Page-based Linear Genetic Programming

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

Abstract--Genetic Programming arguably represents the most general form of Evolutionary Computation. However, such generality is not without significant computational overheads. Particularly, the cost of evaluating the fitness of individuals in any form of Evolutionary Computation represents the single most significant computational bottleneck. A less widely acknowledged computational overhead in GP involves the implementation of the crossover operator. To this end a page-based definition of individuals is used to restrict crossover to equal length code fragments. Moreover, by using a register-machine context, the significance of *a priori* internal register – external output definitions is emphasized.

Index Terms--Genetic Programming, Register Machines

I. Introduction

Genetic Programming is a form of beam search in which the beam – set of candidate solutions – are computer programs, the selection criteria or cost function takes the form of a fitness ranking, and the contents of the beam is manipulated by genetically motivated operators (reproduction, crossover and mutation) [1]. This definition provides a very flexible environment for automatic problem solving. However, the structure of representation and the efficiency of the operators used to advance the search i.e. manipulate the contents of the beam (population), have a significant effect on the overall performance. The first structure to reliably demonstrate characteristics suitable for what is now widely referred to as Genetic Programming (GP) was a tree based structure [1, 2]. Specifically, 'program trees' are defined in terms of functional and terminal sets where such a set is defined *a priori* [3]. The function set defines the internal nodes of the tree and possess the property of syntactic closure. The terminal set defines the set of leaf nodes to the program tree and therefore represents the inputs to the program (variables or constants). The mutation operator randomly substitutes internal or leaf nodes with other elements from the functional or terminal set respectively. Crossover typically involves arbitrarily selecting nodes from a pair of trees and swapping the following branch (or leaf).

Since this pioneering work, other structures have been defined. For example the work of Teller and Veloso uses graphs, hence including the case of a program tree as a special case [4]. The interest in this work however lies in structures that map efficiently to computing platforms based on the concept of a register machine. Such a GP structure is commonly referred to as a linear structure, and takes the form of assembly language type instructions [5]. By the term 'register machine' it is implied that the operation of the host computing platform is expressed in terms of operations on sets of registers, where some registers are associated with specialist hardware. The work of Nordin *et al*. represents by far the most extensive work with linearly structured GPs [5-7]. Common to Nordin's work however is the use of standard Von Neumann CPUs as the target-computing platform. In all cases a 3-address, 32-bit instruction format is employed, where this decision is dictated by the architecture of the host CPU. Moreover, the original definitions of the mutation and crossover operators remain. In contrast, the purpose of this work is to provide succinct computing cores of a register machine nature, where the target computing platform takes the form of a custom computing machine (CCM) [8, 9]. To do so, all operations require analysis from a hardware perspective. In particular the classical definition of the crossover operator in Genetic Programming represents a significant memory management overhead, hence introducing hardware complexity. The crossover operator is therefore constrained to that of swapping equal length 'pages' from linearly structured individuals and a second mutation operator, swap, introduced where this interchanges two instructions from the same individual.

The purpose of this paper is to summarize these ideas whilst investigating the significance of other micro-architectural features. In particular the significance of allowing GP to evolve the relationship between outputs and internal registers, and internal registers and constants is assessed. Moreover, the robustness of linearly structured GP is assessed within the context of solving problems requiring the evolution of constants. The significance of different scale factors is therefore also investigated over a set of benchmark symbolic regression problems.

The remainder of the paper is organized as follows. Firstly the concept of page-based linearly structured GP is introduced, Section II. In addition specific micro-architectural design decisions for the internal registers and outputs are discussed. This provides the basis for the simulation study in Section III. Finally, conclusions and recommendations for future work are given in Section IV.

Dalhousie University. Faculty of Computer Science. 6050 University Avenue, Halifax, Nova Scotia. Canada. B3H 1W5.
email: mheywood@cs.dal.ca

II. PAGE-BASED LINEAR GENETIC PROGRAMMING

The principle operations of a GP based register machine are summarised as follows [10],

1. the fetch decode cycle (as in a general purpose computer);
2. evaluation of a suitable cost function, hence fitness of the population;
3. application of suitable crossover and mutation operators;
4. memory management in support of the latter two activities; and
5. generation of random bit sequences (stochastic selection).

