Stack-Based Genetic Programming

Timothy Perkis
1048 Neilson St., Albany, CA 94706
email: timper@holonet.net

Abstract

Some recent work in the field of Genetic Programming (GP) has been concerned with finding optimum representations for evolvable and efficient computer programs. In this paper, I describe a new GP system in which target programs run on a stack-based virtual machine. The system is shown to have certain advantages in terms of efficiency and simplicity of implementation, and for certain classes of problems, its effectiveness is shown to be comparable or superior to current methods.

1 Introduction

Genetic Programming has emerged as an effective means of automatically generating computer programs to solve a wide variety of problems in many different problem domains. But the optimum representation for "well-bred" programs is perhaps still to be found. Programs that are generated automatically by an evolutionary process have to be successful in two very different environments. First, they must thrive in the evolutionary domain, taking a shape which allows them to be manipulated as a genotype by some set of genetic operators. And secondly they must be effective computer programs which solve the problem set to them; in this guise, they must be utterances in some language which has adequate expressive power for the job at hand. In the absence of some compilation or development process, one program representation must serve these two very different functions.

The tension between these conflicting constraints has generated a variety of approaches to finding an appropriate program representation and concomitant virtual machine for the execution of automatically generated programs. Any chosen representation will have certain biases, and be more adept at creating some program structures than others; the art in designing the representation — for it is still an art, whose principles are not well understood — is in finding a program representation which will readily encode useful program structures under the influence of the evolutionary processes applied. In a sense, we are seeking a representation in which the evolutionary processes tend to create individuals only in the subspace of possible computer programs which are likely candidates for solving the problem at hand. In this paper I propose a new program representation and virtual machine which is designed for simplicity and flexibility in the evolutionary domain, while still offering adequate expressive power to generate programs of arbitrary complexity.

2 Genetic Programming

Genetic Programming (GP) is a development or extension of the Genetic Algorithm(GA). GA, as first described in (Holland, 1975) is a problem solving method in which potential solutions are encoded in fixed length bit strings. A population of these potential solution strings is maintained, and a correct solution is sought through an iterative process of repeated testing and generation of new potential solution strings. New solution strings are created by simple combination and mutation of existing strings, with the right to reproduce allocated proportional to the fitness of the individual strings. Over time, the population as a whole converges on a solution.

Koza (Koza 1992) has developed a hierarchical extension to GA in which a similar process is employed to develop LISP programs. Unlike conventional GA, in which the reproducing individuals are fixed-length binary strings, Koza's GP begins with individuals consisting of randomly generated LISP s-expressions. Each expression is composed of a set of functions and terminals chosen for the problem. The functions each take some fixed number of arguments composed of terminals or other functions, and the terminals are used for input variables, constants, or input sensors, depending on the nature of the problem. A special form of crossover operator is defined which maintains program syntax; a crossover consists of replacement of any program segment contained in matched parentheses in one individual with a similar segment in another. (If the nested LISP expression is thought of as a tree, this process amounts to switching arbitrary subtrees.) In addition, each function used must be defined such that it fails gracefully if given inappropriate data: for example, the division function used must not terminate if asked to divide by zero. Problem solutions are expressed either as the evaluated value of the entire expression, in problems which demand a single numerical result, or through the side-effect action of the functions used, as in a planning problem.

Various extensions and modifications to this method of program representation have been developed by Koza and others. Several of these changes amount to modifications of the virtual machine which interprets target programs. These include the generation of automatically defined functions (Koza 1992), the tagging for reuse of useful program modules that arise (Angeline and Pollack 1993), the incorporation of a pseudo-developmental process for program repair (Banzhaf 1993) and the addition of scratchpad memory which target programs can use to record state information (Teller 1993).

3 Stack GP

3.1 Implementation

In the first implementation of the stack-GP system, as in Koza, target programs (critters) are LISP s-expressions consisting of combinations of functions and terminals, but their organization is different. Functions do not have any LISP arguments, and their return value is irrelevant: all functions receive arguments from a **numerical stack** and return their result by pushing it on the stack. Function calls are not nested: critters consist of flat linear sequences of functions and terminals.

