Tag-accessed memory data analyses

Contents

Overview

Here, we analyze our experimental results comparing tag-accessed memory to more traditional, direct-indexed memory. Specifically, we conducted a series of experiments using simple linear-GP representations on program synthesis problems. In all experiments, we evolved genetic programs that used tag-accessed memory and programs that used direct-indexed memory. Aside from how programs were allowed to access memory, both genetic programming systems/representations were identical (*e.g.*, had identical sets of instructions).

First, we present data from our preliminary experiments, which used a range of numeric argument and tag-based argument mutation rates. Based on these preliminary data, we selected a single mutation rate to use for mutating tags and numeric arguments and ran another set of experiments with increased replication to improve our statistical power.

This document was generated using R markdown with R version 3.3.2 (2016-10-31) (R Core Team, 2016).

Analysis Setup

First, we'll load our R packages.

```
library(tidyr)    # (Wickham & Henry, 2018)
library(ggplot2)  # (Wickham, 2009)
library(plyr)     # (Wickham, 2011)
library(dplyr)    # (Wickham et al., 2018)
library(cowplot)  # (Wilke, 2018)
```

Data Loading

Here, we load data from both our preliminary experiments and our second set of final experiments.

Set path information. Note, this path information is accurate for the directory structure used in our Git repository (LINK ANONYMIZED).

```
# Load data from preliminary runs.
prelim_u500_summary_loc <-
  "../data/prelim-results/min_programs__update_500__solutions_summary.csv"
prelim_u1000_summary_loc <-
  "../data/prelim-results/min_programs__update_1000__solutions_summary.csv"
# Load data for final experiment runs.
solutions_u300_data_loc <-
  "../data/exp/min_programs__update_300.csv"
solutions_u300_summary_data_loc <-
  "../data/exp/min_programs__update_300__solutions_summary.csv"
```

Load data in from file.

```
# Load preliminary experiment data.
prelim_u500_summary <- read.csv(prelim_u500_summary_loc, na.strings = "NONE")
prelim_u1000_summary <- read.csv(prelim_u1000_summary_loc, na.strings = "NONE")
# Load final experiment data.
prog_solutions_u300 <- read.csv(solutions_u300_data_loc, na.strings = "NONE")
prog_solutions_u300_summary <- read.csv(solutions_u300_summary_data_loc,
    na.strings = "NONE")
```

Preliminary Experiments

We ran a set of preliminary experiments, applying our simple linear GP representation (with tag-based memory *and* direct-indexed memory) to 5 problems from the general program synthesis benchmark suite (Helmuth & Spector, 2015): for loop index, grade, median, small or large, and smallest.

We tried several tag-argument and numeric-argument mutation rates in our preliminary runs. For runs that used tag-based arguments, we tried the following per-bit tag-argument mutation rates: 0.001, 0.005, 0.01. For runs that used numeric arguments, we tried the followign per-argument mutation rates: 0.001, 0.005, 0.01.

