Using Data Structures within Genetic Programming

W. B. Langdon

Genetic Programming Group,
Dept. of Computer Science,
University College, London,
Gower Street, London WC1E 6BT, UK
Email: W.Langdon@cs.ucl.ac.uk
Tel: +44 (0) 171 380 7214, Fax: +44 (0) 171 387 1397

Abstract

In earlier work we showed that GP can automatically generate simple data types (stacks, queues and lists). The results presented herein show, in some cases, provision of appropriately structured memory can indeed be advantageous to GP in comparison with directly addressable indexed memory.

Three "classic" problems are solved. The first two require the GP to distinguish between sentences that are in a language and those that are not given positive and negative training examples of the language. The two languages are, correctly nested brackets and a Dyck language (correctly nested brackets of different types). The third problem is to evaluate integer Reverse Polish (postfix) expressions.

Comparisons are made between GP attempting to solve these problems when provided with indexed memory or with stack data structures.

Category: genetic programming

1 Introduction

Human programmers have long recognised that program production, maintenance and testing are eased if the software is written so that it be independent of memory access implementation details. This is achieved by using abstract data structures, such as stacks, queues and lists, to provide an interface between programs and memory.

We seek to show that data abstraction can be beneficially used within genetic programming (GP). So far work by Teller [Tel94], Andre [And94] and others shows GP can automatically create programs which explicitly use directly addressable (indexed) memory to solve problems which cannot be solved without memory and we have demonstrated that GP can automatically generate abstract data structures such as stacks, queues [Lan95] and lists [Lan96]. In this paper we show that GP can evolve programs which solve problems using such data structures. In two cases we show better GP performance when using data structures compared to directly addressable memory.

The first two examples (Sections 2 and 3) show GP automatically producing programs which recognize whether a sample of a language is within a specific context free language. The third (Section 4) shows GP can evolve programs which evaluate Reverse Polish (postfix) expressions. These were chosen in the expectation that data structures at least as powerful as stacks would be required to solve them.

2 Balanced Bracket Problem

Other work on GP evolving language recognizers has concentrated upon using GP to evolve tree based specifications for abstract machines, such as finite state machines [DPB94] [IdS95], deterministic push down automata [Zom95] and composite machines (being composed on simple Turing machines) [DP95]. In contrast we wish to investigate the impact of providing data structures v. indexed memory and so our GP executes the GP tree directly.

The balanced bracket problem is to recognise sentences composed of sequences of two symbols, (and) which are correctly nested. E.g. (()) is correctly nested but ()) is not. A limit of ten symbols per sentence was assumed.

2.1 Architecture

Two automatically defined functions (ADFs) [Koz94] are available to assist the main result producing branch (or tree). The first has no arguments and has the same terminal and function sets (except arg1) as the main tree. The second has one argument but cannot contain terminals and functions with side effects. This allows a cache of previous values returned by it to be maintained, thus reducing run time.

Table 1 shows the parameters used and the terminals and functions provided, n.b. they include indexed memory but not stacks. Parameters not shown are as [Koz94, page 655]. Table 6 gives the actions of most terminals and functions.

Table 1: Balanced Bracket Problem

Objective:	Find a program that classifies sequences of ((represented by 1) and ((-1) as being correctly nested or not.
Primitives: (any tree)	ADD, SUB, PROG2, IFLTE, Ifeq, 0, 1, -1, max, forwhile, i0
(rpb, adf1)	adf2, aux1, read, write, swap, Set_Aux1
(rpb, adf2)	arg1
(rpb only)	adf1
Fitness cases:	175 fixed test cases.
Fitness:	Number of test cases passed (scalar).
Selection:	Tournament group size of 4 used for both parent selection and selecting programs to be removed from the population. Steady state population (elitist).
Wrapper:	Zero represents False (i.e. not in language) otherwise True.
Parameters:	Pop = 10,000, G = 50, 3×3 demes, no CPU penalty, no aborts.
Max prog size	$4 \times 50 = 200$, In initial pop each tree limited to 50.
Success predicate:	A program passes all tests.

