Ripple Crossover In Genetic Programming

Abstract. This paper isolates and identifies the effects of the crossover operator used in Grammatical Evolution. This crossover operator has already been shown to be adept at combining useful building blocks and to outperform engineered crossover operators such as Homologous and BLAH. This crossover operator, Ripple Crossover is described in terms of Genetic Programming and applied to two benchmark problems.
Its performance is compared with that of traditional sub-tree crossover on populations employing the standard functions and terminal set, but also against populations of individuals that encode Context Free Grammars. Ripple crossover is more effective in exploring the search space of possible programs than sub-tree crossover. This is shown by examining the rate of premature convergence during the run. Ripple crossover produces populations whose fitness increases gradually over time, slower than, but to an eventual higher level than that of sub-tree crossover.
[Another paragraph, lets leave it for a bit. I think we should mention the implications for CFGs here. The old para is commented out]

1 Introduction

An important characteristic of the function and terminal set of any Genetic Programming (GP) experiment is that it possesses the property of closure [4]. That is, the return type of any node, either function or terminal, can be taken as an argument by any function. An implication of this is programs produced by GP can only handle a single type, which clearly limits their utility.

Some work on Strongly Typed GP has relieved this by ... [Maarten, does your Grad Student Workshop stuff go in here] [Does Whigham belong here?][Wong and Leung?] [Conor, think so, but leave my work out of it for the moment, we'll sneak it in once the paper is accepted, who should be mentioned are: Montana, Whigham, Wong & Leung, but also for instance Gruau and Tina Yu, as Gruau gives compelling arguments for using CFG's.]

The most convenient way to describe a representation with multiple types is either through a Context Free Grammar(CFG) where one can specify the parameter and return types of each operator, or through an Attribute Grammar, by virtue of which one can pass very rich information through derivation rules. This information can describe anything from the routine parameter and return types, to the exotic, such as rich type information about data structures created on the fly. An example of this would if one wished to create a grammar that describes the multiplication operator for a structure such a matrix, the size of which is not available until derivation time for that particular derivation sequence.

Taking this route, however, one loses the desirable property of closure. Once this characteristic has been discarded, one can no longer expect standard GP crossover to produce syntactically well formed individuals. This is acknowledged by the above researchers, all of whom have special, custom-designed constrained crossover operators for their representation scheme.

Grammatical Evolution (GE)[6],a variable length string, GA-based automatic programming system employs a mapping process, where codons (integers) are read from a genome, and used to govern which decisions to make when generating an expression from a CFG. This paper begins with an investigation into the deceptively simple one-point crossover employed by GE. We term this crossover Ripple Crossover because of its non-local effects on the derivation tree. Changes at one node have a *rippling* effect to the right of the derivation tree. The effects of Ripple Crossover on a string of codons can be interpreted as changes to the derivation tree it denotes, it will be shown that this has an elegant interpretation for the simple function and terminal sets often employed in GP.

We conduct a number of experiments to compare the performance of ripple crossover and traditional *sub-tree* crossover with two different representation schemes.These experiments show that when using standard GP function and terminal sets, and the closure property they enjoy, ripple appears to be less likely to get trapped in a local optimum than sub-tree crossover.

It is argued that the property of Ripple Crossover to transmit on average half of the genetic material for each parent is the main cause of this. While sub-tree crossover exchanges less and less genetic material when the run progresses, ripple crossover will be equally recombinative regardless of the size of the individuals involved.

2 Context Free Grammars and Grammatical Evolution

[maybe we should put this and the next section into one, context free grammars and GE. Might save a bit of space and the choices of production rules and definition of GE go hand in hand]

Context Free Grammars(CFGs) are those grammars in which the syntax of a symbol, either a *terminal* which appears in the final output, or a *non-terminal* which is an interim symbol used to help generate the terminals, is the same regardless of what other symbols surround it.