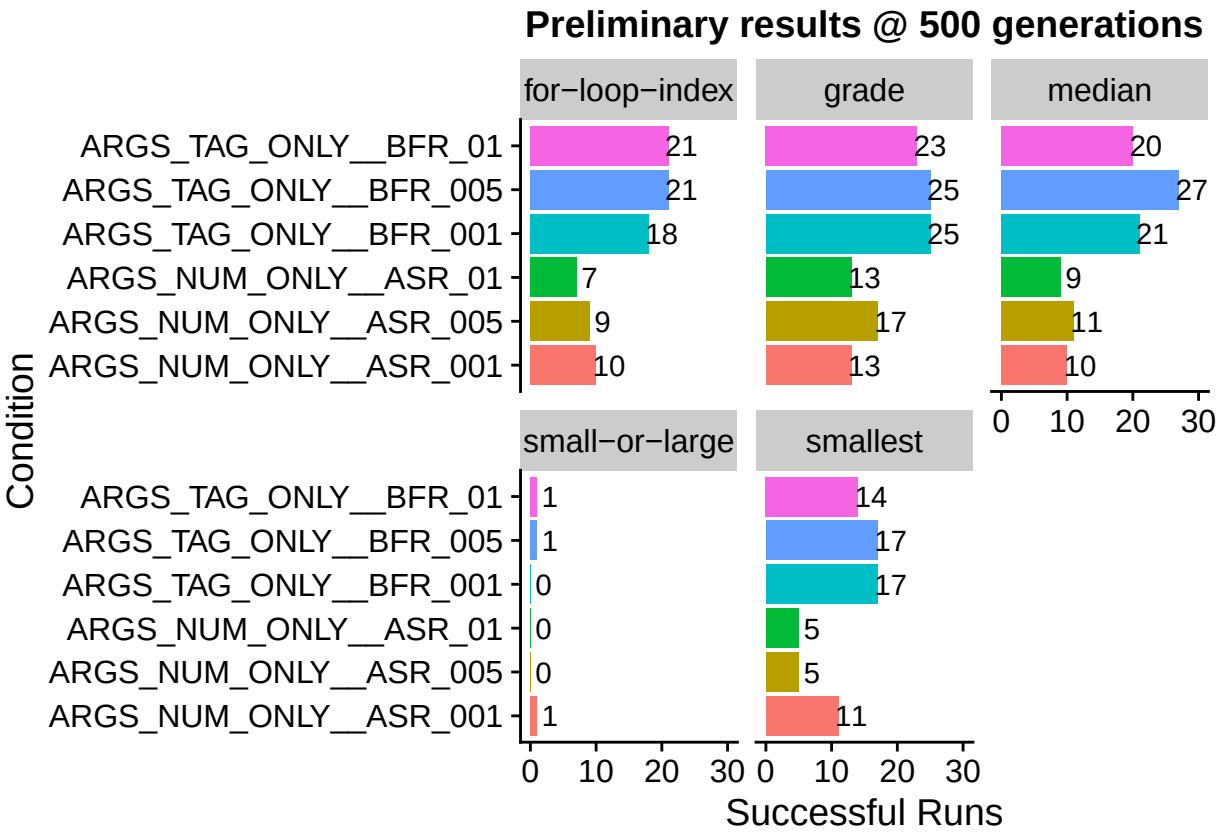
We ran these experiments for 1,000 generations (we'll see data after 500 generations of evolution and 1,000 generations of evolution). For each problem, we looked at the proportion of runs (30 replicates per condition) that produced solutions.

Successful Runs After 500 Generations of Evolution

Note, BFR and ASR in the condition refer to bit-flip rates (BFR) and argument substitution rate (ASR); the number that follows (after the '_') should be read as follows:

- _01: 0.01
- _005: 0.005
- _001: 0.001

ARGS_TAG_ONLY in the condition name means that condition used tag-accessed memory. ARGS_NUM_ONLY means that condition used direct-indexed memory.