Terminals are simply a class of functions which push preset variables onto the stack when they are executed. Typically the error function which is evaluating a critter program will set the value(s) of these terminal variables before running the critter. In addition, a structure analogous to Koza's "ephemeral random constant" (ERK) has been defined. There is one function which violates the basic rule of taking no arguments except for stack arguments; this **SPUSH** function takes one numerical argument and pushes it on the stack. At program generation time, if ERK is enabled, instances of this **SPUSH** function, each with one randomly generated numerical argument, are assembled into the critter program sequence, along with instances of the other functions specified for the problem. For example, the following is an example of a program which solves the symbolic regression problem 3x^3 + 2x^2 +x. ERK range was set in this case to integers in the range (0,4). (**SPUSH** is written here as **>>**).

```
( (.X) (>> 3) (.*) (.X)  (.X)  (.X) (.X) (.X)
(.+) (>> 1) (.+) (.X) (.+) (.X) (.+) (.*) (.+) (>> 1) (.+) (.*))
```

In stack-GP one additional type of closure constraint must be imposed on the functions. Since functions all take arguments from the numerical stack, and the state of the stack at their time of execution is unknown in advance, it is unknown whether their arity will be satisfied. All the functions used are defined to do nothing when their arity is unsatisfied by the current stack state: For example binary ops such as **(+)** and **(-)** will do nothing if they are called when the stack depth is less than two; a logical **not** function will do nothing if stack depth is less than one, and so on. The stack is protected from underflow by this constraint; stack overflow has not been a problem, and has been limited in practice so far by specifying a maximum allowable program length. Following is an example of a typical function definition:

```
(defun .+ ()
      " stack-GP addition function."
    (if (> stack-depth 1)
        (spush (+ (spop) (spop))))
```

Table 1 shows another short program which solves the regression problem 0.5x^2, and traces the state of the numerical stack as each function is executed.

Step	Point of Execution	Stack	Comment
1	(*) | (X) (+)(X) (*) (X) (X) (+) (X) (/) (/)		If the stack doesn't have at least 2 items when (*) is called, it does nothing.
2	(*) (X) | (+)(X) (*) (X) (X) (+) (X) (/) (/)	5.0	Push current value of terminal X on stack.
3	(*) (X) (+) | (X) (*) (X) (X) (+) (X) (/) (/)	5.0	(+) does nothing if there are less than 2 items on stack.
4	(*) (X) (+)(X) | (*) (X) (X) (+) (X) (/) (/)	5.0 5.0	Push current value of terminal X on stack.
5	(*) (X) (+)(X) (*) | (X) (X) (+) (X) (/) (/)	25.0	(*) pops top two stack items, multiplies them and pushes result.
6	(*) (X) (+)(X) (*) (X) | (X) (+) (X) (/) (/)	25.0 5.0	Push current value of terminal X on stack.
7	(*) (X) (+)(X) (*) (X) (X) | (+) (X) (/) (/)	25.0 5.0 5.0	Push current value of terminal X on stack.
8	(*) (X) (+)(X) (*) (X) (X) (+) | (X) (/) (/)	25.0 10.0	Add top two stack items and push result.
9	(*) (X) (+)(X) (*) (X) (X) (+)(X) | (/) (/)	25.0 10.0 5.0	Push current value of terminal X on stack.
1 0	(*) (X) (+)(X) (*) (X) (X) (+) (X) (/) | (/)	25.0 2.0	Divide 2nd from top of stack by value on top and push result.
1 1	(*) (X) (+)(X) (*) (X) (X) (+) (X) (/) (/) |	12.5	A perfect solution to 0.5 x^^2.

Table 1. Example showing stack state as a test program is executed. The sequence in column 1 is an actual 100% correct result found in one run for the 0.5(x^^2) symbolic regression problem (see text.) Notice how the program created the constant 2.0 by computing (X + X) / X in steps 6-10.

Numerical calculations are performed in Reverse Polish Notation(RPN). RPN has an advantage in this context in that the parse tree for the calculation is expressed simply by the **order** of the functions and terminals in the sequence and not by a constrained syntax demanding balanced parentheses. Parse trees of arbitrary complexity can be generated, limited only by the permitted stack depth and program sequence length. This method has been used in other contexts where efficiency of execution and concision of coding are essential: in hand-held calculators, and in the FORTH and Postscript languages. (Kelly and Spies 1986, Adobe 1985).

It is of great value for GP that one can build programs with no grammatical constraints whatsoever: any random sequence of RPN based instructions will form a legal executable program. No special crossover operators are needed: programs sequences can just be cut and recombined at any point and a new parse tree will result.

