Supplemental Material for Tag-based Genetic Regulation for Genetic Programming

Alexander Lalejini, Matthew Andres Moreno, and Charles Ofria

2020-12-10

Contents

Chapter 1

Introduction

This is the supplemental material for our work, 'Tag-based Genetic Regulation for Genetic Programming'. This is not intended as a stand-alone document, but as a companion to our paper.

1.1 About our supplemental material

As you may have noticed (unless you're reading a pdf version of this), our supplemental material is hosted using GitHub pages. We compiled our data analyses and supplemental documentation into this nifty web-accessible book using bookdown.

Our supplemental material includes the following:

- Data availability (Section 2)
- Step-by-step guide to compiling and running our experiments (Section 3)
- More details on the SignalGP representation used in this work (Section 4)
- Visualizations of the exponential regulator used in this work (Section 5)
- Fully-detailed analysis scripts for each experiment (including source code)
  - Changing-signal problem analysis (Section 6)
  - Repeated-signal problem analysis (Section 7)
  - Contextual-signal problem analysis (Section 8)
  - Boolean calculator problem (prefix notation) analysis (Section 9)
  - Boolean calculator problem (postfix notation) analysis (Section 10)

1.2 Contributing authors

- Alexander Lalejini
- Matthew Andrew Moreno
- Charles Ofria

1.3 Research overview

1.3.1 Abstract

We introduce and experimentally demonstrate tag-based genetic regulation, a new genetic programming (GP) technique that allows evolving programs to regulate code modules. Tags are evolvable labels that provide a flexible mechanism for labeling and referring to code modules. Tag-based genetic regulation extends existing tag-based naming schemes to allow programs to ''promote" and ''repress" code modules. This extension allows evolution to form arbitrary gene regulatory networks in a program where genes are program modules and program instructions mediate regulation. We demonstrate the functionality of tag-based genetic regulation on several diagnostic tasks as well as a more challenging program synthesis problem. We find that tag-based regulation improves problem-solving performance on problems responses to particular inputs must change over time (e.g., based on local context). We also observe that our implementation of tag-based genetic regulation can impede adaptive evolution when expected outputs are not context-dependent (i.e., the correct response to a particular input remains static over time). Tag-based genetic regulation is immediately applicable to existing tag-enabled GP systems, and broadens our repertoire of techniques for evolving more dynamic programs.

1.3.2 Tag-based Referencing

Tags are evolvable labels that can be mutated, and the similarity (or dissimilarity) between any two tags can be quantified (Spector et al., 2011). 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 potentially valid references. Further, mutations to tags do not necessarily damage 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, this technique allows the naming and use of modularized code fragments to incrementally co-evolve.

In the tag-based referencing example above, the call instruction uses tag 1001 to reference the closest-matching module (in this case, the yellow module tagged 0001).

1.3.3 Tag-based Regulation

Tag-based regulation allows evolving programs to instantiate gene regulatory networks using tag-based referencing. This functionality allows programs to dynamically adjust which module is triggered by a particular call based on prior inputs. Specifically, we implemented tag-based genetic regulation in the context

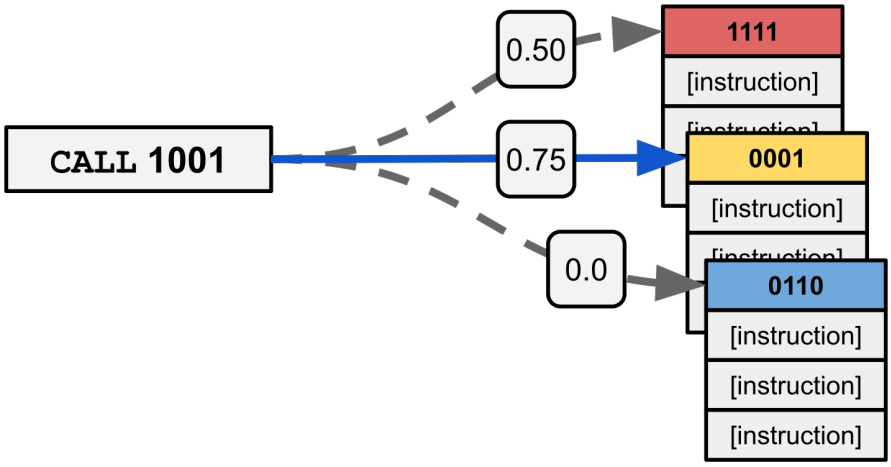


Figure 1.1: Example of tag-based referencing.

of a linear GP system (SignalGP); however, our approach is applicable to any tag-enabled GP system.

To implement tag-based genetic regulation, we supplement the instruction set with promoter and repressor instructions that, when executed, adjust how well subsequent tag-based references match with a target module. Intuitively, promoters increase a target module's tag-match score with subsequent references, thereby increasing its chances of being triggered; repressors have the opposite effect. When determining which module to reference in response to a call instruction, each module's tag-match score is a function of how well the module's tag matches the call instruction's tag as well as the module's regulatory value.

1.3.4 SignalGP

SignalGP defines a way of organizing and interpreting genetic programs to afford computational evolution access to the event-driven programming paradigm. In SignalGP, program execution is signal-driven. Programs are segmented into genetic modules (or functions), and each module can be independently triggered in response to a signal. Each module associates a tag with a linear sequence of instructions. In this work, tags are represented as fixed-length bit strings.

SignalGP makes the concept of events or signals explicit: all signals contain a tag and any associated data. Signals can originate exogenously (e.g, from the environment or other agents) or endogenously (e.g., self-signaling). We use tag-based referencing to determine the most appropriate function to automatically trigger in response to a signal. Signals trigger the function with the closest matching tag.

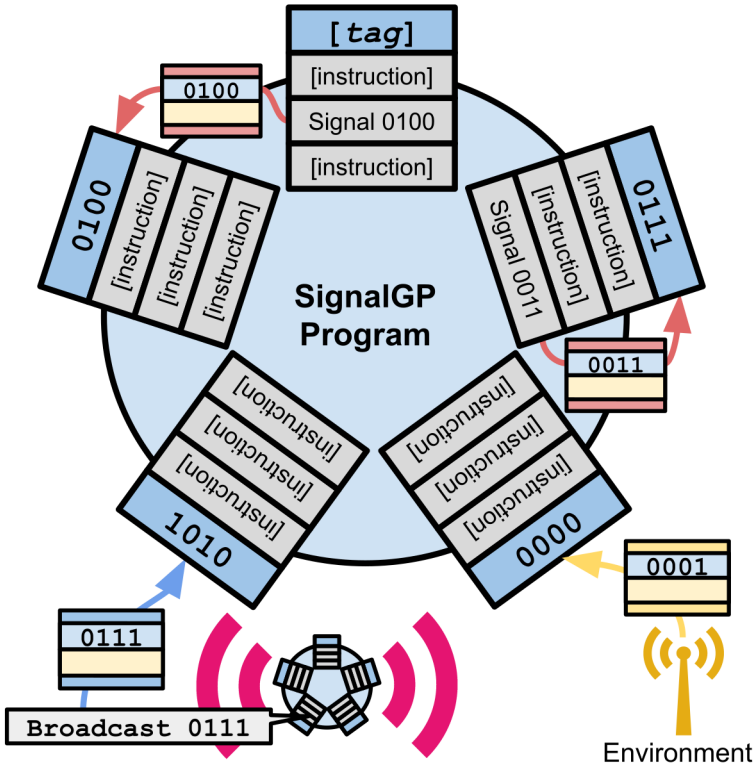


Figure 1.2: Cartoon overview of SignalGP.

For a more detailed description of the SignalGP representation, see (Lalejini and Ofria, 2018).

1.3.4.1 Genetic Regulation in SignalGP

In this work, we augment the SignalGP representation with genetic regulation, allowing programs to alter their responses to signals during their lifetime. We supplement the SignalGP instruction set with promotor and repressor instructions, which, when executed, adjust how well subsequent signals or internal call instructions match with a target function (instruction-level tags and tag-based referencing are used for function targeting).

A simple example of how genetic regulation works (in an event-handling context) is given in the figure below. First (1), an event triggers the yellow function that, when executed, (2) promotes the red function and represses itself. Finally (3), when a subsequent signal (identical to the previous) is received, it triggers the up-regulated red function instead of the yellow function.

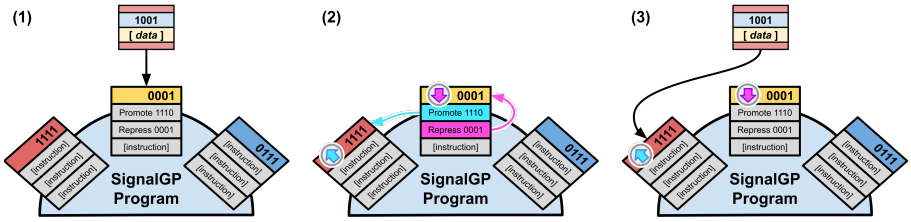


Figure 1.3: Example of tag-based genetic regulation in SignalGP.

1.3.5 Experiments

We compared the performance of regulation-enabled and regulation-disabled SignalGP on five problems:

- Repeated-signal Problem
- Contextual-signal Problem
- Changing-signal Problem
- Boolean Logic Calculator Problem (prefix notation)
- Boolean Logic Calculator Problem (postfix notation)

The repeated-signal, contextual-signal, and prefix notation calculator problems each required programs to dynamically adjust their responses to particular inputs over time. The changing-signal and postfix notation calculator problems did not require programs to adjust responses to inputs over time.

1.3.6 Results

- Proof of method: we observed the evolution of programs capable of leveraging tag-based regulation to dynamically adjust module associations over

time.

- We found that tag-based regulation improved problem-solving performance on context-dependent problems (i.e., problems in which the appropriate response to a particular input changes over time).
- We found that our implementation of tag-based regulation can impede adaptive evolution on problems that do not require programs to adjust responses to particular inputs over time.

Chapter 2

Data Availability

All of our experimental data is available online at (Lalejini et al., 2020).

All project source code and any training/testing sets used in our experiments can be found in this project's GitHub repository: https://github.com/amlalejini/Tag-based-Genetic-Regulation-for-LinearGP.

Chapter 3

Compile and run experiments locally

Here, we provide a brief guide to compiling and running our experiments.

Please file an issue if something is unclear or does not work.

3.1 Docker

On docker hub: https://hub.docker.com/r/amlalejini/tag-based-genetic-regulation-for-gp

```
docker pull amlalejini/tag-based-genetic-regulation-for-gp
```

This will create an image with all the requisite dependencies installed/downloaded and with our experiments compiled.

To run the container interactively:

```
docker run -it --entrypoint bash amlalejini/tag-based-genetic-regulation-for-gp
```

You can exit the container at any point with `ctrl-d`.

Inside the container, you should be at `/opt/Tag-based-Genetic-Regulation-for-LinearGP/`. If you `ls` you should see something like this (maybe not exactly):

```
Dockerfile                                                    documents
Gemfile                                                       experiments
LICENSE                                                       index.Rmd
Makefile                                                      media
README.md                                                     requirements.txt
_bookdown.yml                                                 scripts
_config.yml                                                   source
```

```
_output.yml                                                    style.css
alt-signal-exp_tag-len-256_match-metric-streak_thresh-0_reg-exp  supplemental
bool-calc-exp_tag-len-256_match-metric-streak_thresh-0_reg-exp   supplemental.bib
build_book.sh                                                  supplemental_files
build_exps.sh                                                  tail.Rmd
chg-env-exp_tag-len-256_match-metric-streak_thresh-0_reg-exp     tests
```

The important thing is that there should be three executables (with absurdly long names):

- `chg-env-exp_tag-len-256_match-metric-streak_thresh-0_reg-exp`
  - Use this to run the changing-signal problem.
  - To generate a default configuration file, `chg-env-exp_tag-len-256_match-metric-streak_th` `--gen`
- `alt-signal-exp_tag-len-256_match-metric-streak_thresh-0_reg-exp`
  - Use this to run the repeated-signal problem.
  - To generate a default configuration file, `alt-signal-exp_tag-len-256_match-metric-streak` `--gen`
- `bool-calc-exp_tag-len-256_match-metric-streak_thresh-0_reg-exp`
  - Use this to run any of the boolean logic calculator problems and the contextual-signal problem.
  - To generate a default configuration file, `bool-calc-exp_tag-len-256_match-metric-streak_` `--gen`

Find the exact configurations we used for our experiments here: https://github.com/amlalejini/Tag-based-Genetic-Regulation-for-LinearGP/tree/master/experiments

3.2 Manually

This guide assumes an Ubuntu-flavored Linux operating system. These instructions *should* mostly work for MacOS; otherwise, we recommend using our Docker image or a virtual machine.

Our experiments are implemented in C++, so you'll need a modern C++ compiler capable of compiling C++17 code.

E.g., I'm using:

```
g++ (Ubuntu 10.2.0-13ubuntu1) 10.2.0
```

3.2.1 Get dependencies

First, make a directory were we can put this project and all of its dependencies.

```
mkdir workspace
cd workspace
```

Next, clone this repository into your new directory.

```
git clone https://github.com/amlalejini/Tag-based-Genetic-Regulation-for-LinearGP.git
```

Our experiments depend on the Empirical and SignalGP libraries on GitHub. Inside the `workspace` directory, we'll clone SignalGP and checkout the appropriate commit.

```
git clone https://github.com/amlalejini/SignalGP.git
cd SignalGP
git checkout 83d879cfdb6540862315dc454c1525ccd8054e65
cd ..
```

Next, let's get Empirical (also stick this in the `workspace` directory).

```
git clone https://github.com/amlalejini/Empirical.git
cd Empirical
git checkout e72dae6490dee5caf8e5ec04a634b483d2ad4293
```

We're not quite done with Empirical. We need to grab all of Empirical's dependencies. This will install *all* of Empirical's dependencies, including those needed to build its documentation/tests.

```
make install-dependencies
```

OR, if you don't want *all* of that, instead you could do:

```
git submodule init
git submodule update
```

If you don't have libssl-dev, you'll also want to install that (some of the tag-matching metrics use cryptographic hash). E.g.,

```
sudo apt-get install libssl-dev
```

Now we should be good to compile the three executables that we used for our experiments. Inside `workspace/Tag-based-Genetic-Regulation-for-LinearGP/`:

```
./build_exps
```

This script just sets some environment variables (e.g., to define which experiment to compile, the tag-matching metric, etc.) and runs `make native`.

To use a different compiler (than g++), you'll need to change `CXX_nat` in the makefile.

This should create three executables (with absurdly long names):

- `chg-env-exp_tag-len-256_match-metric-streak_thresh-0_reg-exp`
  - Use this to run the changing-signal problem.
  - To generate a default configuration file, `chg-env-exp_tag-len-256_match-metric-streak_thresh-0_reg --gen`
- `alt-signal-exp_tag-len-256_match-metric-streak_thresh-0_reg-exp`
  - Use this to run the repeated-signal problem.
  - To generate a default configuration file, `alt-signal-exp_tag-len-256_match-metric-streak_thresh-0_ --gen`

- `bool-calc-exp_tag-len-256_match-metric-streak_thresh-0_reg-exp`
  - Use this to run any of the boolean logic calculator problems and the contextual-signal problem.
  - To generate a default configuration file, `bool-calc-exp_tag-len-256_match-metric-streak_ --gen`

Find the exact configurations we used for our experiments here: https://github.com/amlalejini/Tag-based-Genetic-Regulation-for-LinearGP/tree/master/experiments

Chapter 4

SignalGP representation

Here, we give more details on how we configured SignalGP for this work, including the particular instruction sets used in each experiment. For a broad overview of SignalGP, see (Lalejini and Ofria, 2018). For exact configurations used in each experiment, see respective experiment directories.

4.1 Memory model

SignalGP programs have four types of memory buffers with which to carry out computations:

- Working (register) memory
  - Call-local (* thread-local) memory.
  - Memory used by the majority of computation instructions.
- Input memory
  - Call-local (& thread-local) memory.
  - Read-only.
  - This memory is used to specify function call arguments. When a function is called on a thread (i.e., a call instruction is executed), the caller's working memory is copied into the input memory of the new call-state, which is created on top of the thread's call stack.
  - Programs must execute explicit instructions to read from the input memory buffer (into the working memory buffer).
- Output memory
  - Call-local (& thread-local) memory.
  - Write-only.
  - This memory is used to specify the return values of a function call.
  - When a function returns to the previous call-state (i.e., the one just below it on the thread's call stack), positions that were set in the output buffer are returned to the caller's working memory buffer.

  - Programs must execute explicit instructions to write to the output memory buffer (from the working memory buffer).
- Global memory
  - This memory buffer is shared by all executing threads. Threads must use explicit instructions to access it.

Memory buffers are implemented as integer to double maps. Instruction use their integer arguments to specify locations in memory. In this work, evolving programs do not have de-referencing functionality (i.e., memory pointers).

4.2 Mutation operators

- Single-instruction insertions
  - Applied per instruction
- Single-instruction deletions
  - Applied per instruction
- Single-instruction substitutions
  - Applied per instruction
- Single-argument substitutions
  - Applied per argument
- Slip mutations (Lalejini et al., 2017)
  - Applied at a per-function rate.
  - Pick two random positions in function's instructions sequence: A and B
  - If A < B: duplicate sequence from A to B
  - If A > B: delete sequence from A to B
- Single-function duplications
  - Applied per-function
- Single-function deletions
  - Applied per-function
- Tag bit flips
  - Applied at a per-bit rate
  - (applies to both instruction- and function-tags)

4.3 Instruction set

Abbreviations:

- EOP: End of program
- Reg: local register
  - Reg[0] indicates the value at the register specified by an instruction's first *argument* (either tag-based or numeric), Reg[1] indicates the value at the register specified by an instruction's second argument, and Reg[2] indicates the value at the register specified by the instruction's third argument.

  - Reg[0], Reg[1], *etc*: Register 0, Register 1, *etc.*
- Input: input buffer
  - Follows same scheme as Reg
- Output: output buffer
  - Follows same scheme as Reg
- Global: global memory buffer
  - Follows same scheme as Reg
- Arg: Instruction argument
  - Arg[i] indicates the i'th instruction argument (an integer encoded in the genome)
  - E.g., Arg[0] is an instruction's first argument

Instructions that would produce undefined behavior (e.g., division by zero) are treated as no operations.

4.3.1 Default Instructions

I.e., instructions used across all diagnostic tasks.

Instruction	Arguments Used	Description
`Nop`	0	No operation
`Not`	1	Reg[0] = !Reg[0]
`Inc`	1	Reg[0] = Reg[0] + 1
`Dec`	1	Reg[0] = Reg[0] - 1
`Add`	3	Reg[2] = Reg[0] + Reg[1]
`Sub`	3	Reg[2] = Reg[0] - Reg[1]
`Mult`	3	Reg[2] = Reg[0] * Reg[1]
`Div`	3	Reg[2] = Reg[0] / Reg[1]
`Mod`	3	Reg[2] = Reg[0] % Reg[1]
`Nand`	2	Reg[2] = !(Reg[0] & Reg[1])
`TestEqu`	3	Reg[2] = Reg[0] == Reg[1]
`TestNEqu`	3	Reg[2] = Reg[0] != Reg[1]
`TestLess`	3	Reg[2] = Reg[0] < Reg[1]
`TestLessEqu`	3	Reg[2] = Reg[0] <= Reg[1]
`TestGreater`	3	Reg[2] = Reg[0] > Reg[1]

Instruction	Arguments Used	Description
`TestGreaterEqu`	3	Reg[2] = Reg[0] >= Reg[1]
`SetMem`	2	Reg[0] = Arg[1]
`Terminal`	1	Reg[0] = double value encoded by instruction tag
`CopyMem`	2	Reg[0] = Reg[1]
`SwapMem`	2	Swap(Reg[0], Reg[1])
`InputToWorking`	2	Reg[1] = Input[0]
`WorkingToOutput`	2	Output[1] = Reg[0]
`If`	1	If Reg[0] != 0, proceed. Otherwise skip to the next `Close` or EOP.
`While`	1	While Reg[0] != 0, loop. Otherwise skip to next `Close` or EOP.
`Close`	0	Indicate the end of a control block of code (e.g., loop, if).
`Break`	0	Break out of current control flow (e.g., loop).
`Terminate`	0	Kill thread that this instruction is executing on.
`Fork`	0	Generate an internal signal (using this instruction's tag) that can trigger a function to run in parallel.
`Call`	0	Call a function, using this instruction's tag to determine which function is called.
`Routine`	0	Same as call, but local memory is shared. Sort of like a jump that will jump back when the routine ends.

Instruction	Arguments Used	Description
`Return`	0	Return from the current function call.

Note that `Nand` performs a bitwise operation.

4.3.2 Global memory access instructions

For experimental conditions without global memory access, these instructions are replaced with no-operation such that the instruction set remains a constant size regardless of experimental condition.

Instruction	Arguments Used	Description
`WorkingToGlobal`	2	Global[1] = Reg[0]
`GlobalToWorking`	2	Reg[1] = Global[0]
`FullGlobalToWorking`	0	Copy entire global memory buffer into working memory buffer
`FullWorkingToGlobal`	0	Copy entire working memory buffer into global memory buffer

Note that full buffer copies only copy registers that have been set (they do not necessarily stomp all over the entire destination buffer).

4.3.3 Regulation instructions

For experimental conditions without regulation, these instructions are replaced with no-operation such that the instruction set remains a constant size regardless of experimental condition.

Note that several regulation instructions have a baseline and (-) version. The (-) versions are identical to the baseline version, except that they multiply the value they are regulating with by -1. This eliminates any bias toward either up-/down-regulation.

Also note that the emp::MatchBin (in the Empirical library) data structure that manages function regulation is defined in terms of tag DISTANCE, not similarity. So, decreasing function regulation values decreases the distance between potential referring tags, and thus, unintuitively, *up-regulates* the function.

All tag-based referencing used by regulation instructions use unregulated, raw match scores. Thus, programs can still up-regulate a function that was previously

'turned off' with down-regulation.

Instruction	Arguments Used	Description
`SetRegulator`	1	Set regulation value of function (targeted with instruction tag) to Reg[0].
`SetRegulator-`	1	Set regulation value of function (targeted with instruction tag) to -1 * Reg[0].
`SetOwnRegulator`	1	Set regulation value of function (currently executing) to Reg[0].
`SetOwnRegulator-`	1	Set regulation value of function (currently executing) to -1 * Reg[0].
`AdjRegulator`	1	Regulation value of function (targeted with instruction tag) += Reg[0]
`AdjRegulator-`	1	Regulation value of function (targeted with instruction tag) -= Reg[0]
`AdjOwnRegulator`	1	Regulation value of function (currently executing) += Reg[0]
`AdjOwnRegulator-`	1	Regulation value of function (currently executing) -= Reg[0]
`ClearRegulator`	0	Clear function regulation (reset to neutral) of function targeted by instruction's tag.
`ClearOwnRegulator`	0	Clear function regulation (reset to neutral) of currently executing function
`SenseRegulator`	1	Reg[0] = regulator state of function targeted by instruction tag

Instruction	Arguments Used	Description
`SenseOwnRegulator`	1	Reg[0] = regulator state of current function
`IncRegulator`	0	Increment regulator state of function targeted with this instruction's tag
`IncOwnRegulator`	0	Increment regulator state of currently executing function
`DecRegulator`	0	Decrement regulator state of function targeted with this instruction's tag
`DecOwnRegulator`	0	Decrement regulator state of the currently executing function

4.3.4 Task-specific instructions

Each task as a number of response instructions added to the instruction set equal to the possible set of responses that can be expressed by a program. Each of these response instructions set a flag on the virtual hardware indicating which response the program expressed and reset all executing threads such that only function regulation and global memory contents persist.

Chapter 5

Exponential regulator

For this work we used a simple exponential function to apply regulatory modifiers to tag-match scores (see details in paper).

5.1 Analysis Dependencies

Load all required R libraries.

```
library(ggplot2)
library(tidyverse)
library(cowplot)
library(viridis)
library(RColorBrewer)
```

These analyses were conducted in the following computing environment:

```
print(version)
```

```
##                _
## platform       x86_64-pc-linux-gnu
## arch           x86_64
## os             linux-gnu
## system         x86_64, linux-gnu
## status
## major          4
## minor          0.2
## year           2020
## month          06
## day            22
## svn rev        78730
## language       R
```

```
## version.string R version 4.0.2 (2020-06-22)
## nickname       Taking Off Again
```

Configuration:

```
theme_set(theme_cowplot())
output_directory <- "media/"
```

5.2 Regulator modifier equation

Below is the function we use to apply regulation to module match scores.

```
exp_base <- 1.1
# Exponential regulator
exp_regulator <- function(raw_match, modifier, base=1.1) {
  return(raw_match * (base**modifier));
}
```

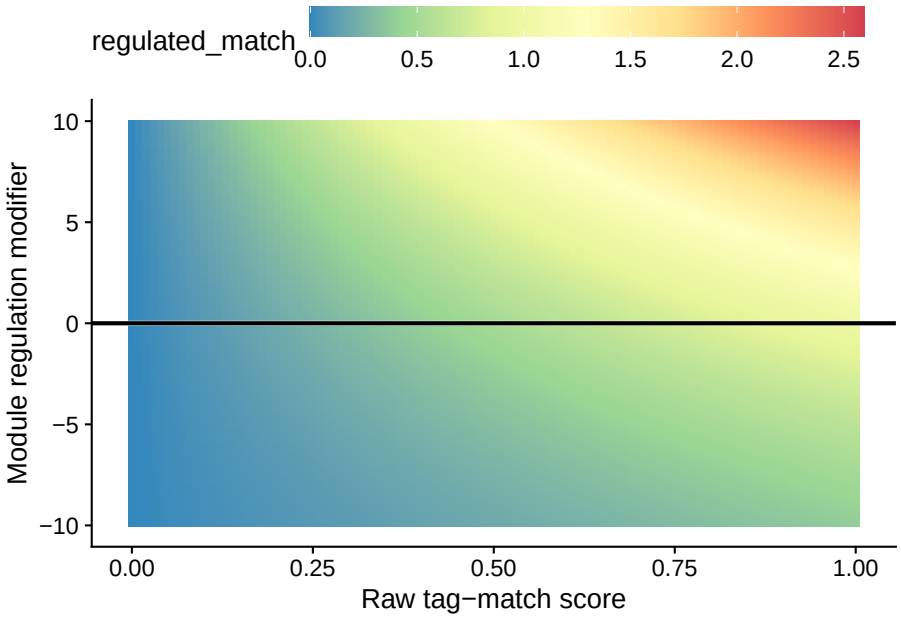
5.3 Regulator behavior

Generate data to graph. We want to visualize regulated match scores as a function of raw tag match scores and module regulatory modifiers.

```
# generate data to visualize
data <- expand_grid(
  raw_match=seq(0, 1.0, 0.01),
  reg_modifier=seq(-10, 10, 0.1)
)

data <- as.data.frame(data)
data$regulated_match <- mapply(
  exp_regulator,
  data$raw_match,
  data$reg_modifier,
  exp_base
)
data$above_perfect <- data$regulated_match > 1.0
```

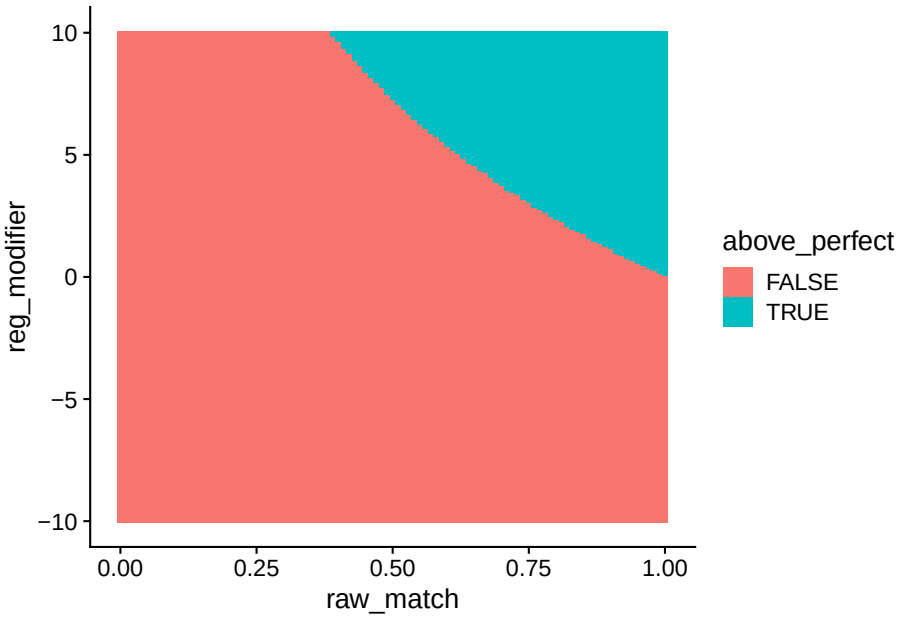
```
ggplot(data, aes(y=reg_modifier, x=raw_match, fill=regulated_match)) +
  geom_raster() +
  geom_hline(yintercept=0, size=1.2, color="white") +
  geom_hline(yintercept=0, size=1, color="black") +
  scale_x_continuous(name="Raw tag-match score") +
  scale_y_continuous(name="Module regulation modifier") +
  scale_fill_distiller(palette = "Spectral") +
  theme(
```

```
    legend.position = "top",
    legend.key.width=unit(2, 'cm')
  ) +
  ggsave(
    paste0(output_directory, "exp-reg-match.png"),
    width=10,
    height=10
  )
```



Does a given raw match + regulatory modifier beat a perfect match (with no regulation)?

```
ggplot(data, aes(y=reg_modifier, x=raw_match, fill=above_perfect)) +
  geom_tile() +
  ggsave(
    paste0(output_directory, "exp-reg-match-above-perfect.png"),
    width=10,
    height=10
  )
```


10
5
0
−5
−10
reg_modifier
0.00
0.25
0.50
0.75
1.00
raw_match
above_perfect
FALSE
TRUE

Chapter 6

Changing-signal problem analysis

Here, we give an overview of the changing-signal diagnostic problem, and we provide our data analyses for related experiments. All of our source code for statistical analyses and data visualizations is embedded in this document. The raw data can be found on the OSF project associated with this work (Lalejini et al., 2020).

