Dynamic Page Based Crossover in Linear Genetic Programming

M.I. Heywood, A.N. Zincir-Heywood

Abstract. Page-based Linear Genetic Programming (GP) is proposed in which individuals are described in terms of a number of pages. Pages are expressed in terms of a fixed number of instructions, constant for all individuals in the population. Pairwise crossover results in the swapping of single pages, thus individuals are of a fixed number of instructions. Head-to-head comparison with Tree structured GP and block-based Linear GP indicates that the page-based approach evolves succinct solutions without penalizing generalization ability.

Keywords: Genetic Programming, Homologous Crossover, Linear Structures, Benchmarking.

I. Introduction

A Darwinism perspective on natural selection implies that a set of individuals compete for a finite set of resources, with individuals surviving more frequently when they demonstrate traits that provide a competitive advantage over those without similar traits. This represents a general methodology used as the principle behind a set of search and optimization techniques often referred to as Evolutionary Computation. Examples include, but are not limited to, Genetic Algorithms [1], Evolution Strategies [2] and Genetic Programming [3]. Each share the same basic principles of operation as motivated by Darwin's concept of natural selection. Moreover, variations in features supported often distinguish between different forms of the same technique. Hence, various selection strategies differentiate between different forms of Evolution Strategy and different structures often differentiate between variants of Genetic Programming [4, 3].

In the case of Genetic Programming (GP), an individual takes the form of executable code, hence "running" the program determines an individuals' fitness. In order to apply GP, it is necessary to define the 'instructions' from which programs are composed – often referred to as the Functional Set. The principle constraint on such a set being that it should provide syntactic closure and require one or more arguments [3, 4]. In addition, a Terminal Set is provided consisting of zero argument instructions, typically representing inputs from the environment or constants. Typically two search operators are employed for (1) exploring new solutions (mutation) and (2) exploiting current solutions (crossover) [3, 4].

This work will investigate linearly structured GP, as opposed to the more widely used tree structured individuals [3], and the effect of different forms of crossover operator. A linearly structured GP, or L-GP, implies that instead of representing an individual in terms of a tree, individuals take the form of a 'linear' list of instructions [5-9]. Execution of an individual therefore mimics the process of program execution normally associated with a simple register machine as opposed to traversing a tree structure (leaves representing an input, the root node the output). Each instruction is defined in terms of an opcode and operand, and modifies the contents of internal registers, memory and program counter.

The second component of interest is the crossover operator. Biologically, crossover is not 'blind,' chromosomes exist as distinct pairs, each with a matching *homologous* partner [10]. Thus, only when chromosome sequences are aligned may crossover take place; the entire process being referred to as meiosis [10]. Until recently, however, crossover as applied in GP has been blind. Typically, the stochastic nature of crossover results in individuals whose instruction count continues to increase with generation without a corresponding improvement to performance. This is often referred to as code bloat. Some of this effect has been attributed to an individual attempting to protect instructions actually contributing positively to an individual's fitness, with instructions that make no contribution. Redundant instructions effectively reduce the likelihood that a crossover operation will decrease the fitness of an individual [11].

In order to address the negative effects of crossover in Tree structured GP, modifications such as "size fair" and homologous crossover have been proposed [11]. Nordin *et al*. also proposed a homologous crossover operator for linearly structured GP (L-GP) [12], hereafter referred to as block-based L-GP. In the work proposed here, an individual is described in terms of a number of *pages*, where each page has the *same* number of *instructions* [13, 14]. Crossover is limited to the exchange of *single* pages between two parents, hence, unlike homologous crossover, the location of pages for crossover is unconstrained, but the number of *instructions* in an individual never changes. For the remainder this method is referred to as page-based L-GP.

The purpose of the following study is, firstly, to identify whether the page-based crossover operator, or fixed length format, produces any obvious limits to the performance of the algorithm. In doing so, a comparison is made against results for both Tree-based GP and block-based L-GP on benchmark problems, where no such comparison between linearly and tree structured GP presently exists. In the case of this study, page-based L-GP is not fixed to a specific instruction set, but interpreted in a high level language for the purposes of comparing the properties of the crossover operator. (Motivations from a hardware perspective are discussed in [13].)

In the following text, section II details the page-based crossover operator. Section III evaluates the performance of

Tree-based GP, block-based L-GP and page-based L-GP on a set of benchmark problems. Finally, the results are discussed and future directions indicated in section IV.

II. Linearly Structured GP

Interest in linearly structured GP (L-GP) extends back to the late 1950s when Friedberg conducted various experiments using what would today be considered linearly structured individuals [7]. In 1985, Cramer directly addressed the problem of defining Turing Equivalent languages, capable of maintaining syntactic correctness, following modification by genetic operators [8]. The first working examples of linearly structured GPs, however, had to wait until the mid 1990s. Nordin and Banzhof emphasize the highly efficient implementation of GP using a linear structure [5, 6]. Moreover, the very efficient kernel and memory footprint have enabled the demonstration of mobile applications, in which individuals are evolved on line as opposed to under simulation [6]. Huelsbergen has taken a different emphasis and concentrated instead, on the evolution of program iteration without explicit instruction support for this in the Functional Set (i.e. 'for', 'do-until' and 'while' loop instructions are not provided) [9].

Before defining page based linearly structured GP, the following definitions are necessary. Firstly, 'classical' crossover for linearly structured GP (L-GP) is defined as that in which arbitrary numbers of instructions, unconstrained by the number of bytes, or their location within an individual are swapped to create children. Secondly, homologous crossover for L-GP follows the definition used by Nordin in which, crossover is performed between *aligned* equal length 'blocks' containing a variable number of instructions, but of a fixed equal number of bytes per block [12].

Sub-sections A and B define the page-based crossover and sub-section C summarizes the mutation operators, all of which form the proposed page-based L-GP reviewed in section IV. Sub-section D summarizes the instruction format.

