Appears in 1998 Conference on Genetic Programming
This paper contains minor revisions to the published article.

Finding General Solutions to the Parity Problem by Evolving Machine-Language Representations

Lorenz Huelsbergen
`lorenz@research.bell-labs.com`
Bell Laboratories, Lucent Technologies
Murray Hill, NJ 07974

Abstract

Evolutionary search on a machine-language program representation is used to find exact and general solutions to the *bit-counting problem*—abstractly, given a set B of boolean variables, compute the number of variables in B with value *true*. Solutions to the bit-counting problem directly solve the *parity problem* that decides whether the number of *true* variables in B is even or odd. Our virtual machine language contains generic instructions for manipulating a register set and includes unrestricted branches and bit-wise logical operations. It does not contain instructions specific only to bit-counting or parity. A two-level lexicographic fitness function first selects for programs that closely approximate (or solve) the problem and then selects programs that properly terminate. We compare two population-based search operators: crossover and macro-mutation. For this problem, macro-mutation significantly outperforms crossover. Introducing single-point mutation significantly improves the crossover search but only slightly improves the macro-mutation search. Comparison with random search verifies the efficacy of the directed search methods on the bit-counting problem.

1 Introduction

Program induction—as practiced in genetic and evolutionary programming—proposes to discover, via search, *general* algorithms, embodied as programs, that solve a specified target problem. A finite, often small, suite of test cases provides sample input/output mappings to guide the search. Ideally, only this set of mappings and a fitness function that measures how "closely" a program satisfies the mappings should convey information about the target problem to the induction system. The underlying representation should therefore contain only *domain independent* computational primitives. Two desirable properties of a program-induction system have been elucidated—that it synthesize abstract and general programs and that it use a generic computational representation.

The *parity problem*—deciding whether the number of boolean variables with assignment *true* in a set B of such variables is even (or odd)—has emerged from the genetic programming (GP) [13] framework as a difficult problem for program induction. Intuitively it is a hard problem because parity (on bit strings interpreted as integers) is non-linear and non-monotonic. In addition to serving as a standard benchmark for quantifying GP techniques, the parity problem has independently received considerable study [8, 17, 13, 14]. Solutions evolved to date fall into one of two classes. The first class contains solutions found by GP's genetic-algorithm search and are constructed solely from general logic primitives (*e.g.*, `NAND`) and terminals (the boolean variables) composed in Lisp expressions. Solutions in this class [8, 14, 13], however, fix the maximum size of the input set B to a small constant (typically ≤ 11) and thus do not generalize to arbitrary input sets. Solutions in the second class [17] are general, logic programs in this case, but were evolved using a grammar specifically devised to solve the parity problem. This grammar, for example, guides the search toward solutions by constraining the arguments of recursive calls to forms known *a priori* to solve the parity problem. Previous evolved solutions to parity are therefore either not general or not domain specific.

Here we present the first evolved solutions to the parity problem that are general and synthesized only from generic operations. To accomplish this, we evolve a *virtual register machine* (VRM) that operates on a set of integer registers. The VRM includes instructions—similar to ones of contemporary processors—to perform arithmetic, move data, and twiddle bits. Branch instructions allow synthesis of arbitrary control flow. This unrestricted control flow enables solution of the parity problem since it may be used to synthesize a recursive decomposition of the problem. The input set B is represented as an n-bit vector V in an input register. The fitness function requires the program to count the number of set bits in V. The least-significant bit of the resultant count answers the parity problem. Although the solutions presented here were evolved using a VRM with registers of finite size, we have manually checked that the solutions are general for all larger register sizes.

In addition to the new approach to the parity problem, this work extends the previous machine-language induction (MLI) work [6, 7, 4, 16, 10]. Through instruction-set extensions and consideration of longer programs with more registers, our system is now searching very large spaces. A *lexicographic fitness function* (see, for example, [2]) improves selection for *secondary program characteristics* such as termination. That is, a two-tier fitness function first selects for program correctness. Within a set of programs with equivalent correctness, the second-tier function favors the programs that properly terminate. Doing so obviates the—often *ad hoc*—credit assignment step that a human must otherwise perform if termination (or *any* secondary characteristic) is combined as a separate term with the correctness term. Unlike Pareto optimality (see [2]), lexicographic fitness prioritizes the multiple objectives instead of optimizing them simultaneously. We describe how lexicographic selection can be integrated with tournament selection and how it may be applied to other program characteristics (size and speed, for example).