A convenient descriptive notation for CFGs is Backus Naur Form (BNF). Like CFGs, BNF grammars consist of `terminals`, which are items that can appear in the language, i.e $+, -$ etc. and `non-terminals`, which can be expanded into one or more terminals and non-terminals. A grammar can be represented by the tuple, $\{N, T, P, S\}$, where N is the set of non-terminals, T the set of terminals, P a set of production rules that maps the elements of N to T, and S is a start symbol which is a member of N. For example, below is a possible BNF for a simple expression, where

$$N = \{expr, op\}$$

$$T = \{+, -, /, *, X, (,)\}$$

$$S = < expr >$$

And P can be represented as:

```
(1) <expr> ::= <expr> <op> <expr>     (A)
             | ( <expr> <op> <expr> ) (B)
             | <var>                  (C)

(2) <op> ::= + (A)
           | - (B)
           | / (C)
           | * (D)

(3) <var> ::= X
```

Table 1 summarizes the production rules and the number of choices associated with each. When generating a sentence for a particular language, one must choose carefully which productions are to used, as, depending on the choices made, a sentence may be quite different from the desired one, possibly even of a different length.

Rule no.	Choices
1	3
2	4
3	1

Table 1. The number of choices available from each production rule.

GE exploits this by maintaining a string of codons (integers) as its genome, which is decoded into a string of choices during the derivation process. In order to select a rule in GE, the next codon value on the genome is examined. As this value typically is larger than the number of available rules for that non-terminal, the modulo of the codon value with the number of rules is taken to decode the choice:

```
choice = codonValue MOD numberOfRules
```

Consider the selection of a production for rule #1 above. This has three choices available, so the subsequent codon value will have *modulos* 3 applied to it to make the choice.

Substitution of the left-most nonterminal continues until all of them have been replaced by terminals. It is possible for an individual to be over-specified, that is, to have unread codons left over. In this case the codons are simply ignored, but can be passed on to offspring. Furthermore, it is also possible for an individual to be under-specified. This happens when all codons have been read, but the individual still contains non-terminals. In this case, a few options are available. The individual can be *wrapped*, that is, read through a second time. Wrapping continues until either the individual is completely mapped or a certain upper limit of wrapping events has been reached. The individual's genome can be *extended* by making random choices until a complete individual is produced or a depth or size limit is exceeded. When this fails, the individual is deemed illegal and assigned a suitably punitive fitness value.

As individuals in GE decode into a string of choices that steer the generation of a derivation tree, it is instructive to examine the form these tree take. Quite often the entire genome does not need to be parsed to map an individual to a correct syntactic program. This means that individuals have a *tail* of non-expressed code. The tail is effectively a stack of unexpressed codons. Section **??** describes how crucial the tail is to the system.

The exact form of the tail depends on the context in which it is used, but it can be treated as one sub-tree, or a number of smaller trees. However, these trees are not expressed phenotypically.

3 Closed vs. Context Free Grammars

We use the term Closed Grammar to describe the type of grammar normally employed by GP practitioners. Although many GP users may be surprised to have it claimed that they have actually being using grammars, rather than simple sets of functions and terminals, this is indeed the case. A function and terminal set implicitly describes a grammar; one indicates the arity of the functions and, as every function can take every terminal as well as the output of every function, it is simply a matter adhering to the arity demands of every function to produce legal programs.

The arity of the functions could easily be described by a CFG. Consider the GP function and terminal set

$$F = \{+, *, -, \%\}$$

$$T = \{x, y\}$$

Where each of the four functions have an arity of two. This could also be described by the Context Free Grammar (CFG):

```
E ::== x | y | (+ E E) | (* E E) | (- E E) | (% E E).
```

where E is the start symbol.

This kind of CFG differs from the standard type only in that there is a single non-terminal. The use of a single non-terminal implies that the grammar has the closure property, so desirous of the GP crossover operator. Notice that use of more than one non-terminal does not preclude a grammar from being closed.

