OMEGA - Ordering Messy GA : Solving Permutation Problems with the Fast Messy Genetic Algorithm and Random Keys

Dimitri Knjazew
Dept. of General Engineering
University of Illinois
Urbana, IL 61801
dimitri@illigal.ge.uiuc.edu

David E. Goldberg
Dept. of General Engineering
University of Illinois
Urbana, IL 61801
deg@uiuc.edu

Abstract

This paper presents an ordering messy genetic algorithm (OmeGA) that is able to solve difficult permutation problems efficiently. It is essentially a fast messy genetic algorithm (fmGA) using random keys to represent chromosomes. Experimental results that show the random key-based simple genetic algorithm (RKGA) being outperformed by its messy competitor in 32-length ordering deceptive problems are presented.

1 INTRODUCTION

Various genetic-evolutionary algorithms (GEAs) have been developed for solving permutation problems over the past few years. Research in this area is very interesting, since there is a great variety of permutation-based commercially interesting applications including scheduling, timetabling and vehicle route planning problems. Unfortunately, many methods use either problem-specific or ad-hoc representation codings and operators. Also, the performance has not been sufficiently tested on hard problems, leaving the question of how the algorithm scales up unanswered.

Therefore, it would be interesting to design genetic algorithms (GAs) that use efficient codings and operators and have good scale-up properties. Furthermore, a more detailed and systematic analysis should be done on the GA performance. We suggest that so-called *competent genetic algorithms* that solve hard problems quickly, reliably and accurately (Goldberg, 1993), would be good candidate approaches for this undertaking. Much research has been done in this area and numerous competent GAs have been developed, including the fmGA (Goldberg, Deb, Kargupta, & Harik, 1993), the gene expression messy genetic algorithm (Bandyopadhyay, Kargupta, & Wang, 1998), the linkage learning genetic algorithm (Harik, 1997) and the Bayesian optimization algorithm (Pelikan, Goldberg, & Cantú-Paz, 1999).

This paper develops the ordering messy GA (OmeGA) specialized for permutation problems which represents the solutions by vectors of real numbers (random keys). In a number of experiments it is shown that the OmeGA significantly outperforms the simple GA in solving ordering deceptive problems, which are hard sequencing problems defined elsewhere (Kargupta, Deb, & Goldberg, 1992).

We start with a background description (section 2) and develop the OmeGA (section 3). We then continue with an introduction to the fast messy genetic algorithm (section 4) and random keys (section 5). Afterwards, we describe the use of multiple epochs (section 6) and introduce ordering deceptive problems (section 7) and different codings that determine the problem difficulty (section 8). Finally, we present experimental results (section 9) and conclude the paper.

2 BACKGROUND

This section gives background information necessary for understanding the following sections. We start with a short overview of some permutation-oriented genetic-evolutionary algorithms (GEAs) and discuss the notion of building blocks and deceptive problems. We take a closer look at competent GAs and explain why we decided to focus on the fmGA for this paper. Finally, we overview some representation models for

permutations.

Numerous genetic-evolutionary algorithms have been designed for a wide variety of permutation-based problems over the past few years. Problems such as the traveling salesman problem (Goldberg & Lingle, 1985), scheduling (Davis, 1985), vehicle route planning (Blanton Jr. & Wainwright, 1993) or integrated circuit design (Louis & Rawlins, 1991) have been tackled. Many of these tasks are commercially important. However, only little research work has been done to examine how permutation-solving GEAs scale up or, in other terms, how their computational complexity increases with the underlying problem difficulty and size.

One approach to investigate the scale-up behavior of a GA is testing it on artificial problems where the solution is known a priori and where the problem difficulty can be varied. For this purpose researchers in the GA community frequently used *deceptive problems* which are hard multimodal optimization problems for binary strings introduced by Goldberg (1989). A deceptive problem may be designed by combining a desired number of *deceptive subfunctions* that mislead the genetic algorithm letting it converge to certain local optimal points. No information about the subfunctions is passed to the GA. To find the global optimum, partial solutions (or schemas) with above-average fitness—the so-called *building blocks*—must be identified and grouped together. Kargupta, Deb, and Goldberg (1992) developed *ordering deceptive problems* for permutations which we will discuss in detail in section 7.

