Efficient Crossover in the GAuGE System

Miguel Nicolau and Conor Ryan

Biocomputing and Developmental Systems Group
Computer Science and Information Systems Department
University of Limerick, Ireland
{Miguel.Nicolau, Conor.Ryan}@ul.ie

Abstract. This paper presents a series of context-preserving crossover operators for the GAuGE system. These operators have been designed to respect the representation of genotype strings in GAuGE, thereby making sensible changes at the genotypic level. Results on a set of problems suggest that some of these operators can improve the maintenance and propagation of building blocks in GAuGE, as well as its scalability, and could be of use to other systems using structural evolving genomes.

1 Introduction

The GAuGE system (Genetic Algorithms using Grammatical Evolution) [12, 9] is a recently introduced position-independent genetic algorithm which, through a mapping process, maps a fixed length genotype string onto a fixed-length phenotype string, ensuring no under- or over-specification in the process. By encoding both the position and value of each phenotypic variable on genotype strings, GAuGE has the ability to structure these, to prioritise information, and to group together partial solutions, to minimise the chances of disrupting them.

Until now, a simple genetic algorithm has been used to generate binary strings that, through a mapping process, are interpreted as GAuGE strings, which in turn specify phenotype strings. A side effect of this is the *ripple effect*, by which a change of context for information exchanged between two individuals can lead to severe changes of its interpretation, at the phenotype level.

In this work, a set of crossover operators is presented, which are adapted to the GAuGE representation. These are context-preserving operators; by applying these, the context of information exchanged is the same, and therefore that information will keep its initial meaning. Experiments conducted suggest that, on the problems analysed, some of these operators scale better to problem difficulty, and are better able to discover, maintain and exchange partial solutions. These operators can therefore not only improve the performance of the GAuGE system, but also of other position-independent systems evolving the structure of genomes.

This paper is organised as follows. Section 2 introduces the GAuGE system, and its relation to Grammatical Evolution, and includes a review of previous work and an example of the mapping process employed. Section 3 introduces the newly designed crossover operators. Section 4 describes the experiments conducted and the results obtained, while Section 5 draws conclusions based on those results, and some lines of future research.

2 GAuGE

The GAuGE system uses many of the biologically inspired features present in Grammatical Evolution (GE) [11], the main ones being a *genotype to phenotype mapping* (GPM), *functional dependency* between genes, and *degenerate code*.

In GE, a GPM process is used to map variable-length binary strings onto syntactically-correct programs, through the use of a grammar. Each binary string is mapped onto an integer string, through a transcription process, and then, through a translation process, those integers are used to choose productions from the grammar, to create a program (phenotype). This process, based on the analogous process in molecular biology, provides a division between the search space (binary strings) and the solution space (evolved programs) [2].

In GAuGE, a similar process is employed. A population of (fixed length) binary strings is created in the genotypic space, and each of these is also transcribed onto an integer string. This integer string is then interpreted as a sequence of $(position, value)$ specifications that, through a mapping process, generate a fixed-length phenotype string, which is neither under nor over-specified.

In GE, the function of a gene can affect the function of the genes that follow it. Indeed, the rule from which a gene is used to choose a production depends on the mapping process up to that point; this means that if the functionality of a gene changes, the functionality of subsequent genes is likely to change as well.

This feature, called functional dependency, is present in GAuGE as well. Each position specification across a genotype string is dependent on previous specifications, in order to create a fully specified phenotype string.

Finally, the use of degenerate code plays an important role in GE: by using the *mod* operator to map an integer to a choice of productions from a grammar rule, neutral mutations can take place [7], creating a many-to-one mapping between the search and solution spaces, and introducing variety at the genotypic level.

In GAuGE, this feature is also present, as a direct result of the mapping process employed. It has also been shown that the explicit introduction of degeneracy can reduce structural bias at the genotypic level [9].

