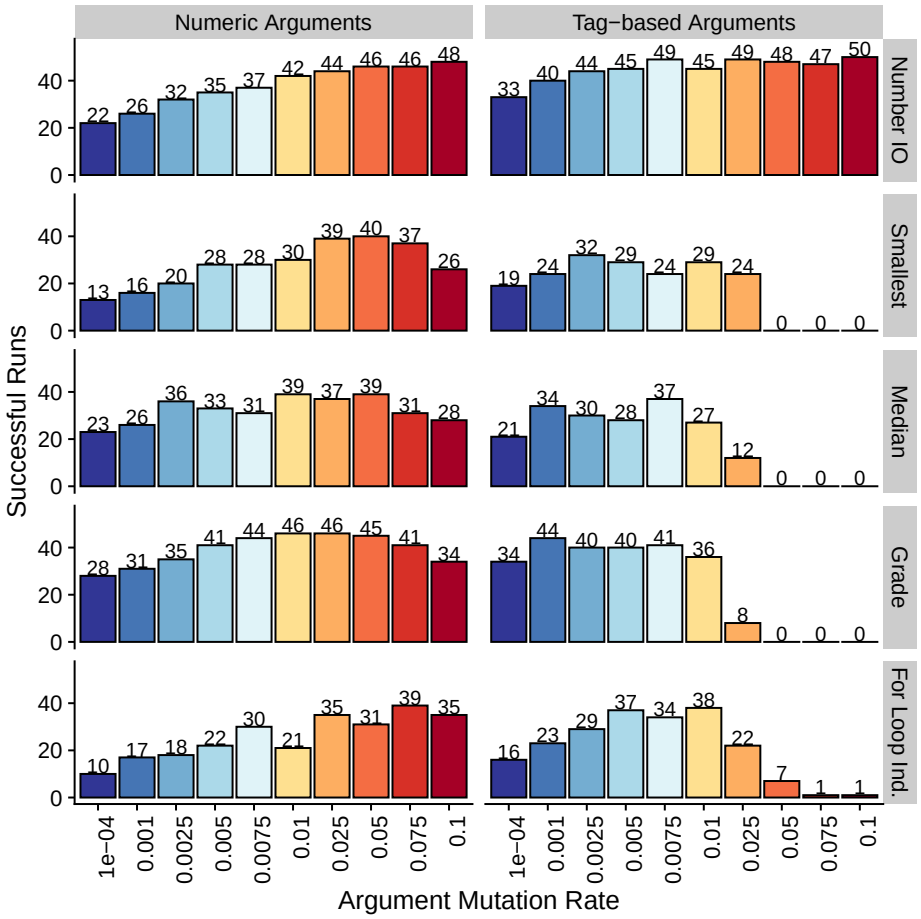


Figure 6.2: **Number of successful runs** when using tag-accessed memory (right column) versus using traditional direct-indexed memory (left column) across five problems and ten instruction argument mutation rates (after 100 generations for number IO and 500 generations for all other problems).

is influenced by how (and at what rate) we mutate instruction arguments. As such, we mutated tag-based arguments (per-bit) and traditional arguments (per-argument) at the following ten rates: 0.0001, 0.001, 0.0025, 0.005, 0.0075, 0.01, 0.025, 0.05, 0.075, and 0.1. See our online supplemental material (Lalejini, 2019) for source code, details on problem-specific configurations (*e.g.*, program evaluation time, *etc.*), and for our more detailed analyses.

Figure 6.2 shows the performance of tag-accessed memory and direct-indexed memory for each problem and mutation rate. For each problem, we selected the best (most successful) mutation rate for tag-based arguments and the best mutation rate for traditional arguments. We compared the performance of tag-based arguments and numeric arguments

at these "optimal" mutation rates, and we tested for we tested for statistical significance using Fisher's exact test (with a significance threshold of 0.05). Across all problems, there was no statistically significant difference between tag-based instruction arguments (tag-accessed memory) and numeric instruction arguments (direct-indexed memory).

Figure 6.2 also seems to indicate that tag-accessed memory is more sensitive to mutation rate than direct-addressed memory. Indeed, on the smallest, median, and grade problems, three mutation rates resulted in no solutions in our tag-based argument treatment, whereas all mutation rates in the numeric arguments treatment resulted in at least one solution on all problems. This result does not *necessarily* indicate that tag-based arguments are less mutationally robust than numeric arguments. We mutated numeric arguments at a per-argument rate, and as such, a mutation rate of 0.1 is an expected one mutation for every ten arguments mutated. We mutated tag-based arguments at a *per-bit* rate, and as such, a mutation rate of 0.1 is an expected one to two bit flips per mutated tag. That is, in practice, the 0.1 per-bit mutation rate for tags is substantially higher than the 0.1 per-argument mutation rate for numeric arguments. More experiments are needed to quantify tag-based argument and numeric argument sensitivity to mutation.

6.3 Conclusion

Our preliminary experiments show that, under favorable mutation rates, both tag-accessed and direct-indexed memory achieve statistically equivalent performance. Because tag-based instruction arguments index into the *closest matching* memory register, single bit-flip mutations may be neutral (not affecting the program's behavior), which affords programs robustness to minor genetic perturbations. The down-side to a more robust genetic encoding for instruction arguments is that mutations are less able to generate novel phenotypic variation (program behavior). For the relatively simple program synthesis problems used in our experiments, the capacity of our GP system to generate novel phenotypic variation is likely more important than robustness to mutation. Future work will continue to

explore the efficacy of tag-accessed memory, supplementing bit-flip mutation operators with more impactful mutation operators that allow tag-mutations to more easily generate novel phenotypic variation. Future work will also investigate the possibility of coevolving register labels (tags) with programs, allowing evolution to adjust the adjacency of memory registers in tag-space.

Chapter 7

Conclusions

The capacity for adaptive phenotypic plasticity is an important characteristic of adaptive systems, including both biological organisms and solutions to computational problems (*e.g.*, computer programs). In this dissertation, I used digital evolution experiments to explore the process by which adaptive plasticity evolves and to illuminate its effects on subsequent evolutionary dynamics. I have also demonstrated the value of phenotypic plasticity in the context of genetic programming, introducing novel techniques that allow us to evolve more dynamically responsive computer programs.

7.1 Contributions

In summary, this dissertation makes the following contributions:

- In **Chapter 2**, I found that both environmental change rate and mutation rate influence the likelihood for adaptive phenotypic plasticity to evolve in populations of digital organisms. By analyzing the lineages of plastic organisms, I identified that unconditional trait expression and imperfect forms of phenotypic plasticity are important evolutionary building blocks for adaptive plasticity.

- In **Chapter 3**, I used populations of digital organisms to empirically test whether the evolution of adaptive phenotypic plasticity alters evolutionary dynamics and influences evolutionary outcomes in cyclically changing environments. I found that the

evolution of adaptive phenotypic plasticity stabilizes populations against environmental fluctuations and constrains subsequent evolution. By buffering populations against environmental change, adaptive plasticity improved novel trait retention and reduced the accumulation of deleterious mutations relative to non-plastic populations evolved in an otherwise identical environment.

- In **Chapter 4**, I introduced SignalGP, a novel genetic programming technique for evolving event-driven computer programs. I showed that SignalGP allows us to evolve programs better able to rapidly interact with the environment or with other agents.

- In **Chapter 5**, I developed tag-based genetic regulation, a new genetic programming technique that allows programs to dynamically adjust the code modules that they express. I described how to augment existing genetic programming systems with tag-based regulation, and I showed that tag-based regulation improves problem-solving performance on context-dependent problems where programs must adjust how they respond to current inputs based on prior inputs.

- In **Chapter 6**, I proposed tag-accessed memory, a new mechanism for labeling and identifying memory positions in genetic programming. With preliminary experiments, I found that, under favorable mutation rates, both tag-accessed memory and conventional direct-indexed memory achieve similar performance on a range of program synthesis problems. These results indicate promise as tag-accessed memory is further integrated into tag-enabled genetic programming systems.

7.2 Future Directions

Thus far, I have focused on using digital evolution techniques to study general principles about evolutionary processes (Chapters 2 and 3) and applying inspiration from biology to evolutionary computing (Chapters 4, 5, and 6). There are many future directions with which to take my research. Here, I highlight just two (of many) planned directions: broadened ap-

plications of SignalGP and transferring techniques and insights from evolutionary computing back into laboratory-based experimental evolution.

