Crossover, Population Dynamics and Convergence 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 study of the effectiveness of a recently presented crossover operator for the GAuGE system. This crossover, unlike the traditional crossover employed previously, preserves the association of positions and values which exists in GAuGE genotype strings, and as such is more adequate for problems where the meaning of an allele is dependent on its placement in the phenotype string. Results obtained show that the new operator improves the performance of the GAuGE system on simple binary problems, both when position-sensitive data is manipulated and not.

1 Introduction

The GAuGE (Genetic Algorithms using Grammatical Evolution) [13, 9] system is a recent approach to position-independence in the field of genetic algorithms. Each individual in a GAuGE genotype string is composed of a sequence of position and value specifications which, through a genotype-to-phenotype mapping process similar to that of the GE (Grammatical Evolution) [12] system, ensures that each position in the resulting phenotype string is always specified, but only once. This mapping process produces a very compact and efficient representation, with neither under- nor over-specification of phenotypic information.

Until recently, a simple genetic algorithm [6] was used to select, combine and mutate genotype strings, which are then mapped using the GAuGE system to produce phenotypic information, to be evaluated. This approach, simple and elegant as it may be, presents some drawbacks. By allowing genetic operators which do not respect the representation of GAuGE genotype strings to manipulate those strings, associations between positions and values, which were previously discovered, may be lost in future generations.

To prevent this scenario from happening, a set of new crossover operators has been introduced [10], which, when applied, do not disrupt the associations between positions and values. Three different problem domains were tackled on that study, and some of the new operators showed a significant improvement of performance, when compared to the original GAuGE approach.

Of those operators, the *pure* crossover showed the most significant improvement in performance, and therefore a more detailed analysis of that operator is

done in the current paper. By restricting the problem domain to that of binary pattern matching, and by turning off the mutation operator on all systems, the advantages of the new crossover become clearer and easier to analyse. The results obtained show that, by respecting the underlying GAuGE representation of the genotype strings, the new crossover operator significantly improves the performance of the GAuGE system, even on irregular binary pattern problems, where using the traditional crossover resulted in a heavy loss of performance.

This paper is structured as follows. The next section presents the GAuGE system, its mapping process, and the crossover operators tested. Section 3 presents the experiments conducted and their results, while Section 4 analyses those results. Finally, Section 5 draws some conclusions on this work, and highlights future work directions.

2 GAuGE

The main principle behind the GAuGE system is the separate encoding of the position and value of each phenotypic variable. Its mapping process interprets each (fixed-length) genotype binary strings as a sequence of (*position*,*value*) pairs, which are used to build a fixed-length phenotype string. This mapping process ensures that each position of the phenotype string is specified exactly once, in much the same way that the GE system, an automatic programming system using grammars, ensures that each codon from the genotype strings chooses an existing production from a specified grammar.

Another feature that GAuGE shares with GE is functional dependency between genes, which is a direct result from the mapping process used. In GE, when a production has been chosen by a codon, the set of available choices available to the next codons changes; in other words, previous choices of grammar productions affect the available choices for the current codon. A similar effect occurs in GAuGE; in its mapping process, the set of free positions left in the phenotype string for the current gene to choose from depends on the choices of previous genes. In short, there is a functional dependency across the genotype strings in both GE and GAuGE, as the function of previous genes dictates the function of the following ones.

Finally, as GE uses the *mod* operator to map gene values to a choice of productions from a grammar rule, this creates a many-to-one mapping from the genotype strings onto the phenotype programs, leading to the occurrence of neutral mutations [7], which in turn introduce variety at the genotypic level. This also occurs in GAuGE, as each position specified at the genotypic level is mapped onto a set of available positions in the phenotype string. 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 approaches and techniques as the ones employed in GAuGE. Some of Bagley's [1] computer simulations used an extended string

representation to encode both the position and the value of each allele, and used an inversion operator to affect the ordering of genes. Holland [6] later presented modifications to the schema theorem, to include the approximate effect of the inversion operator. To tackle the problems associated with the combination of the inversion and crossover operators, these were later combined into a single operation, and a series of reordering operators were created [11].

The so-called messy genetic algorithms applied the principle of separating the *gene* and *locus* specifications with considerable success [4], and have since been followed by many competent GAs.