A deceptive problem's degree of difficulty grows with increasing number and size (order) of the subfunctions. Particularly, for the simple genetic algorithm the problem difficulty increases when a loose coding is chosen such that the elements of the subfunctions are mapped to distant positions within the problem representation. This results in a greater building-block length and, consequently, the building blocks are more likely to be disrupted when traditional recombination operators such as single-point or two-point crossover are applied. In general, no fixed recombination operators guarantee proper mixing with an arbitrary coding. This problem is usually referred to as "the linkage problem" in the literature. Thierens and Goldberg (1993) showed in a dimensional analysis of mixing processes in simple genetic algorithms that the required population size grows exponentially with the building-block length and number. Section 8 introduces some problem codings for ordering deceptive problems.

To tackle the linkage problem and to achieve a better scale-up behavior, first-generation genetic algorithms need to be extended by some additional mechanisms. In this paper we focus on the fast messy GA (fmGA) that uses an adaptive representation of the solutions and can be easily transformed into a permutation-solving GA. We explain the mechanics of the fmGA in section 4.

Various representation models for permutations are proposed in the GA literature. Very often, integer numbers are used to encode a sequence directly. Numerous crossover operators have been designed to keep the offspring generated by recombination operators feasible, for example the partially mapped crossover (PMX) (Goldberg & Lingle, 1985), the uniform ordering crossover (UOX) (Davis, 1991) and the relative ordering crossover (ROX) (Kargupta et al., 1992) we will later refer to in section 7.

An alternative way to encode sequences is using binary matrices that describe the relative order of the permutation elements. If a matrix A has a 1 bit on position (i, j), the element i has to appear before element j in the sequence and vice versa for a 0 bit. Unfortunately, this representation is not perfect, since it involves many nonexistent orderings, as was pointed out in Whitley and Yoo (1994). Therefore, repair mechanisms are required to obtain valid permutations.

For the random-key representation, on the contrary, no repair is needed and traditional crossover operators can be applied in the normal way. We explain how to encode permutations by random keys in section 5.

3 DESIGNING THE OMEGA

In this section we introduce the *ordering messy genetic algorithm* (OmeGA). We design the OmeGA on the basis of three key ideas:

- all basic mechanisms of the fast messy genetic algorithm are applied
- the alleles are real or long integer numbers
- the alleles are treated as random keys to encode permutations.

Like in the fmGA, the "messy" genes are represented as a pair of gene locus and allele, except that real numbers that are treated as random keys replace binary digits:

messy gene: (*gene locus*, *random key*).

All "messy" genetic operators are used in the normal way. This poses no infeasibility problems for the robust random-keys coding of the permutations. By ap-

plying the mechanisms of the fmGA, we ensure a good building-block mixing and independence from the underlying problem coding. Therefore, one would expect the GA to scale up well.

4 A BRIEF INTRODUCTION TO THE FAST MESSY GENETIC ALGORITHM

The first steps towards the development of competent genetic algorithms were taken by Goldberg, Korb, and Deb (1989) who developed the messy GA (mGA) in 1989. However, first-generation messy GAs suffered some handicaps which made application to large-sized problems impossible. An interested reader might refer to Goldberg, Deb, and Korb (1990) for more details. The fast messy GA is an improved version of the messy GA and was developed four years later by Goldberg, Deb, Kargupta, and Harik (1993). This section briefly describes its key features. We first explain the messy representation and operators. Then, we look at the organization of the fmGA and finish with some important techniques that remarkably contribute to its success. For a more detailed description we refer to Goldberg et al. (1993).

Unlike the simple genetic algorithms, that use a fixed chromosome coding, messy GAs represent the genes by the pair (*allele locus*, *allele value*) in chromosomes of variable length. Thus, a string of messy genes may be *over-specified* when multiple versions of the same gene exist or *under-specified* when certain genes are missing. To evaluate over-specified chromosomes the genes are scanned from left to right with a first-come-first-serve precedence rule. For evaluating under-specified chromosomes a *competitive template* which is a completely specified fixed-bit string is used in the fmGA. Before evaluation, the chromosome's missing genes are filled with the corresponding alleles from the template. Note that a chromosome can be both under- and over-specified.