The question of its closure can only be resolved by determining if it can be rewritten as an equivalent grammar with a single non-terminal.

If one were to construct a derivation tree for any expression made up from this set, then clearly, any sub-tree from this grammar can replace a non-terminal, regardless of its position in the derivation tree. GP exploits this fact, although it uses parse trees rather than derivation trees, an entirely reasonable approach given that there is but a single non-terminal available to the grammar.

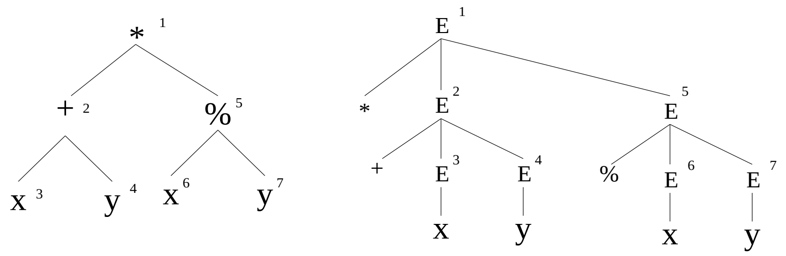


Fig. 1. A parse tree and its derivation tree. Note the numbering of legal crossover points.

Figure 1 shows a parse tree constructed from the above function and terminal set, together with the equivalent derivation tree constructed from the context free grammar given above. Because this context free grammar uses a prefix notation, the terminals of the derivation tree in Figure 1 form a prefix representation of the parse tree. This prefix ordering can be used as a memory-efficient implementation of a parse tree [**?**].

Now the connection between a prefix encoding of a parse tree and a string of rule choices in a context free grammar such as maintained by GE is clear. In the prefix encoding, every element is a reference to either a function or terminal. In GE, as this is a closed grammar, every element in the string will denote a choice in the set of rules that are associated with the same symbol. Thus, the prefix string:

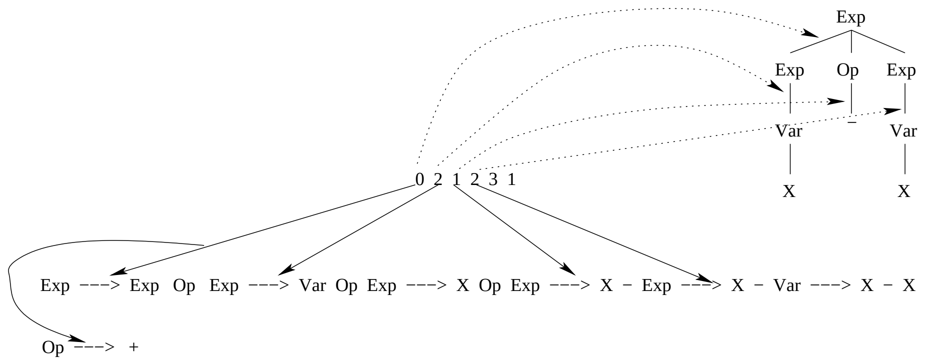
```
* + x y % x y
```

Has a one-to-one correspondence with the string of choices:

```
3 2 0 1 5 0 1
```

In the context of the grammar above. However, GE does not maintain a string of choices, but a string of integers, typically bounded above by a number much larger than the maximum number of rules. The decoding from an integer to a choice is usually done by using the modulo rule. Because of this redundant encoding there is actaully a one to many mapping from a prefix encoding to the integer encoding used by GE.

If one were to introduce more non-terminals into the grammar, sub-tree crossover would have to be constrained to make sure that the result of crossover will be a legal derivation tree. This can be done by employing the type information present in the derivation tree. However, when the number of types in the grammar grows, it can be expected that there will be a limited number of instances of each type in a tree. As sub-tree crossover is usually constraint to swap the same types, it may very well prevent the efficient exploration of the space of possible trees. In grammatical evolution, this is not an issue, as an integer will be decoded into a rule at runtime, i.e. will be decoded in the context of the symbol that is derived at the particular point in the derivation. This property of GE to change its form in the context of a different symbol we call *intrinsic polymorphism*, Figure 2 shows an example of polymorphism in the context free grammar for symbolic regression used in the experiments.



