Appears in 1997 Conference on Genetic Programming
This paper contains minor revisions to the published article.

Learning Recursive Sequences via Evolution of Machine-Language Programs

Lorenz Huelsbergen
lorenz@research.bell-labs.com

Bell Laboratories, Lucent Technologies
600 Mountain Avenue, Murray Hill, NJ 07974

Abstract

We use directed search techniques in the space of computer programs to learn recursive sequences of positive integers. Specifically, the integer sequences of squares, x^2; cubes, x^3; factorial, $x!$; and Fibonacci numbers are studied. Given a small finite prefix of a sequence, we show that three directed searches—machine-language genetic programming with crossover, exhaustive iterative hill climbing, and a hybrid (crossover and hill climbing)—can automatically discover programs that exactly reproduce the finite target prefix and, moreover, that correctly produce the remaining sequence up to the underlying machine's precision.

Our machine-language representation is generic—it contains instructions for arithmetic, register manipulation and comparison, and control flow. We also introduce an output instruction that allows variable-length sequences as result values. Importantly, this representation does not contain recursive operators; recursion, when needed, is automatically synthesized from primitive instructions.

For a fixed set of search parameters (*e.g.*, instruction set, program size, fitness criteria), we compare the efficiencies of the three directed search techniques on the four sequence problems. For this parameter set, an evolutionary-based search always outperforms exhaustive hill climbing as well as undirected random search. Since only the prefix of the target sequence is variable in our experiments, we posit that this approach to sequence induction is potentially quite general.

1 Introduction

Given a finite prefix of a perhaps unknown sequence of values, it is tempting to ask "What is the next, yet unseen, value in the sequence?" If we have reason to suspect that the sequence is generated by a computable function, one can attempt to answer this question, perhaps by inspection of, and induction on, the given prefix. This leads to the more general query: "What algorithm produces this sequence?" In this paper we show empirically that—for *some* non-trivial integer sequences—directed searches in the space of computer programs can automatically provide correct answers to the above questions.

We consider four integer sequences: *square*, *cube*, *factorial* and *Fibonacci* numbers. Figure 1 defines the underlying functions. The *factorial* and *Fibonacci* sequences have natural recursive definitions and *square* and *cube* can also—via the binomial theorem—be expressed recursively. As we shall see, the programs evolved to produce the *square* and *cube* sequences utilize recursion. For a function f, we consider the l element prefix $\{f(0),\ldots,f(l-1)\}$ as the *target sequence* for the searches. For the experiments in this paper, l-prefixes of length ten ($l = 10$) suffice to learn the sequences.

We first examine two distinct types of directed search: a global search in the form of machine-language genetic programming and a local search in the form of an exhaustive iterative hill climber. We then consider a hybrid search composed of machine-language genetic programming and hill climbing; it also performs well on this paper's set of sequence problems. Random search, on the other hand, is found to be ineffective.

Machine-language genetic programming (MLGP) [11, 7, 2] is a form of genetic algorithm (GA) [6] closely related to Koza's genetic programming (GP) [8] of Lisp expressions. GAs use principles from evolutionary theory (populations, fitness criteria, recombination) to search large non-linear spaces. GP and MLGP apply such search to the space of computer programs primarily via the *crossover* (XO) genetic operator. Crossover randomly swaps portions of individuals (programs) with

$$\begin{aligned}
square(x) &\equiv x^2 \\
cube(x) &\equiv x^3 \\
factorial(x) &\equiv \begin{cases} 1 & \text{if } x = 0 \\ x * factorial(x-1) & \text{otherwise} \end{cases} \\
Fibonacci(x) &\equiv \begin{cases} 1 & \text{if } x = 0 \text{ or } x = 1 \\ Fibonacci(x-2) + Fibonacci(x-1) & \text{otherwise} \end{cases}
\end{aligned}$$

Figure 1: Functions defining the sequences considered in this paper. Domains and ranges are the natural numbers.

the goal of producing a yet fitter individual. MLGP has roots in work by Friedberg *et al* [4, 3] on evolving programs; such early approaches however did not improve upon random search (*i.e.* guessing).

Koza has used GP to learn the *Fibonacci* function [8] (among many other, albeit non-recursive, functions and programs).[1] To do so, he added a recursive function (called `SRF`) to the set of GP functions. For a program p, `SRF` retains the values produced by p on previous tests. That is, when evaluating the test for $Fibonacci(x_n)$, p's results of prior tests ($Fibonacci(x_i)$ for $i < n$) are available to p through `SRF`. Arguably, inclusion of `SRF` requires *a prior* knowledge of the structure of the solution. Other researchers (*e.g.* [5]) introduce explicit domain-specific sequence instructions[2] into the representation (instruction set) defining the search space. They thus sidestep (as does `SRF`) the task of synthesizing recursion from non-recursive instructions by requiring a human to provide recursion within the set of supplied instructions. Domain-specific instructions, on the other hand, are useful for solving problems outside the current reach of pure machine-language genetic programming, but entail the risk of making the target problem much easier. Thus, domain-specific operators obscure the cause of a problem's solution: is it the search algorithm or the human supplying suitable domain-specific instructions?

