
Genetic Programming System

Here, we provide a more detailed description of the linear GP system we used in this work. This document also reports configuration details used in this work. Our exact configuration files along with experiment source code can be found in our online GitHub repository: <https://github.com/amlalejini/GECCO-2019-tag-accessed-memory>.

Contents

- GP Representation
 - Programs
 - Virtual CPU
 - Tag-accessed Memory
 - Direct-indexed Memory
- Instruction Set
 - Default Instructions
 - Problem-specific Instructions
 - * Problem - Number IO
 - * Problem - For Loop Index
 - * Problem - Grade
 - * Problem - Median
 - * Problem - Smallest
- Experiment Configuration Details
 - Configuration Details - Number IO
 - Configuration Details - For loop index
 - Configuration Details - Grade
 - Configuration Details - Median
 - Configuration Details - Smallest
- References

GP Representation

Programs

Programs are linear sequences of instructions, and each instruction has three arguments that may modify its behavior. Our instruction set supports basic computations (*e.g.*, addition, subtraction, multiplication, *etc.*) and allows programs to control the flow of execution (*e.g.*, conditional branching, looping, *etc.*).

Virtual CPU

Programs are executed in the context of a simple virtual CPU, which manages the flow of execution (e.g., looping, current instruction, etc.) and gives programs access to 16 memory registers used for storing data and for performing computations.

Tag-accessed Memory

Many traditional GP systems that give genetic programs access to memory (e.g., indexable memory registers) use rigid naming schemes where memory is numerically indexed, and mutation operators must guarantee the validity of memory-referencing instructions. Tag-accessed memory allows programs to use tag-based referencing to index into memory registers. Tags are evolvable labels that give genetic programs a flexible mechanism for specification. Tags allow for *inexact* referencing; a referring tag references the *closest matching* referent. To facilitate inexact referencing, the similarity (or dissimilarity) between any two tags must be quantifiable; thus, a referring tag can *always* reference the closest matching referent tag. This ensures that all possible tags are valid references. When using a tag-accessed memory model, each of the 16 memory registers in the virtual CPU are statically tagged with length-16 bit strings. Tags used for memory registers were generated using the Hadamard matrix and were as follows:

- Register 0: 1111111111111111
- Register 1: 0101010101010101
- Register 2: 0011001100110011
- Register 3: 1001100110011001
- Register 4: 0000111100001111
- Register 5: 1010010110100101
- Register 6: 1100001111000011
- Register 7: 0110100101101001
- Register 8: 0000000011111111
- Register 9: 1010101001010101
- Register 10: 1100110000110011
- Register 11: 0110011010011001
- Register 12: 1111000000001111
- Register 13: 0101101010100101
- Register 14: 0011110011000011
- Register 15: 1001011001101001

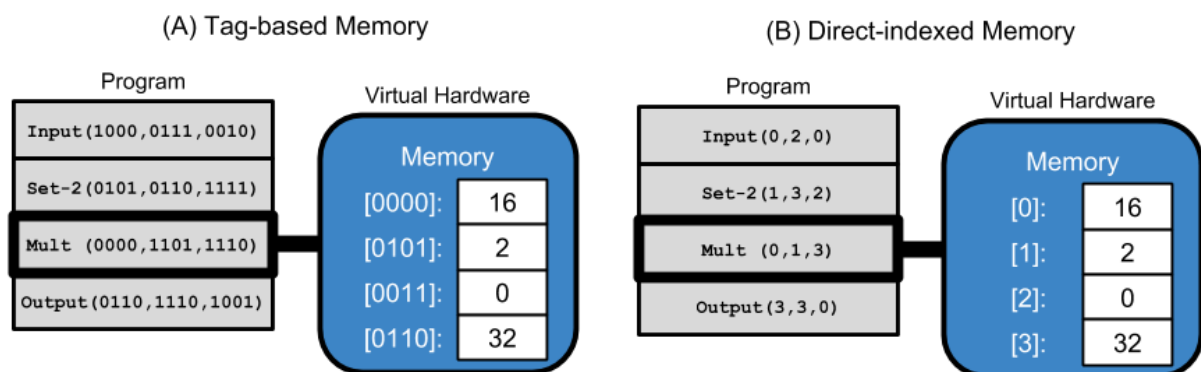
Each program instruction has three *tag* arguments (i.e., each instruction argument is a length-16 bit string). Tag-based instruction arguments reference the memory position with the closest matching

tag; as such, argument tags need not *exactly* match any of the tags with memory positions.