A. Page-Based Crossover Operator

The crossover operator for "page-based" L-GP results in individuals defined in terms of a number of program pages (does not change after initialization) and a page size, as measured in terms of instructions per page (fixed for all members of the population). The crossover operator merely selects which pages are swapped between two parents, where it is only possible to swap single pages. This means that following the initial definition of the population; the length of an individual *never* changes (length measured in terms of the number of pages and instructions per page). The number of pages each individual may contain is selected at initialization using a uniform distribution over the interval [1, max program length]. This is different from classical L-GP as: (1) the concept of pages does not exist; and (2) the number of instructions crossed over in classical L-GP is not constrained to be equal, resulting in changes to the number of instructions per individual.

As indicated by the work of Nordin, however, when GP is implemented on CSIC architectures at the machine code level, instructions are not of uniform length, hence the motivation for a "block-based" approach to crossover in L-GP [12]. Block based crossover swaps equally 'sized' blocks of code, which may contain different numbers of instructions as long as the *total bytes* per block is the *same*. In addition, a homologous crossover operator results if the two blocks happen to be in the same position in each individual. An instruction *block* is therefore defined in terms of an equal number of *bytes*, rather than an equal number of *instructions*. The principle motivation for the 'blocks' concept being to enable efficient crossover in variable length instruction formats as typically seen in CISC architectures [12]. The blocks of such a homologous crossover operator therefore need sufficient space for worst-case instruction bit length combinations with empty words being padded out with NOP instructions. Describing crossover in this manner means that the process of addressing code for transfer between individuals during crossover is now regular (each block always contain the same number of bytes) [12]. This is important when implementing GP at the machine level, but not when using a high-level language implementation, as in the case of the results reported in section III.

B. Dynamic Page-Based Crossover Operator

Given that the page-based approach fixes the number of instructions per page, where this is undoubtedly problem dependent, it would be useful if manipulation of the number of instructions per page was possible without changing the overall number of instructions per individual. To do so an *a priori maximum* number of instructions per page size are specified, where this is the same across all individuals. The selection of different page sizes is then related to the overall fitness of the population. For example a *maximum* page size of 8 also permits page sizes of 4, 2 and 1 whilst retaining page alignment (as measured in instructions not bytes). Now, assuming that it is best to start with small pages, hence encouraging the identification of building blocks of small code sequences, the page-based L-GP begins with a page size equivalent to the smallest divisor of the *maximum* page size – always a single instruction. Let this be the current *working* page size. When the fitness of the population reaches a 'plateau,' the *working* page size is increased to the next divisor, in this case, a page size of two instructions, and the process repeated until the *maximum* page size is reached. A further plateau in the fitness function causes the cycle to restart at the smallest page size. For example, given a *maximum* page size of 8, the following sequence of *working* page size would be expected: $1 \rightarrow 2 \rightarrow 4 \rightarrow 8 \rightarrow 1 \rightarrow 2 \rightarrow$ etc.

An efficient definition for a plateau in the fitness function is now required. For this purpose, a non-overlapping window is used, in which the best-case fitness is accumulated over the length of the window. The result is compared to that of the previous window. If they are the same, then the fitness is assumed to have reached a plateau and the *working* crossover page size is changed. In all the following work, the window size remains fixed at 10 tournaments.

Naturally, the concept of a plateau used in the above definition is a heuristic. That is to say, it can be argued that changing the page size based on such a definition is just as likely to increase search time as reduce it. The empirical observations in section III demonstrate that in practice, the above process is significantly more efficient than retaining a fixed page size.

In summary, a page-based crossover operator has been defined for L-GP. Such a definition avoids the need to estimate additional metrics to ensure minimal code bloat, as in Homologous crossover operators defined for Tree-structured GPs [12], and does not need to combine the classical crossover operator with a Homologous operator as in block-based linear GP [12]. The pay off for this, however, is that individuals are now of fixed as opposed to variable length.

C. Mutation Operators

In the case of this work, two types of mutation operators are employed. The first type of mutation operator is used to manipulate the contents in individual instructions. To do so, an instruction is randomly selected, and then, an X-OR operation performed with a second randomly generated integer to create the new instruction. This is later referred to as an *instruction wide* mutation operator. A second version is also considered in which only a field of the instruction is selected for mutation [6]. This is referred to as *field specific* mutation.

The second type of mutation operator was introduced to enable variation in the *order* of instructions in an individual [13]. In this case, an arbitrary pairwise swap is performed between two instructions in the *same* individual. The motivation here is that the sequence, in which instructions are executed within a program, has a significant effect on the solution. Thus, a program may have the correct composition of instructions but specified in the wrong order.

D. Page-Based Linear GP Instruction Format

A 2-address format is employed in which provision is made for: up to 16 internal registers, up to 16 inputs (Terminal Set), 7 opcodes (Functional Set) – the eighth is retained for a reserved word denoting end of program – and an 8-bit integer field representing constants (0-255). Two mode bits toggle between one of three instruction types: opcode with internal register reference; opcode with reference to input; target register with integer constant. Extension to include further inputs or internal registers merely increases the size of the associated instruction field. The output is taken from the internal register providing best performance on training data. That is to say, the fitness function is estimated across all internal registers, and the single register with smallest error on training data taken as the output for that GP individual. *Thereafter*, on validation and test data sets this represents the output register for that individual [14]. The principle reason for this is that initialization of the population and ensuing application of search operators does not guarantee that all instructions contribute to producing a result in an *a priori* defined register (unlike Tree structured GP in which all instructions contribute to the root node).

III. EVALUATION

The purpose of the following study is to demonstrate the significance of the above modifications and place the results within the context of Tree structured GP (T-GP), as implemented using the lilgp version 1.1 [15], and the block-based L-GP [12], using a free download of Discipulus version 2.0 [16]. The authors are not aware of any such comparative results for linearly structured GP on the discussed benchmark problems; table I. The first problem – two boxes – has found widespread recognition as a benchmark, exercising the ability of GP to sample multiple inputs (six) whilst also being simple to evaluate and nonlinear [3, 17]. The next three problems are all examples of the binary even parity problem – again a widely used benchmark problem [3, 17, 18]. The final set of problems is taken from a set of widely used real world classification problems [19].

