# A Structure Preserving Crossover In Grammatical Evolution

**Robin Harper and Alan Blair**

School of Computer Science & Engineering
University of New South Wales
2052, Australia
robinh@cse.unsw.edu.au[1]

Abstract - Grammatical Evolution is an algorithm for evolving complete programs in an arbitrary language. By utilising a Backus Naur Form grammar the advantages of typing are achieved. A separation of genotype and phenotype allows the implementation of operators that manipulate (for instance by crossover and mutation) the genotype (in Grammatical Evolution - a sequence of bits) irrespective of the genotype to phenotype mapping (in Grammatical Evolution - an arbitrary grammar). This paper introduces a new type of crossover operator for Grammatical Evolution. The crossover operator uses information automatically extracted from the grammar to minimise any destructive impact from the crossover. The information, which is extracted at the same time as the genome is initially decoded, allows the swapping between entities of complete expansions of non-terminals in the grammar without disrupting useful blocks of code on either side of the two point crossover. In the domains tested, results confirm that the crossover is (i) more productive than hill-climbing; (ii) enables populations to continue to evolve over considerable numbers of generations without intron bloat; and (iii) allows populations (in the domains tested) to reach higher fitness levels, quicker.

## 1 Introduction

Grammatical Evolution (GE) is a method of utilising an evolutionary algorithm to evolve code written in any language, provided the grammar for the language can be expressed in a Backus Naur Form (BNF) style of notation. [Ryan *et al.*, 1998]. Traditional genetic programming, as exemplified by Koza [Koza, 1992] has the requirement of "Closure". Closure, as defined by Koza [Koza 1992] is used to indicate that a particular function set should be well defined for any combination of arguments. Previous work by Montana [Montana, 1994] suggests that superior results can be achieved if the limitation of "closure" required in traditional genetic programming can be eliminated, for example through typing. Whigham [Whigham, 1995] demonstrates the use of context free grammars to define the structure of the programming language and thereby overcome the requirement of closure (the typing being a natural consequence of evolving programs in accordance with the grammar). GE utilises the advantages of grammatical programming, but, unlike the method proposed by Whigham, separates the grammar from the underlying representation (or as this is commonly referred to; the genotype from the phenotype). It has been argued that the separation of genotype from phenotype allows unconstrained genotypes (and unconstrained operations on genotypes) to map to syntactically correct phenotypes. Keller [Keller 1996] presents empirical results demonstrating the benefit of this type of mapping. One of the interesting aspects of having such a simple underlying genotype as GE (a bit string) is that it is possible to design a number of operators, both stochastic hill-climbing and more traditional genetic information swapping operators, which act on this simple bit string. For the purposes of this paper all such operators will be referred to as "crossover operators" even although (as will be seen) some of them do not utilise any information from the second entity.

GE typically utilises a simple one-point operator. This has been criticised on the grounds that it is seemingly destructive of the information contained in the second contributing parent. O'Neill has defended the one-point crossover, comparing it to a crossover which exchanges random blocks [O'Neill & Ryan, 2000] and a form of homologous crossover where regions of similarity are swapped [O'Neill *et al.*, 2001]. In both these cases the conclusion is reached that the one-point operator is the most consistent of the operators examined in producing successful runs, despite its disruptive effect.

This paper introduces a method of crossover that utilises information automatically extracted from the grammar to reduce the disruptive impact of the crossover operator. It then describes a series of comparisons using one point and two point crossover operators where the only difference between runs was the type of crossover operator used. The results of these runs confirm the productive effect of the single point crossover as well as indicating that the new crossover operator proposed is more productive than the single point crossover. Sufficient runs were carried out in the test domains to achieve statistically significant results.

## 2 Grammatical Evolution

Rather than representing programs as parse trees GE utilises a linear genome representation to drive the derivation of the encoded program from the rules of an arbitrary BNF grammar. Typically the genome (being a variable length bit string) is split up into 8 bit codons (groups of 8 bits) and these codons are used sequentially to drive the choices of which branch of the grammar to follow. The maximum value of the codon is typically many times larger than the number of possible branches

[1] This research was partially funded by the Australian Research Council Centre of Excellence for Autonomous Systems

for any particular non-terminal in the grammar and a mod operator is utilised to constrain it to the required number.

For instance if a simple program grammar were as follows:

```
<Program> :: = <Lines>
<Lines> :: = <Action> | <Action > <Lines>
<Action> ::= North | South | East | West
```

Assume an individual had the following DNA and codon pattern:

```
DNA: 00100001 00010100 00100000 00000011 00010000
Codons:  33       20       32       3       16
```

No codon would be used for the first expansion (since there is only one choice). The initial codon of the genome would be used to determine whether to expand <Lines> to <Action> or to <Action><Lines>. The value of 33 would be MOD'd with two (since there are two choices) to give a value of 1. The second choice (<Action><Lines>) would then be chosen. The expression now is "<Action><Lines>". The first non-terminal <Action> is expanded by using the next codon (20). <Action> has four choices, so 20 would be MOD'd with four, to give zero and North would be chosen. The expression is now "North <Lines>". The next non-terminal (<Lines>) is expanded using the codon 32. 32 Mod 2 = 0, so <Action> is chosen, leaving us with "North <Action>". <Action> is then expanded using the next codon (3), to give us West. The expression is now "North West". There being no further non-terminals in the expression the expansion is complete. The remaining codon is not used.