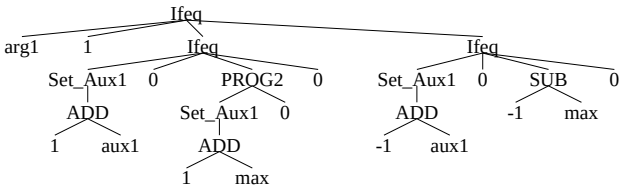
2.2 Fitness Function

The fitness of each trial program was evaluated on a fixed set of 175 example sentences (see Table page 3) containing both correctly nested (positive) and incorrectly nested brackets (negative). The program is called once for each symbol in the sentence. Thus each program is run 1403 times (674 for (and 729 with an argument of)). All stored data within the program initialised to zero at the start of each test sentence. The value

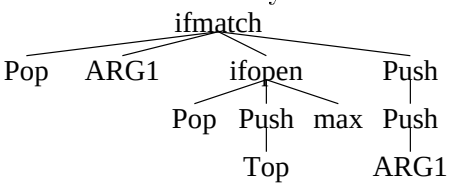
Nested Bracket Test Cases

Length	Positive		Negative	
1			all	2
2	all	1	all	3
3			all	8
4	all	2	all	14
5			random	4
6	all	5	random	5
7			random	5
8	all	14	random	14
9			random	14
10	all	42	random	42
Totals		64		111

Solution to Brack Problem (Simplified)



Fast Solution to Dyke Problem



returned by the program on the last symbol of the sentence gives its verdict as to whether the sequence is correctly nested.

2.3 Results

In the first run a general solution was produced by generation 19.

In contrast to earlier work [Zom95] where GP was obliged to evolved push down automata the evolved solution is effectively a register machine (i.e. less powerful that a push down automata) (Top figure on page 3 gives a simplification of the first solution found). The GP solution only uses a single integer memory cell (aux1), in which it counts the depth of nesting. At the end of a legal sentence this count must be zero, further should the brackets be unbalanced before the end is reached, this is recognised and aux1 is also used as a flag indicating this. The GP solution not only passes all the fitness tests and is a general solution to the problem but (given suitable redefinition of max) is a solution for sequences of any length.

Given the readily found general solution did not exhibit stack like behavior it was decided not to repeat this problem with a GP that had stack primitives.

3 Dyck Problem

Once again the problem is to recognise which sentences are correctly bracketed, however there are now four types of bracket pairs, `(`, `)`, `[`, `]`, {, }, ‘, ’. E.g. {}`[]` is correctly bracketed but `[}` is not. As with the nested brackets problem, a limit of ten symbols per sentence was assumed.

Two sets of experiments have been conducted, the first provided the GP with primitives which implement a stack for it and the second provided primitives like those from which it has been shown GP can evolve stack data structures [Lan95].

3.1 Architecture

In the first experiments (stack given) no ADFs were used, whilst in the second there are three ADFs, having 0, 1 and 0 arguments. It was hoped that these could evolve to

operate like pop, push and top. Each could be called from the main tree, additionally the third (top?) could be called by the first.

The terminals, functions and control parameters are as Section 2 except where given in Table 2.

Table 2: Dyke Language Problem

Objective:	Find a program that classifies sequences of four types of bracket (((represented as 5),) (71), [(13),] (103), { (31), } (137), ‘ (43), ’ (167)) as being correctly nested or not.		
Primitives:	Common	Stack Given	Index Memory
All trees:	ADD, SUB, PROG2, IFLTE, Ifeq, 0, 1, max, aux1	Makenull, Empty, Top, Pop, Push	read, write, inc_aux1, dec_aux1
rpb: as all plus	ifopen, ifmatch, ARG1, Set_Aux1		adf1, adf2, adf3
adf1: as all plus			adf3
adf2: as all plus			arg1, arg2
Fitness cases:	286 fixed test cases.		
Fitness:	Number of correct answers returned.		
Selection:	Tournament size 4 (After 1^{st} solution CPU penalty used giving a two dimensional fitness value, fitness niching used with a sample of up to 81 (9×9) nearest neighbors.)		
Wrapper:	Zero represents True (i.e. in language) and all other values False.		
Parameters:	Pop = 10,000, G = 50, Pareto, 3×3 demes, CPU penalty only after 1^{st} solution found, Abort on first error in sentence.		
Max prog size	Initial tree limit 50	50	$4 \times 50 = 200$
Success predicate:	A program passes all tests.		