In the case of both block-based and page-based L-GP, steady state tournament selection is held between four individuals selected randomly from the population with replacement, and a maximum of 50,000 generations (tournaments) performed. This is equivalent to 50 generations of a population of 4,000 individuals when using a generational selection criterion, as in the work of Koza [3, 18]. Data is collected for 50 different initializations of the population in each experiment. Sub-section A details the nature of the experiments performed and sub-section B presents the results of these experiments.

Over the course of the following experiments, performance is evaluated in terms of: the number of instructions (nodes) in the best-case solution, convergence count, and Koza's metric for Computational Effort [3, 18]. In the latter case, this corresponds to the following expression,

$$E = T \times i \times \frac{\log(1-z)}{\log(1-C(T,i))}$$

where T is the tournament size; i is the generation at which convergence of an individual occurred; z (= 0.99) is the probability of success; and $C(t, i)$ is the cumulative probability of seeing a converging individual in the experiment. By convention, the instance minimizing the above relation over the converging trials is quoted (*opt*). In order to reduce the significance of any one result, average Computational Efficiency (avg) will also be used.

A. Learning Parameters

Parameter selection is generally a thorny subject in learning algorithms as a whole and GP is no different. By way of example, page-based L-GP uses crossover, an instruction specific mutation operator, and a second mutation operator to swap instructions within the same individual. Block-based L-GP uses *two* crossover operators. One is the homologous operator (used in 95% of the crossover operations) and the second provides for the arbitrary interchange of blocks (not aligned and allows swapping between unequal numbers of blocks). Three mutation operators are defined – field specific, instruction specific and block wide [16]. T-GP only requires a single crossover and mutation operator, although there are different probabilities for differentiating between terminal and internal nodes of the tree. All this means that selecting 'equivalent' parameter combinations is very difficult, if not impossible. The approach used here was therefore to fix major parameters such as population size, node (instruction) limits and register counts across an experiment, but experiment with crossover and mutation probabilities to achieve a good fit across all experiments on a particular GP architecture. This resulted in using the crossover and mutation probabilities of table II across all experiments.

Initialization of each architecture also differs. T-GP uses the ramped half-half approach [18] with specific limits to the maximum size of initial individuals being selected as a function of the node limit for that experiment. Page-based L-GP and block-based L-GP share the same general process [6, 13], except that the page-based approach will initialize individuals against the overall maximum instruction limit on account of the fixed length methodology. The block-based approach, on the other hand, begins with much shorter individuals (number of instructions) and evolves up to the maximum instruction limit, as does T-GP. Table III summarizes the respective initialization processes.

In all experiments, a data set is used to describe the problem, where this is the same for all architectures. Experiments themselves are conducted across the aforementioned three problem types – a total of 7 unique problems – for various different population and maximum node (instruction) limits, tables V, VII, IX. Historically, GP is applied with a large population and low level of mutation, with the hypothesis that the code for the correct solution exists in the population and crossover is the principle search operator. In this work, we are interested in a relatively small population and therefore use higher levels of mutation. In addition, several experiments are conducted using different maximum node (instruction) limits. We are therefore asking if solutions can be evolved that are robust to population and maximum instruction limits, where the latter is particularly important in the case of fixed length individuals. Finally, we are also interested in identifying the significance of the different search operators detailed for page-based L-GP, sub-section II.C, where there are four possible variants; table IV.

B. Simulation Study

1) Two Boxes problem: Table V summarizes parameter selection for the volume difference problem. Experiments are conducted using 2, 4 and 8 internal registers, a maximum of 128 instructions and two different population limits (500 and 125). Table VI summarizes performance of the proposed page-based L-GP.

For page-based L-GP, the *dyn* algorithm provides the most robust performance with the highest number of converging cases and most consistent computational effort under all register conditions, Table VI. This is particularly apparent for the experiments using a smaller population size, where cases not using dynamic page sizing either did not converge or produced a very high computational effort.

In comparison to block-based L-GP and T-GP, figure 1, *dyn* page-based L-GP yields the most consistent computational effort and significantly shorter solutions (4-register solutions best for block and page-based L-GP). T-GP was only able to converge when using the larger population of 500; figure 2.

2) Parity Problems: Table VII summarizes parameter selections for the three even parity problems. Experiments are conducted using 8 internal registers, a maximum of 512 instructions, and three different population limits (500, 125 and 75). Given the length of the individuals, a (maximum) page size of 8 instructions is employed in page-based L-GP. Table VIII summarizes performance of the proposed page-based L-GP.

The *dyn* algorithm again provides the most consistent computational effort and percent of converging solutions. Moreover, the next best algorithm is *multi*, indicating that the most significant parameter in this problem is dynamic paging.

Block-based L-GP did not provide a functional set with logical operators, hence the following compares *dyn* page-based L-GP and T-GP alone; figures 3 to 5. Here, T-GP did not converge at all for the 6-parity problem. Computational Effort of T-GP on the 5-parity problem was high, or biased by a single good converging case (c.f. population of 125), whereas the page-based L-GP case was biased towards the smaller population sizes. On the 4-parity problem, this characteristic was emphasized further, with T-GP favoring a larger population, and page-based L-GP a smaller population (this effect possibly being emphasized by the different selection methods; generational-versus-steady state).

Average length of the converging cases, figure 5, emphasizes a general tendency to use longer solutions on the more difficult tasks, with T-GP being more biased by the different sized populations.

3) Classification Problems: As indicated in table I, all three GP architectures are evaluated on three classification benchmarks as an example of operation on real world data sets. In addition, the C5.0 algorithm is used to establish base-line classification accuracy, for the particular partition of training and test data used here. Specifically, 25% of the data is used for test and 75% for training. In the case of page-based L-GP, the *dyn* algorithm is used in all cases.