7.2.1 Broadened applications of SignalGP

There are many extensions and applications of SignalGP that I hope to either pursue, facilitate, or eagerly watch others carry out. Given that SignalGP is a new genetic programming representation able to address new types of problems, there are many fundamental topics ripe for exploration. For example, I have yet to investigate the effects of crossover or any form of horizontal gene transfer in SignalGP. In addition, most of the program synthesis problems that I have applied SignalGP to have been primarily diagnostic; in the future, I look forward to more broadly benchmarking SignalGP on challenging event-driven program synthesis problems.

Below, I discuss in detail two additional extensions to my work: positioning SignalGP as a model organism for digital experimental evolution and developing a multi-representation version of SignalGP that can bring together the most effective aspects of different GP representations.

SignalGP as model organism for digital experimental evolution

In Chapters 4 and 5, I demonstrated SignalGP in an applied genetic programming context. However, one of my original motivations for developing SignalGP was to design a new approach for representing self-replicating computer programs that emphasizes dynamic interactions among digital organisms and between digital organisms and their environment. In Chapters 2 and 3, I experimentally evolved relatively simple forms of adaptive phenotypic plasticity; that is, to achieve an optimal form of plasticity, digital organisms needed to toggle between two relatively simple phenotypes based on environmental conditions. In some of my earliest (unpublished) experiments using Avida, I attempted to evolve more complex forms of adaptive plasticity (*e.g.*, the capacity to independently regulate many traits). Such adaptive plasticity proved to be challenging to evolve (and thus study) in Avida because genomes are

expressed procedurally: actions are performed one at a time in a single chain of execution and must explicitly check for new sensory information. As such, organisms in Avida must continuously generate explicit queries in order to identify (and react to) any changes in their environment. Further, the genetic mechanisms in Avida for encoding adaptive plasticity are not easily scalable; I found that plastic programs typically regulate a few key instructions to alter their phenotype and cannot as easily toggle large sequences of instructions on or off.

There are many different model organisms used in biological research, each with their own benefits and shortcomings for conducting evolution experiments. Yet, historically, there have been very few different forms of self-replicating computer programs used in digital evolution experiments. I envision SignalGP as providing a useful representation for experiments where digital organisms need to dynamically react to signals from the environment or from other agents. I look forward to expanding on my work presented in Chapter 5 on tag-based regulation to further enhance gene regulation and epigenetic inheritance in SignalGP.

I am most excited, however, by ongoing digital evolution work using SignalGP that is being conducted by *other* researchers. For example, Matthew Andres Moreno has incorporated SignalGP into the DISHTINY digital evolution platform (Moreno and Ofria, 2019; Lalejini et al., 2020a). DISHTINY provides independently replicating digital organisms (cells) with the ability and the incentive to unite into higher-level individuals. The system demonstrates *de novo* major evolutionary transitions in individuality without direct interventions by the experimenter. Individual cells are SignalGP agents, which allows them to respond to the environment and communicate with one another in a signal-driven context. Moreno has developed novel approaches to SignalGP module regulation and execution (Moreno, 2021), distributed agent-agent interactions (Moreno and Ofria, 2020), and even implemented substantially more efficient versions of the SignalGP virtual hardware (Moreno and Rodriguez Papa, 2021).

Multi-representation SignalGP

In Chapters 4 and 5, SignalGP functions (modules) associate a tag with the start of a linear sequence of instructions. We can imagine these functions to be black-box input-output machines: when called or triggered by an event, a function is provided with input and can return output, writing to memory or generating signals as it goes. Instead of constructing functions with linear sequences of instructions, we could use other computational substrates capable of receiving input and producing output (*e.g.*, other GP representations, artificial neural networks, Markov brains, hard-coded modules, *etc.*). We could even employ a variety of representations within a single SignalGP agent.

SignalGP's tag-based naming scheme enables this black-box metaphor. Functions composed of different representations can still refer to one another via tags, and events are agnostic to the underlying representation used to handle them, requiring only that the representation is capable of processing event-specific data. Allowing for such multi-representation agents may complicate the SignalGP virtual hardware, program evaluation, and mutation operators, but in exchange, it would provide evolution with a toolbox of diverse representations.

Hintze et al. proposed and demonstrated the evolutionary "buffet method" where Markov brains could be composed of heterogeneous computational substrates, allowing evolution to work out the most appropriate representation for a given problem (Hintze et al., 2019). Indeed, Hintze et al. observed that different problems produced solutions with different distributions of component types, making buffet-style Markov brains a flexible representation for solving a range of different types of problems. Multi-representation SignalGP provides an unexplored, alternative approach to evolving multi-representation agents, bringing the buffet method into an event-driven context.

7.2.2 Transferring algorithms from evolutionary computing to laboratory-based experimental evolution

My eventual goal is to work, teach, and mentor seamlessly across computational and laboratory evolution systems, cyclically transferring insights from biology to evolutionary computing and back again. However, my research thus far has been entirely computational in nature, and I have yet to work with laboratory experimental evolution systems. To this end, my immediate future plans are to learn more about microbial experimental evolution as a postdoctoral researcher in Dr. Luis Zaman's laboratory at the University of Michigan.

In collaboration with Dr. Zaman's lab, I will use cutting-edge laboratory automation and equipment fabrication technology to conduct evolution experiments that take advantage of precise environmental controls and high-resolution sensor feedback. One of the initial ways we aim to bridge computational and laboratory experimental evolution is by converting modern evolutionary computing algorithms to direct the evolution of microbial populations.

Directed evolution wields artificial selection as a tool to generate biomolecules and organisms with enhanced or novel functional traits (Chen and Arnold, 1993; Sánchez et al., 2021). The scale and specificity of artificial selection has been revolutionized by a deeper understanding of evolutionary and molecular biology in combination with technological innovations in sequencing, sensing, and laboratory automation. These advances have cultivated growing interest in directing the evolution of whole microbial communities with functions that can be harnessed in medicine, biotechnology, and agriculture (Sánchez et al., 2021). Yet, the procedures for selecting which communities to propagate tend to focus on those that are the most fit, leaving more nuanced techniques unexplored, despite many highly effective results in evolutionary computation.

Indeed, since the 1960s, evolutionary computing has harnessed the principles of natural evolution as a general purpose search engine for solving challenging computational and engineering problems (Bäck et al., 1997). As evolutionary computing has matured, the field has identified common pitfalls in directing evolution and has developed a robust toolbox of

algorithms to more effectively steer evolutionary processes.

As in evolutionary computing, directed evolution *in vitro* begins with a library of variants (*e.g.*, communities, genomes, or molecules). Variants are scored according to the phenotypic trait (or set of traits) of interest, and the variants with the "best" traits are selected and used to produce the next generation. The method by which we select variants to propagate from generation to generation dramatically influences the success of directing evolution. In complex fitness landscapes, using only the overall "best" variants to produce the next generation can lead to premature convergence on local optima and adaptive stagnation; this is especially likely in scenarios with multiple objectives that have inherent functional trade-offs. These pitfalls are well studied in evolutionary computation and have motivated new selection schemes that have dramatically improved the quality and diversity of evolved solutions.

I plan to apply modern evolutionary computing selection algorithms (*e.g.*, novelty-based algorithms, quality-diversity algorithms, lexicase-based algorithms, and multi-objective algorithms) to steer the evolution of microbial communities. Eventually, we aim to further bridge digital and microbial experimental evolution by coevolving communities of digital organisms with microbial communities in real time, giving each influence over aspects of the other's environment. Such a hybrid experimental evolution platform would provide the opportunity to investigate the *de novo* evolution of feedbacks between different ecosystems.

BIBLIOGRAPHY

BIBLIOGRAPHY

Adami, C. and Brown, C. T. (1994). Evolutionary Learning in the 2D Artificial Life System Avida. *arXiv:adap-org/9405003*. arXiv: adap-org/9405003.

Adami, C., Ofria, C., and Collier, T. C. (2000). Evolution of biological complexity. *Proceedings of the National Academy of Sciences*, 97(9):4463–4468.

Ahlmann-Eltze, C. and Patil, I. (2021). *ggsignif: Significance Brackets for 'ggplot2'*. R package version 0.6.1.

Alberts, B., Johnson, A., Lewis, J., Raff, M., Roberts, K., and Walter, P., editors (2002). *Molecular biology of the cell*. Garland Science, New York, 4th ed edition.

Aldana, M., Balleza, E., Kauffman, S., and Resendiz, O. (2007). Robustness and evolvability in genetic regulatory networks. *Journal of Theoretical Biology*, 245(3):433–448.

Allaire, J., Xie, Y., McPherson, J., Luraschi, J., Ushey, K., Atkins, A., Wickham, H., Cheng, J., Chang, W., and Iannone, R. (2020). *rmarkdown: Dynamic Documents for R*. R package version 2.6.