3.2 Fitness Function

The fitness of every trial program is determined by presenting it with a series of symbols from test sentences and counting how many times it correctly classifies each as to whether it is the last of a correctly balanced sequence. This problem is more complex than that in Section 2 and so the test cases are longer. To contain the increase in runtime, forwhile loops where not used and the number of times each program must be called was reduced by:

- Only using the first half of the test cases (i.e. tests up to length six) unless a program passes them all, where upon test sentences of length seven and eight where also used,
- Using the value returned for each symbol, rather than just the last in the sentence, and ignoring the rest of the sentence after the first error. As some shorter sentences are identical to the start of longer ones they need not be tested explicitly and so could be removed from the test cases.

Negative cases where, for each bracket type, the number of opening and closing symbols are equal where deliberately included (referred to as “balanced” in Table 3) since

Table 3: Dyke Language Test Cases

Length	Positive	Negative		After Removing Duplicates			
		Balanced	Random	Positive	Bal	Rand	Score
1		all 8					0
2	all 4	all 60				9	18
3			16			10	30
4	all 32	all 24	16	27	16		172
5			16			16	80
6	random 32	random 32	32	32	32	32	576
7			16			16	112
8	random 32	random 32	32	32	32	32	768
			Totals	91	112	83	1756

such tests cannot be passed by counting mechanisms, such as those that evolved in Section 2.

Initialy brackets where coded as $\pm 1, \pm 2, \pm 3, \pm 4$ but general solutions proved difficult to find. Instead partial solutions based upon summing up symbol values dominated. Since the purpose of the experiment was to investigate learning correct nesting of symbols rather than learning which symbols match each other the problem was simplified by providing the GP with two new primitives (ifmatch and ifopen) which say which symbols match each other. Further the symbols where recoded to prime values with no simple relationships between them.

3.3 Results

In three runs given the stack primitives general solutions were evolved by generation seven to 23 (in three identical runs but using simple non-demic populations, two runs produced solutions in generations 30 and 39). The lower figure on page 3 shows one of the solutions after 50 generations. As in Section 2 the solutions are not only general solutions to the given problem, but given a deep enough stack would work with any sentences of any length.

In contrast none of 15 runs using the indexed memory primitives passed all the tests. Some of the more promising runs where extended beyond 50 generations up to 140 generations without finding a solution. The best still failed 3 tests (on sequences of up to six symbols). It showed some stack like behaviour which enables it to pass 13 of the tests of length seven and eight but also showed signs of having learnt the test cases.

A program which always returns zero (i.e. True) has a fitness of zero, in contrast a program which never returns zero has a fitness of 714. In both experiments the GP population quickly learns not to return zero, but when using indexed memory it appears to be more difficult than when given a stack to escape this local optima and learn to return zero at some points.

4 Reverse Polish Calculator

In this section the GP evolves a four function ($+$, $-$, / and $\times$) calculator, i.e. evaluates integer arithmetic expression. The problem is simplified by presenting the expression

in Reverse Polish Notation (postfix), which avoids consideration of operator precedence, and by avoiding expressions which include divide by zero. No limit on the length of expressions was assumed, however the expressions tested were between three and fifteen symbols long (see Table 5).

Two sets of experiments have been conducted, the first provided the GP with primitives which implement a stack for it and the second provided primitives like those from which it has been shown GP can evolve stack data structures [Lan95].

4.1 Architecture

Each individual within the population consists of five separate trees (num, plus, minus, times and div) plus zero or two ADFs. As in sections 2 and 3 each test sentence is presented a symbol at a time to the GP, however in this case the appropriate tree is selected. E.g. if the symbol is an integer, then the num tree is executed with the integer as its argument. Each tree returns a value as the current value of the expression (num's answer is ignored).