2.1 Previous Work

Previous work has used similar techniques as the ones employed in GAuGE. Some of Bagley's [1] simulations used an extended representation to encode both the position and the value of each allele. Some reordering operators were also designed [10], which combine inversion and crossover operators on extended representations. Later on, the so-called messy genetic algorithms [5] applied the principle of separating the *gene* and *locus* specifications with considerable success, and have since been followed by many competent GAs.

The Random Keys Genetic Algorithm [3] introduced a mapping process that ensures an error-free and fully specified sequence of ordinal numbers. More recently, Harik [6] applied the principles of functional dependency in the Linkage Learning Genetic Algorithm, in which the functionality of a gene is dependent on a chosen interpretation point, and the genes between that point and itself.

2.2 GAuGE Mapping

A full description and analysis of the mapping process in GAuGE has already been given elsewhere [9], including the description of a context-sensitive grammar which implements that process [13]. As a simple example, consider the case where one would wish to solve a four-variable problem ($l = 4$). The first step is to create a genotype population G, of N individuals. The length of each individual depends on the chosen values for the variables pfs (position field size) and vfs (value field size). The minimum value for pfs for this problem would be 2 bits, as that is the minimum number of bits required to encode 4 positions, whereas the minimum value for vfs depends on the range of the problem's variables: for a binary problem, 1 bit is sufficient.

Suppose the range of the variables is $0 \ldots 7$, and four bits are used to encode the value specifications ($vfs = 4$) (to introduce degeneracy, as three bits would be sufficient), and two bits are used to encode each position specification ($pfs = 2$). The length of each element G_i of the genotypic space G will then be a binary string of 24 bits ($l * (pfs + vfs)$).

Let us use the following individual as an example for the mapping process:

$$G_i = 011100111000111001100010$$

The mapping process then proceeds in creating a phenotype string P_i. The first mapping step consists in creating an integer string, using the pfs and vfs values:

$$X_i = \left((X_i^j, \tilde{X}_i^j)\right)_{0 \leq j \leq l-1} = \big((1,12),(3,8),(3,9),(2,2)\big)$$

This string is then evaluated as a sequence of four $(position, value)$ specification pairs. From this string, a *positions string* R_i and a *values string* $\tilde{R}_i$ can be created, which contain the *real* position and value specifications.

These are created as follows. We start by taking the first position specified, 1, and map it to the number of available positions in the phenotype string (i.e., 4), by using the *mod* operator: $1 \ mod \ 4 = 1$ (i.e. the second position on the phenotype string). We use a similar process to map the value specified, 12, to the *range* specified earlier, giving the *real* value specification $12 \ mod \ 8 = 5$:

$$R_i = (1, ?, ?, ?) \qquad \tilde{R}_i = (5, ?, ?, ?) \qquad P_i = (?, 5, ?, ?)$$

We then take the second pair, $(3, 8)$, and perform a similar mapping. As there are now only three positions remaining in the phenotype string, the position field is calculated by $3 \ mod \ 3 = 0$, that is, the first **available** position of that string; the value specification is calculated as before ($8 \ mod \ 8 = 0$), giving:

$$R_i = (1, 0, ?, ?) \qquad \tilde{R}_i = (5, 0, ?, ?) \qquad P_i = (0, 5, ?, ?)$$

The third pair, $(3, 9)$, is then processed in the same fashion. Its position specification is calculated by $3 \ mod \ 2 = 1$, that is, the second available position in the phenotype string. Since both positions 0 and 1 have already been taken, the second available position in the phenotype string is now position 3 (that is, the last position of that string). The value is calculated as before:

$$R_i = (1, 0, 3, ?) \qquad \tilde{R}_i = (5, 0, 1, ?) \qquad P_i = (0, 5, ?, 1)$$

Finally, the fourth pair is handled in the same fashion, giving the final *real* specification strings and corresponding phenotype string:

$$R_i = (1, 0, 3, 2) \qquad \tilde{R}_i = (5, 0, 1, 2) \qquad P_i = (0, 5, 2, 1)$$

So the phenotype string, $(0, 5, 2, 1)$, is ready for evaluation. Note the clear functional dependency between the position specifications, and the many-to-one mapping existing *both* in the position specifications and the value specifications.