Allen, M., Poggiali, D., Whitaker, K., Marshall, T. R., and Kievit, R. A. (2019). Raincloud plots: a multi-platform tool for robust data visualization. *Wellcome Open Research*, 4:63.

Ancel, L. W. (2000). Undermining the Baldwin Expediting Effect: Does Phenotypic Plasticity Accelerate Evolution? *Theoretical Population Biology*, 58(4):307–319.

Angeline, P. J. and Pollack, J. B. (1992). The evolutionary induction of subroutines. In *Proceedings of the fourteenth annual conference of the cognitive science society*, pages 236–241. Bloomington, Indiana.

Banscherus, D., Banzhaf, W., and Dittrich, P. (2001). Hierarchical Genetic Programming using Local Modules. Technical report, Universität Dortmund. Publication Title: Reihe Computational Intelligence ; 56.

Banzhaf, W. (2003). Artificial Regulatory Networks and Genetic Programming. In Riolo, R. and Worzel, B., editors, *Genetic Programming Theory and Practice*, pages 43–61. Springer US, Boston, MA.

Banzhaf, W. and Yamamoto, L. (2015). *Artificial chemistries*. The MIT Press, Cambridge, MA.

Barrett, R. and Schluter, D. (2008). Adaptation from standing genetic variation. *Trends in Ecology & Evolution*, 23(1):38–44.

Barrick, J. E., Deatherage, D. E., and Lenski, R. E. (2020). A Test of the Repeatability of Measurements of Relative Fitness in the Long-Term Evolution Experiment with Escherichia coli. In Banzhaf, W., Cheng, B. H., Deb, K., Holekamp, K. E., Lenski, R. E., Ofria, C., Pennock, R. T., Punch, W. F., and Whittaker, D. J., editors, *Evolution in Action: Past, Present and Future*, pages 77–89. Springer International Publishing, Cham. Series Title: Genetic and Evolutionary Computation.

Barton, N. H. (2000). Genetic hitchhiking. *Philosophical Transactions of the Royal Society of London. Series B: Biological Sciences*, 355(1403):1553–1562. Publisher: The Royal Society.

Beaumont, H. J. E., Gallie, J., Kost, C., Ferguson, G. C., and Rainey, P. B. (2009). Experimental evolution of bet hedging. *Nature*, 462(7269):90–93.

Beckmann, B. E., Knoester, D. B., Connelly, B. D., Waters, C. M., and McKinley, P. K. (2012). Evolution of Resistance to Quorum Quenching in Digital Organisms. *Artificial Life*, 18(3):291–310.

Beckmann, B. E. and McKinley, P. K. (2009). Evolving quorum sensing in digital organisms. In *Proceedings of the 11th Annual conference on Genetic and evolutionary computation - GECCO '09*, page 97, Montreal, Québec, Canada. ACM Press.

Bender, J. (2002). Plant epigenetics. *Current Biology*, 12(12):R412–R414.

Binard, F. and Felty, A. (2007). An abstraction-based genetic programming system. In Bosman, P. A. N., editor, *Late breaking paper at Genetic and Evolutionary Computation Conference (GECCO'2007)*, pages 2415–2422, London, United Kingdom. ACM Press.

Black, A. R. and Dodson, S. I. (1990). Demographic costs of Chaoborus-induced phenotypic plasticity in Daphnia pulex. *Oecologia*, 83(1):117–122.

Blount, Z. D., Borland, C. Z., and Lenski, R. E. (2008). Historical contingency and the evolution of a key innovation in an experimental population of Escherichia coli. *Proceedings of the National Academy of Sciences*, 105(23):7899–7906.

Boyer, S., Hérissant, L., and Sherlock, G. (2021). Adaptation is influenced by the complexity of environmental change during evolution in a dynamic environment. *PLOS Genetics*, 17(1):e1009314.

Bradshaw, A. (1965). Evolutionary Significance of Phenotypic Plasticity in Plants. In *Advances in Genetics*, volume 13, pages 115–155. Elsevier.

Brameier, M. F. and Banzhaf, W. (2007). *Linear Genetic Programming*. Genetic and Evolutionary Computation. Springer US, Boston, MA.

Buskirk, S. W., Peace, R. E., and Lang, G. I. (2017). Hitchhiking and epistasis give rise to cohort dynamics in adapting populations. *Proceedings of the National Academy of Sciences*, 114(31):8330–8335.

Byers, C., Cheng, B., and McKinley, P. (2012). Exploring the evolution of internal control structure using digital enzymes. In *Proceedings of the fourteenth international conference on Genetic and evolutionary computation conference companion - GECCO Companion '12*, page 1407, Philadelphia, Pennsylvania, USA. ACM Press.

Byers, C. M., Cheng, B. H., and McKinley, P. K. (2011). Digital enzymes: agents of reaction inside robotic controllers for the foraging problem. In *Proceedings of the 13th annual conference on Genetic and evolutionary computation - GECCO '11*, page 243, Dublin, Ireland. ACM Press.

Bäck, T., Fogel, D. B., and Michalewics, Z., editors (1997). *Handbook of evolutionary computation*. Institute of Physics Pub. ; Oxford University Press, Bristol ; Philadelphia : New York.

Canino-Koning, R., Wiser, M. J., and Ofria, C. (2016). The Evolution of Evolvability: Changing Environments Promote Rapid Adaptation in Digital Organisms. In *Proceedings of the Artificial Life Conference 2016*, pages 268–275, Cancun, Mexico. MIT Press.

Canino-Koning, R., Wiser, M. J., and Ofria, C. (2019). Fluctuating environments select for short-term phenotypic variation leading to long-term exploration. *PLOS Computational Biology*, 15(4):e1006445.

Canty, A. and Ripley, B. D. (2019). *boot: Bootstrap R (S-Plus) Functions*. R package version 1.3-23.

Card, K. J., LaBar, T., Gomez, J. B., and Lenski, R. E. (2019). Historical contingency in the evolution of antibiotic resistance after decades of relaxed selection. *PLOS Biology*, 17(10):e3000397.

Cassandras, C. G. (2014). The event-driven paradigm for control, communication and optimization. *Journal of Control and Decision*, 1(1):3–17.

Chadwick, W. and Little, T. J. (2005). A parasite-mediated life-history shift in *Daphnia magna*. *Proceedings of the Royal Society B: Biological Sciences*, 272(1562):505–509.

Chen, K. and Arnold, F. H. (1993). Tuning the activity of an enzyme for unusual environments: sequential random mutagenesis of subtilisin E for catalysis in dimethylformamide. *Proceedings of the National Academy of Sciences*, 90(12):5618–5622.

Chevin, L.-M. and Lande, R. (2010). When do adaptive plasticity and genetic evolution prevent extinction of a density-regulated population? *Evolution*, 64(4):1143–1150.

Chevin, L.-M., Lande, R., and Mace, G. M. (2010). Adaptation, Plasticity, and Extinction in a Changing Environment: Towards a Predictive Theory. *PLoS Biology*, 8(4):e1000357.

Clune, J., Ofria, C., and Pennock, R. T. (2007). Investigating the Emergence of Phenotypic Plasticity in Evolving Digital Organisms. In Almeida e Costa, F., Rocha, L. M., Costa, E., Harvey, I., and Coutinho, A., editors, *Advances in Artificial Life*, volume 4648, pages 74–83. Springer Berlin Heidelberg, Berlin, Heidelberg.

Cooper, T. F. and Ofria, C. (2002). Evolution of Stable Ecosystems in Populations of Digital Organisms. In *Proceedings of the Eighth International Conference on Artificial Life*, ICAL 2003, pages 227–232, Cambridge, MA, USA. MIT Press.

Covert, A. W., Lenski, R. E., Wilke, C. O., and Ofria, C. (2013). Experiments on the role of deleterious mutations as stepping stones in adaptive evolution. *Proceedings of the National Academy of Sciences*, 110(34):E3171–E3178.

Cramer, N. L. (1985). A representation for the adaptive generation of simple sequential programs. In Grefenstette, J. J., editor, *Proceedings of an International Conference on Genetic Algorithms and the Applications*, pages 183–187, Carnegie-Mellon University, Pittsburgh, PA, USA.

Crispo, E. (2007). The Baldwin effect and genetic assimilation: revisiting two mechanisms of evolutionary change mediated by phenotypic plasticity. *Evolution*, 61(11):2469–2479.

Crombach, A. and Hogeweg, P. (2008). Evolution of Evolvability in Gene Regulatory Networks. *PLoS Computational Biology*, 4(7):e1000112.

Crosbie, M. and Spafford, E. H. (1996). Evolving Event-driven Programs. In *Proceedings of the 1st Annual Conference on Genetic Programming*, pages 273–278, Cambridge, MA, USA. MIT Press. event-place: Stanford, California.