Our approach is orthogonal to user-supplied domain-specific instructions since it allows and requires the automatic synthesis of complex (*i.e.* recursive) instruction sequences from only primitive (non-recursive) instructions. In particular, we build on prior work [7] that evolved iteration and control flow using primitive machine instructions. The ability to synthesize arbitrary control flow, coupled with readable and writable memory locations, enables the juxtaposition of non-recursive instructions to yield iterative and recursive behavior.

We restrict our representation to primitive machine instructions on the following grounds. Processors have—in a sense—"evolved" their instruction sets since their inception. Contemporary instruction sets are certainly suitable for solving a variety of general-purpose problems. Furthermore, the instruction sets of different processors share large subsets of instructions with similar functionality: arithmetic, branch-based control flow, register–memory moves, *etc.* Our line of inquiry is to demonstrate that GAs (as MLGP) can utilize primitive machine instructions as building blocks for complex programs.

Our approach is general in that a single instruction set—embodied as a *virtual register machine* (VRM, *cf.* [7]) for *sequences*, called VRM-$\mathcal{S}$—suffices for the four problems under consideration. VRM-$\mathcal{S}$ consists of instructions for basic register manipulation (move, set, clear, increment, decrement), arithmetic (add, subtract, multiply, divide, negate), control flow (conditional and unconditional branches) and output. In this paper, we consider only machine-language programs of fixed size and composed of such instructions.

Hill climbing is a well-known search technique (see, for example, [13]) that, given a candidate individual, examines near neighbors for an improvement. *Iterative* hill climbers continue this process with a "better" neighbor as a replacement for the candidate individual until a (global or local) optimum is reached. Hill climbing, unlike evolutionary searches, is a form of local search since it only examines neighbors a bounded distance from the current candidate. We compare MLGP using crossover (XO) to an *exhaustive* iterative hill climber (EIHC). EIHC is exhaustive because for a program p, it examines *all* individuals that differ from p by a single instruction. From MLGP (using XO) and EIHC, we then also construct and measure a hybrid global-local search mechanism (XO-EIHC).

We find that the evolutionary searches (XO or XO-EIHC), as well as hill climbing (EIHC), can discover solutions to the four sequence problems of this paper.

[1] The *square* and *cube* functions are similar to the simple regressions commonly solved via GP (*e.g.* [8]); their recursive solution by MLGP is novel as is their treatment here as sequences. We are unaware of solutions to the *factorial* function or sequence by GP or other machine-learning techniques.

[2] MLGP's "instructions" correspond to GP's "functions"; we use the terminology interchangeably.

Surprisingly, all discovered solutions are exact (verified by hand) up to to underlying machine's register precision. That is, given registers of arbitrary precision, the resultant programs correctly produce the infinite integer sequences defined by the functions of Figure 1.

In comparing the efficiencies of the various search techniques by counting program evaluations, we find that an evolutionary search (XO or XO-EIHC) outperforms exhaustive hill climbing (EIHC) on the problems considered. The work of O'Reilly and Oppacher [12] similarly contrasts GP, stochastic iterated hill climbing, and hybrids thereof; their problem set and representation, however, differ qualitatively from ours, making direct comparisons of quantitative conclusions unenlightening. Finally, comparison of XO, EIHC, and XO-EIHC to random search reveals that, for the problems considered in this paper, a directed search significantly outperforms random search.

The contributions of this paper are threefold; we

1. demonstrate the feasibility of using directed searches to learn recursive integer sequences using only a generic machine-language representation *without* recursive instructions,

2. compare the efficiencies of evolutionary, hill climbing, hybrid (evolutionary and hill climbing) and random searches, and

3. introduce novel MLGP techniques for manipulating (possibly unbounded) sequences as MLGP result values.

We use the Finnegan system (described briefly in Section 2) to interpret VRM-$\mathcal{S}$ (Section 3). Section 4 describes the experimental setup in general and the four search techniques (XO, EIHC, XO-EIHC, Random) in particular. Results and conclusions of using the searches to learn the four sequences defined by the functions of Figure 1 are in Section 5. We summarize in Section 6.

