, , 1–21 ()


# Solving High-Order Boolean Parity Problems with Smooth Uniform Crossover, Sub-Machine Code GP and Demes

RICCARDO POLI AND JONATHAN PAGE {r.poli, j.page}@cs.bham.ac.uk
*School of Computer Science, The University of Birmingham, Birmingham, B15 2TT, UK*

**Communicated by:**

**Abstract.** We propose and study new search operators and a novel node representation that can make GP fitness landscapes smoother. Together with a tree evaluation method known as sub-machine code GP and the use of demes, these make up a recipe for solving very large parity problems using GP. We tested this recipe on parity problems with up to 22 input variables, solving them with a very high success probability.

## 1. INTRODUCTION

The even-$n$-parity functions have long been recognised as difficult for Genetic Programming (GP) to induce if no bias favourable to their induction is introduced in the function set, the input representation, or in any other part of the algorithm. For this reason they are very interesting and have been widely used as benchmark tests [1, 4, 5, 6, 7, 23, 24, 26]. For an even-parity function of $n$ Boolean inputs, the task is to evolve a function that returns 1 if an even number of the inputs evaluate to 1, 0 otherwise. The task seems to be difficult for at least two reasons. Firstly, the function is extremely sensitive to changes in the value of its inputs, since flipping a single input bit reverses the output. Secondly, the function set that is usually used by GP researchers attempting to induce it, {`OR, AND, NOR, NAND`}, has an inbuilt bias against parity problems since it omits the building block functions `EQ` and `XOR`, either of which can be used to construct parsimonious solutions [9].

The difficulty of a GP problem is usually evaluated by estimating the number of individuals required to solve it with a specified probability, a quantity known as effort (for the calculations involved, see [6]). In [7], Koza showed not only that the even-$n$-parity problems are difficult for standard GP to solve, but that the effort required to solve them rises by about an order of magnitude for each increment of $n$.

The estimation of effort takes no account of the amount of processing required to evaluate a single individual. This generally increases with tree size and since it is well known that candidate solutions often have a tendency to bloat [10], the amount of processing time spent on population evaluation frequently increases as the run progresses. More significant to the research reported here is that large-scale problems often require the evaluation of the same tree over many more fitness cases than smaller problems. This is particularly true of Boolean induction tasks where the number of possible fitness cases increases exponentially with the number

of input variables. Thus, for $n = 6$, there are 64 fitness cases. This rises to 4096 for $n = 12$ and 4,194,304 for $n = 22$. For this reason, solving Boolean induction problems using the standard technique of evaluating the entire tree once for each fitness case has only proved feasible for low numbers of input variables.

In this paper we describe three distinct methodologies designed to solve large-scale problems such as the high-order even-$n$-parity function.

- We improve the search process using the GP uniform crossover and point mutation operators described in [20] and a *sub-symbolic node representation* [16] which uses an enlarged, unbiased function set and allows GP to make small, directed movements around the program space.
- We apply a technique known as Sub-Machine Code GP [18, 21], which allows the parallel evaluation of 32 or 64 fitness cases per program execution.
- We employ a parallel implementation in which GP sub-populations, or demes, are distributed over a number of workstations.

These methods are not mutually exclusive and we demonstrate how they may be combined to produce a single recipe for solving large-scale problems[1]. Finally, we show that the even-$n$-parity problem with $12 \leq n \leq 22$ can be solved with this recipe. To our knowledge, this is the first time that the even-22-parity problem, or indeed any problem of comparable size, has been successfully tackled by a GP implementation that does not make use of recursion.

The remainder of the paper is organised as follows. In the next section, we review other attempts to solve the even-$n$-parity problems using GP. In Section 3 we describe our representation and operators. Section 4 reviews some of our early studies in which we applied the sub-symbolic node representation to two Boolean classification tasks. In Section 5 we describe the sub-machine code technique and, in Section 6, the deme model used. Finally, in Sections 7 and 8 we present and discuss the results of experiments on the even-$n$-parity problem for $n = 12, 13, 15, 17, 20$ and 22 and draw some conclusions.

## 2. RELATED WORK

Koza provided a detailed treatment of the even-$n$-parity problem in his extensive discussions of the standard GP paradigm [6] and Automatically Defined Functions (ADFs) [7]. In both cases he restricted himself to the four arity 2 Boolean functions `AND, OR, NAND, NOR` and whatever input terminals were necessary for the problem. As we have seen, the omission of the `XOR` and `EQ` primitives from the set creates additional problems for standard GP, since it must independently evolve semantically equivalent blocks of code at numerous locations throughout the program tree. Unsurprisingly, solving the parity problems using standard GP without ADFs is computationally expensive and Koza was unable to obtain a result for values of $n > 5$. Of course, the discovery and reuse of building blocks is the idea behind ADFs, and Koza reported greater success when they were used. The solutions he reported for $n = 3, 4, 5, 6, 7, 8, 11$ all evolved, and made extensive use

*Table 1.* Minimum efforts (in thousands of fitness evaluations) required to solve the even-$n$-parity problem using various methods.

| | $N$ | | | | |
|---|---|---|---|---|---|
| Approach | 5 | 6 | 7 | 8 | 9 |
| Standard GP [7] | 6,528 | 70,176[2] | n/a | n/a | n/a |
| GP + ADFs [7] | 464 | 1,344 | 1,440 | solved[3] | |
| EP [4] | 2,100 | n/a | n/a | n/a | n/a |
| EP + ADFs [4] | 126 | 121 | 169 | 321 | 586 |
| GP + ADFs [1] | 359 | 627 | n/a | n/a | n/a |

of, code equivalent to either the `XOR` or `EQ` primitives on their function-defining branches. The estimated computational effort for these and, where available, the other studies discussed here are given in Table 1.

Chellapilla [4] essentially replicates these studies using Evolutionary Programming (EP), omitting the crossover operator and using instead a variety of mutation operators. His results compare favourably with those of Koza and he uses them to argue that the significance of crossover has been overstated.

Aler [1] presents a modification of Koza's ADF technique in which function and result-producing branches are evolved in separate populations. The main branch-evolving population uses the ADF of the best individual from the ADF population of the previous generation and vice versa. His results on the even-5 and -6-parity functions also compare well with those of Koza.

Gathercole and Ross [5] use GP *without* ADFs and a fitness function in which evaluation of the individual ceases once a given number of fitness cases have been misclassified. If this threshold is reached, the remaining, untested, cases are also judged as misclassifications. The result is that low-fitness programs can be identified without being evaluated on the full training set, with considerable savings in CPU time. The order of presentation of the fitness cases is modified at run-time so that the evolving population is exposed to cases of increasing difficulty. Gathercole and Ross's results are very encouraging although they do not estimate the effort.