The stack has been purposely left free to assume any state it will, without any constraints guaranteeing that the arity of any function will be met when that function is called. Likewise, for most problems, the error figure is derived from the top value on the stack when the function returns, without any penalty for leaving extra junk on the stack below.
Of course, the flexibility of using a stack as the returned data structure also allows problems which involve returning complex or multiple values to be easily implemented, without imposing any special syntactical constraints on the program structure. For example, the following is a 100% successful program defining the formula for complex multiplication, if we use the top two values returned on the stack as its answer, even though it returns with five values on the stack. (Values in boldface contribute to the answer; the rest is junk.)

(.X4) (.X2) (.*) (.+) (.X1) (.*) (.-) (.*) (.+) (./) (.X1)(./) (.*) (.X4) (.X1) (.X4) (.X4) (.X3) (.-) (.-) (.+) (.*) (.X2) **(.X4) (.X1) (.*) (.X3) (.X2) (.*) (.+)(.X3) (.X1) (.*) (.X2) (.X4) (.X4) (.X4) (.-) (.+) (.*) (.-)**

Initial populants are just random sequences of functions from the function set chosen for the problem: due to the nature of RPN, this is sufficient to generate program parse trees of varied shape and depth, and no special means are necessary to ensure an initial range of complexity.

3.2 Genetic operators and selection schemes

In terms of its evolutionary behavior, stack-GP is in many ways more similar to a traditional GA than Koza's GP. Since there are no syntactical constraints on the critter sequences, they can be treated as strings, and any of the several genetic crossover operators in use could be chosen. (Goldberg 1989) The crossover operator I've chosen to use for all the experiments is a two point crossover which generates one offspring. (see table 2.) This scheme permits offspring sequences to be shorter or longer than their parents, as necessary.

Mother	(>> 3.223) (.sin) ^(.sin) (.+) (>> 0.3323) (.-) (.*)^ (.*) (./) (.PI) (.sin)
Father	**(./) (.+) (.sin) ^(>>0.184) (.sin) (.PI) ^(.PI) (.*)**
Child	(>> 3.223) (.sin) **(>>0.184) (.sin) (.PI)** (.*) (./) (.PI) (.sin)

Table 2. The crossover operation used in the stack-GP system. Two points are picked at random in each of the parents, and one child sequence is created by inserting the sequence enclosed by the points in the father into the space defined by the points in the mother.

Likewise, any of the various selection and reproduction schemes in use in basic GA systems could be employed. I've chosen to implement a simple steady state tournament selection and reproduction scheme, which operates as follows:

> Loop until termination criterion satisfied:
> - •Pick 3 individuals at random from the population.
> - •Replace the least fit of the three with the offspring of the better two.
> - •With probability μ, perform point mutation on new critter.
> - •Evaluate the fitness of the new critter.

This simple tournament selection scheme is adequate to ensure fitness ranking proportionate reproduction. *

There is a mutation frequency parameter in the system. Typically every 1000th or so individual created will undergo a point mutation, which consists of changing some one function call in the sequence to some other function in the current function set.

4 Experiments

4.1 Method

Koza describes a method for assessing the amount of processing required for a particular problem. A problem is run for a large number of runs, and an instantaneous probability of success for each generation *i* is computed based on the observed performance. From this figure a cumulative probability, *P(M,i)* can be computed, where *M* is the population size and *i* is the generation. *P(M,i)* is the probability of success for all generations between 0 and *i*. From this we compute a number *R(z,i)* , the number of independent runs necessary to solve a problem with probability *z* if the system is allowed to proceed to generation *i*. (In all cases we use *z* = 99%).

$$R(z) = [\ \log(1-z)\ ]\ /\ [\ \log(1 - P(M,i))]$$

For the purpose of assessing the stack-GP system's performance by comparison with Koza's published benchmarks, three problems were run using this method. The problems, all taken from Koza, were run under conditions as close as possible to those of Koza's benchmarks. In our case, since we are using a steady-state reproduction method, the "generations" are virtual, and represent a period of *M* new populants being created and tested. (see figures 1, 2 and 3.) The graphs, following Koza's format, show *I(M,i,z)* in the dotted sequence and *P(M,i)* in the line sequence. *I(M,i,z)* is just *R(i,z)* * *M* * NGens; it represents the number of individuals which must be processed to be certain within probability z of receiving an answer, running *M* individuals to generation *i* in each run. The cartouche contains optimum NGens/run,(called henceforth *i**) and the number of individuals represented by the minimum of the *I(M,i,z)* curve; that is, the minimum number of individuals we can expect to process to get an answer (to probability *z*) if we make each run of optimum length. This number *E* is the figure of interest in