Table IX summarizes parameter selections for the three classification problems. Experiments are conducted using 4 internal registers, a maximum of 64, 128 and 256 instructions and a population of 125. C5.0 base-line test classification accuracy is summarized in table X. Figures 6 to 8 summarize test set accuracy for the three problems using GP. All GP architectures producing best-case classification in excess of the C5.0 base-line.

The Liver problem, figure 6, represented the most difficult problem for all architectures. Page-based L-GP consistently produces the best peak-case (best) and average classification (avg.) accuracies independent of maximum instruction counts. Neither block-based L-GP nor T-GP consistently out performed each other on this data set. On the C-heart problem, figure 7, a similar pattern is followed with the exception of block-based L-GP at the 64-instruction limit, for which the best-case performance on this data set is produced. T-GP was consistently the worst performing architecture on this problem. The Breast cancer data, figure 8, resulted in all methods returning equally good peak performance. However, a lot of variation is seen in the average classification counts for block-based L-GP and T-GP.

Figure 9, summarizes the average number of instructions employed per solution over each trial. In all but one case, page-based L-GP returns solutions using a lower number of instructions, with no general trend apparent for the block-based L-GP and T-GP cases.

IV. DISCUSSION AND CONCLUSION

In this work, page-based L-GP is defined in terms of individuals that are expressed in a fixed number of pages, where each page consists of an equal number of instructions. Crossover always results in the interchange of single pages between two parents. The implication being that the number of instructions (and pages) per individual remains constant. Comparison against block-based L-GP and T-GP indicates that despite the similarity in the definition of pages and blocks, the solutions, as characterized by computational effort, number of converging individuals and length of evolved code, are distinct. Specifically, page-based L-GP is capable of providing concise solutions and does not appear to be sensitive to the maximum number of instructions. Hence, does not need extensive fine-tuning of this parameter, as might be anticipated in a fixed length individual. The empirical evaluation also indicated that, in the case of page-based L-GP in the 2-register address instruction format investigated, field specific mutation operators do not provide any advantage over instruction specific mutation.

Future work will address support for dynamically changing the number of registers, where this is used as a partial solution to evolving variable length individuals. That is to say, the smaller (greater) the number of registers, the higher (lower) the effective length of an individual, and the more (less) brittle an individual’s code is to incorrect instruction sequences. Finally, the authors are also interested in the use of the page-based concept to introduce program structure into the process of evolution, for example, in terms of loop and conditional constructs.

ACKNOWLEDGEMENTS

The authors gratefully recognize the support of Mahmut Tamersoy of TEBA Computing Group for the provision of computing resources and many interesting discussions.

REFERENCES

[1] J.H. Holland, *Adaptation in Natural and Artificial Systems*. Cambridge, MA: MIT Press, 1998.

[2] I. Rechenberg, “Cybernetic Solution Path of an Experimental Problem,” Royal Aircraft Establishment, Library Translation 1122, 1965.

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

[4] P.J. Angeline, *Advances in Genetic Programming – Volume 2*, Cambridge, MA: MIT Press. Angeline P.J., Kinnear K.E. Jr (ed), 1996, Chapter 1, pp. 1-20.

[5] J.P. Nordin, *Advances in Genetic Programming – Volume 1*, Cambridge, MA: MIT Press. Kinnear K.E. Jr (ed), 1994, Chapter 14, pp. 311-331.

[6] J.P. Nordin, *Evolutionary Program Induction of Binary Machine Code and its Applications*. Munster, Germany: Krehl Verlag, 1999.

[7] R.M. Friedberg, “A Learning Machine: Part I,” *IBM Journal of Research and Development*, 2(1), pp 2-13, 1958.

[8] N.L. Cramer, “A Representation for the Adaptive Generation of Simple Sequential Programs,” *Proc. Int. Conf. on Genetic Algorithms and Their Application*, 1985, pp 183-187.

[9] L. Huelsbergen, “Toward Simulated Evolution of Machine-Language Iteration,” *Proc. Conf. on Genetic Programming*, 1996, pp 315-320.

[10] C. Tudge, The Engineer in the Garden – Genetics: From the idea of heredity to the creation of life. London, UK: Pimlico Press, 1993.

[11] W.B. Langdon, “Size Fair and Homologous Tree Crossovers for Tree Genetic Programming,” *Genetic Programming and Evolvable Machines*, 1(1/2), pp. 95-120, 2000.

[12] J.P. Nordin, W. Banzhaf, F.D. Francone, *Advances in Genetic Programming – Volume 3*, Cambridge, MA: MIT Press. Spector L., Langdon W.B., O’Reilly U.-M., Angeline P.J. (eds), 1999, Chapter 12, pp 275-299.

[13] M.I. Heywood, A.N. Zincir-Heywood, “Register Based Genetic Programming on FPGA based Custom Computing Platforms,” *3rd European Conference on Genetic Programming*. Berlin: Springer-Verlag, 2000, LNCS Volume 1802, pp. 44-59.

[14] M.I. Heywood, A.N. Zincir-Heywood, “Page-Based Linear Genetic Programming,” *IEEE Int. Conf. Systems, Man and Cybernetics*. October 2000, pp 3823-3828.

[15] Zongker D., B. Punch, lil-gp 1.0 User’s Manual, Genetic Algorithms Research and Applications Group. Michigan State University. http://garage.cps.msu.edu/software/lil-gp/lilgp-index.html

[16] AIMLearning™ Technology. Discipulus 2.0. http://www.aimlearning.com/Prod-Discipulus.htm

[17] K. Chellapilla, “Evolving Computer Programs without subtree Crossover,” *IEEE Transactions on Evolutionary Computation*. 1(3), 209-216, 1997.

[18] J.R. Koza, *Genetic Programming: Automatic Discovery of Reusable Programmes*. Cambridge, MA: MIT Press, 1994.