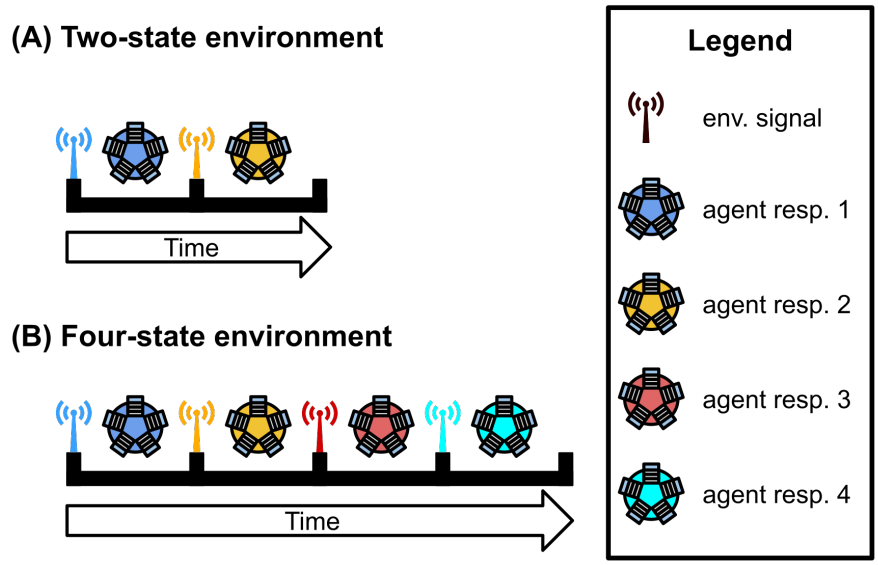
Please file an issue or make a pull request on github to report any mistakes, ask questions, request more explanation, et cetera.

6.1 Overview

```
# Experimental parameters referenced in-text all in one convenient place.
time_steps <- 128
replicates <- 200
population_size <- 1000
generations <- 10000
env_complexities <- c(16)

# Settings for statistical analyses.
alpha <- 0.05
correction_method <- "bonferroni"

# Relative location of data.
working_directory <- "experiments/2020-11-11-chg-sig/analysis/" # << For bookdown
# working_directory <- "./"                                     # << For local analysis
```

The changing-signal task requires programs to express a unique response for

each of K distinct environmental signals (i.e., each signal has a distinct tag); the figure below is given as an example. Because signals are distinct, programs do not need to alter their responses to particular signals over time. Instead, programs may 'hardware' each of the K possible responses to the appropriate environmental signal. However, environmental signals are presented in a random order; thus, the correct *order* of responses will vary and cannot be hardcoded. As in the repeated signal task, programs respond by executing one of K response instructions. Otherwise, evaluation (and fitness assignment) on the changing-signal task mirrors that of the repeated signal task.



Requiring programs to express a distinct instruction in response to each environmental signal represents programs having to perform distinct behaviors.

We afforded programs 128 time steps to express the appropriate response after receiving an environmental signal. Once the allotted time to respond expires or the program expresses any response, the program's threads of execution are reset, resulting in a loss of all thread-local memory. *Only* the contents of a program's global memory and each function's regulatory state persist. The environment then produces the next signal (distinct from all previous signals) to which the program may respond. A program's fitness is equal to the number of correct responses expressed during evaluation.

We evolved populations of 1000 SignalGP programs to solve the changing-signal task at $K = 16$ (where K denotes the number of environmental signals). We evolved populations for 10^4 generations or until an program capable of achieving a perfect score during task evaluation (i.e., able to express the appropriate response to each of the K signals) evolved.

We ran 200 replicate populations (each with a distinct random number seed) of each of the following experimental conditions:

1. a regulation-enabled treatment where programs have access to genetic regulation.
2. a regulation-disabled treatment where programs do not have access to genetic regulation.

Note this task does not require programs to shift their response to particular signals over time, and as such, genetic regulation is unnecessary. Further, because programs experience environmental inputs in a random order, erroneous genetic regulation can manifest as cryptic variation. For example, non-adaptive down-regulation of a particular response function may be neutral given one sequence of environmental signals, but may be deleterious in another. **We expected regulation-enabled SignalGP to exhibit non-adaptive plasticity, potentially resulting in slower adaptation and non-general solutions.**

6.2 Analysis Dependencies

Load all required R libraries.

```
library(ggplot2)
library(tidyverse)
library(cowplot)
library(viridis)
source("https://gist.githubusercontent.com/benmarwick/2a1bb0133ff568cbe28d/raw/fb53bd97121f7f9ce9
```

These analyses were conducted in the following computing environment:

```
print(version)
```

```
##                _
## platform       x86_64-pc-linux-gnu
## arch           x86_64
## os             linux-gnu
## system         x86_64, linux-gnu
## status
## major          4
## minor          0.2
## year           2020
## month          06
## day            22
## svn rev        78730
## language       R
## version.string R version 4.0.2 (2020-06-22)
## nickname       Taking Off Again
```

6.3 Setup

Load data, initial data cleanup, configure some global settings.

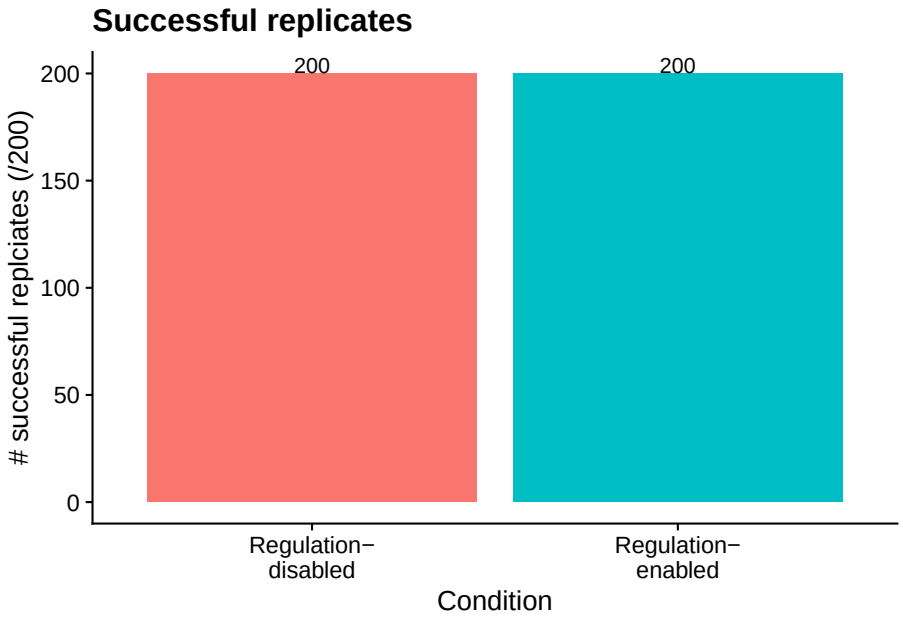
```
# Load data file
data_loc <- paste0(working_directory, "data/max_fit_orgs.csv")
data <- read.csv(data_loc, na.strings="NONE")

# Define function to summarize regulation/memory configurations.
get_con <- function(reg, mem) {
  if (reg == "0" && mem == "0") {
    return("none")
  } else if (reg == "0" && mem=="1") {
    return("memory")
  } else if (reg=="1" && mem=="0") {
    return("regulation")
  } else if (reg=="1" && mem=="1") {
    return("both")
  } else {
    return("UNKNOWN")
  }
}

# Specify experimental condition for each datum.
data$condition <- mapply(
  get_con,
  data$USE_FUNC_REGULATION,
  data$USE_GLOBAL_MEMORY
)

data$condition <- factor(
  data$condition,
  levels=c("regulation", "memory", "none", "both")
)

# For convenience, create a data set with only solutions
# Filter data to include only replicates labeled as solutions
sol_data <- filter(
  data,
  solution=="1"
)

# A lookup table for task complexities
task_label_lu <- c(
  "2" = "2-signal task",
  "4" = "4-signal task",
```

```
  "8" = "8-signal task",
  "16" = "16-signal task",
  "32" ="32-signal task"
)

# Configure our default graphing theme
theme_set(theme_cowplot())
```

6.4 Does regulation hinder the evolution of successful genotypes?

Here, we look at the number of solutions evolved under regulation-enabled and regulation-disabled conditions. A program is categorized as a 'solution' if it can correctly respond to each of the K environmental signals *during evaluation*.

```
# Graph the number of solutions evolved in each condition, faceted by environmental complexity
ggplot( sol_data, aes(x=condition, fill=condition) ) +
  geom_bar() +
  geom_text(
    stat="count",
    mapping=aes(label=..count..),
    position=position_dodge(0.9),
    vjust=0
  ) +
  scale_x_discrete(
    name="Condition",
    breaks=c("memory","both"),
    labels=c("Regulation-\ndisabled","Regulation-\nenabled")
  ) +
  ylab("# successful replciates (/200)") +
  theme(legend.position = "none") +
  ggtitle("Successful replicates")
```



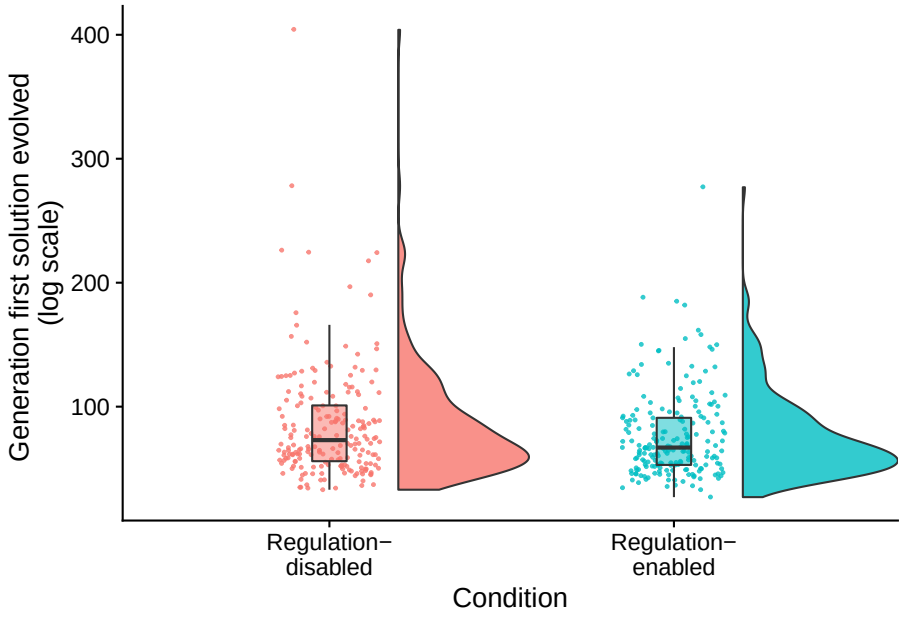
Programs capable of achieving a perfect score on the changing-signal task (for a given sequence of environment signals) evolve in all 200 replicates of each condition (i.e., with and without access to genetic regulation). These programs, however, do not necessarily generalize across all possible sequences of environmental signals.

6.4.1 Does access to regulation slow adaptation?

I.e., did successful regulation-enabled programs take longer (more generations) to evolve than those evolved in the regulation-disabled treatment?

```
ggplot( sol_data, aes(x=condition, y=update, fill=condition) ) +
  geom_flat_violin(
    position = position_nudge(x = .2, y = 0),
    alpha = .8
  ) +
  geom_point(
    aes(y = update, color = condition),
    position = position_jitter(width = .15),
    size = .5,
    alpha = 0.8
  ) +
  geom_boxplot(
    width = .1,
    outlier.shape = NA,
    alpha = 0.5
```

```
  ) +
  scale_x_discrete(
    name="Condition",
    breaks=c("memory", "both"),
    labels=c("Regulation-\ndisabled","Regulation-\nenabled")
  ) +
  scale_y_continuous(
    name="Generation first solution evolved \n(log scale)",
  ) +
  guides(fill = FALSE) +
  guides(color = FALSE)
```




```
print(wilcox.test(formula=update~condition, data=sol_data, exact=FALSE, conf.int=TRUE))
```

```
##
##  Wilcoxon rank sum test with continuity correction
##
## data:  update by condition
## W = 22188, p-value = 0.05845
## alternative hypothesis: true location shift is not equal to 0
## 95 percent confidence interval:
##  -3.860236e-05  1.000000e+01
## sample estimates:
## difference in location
##               5.000013
```

The difference in the number of generations before a solution arises is not significantly different.

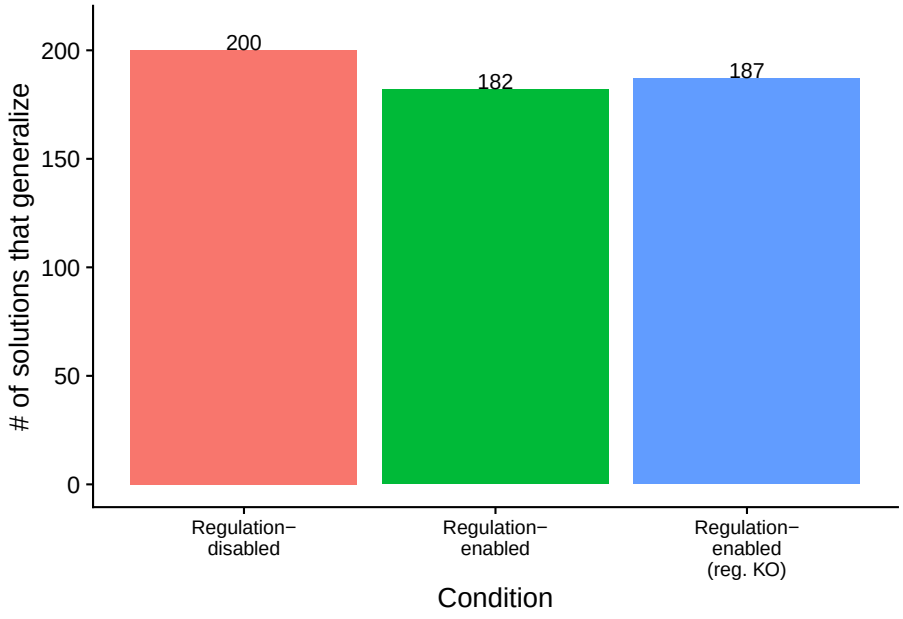
6.4.2 Do they generalize?

Note that solutions may or may not generalize beyond the sequence of environmental signals on which they achieved a perfect score (and were thus categorized as a 'solution'). We re-evaluated each 'solution' on a random sample of 5000 sequences of environmental signals to test for generalization. We deem programs as having successfully generalized only if they responded correctly in all 5000 tests.

To see if regulation is preventing some regulation-enabled solutions from generalizing, we test generalization for regulation-enabled solutions with their regulation faculties knocked out (i.e., regulation instructions replaced with no-operations).

```
# Grab count data to make bar plot life easier
num_solutions_reg <- length(filter(data, condition=="both" & solution=="1")$SEED)
num_generalize_reg <- length(filter(data, condition=="both" & all_solution=="1")$SEED)
num_generalize_ko_reg <- length(filter(data, condition=="both" & all_solution_ko_reg=="

num_generalize_mem <- length(filter(data, condition=="memory" & all_solution=="1")$SEE

sol_cnts <- data.frame(x=1:3)
sol_cnts$type <- c("reg_generalize", "reg_generalize_ko_reg", "mem_generalize")
sol_cnts$val <- c(num_generalize_reg, num_generalize_ko_reg, num_generalize_mem)

ggplot( sol_cnts, aes(x=type, y=val, fill=type) ) +
  geom_bar(stat="identity") +
  geom_text(
    aes(label=val),
    stat="identity",
    position=position_dodge(0.75),
    vjust=-0.01
  ) +
  scale_x_discrete(
    name="Condition",
    limits=c(
      "mem_generalize",
      "reg_generalize",
      "reg_generalize_ko_reg"
      ),
    labels=c(
      "Regulation-\ndisabled",
      "Regulation-\nenabled",
      "Regulation-\nenabled\n(reg. KO)"
    )
```

```
  ) +
  scale_y_continuous(
    name="# of solutions that generalize",
    limits=c(0, 210),
    breaks=seq(0,200,50)
  ) +
  theme(
    legend.position="none",
    axis.text.x = element_text(size=10)
  ) +
  ggsave(paste0(working_directory, "imgs/chg-env-16-generalization.png"), width=4,height=4)
```



All regulation-disabled programs successfully generalized.

```
table <- matrix(c(num_generalize_reg,
                  num_generalize_mem,
                  200 - num_generalize_reg,
                  200 - num_generalize_mem),
                nrow=2)
rownames(table) <- c("reg-augmented", "reg-disabled")
colnames(table) <- c("success", "fail")
print(table)
```

```
##               success fail
## reg-augmented     182   18
```

```
## reg-disabled      200    0
```

```
fisher.test(table)
```

```
##
##  Fisher's Exact Test for Count Data
##
## data:  table
## p-value = 5.113e-06
## alternative hypothesis: true odds ratio is not equal to 1
## 95 percent confidence interval:
##  0.0000000 0.2115509
## sample estimates:
## odds ratio
##          0
```

The difference in number of generalizing solutions between regulation-enabled and regulation-disabled conditions is statistically significant (Fisher's exact test).

Moreover, 5 of the 18 non-generalizing programs generalize when we knockout genetic regulation. Upon close inspection, the other 13 non-general programs relied on genetic regulation to achieve initial success but failed to generalize to arbitrary environment signal sequences.

Chapter 7

Repeated-signal problem analysis

Here, we give an overview of the repeated-signal diagnostic problem, and we provide our data analyses for related experiments. All of our source code for statistical analyses and data visualizations is embedded in this document. The raw data can be found on the OSF project associated with this work (Lalejini et al., 2020).

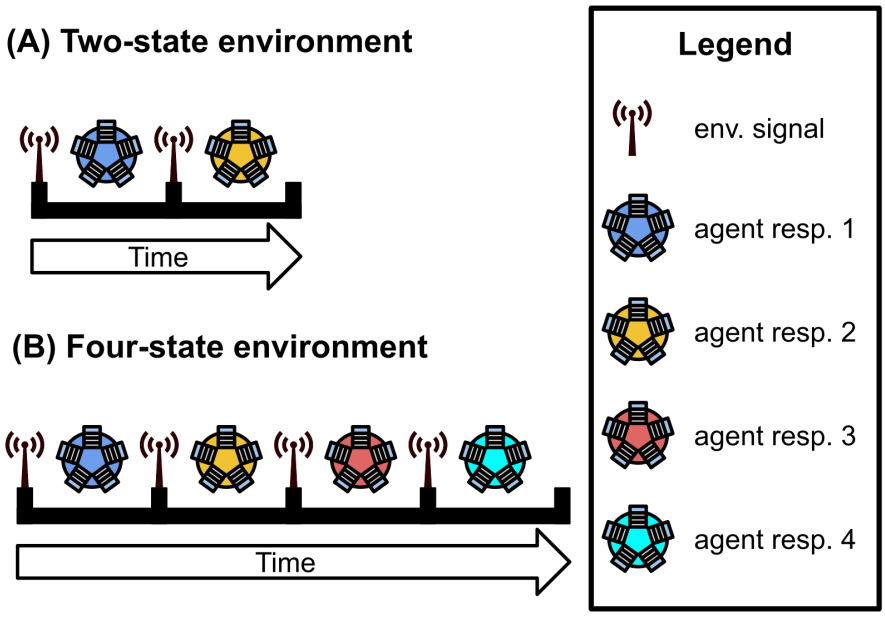
Please file an issue or make a pull request on github to report any mistakes, ask questions, request more explanation, et cetera.

7.1 Overview

```
# Experimental parameters referenced in-text all in one convenient place.
time_steps <- 128
replicates <- 200
population_size <- 1000
generations <- 10000
env_complexities <- c(2, 4, 8, 16)

# Settings for statistical analyses.
alpha <- 0.05
correction_method <- "bonferroni"

# Relative location of data.
working_directory <- "experiments/2020-11-25-rep-sig/analysis/" # << For bookdown
# working_directory <- "./"                                     # << For local analysis
```

The repeated-signal problem requires programs to output the appropriate (dis-

tinct) response to a single environmental signal each of the K times the 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.



We afford programs 128 time steps to respond to each environmental signal. Once the allotted time expires or the program executes any response, the program's threads of execution are 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 correct responses given during evaluation, and a program is considered a solution if it correctly responds to each of the K environmental signals.

7.2 Analysis Dependencies

Load all required R libraries.

```
library(ggplot2)
library(tidyverse)
library(reshape2)
library(cowplot)
library(viridis)
library(igraph)

source("https://gist.githubusercontent.com/benmarwick/2a1bb0133ff568cbe28d/raw/fb53bd97121f7f9ce9
```

These analyses were conducted in the following computing environment:

```
print(version)
```

```
##                _
## platform       x86_64-pc-linux-gnu
## arch           x86_64
## os             linux-gnu
## system         x86_64, linux-gnu
## status
## major          4
## minor          0.2
## year           2020
## month          06
## day            22
## svn rev        78730
## language       R
## version.string R version 4.0.2 (2020-06-22)
## nickname       Taking Off Again
```

7.3 Setup

Load data, initial data cleanup, configure some global settings.

```
max_fit_org_data_loc <- paste0(working_directory, "data/max_fit_orgs_noprogram.csv")
reg_network_data_loc <- paste0(working_directory, "data/reg_graphs_summary.csv")
inst_exec_data_loc <- paste0(working_directory, "data/exec_trace_summary.csv")

####### Load max fit program data #######
max_fit_org_data <- read.csv(max_fit_org_data_loc, na.strings="NONE")

# Specify factors (not all of these matter for this set of runs).
max_fit_org_data$matchbin_thresh <- factor(
  max_fit_org_data$matchbin_thresh,
  levels=c(0, 25, 50, 75)
)
```

```
max_fit_org_data$NUM_SIGNAL_RESPONSES <- factor(
  max_fit_org_data$NUM_SIGNAL_RESPONSES,
  levels=c(2, 4, 8, 16, 32)
)

max_fit_org_data$NUM_ENV_CYCLES <- factor(
  max_fit_org_data$NUM_ENV_CYCLES,
  levels=c(2, 4, 8, 16, 32)
)

max_fit_org_data$TAG_LEN <- factor(
  max_fit_org_data$TAG_LEN,
  levels=c(32, 64, 128, 256)
)

# Define function to summarize regulation/memory configurations.
get_con <- function(reg, mem) {
  if (reg == "0" && mem == "0") {
    return("none")
  } else if (reg == "0" && mem=="1") {
    return("memory")
  } else if (reg=="1" && mem=="0") {
    return("regulation")
  } else if (reg=="1" && mem=="1") {
    return("both")
  } else {
    return("UNKNOWN")
  }
}

# Specify experimental condition for each datum.
max_fit_org_data$condition <- mapply(
  get_con,
  max_fit_org_data$USE_FUNC_REGULATION,
  max_fit_org_data$USE_GLOBAL_MEMORY
)

max_fit_org_data$condition <- factor(
  max_fit_org_data$condition,
  levels=c("regulation", "memory", "none", "both")
)

# Does this program rely on a stochastic strategy?
max_fit_org_data$stochastic <- 1 - max_fit_org_data$consistent
```

```
max_fit_org_data$stochastic <- factor(
  max_fit_org_data$stochastic,
  levels=c(0, 1)
)

# Filter data to include only runs from regulation-enabled ('both') and regulation-disabled ('mem
max_fit_org_data <- filter(max_fit_org_data, condition %in% c("both", "memory"))

# Filter data to include only replicates labeled as solutions
sol_data <- filter(max_fit_org_data, solution=="1")

# Label solution strategies
get_strategy <- function(use_reg, use_mem) {
  if (use_reg=="0" && use_mem=="0") {
    return("use neither")
  } else if (use_reg=="0" && use_mem=="1") {
    return("use memory")
  } else if (use_reg=="1" && use_mem=="0") {
    return("use regulation")
  } else if (use_reg=="1" && use_mem=="1") {
    return("use both")
  } else {
    return("UNKNOWN")
  }
}

# Specify experimental conditions (to make labeling easier).
sol_data$strategy <-  mapply(
  get_strategy,
  sol_data$relies_on_regulation,
  sol_data$relies_on_global_memory
)

sol_data$strategy <- factor(
  sol_data$strategy,
  levels=c(
    "use regulation",
    "use memory",
    "use neither",
    "use both"
  )
)

####### Load network data #######
reg_network_data <- read.csv(reg_network_data_loc, na.strings="NA")
```

```
reg_network_data <- filter(reg_network_data, run_id %in% max_fit_org_data$SEED)

# Make a lookup function to get each run's environment complexity level.
get_num_sig_resps <- function(seed) {
  return(filter(max_fit_org_data, SEED==seed)$NUM_SIGNAL_RESPONSES)
}

reg_network_data$NUM_SIGNAL_RESPONSES <- mapply(
  get_num_sig_resps,
  reg_network_data$run_id
)

reg_network_data$NUM_SIGNAL_RESPONSES <- factor(reg_network_data$NUM_SIGNAL_RESPONSES)


####### Load instruction execution data #######
inst_exec_data <- read.csv(inst_exec_data_loc, na.strings="NA")

inst_exec_data$condition <- mapply(
  get_con,
  inst_exec_data$USE_FUNC_REGULATION,
  inst_exec_data$USE_GLOBAL_MEMORY
)

inst_exec_data$condition <- factor(
  inst_exec_data$condition,
  levels=c("regulation", "memory", "none", "both")
)

inst_exec_data$NUM_SIGNAL_RESPONSES <- factor(
  inst_exec_data$NUM_SIGNAL_RESPONSES,
  levels=c(2, 4, 8, 16, 32)
)

inst_exec_data$NUM_ENV_CYCLES <- factor(
  inst_exec_data$NUM_ENV_CYCLES,
  levels=c(2, 4, 8, 16, 32)
)

# Labels for each
label_lu <- c(
  "2" = "2-signal task",
  "4" = "4-signal task",
  "8" = "8-signal task",
  "16" = "16-signal task",
```

```
  "32" ="32-signal task"
)

####### misc #######
# Configure our default graphing theme
theme_set(theme_cowplot())
```

7.4 Problem-solving success

We expected populations with access to genetic regulation to be more successful on the repeated-signal task than those evolved without access to genetic regulation. Further, we expected the success differential to increase with problem difficulty.

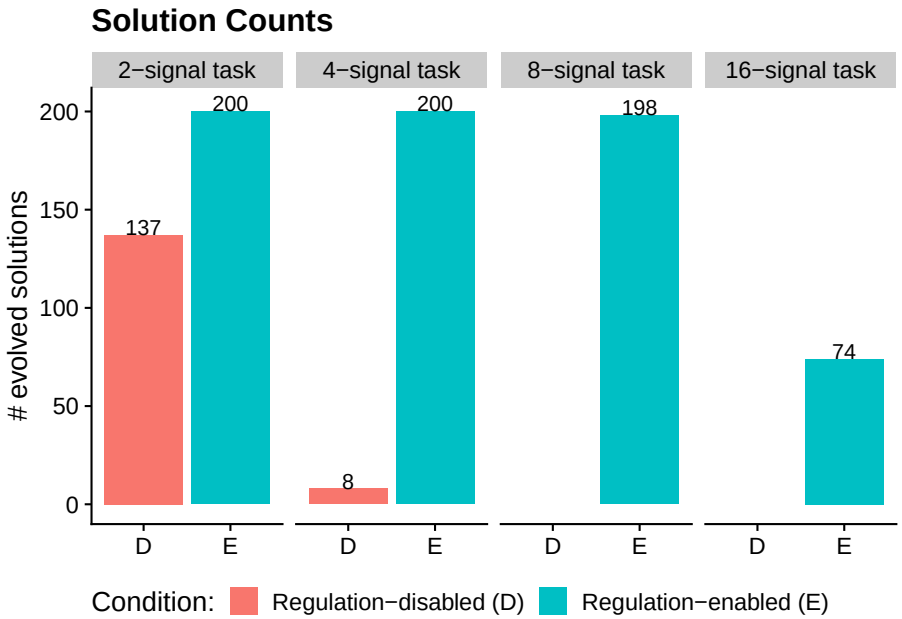
We can look at (1) the number of successful replicates (i.e., replicates in which a program capable of perfectly solving the repeated signal task evolved) per condition and (2) the scores of the highest-fitness program evolved in each replicate.

7.4.1 Number of successful replicates by condition

Note that a program is categorized as a 'solution' only if it can correctly respond to each of repetition of the environment signal.

```
# Graph the number of solutions evolved in each condition, faceted by environmental complexity
ggplot( sol_data, aes(x=condition, fill=condition) ) +
  geom_bar() +
  geom_text(
    stat="count",
    mapping=aes(label=..count..),
    position=position_dodge(0.9),
    vjust=0
  ) +
  scale_y_continuous(
    name="# evolved solutions",
    breaks=seq(0, replicates, 50),
    limits=c(0, replicates+2)
  ) +
  scale_fill_discrete(
    name="Condition:",
    limits=c("memory", "both"),
    labels=c("Regulation-disabled (D)", "Regulation-enabled (E)")
  ) +
  scale_x_discrete(
    name="Condition",
```

```
    limits=c("memory", "both"),
    labels=c("D", "E")
  ) +
  facet_wrap(
    ~ NUM_SIGNAL_RESPONSES,
    nrow=1,
    labeller=labeller(NUM_SIGNAL_RESPONSES=label_lu)
  ) +
  ggtitle("Solution Counts") +
  theme(
    legend.position="bottom",
    axis.title.x=element_blank()
  ) +
  ggsave(
    paste0(working_directory, "imgs/repeated-signal-solultion-cnts.png"),
    width=8,
    height=4
  )
```



We confirmed that each difficulty level of the repeated-signal problem is solvable without regulation using hand-coded SignalGP programs.