---

* In a population of size M, call the rank fitness order of any one individual i N(i) where the most fit individual has N(i)=0 and the least fit has N(i) = M-1. Call the fitness ranking F(i) = N(i) /(M-1) , such that the individual with F(i) = 0.0 is the most fit and that with F(i)=1.0 the least fit. Then the probability that any individual will be the loser of a three-way tournament of the type described above is F(i)^2, that is, proportional to the square of its fitness-ranking.

assessing the performance of the stack-GP system. Tables 3, 4 and 5 describe the test problems and give examples of some of the solution programs created.

Objective:	Find a function of one independent variable and one dependent variable, in symbolic form, that fits a given sample of 10 data points (x(i], y(i)) where the target function is 0.5x^2.
Terminal set:	.X (the independent variable), and ERK, the ephemeral random constant, which pushes a value on the stack in the range [-5.0, 5.0].
Function set:	.+, .-, .+ ./ (protected division.)
Fitness cases:	A given sample of 10 data pairs (x,y) with x from the interval [0,1].
Raw fitness:	Sum over the 10 fitness cases of the absolute difference between the value on the top of the stack-GP numerical stack left by the critter program and the target value y(i) from the input data pairs.
Standardized fitness:	Same as raw fitness for this problem.
Hits:	Number of fitness cases for which the top-of-stack value left by the critter program comes within 0.01 of the target value y(i).
Wrapper:	None.
Parameters:	M = 200, G = 21, μ = 0.05. Initial critter length range: [10->40].
Success predicate:	a critter program scores 10 hits.
Example correct program:	((.X) (>> 2.5503) (./) (./) (>> -0.7833) (./) (.X) (.+) (.X) (.*) (.+))

Table 3. Tableau for regression problem 0.5x^2.

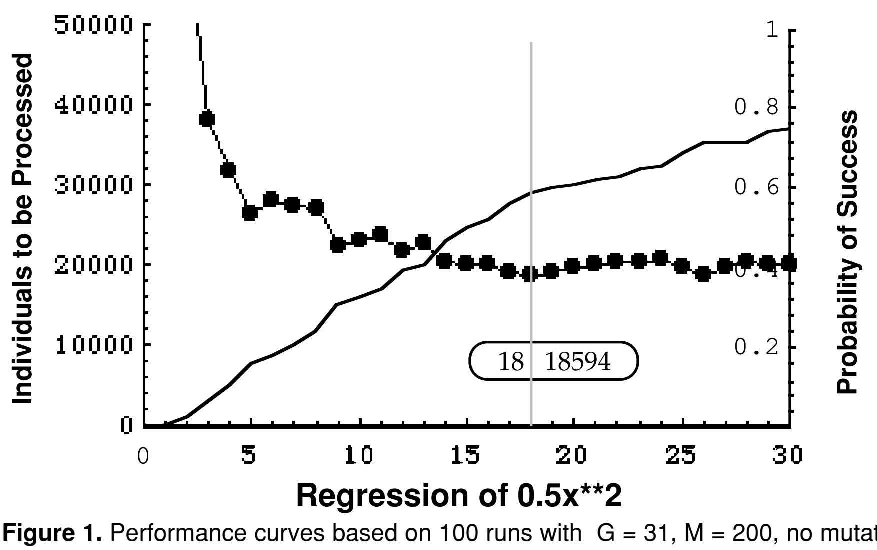

Figure 1. Performance curves based on 100 runs with G = 31, M = 200, no mutation.

Objective:	Find a critter program encoding a boolean expression whose output is equal to a three input boolean function; in this case, the majority on function (truth table: 0010111) .
Terminal set:	.D0, .D1, .D2
Function set:	.and2, .and3, .or, .not
Arity of functions:	2,3,2,1
Fitness cases:	The 8 possible combinations of the three boolean arguments.
Raw fitness:	Number of mismatches between the top of stack value left by the critter program and the correct answer for each of the eight possible input states.
Standardized fitness:	Same as raw fitness for this problem.
Hits:	Same as raw fitness for this problem.
Wrapper:	None.
Parameters:	M = 100, G = 21, μ = 0. Initial critter length range: [10->40].
Success predicate:	a critter program scores 8 hits.
Example correct program:	((.D1) (.AND3) (.AND2) (.D2) (.D0) (.D0) (.AND3) (.D1) (.AND3) (.D1) (.AND3) (.AND2) (.OR) (.OR))

Table 4. Tableau for boolean majority on problem.