It is assumed that the responsibility for forming an initial population of (linearly structured) individuals is performed off-line by the host (most custom computing machine (CCM) platforms take the form of PCI cards [9]). Point 5 implies that random numbers are created off-line[1] and read by the custom computing machine as and when required. Point 2 is application specific, but for the case of the results discussed in section 3, a scalar square error cost function is assumed. However, such an evaluation represents the inner loop of the GP, for which the principle method of accelerated evaluation remains paralleled register machines. Points 1, 3 and 4 are inter-related and therefore the focus of the following discussion. In particular we note that the most expensive operation (from the context of a CCM) is that of a memory access. That is to say, the crossover operator is responsible for creating significant overheads in terms of memory management and memory accesses. Classically the central bottleneck of any computing system is the memory–processor interface. It is therefore desirable to minimise I/O as much as possible, a design philosophy that has resulted in most Von Neumann CPUs dedicating half of the available silicon real estate to memory. Such a solution is not applicable to a CCM due to the low (silicon) efficiency in realising memory using random logic elements typical to (typically Field Programmable Gate Array based) CCM platforms [8, 9].

In the particular case of the crossover operator, as classically applied, the main memory management problem comes from the requirement to swap code fragments of differing lengths between pairs of individuals. This means, for example, that enough memory space needs reserving for each individual up to the maximum program length (irrespective of the individual's actual program length), and entire blocks of code physically shuffled. Moreover, both instances have to be implemented directly in hardware c.f. the spatial implementation of any algorithm on a CCM platform, thus resulting in a lot of 'messy' control circuitry.

[1] By using a reconfigurable computing platform supporting both FPGA and DSP we are able to generate random number sequences using the DSP whilst dedicating the FPGA to evaluation of the GP.

One approach to removing the undesirable effects of crossover is to drop the crossover operator completely. This has indeed been shown to produce very effective GPs in the case of tree based structures [11]. However, this approach requires the definition of a more varied set of mutation operators, some of which effectively replace a terminal with a randomly grown sub-tree, hence another memory management problem. The approach chosen here therefore is to define the initial individuals in terms of the number of program pages and the program page size. Pages are composed of a fixed number of instructions (common to all individuals) and the crossover operator is constrained to defining the pages that are swapped between two parents. Hence we do not allow more than one page to be swapped at a time. This means that, following the initial definition of the population of programs – the number of pages each individual may contain (uniformly randomly selected over the interval [min program length, max program length]) – the program length of an individual remains constant. The memory management now simplifies to reading programs and copying the contents of parents to their children.

In addition to the typical approach to mutation – perform a logical ExOR between the candidate instruction and a random bit sequence – a second mutation operator is introduced. 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.

The overall algorithm is summarized by figure 1. Attention is drawn to the use of steady-state tournament based as opposed to generational population wide selection. That is to say, only a small subset of the total number of individuals compete to have their fitness assessed over the data set. From a hardware perspective, this implies that: (1) there are the same number of register machines as there are individuals in the tournament, providing for parallel evaluation of the individuals; and (2) the fitness ranking may now also be performed directly in hardware, as the number of individuals ranked is decoupled from the size of the overall population.

1. Initialize the population of 'N' individuals.
2. Choose 'M' individuals to participate in a tournament ($M << N$).
3. Evaluate the fitness of individuals participating in the tournament.
4. Rank individuals from tournament in ascending order of fitness.
5. Copy fittest M / 2 individuals over worst (denotes the children).
6. Apply pairwise crossover test.
7. Apply test for standard mutation.
8. Apply test for swap based mutation.

9. Replace individuals in original population corresponding to M / 2 worst case ranked tournament individuals with M / 2 children from step 8.
10. If fittest individual satisfies stop criteria END, else return to step 2.