Csardi, G. and Nepusz, T. (2006). The igraph software package for complex network research. *InterJournal*, Complex Systems:1695.

Cussat-Blanc, S., Harrington, K., and Banzhaf, W. (2019). Artificial Gene Regulatory Networks—A Review. *Artificial Life*, 24(4):296–328.

Dewdney, A. K. (1984). Computer recreations: In the game called core war hostile programs engage in a battle of bits. *Scientific American*, 250(5):14–23.

Dolson, E., Lalejini, A., Jorgensen, S., and Ofria, C. (2020). Interpreting the Tape of Life: Ancestry-Based Analyses Provide Insights and Intuition about Evolutionary Dynamics. *Artificial Life*, 26(1):58–79.

Dolson, E. and Ofria, C. (2017). Spatial resource heterogeneity creates local hotspots of evolutionary potential. In *Proceedings of the 14th European Conference on Artificial Life ECAL 2017*, pages 122–129, Lyon, France. MIT Press.

Dolson, E. L., Pérez, S. G., Olson, R. S., and Ofria, C. (2017). Spatial resource heterogeneity increases diversity and evolutionary potential. preprint, Ecology.

Downing, K. L. (2015). *Intelligence emerging: adaptivity and search in evolving neural systems*. The MIT Press, Cambridge, Massachusetts.

Draghi, J. and Wagner, G. P. (2009). The evolutionary dynamics of evolvability in a gene network model. *Journal of Evolutionary Biology*, 22(3):599–611.

Dunn, A. M., Hogg, J. C., Kelly, A., and Hatcher, M. J. (2005). Two cues for sex determination in *Gammarus duebeni* : Adaptive variation in environmental sex determination? *Limnology and Oceanography*, 50(1):346–353.

Elena, S. F., Wilke, C. O., Ofria, C., and Lenski, R. E. (2007). Effects Of Population Size And Mutation Rate On The Evolution Of Mutational Robustness. *Evolution*, 61(3):666–674.

Ellefsen, K. O., Mouret, J.-B., and Clune, J. (2015). Neural Modularity Helps Organisms Evolve to Learn New Skills without Forgetting Old Skills. *PLOS Computational Biology*, 11(4):e1004128.

Etzion, O. and Niblett, P. (2010). *Event Processing in Action*. Manning Publications Co., Greenwich, CT, USA, 1st edition.

Forsman, A. (2015). Rethinking phenotypic plasticity and its consequences for individuals, populations and species. *Heredity*, 115(4):276–284.

Forsyth, R. (1981). BEAGLE - A Darwinian Approach to Pattern Recognition. *Kybernetes*, 10(3):159–166.

Fortuna, M. A., Barbour, M. A., Zaman, L., Hall, A. R., Buckling, A., and Bascompte, J. (2019). Coevolutionary dynamics shape the structure of bacteria-phage infection networks. *Evolution*, 73(5):1001–1011.

Garnier, S. (2018). *viridis: Default Color Maps from matplotlib*. R package version 0.5.1.

Ghalambor, C. K., Angeloni, L. M., and Carroll, S. P. (2010). Behavior as phenotypic plasticity. In Westneat, D. and Fox, C. W., editors, *Evolutionary behavioral ecology*, pages 90–107. Oxford University Press, New York, NY.

Ghalambor, C. K., Hoke, K. L., Ruell, E. W., Fischer, E. K., Reznick, D. N., and Hughes, K. A. (2015). Non-adaptive plasticity potentiates rapid adaptive evolution of gene expression in nature. *Nature*, 525(7569):372–375.

Ghalambor, C. K., McKAY, J. K., Carroll, S. P., and Reznick, D. N. (2007). Adaptive versus non-adaptive phenotypic plasticity and the potential for contemporary adaptation in new environments. *Functional Ecology*, 21(3):394–407.

Gibert, P., Debat, V., and Ghalambor, C. K. (2019). Phenotypic plasticity, global change, and the speed of adaptive evolution. *Current Opinion in Insect Science*, 35:34–40.

Gibson, G. and Dworkin, I. (2004). Uncovering cryptic genetic variation. *Nature Reviews Genetics*, 5(9):681–690.

Goings, S., Goldsby, H., Cheng, B. H., and Ofria, C. (2012). An ecology-based evolutionary algorithm to evolve solutions to complex problems. In *Artificial Life 13*, pages 171–177. MIT Press.

Goldberg, D. E. and Richardson, J. (1987). Genetic Algorithms with Sharing for Multimodal Function Optimization. In *Proceedings of the Second International Conference on Genetic Algorithms on Genetic Algorithms and Their Application*, pages 41–49, USA. L. Erlbaum Associates Inc. event-place: Cambridge, Massachusetts, USA.

Goldsby, H. J., Dornhaus, A., Kerr, B., and Ofria, C. (2012a). Task-switching costs promote the evolution of division of labor and shifts in individuality. *Proceedings of the National Academy of Sciences*, 109(34):13686–13691.

Goldsby, H. J., Knoester, D. B., and Ofria, C. (2010). Evolution of division of labor in genetically homogenous groups. In *Proceedings of the 12th annual conference on Genetic and evolutionary computation - GECCO '10*, page 135, Portland, Oregon, USA. ACM Press.

Goldsby, H. J., Knoester, D. B., Ofria, C., and Kerr, B. (2014). The Evolutionary Origin of Somatic Cells under the Dirty Work Hypothesis. *PLoS Biology*, 12(5):e1001858.

Goldsby, H. J., Serra, N., Dyer, F., Kerr, B., and Ofria, C. (2012b). The Evolution of Temporal Polyethism. In *Artificial Life 13*, pages 178–185. MIT Press.

Gomulkiewicz, R. and Holt, R. D. (1995). When does Evolution by Natural Selection Prevent Extinction? *Evolution*, 49(1):201.

Good, B. H., McDonald, M. J., Barrick, J. E., Lenski, R. E., and Desai, M. M. (2017). The dynamics of molecular evolution over 60,000 generations. *Nature*, 551(7678):45–50.

Grabowski, L. M., Bryson, D. M., Dyer, F. C., Ofria, C., and Pennock, R. T. (2010). Early evolution of memory usage in digital organisms. In *Artificial Life XII: Proceedings of the Twelfth International Conference on the Synthesis and Simulation of Living Systems*, pages 224–231. MIT Press.

Grabowski, L. M., Bryson, D. M., Dyer, F. C., Pennock, R. T., and Ofria, C. (2013). A Case Study of the De Novo Evolution of a Complex Odometric Behavior in Digital Organisms. *PLoS ONE*, 8(4):e60466.

Grant, N. A., Maddamsetti, R., and Lenski, R. E. (2020). Maintenance of Metabolic Plasticity Despite Relaxed Selection in a Long-Term Evolution Experiment with *Escherichia coli*. preprint, Evolutionary Biology.

Gupta, A. P. and Lewontin, R. C. (1982). A Study of Reaction Norms in Natural Populations of Drosophila pseudoobscura. *Evolution*, 36(5):934.

Hallsson, L. R. and Björklund, M. (2012). Selection in a fluctuating environment leads to decreased genetic variation and facilitates the evolution of phenotypic plasticity: Evolutionary response in a fluctuating environment. *Journal of Evolutionary Biology*, 25(7):1275–1290.

Harrell Jr, F. E., with contributions from Charles Dupont, and many others. (2020). *Hmisc: Harrell Miscellaneous*. R package version 4.4-2.

Harrower, M. and Brewer, C. A. (2003). ColorBrewer.org: An Online Tool for Selecting Colour Schemes for Maps. *The Cartographic Journal*, 40(1):27–37.

Heemels, W., Johansson, K., and Tabuada, P. (2012). An introduction to event-triggered and self-triggered control. In *2012 IEEE 51st IEEE Conference on Decision and Control (CDC)*, pages 3270–3285, Maui, HI, USA. IEEE.

Helmuth, T. and Spector, L. (2015). General Program Synthesis Benchmark Suite. In *Proceedings of the 2015 on Genetic and Evolutionary Computation Conference - GECCO '15*, pages 1039–1046, Madrid, Spain. ACM Press.

Helmuth, T., Spector, L., and Matheson, J. (2015). Solving Uncompromising Problems With Lexicase Selection. *IEEE Transactions on Evolutionary Computation*, 19(5):630–643.

Hendry, A. P. (2016). Key Questions on the Role of Phenotypic Plasticity in Eco-Evolutionary Dynamics. *Journal of Heredity*, 107(1):25–41.

Hernandez, J. G., Lalejini, A., Dolson, E., and Ofria, C. (2019). 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. ACM Press.