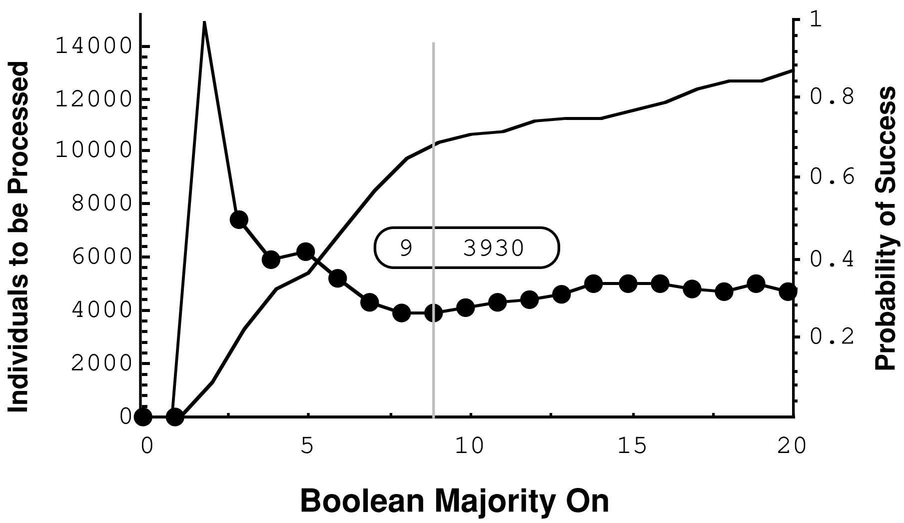

Figure 2. Performance curves based on 100 runs with G=21, M=100, mutation rate 0.05/individual.

Objective:	Find a critter program encoding a boolean expression whose output is equal to a three input boolean function; in this case,the even 3-parity function (truth table: 01101001).
Terminal set:	.D0, .D1, .D2
Function set:	.and, .or, .nand, .nor
Arity of functions:	2,2,2,2
Fitness cases:	The 8 possible combinations of the three boolean arguments.
Raw fitness:	Number of mismatches between the top of stack value left by the critter program and the correct answer for each of the eight possible input states.
Standardized fitness:	Same as raw fitness for this problem.
Hits:	Same as raw fitness for this problem.
Wrapper:	None.
Parameters:	M = 100, G = 20, μ = 0. Initial critter length range: [10->40].
Success predicate:	a critter program scores 8 hits.
Example correct program:	((.AND) (.NAND) (.AND) (.NOR) (.AND) (.AND) (.AND) (.OR) (.NOR) (.NOR) (.OR) (.OR) (.D2) (.OR) (.D2) (.NAND) (.AND) (.NAND) (.D1) (.D0) (.AND) (.D0) (.OR) (.OR) (.D1) (.NAND) (.D2) (.NAND) (.NAND) (.D0) (.NOR) (.NOR))

Table 5. Tableau for boolean even 3-parity problem.

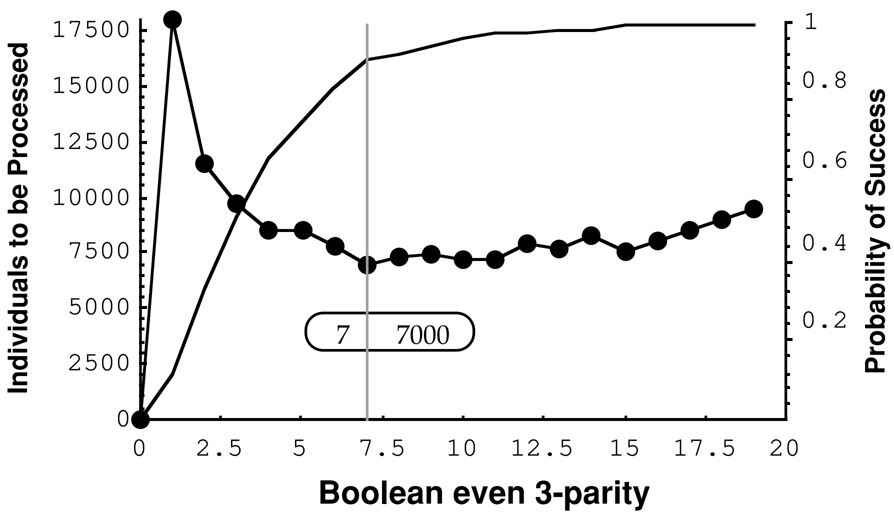

Figure 3. Performance curves based on 100 runs with G=20, M=500, mutation rate 0.05/individual.