Two types of evolutionary search are compared: genetic algorithm with crossover (GA-XO) and genetic algorithm with macro-mutation (GA-MM). (GA-MM uses the GA's population but not its crossover operator.) Similar studies to ascertain the efficacy of crossover have been conducted for some GA (*e.g.*, [11]) and program-induction problems (*e.g.*, [1, 3, 11]). For the bit-counting problem, we find that GA-MM significantly outperforms GA-XO. Furthermore, introduction of single-point mutation has a large effect on crossover search but not on macro-mutation search. Random search confirms that the evolutionary searches were indeed directed and not "blind."

Many of the bit-counting algorithms evolved for this paper directly correspond to well known programming idioms. A few evolved solutions do not appear to be part of programming folklore and hence may be potentially novel algorithms. We furthermore note that evolved solutions exhibit asymptotically different algorithmic time complexities.

The paper is structured as follows. The next section details prior work, Section 3 provides a brief overview of the search system, Section 4 describes the virtual register machine, Section 5 describes the experimental setup (fitness function, *etc.*), and Section 6 contains results. We then summarize with a discussion in Section 7.

2 Related Work

Previous work relevant to this paper loosely falls into four categories: induction of the parity function, machine-language induction, crossover versus (macro) mutation, and techniques for multi-objective optimization (lexicographic and Pareto orderings).

Koza has extensively examined the parity problem in the context of his Lisp-based GP [13]. In this setting, side-effect free Lisp expressions, containing boolean functions such as `NAND`, operate on a *fixed* set of boolean variables $(D_0, \ldots, D_{k-1})$ called terminals. The fitness function selects expression trees that correctly determine whether the number of terminals with value *true* for all possible variable assignments is even or odd. An expression tree that correctly computes k-parity trivially computes j-parity for $j < k$. However, it cannot compute parity for $j > k$ because its data structure, namely the terminal set, is static and hence cannot represent larger instances. That is, Koza's formulation of the problem does not admit general solutions. The machine-language approach to the parity problem can generalize because it may synthesize loops and can access individual bit-vector elements that encode the problem's variables. As we shall see, many solutions discovered by our system are independent of this vector's length and are hence general. Whether GP with loop constructs and a more flexible set representation (*e.g.*, lists or bit vectors) can solve the parity problem *in general* is an open question.

With a powerful procedural-abstraction technique (ADF [14]), Koza has solved instances of k-parity for values of k to 11. Gathercole and Ross [8] have developed a novel fitness mechanism that enables GP to solve fairly large parity instances ($k = 7$, for example) more readily. Since our approach generalizes, we do not need to train with such large instances to produce programs that correctly solve arbitrary instances (all k).

The generic genetic-programming system (GPP) of Wong and Leung [17] takes an approach to the parity problem that is orthogonal to those of GP and MLI. As do GP and MLI, GPP induces programs using search, but does so with guidance of a custom grammar that must be explicitly designed for a given problem. Wong and Leung designed such a grammar for the k-parity problem that allows GPP to find general recursive solutions ($k > 0$) using a logic-program representation. This grammar, however, contains much information about the structure of such solutions. For example, it encodes the syntax of the call to the main function (called `parity`) and stipulates that `parity` be called recursively only with list arguments smaller than the original argument. Machine-language induction too finds solutions that are general, but that also have the further advantage of not requiring *a priori* human specification of a solution's structure.

The groundwork for machine-language induction was laid by Friedberg *et al.* [6, 7]. Their early experiments were not successful when compared to random search, but they elucidated the MLI ideas being explored today. Cramer [4] describes a more modern experiment that uses a representation similar to a machine language, but endowed with high-level iteration operators. Nordin's system [16] manipulates native machine-code (as opposed to the interpreted virtual machines used here) to provide rapid evaluation. We have previously reported experiments that synthesize iterative control flow to learn the Fibonacci sequence, for example [10].

Mutation, without crossover, has been used to search for computational structures. (The evolutionary programming paradigm [5] provides many examples.) Recently, the benefits (or detriments) of crossover in GA-based approaches (such as GP) have begun to receive scrutiny. Jones [11] suggested and performed experiments to ascertain the viability of crossover. In comparison with crossover, he found macro-mutation to perform quite well for many problems, especially for problems containing no obvious "building blocks." Furthermore, even in the presence of such building blocks, macro-mutation effectively found solutions. We describe Jones's approach in Section 5.3.2. Angeline [1] compares macro-mutation to crossover in his (non-standard) GP system and finds that the performance of macro-mutation approaches, and sometimes exceeds that of crossover. Chellapilla [3] also successfully uses macro-mutation to evolve Lisp programs. Our macro-mutation experiments extend this line of inquiry to machine-language representations. We have previously reported experiments—only slightly successful—using exhaustive hillclimbers (based on single-point mutation) on machine languages [10].