If the expression can be fully expanded by the available codons (i.e. the expansion reaches a stage where there are no non-terminals) the individual is valid, if the codons run out before the expression is fully expanded the individual is invalid and removed from the population.

One of the original proposals of GE involved utilising a wrapping of the genome to decrease the number of individuals that are invalid [O'Neill & Ryan 1999]. Where wrapping is enabled, if the expansion is not complete by the time all the codons have been read, the codons are re-read from the beginning. Normally genomes are only allowed to "wrap" a certain number of times. Although the wrapping method is not easily accommodated within some of the proposed crossover methods, the traditional single point crossover and a codon boundary single point crossover were also implemented with wrapping enabled for the purposes of comparison.

## 3 Crossover Operators

A total of 12 crossover operators were implemented, the first seven (which include the standard crossover operator) are one point operators, the remaining five are two point crossovers and consist of the proposed new operator (the LHS Replacement operator) and four other two point operators designed to provide a fair comparison. These are introduced below.

### 3.1 Standard bit crossover

For each of the two individuals a crossover point is selected at random. The head (being the genome up to the selected crossover point) of the first is combined with the tail of the second and vice-versa. It should be appreciated that with GE this can cause a fair amount of disruption with respect to the "tail". The reason for this is twofold: (i) if the crossover point is "mid" codon, the codons in the tail will have a different value; and (ii) especially in more complex grammars, the tail codons will be used to interpret a different part of the grammar in the child, than the part they interpreted in the parent.

### 3.2 Codon crossover

A variation to the standard crossover limits the selection of the crossover point to codon boundaries, thus preserving the actual value of the codons in the tail, although as in the standard bit crossover they may well be used to interpret a different part of the grammar.

### 3.3 Matching Crossover

This crossover requires the extraction of a little additional information from the grammar, as we now explain.

#### 3.3.1 Extracting type information

As previously discussed when each codon is decoded the codons are used to drive the BNF grammar in order to derive an expression in the specified language. During this process it is simple to store with the codon (at the time the codon is being used to determine which RHS branch of the grammar to use) the type of non-terminal the codon expanded.

For instance, the following is an extract of the grammar used for the taxi domain problem (this problem is more fully described in paragraph 4.3) [For ease of reference only; each rule is numbered on the left hand side and each possible branch labelled on the right hand side.]

```
(1) <lines>:: =
        <line> |                                        (A)
        <line> <lines>                                  (B)
(2) <line>:: =
        if ( <cond> ) then { <lines> }|                 (A)
        if ( <cond> ) then { <lines> }else { <lines> } |(B)
        doLines { <lines> }until ( <cond> ) |           (C)
        <action>                                        (D)
(3) <cond> :: =
        <boolTest> |                                    (A)
        <cond> AND <cond> |                             (B)
        <cond> OR <cond> |                              (C)
        NOT <cond> |                                    (D)
        <IntegerValue> <comp> <IntegerValue>            (E)
(4) <boolTest> :: =
        havePassenger |                                 (A)
        NorthBlocked |                                  (B)
        EastBlocked |                                   (C)
        WestBlocked |                                   (D)
        [..etc ... ]
(5) <comp> :: =
        < |                                             (A)
        > |                                             (B)
        =                                               (C)
(6) <IntegerValue> :: =
        currentXpos |                                    (A)
        currentYpos |                                    (B)
        ( <IntegerValue> MINUS <IntegerValue> ) |        (C)
        ( <IntegerValue> PLUS <IntegerValue> ) |         (D)
        [...etc...]
    (7) <Action> :: =
        Pickup |                                         (A)
        Putdown |                                        (B)
        North |                                          (C)
        [...etc...]
```

Listing 1. Simplified Extract of the Taxi Domain Grammar

In decoding a particular codon we might have the following partial sequence of codons which have not yet been used ... 120 26 30 83 ... and we might, say, have reached a non-terminal of <line>.

Traditionally the next codon would be used to expand the <line> branch – labelled 2 in Listing 1. There are 4 alternative expansions, 120 MOD 4 gives 0, so branch (A) is chosen.

The symbol for <line> in the expression is replaced by the following code

if ( <cond> ) then { <lines> }

and the next non-terminal (in this example <cond>) would be considered and expanded utilising the next codon in sequence (in this example 26).

