Genetic Operators and Sequencing in the GAuGE System

Miguel Nicolau and Conor Ryan

Abstract— **This paper investigates the effects of the mapping process employed by the GAuGE system on standard genetic operators. It is shown that the application of that mapping process transforms these operators into suitable sequencing searching tools. A practical application is analysed, and its results compared with a standard genetic algorithm, using the same operators. Results and analysis highlight the suitability of GAuGE and its operators, for this class of problems.**

I. INTRODUCTION

The GAuGE system (Genetic Algorithms using Grammatical Evolution) [1] uses a position-independent representation, which allows it to specify both the position and the value of each phenotypic variable. It employs a genotype-to-phenotype mapping process which ensures all variables are specified once and only once: in other words, no under- nor over-specification ever occurs in the system.

This elegant mapping process has a cost. In order to keep the association between the position and the value of each variable, it has been shown [2] that representation convergence must occur throughout the population. While not damaging on regular binary problems, this disassociation can seriously impair the performance of the system on other problems, and to that end, a series of genetic operators have been developed, which respect those associations [3], [4].

Some problems, however, do not require a strict respect of that association. Job-shop scheduling problems, for example, require a fixed set of values to be optimally ordered; the objective of the system is to search for the optimal representation that solves the problem. In this class of problems, each variable encodes the same type of choice (e.g. a resource), and the values of variables can (and must) be exchanged between them, to find optimal sequences.

An interesting problem domain is where the set of values to be ordered is not fixed a-priori, and neither is it an ordinal sequence; the objective of the system in this case is not only to find the correct sequence of variables, but also the correct set of values for all variables.

GAuGE has been applied to a version of the Mastermind game [1], which exhibits these characteristics, and the results obtained were encouraging. Recently, it was also applied to Sudoku puzzles [5], and again its success suggests that the system is suitable for this class of problems.

In this paper, the effect of standard genetic operators on the GAuGE system is analysed; this analysis shows how their combination is suitable for the mentioned class of problems. A direct comparison with a standard genetic algorithm [6] is

Miguel Nicolau and Conor Ryan are with the Biocomputing and Developmental Systems group, Department of Computer Science and Information Systems, University of Limerick, Ireland (email: Miguel.Nicolau@ul.ie, Conor.Ryan@ul.ie).

performed, for the Sudoku problem, and the results obtained and their analysis highlight the validity of this approach.

This paper is structured as follows. Section II introduces the GAuGE system and its mapping process, while Section III analyses the effect of standard genetic operators on the GAuGE system. Section IV then presents the Sudoku problem, and Section V the experimental setup used. Finally, Section VI analyses the results obtained, and Section VII highlights future work directions.

II. THE GAUGE SYSTEM

The GAuGE system works as a mapping process applied to binary genotype strings, which are evolved using a standard genetic algorithm [6]. It interprets those strings as a sequence of position and value specifications, and constructs a phenotype string, which is then evaluated.

A. Background

Many systems have been developed using similar techniques to the ones employed in GAuGE. Bagley [7] used fixed-length strings of $(position, value)$ specifications, and an inversion operator to move those specifications around in the chromosome strings; both Frantz [8] and Holland [9] extended some of that work, and similar operators were later designed, with the same purpose [10], [11], [12].

The messy genetic algorithms [13], [14] also use a separate encoding of the position and the value of each phenotypic variable. They deal with the problem of over-specification on a "first come, first served" basis, while under-specification is dealt with the use of an evolved template.

More recently, Harik [15] introduced the Linkage Learning Genetic Algorithm, 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.

Finally, the Grammatical Evolution system (GE) [16] uses a standard genetic algorithm (or any other search algorithm) to evolve binary strings, and employs a grammar-based mapping process to create phenotype strings. The design of GAuGE was based on many of the biologically inspired features present in GE, such as a genotype-to-phenotype mapping process, functional dependency between genes, and the use of degeneracy (i.e. a many-to-one correspondence between genotype and phenotype).

B. Mapping Process

Consider a problem composed of four phenotypic variables ($\ell = 4$), ranging between the values 0 and 7; the objective of the system is to find the set of values that solves the problem, and their optimal order.

The underlying search algorithm maintains a population of genotype strings. The length of each string depends on a chosen position field size (pfs) and a value field size (vfs). As there are four variables, a value of $pfs = 2$ 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 (the minimum required is $vfs = 3$). The required length for each binary string is therefore $L = (pfs + vfs) \times \ell = (2 + 4) \times 4 = 24$.