We use a Fisher's exact test to determine if there are significant differences ($p < 0.05$) between the numbers of regulation-enabled versus regulation-disabled solutions for each problem difficulty.

```r
# This code chunk is sort of a monster to have things print out all pretty-like in the knitted HT

# For each environment complexity level, do a fisher's exact test and print results.
for (env in env_complexities) {
  env_data <- filter(max_fit_org_data, NUM_SIGNAL_RESPONSES==env)
  cat("#### ", paste0(env, "-signal task"), " - statistical analysis of solution counts  \n")

  # Extract successes/fails for each condition.
  mem_success_cnt <- nrow(filter(env_data, solution=="1" & condition=="memory"))
  mem_fail_cnt <- nrow(filter(env_data, condition=="memory")) - mem_success_cnt
  both_success_cnt <- nrow(filter(env_data, solution=="1" & condition=="both"))
  both_fail_cnt <- nrow(filter(env_data, condition=="both")) - both_success_cnt

  # Regulation-disabled vs regulation-enabled
  mem_sgp_table <- matrix(c(both_success_cnt,
                            mem_success_cnt,
                            both_fail_cnt,
                            mem_fail_cnt),
                          nrow=2)
  rownames(mem_sgp_table) <- c("reg-enabled", "reg-disabled")
  colnames(mem_sgp_table) <- c("success", "fail")
  mem_sgp_fishers <- fisher.test(mem_sgp_table)

  cat("\n")
  cat("Regulation-enabled SignalGP vs. regulation-disabled SignalGP (original version of SignalGP
  cat("```\n")
  print(mem_sgp_table)
  print(mem_sgp_fishers)
  cat("```\n")
  cat("\n")
}
```

7.4.1.1 2-signal task - statistical analysis of solution counts

Regulation-enabled SignalGP vs. regulation-disabled SignalGP (original version of SignalGP):

```
             success fail
reg-enabled      200    0
reg-disabled     137   63

    Fisher's Exact Test for Count Data

data:  mem_sgp_table
p-value < 2.2e-16
alternative hypothesis: true odds ratio is not equal to 1
```

```
95 percent confidence interval:
 23.54182      Inf
sample estimates:
odds ratio
       Inf
```

7.4.1.2 4-signal task - statistical analysis of solution counts

Regulation-enabled SignalGP vs. regulation-disabled SignalGP (original version of SignalGP):

```
             success fail
reg-enabled      200    0
reg-disabled       8  192

    Fisher's Exact Test for Count Data

data:  mem_sgp_table
p-value < 2.2e-16
alternative hypothesis: true odds ratio is not equal to 1
95 percent confidence interval:
 953.6049      Inf
sample estimates:
odds ratio
       Inf
```

7.4.1.3 8-signal task - statistical analysis of solution counts

Regulation-enabled SignalGP vs. regulation-disabled SignalGP (original version of SignalGP):

```
             success fail
reg-enabled      198    2
reg-disabled       0  200

    Fisher's Exact Test for Count Data

data:  mem_sgp_table
p-value < 2.2e-16
alternative hypothesis: true odds ratio is not equal to 1
95 percent confidence interval:
 2409.412      Inf
sample estimates:
odds ratio
       Inf
```

7.4.1.4 16-signal task - statistical analysis of solution counts

Regulation-enabled SignalGP vs. regulation-disabled SignalGP (original version of SignalGP):

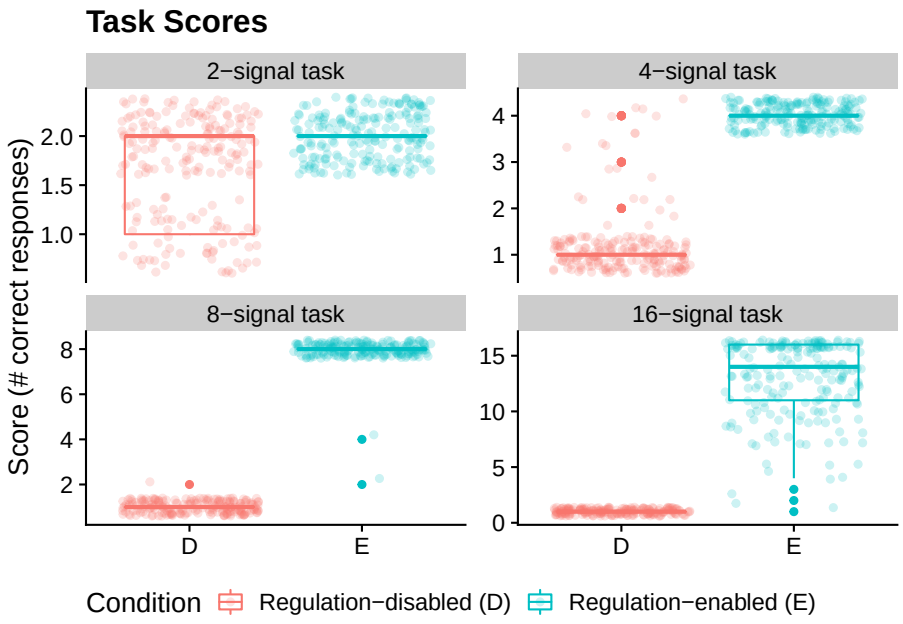
```
             success fail
reg-enabled       74  126
reg-disabled       0  200

    Fisher's Exact Test for Count Data

data:  mem_sgp_table
p-value < 2.2e-16
alternative hypothesis: true odds ratio is not equal to 1
95 percent confidence interval:
 30.12902      Inf
sample estimates:
odds ratio
       Inf
```

7.4.2 Aggregate fitness scores by condition

Here, we visualize the raw task scores for the highest-fitness program from each run across all environments/conditions.

```
ggplot( max_fit_org_data, aes(x=condition, y=score, color=condition) ) +
  geom_boxplot() +
  geom_jitter(alpha=0.2) +
  ylab("Score (# correct responses)") +
  scale_color_discrete(
    name="Condition",
    breaks=c("memory", "both"),
    labels=c("Regulation-disabled (D)", "Regulation-enabled (E)")
  ) +
  scale_x_discrete(
    name="Condition",
    breaks=c("memory", "both"),
    labels=c("D", "E")
  ) +
  facet_wrap(
    ~ NUM_SIGNAL_RESPONSES,
    scales="free_y",
    labeller=labeller(NUM_SIGNAL_RESPONSES=label_lu)
  ) +
  theme(
    legend.position="bottom",
    axis.title.x=element_blank()
```

```
  ) +
  ggtitle("Task Scores") +
  ggsave(
    paste0(working_directory, "imgs/repeated-signal-scores.png"),
    width=16,
    height=8
  )
```



7.5 How many generations elapse before solutions evolve?

Do some conditions lead to the evolution of solutions in fewer generations than other conditions?

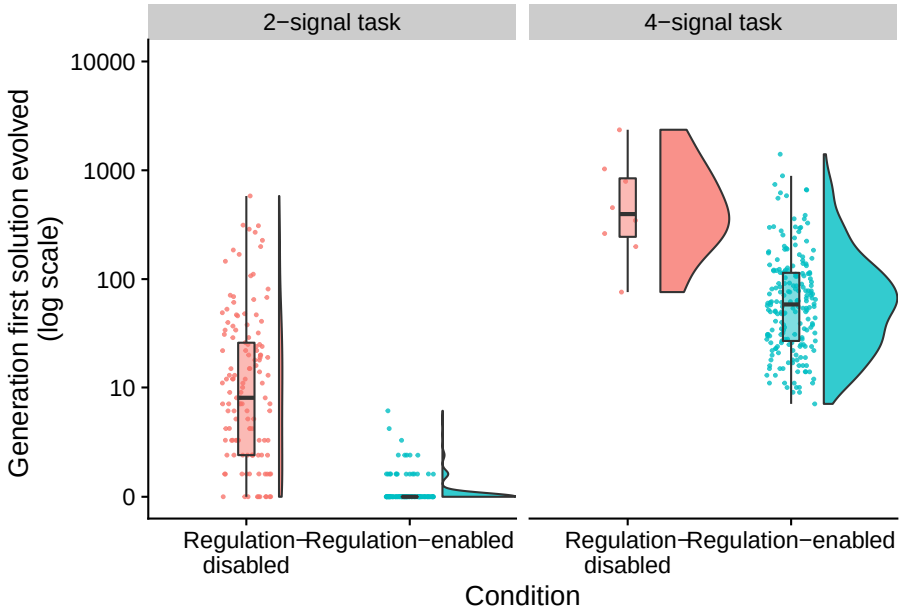
Here, we compare the generation at which solutions arise (only at difficulty levels where regulation-disabled solutions evovled).

```
ggplot( data = filter(sol_data, NUM_SIGNAL_RESPONSES %in% c(2, 4)), aes(x=condition, y=
  geom_flat_violin(
    position = position_nudge(x = .2, y = 0),
    alpha = .8
  ) +
  geom_point(
    aes(y=update, color=condition),
```

```
    position = position_jitter(width = .15),
    size = .5,
    alpha = 0.8
  ) +
  geom_boxplot(
    width = .1,
    outlier.shape = NA,
    alpha = 0.5
  ) +
  scale_x_discrete(
    name="Condition",
    breaks=c("memory", "both"),
    labels=c("Regulation-\ndisabled", "Regulation-enabled")
  ) +
  scale_y_continuous(
    name="Generation first solution evolved \n(log scale)",
    limits=c(0, generations),
    breaks=c(0, 10, 100, 1000, 10000),
    trans="pseudo_log"
  ) +
  facet_wrap(
    ~ NUM_SIGNAL_RESPONSES,
    nrow=1,
    labeller=labeller(NUM_SIGNAL_RESPONSES=label_lu)
  ) +
  guides(fill = FALSE) +
  guides(color = FALSE) +
  ggsave(
    paste0(working_directory, "./imgs/repeated-signal-solve-time-cloud.png"),
    width=5,
    height=4
  )
```

```
## Warning: Removed 104 rows containing missing values (geom_point).
```

```
## Warning: Removed 104 rows containing missing values (geom_point).
```



7.5.1 Two-signal task - statistical analysis

We compare the time to solution using a Wilcoxon rank-sum test.

```
env_2_sol_data <- filter(
  sol_data,
  NUM_SIGNAL_RESPONSES==2
)

print(wilcox.test(formula=update~condition, data=env_2_sol_data, exact=FALSE, conf.int=
```

```
##
##  Wilcoxon rank sum test with continuity correction
##
## data:  update by condition
## W = 24940, p-value < 2.2e-16
## alternative hypothesis: true location shift is not equal to 0
## 95 percent confidence interval:
##   6.000004 11.000006
## sample estimates:
## difference in location
##               7.999963
```

7.5.2 Four-signal task - statistical analysis

We compare the time to solution using a Wilcoxon rank-sum test.

```
env_4_sol_data <- filter(
  sol_data,
  NUM_SIGNAL_RESPONSES==4
)

print(wilcox.test(formula=update~condition, data=env_4_sol_data, exact=FALSE, conf.int=TRUE))
```

```
##
##  Wilcoxon rank sum test with continuity correction
##
## data:  update by condition
## W = 1456, p-value = 8.603e-05
## alternative hypothesis: true location shift is not equal to 0
## 95 percent confidence interval:
##  173 738
## sample estimates:
## difference in location
##                319.636
```

7.6 Teasing apart evolved strategies

We analyzed:

- mechanisms underlying capacity to adjust responses to input signals (using knockout experiments)
- whether programs used stochasticity as part of their strategy
- instruction execution traces

7.6.1 Do solutions rely on genetic regulation or global memory access to dynamically adjust responses?

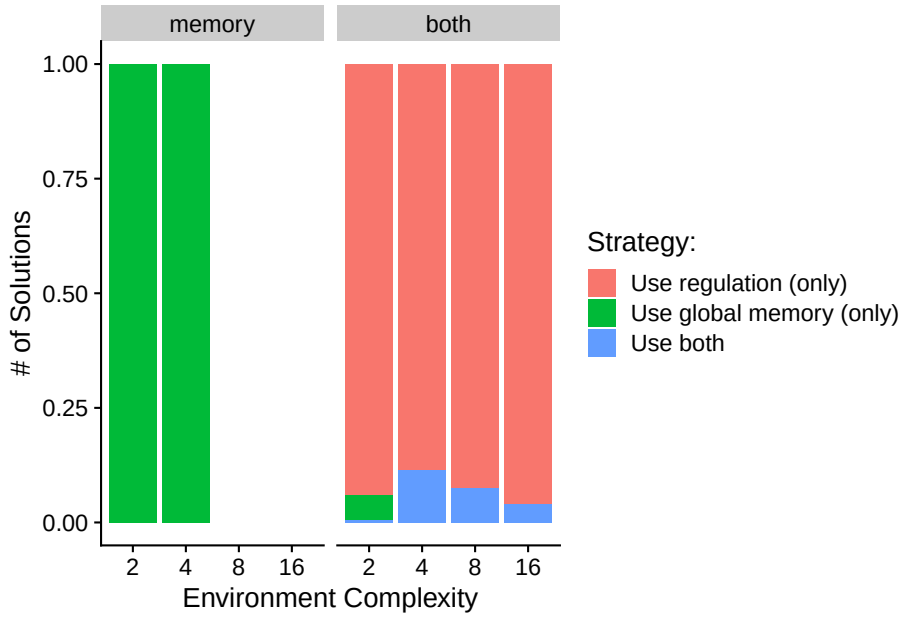
Here, we take a closer at the strategies employed by solutions evolved across environment complexities. For each evolved solution, we independently knocked out (disabled) tag-based regulation and global memory access, and we measured the fitness effects knocking each out. If a knockout resulted in a decrease in fitness, we labeled that program as relying on that functionality (global memory or genetic regulation) for success.

The graph(s) below gives the proportion of solutions that rely exclusively on regulation, exclusively on global memory, on both global memory and regulation, and on neither functionality.

Proportions as stacked bar chart:

```
ggplot( data=sol_data, mapping=aes(x=NUM_SIGNAL_RESPONSES, fill=strategy) ) +
  geom_bar(
    position="fill"
```

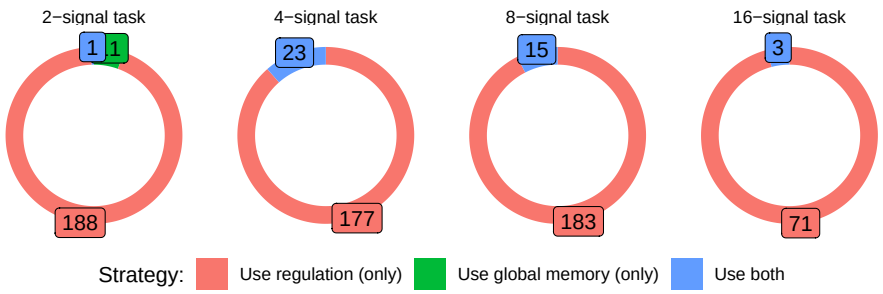
```
  ) +
  ylab("# of Solutions") +
  xlab("Environment Complexity") +
  scale_fill_discrete(
    name="Strategy:",
    breaks=c("use regulation",
             "use memory",
             "use neither",
             "use both"),
    labels=c("Use regulation (only)",
             "Use global memory (only)",
             "Use neither",
             "Use both")
    ) +
  facet_wrap(~condition)
```



As fun donuts(?!):

```
# https://www.r-graph-gallery.com/128-ring-or-donut-plot.html
donut_data <- data.frame(
  env=character(),
  count=numeric(),
  category=character()
)
```

```
for (env in env_complexities) {
  env_donut_data  <- data.frame(
    env=c(env, env, env, env),
    count=c(
      nrow(filter(sol_data, condition=="both" & NUM_SIGNAL_RESPONSES==env & strategy=="use neithe
      nrow(filter(sol_data, condition=="both" & NUM_SIGNAL_RESPONSES==env & strategy=="use memory
      nrow(filter(sol_data, condition=="both" & NUM_SIGNAL_RESPONSES==env & strategy=="use regula
      nrow(filter(sol_data, condition=="both" & NUM_SIGNAL_RESPONSES==env & strategy=="use both")
    ),
    category=c("neither", "memory", "regulation", "both")
  )

  env_donut_data <- filter(env_donut_data, count > 0)
  env_donut_data$fraction <- env_donut_data$count / sum(env_donut_data$count)
  env_donut_data$ymax <- cumsum(env_donut_data$fraction)
  env_donut_data$ymin <- c(0, head(env_donut_data$ymax, n=-1))
  env_donut_data$labelPosition <- (env_donut_data$ymax + env_donut_data$ymin) / 2
  env_donut_data$label <- paste0(env_donut_data$count)

  donut_data<-rbind(donut_data, env_donut_data)
}

ggplot( donut_data, aes(ymax=ymax, ymin=ymin, xmax=4, xmin=3, fill=category) ) +
  geom_rect() +
  geom_label( x=4, aes(y=labelPosition, label=label), size=4, show.legend = FALSE) +
  coord_polar(theta="y") +
  xlim(c(-1, 4)) +
  scale_fill_discrete(
    name="Strategy:",
    limits=c("regulation",
             "memory",
             "both"),
    labels=c("Use regulation (only)",
             "Use global memory (only)",
             "Use both")) +
  theme_void() +
  theme(legend.position = "bottom") +
  facet_wrap(
    ~env,
    nrow=1,
    labeller=labeller(env=label_lu)
  )
```



We can see that in conditions where programs have access to regulation, evolved solutions generally rely on regulation to adjust their responses to input signals. In conditions where memory is the only mechanism for solving the repeated-signal task, we see that all evolved solutions rely exclusively on global memory access for adjusting responses to input signals.

7.6.2 What forms of genetic regulation do evolved programs rely on?

We used two approaches to tease apart forms of genetic regulation that evolved SignalGP programs rely on:

1. We traced program execution step-by-step (including each function's regulatory state) during evaluation on the repeated signal task and extracted regulatory interactions between executing functions as a directed graph. We draw a directed edge from function A to function B if B's regulatory state changes while A is executing. We label each edge as up- or down-regulation. The distribution of edge types in these graphs hints at what strategy the program is using.
2. We independently knockout up-regulation and down-regulation and record the fitness of knockout-variants. If fitness decreases when a target functionality is knocked out, we categorize the program as relying on that functionality.

Note that the knockout data more directly indicates which forms of regulation a program relies on, as the gene regulation networks may include neutral and

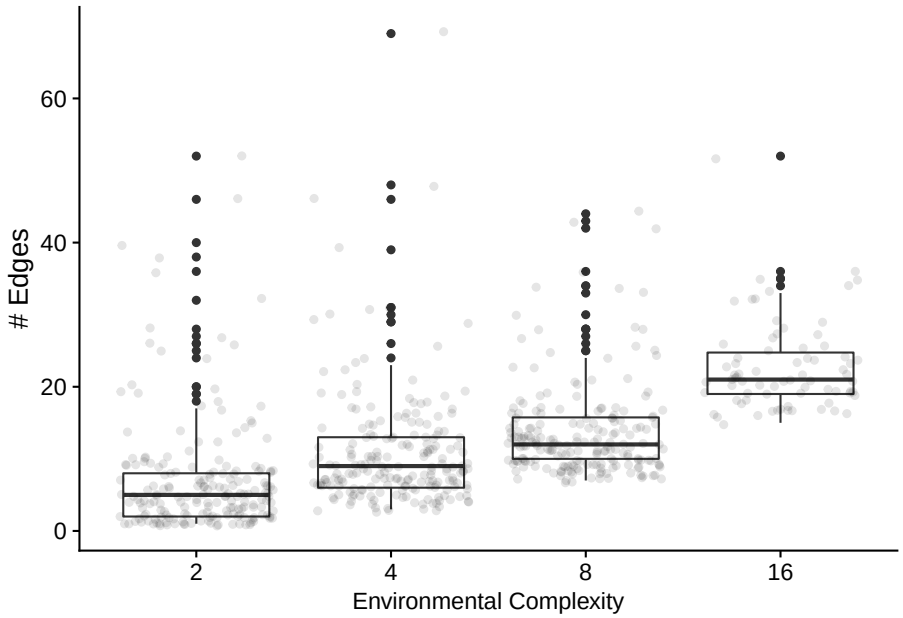
non-adaptive regulatory interactions.

7.6.2.1 Gene regulatory network edges

Let's only look at programs that solved the repeated-signal task and rely on regulation.

First, total edges as a function of problem difficulty.

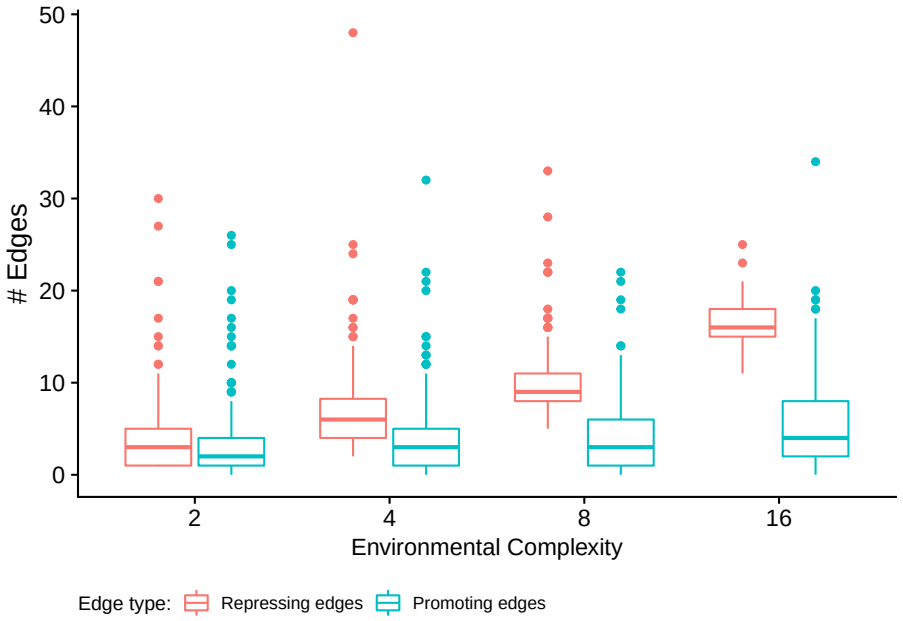
```
relies_on_reg <- filter(
  sol_data,
  relies_on_regulation=="1"
)$SEED

ggplot( filter(reg_network_data, run_id %in% relies_on_reg ), aes(x=NUM_SIGNAL_RESPONSES, y=edge_
  geom_boxplot() +
  geom_jitter(alpha=0.1) +
  xlab("Environmental Complexity") +
  ylab("# Edges") +
  theme(
    legend.position="bottom",
    legend.text=element_text(size=9),
    legend.title=element_text(size=10),
    axis.title.x=element_text(size=12)
  ) +
  ggsave(
    paste0(working_directory, "imgs/repeated-signal-regulation-edges.png"),
    width=4,
    height=3
  )
```



Next, let's look at edges by type.

```
# Get seeds (run ids) of replicates that rely on regulation and are a solution.

melted_network_data <- melt(
  filter(reg_network_data, run_id %in% relies_on_reg),
  variable.name = "reg_edge_type",
  value.name = "reg_edges_cnt",
  measure.vars=c("repressed_edges_cnt", "promoted_edges_cnt")
)

ggplot( melted_network_data, aes(x=NUM_SIGNAL_RESPONSES, y=reg_edges_cnt, color=reg_edg
  geom_boxplot() +
  xlab("Environmental Complexity") +
  ylab("# Edges") +
  scale_color_discrete(
    name="Edge type:",
    limits=c("repressed_edges_cnt", "promoted_edges_cnt"),
    labels=c("Repressing edges", "Promoting edges")
  ) +
  theme(
    legend.position="bottom",
    legend.text=element_text(size=9),
    legend.title=element_text(size=10),
    axis.title.x=element_text(size=12)
```

```
  ) +
  ggsave(
    paste0(working_directory, "imgs/repeated-signal-regulation-edge-types.png"),
    width=4,
    height=3
  )
```




```
for (env in env_complexities) {
  print(paste("Environment", env))
  print(paste0("  Median repressing edges: ", median(filter(melted_network_data, NUM_SIGNAL_RESPO
  print(paste0("  Median promoting edges: ", median(filter(melted_network_data, NUM_SIGNAL_RESPON
  wt <- wilcox.test(
    formula=reg_edges_cnt ~ reg_edge_type,
    data=filter(melted_network_data, NUM_SIGNAL_RESPONSES==env),
    exact=FALSE,
    conf.int=TRUE
  )
  print(wt)
}
```

```
## [1] "Environment 2"
## [1] "  Median repressing edges: 3"
## [1] "  Median promoting edges: 2"
##
##  Wilcoxon rank sum test with continuity correction
```

```
##
## data:  reg_edges_cnt by reg_edge_type
## W = 21990, p-value = 8.308e-05
## alternative hypothesis: true location shift is not equal to 0
## 95 percent confidence interval:
##  6.294052e-06 1.000039e+00
## sample estimates:
## difference in location
##              0.9999429
##
## [1] "Environment 4"
## [1] "  Median repressing edges: 6"
## [1] "  Median promoting edges: 3"
##
##  Wilcoxon rank sum test with continuity correction
##
## data:  reg_edges_cnt by reg_edge_type
## W = 30971, p-value < 2.2e-16
## alternative hypothesis: true location shift is not equal to 0
## 95 percent confidence interval:
##  2.999916 3.999984
## sample estimates:
## difference in location
##               3.000027
##
## [1] "Environment 8"
## [1] "  Median repressing edges: 9"
## [1] "  Median promoting edges: 3"
##
##  Wilcoxon rank sum test with continuity correction
##
## data:  reg_edges_cnt by reg_edge_type
## W = 34138, p-value < 2.2e-16
## alternative hypothesis: true location shift is not equal to 0
## 95 percent confidence interval:
##  5.000045 6.000012
## sample estimates:
## difference in location
##               5.999952
##
## [1] "Environment 16"
## [1] "  Median repressing edges: 16"
## [1] "  Median promoting edges: 4"
##
##  Wilcoxon rank sum test with continuity correction
##
```

```
## data:  reg_edges_cnt by reg_edge_type
## W = 4984, p-value < 2.2e-16
## alternative hypothesis: true location shift is not equal to 0
## 95 percent confidence interval:
##  11.00002 13.00001
## sample estimates:
## difference in location
##                12.00003
```

7.6.2.2 Knockout experiments

Do successful programs rely on:

- neither up- nor down-regulation?
- either up- or down-regulation interchangeably?
- only on down-regulation?
- only on up-regulation?

```
# Limit the genotypes we're looking at to just solutions from the 'both' and 'regulation' conditi
relies_on_reg_orgs <- filter(
  max_fit_org_data,
  solution=="1" & relies_on_regulation=="1"
)
```

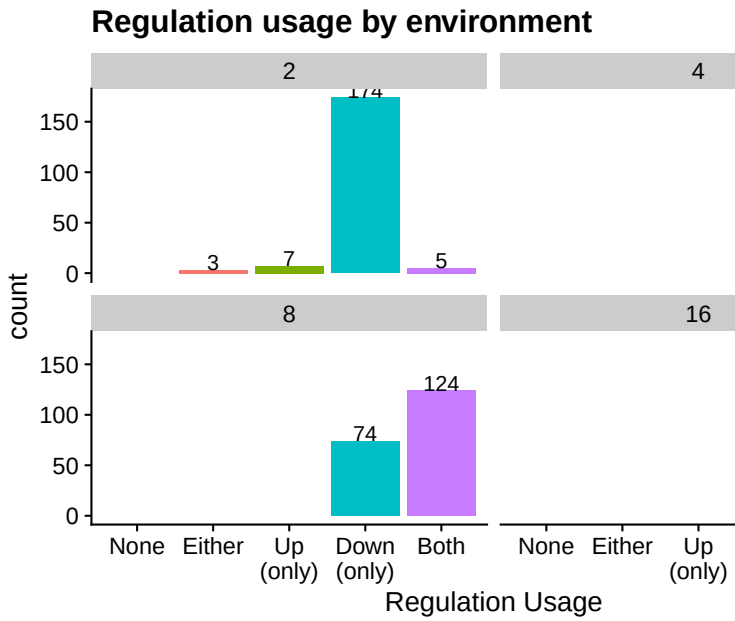
Note that there are 661 total programs represented in the graphs below.

```
# Data processing/clean up
get_reg_relies_on <- function(uses_down, uses_up, uses_reg) {
  if        (uses_down == "0" && uses_up == "0" && uses_reg == "0") {
    return("neither")
  } else if (uses_down == "0" && uses_up == "0" && uses_reg == "1") {
    return("either")
  } else if (uses_down == "0" && uses_up == "1") {
    return("up-regulation-only")
  } else if (uses_down == "1" && uses_up == "0") {
    return("down-regulation-only")
  } else if (uses_down == "1" && uses_up == "1") {
    return("up-and-down-regulation")
  } else {
    return("UNKNOWN")
  }
}

relies_on_reg_orgs$regulation_type_usage <- mapply(
  get_reg_relies_on,
  relies_on_reg_orgs$relies_on_down_reg,
  relies_on_reg_orgs$relies_on_up_reg,
  relies_on_reg_orgs$relies_on_regulation
```

```
)

relies_on_reg_orgs$regulation_type_usage <- factor(
  relies_on_reg_orgs$regulation_type_usage,
  levels=c(
    "neither",
    "either",
    "up-regulation-only",
    "down-regulation-only",
    "up-and-down-regulation"
  )
)
```

```
ggplot(relies_on_reg_orgs, aes(x=regulation_type_usage, fill=regulation_type_usage)) +
  geom_bar() +
  geom_text(
    stat="count",
    aes(label=..count..),
    position=position_dodge(0.9),
    vjust=0
  ) +
  scale_x_discrete(
    name="Regulation Usage",
    limits=c(
      "neither",
      "either",
      "up-regulation-only",
      "down-regulation-only",
      "up-and-down-regulation"
    ),
    labels=c(
      "None",
      "Either",
      "Up\n(only)",
      "Down\n(only)",
      "Both"
    )
  ) +
  facet_wrap(~NUM_SIGNAL_RESPONSES) +
  theme(legend.position="none") +
  ggtitle("Regulation usage by environment") +
  ggsave(
    paste0(working_directory, "imgs/rst-reg-usage-by-env.png"),
    width=8,
```