Lexicographic and Pareto optimization are well known techniques for evaluating a candidate solution according to multiple criteria. See Ben-Tal [2] for a characterization of both techniques. Lexicographic optimization prioritizes the criteria and emphasizes those with higher priority. As the optimization progresses and the high-priority criteria become explored, lower priority criteria come into play. Pareto optimization, on the other hand, optimizes multiple criteria simultaneously. We believe lexicographic fitness to be useful in evolving higher-order program characteristics beyond primary correctness. For example, lexicographic fitness can select for—among many other possible higher-order characteristics—small programs, fast programs, low-power programs, or for terminating programs (the secondary characteristic we use in the experiments of this paper).

3 Finnegan System

We have built a modular GA system—called Finnegan-C (*cf.* [10])—tailored to machine-language induction problems. Its modularity allows experimentation with different problem descriptions, test suites, virtual register machines, and evolution operators by linking together appropriate object files and libraries. A separate driver for conducting random search is also provided. Problem descriptions (fitness functions) are supplied as C functions. Since the VRMs are also implemented in C, their instructions—as we shall see—often inherit this language's operator semantics [12].

4 Virtual Register Machine $\mathcal{B}$

The machine-language representation used for the sequence problems is an instance of a *virtual register machine* called VRM-$\mathcal{B}$. It is virtual because it is interpreted by software. VRM-$\mathcal{B}$ differs from our earlier VRMs (*e.g.*, [10]) in that it contains an additional group of instructions that perform bit-wise operations, such as logical operations and shifts. Conditional branch instructions now also embody the condition check; previously a separate "compare" instruction set an internal flag that governed conditionals.

The notation VRM-$\mathcal{B}_{(n,m)}$ names a particular VRM-$\mathcal{B}$ that consists of external state (m integer registers), internal state (a program counter), and a sequence of n immutable instructions.

4.1 External State: Registers

We define the *register state* as a vector

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

of m signed integers. The register state constitutes the machine's mutable memory. The precision of a register is inherited from the underlying implementation.[1] 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 $\vec{R}$; output is taken from the final register state $\vec{R}'$. Registers may also be used to initially supply the program with constants; alternately, the program can synthesize necessary constants in registers.

For the bit-counting problem of this paper, non-input registers were initialized to zero. We defer description of the concrete input configuration to Section 5.

4.2 Internal State: PC

In addition to the external register state, VRM-$\mathcal{B}$ maintains a piece of internal state: a program counter (PC). The PC is an integer, $0 \leq PC < n$, that selects which

[1] Our implementation inherits 32-bit signed integers via C's `int` type on a 32-bit machine.

instruction to fetch and execute. Branch instructions modify the PC by adding a signed offset to it; all other instructions always increment the PC to point to the next instruction. The PC is initially zero; that is, program execution begins at the first program instruction.

4.3 Instruction Set

A *program* is a vector of n instructions

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

The program counter 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 6.) Figure 1 contains VRM-$\mathcal{B}$'s instructions and their operational semantics. Here we first summarize the instructions VRM-$\mathcal{B}$ shares with prior VRMs [10] and then describe the bit-wise instructions new here.

VRM-$\mathcal{B}$ instructions previously used in other VRMs [10] consist of a register move instruction `MOV`, an unconditional branch `J`, branches conditional on a register's value relative to the zero value (`JZ`, `JNZ`, `JLZ`, `JGZ`), instructions that initialize registers (`SET` and `CLR`), instructions to increment (`INC`) and decrement (`DEC`) a given register, and a nullary instruction `NOP` which does nothing. The arithmetic instructions (`ADD`, `SUB`, `MUL`, `DIV`, `MOD`) perform the respective two's complement operation on source and destination, leaving the result in the destination register. The arithmetic `NEG` instruction negates the value in its argument register. The arithmetic instructions mimic C's behavior and "wrap around" on the exceptional conditions of integer overflow (or underflow) instead of trapping. Arithmetic operations that can generate traps in C are `DIV` and `MOD` which are susceptible to divide-by-zero. Our VRM evaluator checks for zero divisors, a condition which we have (arbitrarily) defined to place zero in the destination register.