In order to extract the information required by the Matching Crossover operator, all that is required is to associate with the codon the type of non-terminal that was expanded by the codon. (In this case the codon (120) expanded non-terminal (2) <line> and the next codon (26) will be used to expand the non-terminal (3) <cond>). At this stage we have something like this:

```
DNA: ... 01111000 00011001 00100100 ...
Codons: ...  120      26      30 ...
Expands: ...  2        3  ...
```

### 3.3.2 Implementing the Matching Crossover

The Matching Crossover proceeds as per the simple one point crossover, however the crossover point selected on the second parent is constrained to that of a codon on the second parent that expands (or is associated with) the same type of non-terminal as the codon following the crossover point on the first parent. For instance, using the example codon extract discussed in paragraph 3.3.1 – if the second codon (which is 26 and has been used to expand a non-terminal of type 3 (<cond>)) happens to be chosen, the crossover will occur at a point in the second entity where the codon at the crossover is also used to expand a non-terminal of type 3. This codon is located by generating a random crossover point and then searching, incrementally, above and below the point until an appropriate match is found. Once the second crossover point has been determined the tails are swapped as in the simple one point crossover. If no match is found in the entity then the crossover operation fails and a new initial crossover point is randomly selected. Whilst this crossover does mean that the tail codons are used to interpret the same part of the grammar as in their donating parent, this crossover can still cause disruption further along the sequence. To appreciate this consider the following simple grammar

```
(1) Lines ::= Line| Lines
(2) Line :: = X | Y | Z.
```

Assume we have parents A and B which, when interpreted, have an expansion of the grammar like this:

```
Parent A:                      Parent B:
      Lines                          Lines
      /   \                            \
   Line  Lines                         Line
    /       \                            \
   Y       Line                           X
              \
               Z
```

For example, Parent A might have the following codons 1, 4, 0, 2, 4 ... This would expand as follows:

```
<Lines> ->
<Line><Lines> ->  [codon at position 1 relates to rule (1)]
 Y <Lines> ->     [codon at position 2 relates to rule (2)]
 Y <Line> ->...   [codon at position 3 relates to rule (1)]
 Y Z.             [codon at position 4 relates to rule (2)]
```

The remaining codons are not used.

Assume a second Parent with codons 0,0,10,5... This becomes:

```
<Lines>->
<Line>->          [codon at position 1 relates to rule (1)]
 X                [codon at position 2 relates to rule (2)]
```

The remaining codons are not used.

If these parents were to be crossed over at codon position 2 in Parent A and codon position 2 in Parent B (both these points relate to Grammar rule (2)), then the child would only inherit the first codon from Parent A and all the rest from Parent B i.e. the child's codons would be 1,0,10,5…

The first codon (from Parent A) (1) would expand <Lines> to <Line> <Lines> then the second codon in the child, being a codon from Parent B (0) would Expand <Line> to X, the expression now being X <Lines>. <Lines> would then need to be expanded but it will now be expanded utilising the previously unsused codons in Parent B (the 10, 5 etc) not the previous codons in Parent A. This has been termed the "Ripple Effect".

## 3.4 Tailless Crossover

In order to provide a test of the productivity of the crossover operator a tailless crossover, effectively a stochastic hill-climbing algorithm, was implemented; following the concepts discussed in [Jones 1995]. In this case the first crossover point was determined randomly as described in 3.1, however, instead of swapping the tail with the tail of a second parent individual, the tail was swapped with a randomly generated bit–string. This would serve to compare each of the other crossover operations with one where the second parent was a randomly generated parent.

## 3.5 Headless Crossover

This crossover is similar in principal to the tailless crossover except that the head was filled with a random bit string. Note that this is likely to be highly disruptive in GE as there is no guarantee that the interpretation of the new head will not terminate before the crossover point is reached. This "crossover" operator is in effect more akin to the production of a completely new individual.

### 3.6 Wrapping Bit Crossover

As discussed in paragraph 2, GE can utilise a method of "wrapping" the genome. That is, if the codons end before the grammar has terminated (i.e. before all non-terminals have expanded into terminals) the codons are re-read from the beginning. This crossover is the same as the Bit Crossover (one random crossover point on each parent), save that wrapping is enabled.

### 3.7 Wrapping Codon Crossover

This crossover is the same as the Codon Crossover (which constrains the crossover to codon boundaries), save that wrapping (as discussed in paragraph 2) is enabled.

### 3.8 Two Point – Bit Crossover

Here the first point is selected at random in each of the parents and a second point is selected after the first point and before the end of the Useful DNA (the Useful DNA is the part of the genome actually used to interpret the genotype – this is further explained in paragraph 4.2). This designates a codon sequence. A similar codon sequence is selected in the second parent. These two sequences are swapped.

### 3.9 Two Point – Codon Crossover

This is similar to the Two Point – Bit Crossover, except that the selected points are constrained to fall at codon boundaries.