For example, take the following example genotype string:

$$G = 001001101101110100010010$$

The mapping process will proceed to create a phenotype string P. It consists of four steps:

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

The first step (Φ_1) creates an integer string X, using the chosen pfs and vfs values:

$$X = \big((X_i, \tilde{X}_i)\big) = \big((0, 9), (2, 13), (3, 4), (1, 2)\big)$$

The next step (Φ_2) interprets X as a sequence of (*position*,*value*) pairs, to create a string of *desired* specifications. This string is created by mapping each position field onto the number of positions left in the phenotype string. For the first position field, $X_0 = 0$, the desired position specified is calculated by $D_0 = (X_0 \ mod \ \ell) = (0 \ mod \ 4) = 0$, as at this stage no positions have been specified yet. The value field is calculated using the range of phenotypic values, giving $\tilde{D}_0 = (\tilde{X}_0 \ mod \ range) = (9 \ mod \ 8) = 1$.

The second set of specifications is calculated in a similar way: the desired position specified is calculated by $D_1 = (X_1 \ mod \ (\ell - 1)) = (2 \ mod \ 3) = 2$, as only three positions remain unspecified in the phenotype string, and the value field becomes $\tilde{D}_1 = (\tilde{X}_1 \ mod \ range) = (13 \ mod \ 8) = 6$.

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

$$D = \big((D_i, \tilde{D}_i)\big) = \big((0, 1), (2, 6), (1, 4), (0, 2)\big)$$

At this stage, it can be seen that there are some conflicts in the position specifications (position 0 is specified twice, and position 3 is still unspecified). The third mapping step (Φ_3) consists in removing these conflicts, creating a string of *real* specifications R.

This string is created as follows. The first position specified, 0, is kept, as there are no conflicts at this stage, so $R_0 = 0$ (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, it is not affected:

$$R = \big((R_i, \tilde{R}_i)\big) = \big((0, 5), (?, ?), (?, ?), (?, ?)\big)$$

The second pair undergoes a similar mapping. The position specified, 2, is interpreted as being the third **available** position of the phenotype string; as the position 0 has already been taken, the third available position is 3. The value specification remains the same, giving:

$$R = \big((0, 5), (3, 6), (?, ?), (?, ?)\big)$$

R_2 is calculated in the same way: the position specified (1) is interpreted as the second position available on the phenotype string, which is position 2:

$$R = \big((0, 5), (3, 6), (2, 4), (?, ?)\big)$$

Finally, the fourth pair is handled in the same fashion, giving the final *real* specifications string:

$$R = \big((0, 5), (3, 6), (2, 4), (1, 2)\big)$$

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

$$P_{R_i} = \tilde{R}_i \qquad (1)$$

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:

$$P = (5, 2, 4, 6)$$

III. GENETIC OPERATORS AND THE GAuGE SYSTEM

A. Crossover

The GAuGE system uses a standard 1-point crossover operator. However, due to the mapping process employed, the phenotypic effect of this operator can be quite drastic. For example, take the following pair of individuals, already expressed as a sequence of ($position, value$) pairs:

$$G1 = (0, A), (3, B), (2, C), (3, D)$$
$$G2 = (3, F), (0, G), (3, C), (1, E)$$

Their corresponding phenotype strings are:

$$P1 = A, B, C, D$$
$$P2 = G, E, C, F$$

If these two parents are elected for crossover, and the chosen crossover point is after the second codon, then the generated offspring will be:

$$G3 = (0, A), (3, B), (3, C), (1, E)$$
$$G4 = (3, F), (0, G), (2, C), (3, D)$$

With the corresponding phenotype strings being:

$$P3 = A, B, E, C$$
$$P4 = G, C, D, F$$

A close inspection of the generated offspring shows that the third phenotypic variable, which both parents specified as containing the value C, now contains the values E and D, respectively. In other words, even though both parents specified C as a potentially useful value for the third variable, none of the offspring respected that.

Crucially though, the *ordering* of the second half of each parent genotype string was maintained. The first parent specified that, at the phenotypic level, the value C should appear before the value D, while the second parent specified that C should appear after E. By inspecting the generated offspring,

one can verify that this ordering has been maintained, even though the relative positions of those values within the phenotype string have changed. This is because the desired positions for those variables are always the same; in the example, for the first parent:

$$D_3 = 2 \bmod 2 = 0 \qquad D_4 = 3 \bmod 1 = 0$$

which translates to $\tilde{D}_3 = C$ appearing before $\tilde{D}_4 = D$, in the phenotype string. The same is true for the second parent:

$$D_3 = 3 \bmod 2 = 1 \qquad D_4 = 2 \bmod 1 = 0$$

which translates to $\tilde{D}_3 = C$ appearing after $\tilde{D}_4 = E$.

These *desired* positions are calculated in the same fashion when the two sections are exchanged. So although their *real* specifications change, in order to adapt to the new context in which they are being inserted, their relative ordering is kept.

The end result of the crossover operator is similar to that of the order crossover [10], a two-point crossover where the section between cut-points is copied from one parent, and the rest of the offspring is reconstructed with the contents of the second parent, in the order that they appear. However, unlike the order operator, the GAuGE crossover is merely a single-point crossover at the genotypic level, thus being a faster, non-specific operator.