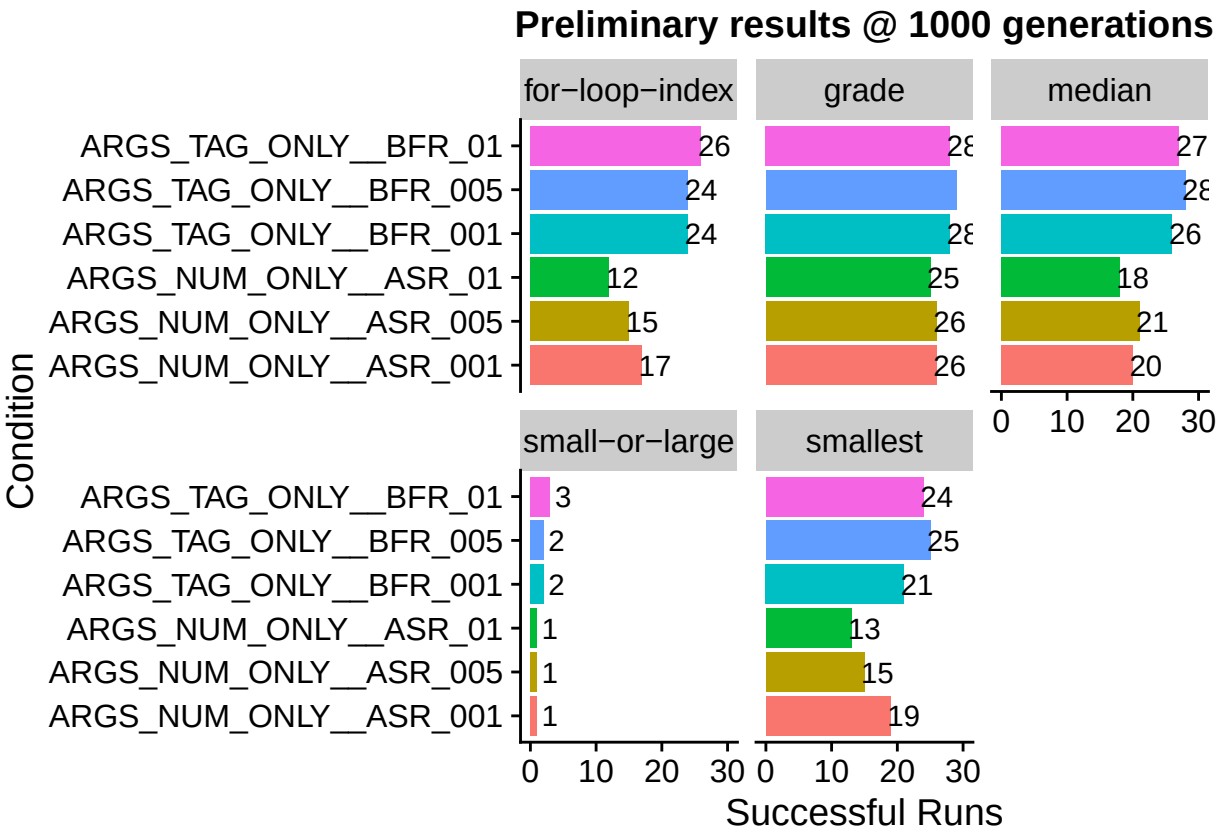


Successful Runs After 1,000 Generations of Evolution

Note, BFR and ASR in the condition refer to bit-flip rates (BFR) and argument substitution rate (ASR); the number that follows (after the '_') should be read as follows:

- _01: 0.01
- _005: 0.005
- _001: 0.001

ARGS_TAG_ONLY in the condition name means that condition used tag-accessed memory. ARGS_NUM_ONLY means that condition used direct-indexed memory.



Takeaways from preliminary results

Results are pretty robust across mutation rates. 0.01 seems like the worst for numeric arguments, so we did not go forward with it. There's not much difference between the 0.001 and 0.005 mutation rates. Thus, in our final set of experiments, we used 0.005 as our mutation rate (for both numeric arguments and tag-based arguments).

Small or large is more challenging than the other four problems: 1,000 generations was not enough to differentiate conditions. Because our goal here is to compare tag-accessed memory with direct-indexed memory, we dropped the small or large problem and instead used the easier (Helmuth & Spector, 2015) number IO problem.

Other than the small or large problem, 1,000 generations of evolution was more than enough for all conditions to produce solutions, and for some problems performance seems to have saturated (*i.e.*, 1,000 generations is enough time for almost all runs to find solutions). Thus, in our final set of experiments, we limited evolution to just 300 generations, which is closer to the number of generations used in (Helmuth & Spector, 2015).

Final Experiments

In our second (final) set of runs, we applied our two linear GP representations (one with tag-accessed memory and one with direct-indexed memory) to five problems: number IO, for loop index, grade, median, and smallest.

Because number IO is much easier than the other four problems, we ran it for 100 generations of evolution. We ran all other problems for 300 generations. Because our preminary results were fairly consistent across mutation rates (with 0.01 being too high for numeric arguments), we only used argument mutation rates

of 0.005. In future (longer than 2 page extended abstract) work, we will do a wider, finer grained sweep of mutation rates.

For each problem, we compared the number of successful runs (*i.e.*, runs that produced a perfect solution) for both memory treatments using Fisher's exact test.