The studies discussed so far have all treated the even-$n$-parity problem for different values of $n$ as distinct tasks and evolved specific solutions for each. Wong and Leung [26] describe an alternative approach in which logic programs are evolved. Programs operate recursively on a list of Boolean values (each list representing a single fitness case), rather than a set of distinct input terminals. Whilst the solutions reported by Wong and Leung are general solutions to the even-$n$-parity problem, their method is necessarily heavily constrained to avoid infinite recursions and contains a great deal of problem-specific information. Another strongly-biased approach is that of Yu [27] who evolves recursive general solutions to the even-$n$-parity problem using lambda abstractions.

A brief examination of Table 1 shows that, as $n$ increases, the problem rapidly becomes very difficult for GP systems which use the traditional function set without any favourable inbuilt bias. For $n \geq 6$, the effort is at least of the order of $10^5$, and for many methods the problem becomes intractable. In the next section we describe a representation and operators that can improve the situation considerably.

## 3. OPERATORS AND REPRESENTATION

### *3.1. UNIFORM CROSSOVER*

GP Uniform crossover (GP-UX)[20], as the name suggests, is a GP operator inspired by the GA operator of the same name [25]. GA uniform crossover (GA-UX) constructs offspring on a bitwise basis, copying each allele from each parent with a 50% probability. Thus the information at each gene location is equally likely to have come from either parent and on average each parent donates 50% of its genetic material. The whole operation, of course, relies on the fact that all the chromosomes in the population are of the same structure and the same length. No such assumption can be made in GP since the parent trees will almost always contain unequal numbers of nodes and be structurally dissimilar.

GP uniform crossover begins with the observation that many parse trees are at least partially structurally similar. This means that if we start at the root node and work our way down each tree, we can frequently go some way before finding function nodes of differing arity at the same locations. Furthermore we can swap every node up to this point with its counterpart in the other tree without altering the structure of either. Working down from the root node, we can define two regions of a pair of trees as follows. Any node in one tree having a corresponding node at the same location in the other is said to be located within the *common region.* Those pairs of nodes within the common region that have the same arity are referred to as *interior.* The interior nodes and the common region of two trees are illustrated in Figure 1. Note that the common region necessarily includes all interior nodes. GP-UX is then as follows. Once the interior nodes have been identified, the parent trees are both copied. Interior nodes are selected for crossover with some probability $p_s$ per node. Crossover involves exchanging the selected nodes between the trees, with those nodes not selected for crossover remaining unaffected. Non-interior nodes within the common region can also be crossed, but in this case the nodes and their subtrees are swapped. As in GA-UX, the value of $p_s$ is generally set to 0.5, resulting in an exchange of 50% of the nodes [20]. The result of GP-UX applied to the trees in Figure 1 is shown in Figure 2.

GP-UX, like GA-UX, is a homologous operator in that it preserves the position of genetic material in the genotype. This is a beneficial property but in some cases it can lead to the phenomenon of lexical convergence whereby a sub-optimal gene becomes fixed at a given location. When this happens, crossover cannot introduce the optimal gene and for this reason it is generally desirable to include a mutation operator to maintain diversity in the population. The operator we use here – GP point mutation (GP-PM)[11] – is also inspired by its GA counterpart (GA-PM).

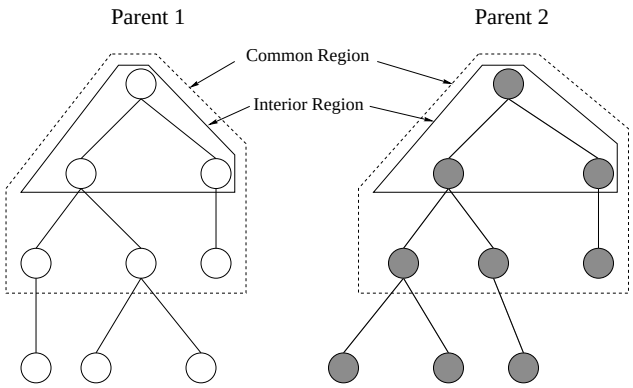


*Figure 1.* Two parental parse trees prior to GP-UX.

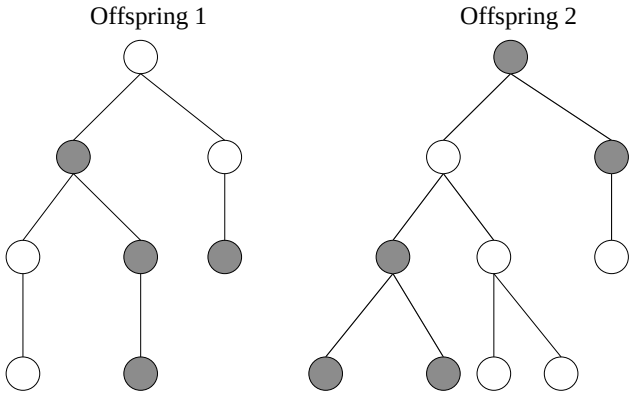


*Figure 2.* Offspring trees after GP-UX.

GP-PM substitutes a single function node with a randomly selected replacement of the same arity. As in GA-PM, the average number of mutations performed on an individual is a function of the program size and a user-defined mutation rate, $p_m$. Since in GP program lengths vary, larger programs undergoing mutation will, on average, be perturbed to a greater degree than smaller ones. Since such perturbations are generally detrimental to the fitness of a highly-evolved program, this will generate an emergent parsimony pressure [19].

### *3.2. SUB-SYMBOLIC NODE REPRESENTATION*

Whilst a single point mutation is the smallest syntactical operation that can be applied to a parse tree under the standard representation, it may nevertheless result in a significant change in behaviour. For example, consider the following subtree: (`AND` $T_1$ $T_2$) where $T_1$ and $T_2$ are Boolean input terminals. If the `AND` node is replaced with `NAND`, the value returned by the subtree will be altered in all the fitness cases. Controlling this means addressing the mechanism used to replace the node. Our solution is simple. We begin by noting that a Boolean function of arity $n$ can be represented as a truth table (bit-string) of length $2^n$, specifying its return value on each of the $2^n$ possible inputs. Thus `AND` may be represented as `1000`, `OR` as `1110`. We refer to this representation as *sub-symbolic* because function nodes are now seen as collections of entities rather than atomic units.

One feature of the sub-symbolic representation of Boolean function nodes is that, in contrast to the reduced function set normally used in Boolean classification tasks, it is unbiased, since it incorporates all $2^n$ nodes of arity $n$ into its function set. Some of these may seem superfluous (e.g. `always-ON` and `always-OFF`) but the effect they have on performance is poorly understood[4].