3 Crossover Operators

Until now, GAuGE has been used as a mapping process, so any search engine can generate the genotypic binary strings. In this work, the advantages of adapting the search operators to the structure of a GAuGE string are explored; to this end, a series of context-preserving crossover operators has been created.

The reason behind the design of these operators is that the functional dependency feature, shown to exist across the position specifications [9], can result in a change of context for certain specified values. This has been shown to be a valid search technique for binary and combinatorial problems [12], but poses certain restrictions for problems where building block maintenance and exchange are critical to finding an optimal solution. By possibly changing the original position associated with a value, interesting sub-solutions found may be lost through the evolutionary search process, and the case may arise where crossover between two individuals, specifying the same value for a given position, may generate two offspring that do not specify the same value for that given position.

The approach taken in this work is to adapt the search algorithm that generates the binary strings to the structure of each individual, by choosing appropriate crossover points between two individuals, or appropriate sections to be swapped. These "context preserving" operators should be able to maintain and exchange partial solutions found during the evolutionary process, thus increasing the parallel exploration and exploitation of those building blocks.

3.1 Standard Crossover

This is the standard crossover operator, used with GAuGE in all experiments up to now. It is a one-point crossover, operating at the genotype level, but with crossover points limited to pair boundaries; that means that there are $l - 1$ possible crossover points between each individual (every $pfs + vfs$ bits).

For example, take the following two individuals, already expressed as a X_i string[1], randomly generated using a problem size of $l = 8$:

X_1=((0,a),(1,b),(3,c),(7,d),(7,e),(0,f),(3,g),(5,h))
X_2=((0,s),(4,t),(1,u),(5,v),(0,w),(2,x),(2,y),(4,z))

[1] With values $a \ldots h$ for the first individual, and $s \ldots z$ for the second individual.

These are the two corresponding *real* specifications strings for these individuals:

R_1=((0,a),(2,b),(5,c),(4,d),(7,e),(1,f),(6,g),(3,h))
R_2=((0,s),(5,t),(2,u),(1,v),(3,w),(7,x),(4,y),(6,z))

A crossover point is then chosen for these two individuals. If the chosen point is after the fourth pair, then the two offspring generated by that operation are:

X_3=((0,a),(1,b),(3,c),(7,d),(0,w),(2,x),(2,y),(4,z))
X_4=((0,s),(4,t),(1,u),(5,v),(7,e),(0,f),(3,g),(5,h))

These generate the following *real* specifications:

R_3=((0,a),(2,b),(5,c),(4,d),(1,w),(7,x),(3,y),(6,z))
R_4=((0,s),(5,t),(2,u),(1,v),(7,e),(3,f),(6,g),(4,h))

As can be seen, each child keeps the information from the first half of one parent, and uses the second half of the other parent to fill in the remaining unspecified positions. This has the side effect that the values specified in the second half of each parent do not necessarily stay in their original positions. In the example, the first parent specified that values (`e,f,g,h`) should be located at positions (`7,0,3,5`), respectively, which correspond to the *real* positions R_1=(`...,7,1,6,3`). However, when those specifications are interpreted within the context of the second child, the *desired* positions (`7,0,3,5`) are kept, but these now correspond to the *real* positions R_4=(`...,7,3,6,4`), as the *real* position `1` was already specified in that child's left side, creating a chain of changes.

This change (or adaptation) of the second half specifications to the new context upon which they are now interpreted is known as the *ripple effect* [11]. Although the way those specifications are interpreted can be quite different when in a new context, it is not random; indeed, the ordering relationship between those specifications is kept. In the example provided, this means that since the values (`e,f,g,h`) appeared in the order (`g,h,f,e`) in the phenotype string, then this ordering will be kept in the second child's phenotype.