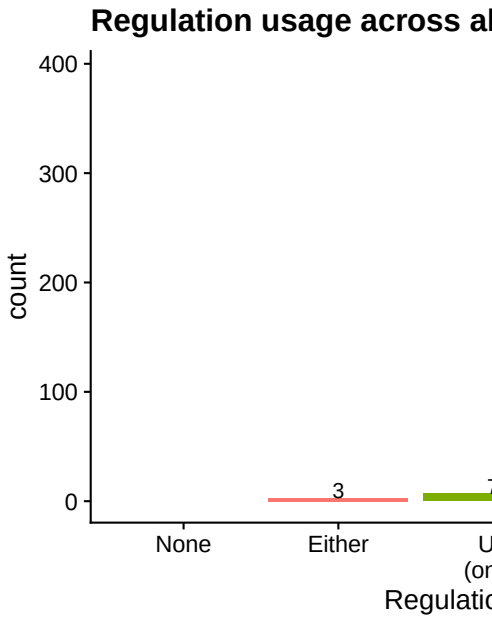
```
    height=6
  )
```



7.6.2.2.1 Regulation usage by environment

```
ggplot(relies_on_reg_orgs, aes(x=regulation_type_usage, fill=regulation_type_usage)) +
  geom_bar() +
  geom_text(
    stat="count",
    aes(label=..count..),
    position=position_dodge(0.9),
    vjust=0
  ) +
  scale_x_discrete(
    name="Regulation Usage",
    limits=c(
      "neither",
      "either",
      "up-regulation-only",
      "down-regulation-only",
      "up-and-down-regulation"
    ),
    labels=c(
      "None",
      "Either",
```

```
      "Up\n(only)",
      "Down\n(only)",
      "Both"
    )
  ) +
  theme(legend.position="none") +
  ggtitle("Regulation usage across all environments") +
  ggsave(
    paste0(working_directory, "imgs/rst-reg-usage-total.png"),
    width=8,
    height=6
  )
```



7.6.2.2.2 Regulation usage across all environments

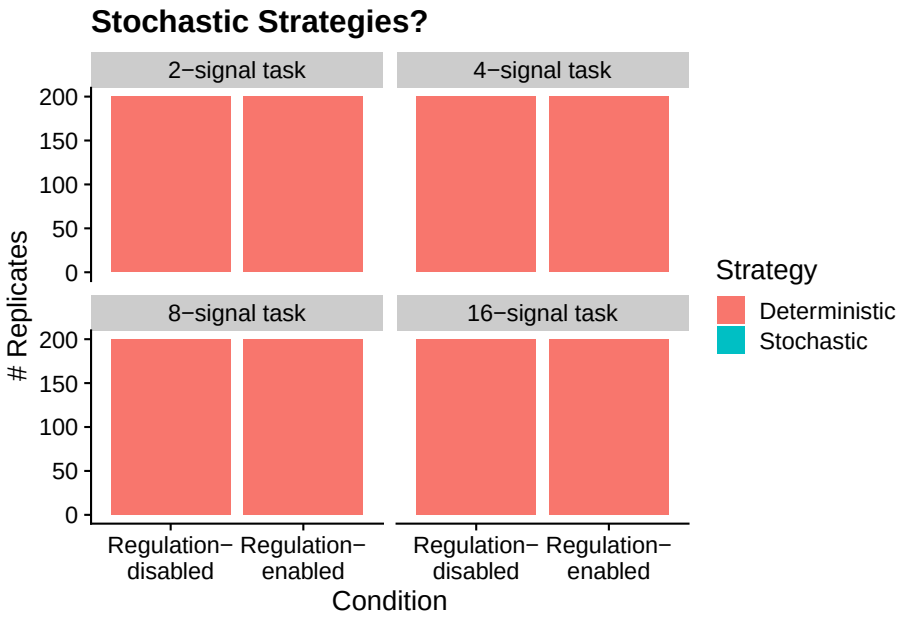
7.6.3 Are evolved programs relying on stochastic strategies?

To confirm that evolved programs are not relying on stochastic approaches to solve the repeated signal task, we tested the most fit individual from each replicate at the end of each run three times. If program's behavior was not identical across each of the three trials, we labeled is as using a stochastic strategy.

```
ggplot( max_fit_org_data, aes(x=condition, fill=stochastic)) +
  geom_bar() +
  ggtitle("Stochastic Strategies?") +
```

```
  ylab("# Replicates") +
  ylim(0, replicates) +
  scale_fill_discrete(
    name="Strategy",
    limits=c(0, 1),
    labels=c("Deterministic", "Stochastic")
  ) +
  scale_x_discrete(
    name="Condition",
    breaks=c("memory", "both"),
    labels=c("Regulation-\ndisabled", "Regulation-\nenabled")
  ) +
  facet_wrap(
    ~NUM_SIGNAL_RESPONSES,
    labeller=labeller(NUM_SIGNAL_RESPONSES=label_lu)
  )
```

```
## Warning: Continuous limits supplied to discrete scale.
## Did you mean `limits = factor(...)` or `scale_*_continuous()`?
```



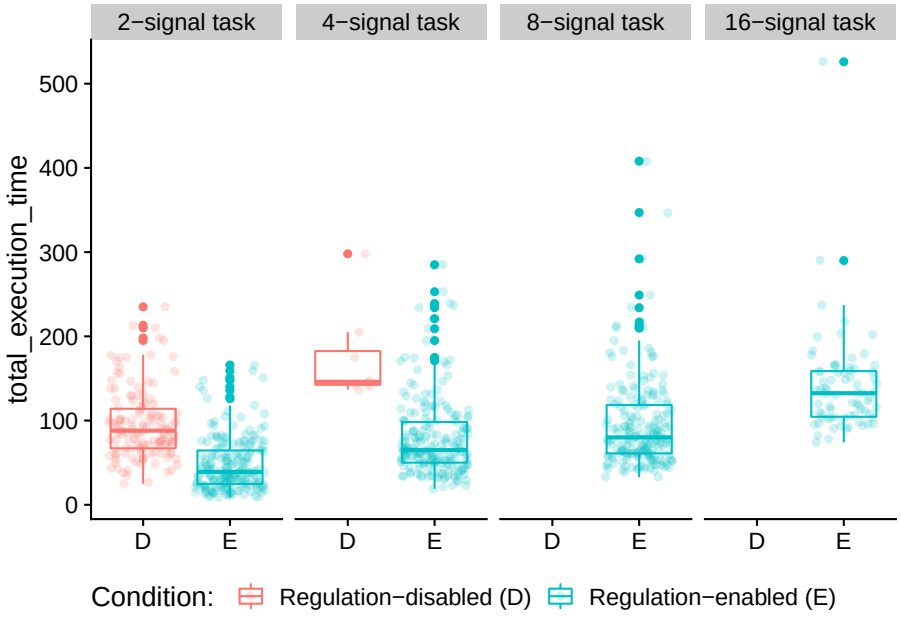
We see no evidence of evolved programs relying on stochastic strategies to solve the repeated signal task: all programs responded consistently across trials. Note, this is unsurprising, as we did not give programs access to instructions capable of generating random values and ensured that the version of SignalGP virtual hardware used in this work operated in a deterministic manner.

7.6.4 Program instruction execution traces

7.6.4.1 Execution time

How many time steps do evolved programs use to solve the repeated-signal task?

```
# only want solutions
solutions_inst_exec_data <- filter(inst_exec_data, SEED %in% sol_data$SEED)

ggplot( solutions_inst_exec_data, aes(x=condition, y=total_execution_time, color=condit
  geom_boxplot() +
  geom_jitter(alpha=0.2) +
  scale_color_discrete(
    name="Condition: ",
    breaks=c("memory", "both"),
    labels=c("Regulation-disabled (D)", "Regulation-enabled (E)")
  ) +
  scale_x_discrete(
    breaks=c("memory", "both"),
    labels=c("D", "E")
  ) +
  facet_wrap(
    ~ NUM_SIGNAL_RESPONSES,
    nrow=1,
    labeller=labeller(NUM_SIGNAL_RESPONSES=label_lu)
  ) +
  theme(
    legend.position="bottom",
    axis.title.x=element_blank()
  )
```



Two-signal task:

```
print(
  wilcox.test(
    formula=total_execution_time~condition,
    data=filter(solutions_inst_exec_data, NUM_SIGNAL_RESPONSES==2),
    exact=FALSE,
    conf.int=TRUE)
)
```

```
##
##  Wilcoxon rank sum test with continuity correction
##
## data:  total_execution_time by condition
## W = 23102, p-value < 2.2e-16
## alternative hypothesis: true location shift is not equal to 0
## 95 percent confidence interval:
##  38.00000 51.99997
## sample estimates:
## difference in location
##               44.99995
```

Four-signal task:

```
print(
  wilcox.test(
    formula=total_execution_time~condition,
```

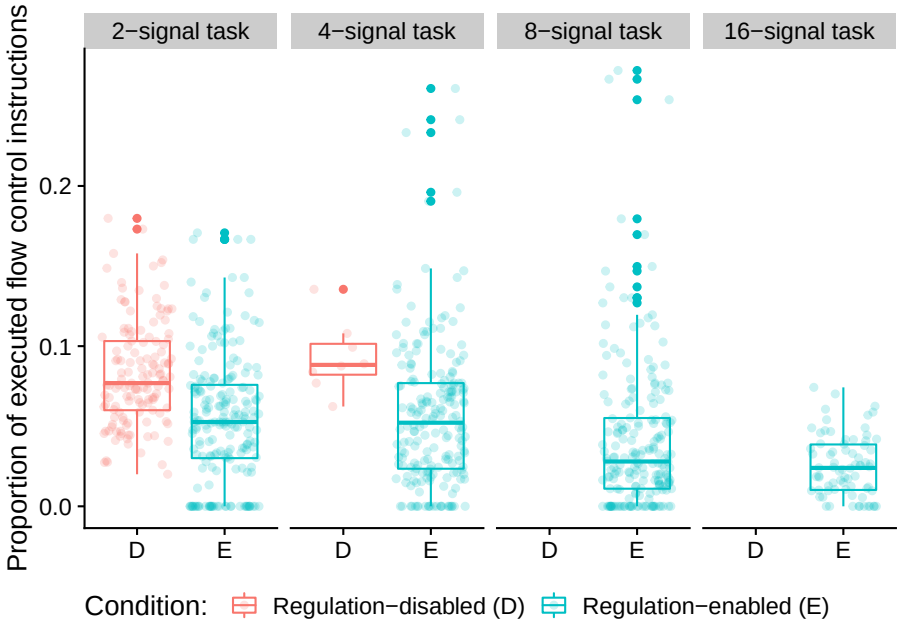
```
    data=filter(solutions_inst_exec_data, NUM_SIGNAL_RESPONSES==4),
    exact=FALSE,
    conf.int=TRUE)
)
```

```
##
##  Wilcoxon rank sum test with continuity correction
##
## data:  total_execution_time by condition
## W = 1494.5, p-value = 3.214e-05
## alternative hypothesis: true location shift is not equal to 0
## 95 percent confidence interval:
##   67.99998 112.00000
## sample estimates:
## difference in location
##               89.00002
```

7.6.4.2 Distribution of executed instruction types

Here, we look at the distribution of instruction types that programs execute during evaluation. For this work, we are primarily interested in the proportions of control flow instructions executed.

```
ggplot( solutions_inst_exec_data, aes(x=condition, y=control_flow_inst_prop, color=cond
  geom_boxplot() +
  geom_jitter(alpha=0.2) +
  scale_color_discrete(
    name="Condition: ",
    breaks=c("memory", "both"),
    labels=c("Regulation-disabled (D)", "Regulation-enabled (E)")
  ) +
  scale_x_discrete(
    breaks=c("memory", "both"),
    labels=c("D", "E")
  ) +
  ylab("Proportion of executed flow control instructions") +
  facet_wrap(
    ~ NUM_SIGNAL_RESPONSES,
    nrow=1,
    labeller=labeller(NUM_SIGNAL_RESPONSES=label_lu)
  ) +
  theme(
    legend.position="bottom",
    axis.title.x=element_blank()
  )
```



Two-signal task statistical comparison:

```
print(
  wilcox.test(
    formula=control_flow_inst_prop~condition,
    data=filter(solutions_inst_exec_data, NUM_SIGNAL_RESPONSES==2),
    exact=FALSE,
    conf.int=TRUE)
)
```

```
##
##  Wilcoxon rank sum test with continuity correction
##
## data:  control_flow_inst_prop by condition
## W = 19580, p-value = 2.118e-11
## alternative hypothesis: true location shift is not equal to 0
## 95 percent confidence interval:
##  0.02022011 0.03692075
## sample estimates:
## difference in location
##             0.02817524
```

Four-signal task statistical comparison:

```
print(
  wilcox.test(
    formula=control_flow_inst_prop~condition,
```

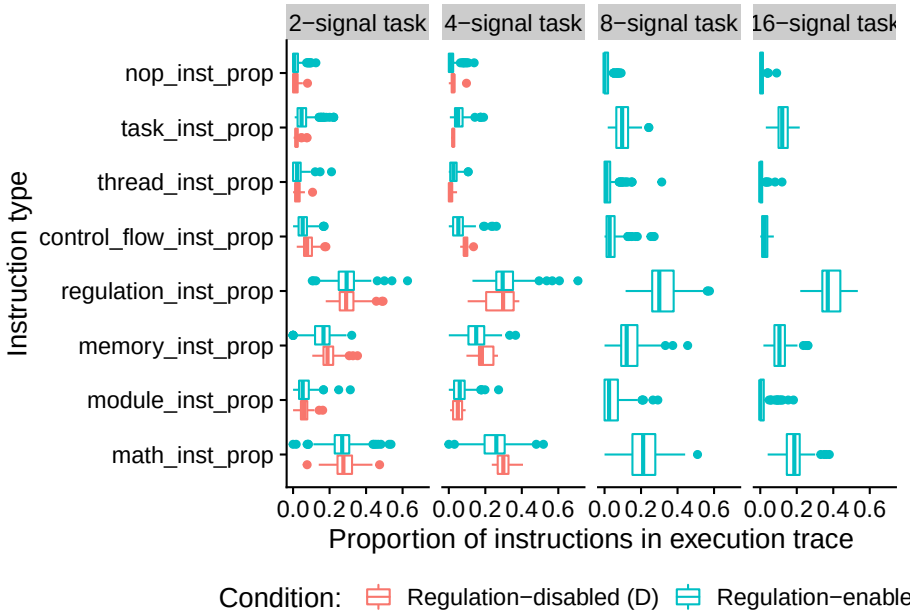
```
    data=filter(solutions_inst_exec_data, NUM_SIGNAL_RESPONSES==4),
    exact=FALSE,
    conf.int=TRUE)
)
```

```
##
##  Wilcoxon rank sum test with continuity correction
##
## data:  control_flow_inst_prop by condition
## W = 1292.5, p-value = 0.003185
## alternative hypothesis: true location shift is not equal to 0
## 95 percent confidence interval:
##  0.01577416 0.06398521
## sample estimates:
## difference in location
##             0.04051067
```

In case you're curious, here's all categories of instructions:

```
melted <- melt(
  solutions_inst_exec_data,
  variable.name = "inst_type",
  value.name = "inst_type_prop",
  measure.vars=c(
    "math_inst_prop",
    "module_inst_prop",
    "memory_inst_prop",
    "regulation_inst_prop",
    "control_flow_inst_prop",
    "thread_inst_prop",
    "task_inst_prop",
    "nop_inst_prop"
  )
)

ggplot( melted, aes(x=inst_type, y=inst_type_prop, color=condition) ) +
  geom_boxplot() +
  scale_color_discrete(
    name="Condition: ",
    breaks=c("memory", "both"),
    labels=c("Regulation-disabled (D)", "Regulation-enabled (E)")
  ) +
  xlab("Instruction type") +
  ylab("Proportion of instructions in execution trace") +
  facet_wrap(
    ~NUM_SIGNAL_RESPONSES,
```

```
    nrow=1,
    labeller=labeller(NUM_SIGNAL_RESPONSES=label_lu)
  ) +
  coord_flip() +
  theme(
    legend.position="bottom"
  )
```



7.7 Case study: visualizing regulation in an evolved program

Let's take a closer look at the behavioral/regulatory profile of a representative program that solves the four-signal version of the repeated signal task.

```
trace_id <- 20203
```

Specifically, we'll be looking at the solution evolved in run id 2.0203×10^4.

7.7.1 Data wrangling

```
case_study_info <- read.csv(
  paste0(working_directory, "data/max_fit_orgs_noprogram.csv"),
  na.strings="NONE"
)
```

```
case_study_info <- filter(
  case_study_info,
  SEED==trace_id
)

# Extract relevant information about solution of interest.
num_envs <- case_study_info$NUM_SIGNAL_RESPONSES
score <- case_study_info$score
is_sol <- case_study_info$solution
num_modules <- case_study_info$num_modules

# Load trace file associated with this solution.
trace_file <- paste0(working_directory, "data/reg-traces/trace-reg_update-10000_run-id
trace_data <- read.csv(trace_file, na.strings="NONE")
trace_data$similarity_score <- 1 - trace_data$match_score

# Data cleanup/summarizing
trace_data$triggered <- (trace_data$env_signal_closest_match == trace_data$module_id) &
trace_data$is_running <- trace_data$is_running > 0 | trace_data$triggered | trace_data$

# Extract which modules responded and when
response_time_steps <- levels(factor(filter(trace_data, is_cur_responding_function=="1"
responses_by_env_update <- list()
for (t in response_time_steps) {
  env_update <- levels(factor(filter(trace_data, time_step==t)$env_cycle))
  if (env_update %in% names(responses_by_env_update)) {
    if (as.integer(t) > as.integer(responses_by_env_update[env_update])) {
      responses_by_env_update[env_update] = t
    }
  } else {
    responses_by_env_update[env_update] = t
  }
}

# Build a list of modules that were triggered & those that responded to a signal
triggered_ids <- levels(factor(filter(trace_data, triggered==TRUE)$module_id))
response_ids <- levels(factor(filter(trace_data, is_cur_responding_function=="1")$modul

trace_data$is_ever_active <-
  trace_data$is_ever_active=="1" |
  trace_data$is_running |
  trace_data$module_id %in% triggered_ids |
  trace_data$module_id %in% response_ids

trace_data$is_cur_responding_function <-
```

```
  trace_data$is_cur_responding_function=="1" &
  trace_data$time_step %in% responses_by_env_update

# function to categorize each regulatory state as promoted, neutral, or repressed
# remember, the regulatory states in our data file operate with tag DISTANCE in mind
# as opposed to tag similarity, so: promotion => reg < 0, repression => reg > 0
categorize_reg_state <- function(reg_state) {
  if (reg_state == 0) {
    return("neutral")
  } else if (reg_state < 0) {
    return("promoted")
  } else if (reg_state > 0) {
    return("repressed")
  } else {
    return("unknown")
  }
}
trace_data$regulator_state_simplified <- mapply(
  categorize_reg_state,
  trace_data$regulator_state
)

# Omit all in-active rows
# Extract only rows that correspond with modules that were active during evaluation.
active_data <- filter(trace_data, is_ever_active==TRUE)

# Do some work to have module ids appear in a nice order along axis.
active_module_ids <- levels(factor(active_data$module_id))
active_module_ids <- as.integer(active_module_ids)
module_id_map <- as.data.frame(active_module_ids)
module_id_map$order <- order(module_id_map$active_module_ids) - 1

get_module_x_pos <- function(module_id) {
  return(filter(module_id_map, active_module_ids==module_id)$order)
}

active_data$mod_id_x_pos <- mapply(get_module_x_pos, active_data$module_id)
```

7.7.2 Function regulation over time

First, let's omit all non-active funcitons.

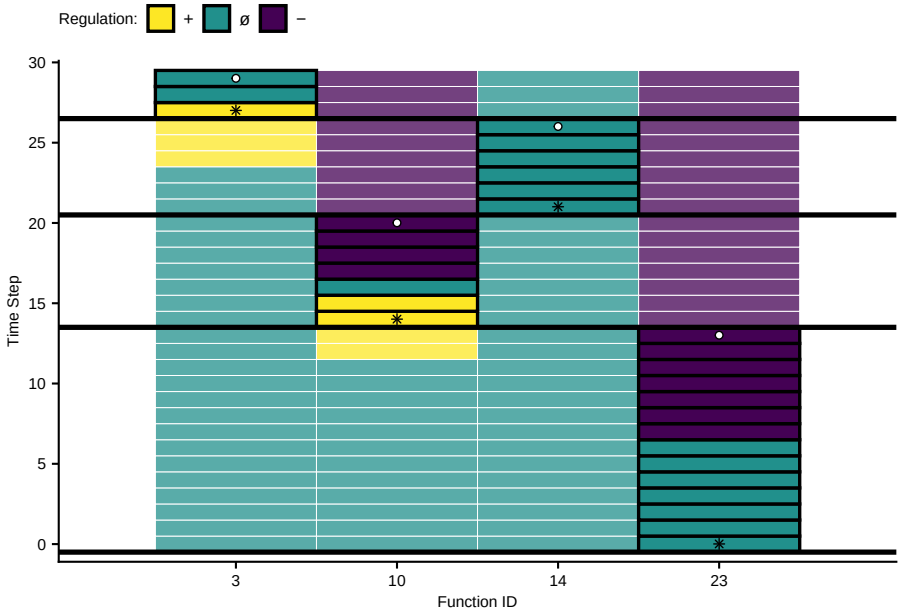
Vertical orientation:

```
out_name <- paste0(
  working_directory,
```

```
  "imgs/case-study-trace-id-",
   trace_id,
   "-regulator-state-vertical.pdf"
)

ggplot(
    active_data,
    aes(x=mod_id_x_pos, y=time_step, fill=regulator_state_simplified)
  ) +
  scale_fill_viridis(
    name="Regulation:",
    limits=c(
      "promoted",
      "neutral",
      "repressed"
    ),
    labels=c(
      "+",
      "\u00F8",
      "-"
    ),
    discrete=TRUE,
    direction=-1
  ) +
  scale_x_discrete(
    name="Function ID",
    limits=seq(0, length(active_module_ids)-1, 1),
    labels=active_module_ids
  ) +
  scale_y_discrete(
    name="Time Step",
    limits=seq(0, 30, 5)
  ) +
  # Background tile color
  geom_tile(
    color="white",
    size=0.2,
    width=1,
    height=1,
    alpha=0.75
  ) +
  # Highlight actively running functions
  geom_tile(
    data=filter(active_data, is_running==TRUE | triggered==TRUE),
    color="black",
```

```
    size=0.8,
    width=1,
    height=1
  ) +
  # Environment delimiters
  geom_hline(
    yintercept=filter(active_data, cpu_step==0)$time_step - 0.5,
    size=1.25,
    color="black"
  ) +
  # Draw points on triggered modules
  geom_point(
    data=filter(active_data, triggered==TRUE),
    shape=8,
    colour="black",
    fill="white",
    stroke=0.5,
    size=1.5,
    position=position_nudge(x = 0, y = 0.01)
  ) +
  geom_point(
    data=filter(active_data, is_cur_responding_function==TRUE),
    shape=21,
    colour="black",
    fill="white",
    stroke=0.5,
    size=1.5,
    position=position_nudge(x = 0, y = 0.01)
  ) +
  theme(
    legend.position = "top",
    legend.text = element_text(size=9),
    legend.title=element_text(size=8),
    axis.text.y = element_text(size=8),
    axis.title.y = element_text(size=8),
    axis.text.x = element_text(size=8),
    axis.title.x = element_text(size=8),
    plot.title = element_text(hjust = 0.5)
  ) +
  ggsave(
    out_name,
    height=3.5,
    width=2.25
  )
```

```
## Warning: Continuous limits supplied to discrete scale.
## Did you mean `limits = factor(...)` or `scale_*_continuous()`?

## Warning: Continuous limits supplied to discrete scale.
## Did you mean `limits = factor(...)` or `scale_*_continuous()`?
```



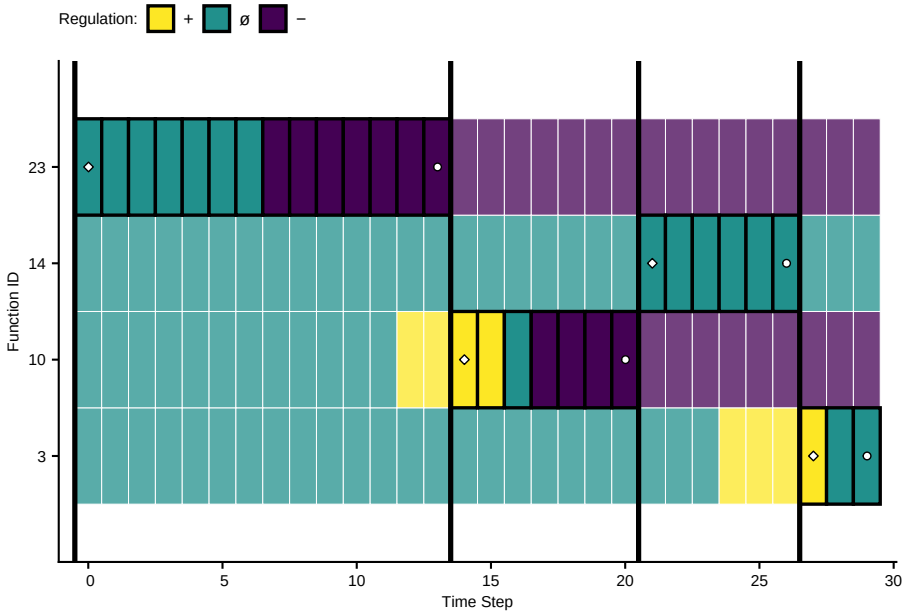
Horizontal orientation:

```
out_name <- paste0(working_directory, "imgs/case-study-trace-id-",trace_id,"-regulator-

ggplot(active_data, aes(x=mod_id_x_pos, y=time_step, fill=regulator_state_simplified))
  scale_fill_viridis(
    name="Regulation:",
    limits=c(
      "promoted",
      "neutral",
      "repressed"
    ),
    labels=c(
      "+",
      "\u00F8",
      "-"
    ),
    discrete=TRUE,
    direction=-1
  ) +
```

```
  scale_x_discrete(
    name="Function ID",
    limits=seq(0, length(active_module_ids)-1, 1),
    labels=active_module_ids
  ) +
  scale_y_discrete(
    name="Time Step",
    limits=seq(0, 30, 5)
  ) +
  # Background tile color
  geom_tile(
    color="white",
    size=0.2,
    width=1,
    height=1,
    alpha=0.75
  ) +
  # Highlight actively running functions
  geom_tile(
    data=filter(active_data, is_running==TRUE | triggered==TRUE),
    color="black",
    size=0.8,
    width=1,
    height=1
  ) +
  # Environment delimiters
  geom_hline(
    yintercept=filter(active_data, cpu_step==0)$time_step - 0.5,
    size=1.25,
    color="black"
  ) +
  # Draw points on triggered modules
  geom_point(
    data=filter(active_data, triggered==TRUE),
    shape=23,
    colour="black",
    fill="white",
    stroke=0.5,
    size=1.5,
    position=position_nudge(x = 0, y = 0.01)
  ) +
  geom_point(
    data=filter(active_data, is_cur_responding_function==TRUE),
    shape=21,
    colour="black",
```

```
    fill="white",
    stroke=0.5,
    size=1.5,
    position=position_nudge(x = 0, y = 0.01)
  ) +
  theme(
    legend.position = "top",
    legend.text = element_text(size=9),
    legend.title=element_text(size=8),
    axis.text.y = element_text(size=8),
    axis.title.y = element_text(size=8),
    axis.text.x = element_text(size=8),
    axis.title.x = element_text(size=8),
    plot.title = element_text(hjust = 0.5)
  ) +
  coord_flip() +
  ggsave(out_name, height=2.25, width=4)
```

```
## Warning: Continuous limits supplied to discrete scale.
## Did you mean `limits = factor(...)` or `scale_*_continuous()`?

## Warning: Continuous limits supplied to discrete scale.
## Did you mean `limits = factor(...)` or `scale_*_continuous()`?
```



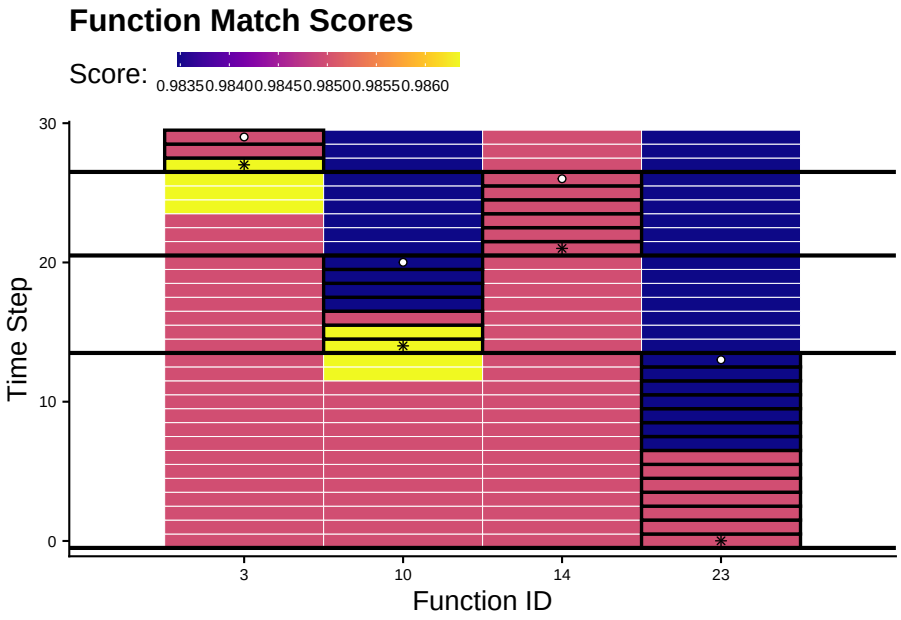
7.7.3 Environmental signal tag-match score over time