2 Finnegan System

Finnegan (*cf.* [7]) is a framework for experimenting with simulated evolution of machine-language programs. It is written in the Standard ML (SML) programming language [10] and implemented using the Standard ML of New Jersey (SML/NJ) compiler [1]. Since SML is highly modular, it is well suited to an experimental MLGP framework into which one can easily plug various representations, fitness functions, *etc.*, selected from libraries of such components.

3 Virtual Register Machine $\mathcal{S}$

The machine-language representation used for the sequence problems is an instance of a virtual register machine called VRM-$\mathcal{S}$. The machine is virtual because it is interpreted by software. The notation VRM-$\mathcal{S}_{(n,m)}$ names a particular VRM-$\mathcal{S}$ that consists of external state (m integer registers), internal state (a program counter and flag) and a sequence of n immutable instructions.

3.1 External State: Registers

We define the *register state* as a vector

$$\vec{R} \equiv \langle R_0, \ldots, R_{m-1} \rangle$$

of m integers; the register state constitutes the machine's mutable memory. The precision of a register is inherited from the underlying implementation.[3] Many program instructions (*e.g.*, `Add`) modify registers directly.

All program input is communicated through the register state; that is, program inputs are supplied in the initial state $\vec{R}_{initial}$. Outputs may be taken from the final register state $\vec{R}_{final}$ and from an output stream of integers denoted *Stdout* and produced by the `Out` instruction (further described below). Registers may also be used to initially supply the program with constants; alternately, the program can synthesize necessary constants in registers.

For the sequence problems of this paper, the registers were initialized to zero.

3.2 Internal State: PC, Flag

In addition to the external register state, VRM-$\mathcal{S}$ maintains two pieces of internal state: a program counter (*PC*) and a comparison flag (*Flag*). The program counter is an integer that selects which instruction to fetch and execute. Branch instructions modify the *PC* to point to the branch's target; all other instructions always increment the *PC* to point to the next instruction. The *PC* is initially set to zero; that is, it points to the first program instruction.

The *Flag* reflects the result of the last comparison instruction executed. It can assume the values *less*, *greater*, and *equal*. *Flag* is initially undefined which we denote as $\perp$ (bottom). Only the comparison instructions (see below) can modify the *Flag* state.

3.3 Instruction Set

A *program* is a vector of n instructions

$$\vec{I} \equiv \langle I_0, \ldots, I_{n-1} \rangle$$

The program counter naturally corresponds to an index of $\vec{I}$. A program *terminates* when $PC = n$; that is, when evaluation steps past the end of the program. (Our evaluation strategy also limits the maximum number of instructions evaluated; see Section 5.) Figure 2 contains VRM-$\mathcal{S}$'s instructions and their operational semantics.

[3]Integers in the SML/NJ compiler used for Finnegan are signed and 31 bit.

$$\texttt{Out}(R_a) \equiv \left(\begin{array}{l} PC \leftarrow PC + 1 \\ \mathit{Write}(\mathit{Stdout}, R_a) \end{array} \right.$$

$$\texttt{Mov}(R_{dst}, R_{src}) \equiv \left(\begin{array}{l} PC \leftarrow PC + 1 \\ R_{dst} \leftarrow R_{src} \end{array} \right.$$

$$\texttt{Set}(R_a) \equiv \left(\begin{array}{l} PC \leftarrow PC + 1 \\ R_a \leftarrow 1 \end{array} \right.$$

$$\texttt{Clear}(R_a) \equiv \left(\begin{array}{l} PC \leftarrow PC + 1 \\ R_a \leftarrow 0 \end{array} \right.$$

$$\texttt{Inc}(R_a) \equiv \left(\begin{array}{l} PC \leftarrow PC + 1 \\ R_a \leftarrow 1 \oplus R_a \end{array} \right.$$

$$\texttt{Dec}(R_a) \equiv \left(\begin{array}{l} PC \leftarrow PC + 1 \\ R_a \leftarrow 1 \ominus R_a \end{array} \right.$$

$$\texttt{Neg}(R_a) \equiv \left(\begin{array}{l} PC \leftarrow PC + 1 \\ R_a \leftarrow R_a \ominus 0 \end{array} \right.$$

$$\texttt{Add}(R_{dst}, R_{src}) \equiv \left(\begin{array}{l} PC \leftarrow PC + 1 \\ R_{dst} \leftarrow R_{dst} \oplus R_{src} \end{array} \right.$$

$$\texttt{Sub}(R_{dst}, R_{src}) \equiv \left(\begin{array}{l} PC \leftarrow PC + 1 \\ R_{dst} \leftarrow R_{dst} \ominus R_{src} \end{array} \right.$$

$$\texttt{Mul}(R_{dst}, R_{src}) \equiv \left(\begin{array}{l} PC \leftarrow PC + 1 \\ R_{dst} \leftarrow R_{dst} \otimes R_{src} \end{array} \right.$$