### 3.10 Two Point – Left Matching Crossover

The two crossover points in the first parent are selected as in the Two Point – Codon Crossover, however (as in the Matching Crossover – paragraph 3.3) the choice of the first point in the second parent is constrained to a codon which expands the same type of non-terminal in the grammar. The second crossover point in the second parent is chosen randomly as in the Two Point – Codon Crossover.

### 3.11 LHS Replacement Crossover

This crossover uses the grammar to constrain both the second point in the first parent and the first and second points in the second parent, as we now explain.

#### 3.11.1 Extracting length information

The final piece of information we need to extract is the number of codons that are required to expand fully the non-terminal selected by the first crossover point. When a codon is used to expand the grammar, as well as saving the type information as described in 3.3.1, we can also push the current codon position onto a stack and add a POP operator after the expansion. If this were to occur the expression in 3.3.1 would look like this after the codon (120) had been used to expand <line>:

... if ( <cond> ) then { <lines> } POP ...

The stack and the POP operator allow the number of codons utilised in fully expanding the non terminal to be determined. (This number is the codon reading position when the POP is encountered less the codon position stored at the top of the stack.)

The computational cost in extracting this information is negligible. It is true that a certain amount of additional storage is now required to store each codon (32 bits per codon rather than 8) but as we demonstrate we believe the benefits outweigh this cost.

#### 3.11.2 Implementing the LHS Crossover Operator

With this type of crossover, the first crossover point in the first parent is, as before, selected randomly. The first crossover point in the second parent is selected as described in 3.3, however, instead of swapping tails between the two entities the number of codons required to fully expand the expression in the first entity are exchanged for the number of codons which are required to fully expand the expression in the second entity. (To put this another way, the second point in each parent is selected so that the codon sequence in between the two crossover points fully expands the non-terminal designated by the first crossover point.)

Building on the example in 3.3.1, the second codon in the example (which was 26), was used to expand the non-terminal <cond>. Assuming the codon sequence continued as:

30, 83, 45, 61

Then this would have been expanded as follows:

```
<cond> -> <cond> AND <cond>
                              (26 MOD 5 = 1 = branch (B))
<cond> AND <cond> ->
          <boolTest> AND <cond>
                              (30 MOD 5 = 0 = branch (A))
<boolTest> AND <cond> ->
          WestBlocked AND <cond>
                              (83 MOD 5 = 3 = branch (D))
WestBlocked AND <cond> ->
          WestBlocked AND <boolTest>
                              (45 MOD 5 = 0 = branch (A))
WestBlocked AND <boolTest> ->
          WestBlocked AND NorthBlocked
                              (61 MOD 5 = 1 = branch (B))
```

Therefore associated with the codon 26 would be the non-terminal it expands, here <cond> and the number of codons required to fully expand it (here 5, being the 26 plus the following sequence of codons: 30, 83, 45, 61). If this codon was selected as the first crossover point then it and the next 4 codons would be replaced by a codon which expands the <cond> non-terminal together with sufficient codons from the second individual to complete the full expansion of the <cond> non-terminal. In the second individual the same would happen, the list of codons (26, 30,83,45,61) would replace a sequence of codons that were used to fully expand a <cond> non-terminal.

It can be seen from this that, in the first individual, the codon which follows the 61 and was used to decode the <lines> non-terminal of the if statement will still continue to do so, despite the fact that the prior sequence of codons has been changed.

It will be noted that the LHS Replacement crossover has many similarities with the type of crossover utilised by Koza in genetic programming [Koza, 1992]. There are, however, differences between the two methods, including, for instance, the automatic encoding of type through the use of the grammar. Not only does this avoid the need for closure (e.g. there is no need to assign arbitrary values to the <Action> expressions in the taxi domain), there is research indicating that strong typing can aid in reducing the search space [Montana 1994]. In addition there are no limits on the depth of any part of the expression (although there are limits on the length of the bit string). A comparison between the normal GE crossover and a parse tree crossover [Keijzer 1991] showed areas where the GE crossover (termed the "ripple crossover") proved superior to the parse tree operator. We believe that more structural information is preserved with the LHS Replacement Crossover whilst maintaining the advantages of using a full BNF grammar. This could explain the increased productivity of the LHS Replacement Crossover.

### 3.12 Gutless Crossover

This crossover proceeds as per the Two Point – Bit Crossover, save that the selected sequences are filled with a random bit pattern rather than swapped. Again this "crossover" operator will operate like a stochastic hill-climbing algorithm.

## 4. The Test procedures

### 4.1 The GE environment

All the crossover operators were tested in various problem domains against a constant GE environment. In particular the following strategies were used:

- Selection of individuals was based on a probability selector, that is the chance of any particular individual being chosen to breed was directly related to its fitness.
- An elitist strategy of retaining the fittest 5% of individuals was adopted.
- A constant mutation rate of 1 in 2,000 (independent of the length of the DNA string) was applied to each child generated.
- Two children were generated for each crossover operator and the mutation operator was applied to each child.
- Invalid individuals were given zero fitness and were not eligible for selection or breeding.
- Individuals were started with a random bit string, in particular no attempt was made to ensure that the first random individuals were a minimum size, although if an individual was invalid it was regenerated.