[19] Universal Problem Solvers Inc., Machine Learning Data Sets. http://www.upso.net/tdl_frames.html

TABLE I – BENCHMARK PROBLEMS

Regression Problems				
Problem	Relation	Num. Exemplar	Input range	Terminal set
Two Boxes	$x_0 x_1 x_2 - x_3 x_4 x_5$	10	[1,10]	$\{x_0, x_1, x_2, x_3, x_4, x_5\}$
Binary Problems				
4 Parity	$D_0 \oplus D_1 \oplus \ldots \oplus D_3$	16	{0, 1}	$\{d_0, d_1, d_2, d_3\}$
5 Parity	$D_0 \oplus D_1 \oplus \ldots \oplus D_4$	32	{0, 1}	$\{d_0, d_1, d_2, d_3, d_4\}$
6 Parity	$D_0 \oplus D_1 \oplus \ldots \oplus D_5$	64	{0, 1}	$\{d_0, d_1, d_2, d_3, d_4, d_5\}$
Classification Problems				
Problem	Num. input features	Num. Patterns Train (Test)	Num. Class Instances {0 (1)}	
Liver	6(1)	259(86)	200(145)	
C-heart	13(1)	227(76)	164(139)	
Breast	9 (1)	524 (175)	458 (241)	

TABLE II – SEARCH OPERATOR SELECTION

Architecture	Parameters
Page-based L-GP	P(Xover) 0.9; P(Mutate) 0.5; P(Swap) 0.9
Block-based L-GP	P(Xover) 0.5; P(Mutate) 0.95
T-GP	P(Xover) 0.9; P(Mutate) 0.5

TABLE III – MAX PROGRAM LIMITS AT INITIALIZATION

GP type	Instruction (node) limit			
	64	128	256	512
Page	16 pages 4 instr./pg	32 pages 4 instr./pg	64 pages 4 instr./pg	64 pages 8 instr./pg
Block	32	80	80	N/a
Tree	2-4	2-4	2-5	2-6

TABLE IV – VERSIONS OF THE PAGE-BASED L-GP

Pneumonic	Description
Std	Fixed page size crossover; instruction wide mutation operator.
Bitmut	Fixed page size crossover; field specific mutation operator.
Dyn	Dynamic page size crossover; instruction wide mutation operator.
Multi	Dynamic page size crossover; field specific mutation operator.

TABLE V – PARAMETER SETTING FOR TWO BOXES PROBLEM

Objective	Fit curve to $x_1 x_2 x_3 - x_4 x_5 x_6$
Terminal Set	$x_1, x_2, x_3, x_4, x_5, x_6$
Functional Set	+, -, *, %
Fitness Cases	50 random values selected over interval [0, 1]
Fitness	Sum Square Error
Hits	Number of cases with absolute error < 0.01
Node Limit	128
Pop. Size	500, 125
Termination	Hits of 50 (success) or 200,000 evaluations (fail)
Experiments	50 independent runs

TABLE VI – PAGE-BASED L-GP ON TWO BOXES BENCHMARK PROBLEM

Algorithm	Num. Int. Reg.	% Solutions (50 trials)	Computational Effort (× 1000)	
			opt	Avg
Population 500				
std	2	4	8,188	9,013
	4	8	3,769	6,347
	8	12	3,101	4,071
bitmut	2	None converged		
	4	6	5,971	6,602
	8	14	2,009	2,947
dyn	2	8	6,511	8,139
	4	14	4,202	6,202
	8	46	421	847
multi	2	None converged		
	4	6	5,033	5,778
	8	4	4,091	5,528
Population 125				
std	2	None Converged		
	4	4	17,192	19,594
	8	4	3,306	3,990
bitmut	2	None Converged		
	4			
	8	2	6,017	6,017
dyn	2	6	2,030	3,055
	4	10	1,480	3,988
	8	10	539	1,255
multi	2	None Converged		
	4	2	14,173	14,173
	8	2	3,994	3,994

TABLE VII – PARAMETER SETTING FOR EVEN PARATY PROBLEM

Objective	Find a Boolean function matching that of the 4 (5), {6}-bit even parity problem(s)
Terminal Set	d_0, d_1, d_2, d_3, $\{(d_4), d_5\}$
Functional Set	AND, OR, NAND, NOR
Fitness Cases	All 2^4 (2^5) $\{2^6\}$ combinations of the Boolean arguments
Fitness	Number of matching fitness cases
Hits	As per 'Fitness'

Node Limit	512
Pop. Size	500, 125, 75
Termination	Hits matching the number of Fitness Cases (success) or 200,000 evaluations (fail)
Experiments	50 independent runs

TABLE VIII – PAGE-BASED L-GP ON PARITY BENCHMARK PROBLEMS.

4 bit even parity			
Algorithm	Pop size	% Solutions (50 trials)	Comp. Eff. (opt) ×1000
std	75	58	711
	125	56	1,007
	500	32	2,241
bitmut	75	66	630
	125	54	1,175
	500	54	993
dyn	75	90	372
	125	82	480
	500	72	553
multi	75	74	535
	125	74	447
	500	82	439
5 bit even parity			
Std	75	16	4,625
	125	22	3,604
	500	14	6,011
	75	20	2,578
	125	12	3,584
	500	10	8,031
Dyn	75	30	2,314
	125	22	3,117
	500	22	3,684
multi	75	32	2,004
	125	14	3,929
	500	24	3,239
6 bit even parity			
std	75	0	Non converge
	125	2	17,915
	500	Non converge	
bitmut	75	3	11,560
	125	6	12,854
	500	2	30,032
dyn	75	12	5,896
	125	20	3,760
	500	2	40,447
multi	75	6	11,587
	125	0	Non converge
	500	6	14,418

TABLE IX – PARAMETER SETTING FOR CLASSIFICATION PROBLEMS

Objective	Find a function correctly classifying the data set
Terminal Set	$d_0,\ldots,d_k$ where k is the problem specific set of features (table II). Constants as per table IV.
Functional Set	+, –, *, %, cos, sin, $arg^2 - 1$
Fitness Cases	See table II