Branches (`J`, `JZ`, `JNZ`, `JLZ`, `JGZ`) 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 conditional branches are parameterized by a register and the jump displacement. `JZ` branches if the register is zero. `JNZ` branches on any value but zero. `JLZ` branches on a negative, and `JGZ` on a positive, register value.

New to this VRM are six bit-wise logical instructions found in the right-hand column of Figure 1. They perform their namesake's operation and are defined in terms of the respective C operators. A departure from this semantics is the interpretation of the shift operators as NOPs if the shift amount is either negative or exceeds the number of bits comprising an implementation register. Since VRM-$\mathcal{B}$ registers are signed, a right shift (`SHR`) of a negative quantity will effectively "reset" the sign bit.

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

$$S(m,n) \equiv 11m^2 + 6m + (4m+1)(n+1) + 1 \qquad (1)$$

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

$$S(m,n)^n \qquad (2)$$

4.4 Evaluation Function

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

$$\mathcal{E}_{\mathcal{B}} : \left(\vec{I}, \vec{R}, K\right) \rightarrow \vec{R}' \qquad (3)$$

$\mathcal{E}_{\mathcal{B}}$ produces the final register state $\vec{R}'$ after executing at most K instructions. Since our VRM-$\mathcal{B}$ evaluator is an interpreter (essentially Figure 1), it can easily be halted after evaluation of K instructions.

5 Experimental Setup

This section describes the three search methods—genetic crossover (GA-XO), population-based macro-mutation (GA-MM), and random search—applied to the bit-counting and parity 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.*, test-case and program size) to the next section (§6).

5.1 Test Cases

Let B denote a set of boolean variables as input to the bit-counting problem and B' an instance (assignment of variables) of B. We restrict the test cases to the set of all possible variable assignments for a fixed size $b = |B|$:

$$T_b = \{B' \mid \; b = |B|\} \qquad (4)$$

For each natural number b, there are 2^b elements in T_b with each serving as a test case. The *answer* to the bit-counting problem for a test case t is the number of *true* assignments in t. The least-significant bit of this answer provides the solution to the parity problem.

A program is supplied both t, encoded as a bit vector, and the problem size b (number of bits in the encoding that specify t) as inputs in registers.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

$$\texttt{NOT}(R_a) \equiv \left(\begin{array}{l} PC \leftarrow PC+1 \\ R_a \leftarrow \sim R_a \end{array} \right.$$

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

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

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

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

$$\texttt{JNZ}(R_a,\mathit{offset}) \equiv \left(PC \leftarrow \begin{cases} \min\left(\max\left(0, PC + \mathit{offset}\right), n\right) & \text{if } R_a \neq 0 \\ PC + 1 & \text{otherwise} \end{cases} \right.$$

$$\texttt{JLZ}(R_a,\mathit{offset}) \equiv \left(PC \leftarrow \begin{cases} \min\left(\max\left(0, PC + \mathit{offset}\right), n\right) & \text{if } R_a < 0 \\ PC + 1 & \text{otherwise} \end{cases} \right.$$

$$\texttt{JGZ}(R_a,\mathit{offset}) \equiv \left(PC \leftarrow \begin{cases} \min\left(\max\left(0, PC + \mathit{offset}\right), n\right) & \text{if } R_a > 0 \\ PC + 1 & \text{otherwise} \end{cases} \right.$$

Figure 1: Operational semantics for the virtual register machine VRM-$\mathcal{B}$. An arithmetic or logical operator on the right-hand side mostly inherits the C language's semantics of that operator. Expressions enclosed in ⟨brackets⟩ yield zero on exceptional cases (*e.g.*, divide-by-zero, invalid shift amounts).

Note that we train programs on a bit-count (and hence parity) problem of fixed set size b , but that many of the resultant solutions are exact and general for input sets of any size (§6).

5.2 Fitness Function

Since a program written in VRM-$\mathcal{B}$ need not terminate and terminating programs are desirable[2], we use a *lexicographic fitness function* (see [2]) of *two tiers* to select for programs that do so. A program's first-tier fitness is its correctness. Two programs with identical first-tier fitness are ranked using the second-tier fitness function which is a termination criterion. By separating the disparate criteria of correctness and termination, we can sidestep the *ad hoc* credit assignment problem that often plagues a single fitness function attempting to express multiple objectives.