In the first experiments (stack given) no ADFs were used, whilst in the second there are two ADFs, having 0 and 1 arguments respectively. It was hoped that these could evolve to operate like pop, push. Each could be called from the five main trees.

The terminals, functions and control parameters are as Section 3 except where given in Table 4.

Table 4: RPN Expression Evaluation Problem

Objective:	Find a program that evaluates RPN arithmetic expressions.		
Primitives:	Common	Stack Given	Index Memory
operation trees:	ADD, SUB, MUL, DIV, PROG2, 0, 1, aux1, Set_Aux1	Top, Pop, Push	inc_aux1, dec_aux1, read, write, adf1, adf2
num: as ops plus	arg1		
adf2: as ops plus			arg1
Fitness cases:	127 fixed test cases.		
Fitness:	Number of correct answers returned.		
Selection:	Pareto tournament size 4, CPU penalty (initial threshold 50 per operation), fitness niching used with a sample of up to 81 other members of the population.		
Wrapper:	Value on num ignored. No Wrapper on other trees.		
Parameters:	Pop = 10,000, G = 50, Pareto, No demes, No threshold on CPU penalty only after 1^{st} solution found, Abort on first wrong answer given in expression.		
Max prog size	Initial tree limit 50	$5 \times 50 = 250$	$7 \times 50 = 350$
Success predicate:	A program passes all tests.		

4.2 Fitness Function

The fixed test cases are a combination of random and pre-defined cases. The random ones where created to have a high chance of testing minus and divide and to be deeply nested.

RPN Test Cases

Operation	No.	Max Score
num	550	163
plus	67	58
minus	103	85
times	85	64
divide	156	127
	420	
Totals	970	497

Depth of Stack in RPN Test Cases

depth	1	2	3	4	5	6	Total
number	387	390	149	31	12	1	970

Fast Solution to RPN Problem

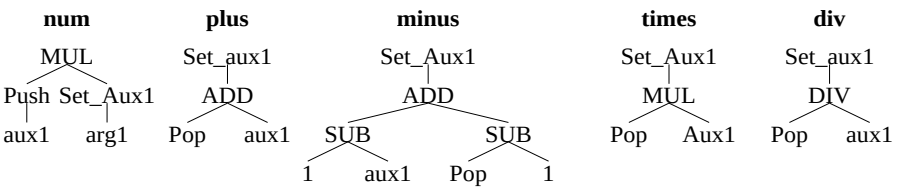


Table 5: Length of RPN Expressions

length	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	Total
number			10	3	55	27	44	2	36	1	5		8		3	194

Integer data values where initialy chosen at random uniformly between $-99 \ldots +99$. However a number of changes were made to arguments of divide:

- divide by zero was not included in the test cases.
- With the same distribution of data values for both arguments of (integer) divide there is a high chance of the result being zero. Such data values where manually changed so that only a few divisions yield values in the range $-3 \ldots +3$.
- To avoid problems with overflow, randomly generated expressions did not allow: arguments to addition or subtraction outside the range $-10^8 \ldots +10^8$ or arguments to multiplication or division outside the range $-65535 \ldots +65535$.
- Data values set by hand where chosen so neither: the product of two arguments of divide nor the square of the second argument exceeded MAXINT ($2.147\ 10^9$).

One case where the stack still contained unevaluated data at the end of the expression was included, in all others the final stack just contains the overall value of the expression.

As before it was necessary to constrain run time. This was done by checking answers during the evaluation of each expression and aborting evaluation following the first error detected; removing test cases which essentially duplicated others and applying a CPU penalty on programs which evaluated more than (on average) 50 primitives per symbol presented to them.

4.3 Results

In eleven runs using stack primitives, six produced solutions which passed all the tests, these where found between generations 11 and 23. In four cases the first programs to pass all the tests were also general solutions to the problem. In the other two the first solutions failed special cases such as $1 - 1$ and $x/y = 0$, however in both runs general solutions were evolved before 50 generations.