Of course, the choice of function set is problem specific and often something of an art. In his studies of the parity problems, Koza restricted himself to the function set `AND, OR, NAND, NOR`, presumably because it combined minimality with completeness (in the sense that solutions to any Boolean function can be constructed from the primitives). However, omitting the `XOR` and `EQ` functions undoubtedly makes life harder for GP, as can be seen from the regularity with which Koza's ADF system evolved them.

Even with knowledge of useful primitives, we should be careful not to minimise the size of the function set excessively. Langdon and Poli [9] have shown that programs constructed exclusively from `EQ` (for even values of $n$) and `XOR` (for odd $n$) are either solutions to the even-$n$-parity problem or score exactly half marks. In other words, the fitness landscapes of such representations offer no gradient information for GP to follow.

In this work, our principal reasons for including all Boolean functions of a given arity in our set are simplicity – to do otherwise would require constraining the smooth search operators (described in the next section) in some way – and the desire to avoid any favourable or unfavourable bias. In doing so, we note that the `EQ` and `XOR` functions are necessarily included in the arity 2 function sets and that these will probably enhance performance on the parity problems. On the other

hand, the function set is much larger than normal and contains several extraneous functions.

### *3.3. SMOOTH OPERATORS*

The binary representation of function nodes means that we can define a point mutation operator which works in much the same manner as GA point mutation: a single randomly-selected bit is flipped in a randomly-selected node. In addition, since GP-UX is homologous, we can extend it to use a GA crossover operator within the nodes at reproduction (in the experiments reported here we use GA-UX). The crossover operation is illustrated in Figure 3. When a pair of interior nodes are selected for crossover, GA uniform crossover is applied to their binary representations. In other words, the bits specifying each node's function are swapped with probability $p_b = 0.5$. Clearly such an operator interpolates the behaviour of the parents' corresponding nodes, rather than exchanging nodes in their entirety. The sub-symbolic node representation allows GP to move around the solution space in a smoother, more controlled manner and hence we refer to these versions of the operators as *smooth point mutation* (GPSPM) and *smooth uniform crossover* (GP-SUX). It should be noted that the fact that these operators can perform very small changes does not necessarily mean that they do so all the time. For example, at the beginning of a run we expect GP-SUX to be a global search operator like its close relative GP-UX [20], although to a lesser extent. We also expect GP-SUX to become more and more local as the run progresses and to do so to a greater extent than GP-UX.

## 4. RESULTS USING THE SUB-SYMBOLIC REPRESENTATION

We saw in Section 3.2 that the sub-symbolic representation lends itself particularly well to the encoding of Boolean functions. Our initial studies therefore dealt with Boolean classification tasks, specifically the even-$n$-parity problem and the $n$-multiplexer, both of which have been widely-used as benchmark problems for GP.

The $n$-multiplexer accepts $n$ input bits, of which $p$ encode an address and the remaining $q = 2^p$ are data. The address bits are converted to an integer address and the task is to return the value of the data bit associated with that address. In the case where $n = 3$ (i.e. $p = 1$) the task is equivalent to evolving an `IF-THEN-ELSE` function and it is therefore unsurprising that in higher-order versions of the problem, the `IF` function is a useful building block and most attempts to solve the $n$-multiplexer using GP have included it in the function set (see, for example, [6] [8]).

Arity 3 functions such as `IF` can be represented in the same way as arity 2 nodes. The truth table of an arity $n$ function contains $2^n$ bits and there are therefore $2^{2^n}$ possible functions. This means that there are 256 functions of arity 3, with `IF-A-THEN-B-ELSE-C` evaluating to rule 83 (assuming the most significant bit encodes the case where A = B = C = 0). The version of GP-SUX described above

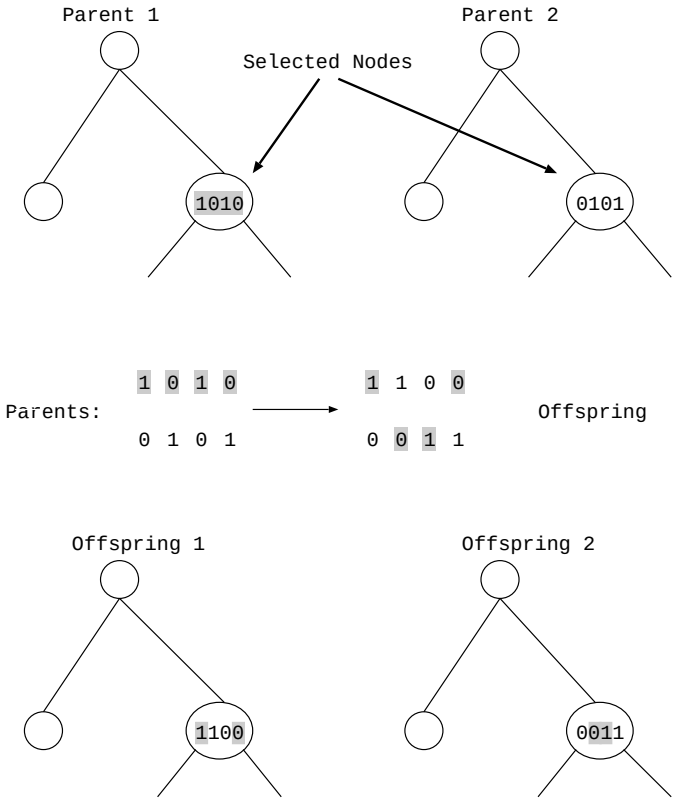


*Figure 3.* Uniform crossover on the sub-symbolic representation.

can be used with a mixture of arity 2 and arity 3 functions but in our studies we did not combine them for two reasons:

1. We did not want to provide a favourable bias to GP.
2. Arity 2 functions are to some extent special cases of arity 3 functions (when one of the inputs does not influence the output).

The set of all arity 3 functions is, by GP standards, a very large set although it does reduce the amount of problem-specific information given to the system from the outset. Our aim in investigating the $n$-multiplexer was to examine the impact this had on standard GP and to see whether uniform crossover and the sub-symbolic representation afforded any advantages in searching so large a space.

Using a function set of all 256 Boolean nodes of arity 3, we compared performance of standard GP without ADFs, uniform crossover with standard node representation (GP-UX), and uniform crossover with the sub-symbolic node representation (GP-SUX) on the 6-multiplexer problem. We used populations of 50 and 500 individuals and ran each condition 50 times, terminating the run if no solution had been

found by generation 100. We used the ramped half-and-half initialisation method with a maximum depth of 5 levels. Since GP-UX and GP-SUX cannot increase tree size beyond that of the deepest full tree at initialisation, this put an upper limit of 364 nodes on tree size in these conditions. In the standard GP conditions, the upper limit on tree depth was set to 17 levels. We set the crossover and mutation probabilities – $p_c$ and $p_m$ – to 0.9 and 0.01 respectively for standard GP, as advised in [6]. However, pilot studies indicated that the optimal values for the subsymbolic representation are $p_c = 0.3$ and $p_c = 0.1$, and these were the values used here. The probabilities of exchanging nodes or bits during GP-UX and GP-SUX were both set to 0.5.