### 4.2 Bloat control

As noted in paragraph 2 the wrapping operator sometimes used in GE is not compatible with either of the two new crossover operators and it was turned off other than in respect of the two wrapping crossover operators (the Wrapping Bit Crossover – paragraph 3.6, and the Wrapping Codon Operator – paragraph 3.7). Previous work [O'Neill and Ryan, 1999] indicates that the wrapping procedure is effective in helping to contain bloat within GE. In this case an alternative strategy was adopted and tested against the Santa Fe ant domain described in their paper. The system had no problems correctly evolving fit individuals for the Santa Fe problem domain.

In formulating the bloat control strategy employed it was noted that there is typically, within any individual, a difference between the number of codons stored by the individual (the DNA Length) and the number of codons actually utilised by that individual (the Useful DNA Length). If a first crossover point is selected anywhere within the DNA Length, then there is a tendency for individuals to evolve long DNA Lengths, whilst keeping their Useful DNA Lengths short. In extreme cases this makes it unlikely any random crossover point will hit within the Useful DNA Length and thus the crossover will have no effect on the individual. By constraining the crossover points (and in the case of two-bit operators both crossover points) to be within the Useful DNA Length this problem is avoided. It should also be noted that it was found to be important to copy the whole of the DNA as the tail (not just the Useful DNA of the tail parent). The reason for this is that additional DNA may be needed if the tail happens to start interpreting at a different place in the grammar tree and requires more codons.

Both the Matching crossover and LHS Replacement crossover operators require this limitation, as the codon expansion information is only available for codons that are actually used in creating the expression. In other words for these crossover operators crossover can only occur within the Useful DNA part of the DNA.

Finally where DNA sizes in excess of 4000 bits were encountered, if two entities had the same score then the entity with the smaller DNA was ranked ahead. If DNA sizes exceeded 6000 the entity was deemed illegal.

### 4.3 Problem Domains

The results of two problem domains are presented here, the Minesweeper Domain discussed by Koza [Koza 1994] and the Taxi Domain [Diettrich 1998]. It should be noted that the results were tested (and confirmed) against other problem domains, including the Mower, Mopper and San Mateo Trail problems discussed by Koza [Koza 1994]. Despite the Santa Fe trail being the problem domain presented by previous papers which analyzed the effect of different crossovers in GE (see for instance [O'Neill *et al.*, 2001]) this was not used here as it is too easily solved by GE, and no statistically significant results could be obtained with respect to most of the crossover operators. As mentioned, whilst the same results were obtained in each of these domains, the difference between each of the five crossover operators tested was most apparent in the more difficult domains.

The Taxi Domain is not a problem that we are aware of previously being used as a test for genetic programming or grammatical evolution, although it is used in demonstrations of new algorithms in Hierarchical Reinforcement Learning. The essence of the problem is that a taxi inhabits a 5 by 5 grid world (see figure 1).

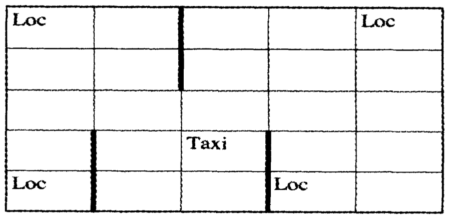

Figure 1 – Taxi Domain

There are 4 specially marked locations that may be either the pickup point of a passenger or the putdown point of a passenger (or both). The Taxi has to navigate (from a random start point) to the correct pickup point, pick up the passenger, navigate to the putdown point and deliver the passenger, within a fixed number of moves. Some modifications to the rewards for the agent were made to make the fitness functions suitable for a genetic algorithm technique. Typically the taxi receives a reward for successful completion and a penalty for an incorrectly executed pickup or putdown. We added an additional reward for a successful pickup as well as giving the taxi a small reward for the first time it moves closer to the pickup point or destination point (as appropriate). The taxi has access to complete information regarding the game world. Finally, whilst evaluation of fitness would normally be against random maps, the variation in reported fitness over time made it difficult to compare results between the crossover operators. Consequently, the taxi was evaluated against 20 fixed maps, the fitness being its combined score in all of the maps. (An extract of the grammar used is shown in listing 1.)

## 5 Results

### 5.1 Population Size versus Generation Count

Koza generally found with his GP experiments that it was better to have a large population with only about 50 generations [Koza 1992; Koza 1994], but in our case we have found that it is better to have smaller populations running for longer.