Like in simple GAs, selection and recombination is used to create a new population, except that traditional crossover is replaced by *cut* and *splice operators* (Goldberg, Korb, & Deb, 1989). The cut operator breaks a messy chromosome into two parts with a cut probability $p_c = p_\kappa(\lambda - 1)$, where p_κ is a specified bitwise cut probability and λ the length of the chromosome. The cut position is randomly chosen along λ. The splice operator joins two chromosomes with a certain splice probability p_s.

The fast messy GA is organized in two nested loops: the *outer loop* and *inner loop*. The outer loop iterates over the order k of the processed building blocks. Every cycle of the outer loop is denoted as an *era*. When a new era starts, the inner loop is invoked which is divided into the three phases:

- Initialization Phase
- Building-Block Filtering Phase
- Juxtapositional Phase

The initialization phase creates a population of random individuals. The population size has to be large enough and chromosomes have to be long enough to ensure the presence of all possible genic and allelic combinations—candidates for building blocks after the initialization phase is completed. Then, the building-block filtering phase is invoked that works like a filter: "bad" genes not belonging to building blocks are supposed to be filtered out such that afterwards the population contains a high proportion of short strings consisting of "good" genes. This is accomplished by repeatedly performing selection and deleting random genes in all chromosomes until the overall string length is reduced to a value near k.

During the juxtapositional phase the selection operator is used and the above described cut and splice operators are applied to combine the short strings together that hopefully form the global optimal solution. After the juxtapositional phase is finished, the inner loop of the fmGA terminates. The actual template is then replaced by the best individual found so far, which becomes the new template for the next level and so on. Thus, the set of local optimal points discovered on level k serves as a launch pad for level $k+1$. The whole procedure can be repeated until a maximum level (era) is reached.

The fmGA uses two important techniques that remarkably contribute to its success: *thresholding* and *tie-breaking*. Thresholding refers to a certain amount of genes two chromosomes must have in common before they can be compared during selection. If two individuals share θ genes together, they are ready to compete against each other. This method prevents the competition of building blocks belonging to different subfunctions—an effect called *cross-competition* in the literature—and successfully tackles problems with non-uniformly scaled building-blocks, that is, building blocks with different contributions to the chromosome fitness.

Tie-breaking extends the selection operator as follows: when two or more individuals of the same fitness are compared during selection, the one with the shortest

string length is preferred. In addition, the fmGA initialization is extended such that chromosomes of all possible building-block lengths from 1 to k are created. Tie-breaking helps solving problems with non-uniformly sized building-blocks.

5 USING RANDOM KEYS FOR REPRESENTATION

This section explains the concept of random keys and the random key-based simple genetic algorithm (RKGA).

The random keys have been introduced by Bean (1994). Here, real or long integer random numbers are used as sort keys to decode a sequence. A permutation of length l is then represented as a real vector $\mathbf{r} = (r_1, r_2, \ldots, r_l)$ with typically $\mathbf{r} \in [0,1]^l$. By sorting the random keys such that

$$r_{\phi(1)} \leq r_{\phi(2)} \leq ... \leq r_{\phi(l)}$$

holds, where $\phi : \{1,..,l\} \rightarrow \{1,..,l\}$ is the corresponding mapping function arranging the keys in ascending order, the permutation is decoded as follows:

$$(\phi(1), \phi(2), \ldots, \phi(l)).$$

The following example demonstrates how random key chromosomes are decoded after single-point crossover. Crossing two parent strings

A: (0.46,0.91|0.33,0.75,0.51) $\equiv$ (3 1 5 4 2)
B: (0.84,0.32|0.64,0.04,0.48) $\equiv$ (4 2 5 3 1)

after the second gene yields the following offspring:

A': (0.46,0.91,0.64,0.04,0.48) $\equiv$ (4 1 5 3 2)
B': (0.84,0.32,0.33,0.75,0.51) $\equiv$ (2 3 5 4 1).

Note that the corresponding permutations on the right side are valid. In general, any sequence of real numbers can be interpreted as a valid permutation. Thus, traditional recombination operators would always generate feasible offspring when used on random key vectors. Besides recombination, a simple mutation operator could be implemented by replacing a random key with a new randomly generated number. More sophisticated mutation operators are discussed in Norman and Bean (1997) for random keys or in Janikow and Michalewicz (1991) for real vectors in general. Also, various mutation techniques from evolution strategies (Schwefel, 1995) can be adopted.