Direct-indexed Memory

Direct-indexed memory is the traditional form of memory access in linear GP. Each program instruction has three numeric arguments (0 through 15) that are used to directly specify memory registers.

Below is a cartoon contrasting tag-accessed memory with direct accessed memory.



Instruction Set

We used identical instruction sets in both memory model conditions (tag-accessed and direct-indexed). However, in conditions using the tag-accessed memory, instructions used tag arguments that used tag-based referencing to index into the virtual CPU's memory registers, and in conditions using the direct-indexed memory, instructions used numeric arguments that directly indexed into the virtual CPU's memory registers.

Below, we describe our default instruction set (used across all problems), and all problem-specific instructions.

Default Instructions

Note:

- EOP: End of program
- Reg: 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.

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

Instruction	Arguments Used	Description
Add	3	$\text{Reg}[2] = \text{Reg}[0] + \text{Reg}[1]$
Sub	3	$\text{Reg}[2] = \text{Reg}[0] - \text{Reg}[1]$
Mult	3	$\text{Reg}[2] = \text{Reg}[0] * \text{Reg}[1]$
Div	3	$\text{Reg}[2] = \text{Reg}[0] / \text{Reg}[1]$
Mod	3	$\text{Reg}[2] = \text{Reg}[0] \% \text{Reg}[1]$
TestNumEqu	3	$\text{Reg}[2] = \text{Reg}[0] == \text{Reg}[1]$
TestNumNEqu	3	$\text{Reg}[2] = \text{Reg}[0] != \text{Reg}[1]$
TestNumLess	3	$\text{Reg}[2] = \text{Reg}[0] < \text{Reg}[1]$
TestNumLessTEqu	3	$\text{Reg}[2] = \text{Reg}[0] <= \text{Reg}[1]$
TestNumGreater	3	$\text{Reg}[2] = \text{Reg}[0] > \text{Reg}[1]$
TestNumGreaterTEqu	3	$\text{Reg}[2] = \text{Reg}[0] >= \text{Reg}[1]$
Floor	1	Floor(Reg[0])
Not	1	$\text{Reg}[0] = !\text{Reg}[0]$
Inc	1	$\text{Reg}[0] = \text{Reg}[0] + 1$
Dec	1	$\text{Reg}[0] = \text{Reg}[0] - 1$
CopyMem	2	$\text{Reg}[0] = \text{Reg}[1]$
SwapMem	2	Swap(Reg[0], Reg[1])
If	1	If Reg[0] != 0, proceed. Otherwise skip to the next C\close or EOP.
IfNot	1	If Reg[0] == 0, proceed. Otherwise skip to the next C\close or EOP.
While	1	While Reg[0] != 0, loop. Otherwise skip to next C\close or EOP.
Countdown	1	Same as While, but decrement Reg[0] if Reg[0] != 0.
C\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).

Instruction	Arguments Used	Description
Return	0	Return from program execution (exit program execution).
Set-0	1	Reg[0] = 0
Set-1	1	Reg[0] = 1
Set-2	1	Reg[0] = 2
Set-3	1	Reg[0] = 3
Set-4	1	Reg[0] = 4
Set-5	1	Reg[0] = 5
Set-6	1	Reg[0] = 6
Set-7	1	Reg[0] = 7
Set-8	1	Reg[0] = 8
Set-9	1	Reg[0] = 9
Set-10	1	Reg[0] = 10
Set-11	1	Reg[0] = 11
Set-12	1	Reg[0] = 12
Set-13	1	Reg[0] = 13
Set-14	1	Reg[0] = 14
Set-15	1	Reg[0] = 15
Set-16	1	Reg[0] = 16

Problem-specific Instructions

Problem - Number IO

Instruction	# Arguments Used	Description
LoadInt	1	Reg[0] = integer input
LoadDouble	1	Reg[0] = double input
SubmitNum	1	Output Reg[0]

