Analyses for Tag-based Genetic Regulation for Genetic Programming

Alexander Lalejini

2020-12-08

Contents

Chapter 1

Test

Chapter 2

SignalGP Digital Organisms

Here, we give more details on the setup of the SignalGP digital organisms used in the diagnostic experiments. For a broad overview of SignalGP, see (Lalejini and Ofria, 2018). For specific parameter choices, see experiment-specific configuration descriptions (TODO - link here).

For DISHTINY SignalGP details, see DISHTINY docs.

Navigation

2.1 Memory model

SignalGP digital organisms 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 (`GlobalToWorking` or `WorkingToGlobal`) to access it.

These are described in more detail in (Lalejini and Ofria, 2018).

Memory buffers are implemented as integer => double maps.

2.2 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)

2.3 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
- 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.

2.3.1 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]
`TestEqu`	3	Reg[2] = Reg[0] == Reg[1]
`TestNEqu`	3	Reg[2] = Reg[0] != Reg[1]

Instruction	Arguments Used	Description
`TestLess`	3	Reg[2] = Reg[0] < Reg[1]
`TestLessEqu`	3	Reg[2] = Reg[0] <= Reg[1]
`TestGreater`	3	Reg[2] = Reg[0] > Reg[1]
`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.

Instruction	Arguments Used	Description
`Routine`	0	Same as call, but local memory is shared. Sort of like a jump that will jump back when the routine ends.
`Return`	0	Return from the current function call.

2.3.2 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]

2.3.3 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

2.3.4 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 digital organism. Each of these response instructions set a flag on the virtual hardware indicating which response the organism expressed and reset all executing threads such that only function regulation and global memory contents persist.

2.4 References

Lalejini, A., & Ofria, C. (2018). Evolving event-driven programs with SignalGP. Proceedings of the Genetic and Evolutionary Computation Conference on - GECCO '18, 1135–1142. https://doi.org/10.1145/3205455.3205523

Lalejini, A., Wiser, M. J., & Ofria, C. (2017). Gene duplications drive the evolution of complex traits and regulation. Proceedings of the 14th European Conference on Artificial Life ECAL 2017, 257–264. https://doi.org/10.7551/ecal_a_045

Chapter 3

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 (link coming).

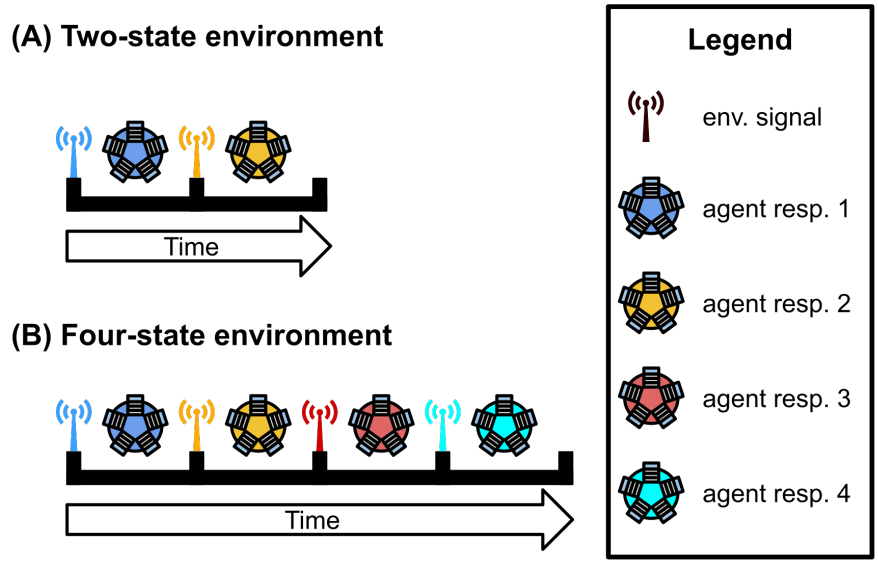
Please file an issue or make a pull request on github to report any mistakes, ask questions, request more explanation, et cetera.

3.1 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)

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 organism 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 organisms have access to genetic regulation.

2. a regulation-disabled treatment where organisms 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.**

3.2 Analysis Dependencies

Load all required R libraries.