For instance, initial results using a typical grammatical programming profile of a large population (5000) over limited generations (50) produced barely acceptable results over 40 runs on the Minesweeper problem and disappointing results for the taxi domain problem. However, by decreasing the population to 100 and increasing the generations to 2500 (keeping the number of individuals generated at 250,000), the results were vastly improved. It is worth noting that although both runs require the same number of computations (250,000 individuals generated), the run with 5,000 individuals requires a lot more information to be kept in memory at the same time.

Our results indicate that the GE system described here is able to continue to evolve over a large number of generations (especially in the more difficult taxi domain, the results of which are presented below) and that, if analysed by numbers of agents created and evaluated, a low population size and long generation count provides better results. Figure 2, taken from the Minesweeper Problem domain, plots two 5,000 member population runs (using two of the crossover operators, namely Single Point Codon and LHS Replacement) as against two 100 member population runs (using the same two crossover operators). The 95% confidence interval is shown for each run. Although the X-axis represents different generations for each run (one unit = one generation in the 5000 population runs, one unit = 50 generations in the 100 population runs) they each equate to the same number of crossovers performed (one unit = 5000 crossovers). The top line in the chart equates to the LHS replacement crossover operator, with a population of 100 and 2500 generations. As can be seen it continues to improve (albeit slowly in this example, as increasingly the runs reach maximum fitness). The LHS Replacement operator with 100 population and 2500 generations was the only one of the four operators illustrated to score the maximum fitness (116), which it did so in 9 out of the 40 runs. The average fitness it achieved over the 40 runs (112) equates to the threshold fitness used by Koza [Koza, 1994] in his description of the problem.