Fig 1. Algorithm for the Genetic Program

In addition to the above comments, several other micro-architectural decisions regarding the relationship between internal registers and global interfaces are likely to have a significant effect on the algorithm. In particular, consider the relationship between outputs and internal registers (i.e. from where are results taken), and to which internal register(s) are constants loaded. In the case of the following study, we are interested in assessing the significance of two different contexts. In the first case, outputs are *a priori* assigned to internal registers 0 to n, where there are n + 1 internal registers in total. All instructions loading a constant to a register are assigned to register n + 1 [5, 6]. In the second case, no constraints are imposed on the register acting as the target, and outputs are defined by the internal register with the best fitness over the entire data set.

In effect, the first case enforces an *a priori* methodology common to all individuals. Any individuals not conforming to this structure are likely to have a poor fitness measure, hence penalized during selection. This means that individuals first need to learn the *a priori* relationship between internal registers (constants) and the outputs, and then begin to evolve solutions[2]. In the second case, each individual is free to define their own relationships between internal registers and results (constants). The penalty for this freedom, however, is an increase in computational effort during evaluation of fitness, and the possibly of an incompatibility between individuals during crossover (individuals are now have their own contexts, where this may be incompatible). One of the purposes of the following study is to provide an empirical assessment of the significance of these issues.

III. Simulation

The modifications proposed above to the GP search operators, represent significant departures from the normal linear GP. The purpose of the following study is therefore to demonstrate that these modifications do not inhibit the problem solving properties of the GP. To this end a software implementation[3] is used to assess the effects of these re-defined operators using well-known function approximation (symbolic regression) benchmarks of $y = x^4 + x^3 + x^2 + x$, $x \in [-1, 1]$; $y = (x + 1)^3$, $x \in [-1, 0)$; and $y = x^6 - 2x^4 + x^2$, $x \in [-1, 1]$. The first problem has been used widely as a benchmark problem [3, 11]. The cubic problem introduces the need to evolve constants, as well as having multiple solutions [12]. The sextic polynomial has also been widely used as a benchmark problem, in particular by Koza, during the evaluation of Automatic Defined Functions in Tree-based GP [3].

[2] A further alternative would be to try to incorporate any *a priori* definition of register purpose during initialisation c.f. population seeding.

[3] Watcom C++, version 11.0, Windows 95, Pentium III 500Mhz, 64Mb RAM.

Table 1. Formats of 2- and 3- register address instructions.

3-register address instruction format		
Field	Bits/ field	Description
Mode	2	<0 0> two internal register sources; <0 1> internal and external register sources; <1 0> two external register sources; <1 1> 8 bit const. to target reg.
Opcode	3	Only 6 defined <+, -, %, *, NOP, EOP>
Target register	3	Internal register identifier
Source 1 register	3	Register identifier c.f. mode bits
Source 2 register	3	Register identifier c.f. mode bits
2-register address instruction format		
Mode	2	<0 0> internal register source; <0 1> external register source; <1 ×> 8 bit const. to target reg.
Opcode	3	Only 6 defined <+, -, %, *, NOP, EOP>
Target/ source 1 register	3	Internal register identifier
Source 2 register	3	Register identifier c.f. mode bits

Table 2. GP versions

GP id	Description
VarRegXX	Variable constant and output registers allocation.
FixRegXX	Fixed constant and output registers allocation.
XX	Max limit of 8 bit constants (0 – XX)

Table 3. GP Parameter Tableau for all experiments

Terminal Set	x
Functional Set	+, -, ×, %, 8 bit constant (0-255, or 0-5; table 2).
Search Operators	Crossover 90%; Std Mutation 50%; Swap 90%.
Fitness	25 patterns in the interval –1…1
Hits	SSE over all 25 patterns
Selection	Cases below absolute error of 0.01
Pop Size	125
Initial Pop	Uniform random generation of integers with instruction type bias of: 16% constants; 50% input; 33% internal registers. Max populations of 8×4, 16×4 or 32×4.
Termination	30,000 tournaments
Experiments	38 independent runs per parameter setting (population limit, register address mode; and table 2).