Again, we'll omit unexecuted functions.

```
out_name <- paste0(working_directory, "imgs/case-study-trace-id-", trace_id, "-similarity-score.p

ggplot(active_data, aes(x=mod_id_x_pos, y=time_step, fill=similarity_score)) +
  scale_fill_viridis(
    option="plasma",
    name="Score:  "
  ) +
  scale_x_discrete(
    name="Function ID",
    limits=seq(0, length(active_module_ids)-1, 1),
    labels=active_module_ids
  ) +
  scale_y_discrete(
    name="Time Step",
    limits=seq(0, 30, 10)
  ) +
  # Background
  geom_tile(
    color="white",
    size=0.2,
    width=1,
    height=1
  ) +
  # Module is-running highlights
  geom_tile(
    data=filter(active_data, is_running==TRUE | triggered==TRUE),
    color="black",
    width=1,
    height=1,
    size=0.8
  ) +
  # Environment delimiters
  geom_hline(
    yintercept=filter(active_data, cpu_step==0)$time_step-0.5,
    size=1
  ) +
  # Draw points on triggered modules
  geom_point(
    data=filter(active_data, triggered==TRUE),
    shape=8,
    colour="black",
    fill="white",
```

```
    stroke=0.5,
    size=1.5,
    position=position_nudge(x = 0, y = 0.01)
  ) +
  geom_point(
    data=filter(active_data, is_cur_responding_function==TRUE),
    shape=21,
    colour="black",
    fill="white",
    stroke=0.5,
    size=1.5,
    position=position_nudge(x = 0, y = 0.01)
  ) +
  theme(
    legend.position = "top",
    legend.text = element_text(size=8),
    axis.text.y = element_text(size=8),
    axis.text.x = element_text(size=8)
  ) +
  guides(fill = guide_colourbar(barwidth = 10, barheight = 0.5)) +
  ggtitle("Function Match Scores") +
  ggsave(out_name, height=3, width=4)
```

```
## Warning: Continuous limits supplied to discrete scale.
## Did you mean `limits = factor(...)` or `scale_*_continuous()`?

## Warning: Continuous limits supplied to discrete scale.
## Did you mean `limits = factor(...)` or `scale_*_continuous()`?
```



7.7.4 Evolved regulatory network

We use the igraph package to draw this program's gene regulatory network.

```
# Networks!
graph_nodes_loc <- paste0(working_directory, "data/igraphs/reg_graph_id-", trace_id, "_nodes.csv"
graph_edges_loc <- paste0(working_directory, "data/igraphs/reg_graph_id-", trace_id, "_edges.csv"
graph_nodes_data <- read.csv(graph_nodes_loc, na.strings="NONE")
```

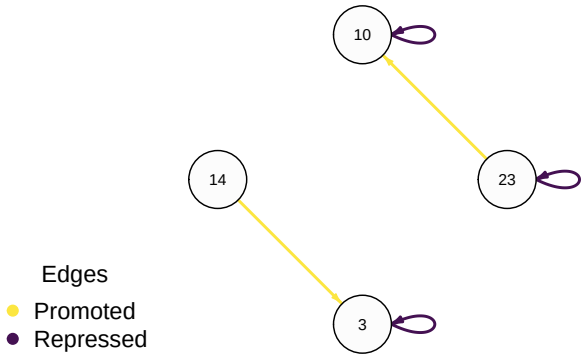
```
## Warning in read.table(file = file, header = header, sep = sep, quote = quote, :
## incomplete final line found by readTableHeader on 'experiments/2020-11-25-rep-
## sig/analysis/data/igraphs/reg_graph_id-20203_nodes.csv'
```

```
graph_edges_data <- read.csv(graph_edges_loc, na.strings="NONE")

network <- graph_from_data_frame(
  d=graph_edges_data,
  vertices=graph_nodes_data,
  directed=TRUE
)

# Setup edge styling
E(network)$color[E(network)$type == "promote"] <- "#FCE640"
E(network)$lty[E(network)$type == "promote"] <- 1
E(network)$color[E(network)$type == "repress"] <- "#441152"
E(network)$lty[E(network)$type == "repress"] <- 1
```

```
network_out_name <- paste0(working_directory, "imgs/case-study-id-", trace_id, "-networ

draw_network <- function(net, write_out, out_name) {
  if (write_out) {
    svg(out_name, width=4,height=1.5)
    # bottom, left, top, right
    par(mar=c(0.2,0,1,0.5))
  }
  plot(
    net,
    edge.arrow.size=0.4,
    edge.arrow.width=0.75,
    edge.width=2,
    vertex.size=40,
    vertex.label.cex=0.65,
    curved=TRUE,
    vertex.color="grey99",
    vertex.label.color="black",
    vertex.label.family="sans",
    layout=layout.circle(net)
  )
  legend(
    x = "bottomleft",      ## position, also takes x,y coordinates
    legend = c("Promoted", "Repressed"),
    pch = 19,              ## legend symbols see ?points
    col = c("#FCE640", "#441152"),
    bty = "n",
    border="black",
    xpd=TRUE,
    title = "Edges"
  )
  if (write_out) {
    dev.flush()
    dev.off()
  }
}

draw_network(network, TRUE, network_out_name)
```

```
## pdf
##   2
```

```
draw_network(network, FALSE, "")
```



7.8 All regulation traces

We generated regulation traces for every replicate: https://osf.io/crdqf/.

Chapter 8

Contextual-signal problem analysis

Here, we give an overview of the contextual-signal diagnostic problem, and we provide our data analyses for related experiments. All of our source code for statistical analyses and data visualizations is embedded in this document. The raw data can be found on the OSF project associated with this work (Lalejini et al., 2020).

Please file an issue or make a pull request on github to report any mistakes, ask questions, request more explanation, et cetera.

8.1 Overview

```
# Experimental parameters referenced in-text all in one convenient place.
time_steps <- 128
replicates <- 200
population_size <- 1000
generations <- 10000

# Settings for statistical analyses.
alpha <- 0.05

# Relative location of data.
working_directory <- "experiments/2020-11-27-context-sig/analysis/" # << For bookdown
# working_directory <- "./"                                         # << For local analysis
```

In the contextual-signal problem, programs must respond appropriately to a sequence of two input signals where the first, ‘’contextual‘‘, signal dictates how a program should respond to each possible second,’‘response”, signal. In this

work, there are a total of four possible input signals and four possible output responses. Programs output these responses by executing one of four response instructions.

The dataframe below gives the correct output for each combination of input signals.

```
testcases <- read.csv(paste0(working_directory, "../hpcc/examples_S4.csv"))
print(testcases)
```

```
##          input output  type
## 1  OP:S0;OP:S0      0 S0;S0
## 2  OP:S0;OP:S1      1 S0;S1
## 3  OP:S0;OP:S2      2 S0;S2
## 4  OP:S0;OP:S3      3 S0;S3
## 5  OP:S1;OP:S0      1 S1;S0
## 6  OP:S1;OP:S1      2 S1;S1
## 7  OP:S1;OP:S2      3 S1;S2
## 8  OP:S1;OP:S3      0 S1;S3
## 9  OP:S2;OP:S0      2 S2;S0
## 10 OP:S2;OP:S1      3 S2;S1
## 11 OP:S2;OP:S2      0 S2;S2
## 12 OP:S2;OP:S3      1 S2;S3
## 13 OP:S3;OP:S0      3 S3;S0
## 14 OP:S3;OP:S1      0 S3;S1
## 15 OP:S3;OP:S2      1 S3;S2
## 16 OP:S3;OP:S3      2 S3;S3
```

8.2 Analysis Dependencies

Load all required R libraries.

```
library(ggplot2)
library(tidyverse)
library(cowplot)
library(viridis)
library(reshape2)
library(igraph)
source("https://gist.githubusercontent.com/benmarwick/2a1bb0133ff568cbe28d/raw/fb53bd9
```

These analyses were conducted in the following computing environment:

```
print(version)
```

```
##                _
## platform       x86_64-pc-linux-gnu
## arch           x86_64
## os             linux-gnu
```

```
## system         x86_64, linux-gnu
## status
## major          4
## minor          0.2
## year           2020
## month          06
## day            22
## svn rev        78730
## language       R
## version.string R version 4.0.2 (2020-06-22)
## nickname       Taking Off Again
```

8.3 Setup

Load data, initial data cleanup, configure some global settings.

```
####### Load max fit program data #######
data_loc <- paste0(working_directory, "data/max_fit_orgs.csv")
data <- read.csv(data_loc, na.strings="NONE")

# Specify factors (not all of these matter for this set of runs).
data$matchbin_thresh <- factor(
  data$matchbin_thresh,
  levels=c(0, 25, 50, 75)
)

data$TAG_LEN <- factor(
  data$TAG_LEN,
  levels=c(32, 64, 128, 256)
)

data$task <- factor(
  data$task,
  levels=c("S2", "S3", "S4")
)

# Filter down to only data we use in paper.
data <- filter(data, task=="S4")

# Define function to summarize regulation/memory configurations.
get_con <- function(reg, mem) {
  if (reg == "0" && mem == "0") {
    return("none")
  } else if (reg == "0" && mem=="1") {
    return("memory")
```

```
  } else if (reg=="1" && mem=="0") {
    return("regulation")
  } else if (reg=="1" && mem=="1") {
    return("both")
  } else {
    return("UNKNOWN")
  }
}
# Specify experimental condition for each datum.
data$condition <- mapply(
  get_con,
  data$USE_FUNC_REGULATION,
  data$USE_GLOBAL_MEMORY
)

data$condition <- factor(
  data$condition,
  levels=c("regulation", "memory", "none", "both")
)

# Given knockout info, what strategy does a program use?
get_strategy <- function(use_reg, use_mem) {
  if (use_reg=="0" && use_mem=="0") {
    return("use neither")
  } else if (use_reg=="0" && use_mem=="1") {
    return("use memory")
  } else if (use_reg=="1" && use_mem=="0") {
    return("use regulation")
  } else if (use_reg=="1" && use_mem=="1") {
    return("use both")
  } else {
    return("UNKNOWN")
  }
}

# Specify experimental conditions (to make labeling easier).
data$strategy <- mapply(
  get_strategy,
  data$relies_on_regulation,
  data$relies_on_global_memory
)

data$strategy <- factor(
  data$strategy,
  levels=c(
```

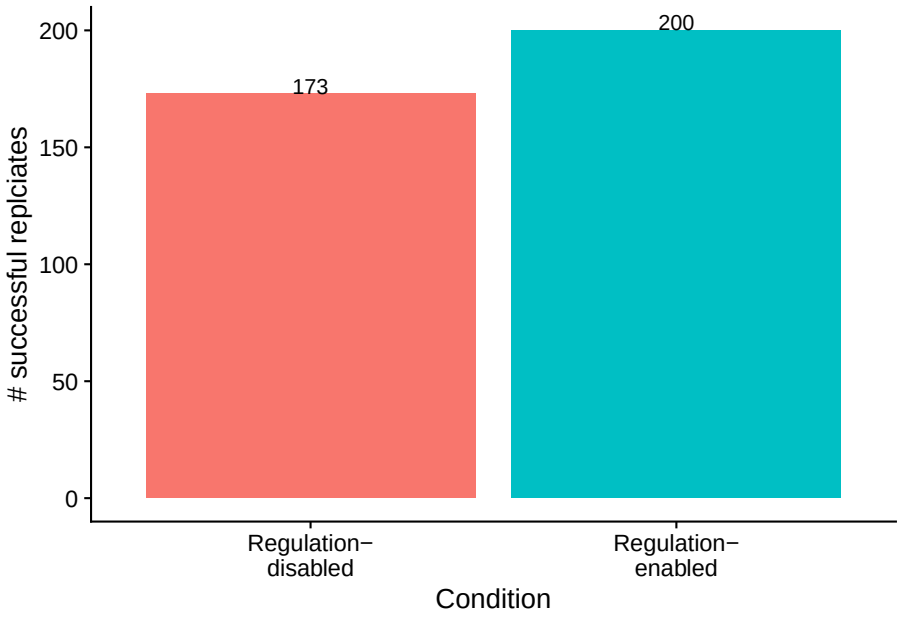
```
    "use regulation",
    "use memory",
    "use neither",
    "use both"
  )
)

# Filter data to include only replicates labeled as solutions
sol_data <- filter(data, solution=="1")

####### Load instruction execution data #######
inst_exec_data <- read.csv(paste0(working_directory, "data/exec_trace_summary.csv"), na.strings="

inst_exec_data$condition <- mapply(
  get_con,
  inst_exec_data$USE_FUNC_REGULATION,
  inst_exec_data$USE_GLOBAL_MEMORY
)

inst_exec_data$condition <- factor(
  inst_exec_data$condition,
  levels=c("regulation", "memory", "none", "both")
)

inst_exec_data$task <- factor(
  inst_exec_data$task,
  levels=c("S2", "S3", "S4")
)

####### Load network data #######
reg_network_data <- read.csv(paste0(working_directory, "data/reg_graphs_summary.csv"), na.strings
reg_network_data <- filter(reg_network_data, run_id %in% data$SEED)

get_task <- function(seed) {
  return(filter(data, SEED==seed)$task)
}

reg_network_data$task <- mapply(
  get_task,
  reg_network_data$run_id
)

reg_network_data$task <- factor(reg_network_data$task)

####### misc #######
```

```
# Configure our default graphing theme
theme_set(theme_cowplot())
```

8.4 Problem-solving success

The number of successful replicates by condition:

```
# Graph the number of solutions evolved in each condition, faceted by environmental co
ggplot(filter(sol_data, task=="S4"), aes(x=condition, fill=condition)) +
  geom_bar() +
  geom_text(
    stat="count",
    mapping=aes(label=..count..),
    position=position_dodge(0.9),
    vjust=0
  ) +
  scale_x_discrete(
    name="Condition",
    breaks=c("memory","both"),
    labels=c("Regulation-\ndisabled", "Regulation-\nenabled")
  ) +
  ylab("# successful replciates") +
  theme(legend.position = "none") +
  ggsave(
    paste0(working_directory, "imgs/context-signal-solution-counts.pdf"),
    width=4,
    height=4
  )
```



Test for significance using Fisher's exact test.

```
# Extract successes/fails for each condition.
reg_disabled_success_cnt <- nrow(filter(sol_data, task=="S4" & solution=="1" & condition=="memory
reg_disabled_fail_cnt <- replicates - reg_disabled_success_cnt

reg_enabled_success_cnt <- nrow(filter(sol_data, task=="S4" & solution=="1" & condition=="both"))
reg_enabled_fail_cnt <- replicates - reg_enabled_success_cnt

# Regulation-disabled vs regulation-enabled
perf_table <- matrix(
  c(
    reg_enabled_success_cnt,
    reg_disabled_success_cnt,
    reg_enabled_fail_cnt,
    reg_disabled_fail_cnt
    ),
    nrow=2
)

rownames(perf_table) <- c("reg-enabled", "reg-disabled")
colnames(perf_table) <- c("success", "fail")

print(perf_table)
```

```
##              success fail
```

```
## reg-enabled      200    0
## reg-disabled     173   27
```

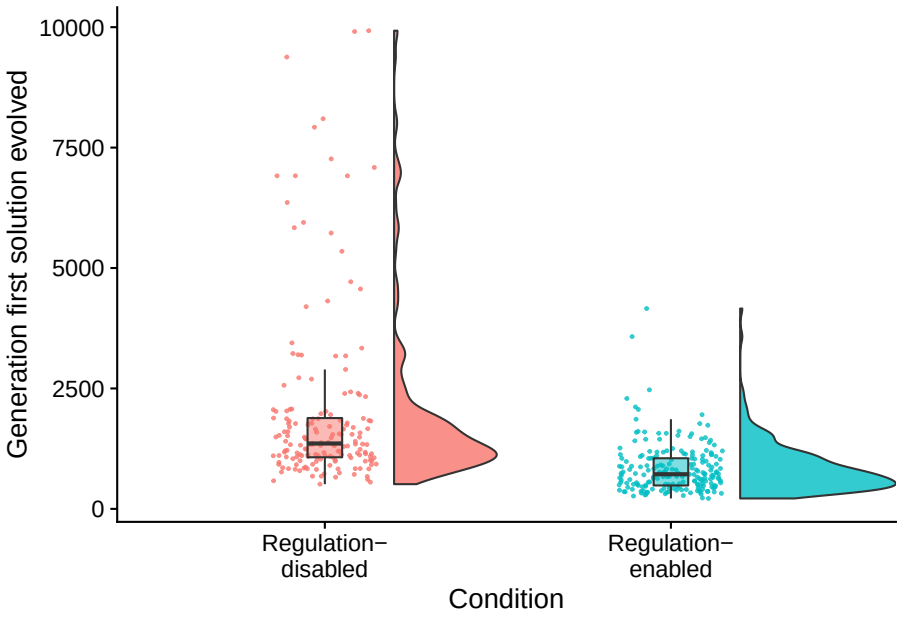
```
print(fisher.test(perf_table))
```

```
##
##  Fisher's Exact Test for Count Data
##
## data:  perf_table
## p-value = 5.818e-09
## alternative hypothesis: true odds ratio is not equal to 1
## 95 percent confidence interval:
##  7.714282      Inf
## sample estimates:
## odds ratio
##        Inf
```

8.5 How many generations elapse before solutions evolve?

```
ggplot( data = filter(sol_data, task=="S4"), aes(x=condition, y=update, fill=condition)
  geom_flat_violin(
    position=position_nudge(x = .2, y = 0),
    alpha=.8
  ) +
  geom_point(
    aes(y=update, color=condition),
    position = position_jitter(width = .15),
    size = .5,
    alpha = 0.8
  ) +
  geom_boxplot(
    width = .1,
    outlier.shape = NA,
    alpha = 0.5
  ) +
  scale_x_discrete(
    name="Condition",
    breaks=c("memory", "both"),
    labels=c("Regulation-\ndisabled", "Regulation-\nenabled")
  ) +
  scale_y_continuous(name="Generation first solution evolved") +
  guides(fill = FALSE) +
  guides(color = FALSE) +
  ggsave(
```

```
    paste0(working_directory, "imgs/context-signal-solve-time-cloud.png"),
    width=4,
    height=4
  )
```



Test for statistical difference between conditions using a Wilcoxon rank sum test.

```
print(wilcox.test(formula=update~condition, data=filter(sol_data, task=="S4"), exact=FALSE, conf.
```

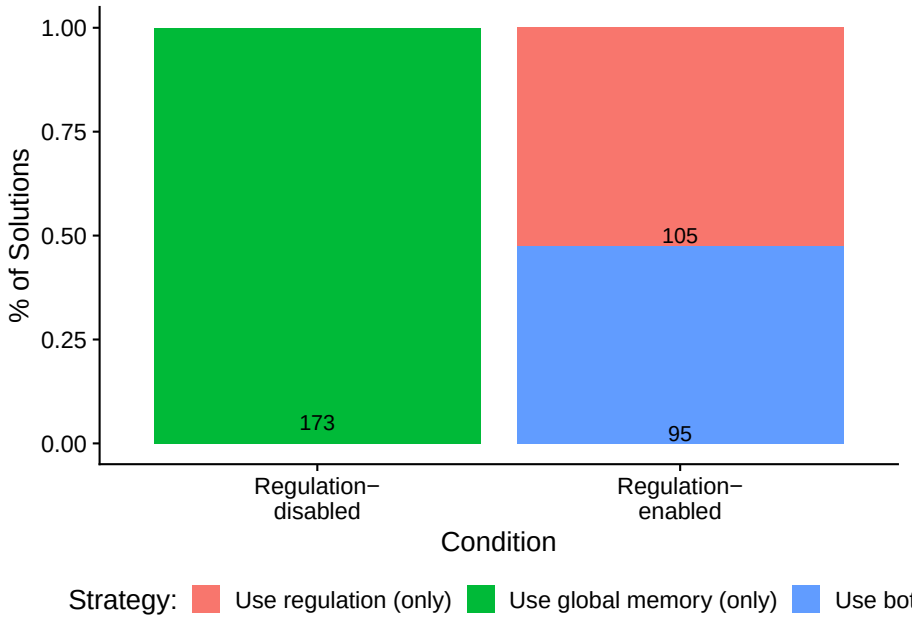
```
##
##  Wilcoxon rank sum test with continuity correction
##
## data:  update by condition
## W = 28950, p-value < 2.2e-16
## alternative hypothesis: true location shift is not equal to 0
## 95 percent confidence interval:
##  557.9999 764.0000
## sample estimates:
## difference in location
##                    657
```

8.6 Evolved strategies

8.6.1 What mechanisms do programs rely on to adjust responses to signals over time?

We used indpendent knockouts of tag-based genetic regulation and global memory buffer access to investigate the mechanisms underpinning successful programs.

```
ggplot( filter(sol_data, task=="S4"), mapping=aes(x=condition, fill=strategy) ) +
  geom_bar(
    position="fill",
    stat="count"
  ) +
  geom_text(
    stat='count',
    mapping=aes(label=..count..),
    position=position_fill(vjust=0.05)
  ) +
  ylab("% of Solutions") +
  scale_fill_discrete(
    name="Strategy:",
    breaks=c(
      "use regulation",
      "use memory",
      "use neither",
      "use both"
    ),
    labels=c(
      "Use regulation (only)",
      "Use global memory (only)",
      "Use neither",
      "Use both"
    )
  ) +
  scale_x_discrete(
    name="Condition",
    breaks=c("memory", "both"),
    labels=c("Regulation-\ndisabled", "Regulation-\nenabled")
  ) +
  theme(legend.position = "bottom")
```



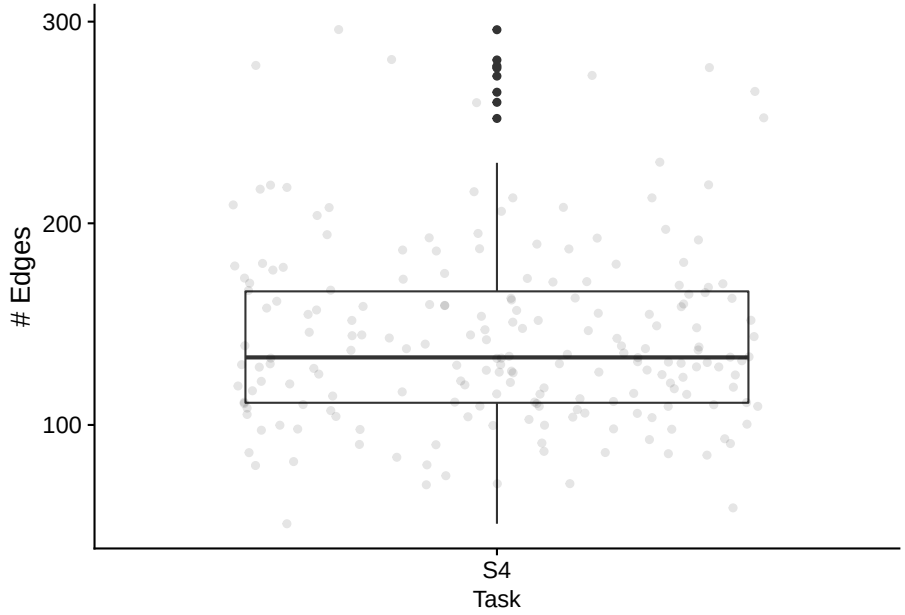
8.6.2 Gene regulatory networks

Looking only at successful programs that rely on regulation. At a glance, what do gene regulatory networks look like?

First, the total edges found in networks:

```
relies_on_reg <- filter(
  sol_data,
  relies_on_regulation=="1"
)$SEED

ggplot( filter(reg_network_data, run_id %in% relies_on_reg & task=="S4"), aes(x=task, y=edge_cnt)
  geom_boxplot() +
  geom_jitter(alpha=0.1) +
  xlab("Task") +
  ylab("# Edges") +
  theme(
    legend.position="bottom",
    legend.text=element_text(size=9),
    legend.title=element_text(size=10),
    axis.title.x=element_text(size=12)
  ) +
  ggsave(
    paste0(working_directory, "imgs/contextual-signal-regulation-edges.png"),
    width=4,
```

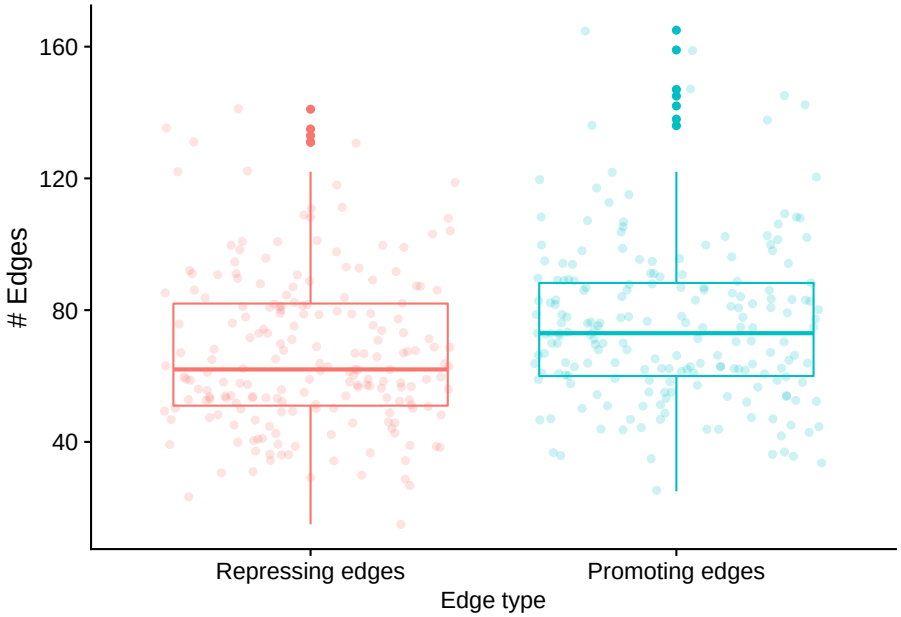
```
    height=3
  )
```



Next, let's look at edges by type.

```
# Process/cleanup the network data
melted_network_data <- melt(
  filter(reg_network_data,
         run_id %in% relies_on_reg
        ),
  variable.name = "reg_edge_type",
  value.name = "reg_edges_cnt",
  measure.vars=c("repressed_edges_cnt", "promoted_edges_cnt")
)

ggplot(filter(melted_network_data, task=="S4"), aes(x=reg_edge_type, y=reg_edges_cnt, c
  geom_boxplot() +
  geom_jitter(alpha=0.2) +
  xlab("Environmental Complexity") +
  ylab("# Edges") +
  scale_x_discrete(
    name="Edge type",
    limits=c("repressed_edges_cnt", "promoted_edges_cnt"),
    labels=c("Repressing edges", "Promoting edges")
  ) +
```

```
  theme(
    legend.position="none",
    legend.text=element_text(size=9),
    legend.title=element_text(size=10),
    axis.title.x=element_text(size=12)
  ) +
  ggsave(
    paste0(working_directory, "imgs/context-signal-regulation-edge-types.png"),
    width=4,
    height=3
  )
```



Test for a statistical difference between edge types using a wilcoxon rank sum test:

```
print(
  paste0(
    "Median # repressed edges: ",
    median(filter(melted_network_data, task=="S4" & reg_edge_type=="repressed_edges_cnt")$reg_edg
  )
)
```

```
## [1] "Median # repressed edges: 62"
```

```
print(
  paste0(
```

```
    "Median # promoting edges: ",
    median(filter(melted_network_data, task=="S4" & reg_edge_type=="promoted_edges_cnt"
  )
)
```

```
## [1] "Median # promoting edges: 73"
```

```
print(wilcox.test(formula=reg_edges_cnt ~ reg_edge_type, data=filter(melted_network_dat
```

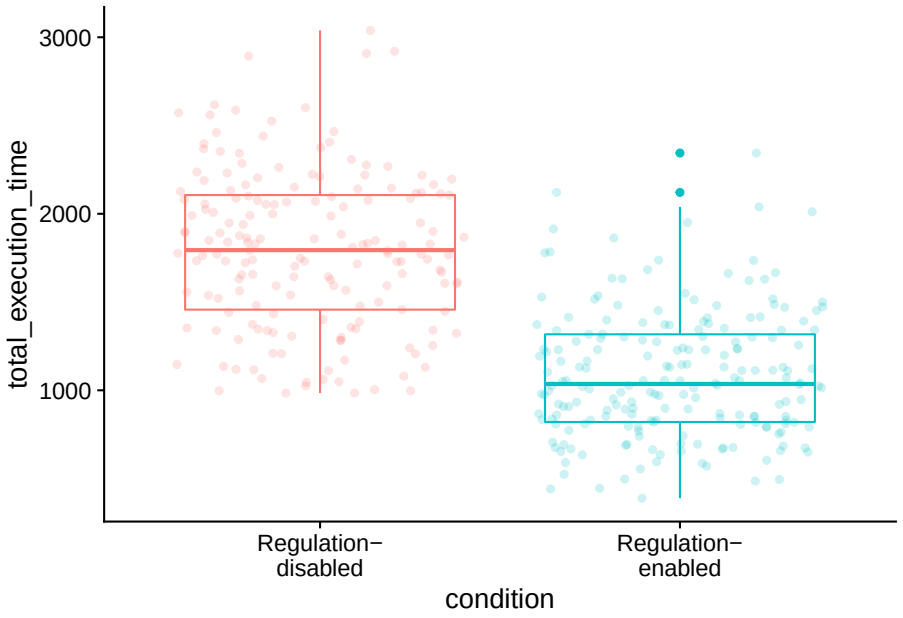
```
##
##  Wilcoxon rank sum test with continuity correction
##
## data:  reg_edges_cnt by reg_edge_type
## W = 15690, p-value = 0.0001927
## alternative hypothesis: true location shift is not equal to 0
## 95 percent confidence interval:
##  -13.000026  -4.000018
## sample estimates:
## difference in location
##              -8.000026
```