Fitness	Number of matching fitness cases
Hits	As per 'Fitness'
Node Limit	64, 128, 256
Pop. Size	125
Wrapper	IF arg < 0.5 THEN class 0; ELSE class 1
Termination	Hits matching the number of Fitness Cases (success) or 200,000 evaluations (fail)
Experiments	50 independent runs

TABLE XI – C5.0 TEST SET CLASSIFICATION ERROR

Problem	Test Set Classification
Liver	65.1%
Breast	95.4%
C-heart	75%

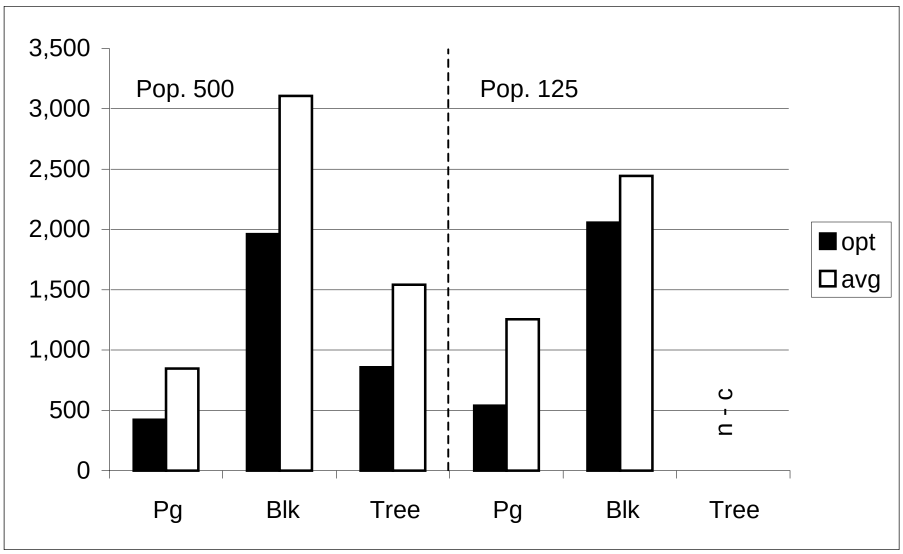

Fig 1. Two Boxes Problem – Computational Effort (×1000).

'Pg' denotes page-based L-GP; 'Blk' denotes block-based L-GP; 'Tree' denotes T-GP; and 'n-c' denotes none converged.

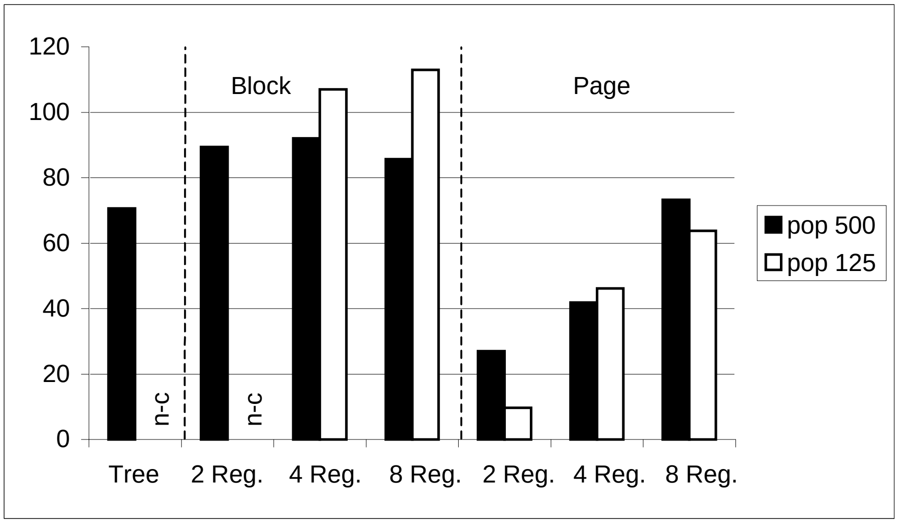

Fig 2. Two Boxes Problem – Average Solution Length.

‘n-c’ denotes none converged. With respect to page and block-based L-GP: ‘2 Reg.’ denotes 2 registers; ‘4 Reg.’ denotes 4 registers; and ‘8 Reg.’ denotes 8 registers.

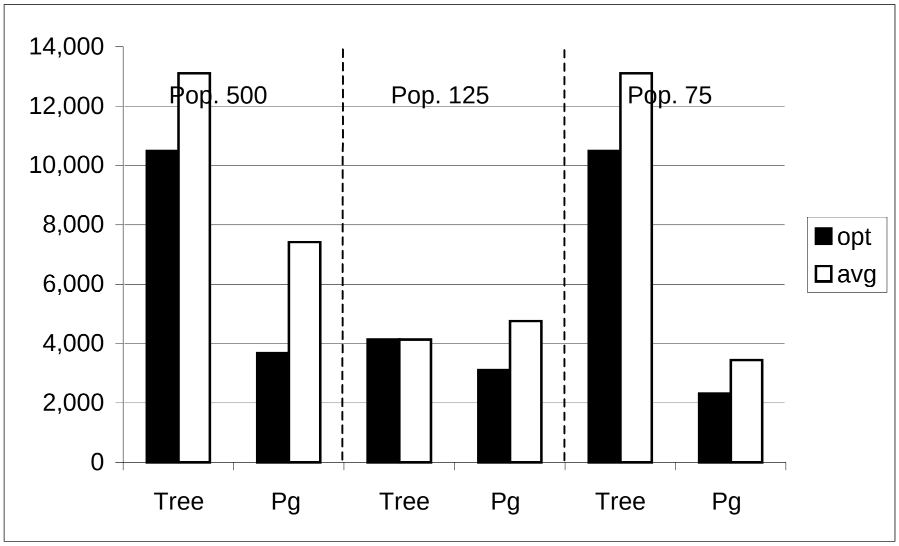

Fig 3. 5-bit Even Parity Problem – Computational Effort (×1000).