The details for the address formats employed here are summarised in Table 1. In each case we design to provide: up to 8 internal registers; up to 7 opcodes (the eighth is retained for a reserved word denoting end of program (EOP)); an eight bit integer constant field; and up to 8 input ports. Table 2 summarizes the labels used throughout the following tests to distinguish different GP configurations.

In terms of specific tests, the significance of the swap based mutation operator has already been established [10]. The purpose of the following study is therefore to: (1) demonstrate the significance of the relationship between

Table 4. Simple Symbolic Regression – Convergence Properties

VarReg255						
	2 address			3 address		
Max. Pages	8	16	32	8	16	32
Av. Tournament	746	2167	4375	3358	3587	3783
Converg. Cases	34	37	28	37	37	35
FixReg255						
Av. Tournament	520	532	1259	5354	3111	4378
Converg. Cases	37	36	37	31	36	34
VarReg5						
Av. Tournament	2692	2818	5173	4052	4185	7779
Converg. Cases	32	34	24	35	36	33
FixReg5						
Av. Tournament	483	755	1571	4531	3572	4831
Converg. Cases	38	37	36	34	34	33

Table 5. Simple Symbolic Regression – Computational Effort.

	2 address (×1000)			3 address (×1000)		
Max. Pages	8	16	32	8	16	32
VarReg255	11.4	20	33.4	4.4	42.7	65.4
FixReg255	6.7	6.7	11.9	58.5	37.2	51.3
VarReg5	26	31.2	84.7	44.6	52.1	65.0
FixReg5	4.8	9.1	13	53.4	39.6	75.5

internal registers and output; and (2) evaluate the significance of different normalizations to the range of constants, where this has already been identified as a significant factor in tree based GP [12]. Moreover, as the page-based crossover operator results in individuals of fixed length, experiments are conducted across *maximum* numbers of pages of 8, 16 and 32 (4 instructions per page in each case) – actual number of pages per individual is selected as a uniform random variate from 1 to max number of pages.

Time constraints dictate that only 25 patterns are used to evolve individuals, however, 250 independent patterns employed to verify the generality of the relation learnt. Table 3 summarizes the GP parameters employed during the study.

A. Simple Symbolic Regression: $y = x^4 + x^3 + x^2 + x$

The simple symbolic regression problem represents a problem in which no constants are necessary. Hence, our interest lies in whether this favours a particular GP algorithm, and whether any favouritism is carried over to problems requiring constants. Results are firstly summarised in terms of the average number of tournaments (converging instances alone) and number of converging instances (38 trials per test); Table 4. It is evident that the configurations with *a priori* fixed relations between internal registers typically converge faster and more frequently than GP without *a priori* definitions. In both cases, the 3 address cases are both slower and provide a lower number of converging instances than the 2 address cases. Moreover, there is little difference between performance of programs using different ranges of constant (0 to 255 verses 0 to 5).

Table 5 summarises minimal computational effort as expressed by the following expression of Koza [3].

Table 6. Cubic Problem – Convergence Properties.

VarReg255						
	2 address (×1000)			3 address (×1000)		
Max. Pages	8	16	32	8	16	32
Av. Tournament	3.7	7.3	4.1	4.5	4.7	7.1
Converg. Cases	32	35	32	37	36	34
FixReg255						
Av. Tournament	6.5	8.3	7.3	10	12	9.3
Converg. Cases	16	28	21	28	24	25
VarReg5						
Av. Tournament	6.8	10.1	12.5	10.4	9.8	10.3
Converg. Cases	16	24	17	31	23	29
FixReg5						
Av. Tournament	7.3	7.5	7.6	10.4	10	11.8
Converg. Cases	17	29	30	28	30	28

Table 7. Cubic Problem – Computational Effort.