B. Mutation

The mutation operator used with GAuGE in this work is a 1-point, bit-wise mutation[1]. Similarly as with the crossover operator, a mutation event occurring in one of the position specifications can have a drastic effect on the generated phenotype. For example, take the following individual, already expressed as a sequence of specification pairs:

$$G_1 = (0, A), (3, B), (2, C), (3, D)$$

Its corresponding phenotype string is:

$$P_1 = A, B, C, D$$

If a mutation event occurs at the first position specification, transforming the above individual into the following:

$$G_2 = (3, A), (3, B), (2, C), (3, D),$$

the phenotypic effect would be drastic, as all variables will receive different values:

$$P_2 = B, C, D, A$$

However, the relative phenotypic ordering of all the specifications which were not modified by the mutation event is maintained. In other words, the value B still comes before the value C, which in turn still comes before D. The reason for this effect is the same as for the crossover operator; all variables with their *desired* position specifications unchanged keep their relative ordering.

The end result of this operator is similar to that of the point mutation of the Random Keys Genetic Algorithm

[1]This is for historical reasons, as Grammatical Evolution uses such an operator. Its use also comes from the fact that GAuGE can be applied as a simple mapping process on top of a bit-level genetic algorithm.

(RKGA) [17], in that only the mutated variables are moved, with all other values keeping their relative ordering; unlike the RKGA, however, GAuGE can handle problems where the variable set is not an ordinal set.

C. Suitability of Operators

These operators suggest that an interesting problem domain for GAuGE would be where not only the contents but also the ordering of variables are to be discovered. Recent results [5] showed the suitability of GAuGE to solve a harder version of the Sudoku puzzle. This work introduces a comparison with a standard genetic algorithm on this problem, and a deeper analysis of the effect of the genetic operators on the results obtained.

IV. SUDOKU

The Sudoku puzzle first originated in the United States in 1979, under the name *Number Place*, but it was in Japan that it gained popularity. In November 2004, it was first published in the British newspaper "The Times", and since then its popularity in the western world increased immensely, to the point where it is being dubbed as "the Rubik's cube of the 21st century" [18].

The standard puzzle is composed of a 3×3 grid of blocks, each composed of 3×3 cells, for a total of 81 cells (Fig. 1). The objective of the puzzle is to fill each cell with a number from 1 to 9, such that the following three conditions hold:

1) Each row contains the numbers 1 to 9 only once;
2) Each column contains the numbers 1 to 9 only once;
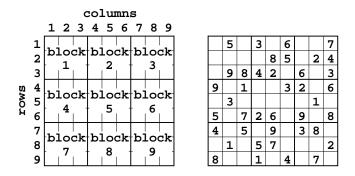3) Each block contains the numbers 1 to 9 only once.

Fig. 1. Example Sudoku boards. The referencing system used is shown on the left, and an example board in the right.

A set of numbers are initially loaded with each puzzle (called *givens*). A puzzle is called *well-formed* if only one solution exists, and it can be solved using logic (that is, no guessing is required to complete it). Fig. 1 shows an example of a well-formed Sudoku puzzle, with 40 givens.

A. Logical Operations

The problem of solving Sudoku puzzles on $n^2 \times n^2$ boards of $n \times n$ blocks is known to be NP-complete [19]; this gives an indication of why solving Sudoku puzzles can be difficult. However, due to the finite size of the puzzle, it can be solved by a deterministic finite automaton that knows the entire

game tree [18]. A different approach is to base each number-placing action purely on logic; this can only be applied to well-formed puzzles, but has the advantage of producing a list of logical actions, which can be reproduced by a human.

There are many such logical operations; below are some of the most common (and simpler) techniques:

1) Last Remaining: This is a simple logic operation, that can be applied to any kind of region (row, column or block). It simply checks if that region has already eight numbers placed, in which case it places the remaining one.

2) Slice and Dice: This is a combination of two operations, *slice* and *dice* [20], and can be applied when trying to place number n in block b. Slice looks for n in each row passing through b; if it contains n, then the three cells intersecting with b cannot contain n. Dice works with columns instead. If, after applying slice and dice, only one cell is available, then it must contain n. Fig. 2 shows an example.
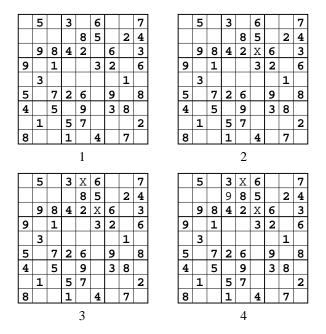

Fig. 2. Applying slice and dice to place number 9 in block 2. Starting with the left board, slice finds a 9 in the third row, so 9 cannot be placed anywhere else in that row (relevant cells are marked with an X in the second board). Dice finds a 9 in the fifth column, so 9 cannot be placed anywhere else in that column (third board). This results in only one cell being available to place 9 in block 2 (last board).