The results for the 6-multiplexer are shown in Table 2. By way of comparison, Koza [6][p.196] reports a minimum effort for standard GP on the 6-multiplexer of 245,000, based on a population of 500 individuals, but this was achieved using the strongly-biased function set of {`IF, NOT, OR, AND`}.

*Table 2.* Minimum efforts for the 6 Multiplexer using all 256 Boolean functions of arity 3.

| Method | Pop. size | Effort |
|---|---|---|
| Std. GP | 50 | 1,117,200 |
| Std. GP | 500 | 216,000 |
| GP-UX | 50 | No Solution Found |
| GP-UX | 500 | 1,938,000 |
| GP-SUX | 50 | 50,460 |
| GP-SUX | 500 | 164,000 |

Given a sufficiently large population, standard GP has little difficulty solving the 6-multiplexer. In fact, performance is slightly better than the effort given by Koza, despite the large function set. It is conceivable that the unbiased inclusion of all the arity 3 functions offered GP many additional building blocks not present in the standard function set. The issue of function sets, particularly with respect to pre-defined building blocks and extraneous functions, is under-investigated in the GP literature and further work is required to explain these effects more fully.

GP-SUX outperformed standard GP on both population sizes. Curiously, unlike standard GP crossover, GP-SUX seems to gain from the use of small populations. GP-UX performed remarkably badly, however, struggling to solve the problem even with populations of 500 individuals. We speculate that this may reflect differences in the nature of the search that the operators perform, a point to which we return below.

The good performance of GP-SUX with small population sizes was confirmed on the even-$n$-parity problems. We compared the same operators on the even-3, -4 and -6 parity problems using a function set of all 16 arity 2 Boolean nodes. We performed 50 runs on each condition using populations of 50 individuals. Again, this unusual choice of parameter was based on the results of pilot studies, which

*Table 3.* Minimum efforts for the even-3, -4 and -6 parity problems using all 16 Boolean functions of arity 2.

| Method | Even-3 | Even-4 | Even-6 |
|---|---|---|---|
| Std. GP | 5,550 | 11,250 | No solution found |
| GP-UX | 850 | 4,200 | 34,850 |
| GP-SUX | 900 | 2,250 | 17,000 |

indicated that GP-SUX was quite capable of solving the parity problems for $n < 7$. The remaining parameters were as for the multiplexer experiments.

The results of these experiments are shown in Table 3 (for more results, see [16]). Here we see a marked superiority of GP-UX over standard GP, and a further improvement on the part of GP-SUX over GP-UX.

These are curious results – the 6-multiplexer is known to be an easier problem for standard GP than the parity functions, whilst it is apparent that for GP-UX and GP-SUX the opposite is true. In GP-UX and GP-SUX runs on the multiplexer, the population frequently converged on trees too small to solve the problem and, since tree size could not then increase, these runs failed. The most dramatic example was when the population converged on trees consisting of a single terminal node (any of the output bits in isolation score 40 out of 64 hits). These single node trees can present a particular problem for GP-SUX and GP-UX since they cannot be altered by crossover and can therefore only be eliminated by selection. The same does not apply to standard GP crossover and in fact the pressure to bloat allowed standard GP to avoid these deceptive attractors.

Whilst GP-SUX and GP-UX were both susceptible to the influence of the deceptive attractors, GP-SUX still outperformed GP-UX on the 6-multiplexer. This is almost certainly due to differences in the nature of the search performed later in the run. A single application of the GP-PM operator on the arity 3 function set changes, on average, 4 of the 8 bits whereas, in contrast, a single application of the GP-SPM operator changes just 1. GP-PM, with the probability used here, therefore maintains a higher node diversity in the population than GP-SPM. In addition, GP-UX swaps nodes in their entirety, rather than interpolating them. GP-UX and GP-PM are therefore unable to perform a very local search, even at the later stages of the run. GP-SUX and GP-SPM are able to perform far more local search towards the end of the run, and we believe that it is the ability to do this that gives the superior performance on the multiplexer.

Standard GP crossover is also a local search operator which means that standard GP behaves like a set of parallel hill-climbers [17]. This fact, together with the previous observations, leads us to believe that the 6-multiplexer is not, as has been stated [3][p.154], unimodal. There are a number of weakly deceptive attractors, although the pressure to bloat means that these generally exert little influence over standard GP.

GP-UX and GP-SUX almost never converged on insufficiently large trees during parity runs. This is because any tree that does not include all $n$ terminals in its coding branches scores exactly half marks. Deceptive attractors of the kind seen in the multiplexer therefore do not exist. Our experiments suggest that at a coarse resolution (i.e. that on which global search operators work), the landscape contains some form of gradient information which can be used to locate better regions of the search space on which to perform local search. GP-UX performed better on this problem both because of the necessity for global search operators and because the arity 2 function set allowed it to perform more local search later in the run. However standard GP, being a purely local search operator, was unable to exploit this information.

The even-$n$-parity functions therefore form a class of problems well suited to operators such as GP-SUX. The ease with which GP-SUX solved the even-6-parity function, a problem which had been considered very difficult for standard GP, led us to ask whether solutions to higher order versions might also be efficiently obtained. Since the number of fitness cases that must be evaluated with each increment of $n$ rises exponentially and can make prohibitive demands on processing power, we used a simple method for parallel evaluation of fitness cases that significantly reduces these demands. This method is described in the next section.

## 5. SUB-MACHINE-CODE GP

Most computer users consider their machines as sequential computers. However, CPUs can be seen as parallel Single Instruction Multiple Data (SIMD) processors made up of many interacting 1-bit processors. In a modern CPU some instructions, such as Boolean operations, are performed in parallel and independently for all the bits in the operands. For example, the bitwise AND operation (see Figure 4(a)) is performed internally by the CPU by concurrently activating a group of AND gates within the arithmetic logic unit as indicated in Figure 4(b). In other instructions the CPU 1-bit processors interact through communication channels.

If we see the CPU as a SIMD computer, then we could imagine that each of its 1-bit processors will be able to produce a result after each instruction. Most CPUs do not allow handling single bits directly. Instead all the values to be loaded into the CPU and the results produced by the CPU are packed into bit vectors, which are normally interpreted as integers in most programming languages. For example, in many programming languages the user will see a bitwise AND operation as a function which receives two integers and returns an integer, as indicated in Figure 4(c).