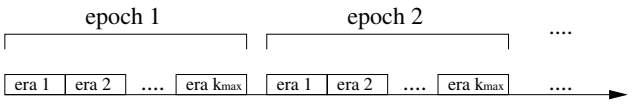


Figure 1: Multiple epochs in the OmeGA. After the outer loop of the OmeGA terminates, an epoch is completed. Then, the best individual found in era k is carried over to the next epoch where it serves as a competitive template.

A simple GA that uses the random key representation is denoted by "random key genetic algorithm" (RKGA) in the literature. A detailed description of the RKGA can be found in Bean (1994).

6 MULTIPLE EPOCHS

Referring to the advice given by Kargupta (1995) to apply the fmGA iteratively at each level, we extended the OmeGA by enclosing the outer loop into an external cycle that iterates over a desired number of *epochs*. An epoch starts with the first era and finishes with the maximum era k_{max}, as illustrated in figure 1. Afterwards, the best individual found so far is used as a competitive template for the succeeding epoch and so on. Multiple epochs are useful when the population size is not large enough but cannot be further increased because of memory reasons. Then, there is still a chance of finding the global optimal solution in a later epoch. This idea is also motivated by research work on the "fundamental tradeoff"—the tradeoff between population size and the number of epochs (Goldberg, 1999).

7 ORDERING DECEPTIVE PROBLEMS

This section introduces two concrete deceptive problems for permutations that will be later used to test the performance of the ordering GAs. We start with a description of two ordering deceptive functions and then show how to construct ordering deceptive problems. We finally give some comments on previous research work that has been done on this topic.

Kargupta, Deb, and Goldberg (1992) defined two deceptive functions of order four: the relative ordering function, here denoted by f_{rel}, and the absolute ordering function, here denoted by f_{abs}. These functions are defined as follows.

relative ordering function f_{rel}:

```
f(1 2 3 4) = 4.0    f(4 2 1 3) = 1.2
f(1 2 4 3) = 1.1    f(4 1 3 2) = 1.2
f(1 3 2 4) = 1.1    f(1 4 2 3) = 1.2
f(1 4 3 2) = 1.1    f(2 3 4 1) = 1.5
f(2 1 3 4) = 1.1    f(4 1 2 3) = 2.1
f(3 2 1 4) = 1.1    f(3 4 1 2) = 2.2
f(4 2 3 1) = 1.1    f(3 1 4 2) = 2.2
f(2 4 3 1) = 1.2    f(2 1 4 3) = 2.4
f(2 3 1 4) = 1.2    f(4 3 2 1) = 2.4
f(3 1 2 4) = 1.2    f(4 3 1 2) = 2.4
f(1 3 4 2) = 1.2    f(2 4 1 3) = 2.4
f(3 2 4 1) = 1.2    f(3 4 2 1) = 3.2
```

absolute ordering function f_{abs}:

```
f(1 2 3 4) = 4.0    f(2 4 3 1) = 2.0
f(4 2 3 1) = 1.8    f(3 2 4 1) = 2.0
f(1 3 2 4) = 1.8    f(1 4 2 3) = 2.0
f(1 2 4 3) = 1.8    f(4 1 2 3) = 2.6
f(1 4 3 2) = 1.8    f(3 4 1 2) = 2.6
f(3 2 1 4) = 1.8    f(2 3 4 1) = 2.6
f(2 1 3 4) = 1.8    f(2 4 1 3) = 2.6
f(4 2 1 3) = 2.0    f(2 1 4 3) = 2.6
f(4 1 3 2) = 2.0    f(4 3 2 1) = 2.6
f(3 1 2 4) = 2.0    f(4 3 1 2) = 2.6
f(1 3 4 2) = 2.0    f(3 1 4 2) = 2.6
f(2 3 1 4) = 2.0    f(3 4 2 1) = 3.3
```

In f_{rel}, only the relative ordering of the permutation elements matters. Here, the global optimal point is `(1 2 3 4)` with a function value equal to 4.0 and the misleading attractor is `(3 4 2 1)` with the second highest function value of 3.2. For the function f_{abs}, the elements have to be placed in correct absolute positions in addition to being arranged in the right relative order.