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

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
```

3.3 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")
```

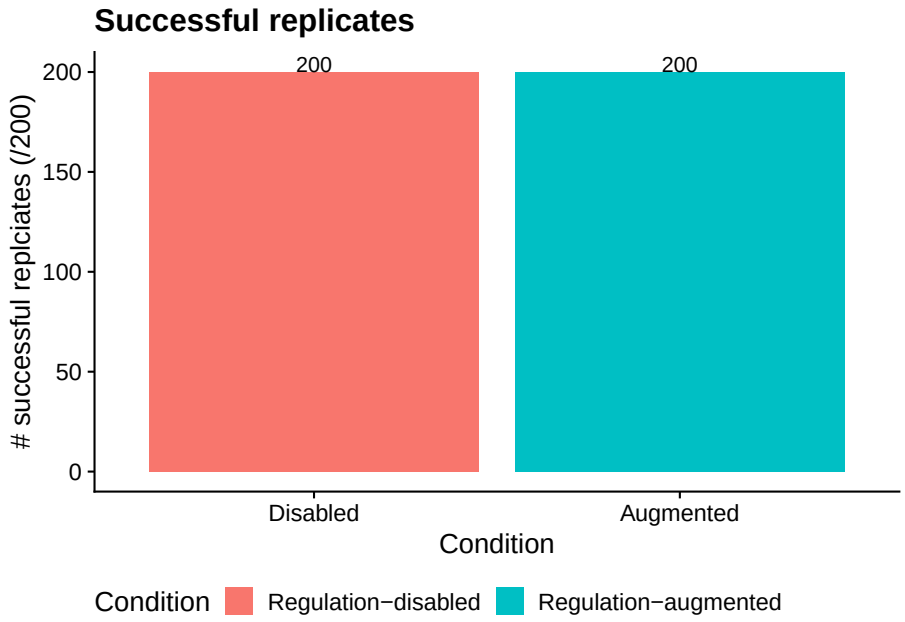
```
# Specify factors (not all of these matter for this set of runs).
data$matchbin_thresh <-
  factor(data$matchbin_thresh,
         levels=c(0, 25, 50, 75))
data$NUM_ENV_STATES <-
  factor(data$NUM_ENV_STATES,
         levels=c(2, 4, 8, 16, 32))
data$NUM_ENV_UPDATES <-
  factor(data$NUM_ENV_UPDATES,
         levels=c(2, 4, 8, 16, 32))
data$TAG_LEN <-
  factor(data$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.
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"
)

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

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

3.4 Does regulation hinder the evolution of successful genotypes?

Here, we look at the number of solutions evolved with access and without access to genetic regulation. An organism is categorized as a 'solution' if it can correctly respond in 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("Disabled",
                            "Augmented")) +
  scale_fill_discrete(name="Condition",
                      breaks=c("memory", "both"),
                      labels=c("Regulation-disabled",
                               "Regulation-augmented")) +
  ylab("# successful replciates (/200)") +
  theme(legend.position = "bottom") +
  ggtitle("Successful replicates")
```