8.6.3 Program instruction execution traces

8.6.3.1 Execution time

How many time steps do evolved programs use to solve the contextual-signal task?

```
# only want solutions
solutions_inst_exec_data <- filter(inst_exec_data, SEED %in% sol_data$SEED & task=="S4"

ggplot( solutions_inst_exec_data, aes(x=condition, y=total_execution_time, color=condit
  geom_boxplot() +
  geom_jitter(alpha=0.2) +
  scale_x_discrete(
    breaks=c("memory", "both"),
    labels=c("Regulation-\ndisabled", "Regulation-\nenabled")
  ) +
  theme(
    legend.position="none"
  )
```



Test for significant difference between conditions using Wilcoxon rank sum test:

```
print(
  wilcox.test(
    formula=total_execution_time~condition,
    data=filter(solutions_inst_exec_data),
    exact=FALSE,
    conf.int=TRUE)
)
```

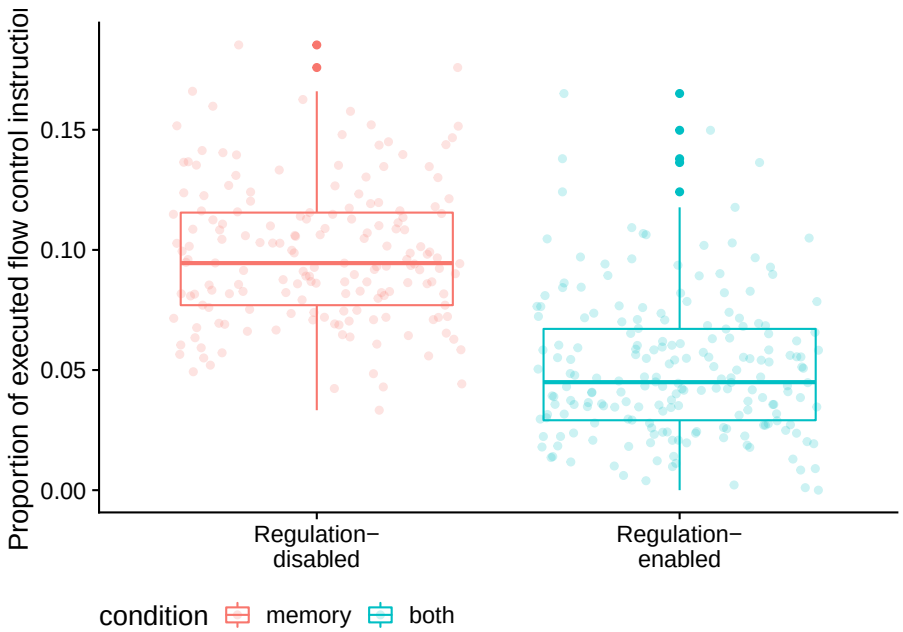
```
##
##  Wilcoxon rank sum test with continuity correction
##
## data:  total_execution_time by condition
## W = 30794, p-value < 2.2e-16
## alternative hypothesis: true location shift is not equal to 0
## 95 percent confidence interval:
##  634.0001 810.0000
## sample estimates:
## difference in location
##               722.8488
```

8.6.3.2 What types of instructions to successful programs execute?

Here, we look at the distribution of instruction types executed by successful programs. We're primarily interested in the proportion of control flow instructions,

so let's look at that first.

```
ggplot( solutions_inst_exec_data, aes(x=condition, y=control_flow_inst_prop, color=cond
  geom_boxplot() +
  geom_jitter(alpha=0.2) +
  scale_x_discrete(
    breaks=c("memory", "both"),
    labels=c("Regulation-\ndisabled", "Regulation-\nenabled")
  ) +
  ylab("Proportion of executed flow control instructions") +
  theme(
    legend.position="bottom",
    axis.title.x=element_blank()
  )
```



Test for significant difference between conditions using a Wilcoxon rank sum test:

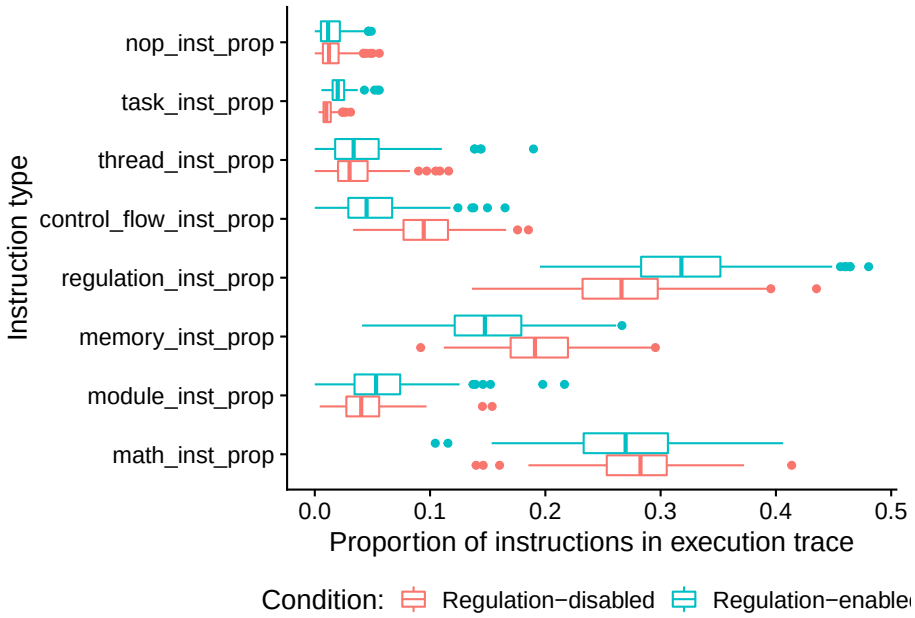
```
print(
  wilcox.test(
    formula=control_flow_inst_prop~condition,
    data=filter(solutions_inst_exec_data),
    exact=FALSE,
    conf.int=TRUE)
)
```

```
##
```

```
##  Wilcoxon rank sum test with continuity correction
##
## data:  control_flow_inst_prop by condition
## W = 30479, p-value < 2.2e-16
## alternative hypothesis: true location shift is not equal to 0
## 95 percent confidence interval:
##  0.04280319 0.05431491
## sample estimates:
## difference in location
##              0.04838185
```

In case you're curious, here's all categories of instructions:

```
melted <- melt(
  solutions_inst_exec_data,
  variable.name = "inst_type",
  value.name = "inst_type_prop",
  measure.vars=c(
    "math_inst_prop",
    "module_inst_prop",
    "memory_inst_prop",
    "regulation_inst_prop",
    "control_flow_inst_prop",
    "thread_inst_prop",
    "task_inst_prop",
    "nop_inst_prop"
  )
)

ggplot( melted, aes(x=inst_type, y=inst_type_prop, color=condition) ) +
  geom_boxplot() +
  scale_color_discrete(
    name="Condition:",
    breaks=c("memory", "both"),
    labels=c("Regulation-disabled", "Regulation-enabled")
  ) +
  xlab("Instruction type") +
  ylab("Proportion of instructions in execution trace") +
  coord_flip() +
  theme(legend.position="bottom")
```



8.7 Visualizing an evolved gene regulatory network

Let's take a closer look at a successful gene regulatory network.

```
trace_id <- 23997
```

Specifically, we'll be looking at the solution evolved in run id 2.3997×10^4 (arbitrarily selected).

8.7.1 Evolved regulatory network

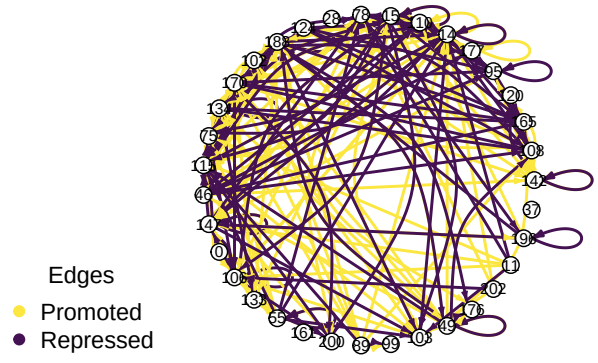
We use the igraph package to draw this program's gene regulatory network.

```
# Networks!
graph_nodes_loc <- paste0(working_directory, "data/igraphs/reg_graph_id-", trace_id, "_
graph_edges_loc <- paste0(working_directory, "data/igraphs/reg_graph_id-", trace_id, "_
graph_nodes_data <- read.csv(graph_nodes_loc, na.strings="NONE")
graph_edges_data <- read.csv(graph_edges_loc, na.strings="NONE")

network <- graph_from_data_frame(
  d=graph_edges_data,
  vertices=graph_nodes_data,
  directed=TRUE
)
```

```
# Setup edge styling
E(network)$color[E(network)$type == "promote"] <- "#FCE640"
E(network)$lty[E(network)$type == "promote"] <- 1
E(network)$color[E(network)$type == "repress"] <- "#441152"
E(network)$lty[E(network)$type == "repress"] <- 1

network_out_name <- paste0(working_directory, "imgs/case-study-id-", trace_id, "-network.svg")

draw_network <- function(net, write_out, out_name) {
  if (write_out) {
    svg(out_name, width=4,height=1.5)
    # bottom, left, top, right
    par(mar=c(0.2,0,1,0.5))
  }
  plot(
    net,
    edge.arrow.size=0.4,
    edge.arrow.width=0.75,
    edge.width=2,
    vertex.size=10,
    vertex.label.cex=0.65,
    curved=TRUE,
    vertex.color="grey99",
    vertex.label.color="black",
    vertex.label.family="sans",
    layout=layout.circle(net)
  )
  legend(
    x = "bottomleft",      ## position, also takes x,y coordinates
    legend = c("Promoted", "Repressed"),
    pch = 19,              ## legend symbols see ?points
    col = c("#FCE640", "#441152"),
    bty = "n",
    border="black",
    xpd=TRUE,
    title = "Edges"
  )
  if (write_out) {
    dev.flush()
    dev.off()
  }
}

draw_network(network, TRUE, network_out_name)
```

```
## pdf
##   2
```

```
draw_network(network, FALSE, "")
```



Chapter 9

Boolean calculator problem (prefix notation)

Here, we give an overview of the boolean logic calculator problem, and we provide our data analyses for related experiments. All of our source code for statistical analyses and data visualizations is embedded in this document. The raw data can be found on the OSF project associated with this work (Lalejini et al., 2020).

Please file an issue or make a pull request on github to report any mistakes, ask questions, request more explanation, et cetera.

9.1 Overview

```
# Experimental parameters referenced in-text all in one convenient place.
time_steps <- 128
replicates <- 200
population_size <- 1000
generations <- 10000

# Settings for statistical analyses.
alpha <- 0.05

# Relative location of data.
working_directory <- "experiments/2020-11-28-bool-calc-prefix/analysis/" # << For bookdown
# working_directory <- "./"                                              # << For local analysis
```

The Boolean logic calculator problem requires programs to implement a 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. 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 and is meant to represent different types of buttons that could be pressed on a physical calculator. Programs receive a sequence of input signals in *prefix notation*, receiving an operator signal followed by the appropriate number of numeric input signals (that each contain an operand to use in the computation). After receiving the requisite input signals, programs must output the correct result of the requested computation.

9.2 Analysis Dependencies

Load all required R libraries.

```
library(ggplot2)
library(tidyverse)
library(cowplot)
library(viridis)
library(reshape2)
library(igraph)
source("https://gist.githubusercontent.com/benmarwick/2a1bb0133ff568cbe28d/raw/fb53bd9
```

These analyses were conducted in the following computing environment:

```
print(version)
```

```
##                _
## platform       x86_64-pc-linux-gnu
## arch           x86_64
## os             linux-gnu
## system         x86_64, linux-gnu
## status
## major          4
## minor          0.2
## year           2020
## month          06
## day            22
## svn rev        78730
## language       R
## version.string R version 4.0.2 (2020-06-22)
## nickname       Taking Off Again
```

9.3 Setup

Load data, initial data cleanup, configure some global settings.

```
data_loc <- paste0(working_directory, "data/max_fit_orgs.csv")
data <- read.csv(data_loc, na.strings="NONE")

# Specify factors (not all of these matter for this set of runs).
data$matchbin_thresh <- factor(
  data$matchbin_thresh,
  levels=c(0, 25, 50, 75)
)

data$TAG_LEN <- factor(
  data$TAG_LEN,
  levels=c(32, 64, 128, 256)
)

data$notation <- factor(
  data$notation,
  levels=c("prefix", "postfix")
)

# Define function to summarize regulation/memory configurations.
get_con <- function(reg, mem) {
  if (reg == "0" && mem == "0") {
    return("none")
  } else if (reg == "0" && mem=="1") {
    return("memory")
  } else if (reg=="1" && mem=="0") {
    return("regulation")
  } else if (reg=="1" && mem=="1") {
    return("both")
  } else {
    return("UNKNOWN")
  }
}

# Specify experimental condition for each datum.
data$condition <- mapply(
  get_con,
  data$USE_FUNC_REGULATION,
  data$USE_GLOBAL_MEMORY
)

data$condition <- factor(
  data$condition,
  levels=c("regulation", "memory", "none", "both")
)
```

```r
# Given knockout info, what strategy does a program use?
get_strategy <- function(use_reg, use_mem) {
  if (use_reg=="0" && use_mem=="0") {
    return("use neither")
  } else if (use_reg=="0" && use_mem=="1") {
    return("use memory")
  } else if (use_reg=="1" && use_mem=="0") {
    return("use regulation")
  } else if (use_reg=="1" && use_mem=="1") {
    return("use both")
  } else {
    return("UNKNOWN")
  }
}

# Specify experimental conditions (to make labeling easier).
data$strategy <- mapply(
  get_strategy,
  data$relies_on_regulation,
  data$relies_on_global_memory
)

data$strategy <- factor(
  data$strategy,
  levels=c(
    "use regulation",
    "use memory",
    "use neither",
    "use both"
  )
)

# Filter data to include only replicates labeled as solutions
sol_data <- filter(data, solution=="1")

####### Load instruction execution data #######
inst_exec_data <- read.csv(paste0(working_directory, "data/exec_trace_summary.csv"), na

inst_exec_data$condition <- mapply(
  get_con,
  inst_exec_data$USE_FUNC_REGULATION,
  inst_exec_data$USE_GLOBAL_MEMORY
)

inst_exec_data$condition <- factor(
```

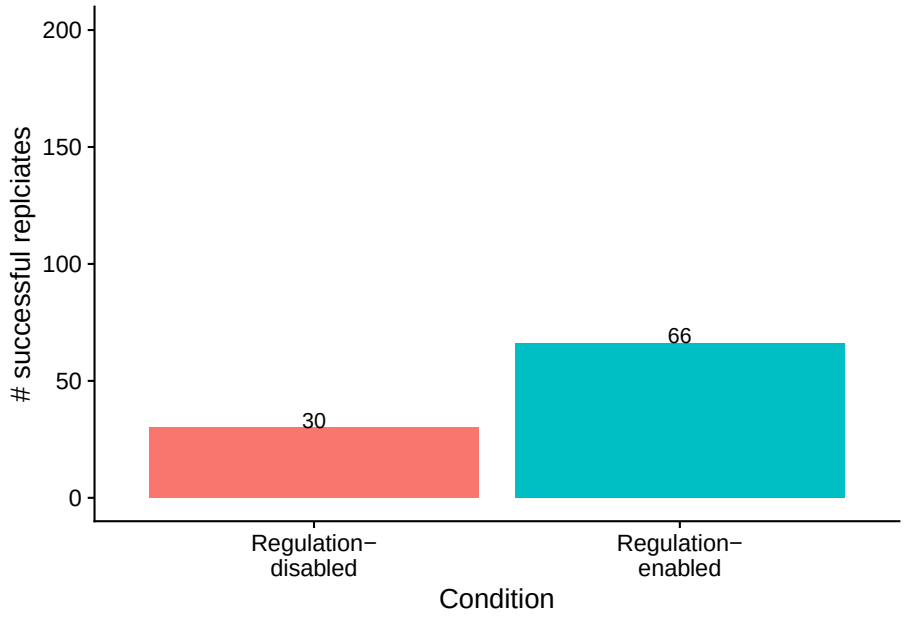
```
  inst_exec_data$condition,
  levels=c("regulation", "memory", "none", "both")
)

inst_exec_data$notation <- factor(
  inst_exec_data$notation,
  levels=c("prefix", "postfix")
)

####### Load network data #######
reg_network_data <- read.csv(paste0(working_directory, "data/reg_graphs_summary.csv"), na.strings
reg_network_data <- filter(reg_network_data, run_id %in% data$SEED)

get_notation <- function(seed) {
  return(filter(data, SEED==seed)$notation)
}

reg_network_data$notation <- mapply(
  get_notation,
  reg_network_data$run_id
)

reg_network_data$notation <- factor(
  reg_network_data$notation,
  levels=c("prefix", "postfix")
)

####### misc #######
# Configure our default graphing theme
theme_set(theme_cowplot())
```

9.4 Problem-solving success

The number of successful replicates by condition:

```
# Graph the number of solutions evolved in each condition, faceted by environmental complexity
ggplot(sol_data, aes(x=condition, fill=condition)) +
  geom_bar() +
  geom_text(
    stat="count",
    mapping=aes(label=..count..),
    position=position_dodge(0.9),
    vjust=0
  ) +
  scale_x_discrete(
```

```
    name="Condition",
    breaks=c("memory","both"),
    labels=c("Regulation-\ndisabled", "Regulation-\nenabled")
  ) +
  ylab("# successful replciates") +
  ylim(0, 200) +
  theme(legend.position = "none") +
  ggsave(
    paste0(working_directory, "imgs/boolean-calc-prefix-solution-counts.pdf"),
    width=4,
    height=4
  )
```



Test for significance using Fisher's exact test.

```
# Extract successes/fails for each condition.
reg_disabled_success_cnt <- nrow(filter(sol_data, solution=="1" & condition=="memory"))
reg_disabled_fail_cnt <- replicates - reg_disabled_success_cnt

reg_enabled_success_cnt <- nrow(filter(sol_data, solution=="1" & condition=="both"))
reg_enabled_fail_cnt <- replicates - reg_enabled_success_cnt

# Regulation-disabled vs regulation-enabled
perf_table <- matrix(
  c(
```

```
    reg_enabled_success_cnt,
    reg_disabled_success_cnt,
    reg_enabled_fail_cnt,
    reg_disabled_fail_cnt
    ),
    nrow=2
)

rownames(perf_table) <- c("reg-enabled", "reg-disabled")
colnames(perf_table) <- c("success", "fail")

print(perf_table)
```

```
##              success fail
## reg-enabled       66  134
## reg-disabled      30  170
```

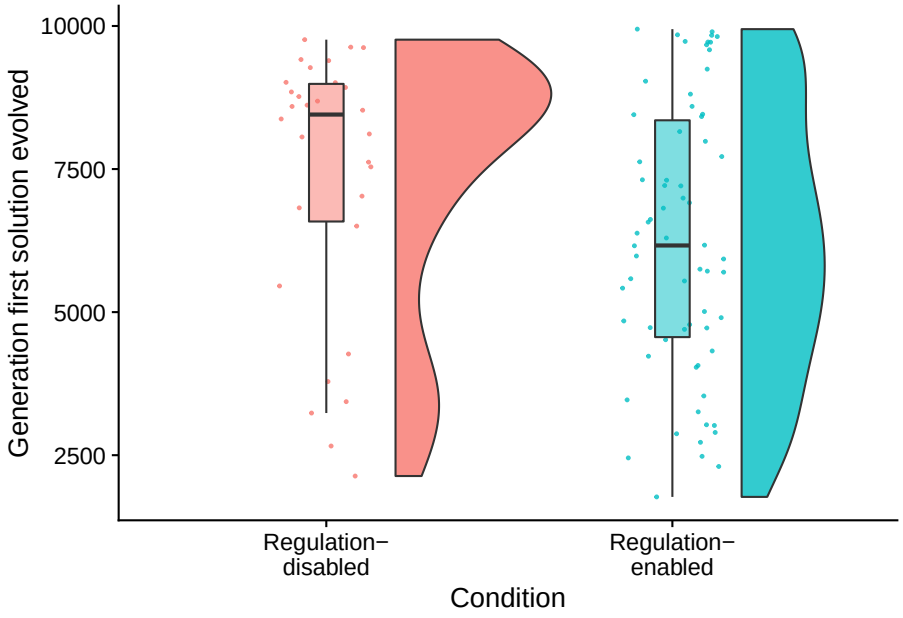
```
print(fisher.test(perf_table))
```

```
##
##  Fisher's Exact Test for Count Data
##
## data:  perf_table
## p-value = 3.585e-05
## alternative hypothesis: true odds ratio is not equal to 1
## 95 percent confidence interval:
##  1.673731 4.711896
## sample estimates:
## odds ratio
##   2.783852
```

9.5 How many generations elapse before solutions evolve?

```
ggplot( data = sol_data, aes(x=condition, y=update, fill=condition) ) +
  geom_flat_violin(
    position=position_nudge(x = .2, y = 0),
    alpha=.8
  ) +
  geom_point(
    aes(y=update, color=condition),
    position = position_jitter(width = .15),
    size = .5,
    alpha = 0.8
```

```
  ) +
  geom_boxplot(
    width = .1,
    outlier.shape = NA,
    alpha = 0.5
  ) +
  scale_x_discrete(
    name="Condition",
    breaks=c("memory", "both"),
    labels=c("Regulation-\ndisabled", "Regulation-\nenabled")
  ) +
  scale_y_continuous(name="Generation first solution evolved") +
  guides(fill = FALSE) +
  guides(color = FALSE) +
  ggsave(
    paste0(working_directory, "imgs/boolean-calc-prefix-solve-time-cloud.png"),
    width=4,
    height=4
  )
```



Test for statistical difference between conditions using a Wilcoxon rank sum test.

```
print(wilcox.test(formula=update~condition, data=sol_data, exact=FALSE, conf.int=TRUE))
```

```
##
##  Wilcoxon rank sum test with continuity correction
```

```
##
## data:  update by condition
## W = 1249, p-value = 0.04102
## alternative hypothesis: true location shift is not equal to 0
## 95 percent confidence interval:
##    45.00003 2448.99997
## sample estimates:
## difference in location
##               1291.265
```

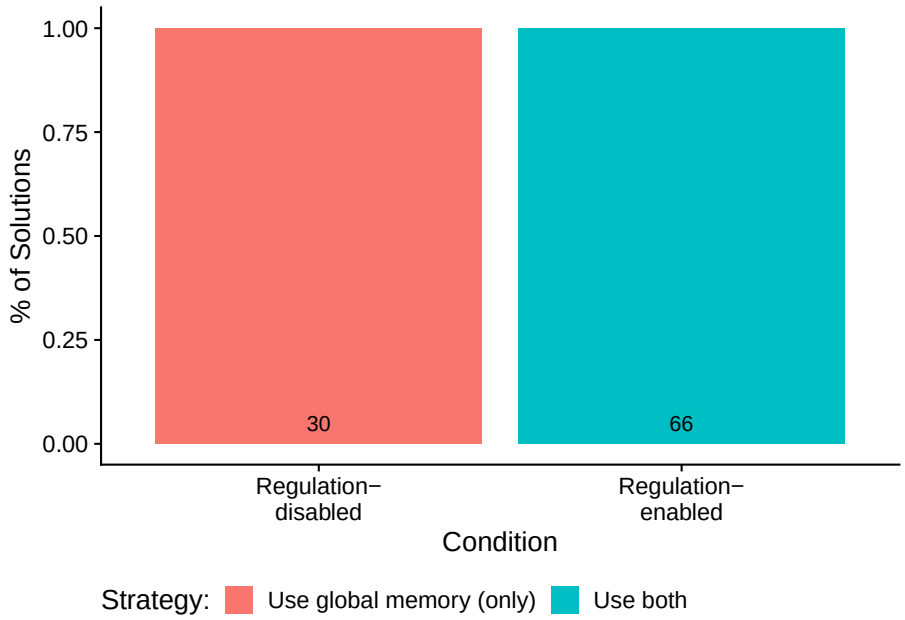
9.6 Evolved strategies

9.6.1 What mechanisms do programs rely on to adjust responses to signals over time?

We used indpendent knockouts of tag-based genetic regulation and global memory buffer access to investigate the mechanisms underpinning successful programs.

```
ggplot( sol_data, mapping=aes(x=condition, fill=strategy) ) +
  geom_bar(
    position="fill",
    stat="count"
  ) +
  geom_text(
    stat='count',
    mapping=aes(label=..count..),
    position=position_fill(vjust=0.05)
  ) +
  ylab("% of Solutions") +
  scale_fill_discrete(
    name="Strategy:",
    breaks=c(
      "use regulation",
      "use memory",
      "use neither",
      "use both"
    ),
    labels=c(
      "Use regulation (only)",
      "Use global memory (only)",
      "Use neither",
      "Use both"
    )
  ) +
  scale_x_discrete(
    name="Condition",
```

```
    breaks=c("memory", "both"),
    labels=c("Regulation-\ndisabled", "Regulation-\nenabled")
  ) +
  theme(legend.position = "bottom")
```



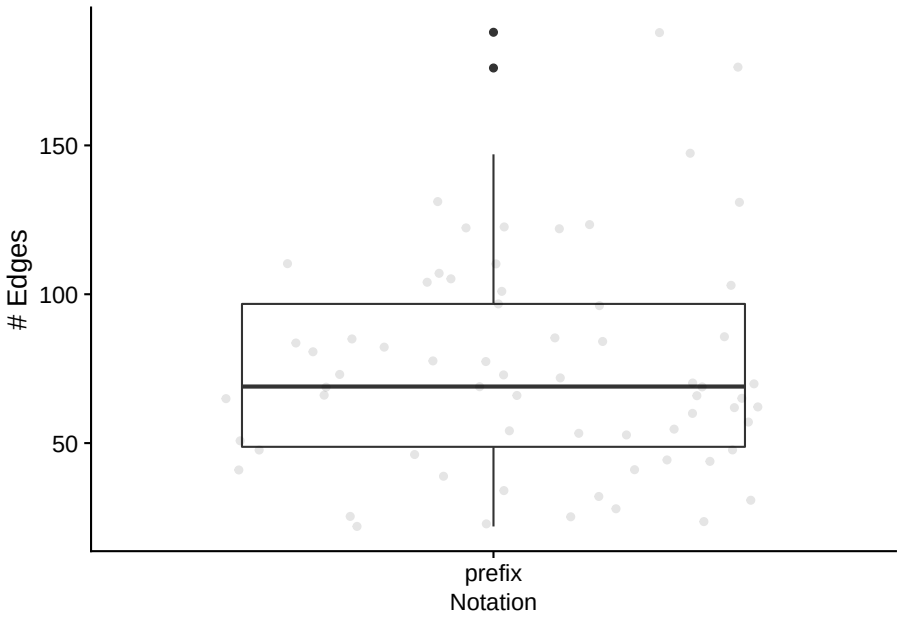
9.6.2 Gene regulatory networks

Looking only at successful programs that rely on regulation. At a glance, what do gene regulatory networks look like?