3) Column Fill: This technique tries to place number n in column c. It looks for n in all rows and blocks passing through c; if a row or block contain n, then the cell(s) corresponding to the intersection of that region and column c cannot contain n. If after checking all rows and blocks there is only one cell available in column c, then that cell must contain n. Fig. 3 shows an example.

4) Row Fill: This technique tries to place the number n in row r, and works in the same way as Column Fill, but going through all the columns instead.

5) Raising Numbers: This technique tries to place the number n in block b, by checking each empty cell in b to

Fig. 3. Applying Column Fill to place number 9 in column 6. By searching through all rows in the left board for the number 9, a set of cells can be marked as being unsuitable to receive that number (second board); then looking through blocks 2, 5 and 8 (the blocks that intersect column 6) another unsuitable cell is discovered (third board); finally, only one cell is available in column 6, so it must contain 9 (last board).

see if n is the only number that can be placed in that cell; Fig. 4 shows an example.

Fig. 4. Applying Raising Numbers to place number 4 in block 4. By analysing the left board, there are four empty cells in block 4, and the numbers 2, 4, 6 and 8 are missing. Through Slice and Dice operations, the set of possible numbers for each of the empty cells can be deduced: the leftmost cell can receive the numbers $\{2, 6\}$, the centre top cell can receive $\{4, 6, 8\}$, the rightmost one $\{2, 4, 6\}$, and the bottom centre $\{4\}$. Since this last cell can only receive the number 4, it is placed there (right board).

Sometimes different operations can be used to place the same number in the same cell; other times only a specific one will do. Note also that often an operation can only be applied if a previous one has placed a specific number; in other words, there are many *logical sequences* when using these logical operations.

B. Blind Sudoku

In order to make this problem interesting and challenging for evolutionary algorithms, the original Sudoku puzzle was

changed into a problem termed *Blind Sudoku*. The difference with the original puzzle lies on the way in which solutions are evaluated:

1) A sequence of instructions is applied to the puzzle;
2) A fitness score is returned to the whole sequence;
3) If the puzzle was not solved, it is reset to its original configuration.

This cycle continues until a solution is found, or a stop condition (such as a maximum number of evaluations) is reached. In other words, the puzzle is never available to the algorithm, neither is a measure of goodness for each individual logic instruction, effectively making the problem much harder.

This way of solving Sudoku puzzles is similar to the Santa Fe Ant Trail problem [21], in which a sequence of instructions is given to an ant in a toroidal grid world, and the number of food objects caught by the ant is returned as the fitness measure.

C. Objective Function

As the number of cells in a 9×9 Sudoku board is always 81, that is used as the size of a sequence of instructions; this makes the system applicable to any Sudoku puzzle, as it is independent of the number of givens. The fitness of a sequence of instructions is simply the sum of the fitness of all its instructions; the fitness of each single instruction is

$$f_i = \begin{cases} k \times (82 - i) & \text{if successful} \\ coverage - 9 & \text{if unsuccessful} \end{cases}, \qquad (2)$$

where k is a constant, and *coverage* is a measure of how many cells were ruled out when unsuccessfully trying to place a number in a region (the X marks)[2]. If the puzzle is completed before using up all the instructions, all remaining instructions are considered neutral, and their fitness is $f_i = 0$.

The fitness function heavily rewards successful instructions, and punishes unsuccessful ones. A linear decreasing reward is applied to each successful instruction: the earlier it is executed, the bigger the reward. This temporal saliency is regulated by the k parameter; the higher its value, the higher the reward (following the recent application of GAuGE to Sudoku [5], a value of $k = 81$ was used).

This problem has interesting characteristics, and can be compared to a class of scheduling problems. There is a clear temporal dependency between each phenotypic variable, as certain instructions can only be successful if a set of instructions is executed before them. The negative score applied to unsuccessful instructions can be seen as an effort factor.

The GAuGE system seems adequate to solving this kind of problem. It allows for successful instructions, which have already been discovered, to change their phenotypic location (by mutating their position specification at the genotypic level), allowing them to be moved to the start of the phenotype string, thus possibly maximising their contribution to the fitness of the set of instructions.

[2]Note that the Raising Numbers technique does not mark any cells as unsuitable, and thus if it is unsuccessful its fitness is always -9; this is judged to be fair, as it is a slightly more expensive technique then all others.

V. EXPERIMENTS

A. Practical Considerations

The experimental setup used in here is identical to that of the previous application of GAuGE to the Blind Sudoku problem [5], that is, sequences of 81 instructions are evolved, from the set {SliceDice, RowFill, ColFill, RaisingNumbers}. If an instruction is successful, the LastRemaining instruction is tried on the corresponding region (as it is a fast instruction), and if successful the remaining number is automatically placed.