3.2 Homologous Crossover

In the standard crossover, the degree of the ripple effect of changes occurring when a set of specifications is interpreted in a new context depends on how different that new context is. Furthermore, if the new context is equivalent (i.e., the same positions have been specified up to that point), then there is no ripple effect of changes, and all *desired* positions will encode the same *real* positions.

Based on this concept, the *homologous crossover* was designed to restrict the choice of crossover points to those points that share the same context (called *regions of similarity* [13]). For example, taking the previous two example parents, there is a region of similarity after the first pair, as both individuals have encoded the *real* position `0`. This means that, if this crossover point were chosen, there would be no change of context for the second halves from the parents, and therefore all position and value associations would be kept.

These two individuals also share a second region of similarity, after their third pair, as both have encoded positions `0`, `2` and `5` (although not in the same order). The homologous crossover will then pick one of these two similarity regions at random, and perform its crossover after the selected region. Considering the region up to the third pair were chosen, the resulting offspring would be:

X_3=((0,a),(1,b),(3,c),(5,v),(0,w),(2,x),(2,y),(4,z))
X_4=((0,s),(4,t),(1,u),(7,d),(7,e),(0,f),(3,g),(5,h))

These generate the following *real* specifications:

R_3=((0,a),(2,b),(5,c),(1,v),(3,w),(7,x),(4,y),(6,z))
R_4=((0,s),(5,t),(2,u),(4,d),(7,e),(1,f),(6,g),(3,h))

We can see that no change of context occurred, and therefore the second halves received from each parent keep their original (*position*,*value*) associations. Should there be no regions of similarity between the two parents, then the standard crossover operator is applied.

3.3 Pascal Crossover

This crossover was designed to deal more effectively with the uneven distribution of regions of similarity between individuals, which occurs due to the mapping process employed in GAuGE. Indeed, the higher the value of l, the more uneven the distribution of those regions, as a region of similarity is simply a combination of position specifications, regardless of their order. Taking $l = 8$, for example, the number of possible different contexts after each pair are:

$pair1 : \binom{8}{1} = 8$ $pair2 : \binom{8}{2} = 28$ $pair3 : \binom{8}{3} = 56$ $pair4 : \binom{8}{4} = 70$
$pair5 : \binom{8}{5} = 56$ $pair6 : \binom{8}{6} = 28$ $pair7 : \binom{8}{7} = 8$

These elements are quite easy to calculate using Pascal's Triangle, hence the name given to this operator. It works as follows: once one or more regions of similarity have been found, each is given a weight, according to their position; regions closer to the center of the genotype strings have a higher weight, as there are more possible different contexts in this area, and therefore fewer regions of similarity. A crossover point is then chosen, through a roulette-wheel choice between the detected regions. Taking the previous example, the first region of similarity will have the weight 8, and the second the weight 56. This means that the first pair has a $8/(8 + 56) = 8/64$ probability of being the chosen crossover point, whereas the third pair has a $56/64$ probability.

The idea behind this weighting is that, by preferring crossover points closer to the center, a more effective search for different structures is performed.

3.4 Structural Crossover

The main theory behind the way individuals are constructed in both GE [11] and GAuGE [13] is that, because of their functional dependency, individuals are firstly defined on their left side, and slowly extend their definition to the whole

genome. For that reason, the *structural crossover* was designed to give more importance to the left side of each individual, to the detriment of its right side.

This crossover works as follows: the structure of each parent is kept, that is, each offspring keeps the position specifications of each parent, but after a randomly chosen crossover point, the values specified are replaced by the corresponding values from the other parent. Taking the previous example, if the crossover point is after pair 4, then the generated offspring are as follows:

X_3=((0,a),(1,b),(3,c),(7,d),(7,x),(0,v),(3,z),(5,w))
X_4=((0,s),(4,t),(1,u),(5,v),(0,c),(2,e),(2,d),(4,g))

These generate the following *real* specifications:

R_3=((0,a),(2,b),(5,c),(4,d),(7,x),(1,v),(6,z),(3,w))
R_4=((0,s),(5,t),(2,u),(1,v),(3,c),(7,e),(4,d),(6,g))

As can be seen, the structure of the corresponding parents is kept, and data is taken from the second parent to replace the values associated with the positions specified after the crossover point. For example, the value `v`, defined in the second parent to be in the *real* position `1`, is now defined in the first child's 6^{th} pair, `(0,v)`, as this is the pair that maps to the *real* position `1` in this individual.