Exp
Exp
Op
Exp
Var
–
Var
X
X
0 2 1 2 3 1
Exp ---> Exp Op Exp ---> Var Op Exp ---> X Op Exp ---> X – Exp ---> X – Var ---> X – X
Op ---> +

Fig. 2. Intrinsic polymorphism: the same string of numbers can decode to different choices, depending on the symbol that they are being grafted onto. The grammar used is given in Section 2.

Notice that, in figure 2, the derivation tree is created in a pre-order fashion, that is, the left-most non-terminal is always the first to be modified by a production rule. This pre-order property has implications for crossover when one views the individuals as trees. We term the crossover employed by GE, and indeed, any tree based system which takes a pre-order view of the trees, *ripple crossover.* The remainder of the paper will focus on *ripple crossover* and its interpretation and utility in a tree based system.

4 Ripple Crossover in GE and GP

The previous section showed that GE individuals can be represented as parse or derivation trees, in a similar way to individuals in GP.

However, ripple crossover does not take this kind of tree structure into consideration, and simply crosses over the linear structures. Thus, when a crossover event takes place, the nodes to the left of the crossover point are preserved, the nodes to the right are removed. Figure 3 depicts what the effect of ripple crossover is on the underlying derivation tree for a closed grammar: it effectively removes all sub-trees to the right of the crossover point, rendering the derivation tree incomplete, with multiple crossover sites.

Each of the removed sub-trees are then added, intact, to the stack in the individual's tail. Crossover then involves swapping the newly modified tail with that of the mate. Each of the vacated ripple sites on an individual's spine are the filled by a sub-tree from the stack in the tail. If there weren't enough sub-trees removed from the mate, the previously unexpressed genetic material from the stack is used to create new sub-trees.

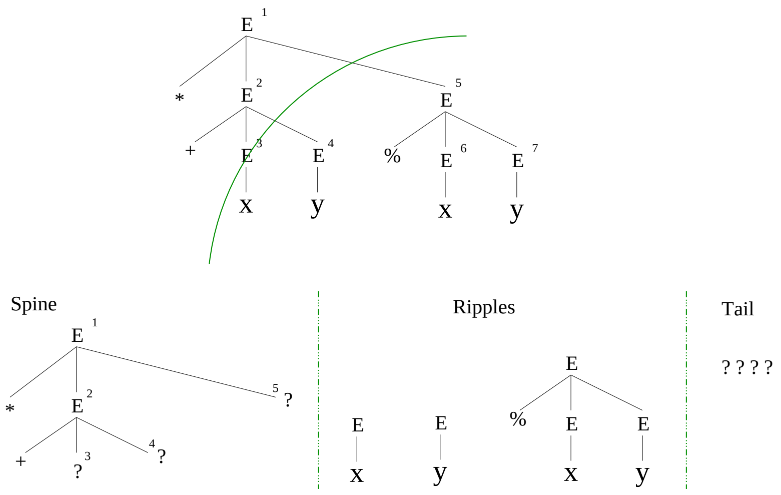


Fig. 3. Selecting crossover point 3 from Figure 1 results in a rippling effect, where points 4,5,6 and 7 will also be removed from the tree. The incomplete tree now needs three subtrees to be complete. The removed sub-trees and the unexpressed tail can be used to fill in crossover points of another tree.