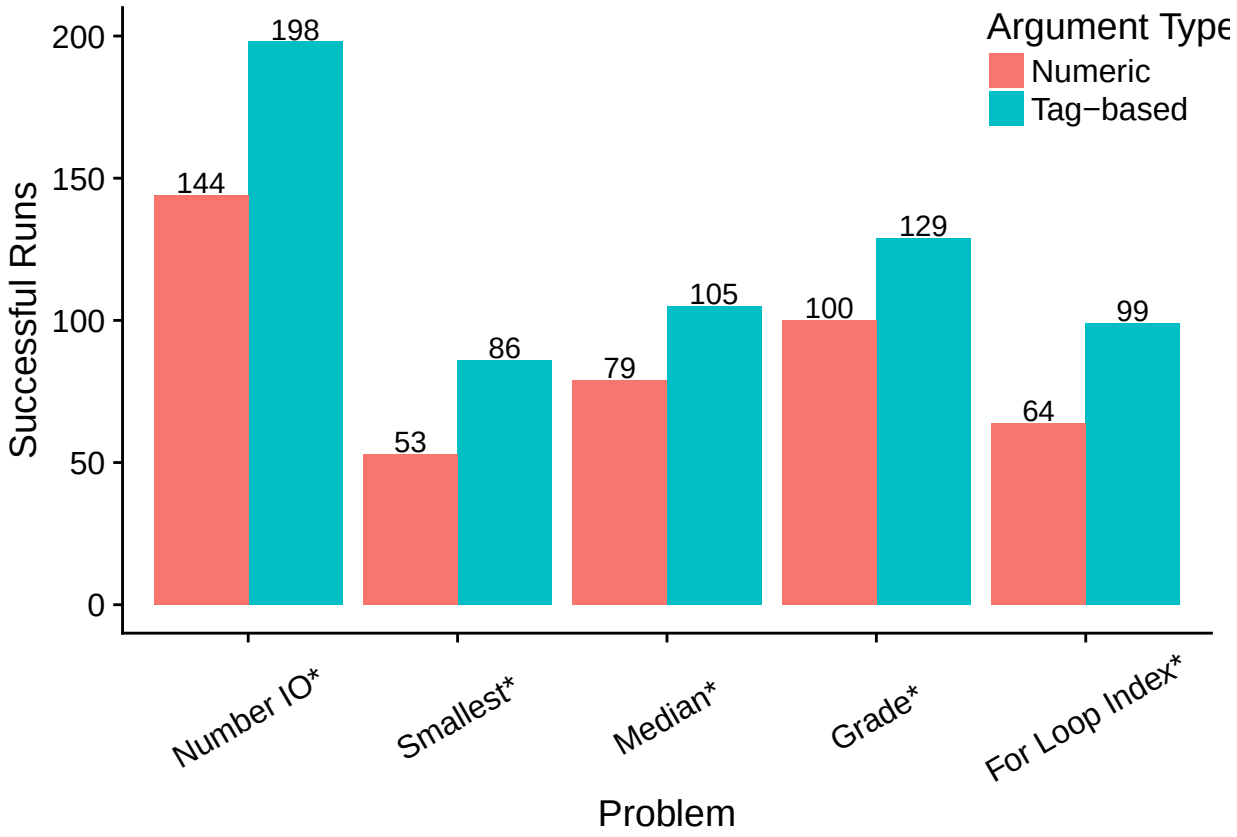
Successful Runs After 300 Generations of Evolution

We'll make this graph prettier than the graphs for our preliminary data.

```
# We'll make this one a pretty one.
prog_solutions_u300_summary$problem <- factor(prog_solutions_u300_summary$problem,
                                              levels=c("number-io",
                                                       "smallest",
                                                       "median",
                                                       "grade",
                                                       "for-loop-index"))

# Weird spacing to make fit in PDF output
ggplot(data = filter(prog_solutions_u300_summary,
                     arg_mut_rate=="0.005"),
       mapping=aes(x=problem,
                   y=solutions_found,
                   fill=arg_type,
                   label=solutions_found)) +
    geom_bar(stat="identity", position="dodge") +
    scale_x_discrete(labels=c("Number IO*",
                              "Smallest*",
                              "Median*",
                              "Grade*",
                              "For Loop Index*")) +
    xlab("Problem") +
    ylab("Successful Runs")  +
    ylim(0, 200) +
    guides(fill=guide_legend(title="Argument Type")) +
    geom_text(position=position_dodge(width=0.9), vjust=-0.1) +
    theme(axis.text.x = element_text(angle=30, vjust=0.5)) +
    theme(legend.position = c(0.78, 0.9)) +
    ggsave("successful_runs.pdf")
```

```
## Saving 6.5 x 4.5 in image
```



Statistical Analysis

For each problem, we used Fisher's exact test to compare the success rates of runs using tag-based memory and runs using direct-indexed memory.