All this powerful parallelism inside our CPUs has been ignored by most of the GP community so far. The most notable exception to this is the work on CGPS/AIMGP [12, 15, 13, 14] which exploits the CPU is its entirety.

Sub-Machine-Code GP (SMC-GP) exploits this parallelism to do GP by making the CPU execute the same program on different data in parallel and independently. This can be done as follows:

1. The function set includes operations which exploit the parallelism of the CPU, e.g. bitwise Boolean operations.

2. The terminal set includes integer input variables and constants, to be interpreted as bit vectors where each bit represents the input to a different 1-bit processor. For example, the integer constant 21, whose binary representation is 00010101 (assuming an 8-bit CPU), would be seen as 1 by the 1-bit processors processing bits 1, 3 and 5. It would be seen as 0 by all other 1-bit processors.

3. The result produced by the evaluation of a program is interpreted as a bit vector, each bit of which represents the result of a different 1-bit processor. E.g. if the output of a GP program is the integer 13, this should be converted into binary (obtaining 00001101) and decomposed to obtain 8 binary results (assuming an 8-bit CPU).

An ideal application for this paradigm is to evaluate multiple fitness cases in parallel. Boolean induction problems lend themselves to this use of sub-machine-code GP. The approach used is a specialisation of the general approach described above:

1. Bitwise Boolean functions are used.

2. Before each program execution the input variables are initialised so as to pass a different fitness case to each of the different 1-bit processors of the CPU.

3. The output integers produced by a program are unpacked and each of their bits is interpreted as the output for a different fitness case.

In Figure 5 we provide a simple C implementation of this idea which demonstrates the changes necessary to do sub-machine-code GP when solving the even-5 parity problem. The function `run()` is a simple recursive interpreter capable of handling variables and a small number of Boolean functions. The interpreter executes the program stored in prefix notation as a vector of bytes in the global variable `program`. The interpreter returns an unsigned long integer which is used for fitness evaluation. The function `e5parity()` computes the target output in the even-5 parity problem for a group of 32 fitness cases. The function `even5_fitness_function(char *Prog)` executes a program once and returns the number of entries of the even-5 parity truth table correctly predicted by the program. More implementation details are available in [21]. A more extended C code fragment is available via anonymous ftp from `ftp.cs.bham.ac.uk` in the directory `/pub/authors/R.Poli/code/`.

In practical terms this evaluation strategy means that all the fitness cases associated with the problem of inducing a Boolean function of $n$ arguments can be evaluated with a *single program execution* for $n \leq 5$ on 32 bit machines, and $n \leq 6$ on 64 bit machines. So, this technique could lead to speedups of up to 1.5 or 1.8 orders of magnitude.

Because of the overheads associated to unpacking of the results produced by GP programs, the speedup factors achieved in practice are slightly lower than 32 or

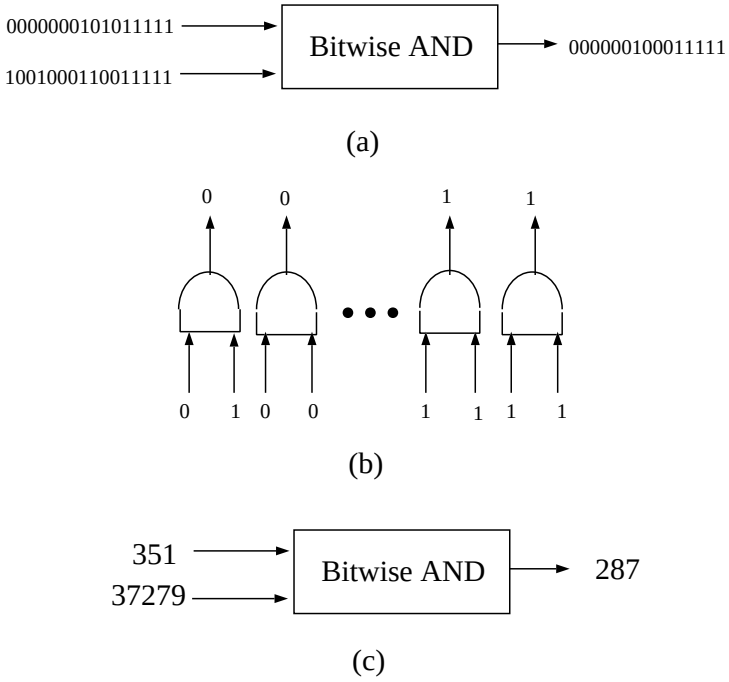


*Figure 4.* (a) bitwise AND between binary numbers, (b) implementation of (a) within the CPU, and (c) the same AND as seen by the user as an operation between integers.

```
enum {X1, X2, X3, X4, X5, NOT, AND, OR, XOR};
unsigned long x1, x2, x3, x4, x5;
char *program;

/* Interpreter */
unsigned long run() {
  switch ( *program++ ) {
    case X1  : return( x1 );
    case X2  : return( x2 );
    case X3  : return( x3 );
    case X4  : return( x4 );
    case X5  : return( x5 );
    case NOT : return( ~run() );        /* Bitwise NOT */
    case AND : return( run() & run() ); /* Bitwise AND */
    case OR  : return( run() | run() ); /* Bitwise OR  */
    case XOR : return( run() ^ run() ); /* Bitwise XOR */
    }
}

/* Bitwise Even-5 parity function */
unsigned long e5parity() { return(~(x1^x2^x3^x4^x5)); }

/* Fitness function */
int even5_fitness_function( char *Prog ) {
  char i;
  int fit = 0;
  unsigned long result, target, matches, filter;
  x1 = 0x0000ffff; /* 00000000000000001111111111111111 */
  x2 = 0x00ff00ff; /* 00000000111111110000000011111111 */
  x3 = 0x0f0f0f0f; /* 00001111000011110000111100001111 */
  x4 = 0x33333333; /* 00110011001100110011001100110011 */
  x5 = 0x55555555; /* 01010101010101010101010101010101 */
  program = Prog;
  result = run();
  target = e5parity();

  /* Count bits where TARGET=RESULT */
  matches = ~(result ^ target);
  filter = 1;
  for( i = 0; i < 32; i ++ ) {
      if(  matches & filter   ) fit ++;
      filter <<= 1;
    }
  return( fit );
}
```

*Figure 5.* C program illustrating the parallel evaluation of fitness cases with SMC-GP.

64 [21]. In tests on an Sun Ultra-10 300MHz workstation using a 32-bit compiler we obtained speedups of 31 times in the evaluation of large programs achieving peaks of around 190 million primitives per second with a C implementation. In tests performed with a DEC Alpha 500 workstation with a 400MHz 64-bit CPU, SMC-GP was able to evaluate on average 550 million primitives per second, which corresponds to 1.3 operations per clock tick! Additional speed-ups can be obtained using CPUs with more bits or exploiting the specialised high-throughput graphics operations (like the MMX extensions on Pentium processors) available on some CPUs.