Under the action of the increased CPU penalty, solutions which took about one third of the CPU time of the first solution found where evolved. (The figure on page 7 shows one of these).

In 59 runs with stack primitives replaced by indexed memory (see right hand side of Table 4) no program passed all the tests. The highest score 148 (of 194) was achieved by a program which used the first adf to implement DIVR and the second to approximate both push and pop on a three level stack. Other unsuccessful trials included adding a third ADF (with two arguments) in the hope that this might evolve the DIVR functionality (best 102 in 33 runs, 16 running out of time before 50 gens) and supplying SUBR and DIVR functions (in place of SUB and DIV) where the best score was 116, in 38 runs.

The non-commutative functions ($-$ and /) are more difficult because the arguments on the stack are in the opposite order to that used by the SUB and DIV functions. (The problem can be readily solved, when given the stack primitives, by replacing SUB and DIV by SUBR and DIVR which process their arguments in the opposite order, i.e. the order on the stack.) The div tree has to use some storage (i.e. aux1) to reverse the order of the stack arguments (sub can simply negate the result of performing the operation in the wrong order to get the right answer). The need to use aux1 makes the problem deceptive, in that many programs can obtain high scores using aux1 as the top of the stack and only fail tests which require deeper use of the stack.

Some of the difficulty the GP with indexed memory found may have been due to trying to use aux1 both as stack pointer (for which inc_aux1 and dec_aux1 make it suitable) and as the top of stack (as evolved in many cases where the stack primitives where given).

5 Conclusions

These experiments have shown GP can solve two new problems (inducing a Dyke language (Section 3) and evaluating reverse Polish expressions (RPN), Section 4) and a GP with memory can solve the nested bracket problem (Section 2) without requiring an intermediate step generating an abstract machine.

All three examples were solved by GP using the appropriate data structure for the problem. The two more complex examples (Dyke and RPN) proved to be more difficult (no solution has been found to date) for GP when provided with indexed memory than when provided with a stack, despite indexed memory being more powerful.

It has often been argued, e.g. [Kin94, page 12], that functional primitives used with GP should be as powerful as possible, in these examples we have shown appropriate data structures are advantageous, that is GP can benefit from data abstraction.

The failure of GP to solve the problems when provided with the more general (i.e. more powerful) directly addressable memory data structure, is worrying as it shows (at present) data structures should be chosen with care and it may not be sufficient to simply over provide, with more powerful structures than are needed. It was hoped that GP could evolve appropriate data structures as it solved the problem but this has yet to be demonstrated.

A Pareto CPU penalty term to fitness has again been shown to curb growth in program size and CPU time.

Acknowledgments

W. B. Langdon is funded by the EPSRC and the National Grid Company plc. I would like to thank my supervisors (M. Levene and P. C. Treleaven), and Andy Singleton for GP-QUICK.

References

[And94] David Andre. Evolution of mapmaking ability: Strategies for the evolution of learning, planning, and memory using genetic programming. In *Proceedings of the 1994 IEEE World Congress on Computational Intelligence*, volume 1, pages 250–255. IEEE Press, June 1994.

[DP95] Bertrand Daniel Dunay and Frederic E. Petry. Solving complex problems with genetic algorithms. In L. Eshelman, editor, *Genetic Algorithms: Proceedings of the Sixth International Conference (ICGA95)*, pages 264–270, San Francisco, CA, USA, July 1995. Morgan Kaufmann.

[DPB94] B. D. Dunay, F. E. Petry, and W. P Buckles. Regular language induction with genetic programming. In *Proceedings of the 1994 IEEE World Congress on Computational Intelligence*, pages 396–400. IEEE Press, 1994.

[IdS95] Hitoshi Iba, Hugo de Garis, and Taisuke Sato. Temporal data processing using genetic programming. In L. Eshelman, editor, *Genetic Algorithms: Proceedings of the Sixth International Conference (ICGA95)*, pages 279–286, San Francisco, CA, USA, July 1995. Morgan Kaufmann.