It can also be seen that some data occurring in the left side of the parents has been duplicated (the values `c`, `d` and `v`), whereas some of the data from their right side has been dropped (the values `f`, `h` and `y`).

3.5 Pure Crossover

This operator is the equivalent of a simple one-point crossover in a normal GA. It works as follows. The first offspring receives the first half of the first parent (up to a randomly chosen crossover point), and all positions after that point receive the values associated with those same positions from the second parent (as in structural crossover). The second offspring keeps the structure of the second parent, but receives from the first parent the values corresponding to the positions specified in the second half of that parent. Taking the previous example, if the crossover point is after pair 4, then the generated offspring are:

X_3=((0,a),(1,b),(3,c),(7,d),(7,x),(0,v),(3,z),(5,w))
X_4=((0,s),(4,t),(1,u),(5,f),(0,h),(2,e),(2,y),(4,g))

These generate the following *real* specifications:

R_3=((0,a),(2,b),(5,c),(4,d),(7,x),(1,v),(6,z),(3,w))
R_4=((0,s),(5,t),(2,u),(1,f),(3,h),(7,e),(4,y),(6,g))

As can be seen, the *real* positions (`1,3,7,6`), in the 4^{th}, 5^{th}, 6^{th} and 8^{th} pairs of the second offspring, receive the corresponding values (`f,h,e,g`) from the first parent, as these are the *real* positions specified in the second half of that parent.

Like structural crossover, in this operator both offspring retain the structure of both parents, but unlike that operator, all information specified is kept, that is, no data is dropped or duplicated.

4 Experiments and Results

In this work, we examined three problems. For each of these, a simple GA was compared to GAuGE, using each of the five crossover operators. The settings across all algorithms were: population size of 800 individuals, 100 generations[2], roulette-wheel selection, steady-state replacement, probability of crossover of .9, and probability of point mutation of .01. No degeneracy was used, so pfs and vfs values were set to the minimum values required by each problem.

4.1 Onemax

This is a standard problem in GA literature. It is defined by the formula:

$$f(x) = \sum_{i=0}^{l-1} x_i \qquad x_i \in \{0, 1\}$$

where l is the phenotype length, and x_i the allele at position i within that string (with positions ranging from 0 to $l-1$). The best individual is a binary string composed of all 1s, and the fitness contribution of each variable is the same.

This is a problem typically used to test the ability of a system to exchange partial solutions, and illustrates the parallel search of building blocks (of size 1) within in each individual (exploration), and their exchange between individuals (exploitation). It was used in this comparison to check if the GAuGE mapping process impairs its performance on problems where the simple GA excels.

Results This experiment was conducted to test the scalability of the algorithms; to that end, problem lengths of 128, 256 and 512 were chosen. The graph in Fig. 1 shows the results for $l = 256$, and plots the mean best fitness per generation, averaged over 100 random runs, with error bars showing the standard deviation. It shows that both GAuGE systems using the structural and pure crossovers find an optimal solution faster, whereas all the other approaches show a similar behaviour, regardless of their operators. The non-overlapping standard deviation error bars for those two operators with all other systems, from generation 15 all the way to generation 80, show the statistical significance of these results.

The results obtained (not reported) with the other lengths show that, with smaller values of l, all systems have a similar performance, but as the length of strings increases, both the structural and especially the pure crossover seem to scale better to the problem (with length 512, the pure crossover is significantly better than all other systems, including the structural crossover).

4.2 BinInt

The BinInt problem is an exponentially scaled problem, defined by the formula:

$$f(x) = \sum_{i=0}^{l-1} x_i 2^{l-i-1} \qquad x_i \in \{0, 1\}$$

[2] Except for the n-block problem, where 400 generations were used.