When working with more than one symbol, the picture changes somewhat. Now the subtrees that are removed do not have a constant interpretation, that is, there is a possibility that a sub-tree will be grafted onto a ripple site that is expecting a different symbol. This is where the intrinsic polymorphism of GE takes effect, as the codon for the root node of the sub-tree takes on an appropriate value, as described in section 3. Crucially, if the new form of the codon, demands more codons to complete a mapping than are available from the current sub-tree, the system performs a *codon grab* from the next sub-tree in the task. This codon grabbing phenomena effectively ripples through the sub-trees until all the ripple sites are filled. If the tree is not fully derived by the time the final sub-tree is exhausted, codons are simply grabbed from the stack.

Because of the more linear nature of ripple crossover, on average half the genetic material of an individual is exchanged during crossover, which is considerably more than for sub-tree crossover, particularly as the average size of individuals increase.

The remainder of the paper will investigate whether the more global nature of ripple crossover indeed succeeds in exploring the search space more effectively than sub-tree crossover.

5 Experiments

In order to compare ripple crossover with sub-tree crossover, a number of experiments are performed on two common benchmark problems: the simple symbolic regression problem and the Santa Fe trail problem. The question investigated here is whether ripple crossover succeeds in exploring the search space more effectively than sub-tree crossover, i.e. whether ripple crossover is less likely to get trapped in local optima. To isolate the effects of crossover within these experiments, all experiments are performed using crossover only, and employ the same initialization procedure. All results have been obtained on 100 independent runs.

The initialization procedure consists of a random walk through the grammar, making random choices at each choice point. The individuals are initialed by extracting either the constructed derivation tree (for sub-tree crossover) or the sequence of choices (for ripple crossover). Because this random walk has a strong tendency to produce short individuals, a simple occurence check is implemented that re-initializes an individual when it is already present in the population.

The sub-tree crossover used in the experiments was implemented in its purest form: no bias was introduced to select terminals less often than non-terminals [?], or, in the case of crossover on context free grammars, no a-priori probabilities were specified to select certain symbols more often for crossover than others [?].

Likewise, the ripple crossover used was also simple. If, during the decoding process the generative string runs out of of genetic material, the individual is killed (i.e. gets worst fitness). No attempt was made to intialize the tail of the individual, also no wrapping was used. The only tweak that we allowed ourselves was to make sure that the crossover point for the spine was chosen in the expressed part of the string, thus eliminating a main source of straight reproduction.

To compare performances, two statistical tests are used. Firstly the widely used two-tailed t-test, and secondly a two-tailed re-sampling test. The re-sampling test does not assume normality of the distribution, nor equality of the standard deviations like the t-test does, and is used as the main check on the validity of the results. Both tests calculate the probability that the observed difference in the mean is due to sampling error and thus insignificant. The cutoff value was set at 5%. The results of both tests were in close accordance with each other on all experiments.

For the symbolic regression problem, two grammars are used, the closed grammar:

```
E ::== x | (+ E E) | (* E E) | (- E E) | (/ E E).
```

And the context free grammar:

```
Exp ::== Var | Exp Op Exp.
Var ::== x.
Op  ::== + | * | - | /.
```

Note that the division operator is not protected, division by zero will result in a runtime error and the individual will get the worst fitness available [1]. Further details of this setup can be found in Table 2.

Objective	find $x^4 + x^3 + x^2 + x^1$
Population Size	500
Success Predicate	Root Mean Squared Error < 0.001
Values for x	twenty equally spaced points between [-1,1]

Table 2. Setup of the symbolic regression problem.

The Santa Fe trail problem used the following closed grammar:

```
E ::== move() | left() | right() | iffoodahead(E E) | prog2(E, E).
```

And the context free grammar:

```
Code      ::== Line | prog2(Line, Code).
Line      ::== Condition | Action.
Action    ::== move() | right() | left().
Condition ::== iffoodahead(Code, Code).
```

Where the function *prog2* executes the commands in sequence, *iffoodahead* checks whether there is food in front of the artificial ant and executes either the first or the second argument depending on the result. The *move* function moves the ant forward and *left* and *right* rotate the ant 90 degrees in the specified direction.