Work by Bean [2] with the Random Keys Genetic Algorithm (RKGA) hinted that a tight linkage between genes would result in both a smoother transition between parents and offspring when genetic operators are applied, and an error-free mapping to a sequence of ordinal numbers. More recently, Harik [5] has applied the principles of functional dependency in the Linkage Learning Genetic Algorithm (LLGA), in which a chromosome is expressed as a circular list of genes, with the functionality of a gene being 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 GAuGE mapping process can be found elsewhere [9]. As an example of this process, consider a simple problem composed of four phenotypic variables ($\ell = 4$), ranging between the values 0 and 7 ($range = 8$). The evolutionary algorithm maintains a genotype population G, of N individuals.

The length of each individual depends on a chosen position field size (pfs) and a value field size (vfs). As this problem is composed of four variables, $pfs = 2$ has been chosen, as that is the minimum number of bits required to encode four positions; for the value fields, a value of $vfs = 4$ has been chosen, to introduce degeneracy in the coding of values (the minimum number of bits required for the range specified is three). The required length of each string G_i, of the genotypic space $\mathcal{G}$, is therefore $L = (pfs + vfs) \times \ell = (2 + 4) \times 4 = 24$.

For example, take the following individual as an example genotype string:

$$G_i = 000101111101111001010010$$

The mapping process will proceed to create a phenotype string P_i. It consists in four steps[1]:

$$\Phi : G \xrightarrow{\Phi_1} X \xrightarrow{\Phi_2} D \xrightarrow{\Phi_3} R \xrightarrow{\Phi_4} P$$

The first mapping process (Φ_1) consists in creating an integer string, using the chosen pfs and vfs values:

$$X_i = \big((X_i^j, \tilde{X}_i^j)\big)_{0 \le j \le \ell - 1} = \big((0,5),(3,13),(3,9),(1,2)\big)$$

[1] In the actual implementation of GAuGE, some of these steps can be reduced.

The second mapping process (Φ_2) consists in interpreting this string as a sequence of four (*position*,*value*) pairs, to create a string of *desired* positions D_i and a string of *desired* values $\tilde{D}_i$.

These are created by mapping each position field onto the number of positions left in the phenotype string. For the first position field, $X_i^0 = 0$, the desired position specified is calculated by $(X_i^0 \ mod \ \ell) = (0 \ mod \ 4) = 0$, as at this stage no positions have been specified yet. The value field is calculated using the range of phenotypic values, giving $(\tilde{X}_i^0 \ mod \ range) = (0 \ mod \ 8) = 5$.

The second set of specifications is calculated in a similar way. For the position field, the desired position specified is calculated by $(X_i^1 \ mod \ (\ell - 1)) = (3 \ mod \ 3) = 0$, as only three positions remain unspecified in the phenotype string. The value field is calculated as before, giving $(\tilde{X}_i^1 \ mod \ range) = (13 \ mod \ 8) = 6$.

After processing all four pairs, the string of desired specifications are:

$$D_i = (0, 0, 1, 0) \qquad \tilde{D}_i = (5, 6, 1, 2)$$

At this stage, it can be seen that there are some conflicts in the position specifications (position 0 is specified three times, and positions 2 and 3 are still unspecified). The third mapping process (Φ_3) consists in removing these conflicts, creating a string of *real* positions R_i and a string of *real* values $\tilde{R}_i$.

These are created as follows. The first position specified, 0, is kept, as there are no conflicts at this stage, so $R_i^0 = 0$ (i.e. the first position on the phenotype string). The desired value specified, 5, is mapped to the range of the first phenotypic variable; as all variables share the same range in this problem, the *real* value specification is the same as before, $(5 \ mod \ 8) = 5$. An X sign is used to signal positions already taken in the phenotype string:

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

We then take the second desired position, 0, and perform a similar mapping. As the value specified is 0, it is interpreted as being the first **available** position of the phenotype string; as the position 0 has already been taken, the first available position is 1. The value specification is calculated as before, giving:

$$R_i = (0, 1, ?, ?) \qquad \tilde{R}_i = (5, 6, ?, ?) \qquad P_i = (X, X, ?, ?)$$

The third set of specifications is calculated in the same fashion. Its position specification is calculated by $(1 \ mod \ 2) = 1$, that is, the second available position in the phenotype string, while the value specification remains unchanged, giving:

$$R_i = (0, 1, 3, ?) \qquad \tilde{R}_i = (5, 6, 1, ?) \qquad P_i = (X, X, ?, X)$$