The parallel evaluation of multiple fitness cases is not the only way SMC-GP can be used. SMC-GP allows the evolution of truly parallel programs for the CPU. In recent research [18, 21] we have shown how this can be done, evolving, for example, parallel adders and multipliers.

## 6. DISTRIBUTED DEMES

The final ingredients we used to solve large parity problems were: a) to use a set of small interacting populations (demes) and b) to distribute the load of the computation across multiple workstations.

Dividing the population into demes helps maintain diversity in the population and has been reported to be, in itself, a way of speeding up the rate of convergence in the even-5 parity problem [2] and other problems [22]. Demes are often organised into rings or toroidal grids. After each generation a small percentage of individuals (the best in each deme) is sent to the neighbouring demes. The migrated individuals are then selectively introduced in the population of each deme, e.g. by replacing the worst individuals. In this approach, if a very good individual is discovered in one deme, spreading that individual to all demes requires several generations.

The deme approach lends itself to efficient parallelisation. Indeed, it is quite easy to run each deme on a separate workstation or processor and then to perform migration via some form of communication. Since communication happens only at the end of each generation, there is no significant communication cost in this approach.

In our work we used a star (client/server) architecture for our demes. In the architecture there is a server deme which sends and receives individuals to and from all the other client demes in the architecture. Each deme includes $P$ individuals. The server deme includes a database with the best $\alpha P$ ($\alpha \in [0.05, 0.10]$) individuals seen so far in all the demes. When new individuals are sent from one client deme to the server deme, the database is updated. When a client deme completes one generation, it sends a message to the server asking for the database, which the server sends as soon as possible. When this is received, the individuals in it are selectively introduced in the deme's population. To maintain diversity in the server database, only individuals with different fitness are stored. If an individual of a given fitness is in the database, and another individual with the same fitness is sent to the server, a random decision is made as to which individual to keep. The server deme also includes a population of $P$ individuals which is run exactly like a client

deme, so that the server alone can perform GP runs when no extra machines are available.

The system is asynchronous. The server deme is able to receive and send the database at any time (even if it is itself running a deme). So, client demes running on slow or heavy loaded machines can still contribute to the success of a run. Also, since in some of our runs we used a large number of workstations in the School and elsewhere, in order to avoid disturbing the activity of other users the client GP processes constantly monitored these activities. As soon as an interactive user was present on a machine, the corresponding GP process would go into sleep mode. In this mode, the process does no processing except checking once per minute whether the machine is free and the GP run should be resumed.

The server is also able to interact with other programs which allow to control and monitor the whole system. For example, the server is able to respond to HTTP requests sending back HTML pages including all the information necessary to check the progress of a run.

The use of a star configuration with a centralised database is an elitist approach with a shared elite. This allows the quick propagation of good individuals across all the demes and in the end makes all the demes converge towards the same area of the search space. Of course this quick propagation may be risky since it reduces diversity. However, since the database included quite diverse solutions thanks to its mechanism to promote diversity, this strategy was extremely beneficial in our experiments.

## 7. RESULTS ON LARGE PARITY PROBLEMS

We have taken up where Koza [7] left off, applying various combinations of the techniques described to the even-$n$-parity problems for $n \geq 12$. Specifically, the values studied were 12, 13, 15, 17, 20 and 22. Koza stopped at $n = 11$ not because GP with ADFs was failing to find a solution, but because the combination of the large population sizes and the increasing number of fitness cases to be evaluated was becoming computationally too expensive. The GP-SUX and GPSPM operators allow us to solve the parity problems much more quickly and with much smaller populations, so we were able to solve the even-12-parity problem on a single machine, running repeated runs with code written in Pop11. For the larger problems, however, it was necessary to utilise parallel populations and sub-machine code GP for a complete run to be executed in a realistic time, and generally we did a single run.

In all the experiments, we used the GP-SUX and GP-SPM operators. Although the mutation rate was varied for different problems, the crossover probability $p_c$ was set to 0.3 throughout. Runs were terminated if a solution had not been found within 500 server generations.

Performance on each problem appears to be sensitive to the initial parameters. In many cases, it was found necessary to vary one or more of the parameters to optimise performance for a specific value of $n$. Those parameters that were varied are given for each $n$ in Table 4. In the table "ramped" represents the ramped half-

*Table 4.* Parameters varied for different values of $n$ on the even-$n$-parity problem.

| $n$ | pop | initial depth/size | init. method | $p_m$ |
|---|---|---|---|---|
| 12 | 100 | 9 | ramped | .01 |
| 13 | 100 | 8 | ramped | .005 |
| 15 | 100 | 8 | ramped | .005 |
| 17 | 500 | 500 | uniform | .01 |
| 20 | 300 | 500 | uniform | .01 |
| 22 | 200 | 1000 | uniform | .005 |

*Table 5.* Number of generations and evaluations required to solve the even-$n$-parity problem for varying $n$. The number of fitness cases per individual is also shown to emphasise how rapidly the actual computational effort increases with $n$.

| $n$ | No. Generations | Individuals evaluated | Fitness cases |
|---|---|---|---|
| 13 | 285 | 28,500 | 8,192 |
| 15 | 542 | 54,200 | 32,768 |
| 17 | 490 | 98,000 | 131,072 |
| 20 | 1188 | 356,400 | 1,048,576 |
| 22 | 2093 | 418,600 | 4,194,304 |

and-half initialisation method, while "uniform" is a method by which the population is initialised using random programs whose length is uniformly distributed between 1 and the size indicated.

We performed 30 independent runs on the even-12-parity problem, and on the basis of the results estimated the effort required to solve it with 99% probability to be 98,800, giving a total of $98,000 \times 2^{12} = 4 \times 10^8$ fitness evaluations. The remaining results, summarised in Table 5, are based on single solutions to each problem. In this table, the number of generations is the total number executed by every machine in the network during the course of the entire run, and the number of individuals processed is therefore simply this value multiplied by the deme's population size. Given the estimated effort for even-12-parity, these values indicate the extremely positive effect of using demes in this class of problems. This was to be expected given that demes help to maintain diversity and therefore make GP-SUX perform more global search for longer. This gives GP more time to locate the good regions in the fitness landscape on which to do local search. These results compare very well with the data reported in the literature for low-order versions (see Table 1).

Solution trees, whilst parsimonious by GP standards, were still too large to display. Our solution to the even-22-parity problem contained 466 nodes. However, we report the distribution of function nodes in the tree in Table 6.