For all program evaluations, a test case encoded as a bit vector is supplied in input register $\vec{R}_0$, the problem size b in register $\vec{R}_1$, and the program's result is taken from output register $\vec{R}'_2$. All other input registers are set to zero before evaluation.

First-tier correctness of program $\vec{I}$ on test case t for problem size b is computed by

$$\mathcal{F}_1^t\left(\vec{I}\right) \equiv |\vec{R}'_2 - \mathit{answer}(t)| \tag{5}$$

where, using (3), $\mathcal{E}_{\mathcal{B}}(\vec{I}, \vec{R}, K) = \vec{R}'$, input registers $\vec{R} \equiv \langle t, b, 0, \ldots, 0_{m-1} \rangle$, and K is the maximum number of evaluation steps. That is, the fitness of a program with regard to a single test case is the absolute magnitude of the difference of the correct answer, $\mathit{answer}(t)$, and the computed value in output register $\vec{R}'_2$. The total first-tier fitness of a program $\vec{I}$ is the sum of the individual test-case fitness values:

$$\mathcal{F}_1\left(\vec{I}\right) \equiv \sum_{t \in T} \mathcal{F}_1^t\left(\vec{I}\right) \tag{6}$$

Lower values indicate better first-tier fitness.

Second-tier termination of program $\vec{I}$ on test case t for problem size b is computed by:

$$\mathcal{F}_2^t\left(\vec{I}\right) \equiv \begin{cases} 1 & \text{if } \vec{I} \text{ terminates on } t \\ 0 & \text{otherwise} \end{cases} \tag{7}$$

Total second-tier fitness of $\vec{I}$ is:

$$\mathcal{F}_2\left(\vec{I}\right) \equiv \sum_{t \in T} \mathcal{F}_2^t\left(\vec{I}\right) \tag{8}$$

Second-tier fitness therefore is the number of test-cases in the test-case set T for which $\vec{I}$ terminates. In this case, higher values indicate better second-tier fitness.

[2]Two terminating programs may readily be composed into a larger terminating program, for example.

It is straightforward to extend this notion to secondary program properties other than termination. For example, code speed can be selected for by having a higher-tier function favor programs that obtain the same result by executing fewer instructions. Code size can be controlled in a similar manner. Other fitness functions could limit register, function-unit, or power usage.

5.3 Search Methods

Here we describe the search methods used to find solutions to the bit-counting and parity problems.

5.3.1 Genetic Search (GA-XO)

Genetic search uses tournament selection as its population-selection mechanism. Crossover and single-point mutation are the evolution operators.

Population Selection Population selection, for the construction of successive generations, is performed via k-tournament selection (see, *e.g.*, [9]). Let P be a population (set) of N individuals (VRM programs). To select a single individual from P, tournament selection examines k individuals in P and selects the one with best fitness. When the fitness function cannot distinguish a single best, one of the best is chosen at random. This mechanism is used to generate input pairs for the crossover operator, for example.

Lexicographic fitness functions (*cf.* [2]) are amenable to implementation via tournament selection. A two-tier fitness function may be integrated into a tournament as follows. Select two individuals at random. Rank them by first-tier fitness. If both first-tier fitness values are identical, break the tie by ranking them with their second-tier values. This generalizes to higher tiers in the obvious manner. Note that it is more difficult to implement lexicographic fitness in population selection methods that require ranking a significant portion of the population—such as in proportional selection (see, *e.g.*, [15])—since this requires not only computing order information for the entire population (*i.e.*, sorting it), but also for the equivalence classes induced by the lexicographic fitness functions.

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 $0 \le p_i < n$ in program $\vec{I}_i$. The length $k > 0$ of the subsequence is chosen randomly such that $p_i + k \le n$. A random point p_j, $0 \le p_j + k \le n$, 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 $Prob_{XO}$. 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.

Operator: Single-Point Mutation We conduct experiments with and without a single-point mutation operator that, with probability $Prob_{mutate}$, alters an instruction by changing its opcode or operand(s). Single-point mutation is applied after a new generation has been filled with selected individuals, but before its individuals are reevaluated.

5.3.2 Macro-Mutation (GA-MM)

Population-based macro-mutation is a search method that, instead of recombining fit individuals, mutates up to n consecutive instructions in a fit individual in order to improve it. Macro-mutation can be used to investigate whether program recombination affects (improves or worsens) search efficiency. Our GA-MM uses the "headless chicken" crossover proposed by Jones [11]. We also ran experiments that combined this operator and single-point mutation.