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

$$R_i = (0, 1, 3, 2) \qquad \tilde{R}_i = (5, 6, 1, 2) \qquad P_i = (X, X, X, X)$$

The fourth and final mapping step (Φ_4) simply consists in interpreting these specifications, creating a phenotype string by using the formula:

$$P_i^{R_i^j} = \tilde{R}_i^j \tag{1}$$

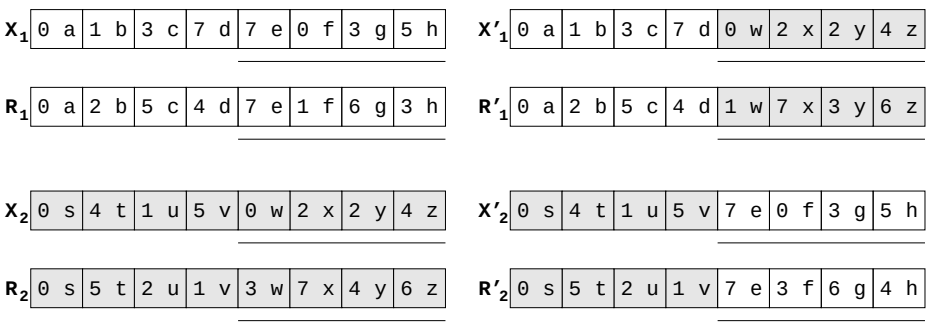

Fig. 1. Standard crossover operator for the GAuGE system. Two individuals, X_1 and X_2 exchange information after the fourth pair, generating the offspring X'_1 and X'_2.

In other words, through a permutation defined by R_i, the elements of $\tilde{R}_i$ are placed in their final positions. The phenotype string, ready for evaluation, is:

$$P_i = (5, 6, 2, 1)$$

2.3 Crossover Operators

Standard Crossover. This crossover operator has been 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 $\ell - 1$ possible crossover points between each individual (every $pfs+vfs$ bits).

An example of how this operator works is shown in Figure 1. Two individuals, randomly generated using a problem of size $\ell = 8$, are shown, already expressed as X_i strings and their corresponding R_i strings[2]. By choosing to crossover these individuals after the fourth pair, two offspring are generated, X'_1 and X'_2.

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, they now correspond to the *real* positions R'_2=(`...,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* [12]. 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

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

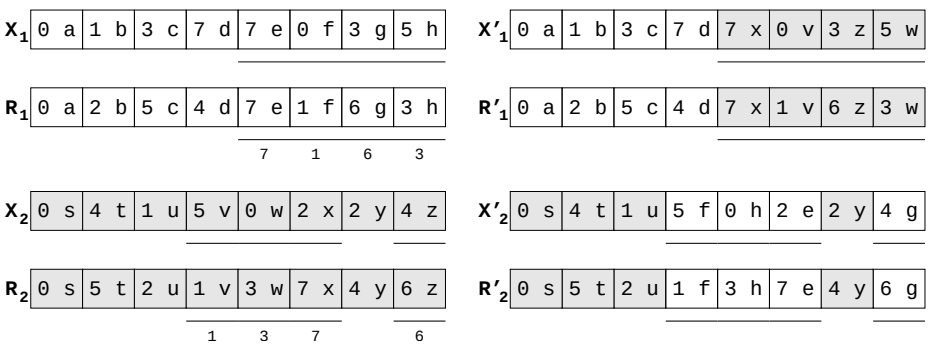

Fig. 2. Pure crossover for the GAuGE system. A crossover point is chosen on the first parent, and the corresponding value specifications from the second parent are used to create the first offspring; the complementary operation is used to create the second offspring. Both offspring keep the structure of their corresponding parent.

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.

Pure Crossover This is a new crossover operator, designed to respect the (*position*,*value*) associations of GAuGE strings. It works by maintaining the structure of each parent on the offspring strings, but exchanging value specifications, corresponding to the positions specified after the crossover point.

An example of how this operator works is shown in Figure 2. The first offspring (X'_1) keeps the position specifications of the first parent (X_1), and the value specifications of the first half of that parent, up to the crossover point. After that point, the values specified by the second parent, corresponding to the same *real* positions, are used instead.

The second offspring (X'_2) is produced in a similar fashion. It keeps the position specifications of the second parent (X_2), and the value specifications *which are not required by the first offspring*; all other value specifications are taken from the first parent, corresponding to the same positions.

In the example provided, it can be seen that 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.