Let us consider a sample permutation and apply f_{abs}. The sequence

$$(\underline{1}\ \underline{2}\ 11\ 14\ \widehat{5}\ \widehat{6}\ \widehat{7}\ \widehat{8}\ 9\ 10\ \underline{3}\ 12\ \underline{4}\ 13\ 15\ 16\ \overline{20}\ \overline{18}\ \overline{19}\ \overline{17})$$

contains a building block consisting of the alleles {5, 6, 7, 8} placed on correct absolute positions $5-8$ and in the optimal order. Thus, the building block would contribute to the overall fitness by the value of 4.0. The elements {17,18,19,20} are present in the right section but their order corresponds to $4 \prec 2 \prec 3 \prec 1$, therefore they gain a score of 1.8. Here, $\prec$ denotes relative order. Finally, the items {1,2,3,4} are in the correct relative order but 3 and 4 are misplaced. In this case, a *partial credit* of 1.0 is given to this o-schema equal to the half of the number of alleles in the correct section.

Ordering deceptive problems can be constructed by concatenating a desired number of the above defined subfunctions. The overall function value of the whole permutation is the sum of the subfunction values. In this paper we consider 32-allele problems consisting of eight subproblems. On these problems Kargupta et al. (1992) tested the performance of the simple GA using several recombination operators: PMX, ROX and UOX. For the absolute problem only the GA using PMX could find the global optimum whereas the optimal solution of the relative problem could only be found with ROX. Since no problem information is supposed to be given beforehand, the simple GA, working with any crossover operator, would scale up badly for one of the two problems. This fact motivates the development of new GAs which are independent from the internal structure of the task to be solved.

subfunction No.	deflen6	loose
1	1, 3, 5, 7	1, 9, 17, 25
2	2, 4, 6, 8	2, 10, 18, 26
3	9, 11, 13, 15	3, 11, 19, 27
4	10, 12, 11, 16	4, 12, 20, 28
5	17, 19, 21, 23	5, 13, 21, 29
6	18, 20, 22, 24	6, 14, 22, 30
7	25, 27, 29, 31	7, 15, 23, 31
8	26, 28, 30, 32	8, 16, 24, 32

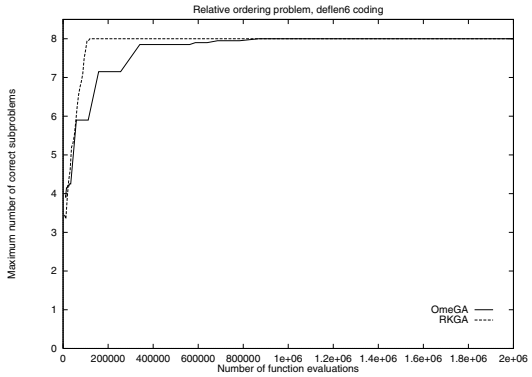
Table 1: Problem Codings. Here, genes belonging to the subfunctions are shown for deflen6 and loose coding.

8 PROBLEM CODINGS

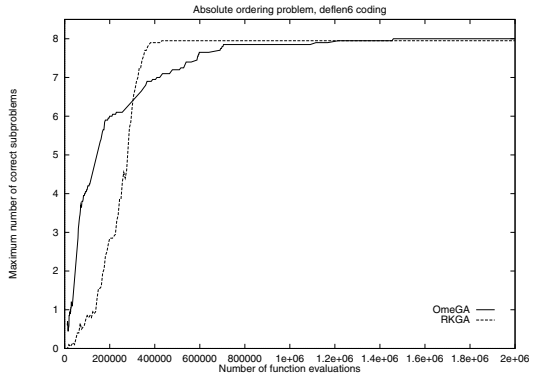
In the following paragraphs we define three different problem codings determining the degree of difficulty of ordering deceptive problems and explain how to decode and evaluate chromosomes. These codings will be later used to compare the scale-up properties of the OmeGA and the RKGA.

By *tight coding* we denote a coding scheme where building blocks are tight with a defining length three. The defining length of a building block is the distance between its outermost genes. In *deflen6 coding* the defining length is six. We further denote a coding where the sum of all defining lengths is maximal by *loose coding*. Table 1 shows the genes comprising the subfunctions for the loose and deflen6 coding.