Problem - For Loop Index

Instruction	# Arguments Used	Description
LoadStart	1	Reg[0] = start input
LoadEnd	1	Reg[0] = end input
LoadStep	1	Reg[0] = step input
SubmitNum	1	Output Reg[0]

Problem - Grade

Instruction	# Arguments Used	Description
SubmitA	0	Classify grade as "A"
SubmitB	0	Classify grade as "B"
SubmitC	0	Classify grade as "C"
SubmitD	0	Classify grade as "D"
SubmitF	0	Classify grade as "F"
LoadThreshA	1	Reg[0] = input threshold for "A"
LoadThreshB	1	Reg[0] = input threshold for "B"
LoadThreshC	1	Reg[0] = input threshold for "C"
LoadThreshD	1	Reg[0] = input threshold for "D"
LoadGrade	1	Reg[0] = input grade to classify

Problem - Median

Instruction	# Arguments Used	Description
LoadNum1	1	Reg[0] = input 1
LoadNum2	1	Reg[0] = input 2
LoadNum3	1	Reg[0] = input 3
SubmitNum	1	Output Reg[0]

Problem - Smallest

Instruction	# Arguments Used	Description
LoadNum1	1	Reg[0] = input 1
LoadNum2	1	Reg[0] = input 2
LoadNum3	1	Reg[0] = input 3
LoadNum4	1	Reg[0] = input 4
SubmitNum	1	Output Reg[0]

Experiment Configuration Details

Here, we discuss only the configuration details for the experiments reported in our extended abstract. For details on preliminary experiments, see our data analysis supplemental material.

We used the lexicase parent selection algorithm to solve five problems from Helmuth and Spector’s general program synthesis benchmark suite (Helmuth and Spector, 2015): number IO, smallest, median, grade, and for loop index. We used identical training and testing sets as in (Helmuth and Spector, 2015). Refer to (Helmuth and Spector, 2015) for more details about these problems.

For each problem, we evolved 50 replicate populations of 500 individuals at a range of mutation rate conditions. In all but the number IO problem, we evolved programs for 500 generations. Because number IO is substantially easier than each of the other problems (Helmuth and Spector, 2015), we only evolved these programs for 100 generations.

We propagated programs asexually and applied mutations to offspring. We applied single-instruction insertions, deletions, and substitutions at a per-instruction rate of 0.005 each and multi-instruction sequence duplications and deletions at a per-program rate of 0.05. Because the relative performance of these two techniques depends heavily on the chosen mutation operations and rates, we used a wide range of argument (numeric and tag) mutation rates for each technique. We mutated tag-based arguments at the following per-bit rates: 0.0001, 0.001, 0.0025, 0.005, 0.0075, 0.01, 0.025, 0.05, 0.075, and 0.1. We mutated numeric arguments at the following per-argument rates: 0.0001, 0.001, 0.0025, 0.005, 0.0075, 0.01, 0.025, 0.05, 0.075, and 0.1.

Each problem is defined by a set of test cases in which programs are given specified input data and are scored on how close their output is to the correct output; depending on the problem, we measured scores either on a gradient or on a binary pass-fail basis. During an evaluation, we limited the total number of instructions a program could execute; this limit varied by problem. Programs were not required to stop on their own as long as they output their results before reaching their execution limit.

Below we give the configuration details specific to each problem.

Configuration Details - Number IO

- Maximum allowed program length: 32
- Maximum number of instruction-execution steps: 32
- Test scores were measured on a pass/fail basis.
- Generations: 100

Configuration Details - For loop index

- Maximum allowed program length: 128
- Maximum number of instruction-execution steps: 256
- Test scores were measured on a gradient, using the Levenshtein distance between the program's output and the correct output sequence.
- Generations: 500

Configuration Details - Grade

- Maximum allowed program length: 128
- Maximum number of instruction-execution steps: 128
- Test scores were measured on a pass/fail basis.
- Generations: 500

Configuration Details - Median

- Maximum allowed program length: 64
- Maximum number of instruction-execution steps: 64
- Test scores were measured on a pass/fail basis.
- Generations: 500

Configuration Details - Smallest

- Maximum allowed program length: 64
- Maximum number of instruction-execution steps: 64
- Test scores were measured on a pass/fail basis.
- Generations: 500

References

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>