Objective	navigate the Santa Fe trail
Population Size	500
Success Predicate	find 89 pieces of food within 600 steps

Table 3. Setup for the Santa Fe trail problem

[1] Strictly speaking the closure property is violated by not protecting the division operator, but on the other hand, in realistic applications, default return values in the case of an arithmatic error are usually less desireable than the occasional faulty individual.

6 Results

Figure 4 shows the success rates for the four different setups on the symbolic regression problem. Although the setups employing ripple crossover both obtain a 100% success rate, it can not be concluded that ripple crossover performs significantly better than sub-tree crossover on the closed grammar for this problem. These three do however perform significantly better than sub-tree crossover on the context free grammar [2]. Failure rates of the ripple crossover on the closed grammar were on average 12% at the end of the run (against 0% on the context free grammar). This did not seem to impede the performance in the least.

For the three top-contenders this problem is easy to solve, so the question remains why the sub-tree crossover on the context free grammar performs so poorly. This might lie in the fact that the *Var* type in this grammar makes up a large part of any tree. Unlike with the closed grammar, the sub-tree crossover on the context free grammar is constrained to swap like with like, thus always swapping a variable with a variable. However, the fact that there is only one variable in the problem definition makes a large number of crossovers result in identical trees. Although this can be remedied by avoiding crossing over on the *Var* type, it does beg the question how much the user of such a system must know of the intricate relationship between the grammar, the derivation trees and the genetic operators to be able to set up such a system to get good results reliably.

For the Santa Fe trail problem, the success rates are depicted in Figure 6. Here the runs on the context free grammars perform significantly better than their counterparts on the closed grammar. The success rates between ripple crossover and sub-tree crossover on the context free grammar are however not significantly different [3]. Also here the failure rate was on average 12% on the closed grammar and 0% on the context free grammar.

It is however important to note that while the sub-tree crossover seems to converge before generation 20, the ripple crossover runs keep on improving. Note that for none of the setups any form of mutation is used: all genetic material is fixed at the beginning of the run. To investigate whether ripple crossover does indeed continue to improve, a new set of 100 runs was executed, this time for 200 generations. Figure **??** clearly depicts the capability of ripple crossover to keep improving over time. An extended run up to 500 generations (not depicted here) showed that ripple crossover approaches a success rate of 70%, which is almost twice the success rate achieved by sub-tree xover.

7 Discussion

The results bring up some important issues. While it is difficult to identify clearly which crossover method is best, each appears to have its own particular strength.

[2] Both the t-test and the re-sampling test indicated that the difference was highly significant (probability of a type 1 error was 0%).

[3] This was tested on an independent set of 100 runs, not depicted here.

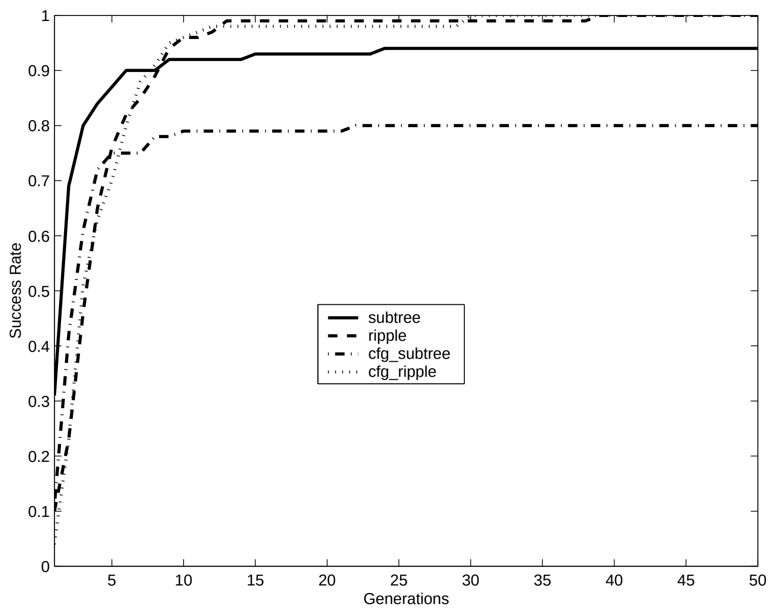


Fig. 4. Success rates on the symbolic regression problem, averaged over 100 runs.