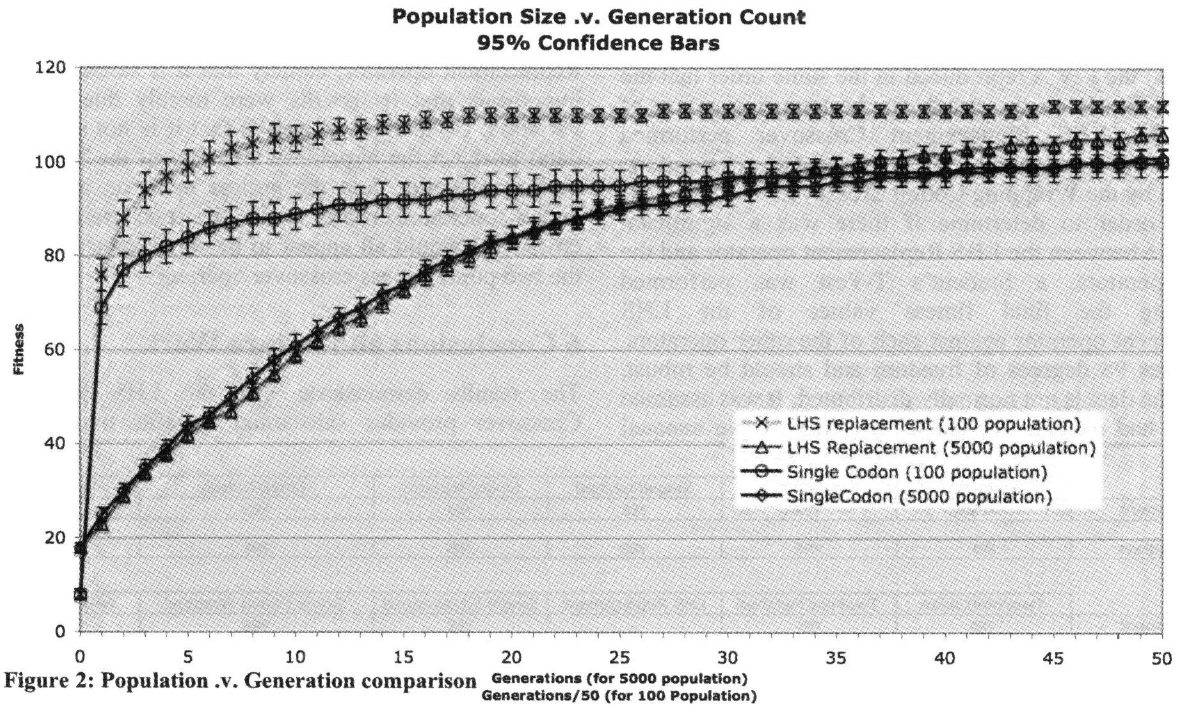

Figure 2: Population .v. Generation comparison

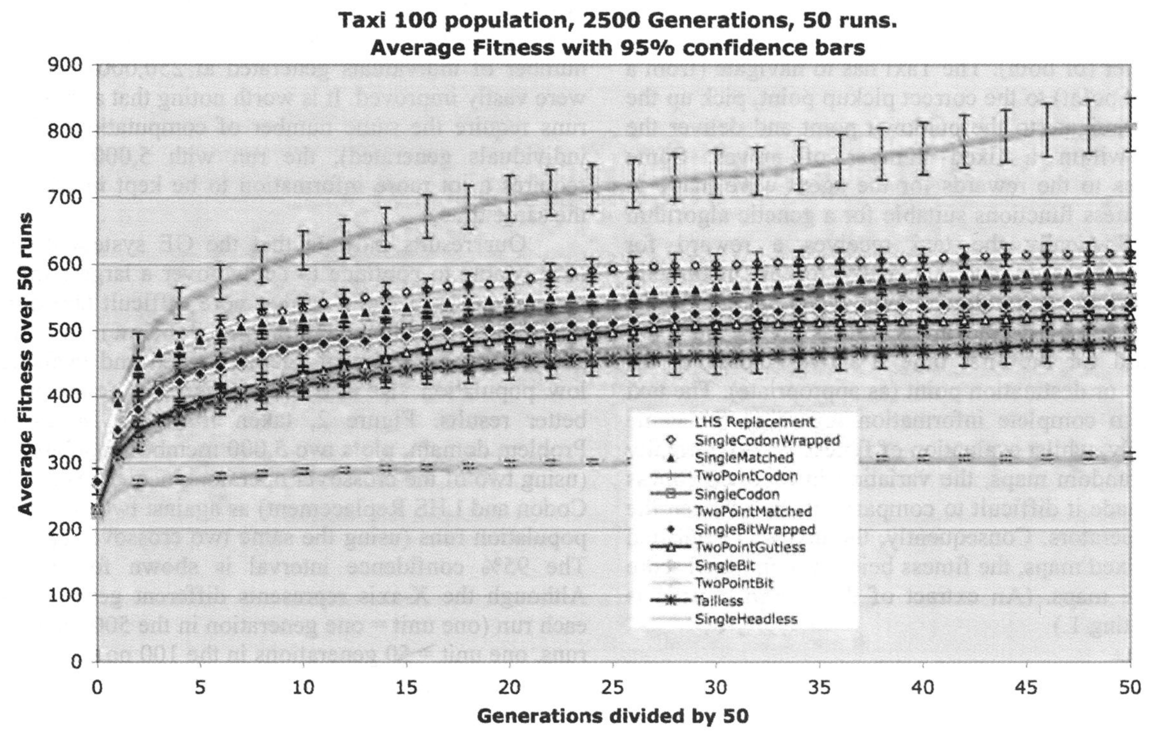

Figure 3: Comparison of Crossover Operators

## 5.2 Comparison of crossover operators

The crossover operators described in paragraph 3 were compared with each other across all problem domains, but most extensively on the minesweeper and taxi domains described in this paper where runs were made with population sizes of 5000 (50 generations) and 100 (2500 generations). Consistent results were achieved.

The results of the taxi domain for a 100 population and 2500 generations are reproduced in figure 3. Figure 3 shows the average of the highest fitness levels achieved by each crossover operator across the 50 runs and the 95% confidence interval levels in each case. To improve the readability of the graph (there being a large number of operators) the key is reproduced in the same order that the operators finish on the graph (highest scoring = top of key). The LHS Replacement Crossover performed significantly better than all the other crossover operators, followed by the Wrapping Codon Crossover.

In order to determine if there was a significant difference between the LHS Replacement operator and the other operators, a Student's T-Test was performed comparing the final fitness values of the LHS Replacement operator against each of the other operators. This gives 98 degrees of freedom and should be robust, even if the data is not normally distributed. It was assumed the runs had different variances; the two-sample unequal variance (heteroscedastic) test was used. The null-hypothesis was rejected at a level of 0.0001 (0.01%) (the null hypothesis being that the results of the runs are "drawn" from the same population). The reason such a low value is used is that there are over 66 operator to operator comparisons (if each operator was to be tested against each other operator), increasing the likelihood of, say, a 5% chance occurring. Table 1 shows the results of this comparison and a similar one for the Two Point Gutless operator (the best performing stochastic hill climbing operator), where rejections are labelled as "YES" i.e. there is a statistical difference or "NO" – there is no statistical difference.

The table confirms the benefit of the LHS Replacement operator; namely that it is safe to reject any hypothesis that its results were merely due to random variation. Of interest as well is that it is not safe (on this data) to reject the hypothesis that any of the bit operators performed better than the gutless operator, although the codon operators (other than the two point matched crossover) would all appear to be significantly better than the two point gutless crossover operator.

## 6 Conclusions and Future Work

The results demonstrate that the LHS Replacement Crossover provides substantial benefits over all other

| | SingleBit | SingleCodon | SingleMatched | SingleHeadless | SingleTailess | TwoPointBit |
|---|---|---|---|---|---|---|
| LHS Replacement | YES | YES | YES | YES | YES | YES |
| | | | | | | |
| Two Point Gutless | **NO** | YES | YES | YES | **NO** | **NO** |

| | TwoPointCodon | TwoPointMatched | LHS Replacement | Single Bit Wrapped | Single Codon Wrapped | TwoPointGutless |
|---|---|---|---|---|---|---|
| LHS Replacement | YES | YES | - | YES | YES | YES |
| | | | | | | |
| Two Point Gutless | YES | **NO** | YES | **NO** | YES | - |

Table 1 - Student's t-test applied for the LHS Replacement and the Single Point Tailless crossover operator as against each other operator.
YES means there is a significant difference (0.01% level)

operators tested including the normal single point operator (the Codon crossover), at least in the domains explored so far.