Also, the logic instructions are mapped onto the original board. For example, if the algorithm tries to execute the instruction *SliceDice(2,8)* (place number 8 in block 2 using SliceDice) on the board from Fig. 1, that instruction is translated to *SliceDice(2,9)*, as 8 already exists in block 2. This is only applied to the original board: if a subsequent instruction is also *SliceDice(2,8)*, then it is considered unsuccessful, as number 9 has already been successfully placed in block 2.

The test set for these experiments consisted of puzzles taken from Carol Vorderman's How to do Sudoku [20] (pp. 178–187). These were taken from the "Difficult" section, and the first twenty puzzles were picked (#111 to #130).

B. Encoding and Parameters

A standard genetic algorithm (SGA) [6] was used in a direct comparison with GAuGE, so as to ascertain the advantage of the mapping process employed, and its effect on the genetic operators. Table I shows the parameters used on both systems, which are identical to those previously reported [5], apart from the number of generations, which was raised to 3200 (so that more SGA runs are successful for comparison). The Minimal Generation Gap model (MGG) [22] was used as a replacement strategy, as previous tests, both published [23], [5] and not, suggest that it is appropriate for the GAuGE system. It was also used with the SGA, as it increased its performance when compared with a generational approach.

TABLE I

EXPERIMENTAL SETUP

Replacement strategy:	MGG
Crossover operator:	1-point
Problem length (ℓ):	81
Population size (N):	100
Max. number of generations:	3200
(GAuGE only) Position field size (pfs):	8 bits
Value field size (vfs):	12 bits
Crossover probability:	0.5
(GAuGE only) Position field mutation probability:	0.0015
Value field mutation probability:	0.001

In GAuGE, the parameter $pfs = 8$ was used, to encode 81 instructions and to allow for a degree of degeneracy and soften the biases of the mod operator [24], [5].

In what concerns the vfs parameter, on both systems, each variable encodes three choices:

1) which instruction to use (out of 4 instructions);
2) which region to apply it to (out of 9 regions, be it blocks, rows or columns);

3) which number to attempt to place (out of 9 numbers).

To encode an instruction, 2 bits are sufficient. To encode a region and a number, the minimum number of bits is 4 ($2^4 = 16$); as with the pfs parameter, degeneracy is used, and 5 bits are used for each of these encodings, so $vfs = 2+5+5 = 12$. This means that the length of genotype string, for each system, is:

$L_{SGA} = vfs \times \ell = 12 \times 81 = 972$ bits

$L_{GAuGE} = (pfs + vfs) \times \ell = (8 + 12) \times 81 = 1620$ bits

VI. ANALYSIS

A. Results

Table II shows the results obtained. For each puzzle, 30 independent runs were performed; each run was stopped after all missing numbers were placed on the board, or the maximum number of generations was reached.

Puzzle	SGA			GAuGE		
	Avg.	Runs	Gen.	Avg.	Runs	Gen.
#111	53.0	30	384	53.0	30	238
#112	51.0	30	335	51.0	30	181
#113	53.0	30	300	53.0	30	166
#114	53.0	30	430	53.0	30	210
#115	51.0	30	157	51.0	30	123
#116	21.0	0	N/A	21.0	0	N/A
#117	13.0	0	N/A	13.0	0	N/A
#118	54.0	30	1021	54.0	30	301
#119	51.0	30	264	51.0	30	188
#120	51.0	30	186	51.0	30	135
#121	51.0	30	183	51.0	30	146
#122	51.0	30	324	51.0	30	157
#123	52.7	30	659	51.8	23	N/A
#124	55.4	30	564	56.0	30	247
#125	55.0	30	201	55.0	30	146
#126	54.8	29	N/A	55.0	30	277
#127	53.0	30	431	53.0	30	222
#128	53.0	30	297	53.0	30	189
#129	51.0	30	421	51.0	30	220
#130	49.7	22	N/A	53.0	30	454

Figure 5 shows an histogram of the performance of both systems for each puzzle used. As previously reported [5], puzzles #116 and #117 were unsolvable with the instruction set available, hence the lack of results for these. Across all other puzzles, GAuGE consistently outperformed the SGA; it found solutions faster (particularly with difficult puzzles, such as #118 and #124), and had more successful runs.

An exception to this was puzzle #123, in which the SGA outperformed GAuGE. Analysis of the results obtained showed however that is was due to the fitness function used (Eq. 2); the average best fitness obtained with the SGA was 254057, whereas with GAuGE it was 265809, but in this particular puzzle instance, a higher fitness did not correspond to a higher amount of numbers placed.
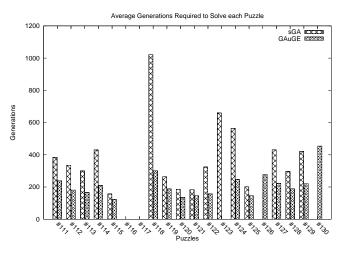

Fig. 5. Average number of generations required to solve each puzzle, for both the SGA and GAuGE. Absence of results indicates a puzzle where less than 30 runs were successful.

