Smooth Uniform Crossover, Sub-Machine Code GP and Demes: A Recipe For Solving High-Order Boolean Parity Problems

Riccardo Poli
School of Computer Science
The University of Birmingham
Birmingham, B15 2TT, UK
R.Poli@cs.bham.ac.uk
Phone: +44-121-414-3739

Jonathan Page
School of Computer Science
The University of Birmingham
Birmingham, B15 2TT, UK
J.Page@cs.bham.ac.uk

W. B. Langdon
Centrum voor Wiskunde en Informatica
Kruislaan 413, NL-1098 SJ Amsterdam
W.B.Langdon@cwi.nl
Phone +31 20 592 4093

Abstract

We describe a recipe to solve very large parity problems using GP. The recipe includes: smooth uniform crossover (a crossover operator inspired by our theoretical research), sub-machine-code GP (a technique to speed up fitness evaluation in Boolean classification problems), and interacting demes (sub-populations) running on separate workstations. 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 have been widely used as benchmark tests [1, 3, 4, 5, 6, 16, 17, 19]. 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 is difficult for at least two reasons. Firstly, the function is extremely sensitive to changes in the value of its inputs, since flipping a single 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 [7]. The difficulty of the problem, as measured by the estimated number of fitness evaluations required to find a solution with 99% probability (a quantity known as *effort*), also rises sharply with the number of inputs, n. Koza estimated that the number of evaluations necessary for the canonical form of GP to solve even-n-parity problems increased by about an order of magnitude for each increment of n [6, p. 192].

The increase in difficulty is compounded by the increased memory and processor demands made by higher-order versions of the problem. In part these demands are not peculiar to the parity problems – the bloating of candidate solutions as a GP run progresses has been observed for a broad variety of applications [8]. However, the number of fitness cases associated with a Boolean induction problem increases exponentially with its order and can be very large (e.g. for $n = 12$, trees are evaluated $2^{12} = 4096$ times).

We address these issues in this paper, bringing a number of diverse techniques to bear on high-order ($12 \geq n \geq 22$) versions of the even-n-parity problem. Firstly, we improve the search process using the GP uniform crossover and point mutation operators described in [13] and a *sub-symbolic node representation* [10] which uses an enlarged function set and allows GP to make small, directed movements around the program space. We also apply a technique known as Sub-Machine Code GP [11, 14], which allows the parallel evaluation of 32 or 64 fitness cases per program execution. Finally, we employ a parallel implementation in which GP sub-populations, or demes, are distributed over a number of workstations.

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. We then describe our representation and operators. In Section 4 we describe the sub-machine code technique and, in Section 5, the deme model used. Finally, in Sections 6 and 7 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 [5] and Automatically Defined Functions (ADFs) [6]. In both cases he restricted himself to the four dyadic Boolean functions `AND, OR, NAND, NOR` and what-

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 [6]	6,528	70,176[a]	n/a	n/a	n/a
GP + ADFs [6]	464	1,344	1,440	solved[b]	
EP [3]	2,100	n/a	n/a	n/a	n/a
EP + ADFs [3]	126	121	169	321	586
GP + ADFs [1]	359	627	n/a	n/a	n/a

[a]Estimated - standard GP did not actually solve this problem.

[b]Koza solved parity problems for n up to 12. However, the heavy computation load required prevented him from performing sufficient runs to estimate the effort.

ever 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 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 [3] essentially replicates these studies, 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 [4] 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 [19] 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. Less biased is the approach of Yu [20] 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. Recently, however, we demonstrated an approach that could solve the even-6-parity problem with a population of 50 individuals and with an effort of 17,000 fitness evaluations [10]. Typical solutions were also comparatively parsimonious, the average complexity being around 50 nodes. These results gave us grounds to believe that very high order versions of the problem could be solved. In the next section, we describe the GP algorithm we used to achieve this level of efficiency.