	2 address (×1000)			3 address (×1000)		
Max. Pages	8	16	32	8	16	32
VarReg255	50	49	28	58	58	57
FixReg255	237	156	229	224	364	159
VarReg5	276	290	462	246	233	237
FixReg5	256	103	98	222	249	239

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

where T is the tournament size (= 4); i is the generation at which convergence of an individual occurred; z (= 0.99) is the target probability of success; and $C(t, i)$ is the cumulative probability of seeing a converging individual in the experiment. The data reported in table 5 is for i^*, the generation minimizing computational effort.

This re-emphasizes the preference for the predefined register format and short program lengths.

Table 8. Sextic Problem – Convergence Properties.

VarReg255						
	2 address (×1000)			3 address (×1000)		
Max. Pages	8	16	32	8	16	32
Av. Tournament	7.7	8.2	10.6	9.55	8.7	9.2
Converg. Cases	12	12	13	9	24	19
FixReg255						
Av. Tournament	N/a	15.4	11.9	12.6	29.4	21.9
Converg. Cases	0	2	11	2	1	3
VarReg5						
Av. Tournament	4.6	13.4	11.6	12.1	9.1	12.1
Converg. Cases	3	11	14	5	6	6
FixReg5						
Av. Tournament	12.1	9.69	9.1	14.1	17.3	12.1
Converg. Cases	11	21	26	4	11	9

Table 9. Sextic Problem – Computational Effort.

	2 address (×1000)			3 address (×1000)		
Max. Pages	8	16	32	8	16	32
VarReg255	295	54.6	34.5	110	222	197
FixReg255	N/a	6,295	220	383	20,319	4128
VarReg5	737	164	649	54	55	32
FixReg5	648	54.6	32.5	527	1,115	904

B. Cubic Problem: $y = (x + 1)^3$

This problem introduces the requirement to evolve constants and is summarised by tables 6 and 7. A definate partition in performance is now apparent. Two address based instruction always out perform their 3 counterpart. However, within the case of each address type the following observations are made. On 2 address instructions with maximum constant range of 0-255, evolved output relations (VarReg255) are at least 3 times as efficient (as measured by Computational Effort) as the fixed register case (FixReg255). Changing to a constant range of 0-5 reverses this relationship, but not to the same extent. The same pattern appearing when comparing 3-address instructions alone.

C. Sextic Problem: $y = x^6 - 2x^4 + x^2$

The final and most difficult problem considered is the sextic symbolic regression problem first used by Koza [3]. As per the above cubic problem, multiple solutions exist and constants are necessary to solve the problem. Tables 8 and 9 summarize performance. The preference identified above for 2-address operation appears to also hold true in this case. Moreover, Thus 'VarReg255' instances provide the highest number of converging trials, lowest computational effort and locate solutions in the minimum number of tournaments. In addition when using 3-address instructions, the evolved register-output programs (VarRegXX) are always preferred to the *a priori* defined cases (FixRefXX).

One further test is performed. Our interest here is whether the programs are in effect still too long to favour the coding format of 3-address instructions. We therefore repeat the sextic problem for a maximum page size of 6 (all other parameters remain unchanged). These results are summarised in table 10. Performance of the 2-address register instruction format is still significantly better than that of 3-address. Moreover, the number of converging cases has dropped significantly in both cases from that achieved using longer programs, suggesting that the optimal maximum program length for both 2 and 3-address instruction formats was larger than that provided by 6 pages.