4.2 Performance comparisons and stack manipulation functions

The system worked using the same function sets on the problems as those used by Koza. But these function sets don't take full advantage of the character of the stack-based system; therefore, an attempt was made to assess the performance of the system when special stack manipulation functions were added.

In the symbolic regression problem an additional set of runs was made which structured the problem in a form more fitting for the stack architecture. There was no **x** terminal for input values: instead the stack is seeded with the input value for the test case before each critter is run. Two stack manipulation functions are added to the function set: DUP, which simply duplicates the top of stack item and pushes the copy onto the stack, and SWAP which swaps the two top stack items.

In the Boolean problems, sets of runs were made using two different function sets in addition to the set of runs made using Koza's original function set. One set used conditional ?DUP and ?DROP functions which respectively duplicated or removed the top of stack item if it's value was TRUE. The other set of runs used unusual logical operators which were discovered serendipitously: they were buggy versions of logical operators which, for certain combinations of arguments, left one of their original arguments on the stack beneath their result. I include their results because the performance improvement using these functions is so striking. * The following tables 6, 7 and 8 summarize the comparative performance of runs of the stack-gp system under different conditions and benchmark performance published in (Koza 1992).

Run conditions:	E
Koza's published results M= 200, G=31 F = { *, /, + ,- } T = { x }	19,800
Stack-GP: M, G , F as above.	18,594
Stack-GP: M, G as above, F = { *,/,+, -, dup}, no terminals	16,882

Table 6. Performance in the0.5x^2 regression problem. Based on 100 runs.

Run conditions:	E
Koza's published results M= 100, G=21 F = { and2, and3, or, not}	4800
Stack-GP: M, G , F as above.	7474
Stack-GP: M, G as above, F = { and2, and3, or, not , ?dup, ?drop}	4319
Stack-GP; M,G as above, F = (bug-and2, bug-and3, bug-or, not} (see text)	3930

Table 7. Performance in the3-majority-on problem. Based on 100 runs.

Run conditions:	E
Koza's published results M= 4000, G=21 F = { and, or, nand, nor}	80,000
Stack-GP: M= 500, G=21, F as above.	NO ANSWER FOUND
Stack-GP: M= 500, G=21, F = { and, or, nand, nor, ?dup }	25664
Stack-GP: M= 500, G=21, F = { bug-and, bug-or, bug-nand, bug-nor} (see text)	7000

Table 8. Performance in the even-3-parity problem. Based on 100 runs. Limited computational resources necessitated limiting population sizes to 500, rather than using Koza's 4000.

---

* The strange, buggy version of AND was defined as follows:

```
 (defun and ()   ; BUGGY VERSION!
            (spush (and (spop) (spop)))
```

But notice: normal Lisp "and" doesn't evaluate the second argument if it doesn't need it to determine its result, so if the top-of-stack value is TRUE, then the second value is never popped. There is analogous behavior for the OR, NAND and NOR functions.

5 Discussion and future work

An array of other problems have been run without collecting the full performance statistics of the three problems above, including symbolic regression of trigonometric and quadratic functions. The system has also been used to compose musical compositions, with a functional set consisting of operators for time delays, setting musical intervals relative and absolute, and changing of rhythmic and melodic modes. The fitness function in this case was driven either by manual aesthetic assessment or by an algorithm which assessed harmonic and rhythmic complexity and rewarded attempts near a previously determined optimum.

I suspect that the fact that the stack is free to accumulate "junk" without effecting fitness may be a strong positive feature of this system: each program contains, along with code that determines the final result, "introns", code sequences that perform calculations which create that stack junk. Certainly these sequences may rise to become relevant in future offspring; and, more importantly, they may be buried fragments that were found useful in previous individuals. They represent, as in biology, genes which may have been useful in the past and may become expressed again in some future mutation. Future work to be done includes some systematic assessment of the evolution of these intron fragments, to see whether they improve as a population over time, and how performance is effected if they are excised along the way.