$$\texttt{Div}(R_{dst}, R_{src}) \equiv \left(\begin{array}{l} PC \leftarrow PC + 1 \\ R_{dst} \leftarrow R_{dst} \oslash R_{src} \end{array} \right.$$

$$\texttt{Nop} \equiv \left(\; PC \leftarrow PC + 1 \right.$$

$$\texttt{Cmp}(R_a, R_b) \equiv \left(\begin{array}{l} PC \leftarrow PC + 1 \\ \mathit{Flag} \leftarrow \begin{cases} \mathit{less} & \text{if } R_a < R_b \\ \mathit{greater} & \text{if } R_a > R_b \\ \mathit{equal} & \text{otherwise} \end{cases} \end{array} \right.$$

$$\texttt{J}(\mathit{offset}) \equiv \left(\; PC \leftarrow \min\left(\max\left(0, PC + \mathit{offset}\right), n\right) \right.$$

$$\texttt{Jl}(\mathit{offset}) \equiv \left(\; PC \leftarrow \begin{cases} \min\left(\max\left(0, PC + \mathit{offset}\right), n\right) & \text{if } \mathit{Flag} = \mathit{less} \\ PC + 1 & \text{otherwise} \end{cases} \right.$$

$$\texttt{Jg}(\mathit{offset}) \equiv \left(\; PC \leftarrow \begin{cases} \min\left(\max\left(0, PC + \mathit{offset}\right), n\right) & \text{if } \mathit{Flag} = \mathit{greater} \\ PC + 1 & \text{otherwise} \end{cases} \right.$$

$$\texttt{Je}(\mathit{offset}) \equiv \left(\; PC \leftarrow \begin{cases} \min\left(\max\left(0, PC + \mathit{offset}\right), n\right) & \text{if } \mathit{Flag} = \mathit{equal} \\ PC + 1 & \text{otherwise} \end{cases} \right.$$

Figure 2: Operational semantics for the virtual register machine VRM-$\mathcal{S}$. Evaluation commences with $PC = 0$ and *Flag* uninitialized. The trapping arithmetic operators ($\oplus$, $\ominus$, $\otimes$, and $\oslash$) denote the respective integer operation, but yield zero on exceptional cases (overflow, underflow, divide-by-zero).

VRM-$\mathcal{S}$ is a proper superset of VRM-$\mathcal{M}$ [7] previously used to evolve machine-language iteration. The `Out` instruction is, however, novel to VRM-$\mathcal{S}$ and to MLGP in general. Programs can use `Out` to place an integer on an output stream. In particular, the `Out` instructions appends the integer in its register argument to the tail of the global output stream called *Stdout*. A program that does not execute an `Out` instruction produces the *empty* stream; that is, *Stdout* contains no values after execution of such a program. The `Out` instruction mimics native machine-language instructions that write values to hardware ports.

The instruction set further consists of a nullary instruction `Nop` which does nothing, a register move instruction `Mov`, a comparison instruction `Cmp` that reflects the relation of its argument registers in the *Flag* state, an unconditional branch `J`, branches conditional on the *Flag* state (`Jl`, `Jg`, `Je`), instructions that initialize registers (`Set` and `Clear`), and instructions to increment (`Inc`) and decrement (`Dec`) a given register. The arithmetic instructions (`Add`, `Sub`, `Mul`, `Div`) perform the respective operation, leaving the result in the destination register. The arithmetic `Neg` instruction negates the value in its argument register. The arithmetic instructions trap exceptional conditions (overflow, underflow, divide-by-zero) for which they return zero (see Figure 2).

Branches (`J`, `Jl`, `Je`, `Jg`) are always relative to the program counter. Negative offsets describe a backward branch. Note that the operational semantics rewrites a branch to an address < 0 as a branch to I_0 (*i.e.* $PC \leftarrow 0$) and a branch past the end of the program $(n-1)$ as termination (*i.e.* $PC \leftarrow n$). A jump instruction I_j can therefore branch to any one of $n+1$ distinct addresses.