Operator: Headless-Chicken Crossover To isolate the *mechanism* of crossover (replacing a contiguous sequence of instructions) from the *idea* of crossover (inheriting useful "building blocks" from other individuals), Jones proposed and studied replacing conventional crossover with one that swaps information (*i.e.* instructions) with a randomly generated individual [11]. He calls this operation "headless chicken" crossover (HCXO). Others have investigated this form of macro-mutation in GP and EP systems [1, 3].

HCXO—as we apply it to machine language representations in this paper—selects values p_i, p_j, and k in the same manner as for conventional crossover (§5.3.1). However, instead of interchanging the k instructions, HCXO generates two sequences of k random instructions and inserts them in the two parents $\vec{I}_i$ and $\vec{I}_j$ at positions p_i and p_j. HCXO, like XO, thus generates two offspring but no information "crosses over" from one individual to another. HCXO macro-mutation is applied to the population with probability $Prob_{MM}$.

Operator: Single-Point Mutation Though HCXO sometimes functions as single-point mutation (when the sequence length is randomly chosen as $k = 1$), we investigate the effect—on search efficiency—of an additional single-point mutation applied after HCXO. Single-point mutation for the macro-mutation search is implemented as above (§5.3.1).

	#Solns	#Evals	$\frac{\#\text{Solns}}{\#\text{Evals}}$
GA-XO	9	(max-evals)	9.0×10^{-10}
GA-XO-SPM	10	5.6×10^{9}	1.8×10^{-9}
GA-MM	10	2.1×10^{9}	4.8×10^{-9}
GA-MM-SPM	10	1.9×10^{9}	5.3×10^{-9}
Random	1	(max-evals)	1.0×10^{-10}

Table 1: Results of crossover, crossover with single-point mutation, macro-mutation, macro-mutation with single-point mutation, and random search applied to the bit-counting problem. For each search method, the number of solutions discovered, number of evaluations required to discover the solutions, and efficiency (solutions per evaluation) are given. Experiments were limited to the first of either ten solutions or 1×10^{10} (max-evals) evaluations.

5.3.3 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.

6 Results

Here we describe the parameters for the search experiments, report results, and comment on some sample evolved programs.

We ran four population based searches: GA with crossover (GA-XO), GA with crossover and single-point mutation (GA-XO-SPM), GA with macro-mutation (GA-MM), and GA with macro-mutation and single-point mutation (GA-MM-SPM). Population size was $N = 8192$. The initial population was created randomly. Binary tournament selection was used. Crossover (macro-mutation) probability was set to $Prob_{XO} = 25\%$ ($Prob_{MM} = 25\%$). That is, a quarter of the population was generated with crossover (macro-mutation) and the remaining three quarters was filled by the unmodified[3] winners of tournaments. In the experiments using it, single-point mutation was applied to $probMutate = 0.1\%$ of the instructions in the population. A run was deemed complete after 50 consecutive generations without improvement (stasis).

VRM parameters were set as follows: program size $n = 20$, number of registers $m = 10$, maximum number of instructions executed $K = 100$. From Equation 2 the

[3]Programs not modified between successive generations do not require reevaluation and are hence not counted in the total number of evaluations.

size of the search space measured in syntactic programs is approximately 2^{219}.

The test-case set contained all instances of the 5-parity problem. Using the earlier notation, $b = 5$ giving 2^5 test cases.

Table 1 contains the search results including random search. We limited an experiment to 1×10^{10} evaluations or to ten general solutions to curb both run time and manual verification time. The stasis value of 50 along with this upper bound on the number of evaluations created a variable number of runs per experiment; a typical experiment contained hundreds of runs. We count a program as a *solution* in the table if it produces the correct answer and terminates for every test case. Recall that the least-significant bit of such a solution's result solves the parity problem. We verified that the solutions are general for all bit-vector inputs (register $\mathtt{R}_0$) provided the VRM's registers are sufficiently large. Since registers are signed 32-bit integers in this implementation, bit-counts and parity of 31-bit sets can be computed with the evolved programs. By increasing the size of registers, the evolved solutions can directly be used to solve arbitrarily large instances. Solutions rarely exploited the problem size b supplied in register $\mathtt{R}_1$ in producing bit-counting solutions. Some, however, used this value as a constant to govern, for example, control flow. Some solutions, therefore are general with respect to the encoding in $\mathtt{R}_0$, but require $\mathtt{R}_1$ to be fixed at $b = 5$. This problem can perhaps be avoided by including test instances of various sizes in the test suite.