Problem - Number IO

Reminder, this is actually at generation 100 for number IO runs (despite the file being labeled generation 300).

```
prob_data <- filter(prog_solutions_u300_summary,
                    problem=="number-io" & arg_type!="Both" & arg_mut_rate=="0.005")
contingency_table <- matrix(data=c(prob_data$solutions_found,
                                   prob_data$total_runs - prob_data$solutions_found),
                            nrow=length(prob_data$solutions_found))
fisher.test(contingency_table)
```

```
##
##  Fisher's Exact Test for Count Data
##
## data:  contingency_table
## p-value < 2.2e-16
## alternative hypothesis: true odds ratio is not equal to 1
## 95 percent confidence interval:
##  0.003042058 0.101777565
## sample estimates:
## odds ratio
```

```
## 0.02613986
```

Problem - Smallest

```
prob_data <- filter(prog_solutions_u300_summary,
                    problem=="smallest" & arg_type!="Both" & arg_mut_rate=="0.005")
contingency_table <- matrix(data=c(prob_data$solutions_found,
                                   prob_data$total_runs - prob_data$solutions_found),
                            nrow=length(prob_data$solutions_found))
fisher.test(contingency_table)
```

```
##
##  Fisher's Exact Test for Count Data
##
## data:  contingency_table
## p-value = 0.0007505
## alternative hypothesis: true odds ratio is not equal to 1
## 95 percent confidence interval:
##  0.3065287 0.7431869
## sample estimates:
## odds ratio
##  0.4788245
```

Problem - Median

```
prob_data <- filter(prog_solutions_u300_summary,
                    problem=="median" & arg_type!="Both" & arg_mut_rate=="0.005")
contingency_table <- matrix(data=c(prob_data$solutions_found,
                                   prob_data$total_runs - prob_data$solutions_found),
                            nrow=length(prob_data$solutions_found))
fisher.test(contingency_table)
```

```
##
##  Fisher's Exact Test for Count Data
##
## data:  contingency_table
## p-value = 0.01204
## alternative hypothesis: true odds ratio is not equal to 1
## 95 percent confidence interval:
##  0.3894888 0.8955221
## sample estimates:
## odds ratio
##  0.5914876
```

Problem - Grade

```
prob_data <- filter(prog_solutions_u300_summary,
                    problem=="grade" & arg_type!="Both" & arg_mut_rate=="0.005")
contingency_table <- matrix(data=c(prob_data$solutions_found,
                                   prob_data$total_runs - prob_data$solutions_found),
                            nrow=length(prob_data$solutions_found))
fisher.test(contingency_table)
```

```
##
```

```
##  Fisher's Exact Test for Count Data
##
## data:  contingency_table
## p-value = 0.004591
## alternative hypothesis: true odds ratio is not equal to 1
## 95 percent confidence interval:
##  0.3611841 0.8380632
## sample estimates:
## odds ratio
##  0.5512133
```

Problem - For Loop Index

```
prob_data <- filter(prog_solutions_u300_summary,
                    problem=="for-loop-index" & arg_type!="Both" & arg_mut_rate=="0.005")
contingency_table <- matrix(data=c(prob_data$solutions_found,
                                   prob_data$total_runs - prob_data$solutions_found),
                            nrow=length(prob_data$solutions_found))
fisher.test(contingency_table)
```

```
##
##  Fisher's Exact Test for Count Data
##
## data:  contingency_table
## p-value = 0.0005227
## alternative hypothesis: true odds ratio is not equal to 1
## 95 percent confidence interval:
##  0.3131273 0.7352627
## sample estimates:
## odds ratio
##  0.4809916
```

References

Claus O. Wilke (2018). cowplot: Streamlined Plot Theme and Plot Annotations for 'ggplot2'. R package version 0.9.3. https://CRAN.R-project.org/package=cowplot

Helmuth, T., & Spector, L. (2015). General Program Synthesis Benchmark Suite. In Proceedings of the 2015 on Genetic and Evolutionary Computation Conference - GECCO '15 (pp. 1039–1046). New York, New York, USA: ACM Press. https://doi.org/10.1145/2739480.2754769

R Core Team (2016). R: A language and environment for statistical computing. R Foundation for Statistical Computing, Vienna, Austria. URL https://www.R-project.org/.

H. Wickham. ggplot2: Elegant Graphics for Data Analysis. Springer-Verlag New York, 2009.

Hadley Wickham and Lionel Henry (2018). tidyr: Easily Tidy Data with 'spread()' and 'gather()' Functions. R package version 0.8.1. https://CRAN.R-project.org/package=tidyr

Hadley Wickham (2011). The Split-Apply-Combine Strategy for Data Analysis. Journal of Statistical Software, 40(1), 1-29. URL http://www.jstatsoft.org/v40/i01/.

Hadley Wickham, Romain François, Lionel Henry and Kirill Müller (2018). dplyr: A Grammar of Data Manipulation. R package version 0.7.5. https://CRAN.R-project.org/package=dplyr