*Table 6.* Distribution of function nodes in a 466 node solution to the even-22-parity function. The 4 bits give the truth table of each arity 2 function, with the rightmost bit refering to the case where both arguments are true.

| Function | Instances | Function | Instances |
|---|---|---|---|
| f0000 | 29 | f1000 | 26 |
| f0001 | 28 | f1001 | 27 |
| f0010 | 18 | f1010 | 37 |
| f0011 | 29 | f1011 | 20 |
| f0100 | 35 | f1100 | 34 |
| f0101 | 28 | f1101 | 29 |
| f0110 | 47 | f1110 | 30 |
| f0111 | 28 | f1111 | 21 |

These data give only a crude indication of which functions are useful in solving the even-22-parity function since a portion of the tree will be introns where mutations can accumulate without affecting fitness. Nevertheless the `XOR` function – `f0110` – is clearly overrepresented implying that it has been widely used in the solution.

## 8. CONCLUSIONS

In this paper we have described three distinct methodologies designed to solve very large parity problems using GP without ADFs. These are: a) smooth operators which are based on a fine grain program representation, b) sub-machine-code GP, which allows the exploitation of the internal parallelism of the CPU, and c) a parallel distributed GP implementation with shared elitism.

We combined these three ingredients into a single recipe which we have used to solve problems that include three to four orders of magnitude more fitness cases than anything tried before. However, this does not describe fully the difficulty of these large parity problems: it is well known that as $n$ increases, the number of fitness evaluations necessary for standard GP to solve even-$n$-parity problems grows much faster than linearly. So, it is arguable that the even-22-parity problem (the largest problem we tried, and solved) is millions of times harder than the largest parity problem solved by standard GP without ADFs.

How did we do that? Firstly, we need to consider that SMC-GP and the use of up to 50 workstations gave us a speed up factor of slightly more than three orders

of magnitude (many of our workstations used 32 bit code). Secondly, the use of demes gave us considerable extra efficiency. However, our experiments show that a very important ingredient for the success of our runs was the use of a function set including all the Boolean functions of arity 2 in conjunction with smooth uniform crossover and smooth point mutation. As shown in Table 3, the presence in the function set of the XOR and EQ functions alone would not provide these performance improvements, without the ability of the smooth operators to move from one point in the search space to any other point with continuity and without obstacles at all scales of resolution.

In the future we intend to study in more detail the contributions each of these ingredients made in isolation. This was simply not possible in this study owing to the size of the problems under investigation. Future work on problems of more limited scale will address this issue.

The work described here immediately offers a number of other directions for future research. The role of global and local search operators needs to be investigated more thoroughly. We have argued that, especially with the full arity 3 function set, GP-UX is unable to perform local search to the same degree as GP-SUX whereas, conversely, standard GP crossover rapidly behaves as a local search operator as the sizes of the parents increase. If this is the case, standard GP should outperform both GP-UX and GP-SUX on problems suited to a purely local search operator.

The effects of including or excluding building blocks or extraneous functions in the function set have been very under-researched so far. Our results suggest that GP is quite capable of selecting the functions it needs from a large unbiased function set. Whilst further work is undoubtedly needed to assess the extent to which this holds for other problems and function sets, it does also suggest that, over successive runs, GP may be able to *learn the bias* it needs to solve the problem most efficiently. We intend to investigate this by constructing a distribution table similar to that shown in Table 6 from the best individual in the previous run and using it to bias population initialisation in the next.

Finally, we are confident that these techniques, either individually or in combination, will prove beneficial when applied to a variety of other problem domains, and we are currently pursuing a number of strands of research to investigate this.

*Acknowledgements* The authors wish to thank Bill Langdon of the Centrum voor Wiskunde en Informatica in Amsterdam who contributed to early versions of this work, and the members of the Evolutionary and Emergent Behaviour Intelligence and Computation (EEBIC) group at Birmingham for their useful comments. Thanks are also due to the three reviewers and to Wolfgang Banzhaf for their helpful feedback on an earlier draft of this paper.

## Notes

1. We could, of course, add other ingredients such as ADFs to the recipe and presumably improve performance further. However our aim here was to show that these three elements are sufficient to solve the problems under investigation.

2. Estimated - standard GP did not actually solve this problem.
3. Koza solved parity problems for $n$ up to and including 11. However, the heavy computation load required prevented him from performing sufficient runs to estimate the effort.
4. Rosca [23] notes that increasing the size of the function set from 4 to 8 increases the fitness diversity of randomly generated trees on the even-5-parity problem, but that this effect is slightly reduced when the size is further increased to 16 functions. Koza [6] examined the effects of extraneous functions on a number of problems including the 6-multiplexer and found performance using set sizes of less than 6 to be superior to that using larger sets, because of greater competition for space from inferior nodes.

## References


1. R. Aler. Immediate Transference of Global Improvements to all Individuals in a Population in Genetic Programming Compared to Automatically Defined Functions for the Even-5 Parity Problem. In W. Banzhaf, R. Poli, M. Schoenauer, and T. C. Fogarty, editors, *Proceedings of the First European Workshop on Genetic Programming*, volume 1391 of *LNCS*, pages 60–70, Paris, 14-15 Apr. 1998. Springer-Verlag.
2. D. Andre and J. R. Koza. Parallel Genetic Programming: A Scalable Implementation Using the Transputer Network Architecture. In P. J. Angeline and K. E. Kinnear, Jr., editors, *Advances in Genetic Programming 2*, chapter 16, pages 317–338. MIT Press, Cambridge, MA, USA, 1996.
3. W. Banzhaf, P. Nordin, R. E. Keller, and F. D. Francone. *Genetic Programming – An Introduction; On the Automatic Evolution of Computer Programs and its Applications.* Morgan Kaufmann, dpunkt.verlag, Jan. 1998.
4. K. Chellapilla. A Preliminary Investigation into Evolving Modular Programs without Subtree Crossover. In J. R. Koza, W. Banzhaf, K. Chellapilla, K. Deb, M. Dorigo, D. B. Fogel, M. H. Garzon, D. E. Goldberg, H. Iba, and R. Riolo, editors, *Genetic Programming 1998: Proceedings of the Third Annual Conference*, pages 23–31, University of Wisconsin, Madison, Wisconsin, USA, 22-25 July 1998. Morgan Kaufmann.
5. C. Gathercole and P. Ross. Tackling the Boolean Even N Parity Problem with Genetic Programming and Limited-Error Fitness. In J. R. Koza, K. Deb, M. Dorigo, D. B. Fogel, M. Garzon, H. Iba, and R. L. Riolo, editors, *Genetic Programming 1997: Proceedings of the Second Annual Conference*, pages 119–127, Stanford University, CA, USA, 13-16 July 1997. Morgan Kaufmann.
6. J. R. Koza. *Genetic Programming: On the Programming of Computers by Natural Selection.* MIT Press, Cambridge, MA, USA, 1992.
7. J. R. Koza. *Genetic Programming II: Automatic Discovery of Reusable Programs.* MIT Press, Cambridge Massachusetts, May 1994.
8. W. B. Langdon. Size Fair and Homologous Tree Genetic Programming Crossovers. In W. Banzhaf, J. Daida, A. E. Eiben, M. H. Garzon, V. Honavar, M. Jakiela, and R. E. Smith, editors, *GECCO-99: Proceedings of the Genetic and Evolutionary Computation Conference*, Orlando, Florida, USA, 13-17 July 1999. Morgan Kaufmann.
9. W. B. Langdon and R. Poli. Why "Building Blocks" Don't Work on Parity Problems. Technical Report CSRP-98-17, University of Birmingham, School of Computer Science, 13 July 1998.
10. W. B. Langdon, T. Soule, R. Poli, and J. A. Foster. The Evolution of Size and Shape. In L. Spector, W. B. Langdon, U.-M. O'Reilly, and P. J. Angeline, editors, *Advances in Genetic Programming 3*, chapter 8, pages 163–190. MIT Press, Cambridge, MA, USA, June 1999.
11. B. McKay, M. J. Willis, and G. W. Barton. Using a Tree Structured Genetic Algorithm to Perform Symbolic Regression. In A. M. S. Zalzala, editor, *First International Conference on Genetic Algorithms in Engineering Systems: Innovations and Applications, GALESIA*, volume 414, pages 487–492, Sheffield, UK, 12-14 Sept. 1995. IEE.
12. P. Nordin. A Compiling Genetic Programming System that Directly Manipulates the Machine Code. In K. E. Kinnear, Jr., editor, *Advances in Genetic Programming*, chapter 14, pages 311–331. MIT Press, 1994.