First, the total edges found in networks:

```
relies_on_reg <- filter(
  sol_data,
  relies_on_regulation=="1"
)$SEED

ggplot( filter(reg_network_data, run_id %in% relies_on_reg), aes(x=notation, y=edge_cnt
  geom_boxplot() +
  geom_jitter(alpha=0.1) +
  xlab("Notation") +
  ylab("# Edges") +
  theme(
    legend.position="bottom",
    legend.text=element_text(size=9),
```

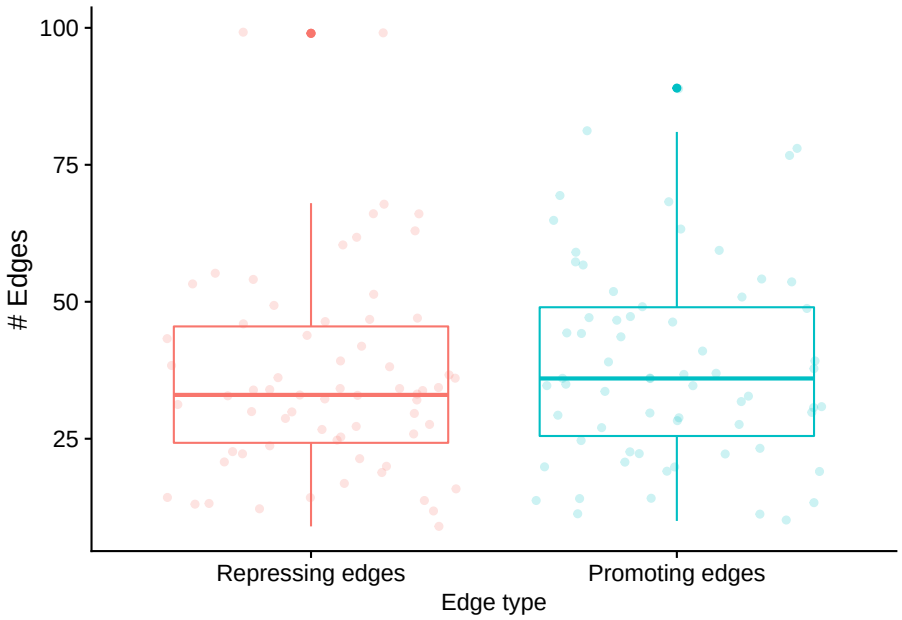
```
    legend.title=element_text(size=10),
    axis.title.x=element_text(size=12)
  ) +
  ggsave(
    paste0(working_directory, "imgs/boolean-calc-prefix-regulation-edges.png"),
    width=4,
    height=3
  )
```



Next, let's look at edges by type.

```
# Process/cleanup the network data
melted_network_data <- melt(
  filter(reg_network_data,
         run_id %in% relies_on_reg
        ),
  variable.name = "reg_edge_type",
  value.name = "reg_edges_cnt",
  measure.vars=c("repressed_edges_cnt", "promoted_edges_cnt")
)

ggplot( melted_network_data, aes(x=reg_edge_type, y=reg_edges_cnt, color=reg_edge_type) ) +
  geom_boxplot() +
  geom_jitter(alpha=0.2) +
  xlab("Environmental Complexity") +
```

```
  ylab("# Edges") +
  scale_x_discrete(
    name="Edge type",
    limits=c("repressed_edges_cnt", "promoted_edges_cnt"),
    labels=c("Repressing edges", "Promoting edges")
  ) +
  theme(
    legend.position="none",
    legend.text=element_text(size=9),
    legend.title=element_text(size=10),
    axis.title.x=element_text(size=12)
  ) +
  ggsave(
    paste0(working_directory, "imgs/boolean-calc-prefix-regulation-edge-types.png"),
    width=4,
    height=3
  )
```



Test for a statistical difference between edge types using a wilcoxon rank sum test:

```
print(
  paste0(
    "Median # repressed edges: ",
    median(filter(melted_network_data, reg_edge_type=="repressed_edges_cnt")$reg_edges_
```

```
  )
)
```

```
## [1] "Median # repressed edges: 33"
```

```
print(
  paste0(
    "Median # promoting edges: ",
    median(filter(melted_network_data, reg_edge_type=="promoted_edges_cnt")$reg_edges_cnt)
  )
)
```

```
## [1] "Median # promoting edges: 36"
```

```
print(wilcox.test(formula=reg_edges_cnt ~ reg_edge_type, data=melted_network_data, exact=FALSE, c
```

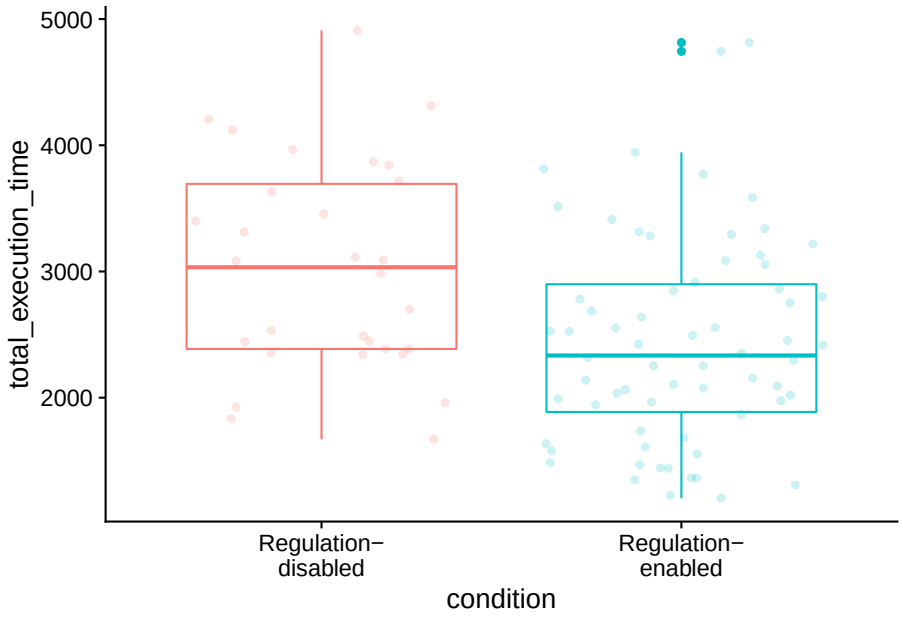
```
##
##  Wilcoxon rank sum test with continuity correction
##
## data:  reg_edges_cnt by reg_edge_type
## W = 1950.5, p-value = 0.3014
## alternative hypothesis: true location shift is not equal to 0
## 95 percent confidence interval:
##  -8.999949  2.999926
## sample estimates:
## difference in location
##              -2.999963
```

9.6.3 Program instruction execution traces

9.6.3.1 Execution time

How many time steps do successful programs take to solve the boolean calculator problem?

```
# only want solutions
solutions_inst_exec_data <- filter(inst_exec_data, SEED %in% sol_data$SEED)

ggplot( solutions_inst_exec_data, aes(x=condition, y=total_execution_time, color=condition) ) +
  geom_boxplot() +
  geom_jitter(alpha=0.2) +
  scale_x_discrete(
    breaks=c("memory", "both"),
    labels=c("Regulation-\ndisabled", "Regulation-\nenabled")
  ) +
  theme(
    legend.position="none"
  )
```



Test for significant difference between conditions using Wilcoxon rank sum test:

```
print(
  wilcox.test(
    formula=total_execution_time~condition,
    data=filter(solutions_inst_exec_data),
    exact=FALSE,
    conf.int=TRUE)
)
```

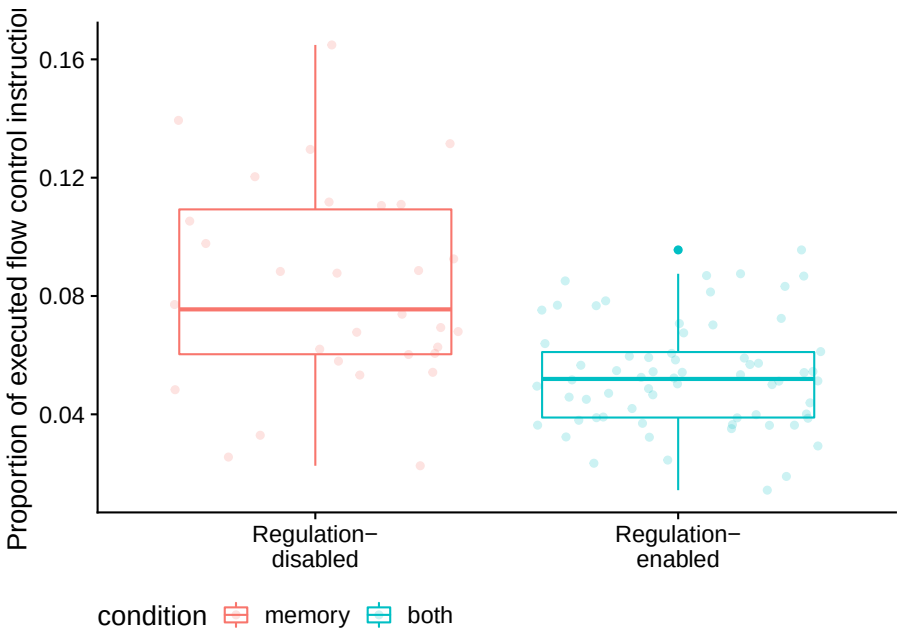
```
##
##  Wilcoxon rank sum test with continuity correction
##
## data:  total_execution_time by condition
## W = 1374, p-value = 0.002434
## alternative hypothesis: true location shift is not equal to 0
## 95 percent confidence interval:
##  240 986
## sample estimates:
## difference in location
##               587.0774
```

9.6.3.2 What types of instructions to successful programs execute?

Here, we look at the distribution of instruction types executed by successful programs. We're primarily interested in the proportion of control flow instructions,

so let's look at that first.

```
ggplot( solutions_inst_exec_data, aes(x=condition, y=control_flow_inst_prop, color=condition) ) +
  geom_boxplot() +
  geom_jitter(alpha=0.2) +
  scale_x_discrete(
    breaks=c("memory", "both"),
    labels=c("Regulation-\ndisabled", "Regulation-\nenabled")
  ) +
  ylab("Proportion of executed flow control instructions") +
  theme(
    legend.position="bottom",
    axis.title.x=element_blank()
  )
```



Test for significant difference between conditions using a Wilcoxon rank sum test:

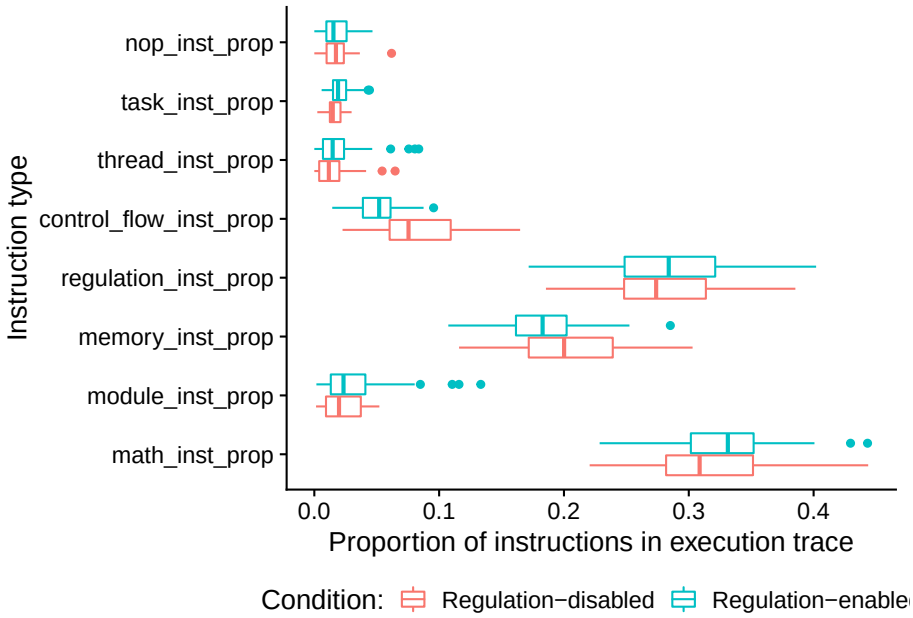
```
print(
  wilcox.test(
    formula=control_flow_inst_prop~condition,
    data=filter(solutions_inst_exec_data),
    exact=FALSE,
    conf.int=TRUE)
)
```

```
##
```

```
##  Wilcoxon rank sum test with continuity correction
##
## data:  control_flow_inst_prop by condition
## W = 1541.5, p-value = 1.328e-05
## alternative hypothesis: true location shift is not equal to 0
## 95 percent confidence interval:
##  0.01486019 0.03910795
## sample estimates:
## difference in location
##              0.02639857
```

In case you're curious, here's all categories of instructions:

```
melted <- melt(
  solutions_inst_exec_data,
  variable.name = "inst_type",
  value.name = "inst_type_prop",
  measure.vars=c(
    "math_inst_prop",
    "module_inst_prop",
    "memory_inst_prop",
    "regulation_inst_prop",
    "control_flow_inst_prop",
    "thread_inst_prop",
    "task_inst_prop",
    "nop_inst_prop"
  )
)

ggplot( melted, aes(x=inst_type, y=inst_type_prop, color=condition) ) +
  geom_boxplot() +
  scale_color_discrete(
    name="Condition:",
    breaks=c("memory", "both"),
    labels=c("Regulation-disabled", "Regulation-enabled")
  ) +
  xlab("Instruction type") +
  ylab("Proportion of instructions in execution trace") +
  coord_flip() +
  theme(legend.position="bottom")
```



9.7 Visualizaing an evolved regulatory network

Let's take a closer look at a successful gene regulatory network.

```
# 24386-393 24391-394 24392-394 24393 24398 24400
trace_id <- 24400
test_id <- 420
```

Specifically, we'll be looking at the solution evolved in run id 2.44×10^4 (arbitrarily selected).

9.7.1 Data wrangling

```
case_study_info <- read.csv(
  paste0(working_directory, "data/max_fit_orgs.csv"),
  na.strings="NONE"
)

case_study_info <- filter(
  case_study_info,
  SEED==trace_id
)

# Extract relevant information about solution of interest.
```

```
is_sol <- case_study_info$solution
num_modules <- case_study_info$num_modules

# Load trace file associated with this solution.
trace_file <- paste0(working_directory, "data/reg-traces/trace-reg_update-10000_run-id
trace_data <- read.csv(trace_file, na.strings="NONE")
trace_data <- filter(trace_data, cur_test_id == test_id)

# Data cleanup/summarizing
trace_data$is_running <- trace_data$is_running == "1" | trace_data$is_triggered == "1"

# Build a list of modules that were triggered & those that responded to a signal
triggered_ids <- levels(factor(filter(trace_data, is_triggered=="1")$module_id))
response_ids <- levels(factor(filter(trace_data, is_cur_responding_function=="1")$modul
regulated_ids <- levels(factor(filter(trace_data, regulator_state != 0)$module_id))
final_response_ids <- levels(factor(filter(trace_data, is_cur_responding_function=="1"

trace_data$is_ever_active <-
  trace_data$is_ever_active=="1" |
  trace_data$is_running |
  trace_data$module_id %in% triggered_ids |
  trace_data$module_id %in% response_ids |
  trace_data$module_id %in% regulated_ids

# function to categorize each regulatory state as promoted, neutral, or repressed
# remember, the regulatory states in our data file operate with tag DISTANCE in mind
# as opposed to tag similarity, so: promotion => reg < 0, repression => reg > 0
categorize_reg_state <- function(reg_state) {
  if (reg_state == 0) {
    return("neutral")
  } else if (reg_state < 0) {
    return("promoted")
  } else if (reg_state > 0) {
    return("repressed")
  } else {
    return("unknown")
  }
}

trace_data$regulator_state_simplified <- mapply(
  categorize_reg_state,
  trace_data$regulator_state
)

# Omit all in-active rows
```

```
# Extract only rows that correspond with modules that were active during evaluation.
active_data <- filter(trace_data, is_ever_active==TRUE)

# Do some work to have module ids appear in a nice order along axis.
active_module_ids <- levels(factor(active_data$module_id))
active_module_ids <- as.integer(active_module_ids)
module_id_map <- as.data.frame(active_module_ids)
module_id_map$order <- order(module_id_map$active_module_ids) - 1

get_module_x_pos <- function(module_id) {
  return(filter(module_id_map, active_module_ids==module_id)$order)
}

active_data$mod_id_x_pos <- mapply(get_module_x_pos, active_data$module_id)
```

9.7.2 Function regulation over time

First, let's omit all non-active funcitons.

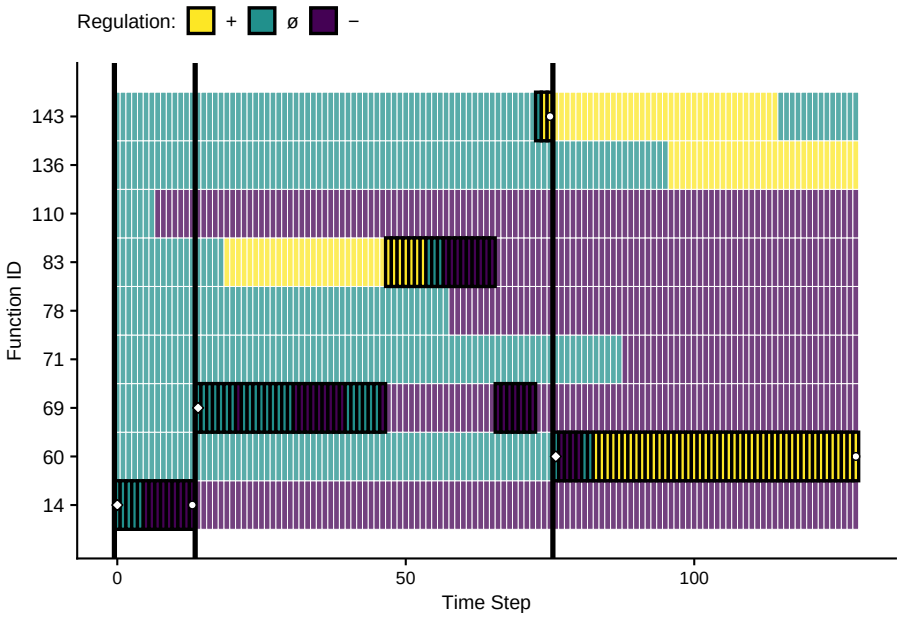
Horizontal orientation:

```
out_name <- paste0(
  working_directory,
  "imgs/case-study-trace-id-",
   trace_id,
   "-test_id-",
   test_id,
   "-regulator-state-horizontal.pdf"
)

active_data$module_id <- factor(active_data$module_id)
active_data$graph_time_step <- active_data$time_step - min(active_data$time_step)

ggplot(active_data, aes(x=mod_id_x_pos, y=graph_time_step, fill=regulator_state_simplified)) +
  scale_fill_viridis(
    name="Regulation:",
    limits=c(
      "promoted",
      "neutral",
      "repressed"
    ),
    labels=c(
      "+",
      "\u00F8",
      "-"
    ),
```

```
    discrete=TRUE,
    direction=-1
  ) +
  scale_x_discrete(
    name="Function ID",
    limits=seq(0, length(active_module_ids)-1, 1),
    labels=active_module_ids
  ) +
  ylab("Time Step") +
  # Background tile color
  geom_tile(
    color="white",
    size=0.2,
    width=1,
    height=1,
    alpha=0.75
  ) +
  # Highlight actively running functions
  geom_tile(
    data=filter(active_data, is_running==TRUE | is_triggered=="1"),
    color="black",
    size=0.8,
    width=1,
    height=1
  ) +
  # Environment delimiters
  geom_hline(
    yintercept=filter(active_data, cpu_step==0)$graph_time_step - 0.5,
    size=1.25,
    color="black"
  ) +
  # Draw points on triggered modules
  geom_point(
    data=filter(active_data, is_triggered=="1"),
    shape=23,
    colour="black",
    fill="white",
    stroke=0.5,
    size=1.75,
    position=position_nudge(x = 0, y = 0.01)
  ) +
  geom_point(
    data=filter(active_data, is_cur_responding_function=="1"),
    shape=21,
    colour="black",
```

```
    fill="white",
    stroke=0.5,
    size=1.5,
    position=position_nudge(x = 0, y = 0.01)
  ) +
  theme(
    legend.position = "top",
    legend.text = element_text(size=10),
    legend.title=element_text(size=10),
    axis.text.y = element_text(size=10),
    axis.title.y = element_text(size=10),
    axis.text.x = element_text(size=9),
    axis.title.x = element_text(size=10),
    plot.title = element_text(hjust = 0.5)
  ) +
  coord_flip() +
  ggsave(out_name, height=4, width=8)
```

```
## Warning: Continuous limits supplied to discrete scale.
## Did you mean `limits = factor(...)` or `scale_*_continuous()`?
```



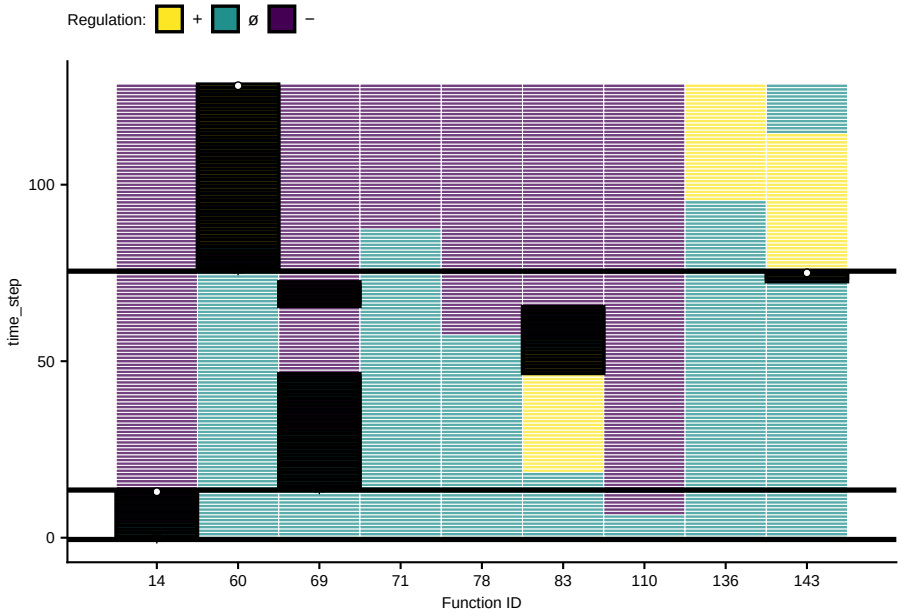
Vertical orientation:

```
out_name <- paste0(
  working_directory,
  "imgs/case-study-trace-id-",
```

```
    trace_id,
    "-test_id-",
    test_id,
    "-regulator-state-vertical.pdf"
)

ggplot(
    active_data,
    aes(x=mod_id_x_pos, y=time_step, fill=regulator_state_simplified)
  ) +
  scale_fill_viridis(
    name="Regulation:",
    limits=c(
      "promoted",
      "neutral",
      "repressed"
    ),
    labels=c(
      "+",
      "\u00F8",
      "-"
    ),
    discrete=TRUE,
    direction=-1
  ) +
  scale_x_discrete(
    name="Function ID",
    limits=seq(0, length(active_module_ids)-1, 1),
    labels=active_module_ids
  ) +
  # scale_y_discrete(
  #   name="Time Step"
  # ) +
  # Background tile color
  geom_tile(
    color="white",
    size=0.2,
    width=1,
    height=1,
    alpha=0.75
  ) +
  # Highlight actively running functions
  geom_tile(
    data=filter(active_data, is_running==TRUE | is_triggered=="1"),
    color="black",
```

```
    size=0.8,
    width=1,
    height=1
  ) +
  # Environment delimiters
  geom_hline(
    yintercept=filter(active_data, cpu_step==0)$time_step - 0.5,
    size=1.25,
    color="black"
  ) +
  # Draw points on triggered modules
  geom_point(
    data=filter(active_data, is_triggered=="1"),
    shape=8,
    colour="black",
    fill="white",
    stroke=0.5,
    size=1.5,
    position=position_nudge(x = 0, y = 0.01)
  ) +
  geom_point(
    data=filter(active_data, is_cur_responding_function=="1"),
    shape=21,
    colour="black",
    fill="white",
    stroke=0.5,
    size=1.5,
    position=position_nudge(x = 0, y = 0.01)
  ) +
  theme(
    legend.position = "top",
    legend.text = element_text(size=9),
    legend.title=element_text(size=8),
    axis.text.y = element_text(size=8),
    axis.title.y = element_text(size=8),
    axis.text.x = element_text(size=8),
    axis.title.x = element_text(size=8),
    plot.title = element_text(hjust = 0.5)
  ) +
  ggsave(
    out_name,
    height=3.5,
    width=2.25
  )
```

```
## Warning: Continuous limits supplied to discrete scale.
## Did you mean `limits = factor(...)` or `scale_*_continuous()`?
```



9.7.3 Evolved regulatory network

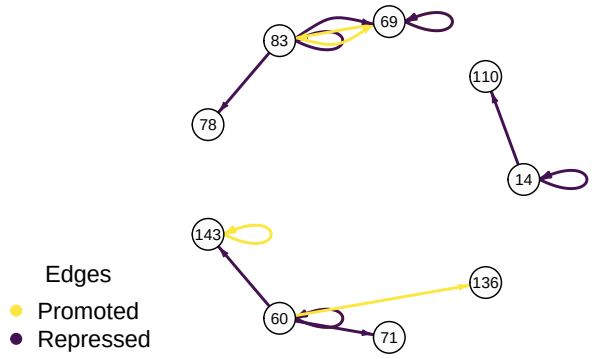
We use the igraph package to draw this program's gene regulatory network.

```
# Networks!
graph_nodes_loc <- paste0(working_directory, "data/igraphs/reg_graph_id-", trace_id, "_
graph_edges_loc <- paste0(working_directory, "data/igraphs/reg_graph_id-", trace_id, "_
graph_nodes_data <- read.csv(graph_nodes_loc, na.strings="NONE")
graph_edges_data <- read.csv(graph_edges_loc, na.strings="NONE")

network <- graph_from_data_frame(
  d=graph_edges_data,
  vertices=graph_nodes_data,
  directed=TRUE
)

# Setup edge styling
E(network)$color[E(network)$type == "promote"] <- "#FCE640"
E(network)$lty[E(network)$type == "promote"] <- 1
E(network)$color[E(network)$type == "repress"] <- "#441152"
E(network)$lty[E(network)$type == "repress"] <- 1

network_out_name <- paste0(working_directory, "imgs/case-study-id-", trace_id, "-test-
```

```
draw_network <- function(net, write_out, out_name) {
  if (write_out) {
    svg(out_name, width=6,height=3)
    # bottom, left, top, right
    par(mar=c(0.2,0,1,0.5))
  }
  plot(
    net,
    edge.arrow.size=0.4,
    edge.arrow.width=0.75,
    edge.width=2,
    vertex.size=20,
    vertex.label.cex=0.65,
    curved=TRUE,
    vertex.color="grey99",
    vertex.label.color="black",
    vertex.label.family="sans",
    layout=layout.circle(net)
  )
  legend(
    x = "bottomleft",      ## position, also takes x,y coordinates
    legend = c("Promoted", "Repressed"),
    pch = 19,              ## legend symbols see ?points
    col = c("#FCE640", "#441152"),
    bty = "n",
    border="black",
    xpd=TRUE,
    title = "Edges"
  )
  if (write_out) {
    dev.flush()
    dev.off()
  }
}

draw_network(network, TRUE, network_out_name)
```

```
## pdf
##   2
```

```
draw_network(network, FALSE, "")
```


69
83
110
78
14
143
136
60
71
Edges
Promoted
Repressed

Chapter 10

Boolean calculator problem (postfix notation)

Here, we give an overview of the boolean logic calculator problem, and we provide our data analyses for related experiments. All of our source code for statistical analyses and data visualizations is embedded in this document. The raw data can be found on the OSF project associated with this work (Lalejini et al., 2020).

Please file an issue or make a pull request on github to report any mistakes, ask questions, request more explanation, et cetera.

10.1 Overview

```
# Experimental parameters referenced in-text all in one convenient place.
time_steps <- 128
replicates <- 200
population_size <- 1000
generations <- 10000

# Settings for statistical analyses.
alpha <- 0.05

# Relative location of data.
working_directory <- "experiments/2020-11-28-bool-calc-postfix/analysis/" # << For bookdown
# working_directory <- "./"                                               # << For local analysis
```

We use a modified version of the Boolean logic calculator problem to further investigate the potential for our implementation of tag-based regulation to impede adaptive evolution. Our previous experiments with the Boolean logic calculator

problem provided 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 (e.g., AND, OR, XOR, etc.). This requires programs to adjust their response to the final input signal based on the context provided by the previous two signals, thereby increasing the value of regulation.

We explore whether the calculator problem's context-dependence is driving the benefit of tag-based regulation that we identified in previous experiments. 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 changing-signal problem, a distinct signal will differentiate which of the response types a program should execute.