Table 10. Sextic Problem – Max 6 Page Programs

2 address				
	VarReg255	FixReg255	VarReg5	FixReg5
Av. Tourna.	14,006	10,586	16,241	11,896
No. Converg.	6	2	2	2
Comp. Effort	1.9×10^6	9.6×10^6	6.45×10^6	4.5×10^6
3 address				
Av. Tourna.	15,036	—	20,659	17,246
No. Converg.	6	None	6	2
Comp. Effort	2.0×10^6	—	3.3×10^6	7.8×10^6

IV. Conclusion

The crossover operator is modified to minimize memory management overheads in linearly structured GP. This results in the description of individuals in terms of the number program pages, where a program page is defined by a fixed number of instructions, constant across all the individuals. The number of pages for each individual, however, is determined randomly at initialization. Simulation on benchmark problems has previously illustrated the appropriateness of the method from the context of different register addressing modes. There we concentrate on assessing the significance of evolved verses *a priori* defined register to output and register to constant relationships. It is found, at least for the single-input, single-output problems considered here, that by leaving these relations open to evolution, significantly better results are achieved.

Future work will investigate extensions to arbitrary length programs, where this has been previously demonstrated using the concept of register based I/O [6, 7]. Attention is drawn to the significance of internal register sets, where the identification of optimal register requirements is undoubtedly problem dependent. Future work will therefore incorporate the definition of register sets into the cost function using the concept of Maximum Description Length. Finally, we are also interested in providing dynamic modification of page size definitions during the evolutionary process. The motivation in this case being to encourage the generation of code fragments from page sizes of 1 or 2 up to some maximum page size limit, say 4 or 8, as the error profile of the individual changes. Finally significant interest also lies in the assessment of the page-based crossover operator in minimizing the effects of introns or code bloat [12].

Acknowledgement

The authors gratefully acknowledge Mahmut Tamersoy, Computer Processing Center of the Teba Company Group, for supplying the computational facilities on which this research was conducted. This work was also supported by TUBITAK research grant 199E023.

References

[1] Koza J.R..: Hierarchical genetic algorithms operating on populations of computer programs. Proc. of the 11th International Joint Conference on Artificial Intelligence. (1989) 768-774.

[2] Cramer N.L.: A Representation for Adaptive Generation of Simple Sequential Programs. Proc. 1st Int. Conf. on Genetic Algorithms and their Applications. (1985) 183-187.

[3] Koza J.R..: Genetic Programming: On the programming of computers by means of natural selection. MIT Press (1992).

[4] Teller A., Veloso M.: PADO: A new learning architecture for object recognition. In Ikeuchi K., Veloso M. (eds): Symbolic Visual Learning. Oxford University Press (1996).

[5] Banzhaf W., Nordin P., Keller R.E., Francone F.D.: Genetic Programming An Introduction: On the automatic evolution of computer programs and its applications. Morgan Kaufmann. ISBN 1-55860-510-X (1998).

[6] Nordin J.P.: Evolutionary Program Induction of Binary Machine Code and its Applications. Krehl Verlag. IBSN 3-931546-07-1 (1997).

[7] Banzhaf W., Nordin P., Francone F.D., Efficient Evolution of Machine Code for CISC Architectures using Instruction Blocks of Homologous Crossover, in Advances in Genetic Programming. Volume 3. Spector L. et al. (eds) MIT Press, pp 275-301, ISBN 0-262-19423-6, 1999.

[8] DeHon A., “Reconfigurable Architectures for General Purpose Computing,” AI Tech Report 1586, MIT AI Laboratory 1996.
[9] http://www.io.com/~guccione/HW_list.html
[10] Heywood M.I., Zincir-Heywood A.N., “Register Based Genetic Programming on FPGA Computing Platforms,” European Conference on Genetic Programming, EuroGP’2000, Lecture Notes in Computer Science. 1802. pp 44-59, 2000.
[11] Chellapilla K.: Evolving Computer Programs without subtree Crossover. IEEE Trans. on Evolutionary Computation. 1(3) (1997) 209-216.
[12] Daida J.M., et al., “Analysis of Single-Node (Building) Blocks in Genetic Programming,” in Advances in Genetic Programming. Vol. 3, Spector L., *et al*. (eds), (1999), pp 217-241.
[13] Langdon W.B., “Size Fair and Homologous Tree Crossovers for Tree based Genetic Programming,” Genetic Programming and Evolvable Machines, 1(1/2), pp 95-120, April 2000.