‘Tree’ denotes T-GP and ‘Pg’ page-based L-GP. ‘Pop. N’ denotes a population of size ‘N’

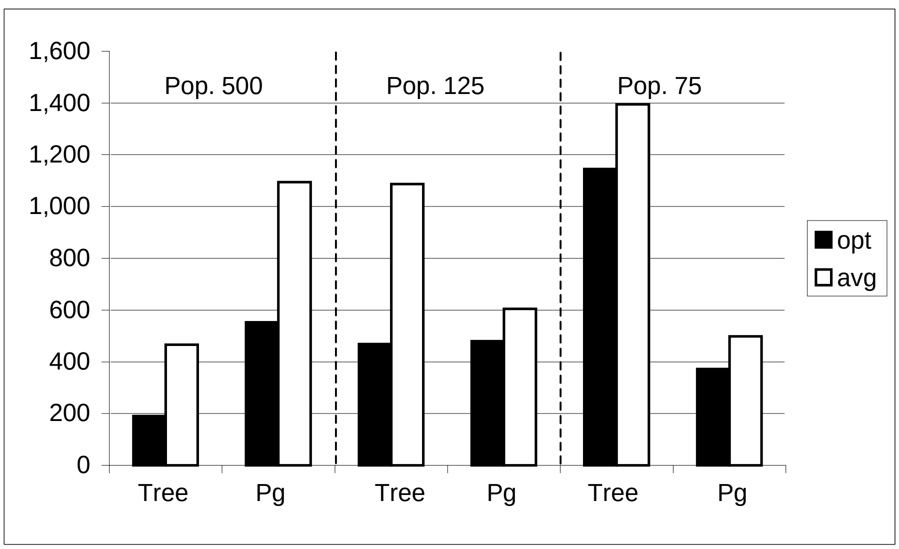

Fig 4. 4-bit Even Parity Problem – Computational Effort (×1000).

‘Tree’ denotes T-GP and ‘Pg’ page-based L-GP. ‘Pop. N’ denotes a population of size ‘N’.

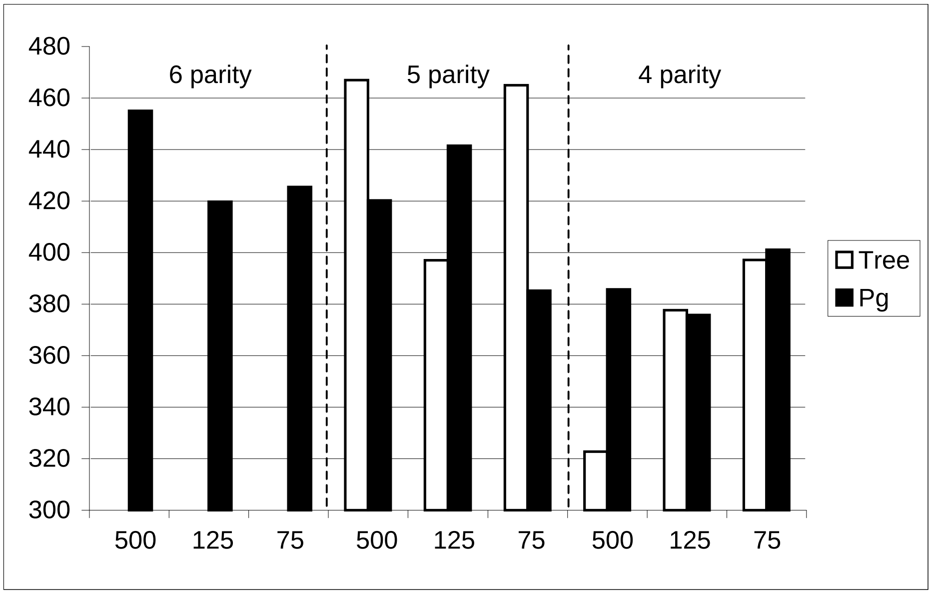

Fig 5. Even Parity Problem – Average Solution Length.

No T-GP cases converge on 6-parity. 500, 125, 75 denote population sizes.

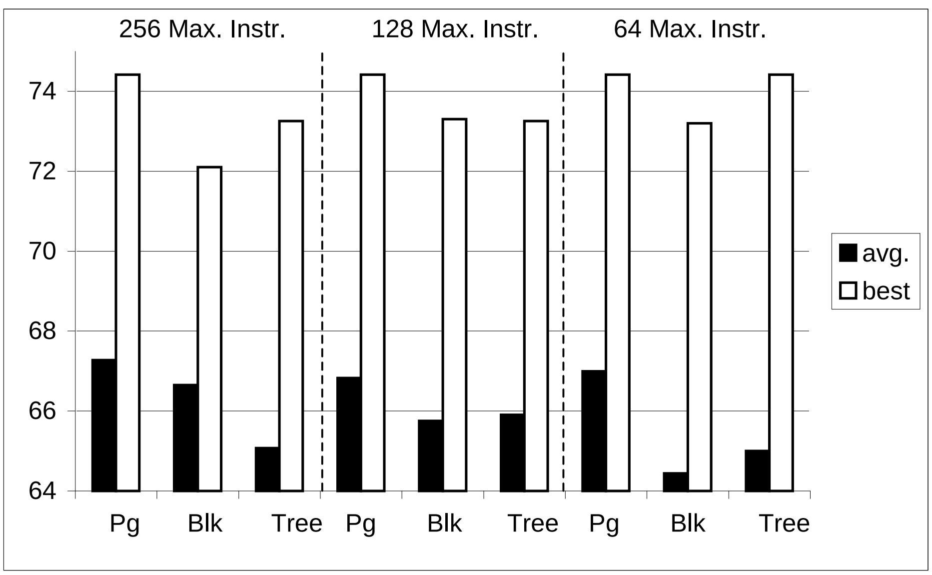

Fig 6. Test Classification Accuracy – Liver Data Set.