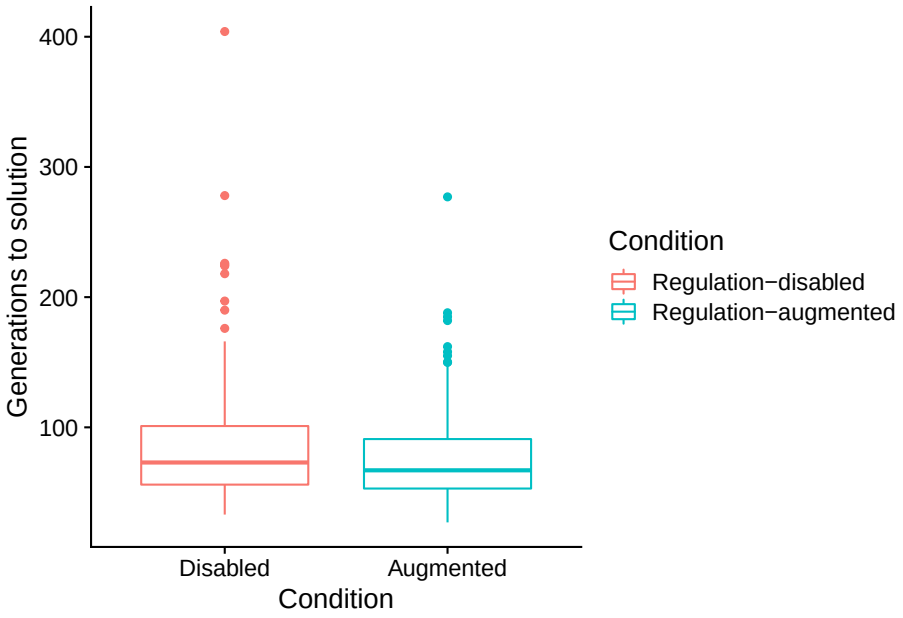


Programs capable of achieving a perfect score on the changing signal task (for a given sequence of environment signals) evolve in all 200 replicates of each condition (i.e., with and without access to genetic regulation). These programs, however, do not necessarily generalize across all possible sequences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

3.4.1 Does access to regulation slow adaptation?

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

```
ggplot(data, aes(x=condition, y=update, color=condition)) +
  geom_boxplot() +
  scale_x_discrete(name="Condition",
                   breaks=c("memory","both"),
                   labels=c("Disabled",
                            "Augmented")) +
  scale_color_discrete(name="Condition",
                       breaks=c("memory", "both"),
                       labels=c("Regulation-disabled",
                                "Regulation-augmented")) +
  ylab("Generations to solution")
```




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

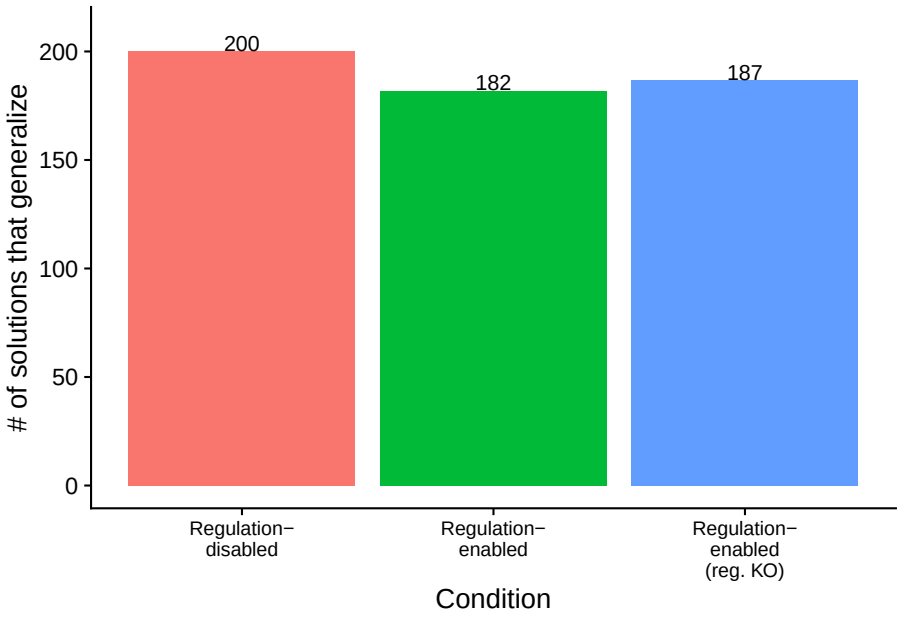
```
##
##  Wilcoxon rank sum test with continuity correction
##
## data:  update by condition
## W = 22188, p-value = 0.05845
## alternative hypothesis: true location shift is not equal to 0
## 95 percent confidence interval:
##  -3.860236e-05  1.000000e+01
## sample estimates:
## difference in location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

We find no evidence (Wilcoxon rank-sum test) that access to regulation resulted in slower adaptation.

3.4.2 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 organisms as having successfully generalized only if they responded correctly in all 5000 tests. Additionally, we measured generalization in the regulation-augmented condition with regulation knocked out.

```
# 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=="1")$SEED)

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

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_fill_discrete(name="Condition",
  #                     limits=c("mem_generalize",
  #                              "reg_generalize",
  #                              "reg_generalize_ko_reg"),
  #                     labels=c("Regulation-disabled",
  #                              "Regulation-augmented",
  #                              "Regulation-augmented\n(reg. knockout)")) +
  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,heigh
```



All programs evolved without access to regulation successfully generalized. However, 18 out of 200 solutions evolved with access to regulation failed to generalize (statistically significant, Fisher's exact test).

```
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")
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
```

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.