The availability of the stack for general use by the evolving program means problems which must evolve state representations for themselves could do so with this system.

The sensitivity of this system to limitations of maximum allowable program length needs to be explored further: I have noticed that performance suffers if one allows programs to be too long. This may be because long programs tend to leave more junk on the stack, such that only the end of the program really effects the outcome in those cases where only the returned top-of-stack value is of interest. Therefore allowing long programs just slows down the rate of meaningful crossovers; that is, those which effect the short sequence at the end of the program that really matters.

I have also only tried this scheme using one particular type of crossover operator, selection and reproduction scheme; more work needs to be done exploring alternatives in this area. And, with limited computational resources, the problems I have tried so far have been necessarily small; it is uncertain whether this method will scale as well or better than other GP methods.

The program was first implemented in Macintosh Common Lisp and then ported to C under Linux. Lisp was chosen for its desirable attributes as an interactive development environment, but efficiency increased over 40-fold with the C rewrite. The highly efficient run-time behavior of the critter programs is one of the most attractive features of the stack based approach. The virtual machine to run the critter programs is extremely small–less than 50 lines of C code– making this system attractive for embedded processor applications such as in control and signal processing.

Notice that in the stack-GP system as described so far there is no mechanism for allowing branching program execution. There are a large number of problems for which branching execution is not necessary. But in many problems of planning and strategy the side-effects of functions are of primary importance, and the actual sequence of program execution is of interest rather than the final calculated result. For these cases I have begun working with an extended stack-GP system, in which a second "execution stack" (XSTACK) is added to the system's virtual machine, as well as several functions to make use of it. NXPUSH is a function which takes a numerical argument from the numerical stack, and grabs that many items from the execution stream (that is, the queue of functions still to be executed), pushing this fragment onto the XSTACK. XPOP pops a code fragment off the XSTACK, appending it to the front of the execution stream, and NXPICK, (analogous to the FORTH "pick" operator) takes a numerical argument n as a pointer into the XSTACK, copying an item out of the XSTACK at depth n and pushing it onto the execution stream. One hopes these operators will permit the evolution of useful subroutines, branching and recursion, but work in this area is just beginning.

6 Conclusions

I have described a new method of genetic programming based on a stack-based virtual machine, and compared its performance on several problems with the basic GP system as published by Koza. In all problems tested, the stack-GP system took nearly the same number of candidate program evaluations to find solutions as Koza's system when the same function set was used. When additional stack manipulation functions were added, which took advantage of the special characteristics of the stack-based system, a considerable increase in efficiency was noted. This performance, and the simplicity and efficiency of the system's implementation, lead to the conclusion that stack-based genetic programming is a promising technique of evolutionary programming worthy of further consideration.

7 References

Adobe Systems Incorporated. (1985) *Postscript Language Reference Manual.* Reading, MA: Addison-Wesley.

Angeline, Peter J. and Jordan B. Pollack. "Coevolving high-level representations." Technical report 92-PA-COEVOLVE. Laboratory for Artificial Intelligence. Ohio State University. July 1993.

Banzhaf, Wolfgang. (1993) "Genetic Programming for Pedestrians." in *Proc. of the Fifth International Conference on Genetic Algorithms.* (ed. Stephanie Forrest), San Mateo, CA: Morgan Kaufman.

Goldberg, David E. (1989) *Genetic Algorithms in Search, Optimization, and Machine Learning.* Reading MA: Addison Wesley.

Holland, John H. (1992) *Adaptation in Natural and Artificial Systems.* Cambridge, MA: MIT Press.

Kelly, Mahlon G. and Nicholas Spies. (1986) *Forth: A Text and Reference.* Englewood Cliffs, NJ: Prentice-Hall.

Koza, John R. (1992) *Genetic Programming.* Cambridge, MA: MIT Press.

Tackett, Walter Alden. (1993). "Genetic Programming for Feature Discovery and Image Discrimination." in *Proc. of the Fifth International Conference on Genetic Algorithms.* (ed. Stephanie Forrest) San Mateo, CA: Morgan Kaufman.

Teller, Astro. (1994) "The Evolution of Mental Models." *Advances in Genetic Programming*, K.Kinnear, ed. MIT Press, 1994.