B. Example Puzzle

In this subsection, the results obtained with both systems for puzzle #111 are analysed (results for practically all other puzzles are comparable to these). Fig. 6 shows the average amount of numbers placed per generation, Fig 7 shows the average best fitness per generation, and finally Fig. 8 shows the cumulative frequency of success for both systems.

Fig. 6. Mean numbers placed per generation, for puzzle #111 (out of a maximum of 53). Error bars indicate standard deviation for all 30 runs.

These graphs show how GAuGE converges much faster to a solution than the SGA, and typically with less variance across runs. This is quite explicit when observing Fig. 7, where there is a significant difference between the performance of both systems, with only a minimal amount of overlapping of standard deviation error bars. It is also interesting to observe how GAuGE achieves a much higher fitness than the SGA, and how this fitness keeps increasing at a substantial rate, even after generation 360 (by which all
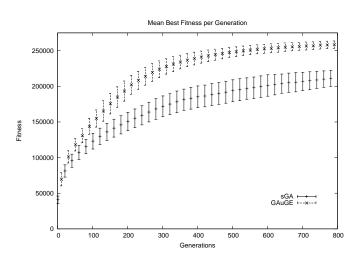
Fig. 7. Mean best fitness per generation, for puzzle #111. Error bars indicate standard deviation for all 30 runs.
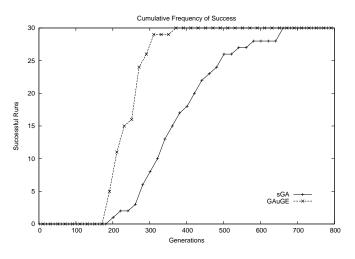

Fig. 8. Cumulative frequency of success per generation, for puzzle #111.

```
10000100000000000001000000100000000001000000100000000001000100000000000000000000010
10000100110000100010000100011100010001110010010000000001000000000000000000000000000
10010100110000100010000100011000100011100100100000000000000000000000100000000000010
11000110110001110001000100000101010010100000000000000000000000000010000010000000011
11000110110001110001011100000101010010100000000000000000011001000000000010000000010
11010110110010110100000100100010010100000000000001100100000000001001001000000
10101101001011110000101000001000100100101001010000000000000010010001001001000000
11011110110101001001100000100001000100000000100100000000000010010001001001000000
10111110110011110000110010010000100100000010100000001101000000000000010001000000000
10111110110011110000011001000000001011000000101000001101000010000000001000000000000
10111110110011110000011001000000001011000011100000011011000100010000000100100000000
10111110110011110000011001000000001011000001110000011011000100011101000000000000
10111110110011110000011001000000001011000001110000011011000100010010011000000000000
10111110110011110000011001000000001011000001110000011011000100010001100100000010100
10111110110011110000011001000000001011000001110000011011000101001011010000110000
10111110110011110001100010000000010110000011000001101010101000001011000001100000
10111110110011110001100010000000010110000011000001101010101000001011000001100100
10111110110011110001100010000000010110000011000001101010101010101101000001100100
10111110110011110001100000001010000000110000011100001101010101010101101000000110100
10111110110111100001100000001010000011100001101001010101010101101000000110100
10111110110111100001100000000010000011100001101001010101010101101000000110100
10111110110111100001100000000010000011100001101001010101010101101000000110100
10111110110111100001100000000010000011100001101001010101010101101000000110100
10111110110111100001100000000010000011100001101001010101011100000110100
10111110110111100001010000000010000011100001101001010101011100000110100
10111110110111100001100010000000010000011100001101001010101011100000110100
10111110110111100100101100000010010000011100001101001010101011100000110100
10111110110111100100101100000010010000011100001101001010101011100000110100
10111110110111100101000010000000001110000110001101001010101011110000110100
10111110110111100101000010000000001110000110001101001010101011110000010100
10111110110111100101100000000001011000011000011100100101011111000010000
10111110110111100101100000000001011000011000011100100101011111000010000
10111110110111100101100000000001011000011000011100100101011111000010000
10111110110111100101100000000001011000011000100101001111000010000
```

Fig. 9. Best individual found with the SGA, every 20 generations (from sample run). Strings represent sequences of instructions: a 0 indicates an unsuccessful instruction, and a 1 a successful one. Successful instructions keep the original phenotypic locations where they were first found, until mutation creates a similar instruction closer to the start of the individual.

runs have successfully placed all 53 numbers).

But how do the genetic operators contribute to GAuGE's performance? An analysis of a sample run for puzzle #111 sheds some light into the matter. Figs. 9 and 10 show the best phenotype string evolved with the SGA and GAuGE respectively, every 20 generations: these strings are expressed as a sequence of logical instructions, with a 0 indicating an unsuccessful instruction, and a 1 a successful one.