The number of syntactically-distinct instructions in a VRM-$\mathcal{S}_{(n,m)}$ is

$$S(m,n) \equiv 6m^2 + 6m + 4(n+1) + 1 \tag{1}$$

because there are six binary register instructions, six unary register instructions, four relative branch instructions, and one nullary instruction (`Nop`). The number of possible (syntactic) programs in a such a machine is therefore:

$$S(m,n)^n \tag{2}$$

3.4 Evaluation Function

Our interpreter evaluates an n-instruction VRM-$\mathcal{S}_{(n,m)}$ program $\vec{I}$ with respect to an m-register input state $\vec{R}$ and an integer number of evaluation steps (instructions), $K > 0$. $\mathcal{E}_{\mathcal{S}}$ maps a triple to a pair:

$$\mathcal{E}_{\mathcal{S}} : \left(\vec{I}, \vec{R}, K\right) \rightarrow \left(\vec{R}', \mathit{Stdout}\right) \tag{3}$$

$\mathcal{E}_{\mathcal{S}}$ produces the final register state $\vec{R}'$ and the output stream *Stdout* after evaluation of at most K instructions.[4]

[4]Since our VRM-$\mathcal{S}$ evaluator is an interpreter (essentially Figure 2), it can be easily halted after evaluation of K instructions.

4 Experimental Setup

This section describes the four search methods—genetic crossover (XO), exhaustive iterative hill climbing (EIHC), hybrid (XO-EIHC), and random search—applied to the four sequence problems. Before describing the individual methods, we first define the test-case model and the fitness function they have in common. We defer elaboration of the quantitative settings (*e.g.*, program size) to the next section (§5).

4.1 Test Case

The l-prefix test case for a sequence problem defined by function f is defined as $s \equiv \{f(0), \ldots, f(l-1)\}$. A prefix length of $l = 10$ suffices for the four sequence problems of this paper; the effect of other prefix sizes has not been investigated.

4.2 Fitness Function

A program's fitness is computed only from the *Stdout* stream returned by the evaluation function (Equation 3). Lower fitness values are better. For sequence function f, the raw fitness of an n-instruction program $\vec{I}$ on test case s is given by

$$\mathcal{F}_s\left(\vec{I}\right) \equiv \sum_{i=0}^{l-1} |v_i - f(i)| \cdot \mathit{scale}_s(i) \tag{4}$$

where, using (3), $\mathcal{E}_{\mathcal{S}}(\vec{I}, \vec{R}, K) = \left(\vec{R}', \mathit{Stdout}\right)$, input registers $\vec{R} \equiv \langle 0, \ldots, 0_{m-1}\rangle$, and K is the maximum number of evaluation steps. Take the sequence $\{v_0, \ldots, v_{l-1}\}$ to be the first l values of *Stdout* if this stream contains at least l values; otherwise, when *Stdout* contains $j < l$ values, take values $v_j, \ldots, v_{l-1}$ as $s_{\max}$ (where $s_{\max}$ is the largest value in sequence $\mathcal{S}$; that is, $s_{\max} \equiv \max\{f(0), \ldots, f(l-1)\}$ for the sequence defined by f). The function scale_s, returning a real value, is defined as

$$\mathit{scale}_s(i) \quad \equiv \quad \begin{cases} s_{\max} & \text{if } f(i) = 0 \\ s_{\max}/f(i) & \text{otherwise} \end{cases} \tag{5}$$

It serves to scale an incorrect result value, $|v_i - f(i)| > 0$ from (4) above, in proportion to the magnitude of the error.[5]

4.3 Search Methods

Here, we describe the search methods used to find solutions to the sequence problems.

[5]When $f(i) = 0$, $s_{\max}/f(i)$ is undefined and we penalize the program with $s_{\max}$.

*4.3.1 Genetic Search (*XO*)*

The MLGP genetic search uses proportional selection as its population-selection mechanism and crossover (XO) as its sole genetic operator.

Population Selection Population selection, for the construction of successive generations, is performed via *proportional selection* (see, *e.g.*, [8, 9]). Let P be a population (set) of N programs. The selection mechanism first normalizes an individual's raw fitness (4) to the unit interval:

$$\widehat{\mathcal{F}_s}\left(\vec{I}\right) \equiv \frac{1}{\mathcal{F}_s\left(\vec{I}\right)\sum_{p\in P}\mathcal{F}_s\left(p\right)^{-1}} \tag{6}$$

It then selects N random reals, $X \equiv \{x_1, \ldots, x_N\}$, in the unit interval. For each $x \in X$, population selection then chooses a program $\vec{I}$ for the next generation if $x < \widehat{\mathcal{F}_s}\left(\vec{I}\right)$ and there does not exist an $\vec{I}'$ such that $x < \widehat{\mathcal{F}_s}\left(\vec{I}'\right) < \widehat{\mathcal{F}_s}\left(\vec{I}\right)$.

Genetic Operator: Two-Point Crossover We use a single recombination operator that performs two-point crossover.