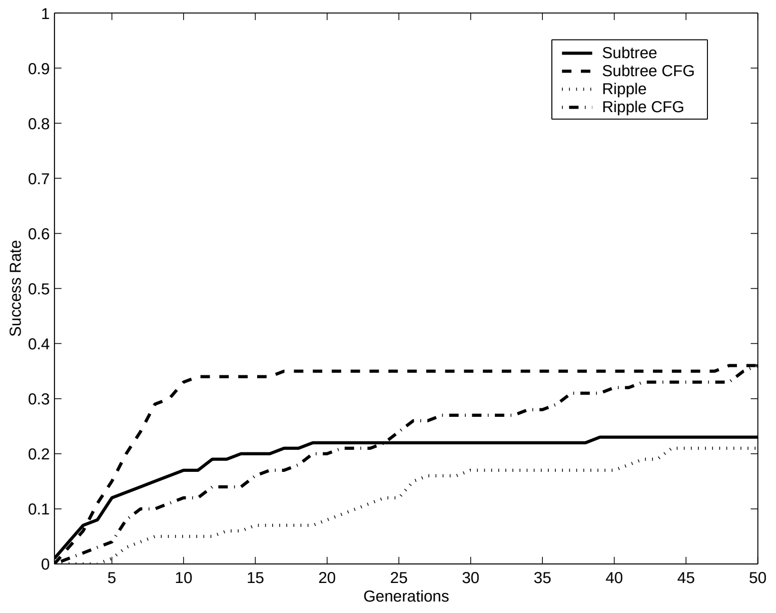


Fig. 5. Success rates on the Santa Fe trail problem, averaged over 100 runs.

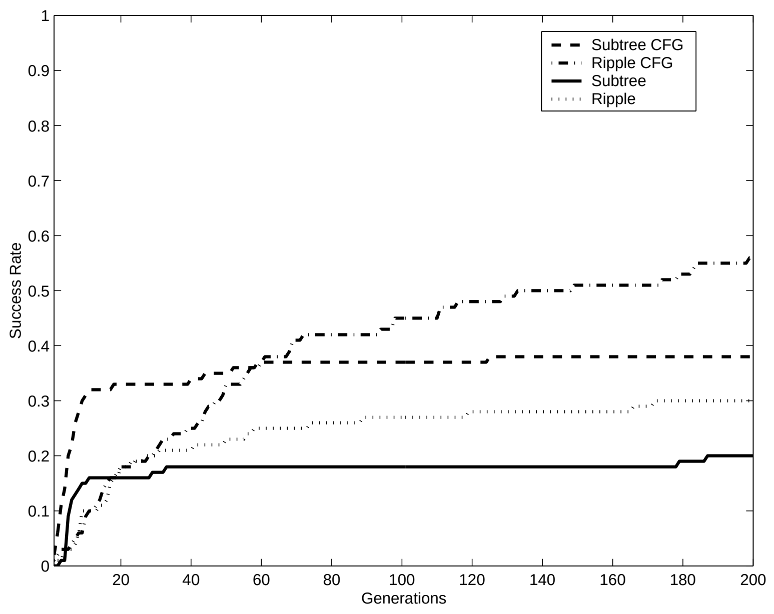


Fig. 6. Success rates on the Santa Fe trail problem, averaged over 100 runs.