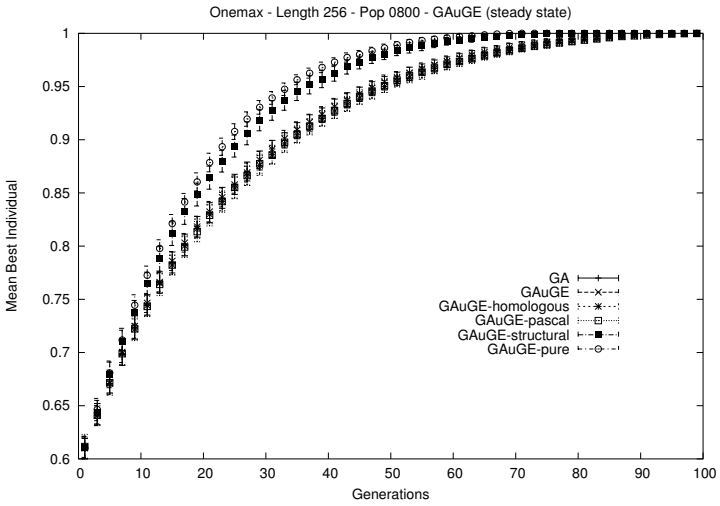

Fig. 1. Results obtained for the Onemax problem. The *x-axis* shows the generation number, and the *y-axis* the mean best individual (from 100 independent runs). The vertical error bars plot the standard deviation for all runs, for each system.

This problem transforms a binary string into its integer equivalent, and uses that value as a fitness reward; the best individual is a string composed of all 1s.

In this problem, the fitness contribution of each allele is higher than the combined fitness of all the following alleles (e.g., $f(0111) = 7$ but $f(1000) = 8$); this means that the salience of each allele decreases from left to right within the phenotype string. It was used in this comparison to test if the structural flexibility of the GAuGE system enhances its performance in this class of problems.

Results Fig. 2 shows the results for the BinInt problem, using a phenotype length of 64 bits. Error bars for standard deviation could not be plotted, as the *y-axis* is a logarithmic scale; figure analysis however showed that there is no significant difference in the performance of all systems, which is partly due to the high standard deviation in the simple GA results for the 100 runs.

The salience adaptation that occurs as a result of the functional dependency across the position specifications, which has been shown to guide the standard GAuGE approach to finding a suitable representation [8], explains the fact that there is no significant difference in performance between the original GAuGE approach and the GAuGE systems using both the structural and pure crossovers.

4.3 *n*-Block

This is a new problem, designed to test the ability of a system to find, maintain and propagate building blocks. These are deliberately hard to find, making this problem good for testing both the exploration and exploitation of knowledge.

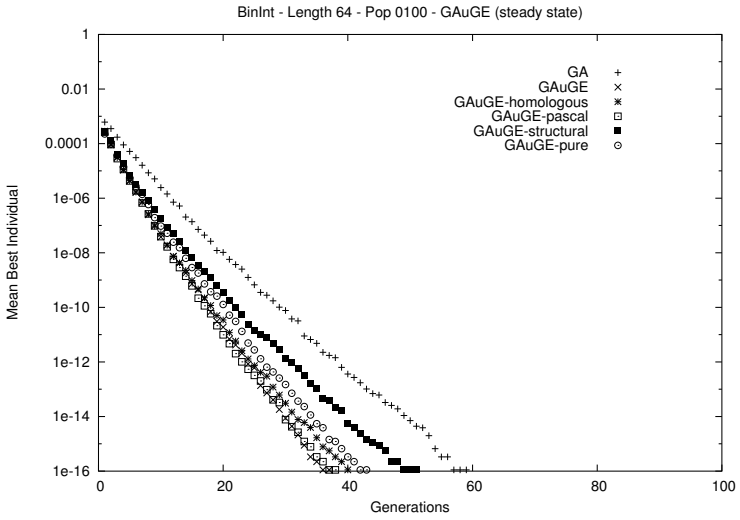

Fig. 2. Results obtained for the BinInt problem. The *x-axis* shows the generation number, and the *y-axis* shows the mean best individual (from 100 independent runs), on a logarithmic scale (lower is better; value plotted is $1 - fitness$).