Crossover of two n-instruction programs $\vec{I_i}$ and $\vec{I_j}$ first selects a subsequence of instructions starting at a random point $0 \leq p_i < n$ in program $\vec{I_i}$. The length $k > 0$ of the subsequence is chosen randomly such that $p_i < p_i + k < n$. A random point p_j, $0 \leq p_j + k < n$, is then chosen in program $\vec{I_j}$. Finally, the k instructions in $\vec{I_i}$ starting at p_i are interchanged with the k instructions in $\vec{I_j}$ starting at p_j.

Crossover in a population P is performed by first selecting a subset $P' \subseteq P$ of programs from the population; an individual program is randomly selected for P' with probability $Prob_{xover}$. The programs in P' are then randomly paired and the crossover operator is applied to each pair. The pairs resulting from crossover replace the corresponding original pairs in the population.

*4.3.2 Exhaustive Iterative Hill Climbing (*EIHC*)*

Hill climbing is the second search method we examined in the context of learning integer sequences using a machine-language representation. In particular, we implemented an exhaustive iterative hill climber (EIHC) that works as follows.

EIHC first randomly generates an n-instruction candidate program p, evaluates it, and computes its fitness. For each of the n instruction positions i in p, EIHC iteratively replaces the instruction at i with all possible VRM-$\mathcal{S}$ instructions, evaluates the resulting neighbors, and records their fitness values. The neighbor p' with the largest improvement in fitness replaces the candidate p. If no such neighbor exists—that is, no single-point instruction replacement improves on p's fitness—the search has reached an optimum (local or global) and the search terminates. A search that fails to find a global optimum is restarted with a new, random, candidate program.

Since EIHC replaces all program instructions with all possible VRM-$\mathcal{S}$ instructions, it exhaustively examines all single-point changes. Since it moves in the direction with the greatest improvement in fitness, it is a "steepest gradient" search. Many variations on hill climbing are possible: movement in the direction of the *first* improvement (instead of the best), stochastic selection of *some* single-point changes (*cf.* [12]), examination of *multi-point* changes. We have not explored the impact on performance that such variations may have.

*4.3.3 Hybrid Search (*XO-EIHC*)*

We combine XO with EIHC to form a hybrid search method (XO-EIHC) as follows. Recall that MLGP with XO proceeds in generations. At the start of a new generation for XO-EIHC, we randomly select a single individual p from the population P, apply EIHC to p until a local or global optimum is found, and reinsert the result of the EIHC search into P. Many variations of this scheme are possible: select fewer/more individuals per generation, select only from the fittest individuals, use an alternate form of local search, *etc.* As with pure EIHC, we have not explored the effects of such variations.

4.3.4 Random Search

Random search randomly generates an individual p, evaluates p and computes its fitness, and (optionally) records p's fitness as the best if p improves on the current best fitness. This process continues until a sufficient number of global solutions are found or until the number of program evaluations exceeds a predetermined threshold.

5 Results

For each of the four sequence problems defined by the functions of Figure 1, we conducted four experiments, one for each of the search methods of the the previous section. Table 1 holds the quantitative results.

We used a single parameter set for all experiments; only the target test sequence (§4.1) was variable. The virtual register machine was instantiated to VRM-$\mathcal{S}_{(12,12)}$; that is, programs were of length $n = 12$ and had $m = 12$ registers.[6] The maximum number of

[6]Equation 2 indicates that the number of syntactic programs in VRM-$\mathcal{S}_{(12,12)}$ is on the order of 10^{35}.

	#Solns	#Evals	$\frac{\#\text{Evals}}{\#\text{Solns}}$
XO	10	506443	5.06×10^5
EIHC	10	3455593	3.45×10^6
XO-EIHC	10	1931384	1.93×10^6
Random	3	(max-evals)	1.6×10^7

square

	#Solns	#Evals	$\frac{\#\text{Evals}}{\#\text{Solns}}$
XO	10	32877175	3.28×10^6
EIHC	5	(max-evals)	1.00×10^7
XO-EIHC	10	32552681	3.25×10^6
Random	0	(max-evals)	—

cube

	#Solns	#Evals	$\frac{\#\text{Evals}}{\#\text{Solns}}$
XO	10	26692924	2.66×10^6
EIHC	8	(max-evals)	6.25×10^6
XO-EIHC	9	(max-evals)	5.55×10^6
Random	0	(max-evals)	—

factorial

	#Solns	#Evals	$\frac{\#\text{Evals}}{\#\text{Solns}}$
XO	0	(max-evals)	—
EIHC	10	21114026	2.11×10^6
XO-EIHC	10	10239228	1.02×10^6
Random	0	(max-evals)	—

Fibonacci

Table 1: Results of the four search methods applied to the sequence problems. For each sequence and search method, the number of solutions discovered, number of evaluations required to discover the solutions, and efficiency (#Evaluations per #Solutions) are given. Experiments were limited to the first of either ten solutions or 5×10^7 (max-evals) evaluations.

program evaluation steps was set to $K = 100$.