As suggested by an initially steep curve, sub-tree crossover is particularly adept at obtaining solutions very early in a run. However, in all experiments, performance soon plateaus, with only the occasional increase in performance. This finding is in keeping with [?] in which is was suggested that GP performs a global search early on in a run, before gradually changing to a more local search as the run progresses and the population becomes characterised by large and often bloated individuals of similar if not identical fitness.

Ripple crossover, on the other hand, performs a more global search throughout a run, and is far less likely to become trapped at a local minimum. Indeed, for the Symbolic Regression problem, it never got trapped, while in the case of the Santa Fe tail experiments, fitness kept improving. This is because, regardless of how large individuals get, on average half the genetic material is exchanged during each crossover. It is this disruptive behaviour of the crossover operator that drives the population on to continually higher areas in the fitness landscape, but, ironically, it is also the cause of the relatively slow performance at the start of run.

This suggests that the use of ripple crossover will permit longer runs, with less chance of premature convergence. Such a property could be extremely valuable when one tackles more difficult problems that require more time to produce an optimal solution.

8 Conclusions

In this paper we have described the GE crossover operator in terms of both derivation and parse trees, and identified a phenomena we refer to as the *ripple effect*, which is the manner in which several sub-trees can be removed from the *spine* of an individual prior to crossover.

Our experiments clearly indicate that experiments which use ripple crossover have an altogether more graceful evolution than their sub-tree employing counterparts. Although the rate of increase is slower at the start relative to sub-tree crossover, ripple crossover invariably goes on to find a higher fitness level than sub-tree. In none of our experiments did a population which employed ripple crossover find itself trapped on a local minimum.

The use of Context Free Grammars in GP holds tremendous promise, as their descriptive power can be used to apply GP to all manner of highly complex problems, however, tree based crossover must be highly constrained to guarantee legal individuals. With ripple crossover, on the other hand, we demonstrate that tree encodings of CFGs not only routinely produce legal individuals, but also vastly outperform populations with sub-tree crossover.

In the writing of this paper, the value of linear chromosomes in general, and the GE system in particular, became quite clear to us. Ripple crossover occurs effectively for free in a linear system, because of the the pre-order nature of tree construction. Furthermore, the phenomenon of *intrinsic polymorphism* identified by this paper demonstrates the utility of context sensitive genes, that is, genes that can change their behaviour depending on the manner in which they are used. Rather elegantly, although the genes are polymorphic, they will always return to their initial state if used in the same manner again.

And then the beautiful princess woke up and discovered Conor had found his way into her room. After persuading her not to call security or to enforce the barring order that the so-called court had imposed, he got down on one knee and proposed. So delighted was the princess, that she agreed immediately, and the two of them soon married, and lived happily ever after. The evil trio of wizards, Mike, Mikeon and Maarten were thrown into the dungeon where they spent their days telling tales of the glory days with the arch wizard Azok.

References

1. Angeline, P.J. 1997. Subtree Crossover: Building block engine or macromutation? In *Proceedings of GP'97*, pages 9-17.
2. Francone F. D., Banzhaf W., Conrads M, Nordin P. 1999. Homologous Crossover in Genetic Programming. In *Proceedings of the Genetic and Evolutionary Computation Conference, GECCO 99*, pages 1021-1038.
3. Goldberg, David E. 1989. Genetic Algorithms in Search, Optimization and Machine Learning. Addison Wesley.
4. Koza, J. 1992. *Genetic Programming.* MIT Press.
5. Langdon W.B. 1999. Size Fair and Homologous Tree Genetic Programming Crossovers. In *Proceedings of the Genetic and Evolutionary Computation Conference, GECCO 99*, pages 1092-1097.

6. O'Neill M., Ryan C. Grammatical Evolution. *IEEE Trans. Evolutionary Computation*, To appear 2001.
7. O'Neill M., Ryan C. 2000. Crossover in Grammatical Evolution: A Smooth Operator? *Lecture Notes in Computer Science ????, Proceedings of the European Conference on Genetic Programming*, pages ???-???. Springer-Verlag.