The problem is based on two variables: l, the length of the phenotype strings, and n, the size of each building block, where l is divisible by n. The optimal solution is a string composed of l/n intertwined building blocks, with the first being composed of all 0s, the second of all 1s, the third of all 2s, and so on. The fitness of each building block is $1/(l/n)$ if it is correct, and 0 otherwise.

For example, with $l = 8$ and $n = 2$, the optimal solution is `01230123`, whereas with $l = 12$ and $n = 4$, the optimal solution is `012012012012`. For this last example, the string `012112012011` would receive a fitness score of $1/(12/4) = 1/3$, has there is only one fully specified building-block (`*1**1**1**1*`).

Results For this problem, a block size of $n = 2$ was used, with a phenotype string size of $l = 32$. Fig. 3 shows the results for this experiment; again error bars plot the standard deviation of all results. These show the pure crossover outperforming most other approaches, with a non-overlapping standard deviation compared to all systems except the structural crossover for most generations.

An analysis of the results obtained shows the difficulty of the problem, with most systems showing a large standard deviation across the 100 independent runs. It also shows that the ripple effect of the standard crossover from GAuGE impairs its performance in this problem, where it is important to keep previous information, but no saliency exists to guide the structuring of individuals.

Although the pure crossover's error bars are non-overlapping with the simple GA only from generation 220 onward, this is explained by the high variance in the simple GA's results, which shows its dependency in the initial population.

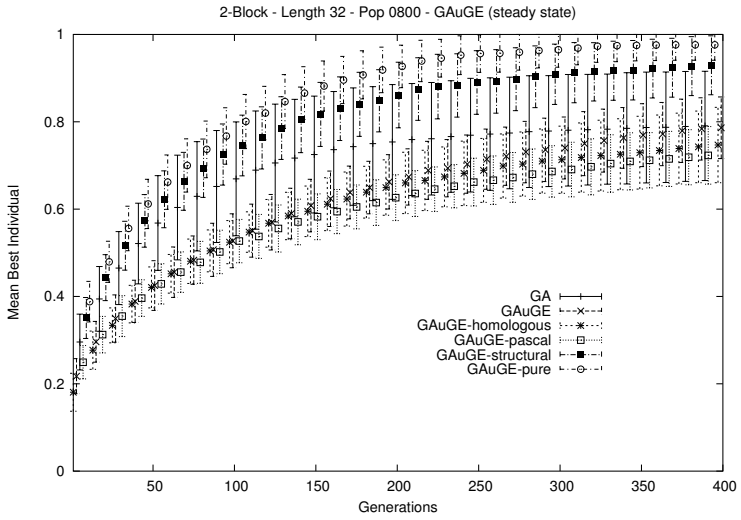

Fig. 3. Results obtained for the n-Block problem, with $n = 2$ and $l = 32$. The *x-axis* shows the generation number, and the *y-axis* the mean best individual (from 100 independent runs); error bars plot the standard deviation for each system. The top two plots belong to the systems using the pure and structural crossover, respectively.

5 Conclusions and Future Work

This paper has introduced a new set of crossover operators, adapted to the way in which GAuGE strings are represented. By keeping the context in which exchanged information is interpreted, some of these operators have been shown to improve the scaling and exchange of partial solutions, in the problems tested. These findings should be useful not only for the further development of the GAuGE system, but for other systems using structure evolving genomes as well.

The homologous and pascal crossovers did not perform better than the standard crossover. An analysis of their behaviour suggests this is due to the exponential number of ways to structure a GAuGE string. On problems with l large, the performance of these crossovers dropped when using smaller populations, as there is less variety in the population. With fewer regions of similarity, the standard crossover is applied quite often, while these operators are applied almost exclusively at the ends of the genotype strings. As the evolutionary process moves on, the standard crossover causes the representations of all genotype strings to converge, leading to the homologous and pascal crossovers being applied on converged sections of these strings, thereby exchanging the same information. The priorities used in the pascal crossover did not seem to improve on this.