3 Experiments and Results

To test how effectively the new operator maintains the association between positions and values, a set of four binary problems was used. These problems share the common feature that the fitness contribution of each variable is the same,

Table 1. Experimental setup, used on all experiments

Problem length (ℓ):	128
Population size (N):	100
Number of generations:	100
Position field size (pfs):	7 bits
Value field size (vfs):	1 bit
Crossover probability:	1.0
Position field mutation probability:	0.0
Value field mutation probability:	0.0

regardless of its location (i.e. no salience). In the two first problems, onemax and zeromax, all alleles have the same value on the global optimum, regardless of their position, whereas on the other problems, the association between position and value is important and must be kept.

These problems were used as it is easy to demonstrate and visualise the effects on the population of the genetic operators used. By understanding how these operators affect population dynamics and performance in these simple binary problem domains, important information is gathered which can be used on the design and refinement of these operators.

The GAuGE system using the two crossover operators was compared to a simple GA. The experimental setup used on all experiments is shown in Table 1. In these experiments, the mutation operator was turned off, to test how effectively the crossover operators combine the information that is currently on the population.

3.1 Onemax

The onemax problem is a well-known problem in the GA community. It is defined by the following formula:

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

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

This problem has been used before to demonstrate that the GAuGE mapping process does not impair its performance on simple binary maximisation problems.

3.2 Zeromax

This problem is the opposite of the onemax problem. It is defined by the formula:

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

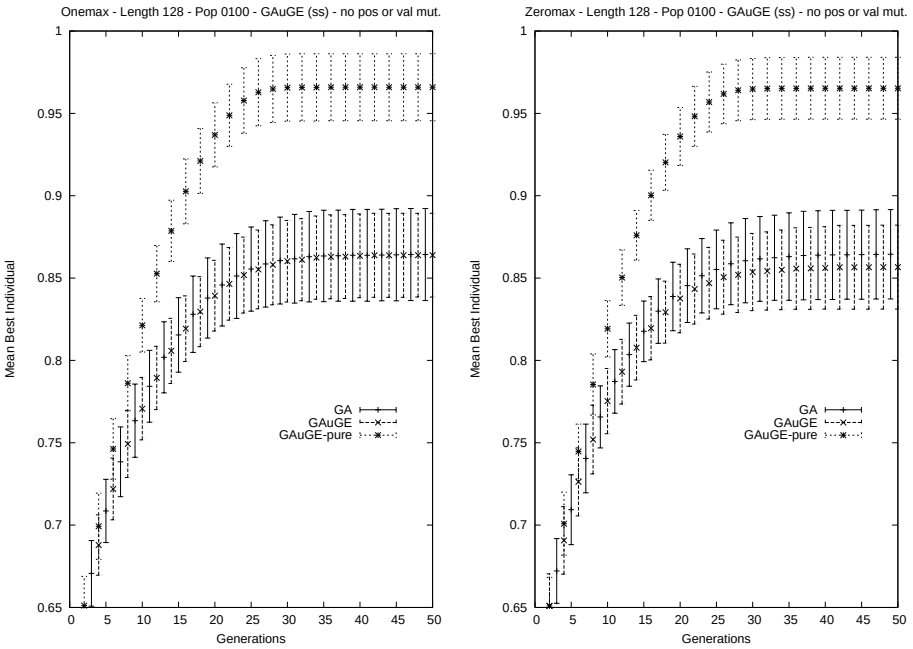

Fig. 3. Results obtained for the onemax (left) and zeromax (right) problems with length 128. The *x-axis* shows the generation number, and the *y-axis* the mean best individual (from 100 independent runs). The vertical error bars plot the standard deviation for all runs, for each system.

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

This experiment was chosen, along with the onemax problem, to test the performance of the systems being compared under easy maximisation problems, where the location of an allele in the genotype string is unimportant. The results obtain for the onemax and zeromax problems are shown in Figure 3.

3.3 Zero-Onemax

This is yet another binary matching problem. It is defined by the formula:

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

where ℓ is the phenotype length, and x_i the allele at position i within that string (with positions ranging from 0 to $\ell - 1$). For this problem, the best individual is a binary string composed of 0s and 1s constantly alternated.

This problem was used as the location of an allele on the phenotype string is important. An operator which does not respect the association between alleles and their positions should perform badly on this kind of problem.