3 OPERATORS AND REPRESENTATION

3.1 UNIFORM CROSSOVER

GP Uniform crossover (GPUX)[13], as the name suggests, is a GP operator inspired by the GA operator of the same name [18]. GA uniform crossover (GAUX) 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

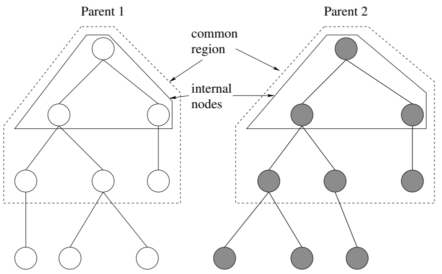

Figure 1: Two parental parse trees prior to GPUX.

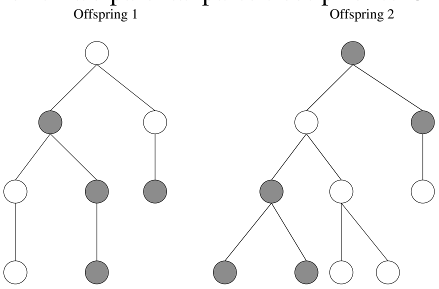

Figure 2: Offspring trees after GPUX.

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. GPUX 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_c. 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 GAUX, the value of p_c is generally set to 0.5, resulting in an exchange of 50% of the nodes. The result of GPUX applied to the trees in Figure 1 is shown in Figure 2.

GPUX, like GAUX, is a homologous operator, that is 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 suboptimal 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 (GPPM)[9] – is also inspired by its GA counterpart (GAPM). GPPM substitutes a single function node with a randomly selected replacement of the same arity. As in GAPM, the average number of mutations performed on an individual is a function of the program size and a user-defined mutation rate. 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 [12].

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 the representation, and hence the behaviour, of a function node can be modified slightly during the course of a GP run. For example, flipping a single bit will alter the behaviour of the node for just one of its possible input combinations.

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 are obviously superfluous (e.g. `always-ON` and `always-OFF`) although what effect they have on performance is poorly understood. Rosca [16] 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 [5] 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.

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 [7] 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 reason for including all 16 dyadic Boolean functions in our set is simplicity – to do otherwise would require constraining the smooth search operators (described in the next section) in some way. In doing so, we note that the EQ and XOR functions are necessarily included and that these will probably enhance performance. On the other hand, the function set is much larger than normal and contains several extraneous functions.

3.3 SMOOTH OPERATORS

We can define a point mutation operator which works in exactly this manner – a single randomly-selected bit is flipped in a single randomly-selected node. In addition, since GPUX is homologous, we can extend it to use a GA crossover operator within the nodes at reproduction (in the experiments reported here we use GAUX). 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 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* (GPSUX).

4 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.

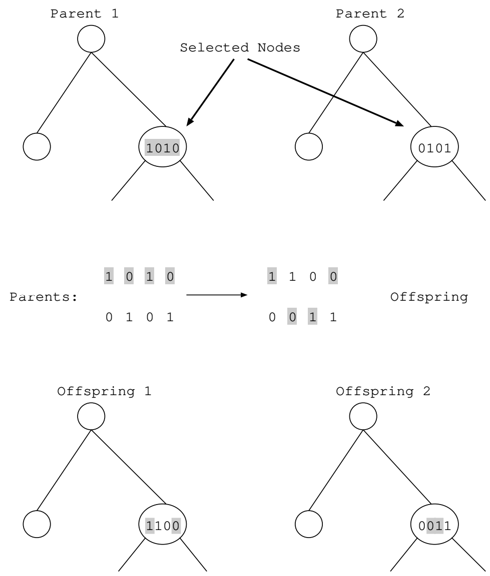

Figure 3: Uniform crossover on the sub-symbolic representation.

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).

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 as follows: (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 the 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 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 and returns the number of entries of the even-5 parity truth table correctly predicted by the program. More implementation details are available in [14]. 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 64 [14]. 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!

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 [11, 14] we have shown how this can be done, evolving, for example, parallel adders and multipliers.