Hintze, A., Edlund, J. A., Olson, R. S., Knoester, D. B., Schossau, J., Albantakis, L., Tehrani-Saleh, A., Kvam, P., Sheneman, L., Goldsby, H., Bohm, C., and Adami, C. (2017). Markov Brains: A Technical Introduction. *arXiv:1709.05601 [cs, q-bio]*. arXiv: 1709.05601.

Hintze, A., Schossau, J., and Bohm, C. (2019). The Evolutionary Buffet Method. In Banzhaf, W., Spector, L., and Sheneman, L., editors, *Genetic Programming Theory and Practice XVI*, pages 17–36. Springer International Publishing, Cham. Series Title: Genetic and Evolutionary Computation.

Holland, J. H. (1987). Genetic algorithms and classifier systems: foundations and future directions. Technical report, Michigan Univ., Ann Arbor (USA).

Holland, J. H. (1990). Concerning the emergence of tag-mediated lookahead in classifier systems. *Physica D: Nonlinear Phenomena*, 42(1-3):188–201.

Holland, J. H. (1993). The effect of labels (tags) on social interactions. Technical report, Santa Fe Institute Working Paper 93-10-064. Santa Fe, NM.

Holland, J. H. (2006). Studying Complex Adaptive Systems. *Journal of Systems Science and Complexity*, 19(1):1–8.

Huey, R., Hertz, P., and Sinervo, B. (2003). Behavioral Drive versus Behavioral Inertia in Evolution: A Null Model Approach. *The American Naturalist*, 161(3):357–366.

Huizinga, J., Mouret, J.-B., and Clune, J. (2016). 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. Association for Computing Machinery. event-place: Denver, Colorado, USA.

Jablonka, E. and Raz, G. (2009). Transgenerational Epigenetic Inheritance: Prevalence, Mechanisms, and Implications for the Study of Heredity and Evolution. *The Quarterly Review of Biology*, 84(2):131–176.

Johnson, A., Strauss, E., Pickett, R., Adami, C., Dworkin, I., and Goldsby, H. (2014). More Bang For Your Buck: Quorum-Sensing Capabilities Improve the Efficacy of Suicidal Altruism. In *Artificial Life 14: Proceedings of the Fourteenth International Conference on the Synthesis and Simulation of Living Systems*, pages 120–128. The MIT Press.

Karlebach, G. and Shamir, R. (2008). Modelling and analysis of gene regulatory networks. *Nature Reviews Molecular Cell Biology*, 9(10):770–780.

Kassambara, A. (2021). *rstatix: Pipe-Friendly Framework for Basic Statistical Tests*. R package version 0.7.0.

Kawecki, T. J., Lenski, R. E., Ebert, D., Hollis, B., Olivieri, I., and Whitlock, M. C. (2012). Experimental evolution. *Trends in Ecology & Evolution*, 27(10):547–560.

Keijzer, M., Ryan, C., and Cattolico, M. (2004). Run Transferable Libraries — Learning Functional Bias in Problem Domains. In Kanade, T., Kittler, J., Kleinberg, J. M., Mattern, F., Mitchell, J. C., Naor, M., Nierstrasz, O., Pandu Rangan, C., Steffen, B., Sudan, M., Terzopoulos, D., Tygar, D., Vardi, M. Y., Weikum, G., and Deb, K., editors, *Genetic and Evolutionary Computation – GECCO 2004*, volume 3103, pages 531–542. Springer Berlin Heidelberg, Berlin, Heidelberg. Series Title: Lecture Notes in Computer Science.

Keijzer, M., Ryan, C., Murphy, G., and Cattolico, M. (2005). Undirected Training of Run Transferable Libraries. In Hutchison, D., Kanade, T., Kittler, J., Kleinberg, J. M., Mattern, F., Mitchell, J. C., Naor, M., Nierstrasz, O., Pandu Rangan, C., Steffen, B., Sudan, M., Terzopoulos, D., Tygar, D., Vardi, M. Y., Weikum, G., Keijzer, M., Tettamanzi, A., Collet, P., van Hemert, J., and Tomassini, M., editors, *Genetic Programming*, volume 3447, pages 361–370. Springer Berlin Heidelberg, Berlin, Heidelberg.

Kelly, S. and Heywood, M. I. (2017). Multi-task learning in Atari video games with emergent tangled program graphs. In *Proceedings of the Genetic and Evolutionary Computation Conference*, pages 195–202, Berlin Germany. ACM.

Kitano, H. (2004). Biological robustness. *Nature Reviews Genetics*, 5(11):826–837.

Knoester, D. B., Goldsby, H. J., and McKinley, P. K. (2013). Genetic Variation and the Evolution of Consensus in Digital Organisms. *IEEE Transactions on Evolutionary Computation*, 17(3):403–417.

Knoester, D. B. and McKinley, P. K. (2011). Evolution of Synchronization and Desynchronization in Digital Organisms. *Artificial Life*, 17(1):1–20.

Knoester, D. B., McKinley, P. K., and Ofria, C. A. (2007). Using group selection to evolve leadership in populations of self-replicating digital organisms. In *Proceedings of the 9th annual conference on Genetic and evolutionary computation - GECCO '07*, page 293, London, England. ACM Press.

Koza, J. R. (1989). Hierarchical genetic algorithms operating on populations of computer programs. In Sridharan, N. S., editor, *Proceedings of the Eleventh International Joint Conference on Artificial Intelligence IJCAI-89*, volume 1, pages 768–774, Detroit, MI, USA. Morgan Kaufmann.

Koza, J. R. (1992). *Genetic programming: on the programming of computers by means of natural selection*. Complex adaptive systems. MIT Press, Cambridge, Mass.

Koza, J. R. (1994). *Genetic programming II: automatic discovery of reusable programs*. Complex adaptive systems. MIT Press, Cambridge, Mass.

Krawiec, K. and Wieloch, B. (2009). Functional modularity for genetic programming. In *Proceedings of the 11th Annual conference on Genetic and evolutionary computation - GECCO '09*, page 995, Montreal, Québec, Canada. ACM Press.

Kruskal, W. H. and Wallis, W. A. (1952). Use of Ranks in One-Criterion Variance Analysis. *Journal of the American Statistical Association*, 47(260):583–621.

La Cava, W., Helmuth, T., Spector, L., and Danai, K. (2015). Genetic Programming with Epigenetic Local Search. In *Proceedings of the 2015 on Genetic and Evolutionary Computation Conference - GECCO '15*, pages 1055–1062, Madrid, Spain. ACM Press.

La Cava, W. and Spector, L. (2015). Inheritable Epigenetics in Genetic Programming. In Riolo, R., Worzel, W. P., and Kotanchek, M., editors, *Genetic Programming Theory and Practice XII*, pages 37–51. Springer International Publishing, Cham. Series Title: Genetic and Evolutionary Computation.

Lahti, D. C., Johnson, N. A., Ajie, B. C., Otto, S. P., Hendry, A. P., Blumstein, D. T., Coss, R. G., Donohue, K., and Foster, S. A. (2009). Relaxed selection in the wild. *Trends in Ecology & Evolution*, 24(9):487–496.

Lalejini, A. (2018). Supplemental material for "Evolving Event-driven programs with SignalGP" hosted on GitHub. Zenodo. doi:10.5281/zenodo.1283352. https://github.com/amlalejini/GECCO-2018-Evolving-Event-driven-Programs-with-SignalGP.

Lalejini, A. (2019). Supplemental material for "Tag-accessed Memory For Genetic Programming". Zenodo. doi: 10.5281/zenodo.2641176. url: `https://github.com/amlalejini/GECCO-2019-tag-accessed-memory`.

Lalejini, A. and Ferguson, A. (2021a). Supplemental material for "the evolutionary consequences of phenotypic plasticity". Zenodo. doi: 10.5281/zenodo.4642704. url: https://github.com/amlalejini/evolutionary-consequences-of-plasticity.

Lalejini, A., Moreno, M. A., and Ofria, C. (2020a). Case study of adaptive gene regulation in dishtiny. Open Science Framework. doi: 10.17605/OSF.IO/KQVMN.

Lalejini, A., Moreno, M. A., and Ofria, C. (2020b). Tag-based genetic regulation for genetic programming. Preprint. arXiv:2012.09229.

Lalejini, A., Moreno, M. A., and Ofria, C. (2021). Supplemental material for "Improving context-dependent problem solving with a tag-based approach to regulating genetic programming modules" hosted on GitHub. Zenodo. doi: 10.5281/zenodo.4316015. url: https://lalejini.com/Tag-based-Genetic-Regulation-for-LinearGP/.