(It can be argued that the values n, m, and K implicitly carry information about the solutions being sought. One could however automatically find such values by starting with small values (*e.g.* one) and—if unsuccessful—increasing the values, by perhaps doubling, and retrying the search.)

For the evolutionary searches (XO, XO-EIHC) the population size was set to $N = 256$ and the crossover probability to $Prob_{xover} = 0.25$. A run was deemed complete after 32 generations without an improvement in best fitness (*stasis*). Different values for the evolutionary parameters of course influence the comparative results; conclusions drawn from the comparisons must take this into account.

For each sequence and search method, the number of solutions discovered, number of evaluations required to discover the solutions, and efficiency are given in Table 1. Efficiency is measured in evaluations per solution, where a lower value indicates higher efficiency; efficiency is undefined when a search method yields no solutions. To curtail manual verification time and cpu time, searches were limited to the first ten solutions or 5×10^7 evaluations, whichever came first.

From the table one can conclude that the evolutionary searches perform quite well and that, as to be expected, random search performs extremely poorly. Pure hill climbing (EIHC) never performs as well as an evolutionary search in this experimental setup. From the results, it is not apparent when to use the hybrid (XO-EIHC) search. XO-EIHC is competitive with XO on *cube*, but lags XO on *square* and *factorial*. XO-EIHC, however, produces solutions for *Fibonacci* where XO surprisingly produces no solutions (we suspect this to be an artifact of the evolutionary parameters, program length and number of evaluation steps).

All solutions discovered by the four search methods are exact—they produce the infinite sequence of their respective function up to the machine's underlying register precision. Solutions were checked for exactness by hand. We note that program and register sizes (n and m) must govern the tradeoff of solution exactness versus overfitting. That is, the incorporation of the target sequence into the program as data—instead of its production by an algorithm—to yield a non-general solution seems more likely as the difference between n and the length of the target prefix l increases. Further experiments are necessary to ascertain if and when overfitting in MLGP occurs.

Sample solutions to the *square*, *cube*, *factorial*, and *Fibonacci* sequence problems found by the hybrid XO-EIHC search are in Figure 3. The solutions produce the respective infinite sequence via an infinite loop synthesized from jump instructions. Since a solution must produce an output sequence, it also contains at least one `Out` instruction. Note how the solution for *Fibonacci* economizes on instructions by using two instances of `Out`.

The programs for *Fibonacci* and *factorial* first (partially) generate the required base cases before entering their loop. Note the exploitation of `Div`'s divide-by-zero behavior in lines 7–8 of *factorial*'s solution: after emit-

Address	Instruction
0:	~~Nop~~
1:	Out(R$_6$)
2:	Inc(R$_1$)
3:	Add(R$_6$,R$_1$)
4:	~~Inc(R$_2$)~~
5:	Inc(R$_1$)
6:	Jg(+2)
7:	Cmp(R$_6$,R$_0$)
8:	~~J(+1)~~
9:	Jg(-10)
10:	~~Clear(R$_1$)~~
11:	~~J(+1)~~

square
(generation 6)

Address	Instruction
0:	Inc(R$_4$)
1:	Out(R$_0$)
2:	Add(R$_{10}$,R$_4$)
3:	Add(R$_0$,R$_{10}$)
4:	Inc(R$_4$)
5:	Add(R$_{10}$,R$_4$)
6:	~~J(+2)~~
7:	~~Inc(R$_5$)~~
8:	Inc(R$_4$)
9:	J(-9)
10:	~~J(+3)~~
11:	~~J(+1)~~

cube
(generation 88)

Address	Instruction
0:	Inc(R$_1$)
1:	~~Mov(R$_6$,R$_0$)~~
2:	Add(R$_1$,R$_6$)
3:	Out(R$_1$)
4:	~~Add(R$_6$,R$_1$)~~
5:	Mov(R$_6$,R$_1$)
6:	Mul(R$_1$,R$_9$)
7:	Div(R$_1$,R$_{10}$)
8:	Set(R$_{10}$)
9:	Inc(R$_9$)
10:	J(-8)
11:	~~J(+1)~~

factorial
(generation 26)

Address	Instruction
0:	Dec(R$_1$)
1:	~~Neg(R$_0$)~~
2:	Set(R$_6$)
3:	Sub(R$_5$,R$_1$)
4:	Cmp(R$_7$,R$_6$)
5:	Add(R$_5$,R$_4$)
6:	Out(R$_5$)
7:	Add(R$_4$,R$_5$)
8:	Clear(R$_1$)
9:	Out(R$_4$)
10:	Jl(-7)
11:	~~J(R$_{+1}$)~~

Fibonacci
(generation 31)

Figure 3: Sample solutions to the *square*, *cube*, *factorial*, and *Fibonacci* sequence problems found by the hybrid XO-EIHC search. Instructions that do not affect a program's result have a ~~line~~ through them.

ting *factorial*(0), and only in the first loop iteration, it clears R$_1$ in preparation for *factorial*(1).