In addition to the number of general solutions found, the table lists the number of evaluations used and the search efficiency. Search efficiency is the ratio of the number of solutions to the number of evaluations; this ratio ranges from zero (ineffective) to one (highly effective). We first note that random search is rather ineffective for this problem and VRM parameter settings. GA-MM performed well overall; its performance increased slightly in the presence of single-point mutation (GA-MM-SPM). Crossover (GA-XO) too solved the problem rather effectively, but gained more from single-point mutation (GA-XO-SPM) than the macro-mutation search. One explanation is that GA-MM already contains single-point mutation in its random k-instruction mutation operation. It would be interesting to examine whether macro-mutations of a certain size (they now range from one to the program size) are more effective than others.

Three sample solutions found with GA-MM are in Figure 2. Instructions that do not influence the bit-counting results have been crossed out. Solutions tended to fall into one of about five types; three are represented by the sample solutions. Figure 2 also contains C code that implements the three depicted types of bit-count algorithm. Type A iterates through the vector checking one bit at a time and counting the number of set bits. It has time complexity proportional to the position of the most significant set bit. Note that this particular type A example is general but requires the problem size supplied in register $\mathtt{R}_1$ to be odd (see discussion above). Type B repeatedly uses subtraction in conjunction with `AND` to remove the least-significant set bit. Its time complexity is proportional to the number of set bits. Type B solutions predominate (34 of the 40 are type B). Type C solutions first copy the bit-vector into an "accumulator" and iteratively subtract from this accumulator the vector successively divided by two. Since integer division drops remainders of one, the accumulator will hold the sum of such remainders, which corresponds to the number of set bits in the input vector.

Type A and B solutions are given as a conventional programming example and exercise in the literature [12]. We posit that type C solutions are perhaps novel algorithms for bit-counting, but this a daunting proposition to prove in the positive. We have not yet found type C examples in literature, code, or folklore.

7 Summary

Population-based search—using either crossover or macro-mutation—can find machine-language programs that solve the bit-counting and parity problems in general. Importantly, this paper's findings demonstrate that a machine-language representation *without domain-specific operators* can readily find *general solutions* to these problems. For this problem domain and a specific virtual-machine instruction set, we find that macro-mutation substantially outperforms crossover. This raises the question of how beneficial communication among partial solutions (*i.e.*, genetic crossover) is to such problems' solution. This paper's macro-mutation results imply that the parity and bit-counting problems—as formulated herein—do not contain sufficient "building blocks" for crossover to exploit. This does not mean, however, that building blocks do not appear in machine-language programs; it is plausible and likely that evolution of larger programs solving more difficult problems can, or must, exploit building blocks. An obvious direction is to apply both macro-mutation and crossover to larger and harder program-induction problems.

Acknowledgments

Thanks to Chris Fraser, Brian Kernighan and Ed Rietman for acute comments, to Andy Pargellis for stimulating discussions on evolution, and to the anonymous referees for helpful suggestions.

Type A
(generation 387)

Address	Instruction
0:	J(+13)
1:	~~JLZ(R_5,+14)~~
2:	~~SET(R_2)~~
3:	~~OR(R_9,R_0)~~
4:	AND(R_4,R_1)
5:	~~NOP~~
6:	MOV(R_1,R_0)
7:	~~XOR(R_6,R_7)~~
8:	SHR(R_0,R_5)
9:	JNZ(R_4,+4)
10:	INC(R_2)
11:	~~SUB(R_2,R_4)~~
12:	~~MOV(R_4,R_8)~~
13:	SET(R_4)
14:	~~JNZ(R_3,-8)~~
15:	NOT(R_1)
16:	OR(R_5,R_4)
17:	JZ(R_0,+16)
18:	JZ(R_8,-15)
19:	~~JNZ(R_1,+13)~~

```
int bitcount_typeA(int x)
{
  int b;

  for (b = 0; x != 0; x >>= 1)
    if (x & 0x1) b++;
  return b;
}
```

Type B
(generation 263)