Lalejini, A. and Ofria, C. (2016). The Evolutionary Origins of Phenotypic Plasticity. In *Proceedings of the Artificial Life Conference 2016*, pages 372–379, Cancun, Mexico. MIT Press.

Lalejini, A. and Ofria, C. (2018). Evolving event-driven programs with SignalGP. In *Proceedings of the Genetic and Evolutionary Computation Conference on - GECCO '18*, pages 1135–1142, Kyoto, Japan. ACM Press.

Lalejini, A. and Ofria, C. (2019a). 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. ACM Press.

Lalejini, A. and Ofria, C. (2019b). What Else Is in an Evolved Name? Exploring Evolvable Specificity with SignalGP. In Banzhaf, W., Spector, L., and Sheneman, L., editors, *Genetic Programming Theory and Practice XVI*, pages 103–121. Springer International Publishing, Cham.

Lalejini, A., Wiser, M. J., and Ofria, C. (2017). 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. MIT Press.

Lalejini, A. M. and Ferguson, A. J. (2021b). Data for "evolutionary consequences of phenotypic plasticity". OSF. doi: 10.17605/OSF.IO/SAV2C. url: https://osf.io/sav2c.

Lande, R. and Arnold, S. J. (1983). The Measurement of Selection on Correlated Characters. *Evolution*, 37(6):1210.

Le Goues, C., Dewey-Vogt, M., Forrest, S., and Weimer, W. (2012a). A systematic study of automated program repair: Fixing 55 out of 105 bugs for $8 each. In *2012 34th International Conference on Software Engineering (ICSE)*, pages 3–13, Zurich. IEEE.

Le Goues, C., Nguyen, T., Forrest, S., and Weimer, W. (2012b). GenProg: A Generic Method for Automatic Software Repair. *IEEE Transactions on Software Engineering*, 38(1):54–72.

Lenski, R. E., Ofria, C., Collier, T. C., and Adami, C. (1999). Genome complexity, robustness and genetic interactions in digital organisms. *Nature*, 400(6745):661–664.

Lenski, R. E., Ofria, C., Pennock, R. T., and Adami, C. (2003). The evolutionary origin of complex features. *Nature*, 423(6936):139–144.

Leroi, A. M., Bennett, A. F., and Lenski, R. E. (1994). Temperature acclimation and competitive fitness: an experimental test of the beneficial acclimation assumption. *Proceedings of the National Academy of Sciences*, 91(5):1917–1921.

Levis, N. A. and Pfennig, D. W. (2016). Evaluating 'Plasticity-First' Evolution in Nature: Key Criteria and Empirical Approaches. *Trends in Ecology & Evolution*, 31(7):563–574.

Li, Y. and Wilke, C. O. (2004). Digital Evolution in Time-Dependent Fitness Landscapes. *Artificial Life*, 10(2):123–134.

Londe, S., Monnin, T., Cornette, R., Debat, V., Fisher, B. L., and Molet, M. (2015). Phenotypic plasticity and modularity allow for the production of novel mosaic phenotypes in ants. *EvoDevo*, 6(1):36.

Lones, M. A., Fuente, L. A., Turner, A. P., Caves, L. S. D., Stepney, S., Smith, S. L., and Tyrrell, A. M. (2014). Artificial Biochemical Networks: Evolving Dynamical Systems to Control Dynamical Systems. *IEEE Transactions on Evolutionary Computation*, 18(2):145–166.

Lones, M. A., Turner, A. P., Fuente, L. A., Stepney, S., Caves, L. S. D., and Tyrrell, A. M. (2013). Biochemical connectionism. *Natural Computing*, 12(4):453–472.

Lones, M. A. and Tyrrell, A. M. (2004). Modelling biological evolvability: implicit context and variation filtering in enzyme genetic programming. *Biosystems*, 76(1-3):229–238.

Lopes, R. L. and Costa, E. (2012). The Regulatory Network Computational Device. *Genetic Programming and Evolvable Machines*, 13(3):339–375. Place: USA Publisher: Kluwer Academic Publishers.

Maynard Smith, J. (1992). Byte-sized evolution. *Nature*, 355(6363):772–773.

McDermott, J. and O'Reilly, U.-M. (2015). Genetic Programming. In Kacprzyk, J. and Pedrycz, W., editors, *Springer Handbook of Computational Intelligence*, pages 845–869. Springer Berlin Heidelberg, Berlin, Heidelberg.

Miller, J. F. (1999). An Empirical Study of the Efficiency of Learning Boolean Functions Using a Cartesian Genetic Programming Approach. In *Proceedings of the 1st Annual Conference on Genetic and Evolutionary Computation - Volume 2*, GECCO'99, pages 1135–1142, San Francisco, CA, USA. Morgan Kaufmann Publishers Inc. event-place: Orlando, Florida.

Miller, M. B. and Bassler, B. L. (2001). Quorum Sensing in Bacteria. *Annual Review of Microbiology*, 55(1):165–199.

Misevic, D., Ofria, C., and Lenski, R. E. (2006). Sexual reproduction reshapes the genetic architecture of digital organisms. *Proceedings of the Royal Society B: Biological Sciences*, 273(1585):457–464.

Misevic, D., Ofria, C., and Lenski, R. E. (2010). Experiments with Digital Organisms on the Origin and Maintenance of Sex in Changing Environments. *Journal of Heredity*, 101(Supplement 1):S46–S54.

Mohn, F. and Schübeler, D. (2009). Genetics and epigenetics: stability and plasticity during cellular differentiation. *Trends in Genetics*, 25(3):129–136.

Moreno, M. A. (2021). Evaluating function dispatch methods in signalgp. OSF. doi: 10.17605/OSF.IO/RMKCV. url: osf.io/rmkcv.

Moreno, M. A. and Ofria, C. (2019). Toward Open-Ended Fraternal Transitions in Individuality. *Artificial Life*, 25(2):117–133.

Moreno, M. A. and Ofria, C. (2020). Practical steps toward indefinite scalability: In pursuit of robust computational substrates for open-ended evolution. OSF. doi: 10.17605/OSF.IO/53VGH. url: osf.io/53vgh.

Moreno, M. A. and Rodriguez Papa, S. (2021). signalgp-lite. OSF. doi: 10.17605/OSF.IO/J8PGE. url: osf.io/j8pge.

Moxon, R., Bayliss, C., and Hood, D. (2006). Bacterial Contingency Loci: The Role of Simple Sequence DNA Repeats in Bacterial Adaptation. *Annual Review of Genetics*, 40(1):307–333.

Murren, C. J., Auld, J. R., Callahan, H., Ghalambor, C. K., Handelsman, C. A., Heskel, M. A., Kingsolver, J. G., Maclean, H. J., Masel, J., Maughan, H., Pfennig, D. W., Relyea, R. A., Seiter, S., Snell-Rood, E., Steiner, U. K., and Schlichting, C. D. (2015). Constraints on the evolution of phenotypic plasticity: limits and costs of phenotype and plasticity. *Heredity*, 115(4):293–301.

Nahum, J. R., West, J., Althouse, B. M., Zaman, L., Ofria, C., and Kerr, B. (2017). Improved adaptation in exogenously and endogenously changing environments. In *Proceedings of the 14th European Conference on Artificial Life ECAL 2017*, pages 306–313, Lyon, France. MIT Press.

Nakazawa, M. (2019). *fmsb: Functions for Medical Statistics Book with some Demographic Data*. R package version 0.7.0.

Neuwirth, E. (2014). *RColorBrewer: ColorBrewer Palettes*. R package version 1.1-2.

Nolfi, S., Miglino, O., and Parisi, D. (1994). Phenotypic plasticity in evolving neural networks. In *Proceedings of PerAc '94. From Perception to Action*, pages 146–157, Lausanne, Switzerland. IEEE Comput. Soc. Press.

Ofria, C., Bryson, D. M., and Wilke, C. O. (2009). Avida: A Software Platform for Research in Computational Evolutionary Biology. In Komosinski, M. and Adamatzky, A., editors, *Artificial Life Models in Software*, pages 3–35. Springer London, London.

Ofria, C., Huang, W., and Torng, E. (2008). On the Gradual Evolution of Complexity and the Sudden Emergence of Complex Features. *Artificial Life*, 14(3):255–263.

Ofria, C., Moreno, M. A., Dolson, E., Lalejini, A., Rodriguez-Papa, S., Fenton, J., Perry, K., Jorgensen, S., Hoffman, R., Miller, R., Edwards, O. B., Stredwick, J., G, N. C., Clemons, R., Vostinar, A., Moreno, R., Schossau, J., Zaman, L., and Rainbow, D. (2020). Empirical: A scientific software library for research, education, and public engagement. doi: 10.5281/zenodo.4141943. url: https://github.com/devosoft/Empirical.