13. P. Nordin. *Evolutionary Program Induction of Binary Machine Code and its Applications.* PhD thesis, Fachbereich Informatik, Universitaet Dortmund, 1997.
14. P. Nordin. AIMGP: A Formal Description. In J. R. Koza, editor, *Late Breaking Papers at the Genetic Programming 1998 Conference*, University of Wisconsin, Madison, Wisconsin, USA, 22-25 July 1998. Stanford University Bookstore.
15. P. Nordin and W. Banzhaf. Evolving Turing-Complete Programs for a Register Machine with Self-Modifying Code. In L. Eshelman, editor, *Genetic Algorithms: Proceedings of the Sixth International Conference (ICGA95)*, pages 318–325, Pittsburgh, PA, USA, 15-19 July 1995. Morgan Kaufmann.
16. J. Page, R. Poli, and W. B. Langdon. Smooth Uniform Crossover with Smooth Point Mutation in Genetic Programming: A Preliminary Study. In R. Poli, P. Nordin, W. B. Langdon, and T. Fogarty, editors, *Proceedings of the Second European Workshop on Genetic Programming – EuroGP'99*, Goteborg, May 1999. Springer-Verlag.
17. R. Poli. Is Crossover a Local Search Operator? Position Paper at the Workshop on Evolutionary Computation with Variable Size Representation at ICGA-97, 20 July 1997.
18. R. Poli. Sub-Machine-Code GP: New Results and Extensions. In R. Poli, P. Nordin, W. B. Langdon, and T. C. Fogarty, editors, *Genetic Programming, Proceedings of EuroGP'99*, LNCS, Goteborg, Sweeden, 26-27 May 1999. Springer-Verlag.
19. R. Poli and W. B. Langdon. Genetic Programming with One-Point Crossover. In P. K. Chawdhry, R. Roy, and R. K. Pant, editors, *Second On-line World Conference on Soft Computing in Engineering Design and Manufacturing.* Springer-Verlag London, 23-27 June 1997.
20. R. Poli and W. B. Langdon. On the Search Properties of Different Crossover Operators in Genetic Programming. In J. R. Koza, W. Banzhaf, K. Chellapilla, K. Deb, M. Dorigo, D. B. Fogel, M. H. Garzon, D. E. Goldberg, H. Iba, and R. Riolo, editors, *Genetic Programming 1998: Proceedings of the Third Annual Conference*, pages 293–301, University of Wisconsin, Madison, Wisconsin, USA, 22-25 July 1998. Morgan Kaufmann.
21. R. Poli and W. B. Langdon. Sub-Machine-Code Genetic Programming. In L. Spector, W. B. Langdon, U.-M. O'Reilly, and P. J. Angeline, editors, *Advances in Genetic Programming 3*, chapter 13. MIT Press, Cambridge, MA, USA, 1999.
22. W. F. Punch. How Effective Are Multiple Populations in Genetic Programming. In J. R. Koza, W. Banzhaf, K. Chellapilla, K. Deb, M. Dorigo, D. B. Fogel, M. H. Garzon, D. E. Goldberg, H. Iba, and R. Riolo, editors, *Genetic Programming 1998: Proceedings of the Third Annual Conference*, pages 308–313, University of Wisconsin, Madison, Wisconsin, USA, 22-25 July 1998. Morgan Kaufmann.
23. J. P. Rosca. *Hierarchical Learning with Procedural Abstraction Mechanisms.* PhD thesis, University of Rochester, Rochester, NY 14627, Feb. 1997.
24. T. Soule and J. A. Foster. Code Size and Depth Flows in Genetic Programming. In J. R. Koza, K. Deb, M. Dorigo, D. B. Fogel, M. Garzon, H. Iba, and R. L. Riolo, editors, *Genetic Programming 1997: Proceedings of the Second Annual Conference*, pages 313–320, Stanford University, CA, USA, 13-16 July 1997. Morgan Kaufmann.
25. G. Syswerda. Uniform Crossover in Genetic Algorithms. In J. Schaffer, editor, *Proceedings of the Third International Conference on Genetic ALgorithms.* MOrgan Kaufmann, 1989.
26. M. L. Wong and K. S. Leung. Evolving Recursive Functions for the Even-Parity Problem Using Genetic Programming. In P. J. Angeline and K. E. Kinnear, Jr., editors, *Advances in Genetic Programming 2*, chapter 11, pages 221–240. MIT Press, Cambridge, MA, USA, 1996.
27. T. Yu and C. Clark. Recursion, Lambda-Abstractions and Genetic Programming. In R. Poli, W. B. Langdon, M. Schoenauer, T. Fogarty, and W. Banzhaf, editors, *Late Breaking Papers at EuroGP'98: the First European Workshop on Genetic Programming*, pages 26–30, Paris, France, 14-15 Apr. 1998. The University of Birmingham, UK.