3.4 Binary Matching

This is the last problem analysed. It is defined by the formula:

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

where ℓ is the phenotype length, x_i is the allele at position i within that string (with positions ranging from 0 to $\ell - 1$), and y_i is the element at position i of a randomly created binary string. The best individual in this case is a binary string equal to the randomly created one. The following string was used:

00111011000011011011110101101011001100001000011100010110001100101001011101011011000100001100010111000111110001000100110010101011

As with the zero-onemax problem, in this problem the associations between positions and values are important, as a value 1 will only contribute to the fitness of an individual if placed on the second half of its phenotype string. The results obtain for this and the zero-onemax problems are shown in Figure 4.

4 Analysis

The results obtained for the first two problems show both the simple GA and the original GAuGE system have a similar performance, whereas GAuGE with the new crossover operator shows a significantly better performance than those two systems. All three systems have a similar behaviour for these two problems, which was to be expected.

For the remaining problems, however, there is a significant drop in performance for the original GAuGE system, whereas GAuGE with the new crossover and the simple GA have a similar performance as on the previous problems (Figure 4). The reason for this difference in performance between the standard and pure crossover operators is explained by representation convergence: while the standard crossover requires the population to converge in its representation to keep the association between positions and values [14], the pure crossover always respects those associations even when individuals do not share the same representation. As a result, from the first generation, the pure crossover exchanges valuable information between individuals, while the population slowly converges in its representation [8] (due to selection pressure); the standard crossover is however actively working in the population to achieve a convergence in the representation of individuals, and only then is it capable of exchanging sensible information, as at that stage a crossover operation between two individuals will not break position-value associations.

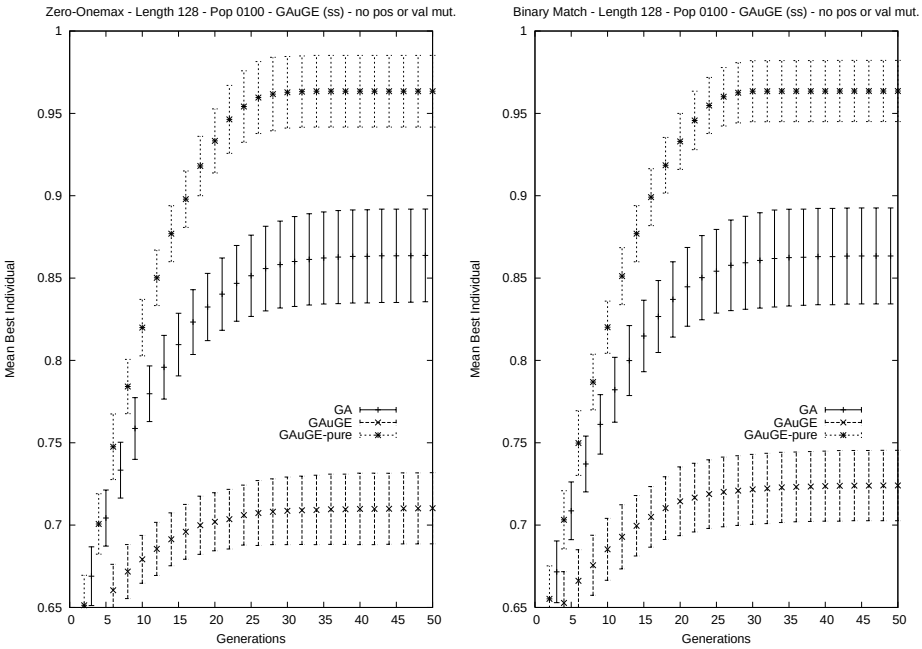

Fig. 4. Results obtained for the zero-onemax and binary-matching problems with length 128. The *x-axis* shows the generation number, and the *y-axis* the mean best individual (from 100 independent runs). The vertical error bars plot the standard deviation for all runs, for each system.

This behaviour can be seen by observing Figures 5 and 6, for the onemax and zero-onemax experiments (respectively). These figures plot a typical run of the GAuGE system, with both crossovers. Each square represents the state of the population at the specified generation; each horizontal line inside the square represents an individual (run settings are the same as on the previous experiments). The information plotted for each individual is its representation, that is, the R_i string: position 0 is represented by a black dot, position 127 is represented by a white dot, and all interim positions are represented with grey levels in between. Individuals sharing the same representation cause the vertical lines observed. It can be seen in both figures that representation converges faster with the standard crossover, as this is required for sensible information to be exchanged between individuals; on the contrary, the representation is slower to converge with the pure crossover[3], and even at generation 50, when all individuals share the same fitness, different representations co-exist in the population.