These figures show a fundamental difference in the way both systems solve this problem. Whenever the SGA discovers new instructions, they are exchanged between individuals while keeping their global phenotypic location. As a result, the left side of the best phenotype string tends to converge quickly, often including a few unsuccessful instructions. The only mechanism available to the SGA to move successful instructions towards the left-side of these sequences (leading to better fitness, as per Eq. 2) is to rediscover those instructions at a new location, closer to the start of the sequence.

GAuGE solves this problem in a totally different manner, thanks to its genetic operators. Whenever a successful instruction is discovered, it can be moved towards the start of the sequence, by means of mutating its position specification at the genotypic level; this will not affect most of the other successful instructions, as they will keep their relative ordering between themselves and the moved instruction. Furthermore, moving an instruction leftwards will open up the possibility of moving instructions which depend on it for success to be moved leftwards as well, thus increasing the fitness of the individual.

Crossover, on the other hand, plays a role in exchanging discovered sequences: as the relative ordering of the exchanged parts is maintained, so is the dependency of certain instructions on previous ones for success. While the SGA exchanges the absolute location of those sequences, GAuGE exchanges their relative location, thus enabling the mixing of logical sequences of instructions originating from both parents, creating potentially fitter offspring.

VII. CONCLUSIONS

There is no perfect algorithm, and this is certainly the case with GAuGE; the effect of its mapping process on genotypic strings can be quite drastic, seriously affecting its performance on certain problems, and leading to the design of specific genetic operators [3]. This is certainly the case for problems with "position-sensitive data", in which the functionality of a value is completely dependent on its phenotypic location, and different phenotypic variables encode different types of data.

```
00000110000010000000000000001000100000000000100010000001100000000100000000000000000
10001110000011000001110000000000010000001100100000000010100000000000000000000000000
11011100000011000010000000001101100000001000100000001010000000000000000000000000000
11101101010101010000001100100101100000010000000000000000010000000000000000000000000
10011100011110011110001001110100000010000000000101000000000001000000000000000000001
11011010011111110011110010010000010000000001100010000001100000000100000000000000001
11011110110111110011110000000100000000001100010000101100001000000000000010010000000
10011101111111111001100010001100000101000001011000000001100000000000001000001000000
11011110110111110101010010010001100011000010010001100000100000000000000010010000000
11011110110110111100110000110010100000010100011011001000001000000000010010010010000
11011110111111101010001010100010001000110000111000110000010000010011000000000000000
11111110111111111101101010010011000100001010010101000101001010001010010100000000000
11111110111111111101111010010001000010010001111110100000000001001100000000000000000
11111110111111111111010100001001000000001111010001001001100100000100000000000000000
11111110111111111111101010000010001100000011110100000010010100000010000000000000000
11111110111111111111111001000100010000100010000101110100010100100000000000000000000
11111111111111111101111110100100101000011110010100000010100000000000000000000000000
11111111111111111111110010110010000101100110010010010000000000000000000000000000000
11111111111111111111110011111001011101001000001011001100100000000000000000000000000
11111111111111111111110011110010100000001011001011001100100000000000000000000000000
11111111111111111111110101001001000100011010100010001000010000000000000000000000000
11111111111111111111110011100110001001100110000100100000100000000000000000000000000
11111111111111111111110100110101011101000100000100000000000000000000000000000000000
11111111111111111111110101011110010100010001000000000000000000000000000000000000000
11111111111111111111110110111110010010000110000000000000000000000000000000000000000
11111111111111111111110101111110010110010000011100000000000000000000000000000000000
11111111111111111111111101110111000010000110000000000000000000000000000000000000000
11111111111111111111111101111110100000001110000000000000000000000000000000000000000
11111111111111111111111101111101000001001110000000000000000000000000000000000000000
11111111111111111111111110010010001000001000000000000000000000000000000000000000000
11111111111111111111111110010100101010000100000000000000000000000000000000000000000
11111111111111111111111111110011110100000000000000000000000000000000000000000000000
```

Fig. 10. Best individual found, every 20 generations (from sample run). Strings represent sequences of instructions: a 0 indicates an unsuccessful instruction, and a 1 a successful one. As evolution progresses, successful instructions are moved towards the start, due to GAuGE's disassociation between position and value specifications, and to fitness pressure.

For certain classes of problems, however, those effects might not be detrimental, and in certain cases they are indeed a welcome feature. GAuGE has been shown to perform just as well as the SGA on simple binary maximisation problems such as Onemax [1] and Binint [2], and in a problem where the order of the discovered data is paramount, GAuGE was shown to outperform [1] both a SGA and the original messy genetic algorithm [13].

The interesting characteristic of GAuGE's search operators is that they do not have to be specifically encoded; they derive from standard genetic algorithm operators, which, by the nature of the mapping process, are transformed into relative ordering operators.