10.2 Analysis Dependencies

Load all required R libraries.

```
library(ggplot2)
library(tidyverse)
library(cowplot)
library(viridis)
library(reshape2)
library(igraph)
source("https://gist.githubusercontent.com/benmarwick/2a1bb0133ff568cbe28d/raw/fb53bd9
```

These analyses were conducted in the following computing environment:

```
print(version)
```

```
##                _
## platform       x86_64-pc-linux-gnu
## arch           x86_64
## os             linux-gnu
## system         x86_64, linux-gnu
## status
## major          4
## minor          0.2
## year           2020
## month          06
## day            22
## svn rev        78730
## language       R
## version.string R version 4.0.2 (2020-06-22)
```

```
## nickname       Taking Off Again
```

10.3 Setup

Load data, initial data cleanup, configure some global settings.

```
data_loc <- paste0(working_directory, "data/max_fit_orgs.csv")
data <- read.csv(data_loc, na.strings="NONE")

# Specify factors (not all of these matter for this set of runs).
data$matchbin_thresh <- factor(
  data$matchbin_thresh,
  levels=c(0, 25, 50, 75)
)

data$TAG_LEN <- factor(
  data$TAG_LEN,
  levels=c(32, 64, 128, 256)
)

data$notation <- factor(
  data$notation,
  levels=c("prefix", "postfix")
)

# Define function to summarize regulation/memory configurations.
get_con <- function(reg, mem) {
  if (reg == "0" && mem == "0") {
    return("none")
  } else if (reg == "0" && mem=="1") {
    return("memory")
  } else if (reg=="1" && mem=="0") {
    return("regulation")
  } else if (reg=="1" && mem=="1") {
    return("both")
  } else {
    return("UNKNOWN")
  }
}

# Specify experimental condition for each datum.
data$condition <- mapply(
  get_con,
  data$USE_FUNC_REGULATION,
  data$USE_GLOBAL_MEMORY
```

```
)

data$condition <- factor(
  data$condition,
  levels=c("regulation", "memory", "none", "both")
)

# Given knockout info, what strategy does a program use?
get_strategy <- function(use_reg, use_mem) {
  if (use_reg=="0" && use_mem=="0") {
    return("use neither")
  } else if (use_reg=="0" && use_mem=="1") {
    return("use memory")
  } else if (use_reg=="1" && use_mem=="0") {
    return("use regulation")
  } else if (use_reg=="1" && use_mem=="1") {
    return("use both")
  } else {
    return("UNKNOWN")
  }
}

# Specify experimental conditions (to make labeling easier).
data$strategy <- mapply(
  get_strategy,
  data$relies_on_regulation,
  data$relies_on_global_memory
)

data$strategy <- factor(
  data$strategy,
  levels=c(
    "use regulation",
    "use memory",
    "use neither",
    "use both"
  )
)

# Filter data to include only replicates labeled as solutions
sol_data <- filter(data, solution=="1")

####### Load instruction execution data #######
inst_exec_data <- read.csv(paste0(working_directory, "data/exec_trace_summary.csv"), na
```

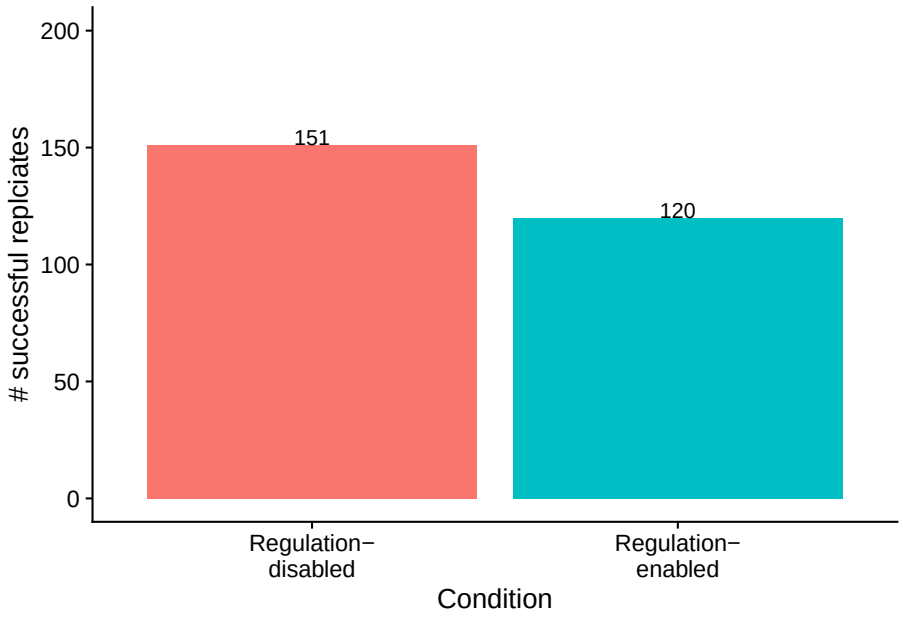
```
inst_exec_data$condition <- mapply(
  get_con,
  inst_exec_data$USE_FUNC_REGULATION,
  inst_exec_data$USE_GLOBAL_MEMORY
)

inst_exec_data$condition <- factor(
  inst_exec_data$condition,
  levels=c("regulation", "memory", "none", "both")
)

inst_exec_data$notation <- factor(
  inst_exec_data$notation,
  levels=c("prefix", "postfix")
)

####### Load network data #######
reg_network_data <- read.csv(paste0(working_directory, "data/reg_graphs_summary.csv"), na.strings
reg_network_data <- filter(reg_network_data, run_id %in% data$SEED)

get_notation <- function(seed) {
  return(filter(data, SEED==seed)$notation)
}

reg_network_data$notation <- mapply(
  get_notation,
  reg_network_data$run_id
)

reg_network_data$notation <- factor(
  reg_network_data$notation,
  levels=c("prefix", "postfix")
)

####### misc #######
# Configure our default graphing theme
theme_set(theme_cowplot())
```

10.4 Problem-solving success

The number of successful replicates by condition:

```
# Graph the number of solutions evolved in each condition, faceted by environmental complexity
ggplot(sol_data, aes(x=condition, fill=condition)) +
  geom_bar() +
```

```
  geom_text(
    stat="count",
    mapping=aes(label=..count..),
    position=position_dodge(0.9),
    vjust=0
  ) +
  scale_x_discrete(
    name="Condition",
    breaks=c("memory","both"),
    labels=c("Regulation-\ndisabled", "Regulation-\nenabled")
  ) +
  ylab("# successful replciates") +
  ylim(0, 200) +
  theme(legend.position = "none") +
  ggsave(
    paste0(working_directory, "imgs/boolean-calc-postfix-solution-counts.pdf"),
    width=4,
    height=4
  )
```



Test for significance using Fisher's exact test.

```
# Extract successes/fails for each condition.
reg_disabled_success_cnt <- nrow(filter(sol_data, solution=="1" & condition=="memory"))
reg_disabled_fail_cnt <- replicates - reg_disabled_success_cnt
```

```
reg_enabled_success_cnt <- nrow(filter(sol_data, solution=="1" & condition=="both"))
reg_enabled_fail_cnt <- replicates - reg_enabled_success_cnt

# Regulation-disabled vs regulation-enabled
perf_table <- matrix(
  c(
    reg_enabled_success_cnt,
    reg_disabled_success_cnt,
    reg_enabled_fail_cnt,
    reg_disabled_fail_cnt
    ),
    nrow=2
)

rownames(perf_table) <- c("reg-enabled", "reg-disabled")
colnames(perf_table) <- c("success", "fail")

print(perf_table)
```

```
##              success fail
## reg-enabled      120   80
## reg-disabled     151   49
```

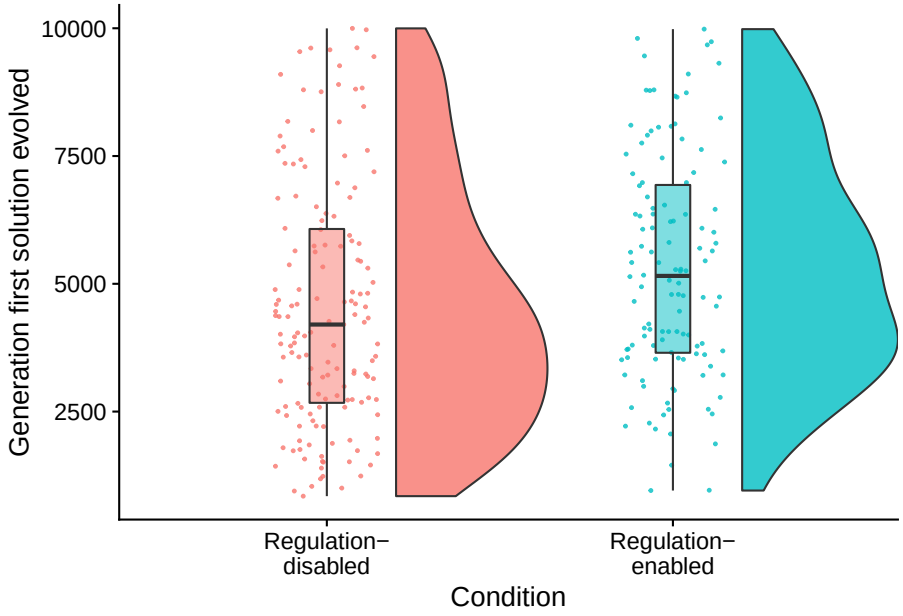
```
print(fisher.test(perf_table))
```

```
##
##  Fisher's Exact Test for Count Data
##
## data:  perf_table
## p-value = 0.001286
## alternative hypothesis: true odds ratio is not equal to 1
## 95 percent confidence interval:
##  0.3093253 0.7635173
## sample estimates:
## odds ratio
##  0.4876392
```

10.5 How many generations elapse before solutions evolve?

```
ggplot( data = sol_data, aes(x=condition, y=update, fill=condition) ) +
  geom_flat_violin(
    position=position_nudge(x = .2, y = 0),
    alpha=.8
```

```
  ) +
  geom_point(
    aes(y=update, color=condition),
    position = position_jitter(width = .15),
    size = .5,
    alpha = 0.8
  ) +
  geom_boxplot(
    width = .1,
    outlier.shape = NA,
    alpha = 0.5
  ) +
  scale_x_discrete(
    name="Condition",
    breaks=c("memory", "both"),
    labels=c("Regulation-\ndisabled", "Regulation-\nenabled")
  ) +
  scale_y_continuous(name="Generation first solution evolved") +
  guides(fill = FALSE) +
  guides(color = FALSE) +
  ggsave(
    paste0(working_directory, "imgs/boolean-calc-postfix-solve-time-cloud.png"),
    width=4,
    height=4
  )
```



Test for statistical difference between conditions using a Wilcoxon rank sum test.

```
print(wilcox.test(formula=update~condition, data=sol_data, exact=FALSE, conf.int=TRUE))
```

```
##
##  Wilcoxon rank sum test with continuity correction
##
## data:  update by condition
## W = 7175.5, p-value = 0.003285
## alternative hypothesis: true location shift is not equal to 0
## 95 percent confidence interval:
##  -1422  -310
## sample estimates:
## difference in location
##                   -872
```

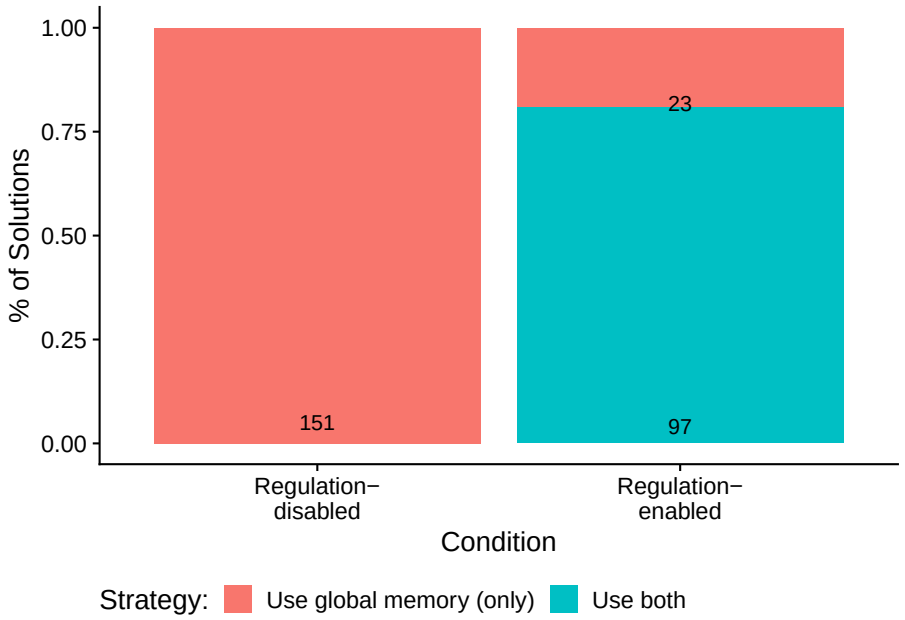
10.6 Evolved strategies

10.6.1 What mechanisms do programs rely on to adjust responses to signals over time?

We used indpendent knockouts of tag-based genetic regulation and global memory buffer access to investigate the mechanisms underpinning successful programs.

```
ggplot( sol_data, mapping=aes(x=condition, fill=strategy) ) +
  geom_bar(
```

```
    position="fill",
    stat="count"
  ) +
  geom_text(
    stat='count',
    mapping=aes(label=..count..),
    position=position_fill(vjust=0.05)
  ) +
  ylab("% of Solutions") +
  scale_fill_discrete(
    name="Strategy:",
    breaks=c(
      "use regulation",
      "use memory",
      "use neither",
      "use both"
    ),
    labels=c(
      "Use regulation (only)",
      "Use global memory (only)",
      "Use neither",
      "Use both"
    )
  ) +
  scale_x_discrete(
    name="Condition",
    breaks=c("memory", "both"),
    labels=c("Regulation-\ndisabled", "Regulation-\nenabled")
  ) +
  theme(legend.position = "bottom")
```



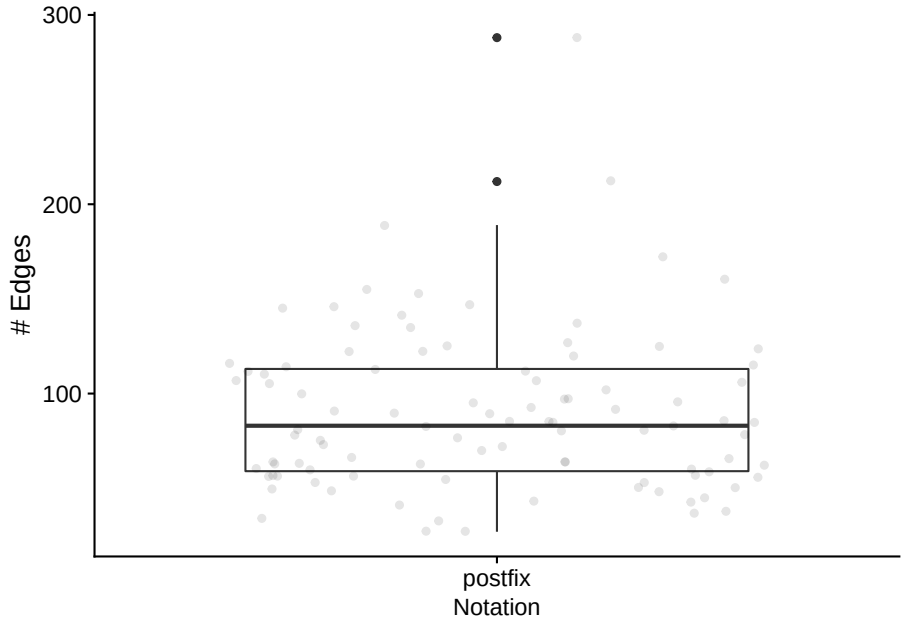
10.6.2 Gene regulatory networks

Looking only at successful programs that rely on regulation. At a glance, what do gene regulatory networks look like?

First, the total edges found in networks:

```
relies_on_reg <- filter(
  sol_data,
  relies_on_regulation=="1"
)$SEED

ggplot( filter(reg_network_data, run_id %in% relies_on_reg), aes(x=notation, y=edge_cnt) ) +
  geom_boxplot() +
  geom_jitter(alpha=0.1) +
  xlab("Notation") +
  ylab("# Edges") +
  theme(
    legend.position="bottom",
    legend.text=element_text(size=9),
    legend.title=element_text(size=10),
    axis.title.x=element_text(size=12)
  ) +
  ggsave(
    paste0(working_directory, "imgs/boolean-calc-postfix-regulation-edges.png"),
    width=4,
```

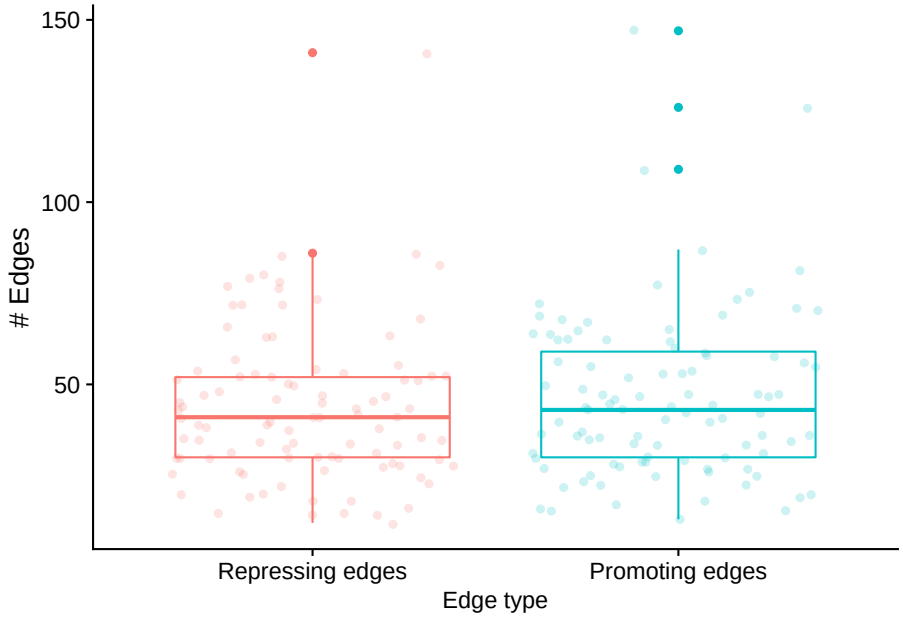
```
    height=3
  )
```



Next, let's look at edges by type.

```
# Process/cleanup the network data
melted_network_data <- melt(
  filter(reg_network_data,
         run_id %in% relies_on_reg
        ),
  variable.name = "reg_edge_type",
  value.name = "reg_edges_cnt",
  measure.vars=c("repressed_edges_cnt", "promoted_edges_cnt")
)

ggplot( melted_network_data, aes(x=reg_edge_type, y=reg_edges_cnt, color=reg_edge_type)
  geom_boxplot() +
  geom_jitter(alpha=0.2) +
  xlab("Environmental Complexity") +
  ylab("# Edges") +
  scale_x_discrete(
    name="Edge type",
    limits=c("repressed_edges_cnt", "promoted_edges_cnt"),
    labels=c("Repressing edges", "Promoting edges")
  ) +
```

```
  theme(
    legend.position="none",
    legend.text=element_text(size=9),
    legend.title=element_text(size=10),
    axis.title.x=element_text(size=12)
  ) +
  ggsave(
    paste0(working_directory, "imgs/boolean-calc-postfix-regulation-edge-types.png"),
    width=4,
    height=3
  )
```



Test for a statistical difference between edge types using a wilcoxon rank sum test:

```
print(
  paste0(
    "Median # repressed edges: ",
    median(filter(melted_network_data, reg_edge_type=="repressed_edges_cnt")$reg_edges_cnt)
  )
)
```

```
## [1] "Median # repressed edges: 41"
```

```
print(
  paste0(
```

```
    "Median # promoting edges: ",
    median(filter(melted_network_data, reg_edge_type=="promoted_edges_cnt")$reg_edges_
  )
)
```

```
## [1] "Median # promoting edges: 43"
```

```
print(wilcox.test(formula=reg_edges_cnt ~ reg_edge_type, data=melted_network_data, exa
```

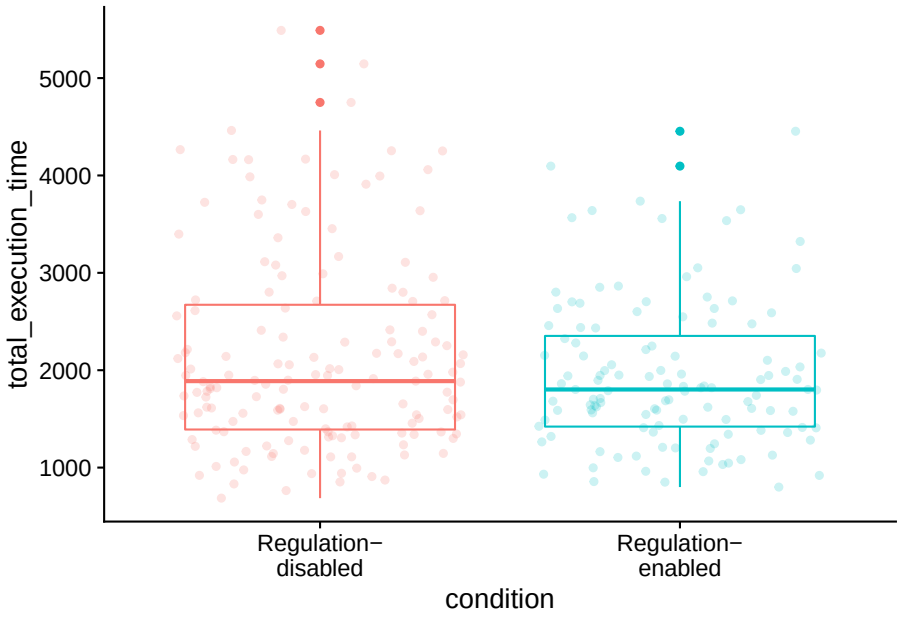
```
##
##  Wilcoxon rank sum test with continuity correction
##
## data:  reg_edges_cnt by reg_edge_type
## W = 4411, p-value = 0.4535
## alternative hypothesis: true location shift is not equal to 0
## 95 percent confidence interval:
##  -7.999975  3.000061
## sample estimates:
## difference in location
##              -1.999985
```

10.6.3 Program instruction execution traces

10.6.3.1 Execution time

How many time steps do successful programs take to solve the boolean calculator problem?

```
# only want solutions
solutions_inst_exec_data <- filter(inst_exec_data, SEED %in% sol_data$SEED)

ggplot( solutions_inst_exec_data, aes(x=condition, y=total_execution_time, color=condi
  geom_boxplot() +
  geom_jitter(alpha=0.2) +
  scale_x_discrete(
    breaks=c("memory", "both"),
    labels=c("Regulation-\ndisabled", "Regulation-\nenabled")
  ) +
  theme(
    legend.position="none"
  )
```



Test for significant difference between conditions using Wilcoxon rank sum test:

```
print(
  wilcox.test(
    formula=total_execution_time~condition,
    data=filter(solutions_inst_exec_data),
    exact=FALSE,
    conf.int=TRUE)
)
```

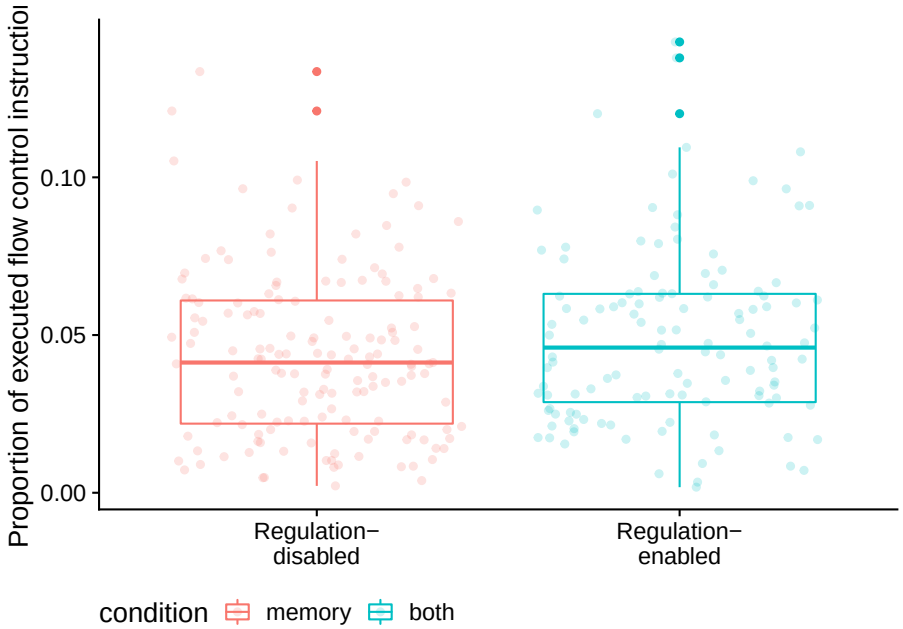
```
##
##  Wilcoxon rank sum test with continuity correction
##
## data:  total_execution_time by condition
## W = 9737, p-value = 0.2912
## alternative hypothesis: true location shift is not equal to 0
## 95 percent confidence interval:
##  -78.00002 274.00000
## sample estimates:
## difference in location
##               96.00004
```

10.6.3.2 What types of instructions to successful programs execute?

Here, we look at the distribution of instruction types executed by successful programs. We're primarily interested in the proportion of control flow instructions,

so let's look at that first.

```
ggplot( solutions_inst_exec_data, aes(x=condition, y=control_flow_inst_prop, color=cond
  geom_boxplot() +
  geom_jitter(alpha=0.2) +
  scale_x_discrete(
    breaks=c("memory", "both"),
    labels=c("Regulation-\ndisabled", "Regulation-\nenabled")
  ) +
  ylab("Proportion of executed flow control instructions") +
  theme(
    legend.position="bottom",
    axis.title.x=element_blank()
  )
```



Test for significant difference between conditions using a Wilcoxon rank sum test:

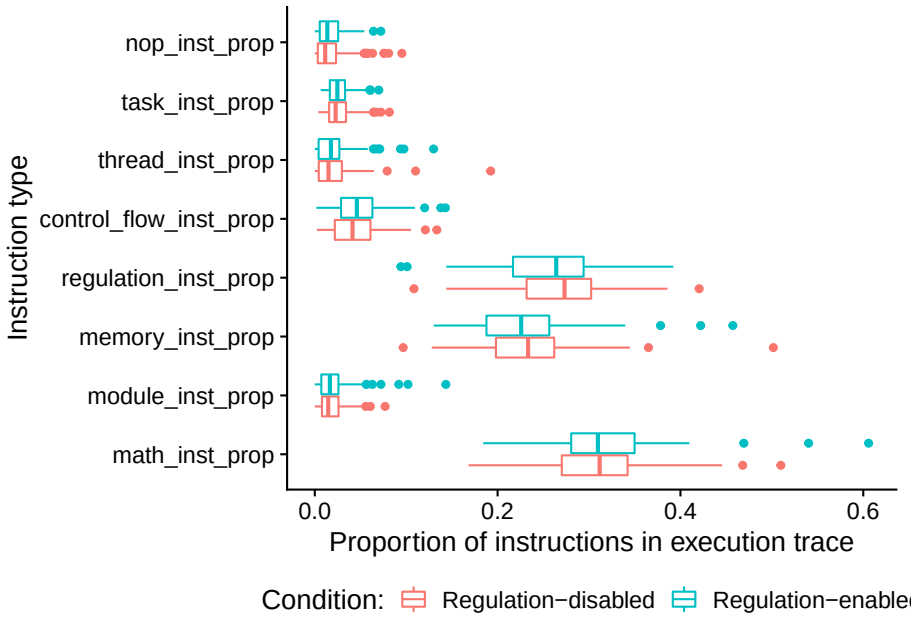
```
print(
  wilcox.test(
    formula=control_flow_inst_prop~condition,
    data=filter(solutions_inst_exec_data),
    exact=FALSE,
    conf.int=TRUE)
)
```

```
##
```

```
##  Wilcoxon rank sum test with continuity correction
##
## data:  control_flow_inst_prop by condition
## W = 8043, p-value = 0.1127
## alternative hypothesis: true location shift is not equal to 0
## 95 percent confidence interval:
##  -0.011679066  0.001195076
## sample estimates:
## difference in location
##            -0.00543344
```

In case you're curious, here's all categories of instructions:

```
melted <- melt(
  solutions_inst_exec_data,
  variable.name = "inst_type",
  value.name = "inst_type_prop",
  measure.vars=c(
    "math_inst_prop",
    "module_inst_prop",
    "memory_inst_prop",
    "regulation_inst_prop",
    "control_flow_inst_prop",
    "thread_inst_prop",
    "task_inst_prop",
    "nop_inst_prop"
  )
)

ggplot( melted, aes(x=inst_type, y=inst_type_prop, color=condition) ) +
  geom_boxplot() +
  scale_color_discrete(
    name="Condition:",
    breaks=c("memory", "both"),
    labels=c("Regulation-disabled", "Regulation-enabled")
  ) +
  xlab("Instruction type") +
  ylab("Proportion of instructions in execution trace") +
  coord_flip() +
  theme(legend.position="bottom")
```



10.7 Visualizaing an evolved regulatory network

Let's take a closer look at a successful gene regulatory network.

```
trace_id <- 25392
```

Specifically, we'll be looking at the solution evolved in run id 2.5392×10^4 (arbitrarily selected).

10.7.1 Evolved regulatory network

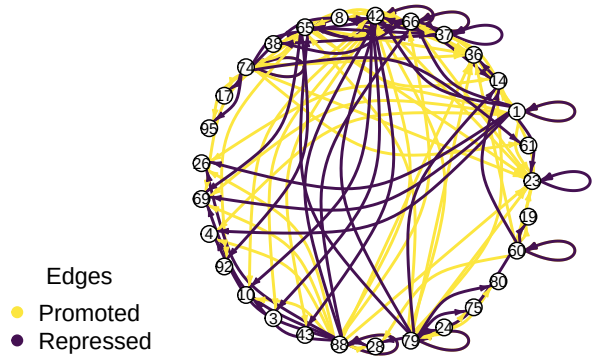
We use the igraph package to draw this program's gene regulatory network.

```
# Networks!
graph_nodes_loc <- paste0(working_directory, "data/igraphs/reg_graph_id-", trace_id, "_
graph_edges_loc <- paste0(working_directory, "data/igraphs/reg_graph_id-", trace_id, "_
graph_nodes_data <- read.csv(graph_nodes_loc, na.strings="NONE")
graph_edges_data <- read.csv(graph_edges_loc, na.strings="NONE")

network <- graph_from_data_frame(
  d=graph_edges_data,
  vertices=graph_nodes_data,
  directed=TRUE
)
```

```
# Setup edge styling
E(network)$color[E(network)$type == "promote"] <- "#FCE640"
E(network)$lty[E(network)$type == "promote"] <- 1
E(network)$color[E(network)$type == "repress"] <- "#441152"
E(network)$lty[E(network)$type == "repress"] <- 1

network_out_name <- paste0(working_directory, "imgs/case-study-id-", trace_id, "-network.svg")

draw_network <- function(net, write_out, out_name) {
  if (write_out) {
    svg(out_name, width=4,height=1.5)
    # bottom, left, top, right
    par(mar=c(0.2,0,1,0.5))
  }
  plot(
    net,
    edge.arrow.size=0.4,
    edge.arrow.width=0.75,
    edge.width=2,
    vertex.size=10,
    vertex.label.cex=0.65,
    curved=TRUE,
    vertex.color="grey99",
    vertex.label.color="black",
    vertex.label.family="sans",
    layout=layout.circle(net)
  )
  legend(
    x = "bottomleft",      ## position, also takes x,y coordinates
    legend = c("Promoted", "Repressed"),
    pch = 19,              ## legend symbols see ?points
    col = c("#FCE640", "#441152"),
    bty = "n",
    border="black",
    xpd=TRUE,
    title = "Edges"
  )
  if (write_out) {
    dev.flush()
    dev.off()
  }
}

draw_network(network, TRUE, network_out_name)
```

```
## pdf
##   2
```

```
draw_network(network, FALSE, "")
```



Bibliography

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., 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., Moreno, M. A., and Ofria, C. (2020). Tag-based genetic regulation for genetic programming.

Spector, L., Martin, B., Harrington, K., and Helmuth, T. (2011). 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.