Ofria, C. and Wilke, C. O. (2004). Avida: A Software Platform for Research in Computational Evolutionary Biology. *Artificial Life*, 10(2):191–229.

Ogle, D. H. (2017). *FSA: Fisheries Stock Analysis*. R package version 0.8.17.

O'Neill, M. and Ryan, C. (2000). Grammar based function definition in Grammatical Evolution. In Whitley, D., Goldberg, D., Cantu-Paz, E., Spector, L., Parmee, I., and Beyer, H.-G., editors, *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO-2000)*, pages 485–490, Las Vegas, Nevada, USA. Morgan Kaufmann.

Orzechowski, P., La Cava, W., and Moore, J. H. (2018). Where are we now?: a large benchmark study of recent symbolic regression methods. In *Proceedings of the Genetic and Evolutionary Computation Conference*, pages 1183–1190, Kyoto Japan. ACM.

O'Neill, M. and Spector, L. (2019). Automatic programming: The open issue? *Genetic Programming and Evolvable Machines*.

Paaby, A. B. and Rockman, M. V. (2014). Cryptic genetic variation: evolution's hidden substrate. *Nature Reviews Genetics*, 15(4):247–258.

Paenke, I., Sendhoff, B., and Kawecki, T. (2007). Influence of Plasticity and Learning on Evolution under Directional Selection. *The American Naturalist*, 170(2):E47–E58.

Perkis, T. (1994). Stack-based genetic programming. In *Proceedings of the First IEEE Conference on Evolutionary Computation. IEEE World Congress on Computational Intelligence*, pages 148–153, Orlando, FL, USA. IEEE.

Pigliucci, M. (2006). Phenotypic plasticity and evolution by genetic assimilation. *Journal of Experimental Biology*, 209(12):2362–2367.

Poli, R., Langdon, W. B., and McPhee, N. F. (2008). *A Field Guide to Genetic Programming*. Lulu Enterprises, UK Ltd.

Pontes, A. C., Mobley, R. B., Ofria, C., Adami, C., and Dyer, F. C. (2020). The Evolutionary Origin of Associative Learning. *The American Naturalist*, 195(1):E1–E19.

Pontes, A. C., Whalen, I., Mitchell, A. C., Mobley, R. B., Dyer, F. C., and Ofria, C. (2017). Investigations into the evolutionary origin of navigation and learning. In *Proceedings of the 14th European Conference on Artificial Life ECAL 2017*, pages 358–359, Lyon, France. MIT Press.

Price, T. D., Qvarnström, A., and Irwin, D. E. (2003). The role of phenotypic plasticity in driving genetic evolution. *Proceedings of the Royal Society of London. Series B: Biological Sciences*, 270(1523):1433–1440.

Quigley, M., Conley, K., Gerkey, B., Faust, J., Foote, T., Leibs, J., Wheeler, R., and Ng, A. Y. (2009). ROS: an open-source Robot Operating System. In *ICRA workshop on open source software*, volume 3, page 5. Kobe.

R Core Team (2016). *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing, Vienna, Austria.

R Core Team (2020). *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing, Vienna, Austria.

R Core Team (2021). *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing, Vienna, Austria.

Rainey, P. B., Beaumont, H. J., Ferguson, G. C., Gallie, J., Kost, C., Libby, E., and Zhang, X.-X. (2011). The evolutionary emergence of stochastic phenotype switching in bacteria. *Microbial Cell Factories*, 10(Suppl 1):S14.

Rasmussen, S., Feldberg, R., Hindsholm, M., and Knudsen, C. (1989). Core evolution: Emergence of cooperative structures in a computational chemistry.

Rasmussen, S., Knudsen, C., Feldberg, R., and Hindsholm, M. (1990). The coreworld: Emergence and evolution of cooperative structures in a computational chemistry. *Physica D: Nonlinear Phenomena*, 42(1):111 – 134.

Ray, T. S. (1991). An Approach to the Synthesis of Life. In Langton, C. G., Taylor, C., Farmer, J. D., and Rasmussen, S., editors, *Artificial Life II*, volume XI, pages 371–408, Redwood City, CA. Addison-Wesley.

Ricalde, E. and Banzhaf, W. (2017). 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. IEEE.

Rice, W. R. (1989). Analyzing Tables of Statistical Tests. *Evolution*, 43(1):223.

Roberts, S. C., Howard, D., and Koza, J. R. (2001). Evolving Modules in Genetic Programming by Subtree Encapsulation. In Goos, G., Hartmanis, J., van Leeuwen, J., Miller, J., Tomassini, M., Lanzi, P. L., Ryan, C., Tettamanzi, A. G. B., and Langdon, W. B., editors, *Genetic Programming*, volume 2038, pages 160–175. Springer Berlin Heidelberg, Berlin, Heidelberg.

Rosca, J. and Ballard, D. (1994). 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. IEEE.

Saini, A. K. and Spector, L. (2019). Modularity metrics for genetic programming. In *Proceedings of the Genetic and Evolutionary Computation Conference Companion on - GECCO '19*, pages 2056–2059, Prague, Czech Republic. ACM Press.

Saini, A. K. and Spector, L. (2020). Using Modularity Metrics as Design Features to Guide Evolution in Genetic Programming. In Banzhaf, W., Goodman, E., Sheneman, L., Trujillo, L., and Worzel, B., editors, *Genetic Programming Theory and Practice XVII*, pages 165–180. Springer International Publishing, Cham. Series Title: Genetic and Evolutionary Computation.

Schaum, C. E. and Collins, S. (2014). Plasticity predicts evolution in a marine alga. *Proceedings of the Royal Society B: Biological Sciences*, 281(1793):20141486.

Schlichting, C. D. (2003). Origins of differentiation via phenotypic plasticity. *Evolution and Development*, 5(1):98–105.

Schlichting, C. D. (2008). Hidden Reaction Norms, Cryptic Genetic Variation, and Evolvability. *Annals of the New York Academy of Sciences*, 1133(1):187–203.

Schlichting, C. D. and Wund, M. A. (2014). Phenotypic Plasticity and Epigenetic Marking: An Assessment of Evidence for Genetic Accommodation. *Evolution*, 68(3):656–672.

Schwander, T. and Leimar, O. (2011). Genes as leaders and followers in evolution. *Trends in Ecology & Evolution*, 26(3):143–151.

Seger, J. and Brockmann, H. (1987). What is Bet-Hedging? *Oxf. Surv. Evol. Biol.*, 4:182–211.

Skocelas, K. G. and DeVries, B. (2020). Test Data Generation for Recurrent Neural Network Implementations. In *2020 IEEE International Conference on Electro Information Technology (EIT)*, pages 469–474, Chicago, IL, USA. IEEE.

Smith, J. M. and Haigh, J. (1974). The hitch-hiking effect of a favourable gene. *Genetical Research*, 23(1):23–35.

Smith, Z. D. and Meissner, A. (2013). DNA methylation: roles in mammalian development. *Nature Reviews Genetics*, 14(3):204–220.

Sniegowski, P. D., Gerrish, P. J., Johnson, T., and Shaver, A. (2000). The evolution of mutation rates: separating causes from consequences. *BioEssays*, 22(12):1057–1066.

Sniegowski, P. D., Gerrish, P. J., and Lenski, R. E. (1997). Evolution of high mutation rates in experimental populations of E. coli. *Nature*, 387(6634):703–705.

Soltoggio, A., Stanley, K. O., and Risi, S. (2018). Born to learn: The inspiration, progress, and future of evolved plastic artificial neural networks. *Neural Networks*, 108:48–67.

Spector, L. (1996). Simultaneous Evolution of Programs and Their Control Structures. In Angeline, P. J. and Kinnear, Jr., K. E., editors, *Advances in Genetic Programming*, pages 137–154. MIT Press, Cambridge, MA, USA.

Spector, L. (2001). Autoconstructive Evolution: Push, PushGP, and Pushpop. In *Proceedings Of The Genetic And Evolutionary Computation Conference*, pages 137–146. Morgan Kaufmann Publishers.

Spector, L. (2011). Towards Practical Autoconstructive Evolution: Self-Evolution of Problem-Solving Genetic Programming Systems. In Riolo, R., McConaghy, T., and Vladislavleva, E., editors, *Genetic Programming Theory and Practice VIII*, volume 8, pages 17–33. Springer New York, New York, NY. Series Title: Genetic and Evolutionary Computation.

Spector, L., Harrington, K., and Helmuth, T. (2012). Tag-based modularity in tree-based genetic programming. In *Proceedings of the fourteenth international conference on Genetic and evolutionary computation conference - GECCO '12*, page 815, Philadelphia, Pennsylvania, USA. ACM Press.