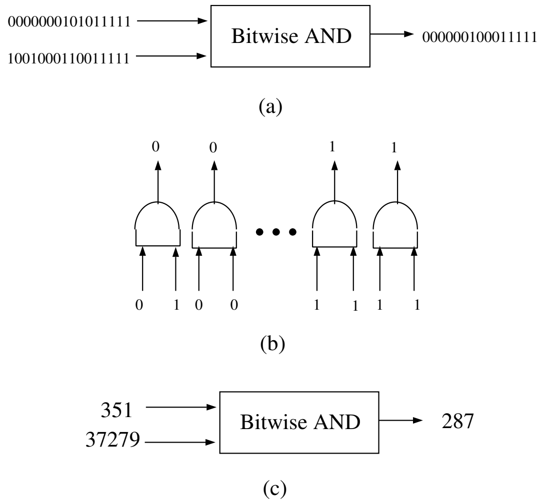

Figure 4: (a) bitwise AND between binary numbers, (b) implementation of (a) within the CPU, and (c) the same AND as seen by the user as an operation between integers.

```
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.

5 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 [15]. 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 αP ($\alpha \in [0.05, 0.10]$) 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.

6 RESULTS

We have taken up where Koza [6] 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 GP-SPM 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 generations.

Performance on each problem appears to be highly 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 6. In the table "ramped" represents the ramped half-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.

Table 2: 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	0.01
13	100	8	ramped	.005
15	100	8	ramped	.005
17	1000	500	uniform	.01
20	300	500	uniform	.01
22	200	1000	uniform	.005

Table 3: Number of generations and evaluations required to solve the even-n-parity problem for varying n.

n	No. Generations	Individuals evaluated
13	285	28,500
15	542	54,200
17	490	98,000
20	1188	356,400
22	2093	418,600

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 fitness evaluations. The remaining results, summarised in Table 3, 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. These results compare very well with the data reported in the literature for low-order versions (see Table 1).

7 CONCLUSIONS

In this paper we have described a recipe to solve very large parity problems using GP without ADFs. The recipe includes three main ingredients: 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.

With this recipe we have solved 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 to 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 probably gave us considerable extra efficiency. However, we believe 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. The presence in the function set of the XOR and EQ functions alone would not provide this 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.

In future research we intend to study whether the recipe for solving the even-n-parity problems is applicable to other Boolean classification problems. We also want to develop a deeper understanding of the mechanisms with which the smooth operators mentioned above build solutions.

Acknowledgements

The authors wish to thank the members of the Evolutionary and Emergent Behaviour Intelligence and Computation (EEBIC) group at Birmingham for their useful comments.

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] 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.

[4] 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.

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

[6] J. R. Koza. *Genetic Programming II: Automatic Discovery of Reusable Programs*. MIT Press, Cambridge Massachusetts, May 1994.

[7] 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.

[8] 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, May 1999. Forthcoming.

[9] 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.

[10] 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. C. Fogarty, editors, *Genetic Programming, Proceedings of EuroGP'99*, LNCS, Goteborg, Sweeden, 26-27 May 1999. Springer-Verlag. Forthcoming.

[11] R. Poli. Sub-machine-code GP: New results and extensions. 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.

[12] 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.

[13] 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. Forthcoming.

[14] 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. Forthcoming.

[15] 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.

[16] J. P. Rosca. *Hierarchical Learning with Procedural Abstraction Mechanisms*. PhD thesis, University of Rochester, Rochester, NY 14627, Feb. 1997.

[17] 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.

[18] G. Syswerda. Uniform crossover in genetic algorithms. In J. Schaffer, editor, *Proceedings of the Third International Conference on Genetic Algorithms*. Morgan Kaufmann, 1989.

[19] 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.

[20] T. Yu and C. Clack. Recursion, lambda abstractions and 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 422–431, University of Wisconsin, Madison, Wisconsin, USA, 22-25 July 1998. Morgan Kaufmann.