The retention of structural information from fit individuals and the swapping of complete expansions of non-terminals without disrupting the existing blocks of information was, in every case, a better method of traversing the search space to reach higher fitness levels.

Perhaps surprisingly, there appeared to be little difference between most of the "codon" boundary crossover operators, although all performed better than the stochastic hill climbers. As might be expected the bit crossovers were amongst the poorest performers. The impact of the bit crossovers is that 7 times out of 8, not only are the tail codons put into a new context, but also their values are changed. Across all the problem domains that were examined the bit operators performed substantially the same as the stochastic hill-climbing operators, indicating that they offer little benefit as crossover operators. Finally all the crossover operators performed better than the Headless Crossover, confirming that in each case the crossover (or hill climbing) is productive in improving the population. The performance of the Wrapping Codon Crossover confirms previous findings relating to its productivity e.g. [O'Neill and Ryan 2001; Keijzer 1991].

The work here demonstrates that, at least in the domains explored, the LHS Replacement Crossover is the most successful in exploring the search space. The problem domains we have used are quite distinct despite some superficial similarities (e.g. involving the control of an agent). What will be important is to expand the problem domains to see whether the same benefits are seen over a wider range of problem types. The hope is that the LHS Replacement Crossover is an efficient way of searching the genotype irrespective of the phenotype into which it is translated.

In addition there appears to be some evidence as to the ability of GE, with an appropriate crossover operator, to continue to explore effectively the fitness landscape even after the individuals have apparently converged to similar structures, partly by providing an effective control over bloat. The ability of the LHS Replacement Crossover to continue to evolve even when faced with more difficult problems needs to be further explored. If the LHS Replacement Crossover is able to evolve relatively small populations through a large number of generations to solve problems that currently require populations of tens of thousands of individuals then this will allow the genetic programming technique to be used in a far wider range of situations. The ability to search the problem space whilst keeping a smaller population current reduces the amount of information needed to be kept in main memory, even although the number of crossovers performed may be the same.

## References

[Dietterich 1998] Thomas G. Dietterich The MAXQ method for Hierarchical Reinforcement Learning Proc. 15th International Conf. on Machine Learning

[Jones 1995] T. C. Jones. Crossover, macromutation, and population-based search. In Larry J. Eshelman, editor, Proceedings of the Sixth International Conference on Genetic Algorithms, pages 73-80, San Francisco, 1995. Morgan Kaufmann.

[Keijzer 1991] Keijzer M., Ryan C., Cattolico M., and Babovic V. Ripple Crossover in Genetic Programming. In proceedings of EuroGP 2001.

[Keller 1996] Robert E. Keller, Wolfgang Banzhaf 1996. Genetic Programming using Genotype-Phenotype Mapping from Linear Genomes into Linear Phenotypes (1996) Genetic Programming 1996: Proceedings of the First Annual Conference

[Koza 1992] J.R. Koza Genetic Programming MIT Press/Bradford Books, Cambridge M.A., 1992

[Koza 1994] J.R. Koza Genetic Programming II: Automatic Discovery of Reusable Programs. MIT Press, Cambridge, Mass., 1994.

[Montana, 1994] Montana D, Stronly Typed Genetic Prgramming. Technical Report 7866, Bolt Beranek and Newman Inc.

[O'Neill and Ryan 1999] O'Neill M., Ryan C. Under the Hood of Grammatical Evolution. In Banzhaf, W., Daida, J., Eiben, A.E., Garzon, M.H., Honavar, V., Jakiela, M., & Smith, R.E. (eds.). GECCO-99: Proceedings of the Genetic and Evolutionary Computation Conference, July 13-17, 1999.

[O'Neill and Ryan 2000] Crossover in Grammatical Evolution: A Smooth Operator? In Proceedings of the Third European Workshop on Genetic Programming 2000, pages 149-162

[O'Neill *et al.*, 2001] O'Neill M., Ryan C., Keijzer M., Cattolico M., Crossover in Grammatical Evolution: The Search Continues. In Proceedings of EuroGP 2001.

[Ryan *et al.*, 1998] Ryan C., Collins J.J., O'Neill M. Grammatical Evolution: Evolving Programs for an Arbitrary Language. Lecture Notes in Computer Science 1391. First European Workshop on Genetic Programming 1998.

[Whigham, 1995] Whigham P.A, Grammatically-based Genetic Programming (1995) Proceedings of the Workshop on Genetic Programming: From Theory to Real-World Applications