The structural and pure crossovers showed a substantial increase in performance on these problems, when compared to the other approaches. The somewhat disappointing performance of the structural crossover can be partially ex-

plained by the lack of a context-preserving restructuring operator. This means that prioritising the left side of individuals is of no advantage to the system.

Across all problems, a high standard deviation has been observed for the simple GA, typically double that of the GAuGE systems. This seems to suggest that these are less affected by the random initialisation of the first generation, which in turn suggests more reliability for the GAuGE results.

Future work will see the introduction of context-preserving restructuring operators [4], which should further improve the performance of both the structural and pure crossovers. With these context-preserving operators, the selection pressure can also be lowered, as there is no need to filter the negative results of the ripple effect, so elitist approaches will be considered.

References

1. Bagley, J. D.: The behaviour of adaptive systems which employ genetic and correlation algorithms. Doctoral Dissertation, University of Michigan (1967)
2. Banzhaf, W.: Genotype-Phenotype-Mapping and Neutral Variation - A case study in Genetic Programming. In: Davidor et al., (eds.): Proceedings of the third conference on Parallel Problem Solving from Nature. Lecture Notes in Computer Science, Vol. 866. Springer-Verlag. (1994) 322-332
3. Bean, J.: Genetic Algorithms and Random Keys for Sequencing and Optimization. ORSA Journal on Computing, Vol. **6**, No. 2. (1994) 154-160
4. Chen, Y. and Goldberg, D. E.: An Analysis of a Reordering Operator with Tournament Selection on a GA-Hard Problem. In: Cantu-Paz et al., (eds.): Genetic and Evolutionary Computation - GECCO 2003. Springer. (July 2003) 825-836
5. Goldberg, D. E., Korb, B., and Deb, K.: Messy genetic algorithms: Motivation, analysis, and first results. Complex Systems, Vol. **3**. (1989) 493-530
6. Harik, G.: Learning Gene Linkage to Efficiently Solve Problems of Bounded Difficulty Using Genetic Algorithms. Doctoral Dissertation, University of Illinois (1997)
7. Kimura, M.: The Neutral Theory of Molecular Evolution. Cambridge University Press. (1983)
8. Nicolau, M., and Ryan, C.: How Functional Dependency Adapts to Salience Hierarchy in the GAuGE System. In: Ryan et al, (eds.): Proceedings of EuroGP-2003. Lecture Notes in Computer Science, Vol. 2610. Springer-Verlag. (2003) 153-163
9. Nicolau, M., Auger, A., and Ryan, C.: Functional Dependency and Degeneracy: Detailed Analysis of the GAuGE System. In: Liardet et al, (eds.): Proceedings of Évolution Artificielle 2003. Lecture Notes in Computer Science (to be published). Springer-Verlag. (2003)
10. Oliver, I. M., Smith, D. J., and Holland, J. R. C.: A Study of Permutation Crossover Operators on the Traveling Salesman Problem. In: Proceedings of the Second International Conference on Genetic Algorithms. (1987) 224-230
11. O'Neill, M. and Ryan, C.: Grammatical Evolution - Evolving programs in an arbitrary language. Kluwer Academic Publishers. (2003)
12. Ryan, C., Nicolau, M., and O'Neill, M.: Genetic Algorithms using Grammatical Evolution. In: Foster et al, (eds.): Proceedings of EuroGP-2002. Lecture Notes in Computer Science, Vol. 2278. Springer-Verlag. (2002) 278-287
13. Ryan, C., and Nicolau, M.: Doing Genetic Algorithms the Genetic Programming Way. In: Riolo, R., and Worzel, B. (eds.): Genetic Programming Theory and Practice. Kluwer Publishers, Boston, MA. (2003)