[3] It does converge, although at a slower rate, as a result of selection pressure.

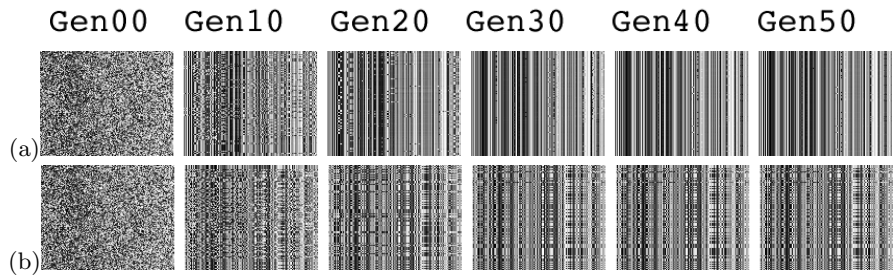

Fig. 5. Population representation convergence for the GAuGE system, with standard crossover (a) and with pure crossover (b), for the onemax problem. Each square represents the state of the population at generation 0, 10, and so on; a black dot represents position 0, and a white dot represents position 127, with grey levels for all interim positions. Each horizontal line in a square represents an individual.

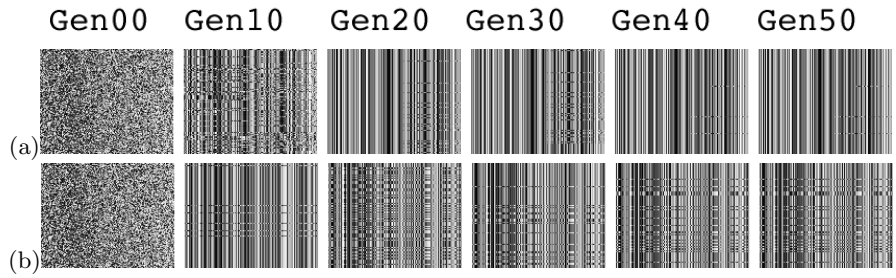

Fig. 6. Population representation convergence for the GAuGE system, with standard crossover (a) and with pure crossover (b), for the zero-onemax problem. Each square represents the state of the population at generation 0, 10, and so on; a black dot represents position 0, and a white dot represents position 127, with grey levels for all interim positions. Each horizontal line in a square represents an individual.

5 Conclusions and Future Work

The performance of the pure crossover for the GAuGE system has been analysed in this paper. By adapting to the representation of GAuGE genotype strings, this crossover does not depend on representation convergence to exchange context-sensitive data, and as such is fast an effective in combining information present in the population.

Future work will continue the analysis and possible enhancement to the presented operator, and the design of a reordering genetic operator [3], to maintain diversity at representation level, but without breaking the association between values and their positions. This operator should also allow for the discovery and maintenance of linkages between genotypic locations.

Acknowledgments

The authors would like to thank an anonymous reviewer of a previous paper, whose comments and suggestions lead to the investigation presented in this work.

References

1. Bagley, J. D.: The behaviour of adaptive systems which employ genetic and correlation algorithms. Doctoral Dissertation, University of Michigan (1967)
2. Bean, J.: Genetic Algorithms and Random Keys for Sequencing and Optimization. ORSA Journal on Computing, Vol. **6**, No. 2. (1994) 154-160
3. 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
4. Goldberg, D. E., Korb, B., and Deb, K.: Messy genetic algorithms: Motivation, analysis, and first results. Complex Systems, Vol. **3**. (1989) 493-530
5. Harik, G.: Learning Gene Linkage to Efficiently Solve Problems of Bounded Difficulty Using Genetic Algorithms. Doctoral Dissertation, University of Illinois (1997)
6. Holland, J. H.: Adaptation in Natural and Artificial Systems. Ann Arbor, MI: University of Michigan Press. (1975)
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. Nicolau, M. and Ryan, C.: Efficient Crossover in the GAuGE system. In: Keijzer et al, (eds.): Proceedings of EuroGP-2004. Lecture Notes in Computer Science (to be published). Springer-Verlag. (2004)
11. 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
12. O'Neill, M. and Ryan, C.: Grammatical Evolution - Evolving programs in an arbitrary language. Kluwer Academic Publishers. (2003)
13. 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
14. 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) 189-204