In the solutions exhibited, the results from previous loop iterations are used in computing the current iteration's output value(s). Solutions to *factorial* and *Fibonacci* follow their natural recursive definitions (Figure 1); for example, the solution for *Fibonacci* maintains prior *Fibonacci* values in registers R$_4$ and R$_5$. The solutions for *square* and *cube* use the expansion of the binomial theorem[7] and recursively compute their results as well. In other words, *square* computes $(x+1)^2$ by adding $2x+1$ to the value of x^2 computed in the previous iteration. Similarly, *cube* computes $(x+1)^3$ from x^3. Solution programs for *square* and *cube* typically converged on this recursive structure. Solutions using multiplication to compute *square* and *cube* were atypical; no solutions for *cube* used multiplication and only a single solution to *square* did so (and this was found by random search!). This suggests that the ability to synthesize control-flow can potentially increase the number of possible solutions on certain problems, implicitly increasing search efficiency.

6 Summary

We have demonstrated that the directed search of machine-language genetic programming, as well as that of hill climbing, can be applied to the space of machine-language computer programs to learn the integer sequences generated by the functions for *square*, *cube*, *factorial* and *Fibonacci*. A hybrid global-local search (genetic crossover and hill climbing) also performs well in that it finds solutions to all four problems. Furthermore, directed searches always outperformed random search.

Notably, our machine-language representation does not require recursive operators to produce programs that—for the experimental solutions generated—exactly produce the infinite recursive sequence implied by the sequence's ten-element prefix, supplied as the test case. The instruction set is generic in that it contains only primitive instructions for register manipulation, conditional and unconditional branches, and arithmetic. For this work, we also introduced an instruction that allows programs to produce value streams as output.

Further future investigation of search-parameter settings (*e.g.*, program or population size) is necessary to enable more definitive comparisons of the search techniques and to guide their further integration into machine-language genetic-programming systems. Relatedly, a study of the relationship of program size and evaluation steps to solution quality (generality) is important. Refinement of technique should allow MLGP solution of more difficult sequence-induction problems.

Acknowledgments

Thanks to David Hull and Andy Pargellis for discussions of this work and comments on this paper.

References

[1] A. W. Appel and D. B. MacQueen. A Standard ML compiler. *Functional Programming Languages and Computer Architecture*, 274:301–324, 1987.

[2] N. L. Cramer. A representation for the adaptive generation of simple sequential programs. In *Proceeding of the International Conference on Genetic Algorithms and their Applications*, pages 183–187. Texas Instruments, July 1985.

[3] R. M. Friedberg. A learning machine: Part I.

[7] $(a+b)^n = \sum_{k=0}^{n} \binom{n}{k} a^{n-k} b^k$

IBM Journal of Research and Development, 2:2–13, 1958.

[4] R. M. Friedberg, B. Dunham, and J. H. North. A learning machine: Part II. *IBM Journal of Research and Development*, 3:282–287, 1959.

[5] S. Handley. A new class of function sets for solving sequence problems. In *Proceedings of the Conference on Genetic-Programming*, pages 301–308, July 1996.

[6] J. Holland. *Adapation in Natural and Artifical Systems.* University of Michigan Press, 1975.

[7] L. Huelsbergen. Toward simulated evolution of machine-language iteration. In *Proceedings of the Conference on Genetic-Programming*, pages 315–320, July 1996.

[8] J. Koza. *Genetic Programming: On the Programming of Computers by the Means of Natural Selection.* MIT Press, Cambridge, MA, 1992.

[9] Z. Michalewicz. *Genetic Algorithms + Data Structures = Evolution Programs.* Springer Verlag, Berlin, 1992.

[10] R. Milner, M. Tofte, and R. Harper. *The Definition of Standard ML.* MIT Press, 1990.

[11] P. Nordin. A compiling genetic programming system that directly manipulates the machine-code. In K. Kinnear Jr., editor, *Advances in Genetic Programming*, chapter 14, pages 311–331. MIT Press, 1994.

[12] U.-M. O'Reilly and F. Oppacher. A comparative analysis of genetic programming. In K. Kinnear and P. J. Angeline, editors, *Advances in Genetic Programming II*, chapter 2, pages 23–44. MIT Press, 1996.

[13] E. Rich. *Artificial Intelligence.* McGraw-Hill, 1983.