Address	Instruction
0:	MOV(R_7,R_0)
1:	JZ(R_0,+16)
2:	~~DEC(R_4)~~
3:	~~CLR(R_8)~~
4:	~~NOP~~
5:	~~JGZ(R_8,+6)~~
6:	~~AND(R_6,R_4)~~
7:	~~DIV(R_3,R_8)~~
8:	DEC(R_0)
9:	~~OR(R_3,R_0)~~
10:	~~MUL(R_9,R_6)~~
11:	AND(R_0,R_7)
12:	~~AND(R_1,R_5)~~
13:	~~SET(R_3)~~
14:	INC(R_2)
15:	J(-16)
16:	~~OR(R_0,R_4)~~
17:	~~NOP~~
18:	~~XOR(R_3,R_2)~~
19:	~~CLR(R_0)~~

```
int bitcount_typeB(int x)
{
  int b;

  for (b = 0; x != 0; b++)
    x &= x-1;
  return b;
}
```

Type C
(generation 220)

Address	Instruction
0:	~~JNZ(R_9,+1)~~
1:	SET(R_6)
2:	MOV(R_2,R_0)
3:	~~JLZ(R_7,-12)~~
4:	SHR(R_0,R_6)
5:	~~JLZ(R_5,+11)~~
6:	SUB(R_2,R_0)
7:	~~SHL(R_4,R_7)~~
8:	~~AND(R_8,R_6)~~
9:	~~SUB(R_4,R_6)~~
10:	~~SHR(R_3,R_3)~~
11:	~~NOT(R_3)~~
12:	JNZ(R_0,-8)
13:	~~DEC(R_8)~~
14:	~~MUL(R_7,R_1)~~
15:	~~SUB(R_8,R_3)~~
16:	~~J(+16)~~
17:	~~ADD(R_4,R_9)~~
18:	~~J(+10)~~
19:	~~NOT(R_8)~~

```
int bitcount_typeC(int x)
{
  int b;

  for (b = x; x != 0; b -= x)
    x >>= 1;
  return b;
}
```

Figure 2: Sample solutions to bit-counting found using population-based macro-mutation search. The input bit-vector is supplied in R_0 and the problem size in R_1. The bit-count result is taken from R_2; its least-significant bit solves the parity problem. Instructions that do not affect the bit-count result have a ~~line~~ through them. The C functions embody the corresponding bit-counting algorithms.

References

[1] P. J. Angeline. Subtree crossover: Building block engine or macromutation? In *Proceedings of the Second Conference on Genetic Programming*, pages 9–17, July 1997.

[2] A. Ben-Tal. Characterization of Pareto and lexicographic optimal solutions. In G. Fandel and T. Gal, editors, *Multiple Criteria Decision Making Theory and Application*, pages 1–11. Lecture Notes in Economics and Mathematical Systems, No. 177, Springer-Verlag, 1979.

[3] K. Chellapilla. Evolutionary programming with tree mutations: Evolving computer programs without crossover. In *Proceedings of the Second Conference on Genetic Programming*, pages 240–248, July 1997.

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

[5] L. J. Fogel, A. J. Owens, and M. J. Walsh. *Artificial Intelligence through Simulated Evolution.* New York: John Wiley, 1966.

[6] R. M. Friedberg. A learning machine: Part I. *IBM Journal of Research and Development*, 2:2–13, 1958.

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

[8] C. Gathercole and P. Ross. Tackling the boolean even n parity problem with genetic programming and limited-error fitness. In *Proceedings of the Second Conference on Genetic Programming*, pages 119–127, July 1997.

[9] D. E. Goldberg and K. Deb. A comparitive analysis of selection schemes used in genetic algorithms. In G. Rawlins, editor, *Foundations of Genetic Algorithms.* Morgan Kauffman, 1991.

[10] L. Huelsbergen. Learning recursive sequences via evolution of machine-language programs. In *Proceedings of the Second Conference on Genetic Programming*, pages 186–194, July 1997.

[11] T. Jones. Crossover, macromutation, and population-based search. In *Proceedings of the Sixth International Conference on Genetic Algorithms*, pages 73–80, 1995.

[12] B. W. Kernighan and D. M. Ritchie. *The C Programming Language.* Prentice-Hall, second edition, 1988.

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

[14] J. Koza. *Genetic Programming II.* MIT Press, Cambridge, MA, 1994.

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

[16] 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.

[17] M. L. Wong and K. S. Leung. Learning recursive functions from noisy examples using generic genetic programming. In *Proceedings of the First Conference on Genetic Programming*, pages 238–246, July 1996.