The problem in this paper is an example of the class of problems for which GAuGE is well suited: problems where not only the contents of variables have to be found (and the alphabet is too extensive for a local exhaustive search), but also the ordering of those contents. Many scheduling problems share these characteristics, so it would be interesting to test the suitability of GAuGE to such problems.

REFERENCES

[1] 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

[2] 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

[3] 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, Vol. 3003. Springer-Verlag. (2004) pp. 125–137

[4] Nicolau, M. and Ryan, C.: Crossover, Population Dynamics and Convergence in the GAuGE System. In: Deb et al. (Eds.): Genetic and Evolutionary Computation - GECCO 2004. Lecture Notes in Computer Science, Vol. 3102. Springer. (2004) pp. 1414–1425

[5] Nicolau, M., and Ryan, C.: Solving Sudoku with the GAuGE System. In: Collet et al. (Eds.): Genetic Programming. Proceedings of the 9th European Conference, EuroGP 2006. Lecture Notes in Computer Science, Vol. 3905. Springer-Verlag. (2006) pp. 213-224

[6] Goldberg, D. E.: Genetic Algorithms in Search, Optimization and Machine Learning. Addison Wesley. (1989)

[7] Bagley, J. D.: The Behaviour of Adaptive Systems which Employ Genetic and Correlation Algorithms. PhD Thesis, University of Michigan. (1967)

[8] Frantz, D. R.: Non-linearities in Genetic Adaptive Search. PhD Thesis, University of Michigan. (1972)

[9] Holland, J. H.: Adaptation in Natural and Artificial Systems (Second Edition). University of Michigan Press. (1992)

[10] Davies, L.: Applying Adaptive Algorithms to Epistatic Domains. In Joshi, A. K. (Ed.): Proceedings of the Ninth International Joint Conference on Artificial Intelligence. Morgan Kaufmann Publishers. (1985) pp. 162–164

[11] Goldberg, D. E., and Lingle, J. R.: Alleles, Loci, and the Travelling Salesman Problem. In Grefenstette, J. J. (Ed.): Proceedings of the First International Conference on Genetic Algorithms. Lawrence Erlbaum Associates. (1985) pp. 154–159

[12] Oliver, I. M., Smith, D. J., and Holland, J. R. C.: A Study of Permutation Crossover Operators on the Travelling Salesman Problem. In Grefenstette, J. J. (Ed.): Proceedings of the Second International Conference on Genetic Algorithms. Lawrence Erlbaum Associates. (1987) pp. 224–230

[13] Goldberg, D. E., Korb, B., and Deb, K.: Messy genetic algorithms: Motivation, analysis, and first results. Complex Systems, Vol. 3, number 5. (1989) pp. 493–530

[14] Goldberg, D. E., Deb, K., Kargupta, H., and Harik, G.: Rapid, Accurate Optimization of Difficult Problems Using Fast Messy Genetic Algorithms. In Forrest, S. (Ed.): Proceedings of the Fifth International Conference on Genetic Algorithms. Morgan Kaufmann Publishers. (1993) pp. 56–64

[15] Harik, G.: Learning Gene Linkage to Efficiently Solve Problems of Bounded Difficulty Using Genetic Algorithms. Doctoral Dissertation, University of Illinois (1997)

[16] O'Neill, M. and Ryan, C.: Grammatical Evolution - Evolving programs in an arbitrary language. Kluwer Academic Publishers. (2003)

[17] Bean, J.: Genetic Algorithms and Random Keys for Sequencing and Optimization. ORSA Journal on Computing, Vol. 6, No. 2. (1994) pp. 154–160

[18] Wikipedia: Sudoku. http://en.wikipedia.org/wiki/Sudoku

[19] Yato, T., and Seta, T.: Complexity and Completeness of Finding Another Solution and its Application to Puzzles. IEICE Transactions on Fundamentals of Electronics, Communications and Computer Sciences, Vol. 86, No. 5. (2003) pp. 1052–1060

[20] Vorderman, C.: Carol Vorderman's How To Do Sudoku. Ebury Press. (2005)

[21] Koza, J. R.: Genetic Programming: On the Programming of Computers by Means of Natural Evolution. MIT Press. (1992)

[22] Satoh, H., Yamamura, M., and Kobayashi, S.: Minimal Generation Gap Model for GAs Considering Both Exploration and Exploitation. In: Proceedings of the 4th International Conference on Fuzzy Systems, Neural Networks and Soft Computing (Vol. 2). World Scientific, Singapore. (1996) pp. 494–497

[23] Ohnishi, K. Sastry, K., Chen, Y.-P., and Goldberg, D.: Inducing Sequentiality Using Grammatical Genetic Codes. In: Deb et al. (Eds.): Genetic and Evolutionary Computation - GECCO 2004. Lecture Notes in Computer Science, Vol. 3102. Springer. (2004) pp. 1426–1437

[24] 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, Vol. 2936. Springer-Verlag. (2003) pp. 15–26