Spector, L., Harrington, K., Martin, B., and Helmuth, T. (2011a). What's in an Evolved Name? The Evolution of Modularity via Tag-Based Reference. In Riolo, R., Vladislavleva, E., and Moore, J. H., editors, *Genetic Programming Theory and Practice IX*, pages 1–16. Springer New York, New York, NY.

Spector, L., Martin, B., Harrington, K., and Helmuth, T. (2011b). Tag-based modules in genetic programming. In *Proceedings of the 13th annual conference on Genetic and evolutionary computation - GECCO '11*, page 1419, Dublin, Ireland. ACM Press.

Sánchez, Á., Vila, J. C., Chang, C.-Y., Diaz-Colunga, J., Estrela, S., and Rebolleda-Gomez, M. (2021). Directed Evolution of Microbial Communities. *Annual Review of Biophysics*, 50(1):annurev–biophys–101220–072829.

Travisano, M., Mongold, J., Bennett, A., and Lenski, R. (1995). Experimental tests of the roles of adaptation, chance, and history in evolution. *Science*, 267(5194):87–90.

Turner, A. J. and Miller, J. F. (2015). Neutral genetic drift: an investigation using Cartesian Genetic Programming. *Genetic Programming and Evolvable Machines*, 16(4):531–558.

Turner, A. P., Caves, L. S. D., Stepney, S., Tyrrell, A. M., and Lones, M. A. (2017). 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.

Van den Bergh, B., Swings, T., Fauvart, M., and Michiels, J. (2018). Experimental Design, Population Dynamics, and Diversity in Microbial Experimental Evolution. *Microbiology and Molecular Biology Reviews*, 82(3):e00008–18, /mmbr/82/3/e00008–18.atom.

Wagenaar, D. A. and Adami, C. (2004). Influence of Chance, History, and Adaptation on Digital Evolution. *Artificial Life*, 10(2):181–190.

Wagner, A. P., Zaman, L., Dworkin, I., and Ofria, C. (2014). Behavioral Strategy Chases Promote the Evolution of Prey Intelligence. *arXiv:1310.1369 [q-bio]*. arXiv: 1310.1369.

Wagner, A. P., Zaman, L., Dworkin, I., and Ofria, C. (2020). Behavioral Strategy Chases Promote the Evolution of Prey Intelligence. In Banzhaf, W., Cheng, B. H., Deb, K., Holekamp, K. E., Lenski, R. E., Ofria, C., Pennock, R. T., Punch, W. F., and Whittaker, D. J., editors, *Evolution in Action: Past, Present and Future*, pages 225–246. Springer International Publishing, Cham. Series Title: Genetic and Evolutionary Computation.

Wagner, G. P., Pavlicev, M., and Cheverud, J. M. (2007). The road to modularity. *Nature Reviews Genetics*, 8(12):921–931.

Walker, J. and Miller, J. (2008). The Automatic Acquisition, Evolution and Reuse of Modules in Cartesian Genetic Programming. *IEEE Transactions on Evolutionary Computation*, 12(4):397–417.

Weiner, S. A. and Toth, A. L. (2012). Epigenetics in Social Insects: A New Direction for Understanding the Evolution of Castes. *Genetics Research International*, 2012:1–11.

Weise, T. and Tang, K. (2012). Evolving Distributed Algorithms With Genetic Programming. *IEEE Transactions on Evolutionary Computation*, 16(2):242–265.

Wennersten, L. and Forsman, A. (2012). Population-level consequences of polymorphism, plasticity and randomized phenotype switching: a review of predictions. *Biological Reviews*, 87(3):756–767.

West-Eberhard, M. J. (2003). *Developmental Plasticity and Evolution.* Oxford University Press.

West-Eberhard, M. J. (2005). Developmental plasticity and the origin of species differences. *Proceedings of the National Academy of Sciences*, 102(Supplement 1):6543–6549.

Wickham, H. (2020). *reshape2: Flexibly Reshape Data: A Reboot of the Reshape Package.* R package version 1.4.4.

Wickham, H., Averick, M., Bryan, J., Chang, W., McGowan, L. D., François, R., Grolemund, G., Hayes, A., Henry, L., Hester, J., Kuhn, M., Pedersen, T. L., Miller, E., Bache, S. M., Müller, K., Ooms, J., Robinson, D., Seidel, D. P., Spinu, V., Takahashi, K., Vaughan, D., Wilke, C., Woo, K., and Yutani, H. (2019). Welcome to the tidyverse. *Journal of Open Source Software*, 4(43):1686.

Wickham, H., Chang, W., Henry, L., Pedersen, T. L., Takahashi, K., Wilke, C., Woo, K., Yutani, H., and Dunnington, D. (2020). *ggplot2: Create Elegant Data Visualisations Using the Grammar of Graphics.* R package version 3.3.2.

Wickham, H. and Seidel, D. (2020). *scales: Scale Functions for Visualization.* R package version 1.1.1.

Wilcoxon, F. (1992). Individual Comparisons by Ranking Methods. In Kotz, S. and Johnson, N. L., editors, *Breakthroughs in Statistics*, pages 196–202. Springer New York, New York, NY. Series Title: Springer Series in Statistics.

Wilke, C. O. (2020). *cowplot: Streamlined Plot Theme and Plot Annotations for ggplot2*. R package version 1.1.0.

Wilke, C. O. and Adami, C. (2002). The biology of digital organisms. *Trends in Ecology & Evolution*, 17(11):528–532.

Wilke, C. O., Wang, J. L., Ofria, C., Lenski, R. E., and Adami, C. (2001). Evolution of digital organisms at high mutation rates leads to survival of the flattest. *Nature*, 412(6844):331–333.

Wilson, G. and Banzhaf, W. (2008). A Comparison of Cartesian Genetic Programming and Linear Genetic Programming. In Hutchison, D., Kanade, T., Kittler, J., Kleinberg, J. M., Mattern, F., Mitchell, J. C., Naor, M., Nierstrasz, O., Pandu Rangan, C., Steffen, B., Sudan, M., Terzopoulos, D., Tygar, D., Vardi, M. Y., Weikum, G., O'Neill, M., Vanneschi, L., Gustafson, S., Esparcia Alcázar, A. I., De Falco, I., Della Cioppa, A., and Tarantino, E., editors, *Genetic Programming*, volume 4971, pages 182–193. Springer Berlin Heidelberg, Berlin, Heidelberg. Series Title: Lecture Notes in Computer Science.

Winger, B. M., Auteri, G. G., Pegan, T. M., and Weeks, B. C. (2019). A long winter for the Red Queen: rethinking the evolution of seasonal migration. *Biological Reviews*, 94(3):737–752.

Wiser, M. J. (2015). *An analysis of fitness in long-term asexual evolution experiments*. OCLC: 945891091.

Wiser, M. J., Dolson, E. L., Vostinar, A., Lenski, R. E., and Ofria, C. (2018). The Boundedness Illusion: Asymptotic projections from early evolution underestimate evolutionary potential. preprint, PeerJ Preprints.

Wiser, M. J., Ribeck, N., and Lenski, R. E. (2013). Long-Term Dynamics of Adaptation in Asexual Populations. *Science*, 342(6164):1364–1367.

Wróbel, B. and Joachimczak, M. (2014). Using the Genetic Regulatory Evolving Artificial Networks (GReaNs) Platform for Signal Processing, Animat Control, and Artificial Multicellular Development. In Kowaliw, T., Bredeche, N., and Doursat, R., editors, *Growing Adaptive Machines*, volume 557, pages 187–200. Springer Berlin Heidelberg, Berlin, Heidelberg. Series Title: Studies in Computational Intelligence.

Wund, M. A. (2012). Assessing the Impacts of Phenotypic Plasticity on Evolution. *Integrative and Comparative Biology*, 52(1):5–15.

Xie, Y. (2020). *bookdown: Authoring Books and Technical Documents with R Markdown*. R package version 0.21.

Yeboah-Antwi, K. (2012). 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. ACM Press.

Yuan, Y. and Banzhaf, W. (2020). ARJA: Automated Repair of Java Programs via Multi-Objective Genetic Programming. *IEEE Transactions on Software Engineering*, 46(10):1040–1067.

Zheng, J., Payne, J. L., and Wagner, A. (2019). Cryptic genetic variation accelerates evolution by opening access to diverse adaptive peaks. *Science*, 365(6451):347–353.

Zimmer, C. and Emlen, D. J. (2013). *Evolution: making sense of life*. Roberts and Company Publishers, Greenwood Village, CO.