'Pg' denotes page-based L-GP; 'Blk' block-based L-GP; and 'Tree' T-GP. 'N Max. Instr.' denotes a Maximum Instruction (node) limit of 'N'.

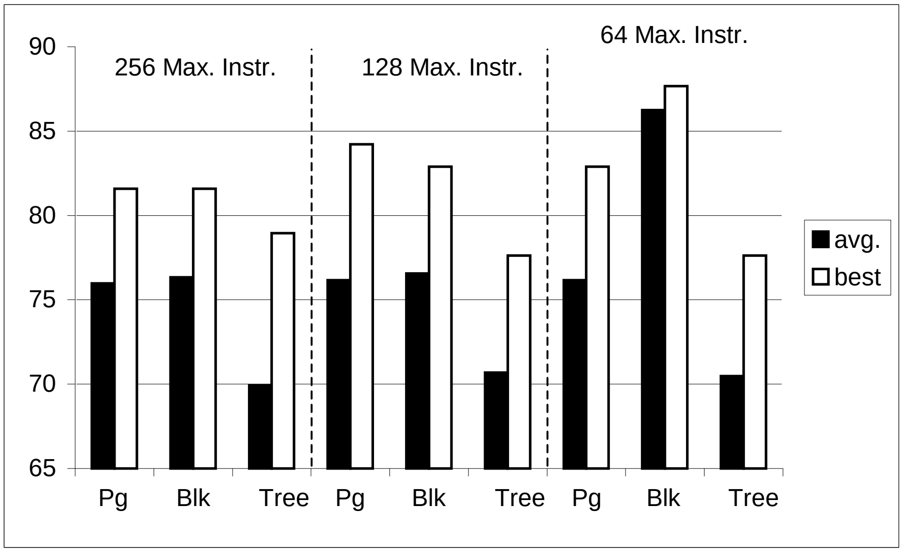

Fig 7. Test Classification Accuracy – C-heart Data Set.

‘Pg’ denotes page-based L-GP; ‘Blk’ block-based L-GP; and ‘Tree’ T-GP. ‘N Max. Instr.’ denotes a Maximum Instruction (node) limit of ‘N’.

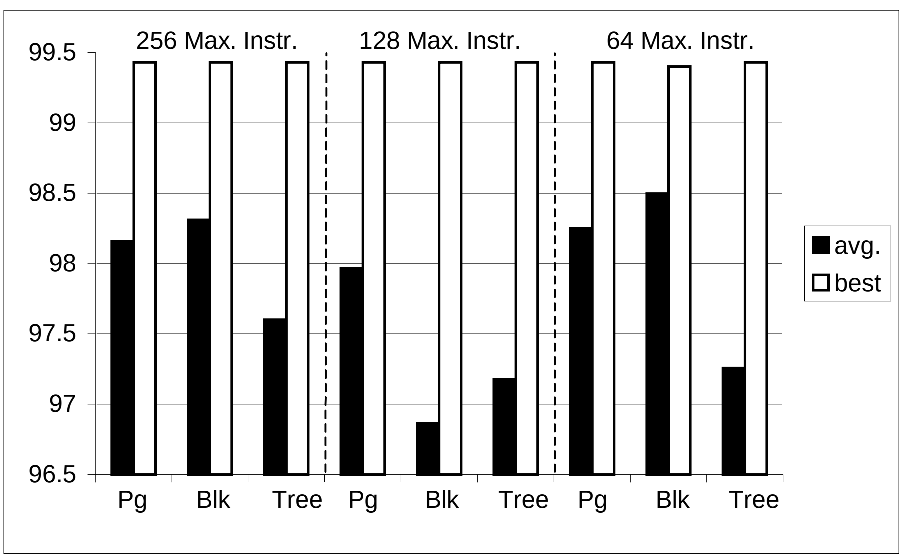

Fig 8. Test Classification Accuracy – Breast Data Set.

‘Pg’ denotes page-based L-GP; ‘Blk’ block-based L-GP; and ‘Tree’ T-GP. ‘N Max. Instr.’ denotes a Maximum Instruction (node) limit of ‘N’.

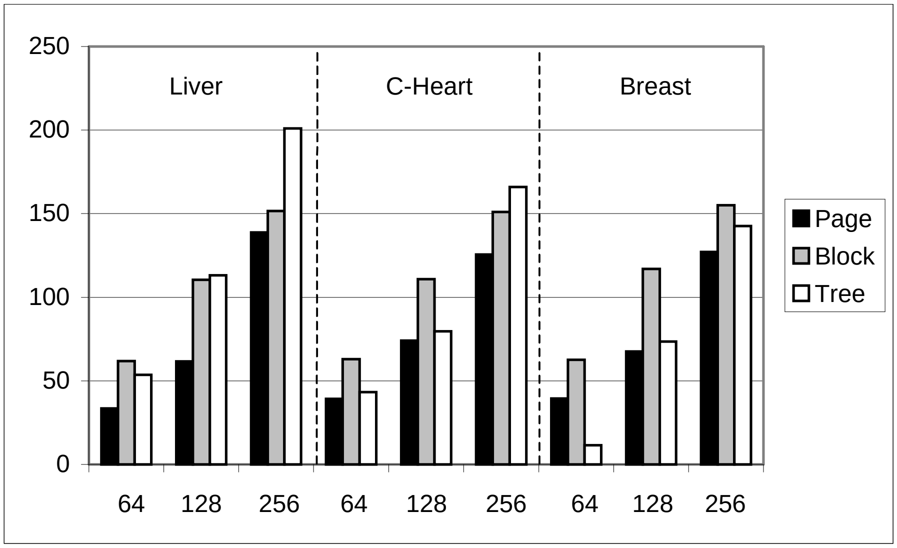

Fig 9. Classification Problems – Average Solution Length.

'Page' denotes page-based L-GP; 'Block' block-based L-GP; and 'Tree' T-GP. 64, 128, 256 denote Maximum Instruction (node) limits.