[Kin94] Kenneth E. Kinnear, Jr. A perspective on the work in this book. In Kenneth E. Kinnear, Jr., editor, *Advances in Genetic Programming*, chapter 1, pages 3–19. MIT Press, 1994.

[Koz94] John R. Koza. *Genetic Programming II: Automatic Discovery of Reusable Programs*. MIT Press, Cambridge Massachusetts, May 1994.

[Lan95] W. B. Langdon. Evolving data structures using genetic programming. In L. Eshelman, editor, *Genetic Algorithms: Proceedings of the Sixth International Conference (ICGA95)*, pages 295–302, San Francisco, CA, USA, July 1995. Morgan Kaufmann.

[Lan96] William B. Langdon. Data structures and genetic programming. In P. Angeline and K. E. Kinnear, Jr., editors, *Advances in Genetic Programming 2*, chapter 20. MIT Press, Cambridge, MA, USA, 1996.

[Tel94] Astro Teller. The evolution of mental models. In Kenneth E. Kinnear, Jr., editor, *Advances in Genetic Programming*, chapter 9, pages 199–219. MIT Press, 1994.

[Zom95] Afra Zomorodian. Context-free language induction by evolution of deterministic push-down automata using genetic programming. In E. S. Siegel and J. R. Koza, editors, *Working Notes for the AAAI Symposium on Genetic Programming*, pages 127–133, 445 Burgess Drive, Menlo Park, CA 94025, USA, 10–12 November 1995. AAAI.

Table 6: Actions Performed by Terminals and Functions

Primitive	Purpose
`DIV`(x,y)	**if** $y \neq 0$ **return** x/y **else return** 1
`SUBR`(x,y) `DIVR`(x,y)	As `SUB` and `DIV` except yield $y-x$ and y/x
`max`	constant 10 ($\geq$ max input size).
`PROG2`(t,u)	evaluate t; **return** u
`ARG1, arg1, arg2`	arguments of current operation or ADF
`aux1`	an auxiliary variable (i.e. in addition to indexed memory).
`Set_Aux1`(x)	`aux1` = x; **return** `aux1`
`forwhile`(s,e,l)	**for** `i0` = s; `i0` $\leq$ e; `i0++` **if** timeout (128) **exit loop** **if** l returns zero **exit loop** **return** `i0`
`IFLTE`(x,y,*t1*,*t2*)	**if** $x \leq y$ **return** *t1* **else return** *t2*
`Ifeq`(x,y,*t1*,*t2*)	**if** $x = y$ **return** *t1* **else return** *t2*
`ifopen`(x,*t1*,*t2*)	**if** x = 5, 13, 31 or 43 **return** *t1* **else return** *t2*
`ifmatch`(x,y,*t1*,*t2*)	**if** x = 5, 13, 31 or 43 evaluate y **if** (x,y) = (5,71), (13,103), (31,137) or (43,167) **return** *t1* **else return** *t2* **else return** *t2*
`Makenull`	clear stack; **return** 0
`Empty`	**if** stack is empty **return** 0; **else return** 1
`Top`	**if** stack is empty **return** 0; **else return** top of stack
`Pop`	**if** stack is empty **return** 0; **else** pop stack and **return** popped value
`Push`(x)	Evaluate x; **if** $<$ 99 items on stack push x; **return** x **else return** 0
Indexed memory is held in store[$-l$ ... $+l$], where $l = 63$, i.e. a total of 127 cells.	
`read`(x)	**if** $\|x\| \leq l$ **return** store[x] **else return** 0
`write`(x,d)	**if** $\|x\| \leq l$ store[x] = d; **return** original contents of store[x] **else** evaluate d; **return** 0
`swap`(x,y)	**if** $\|x\| \leq l$ **and** $\|y\| \leq l$ exchange contents of store[x] and store[y] **if** $\|x\| > l$ **and** $\|y\| \leq l$ store[y] = 0 **if** $\|x\| \leq l$ **and** $\|y\| > l$ store[x] = 0 **return** 1