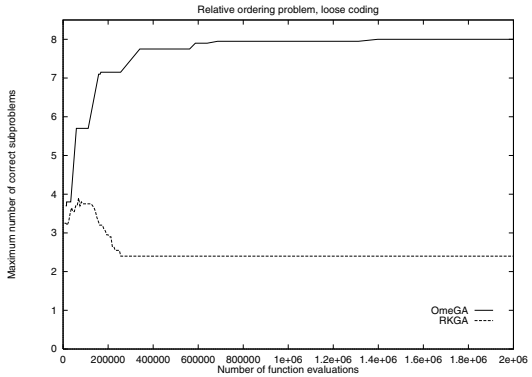
Coding-oriented function evaluation of a random key vector $\mathbf{r}$ works as follows. First, a copy of $\mathbf{r}$ is created. Afterwards, the elements of the copy are rearranged according to the coding scheme, yielding a new vector $\mathbf{r}'$ which is then transformed into a permutation and evaluated as described in section 7. Finally, the function value is assigned to $\mathbf{r}$ and $\mathbf{r}'$ is discarded. For instance, when using the loose coding in the 32-length



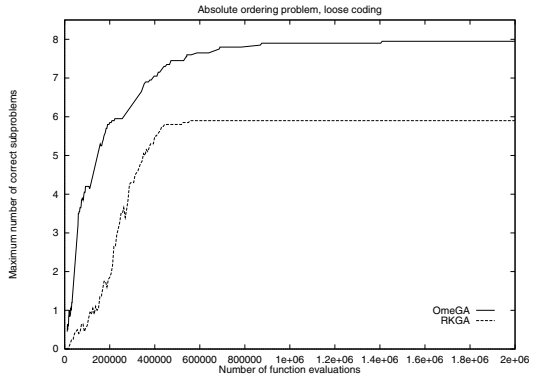
(a) relative problem, deflen6 coding



(b) absolute problem, deflen6 coding



(c) relative problem, loose coding



(d) absolute problem, loose coding

Figure 2: Maximum number of correct subfunctions found by the OmeGA and the RKGA for the relative and absolute ordering problem with deflen6 and loose coding.

ordering deceptive problem, the copy of the string $\mathbf{r}$ is rearranged in the following way, yielding $\mathbf{r}'$:

$$\mathbf{r} = (r_1, r_2, r_3, r_4, r_5, r_6, r_7, r_8, \quad \ldots \quad)$$
$$\mathbf{r}' = (r_1, r_9, r_{17}, r_{25}, r_2, r_{10}, r_{18}, r_{26}, \quad \ldots \quad).$$

The probability of building-block disruption grows with the defining length of the building blocks for a random-key based simple GA using one or n-point uniform crossover operators. Therefore, ordering problems coded with loose coding are harder for the GA to solve than those with tight coding. On the contrary, we would expect the OmeGA to find the global optimum independently from the underlying coding thanks to its flexible messy representation.

9 EXPERIMENTS

This section presents experimental results comparing the performance of the RKGA and the OmeGA for absolute and relative ordering problems of length 32. In all following experiments the mutation probability was kept zero to observe the effect of recombination alone. Moreover, in all runs the crossover probability in the RKGA and the splice probability in the OmeGA were set to 1.0. With these parameters the best results were obtained. Binary tournament selection without replacement was used. The RKGA performed the best with single-point crossover.

The OmeGA was organized as follows. The inner loop processed over 60 generations, including the juxtapositional phase which took 30 generations. The outer loop iterated over four eras and the maximum number of epochs was set to four. We used an empirically determined population size such that the global optimal solution could be found within four epochs. At the same time, we tried to keep the amount of function evaluations small. The number of individuals in the first, second, third and fourth era were 750, 1750,

3250 and 6250.

During the building-block filtering phase the chromosomes started with an initial length equal to the problem size 32 and were reduced to their corresponding building block length k at the end of the filtering phase. We used tie-breaking in all experiments, since it significantly contributed to the success of the OmeGA. For example, in era 3 there were 750 individuals of length 1, 1000 individuals of length 2 and 1500 individuals of length 3 present after the building-block filtering process. For the juxtapositional phase we limited the maximum allowed string length of the messy chromosomes to $2l$. The cut probability was set to 0.03.

The RKGA processed over 350 generations with a population size of 6250, equal to the maximal population size in the OmeGA. With these parameters we made 20 independent runs for the absolute and relative problems with all three codings. Since the experiments with tightly coded problems yielded almost the same results as with deflen6 coding, they are not presented here.

In figure 2a and 2b the maximum number of correct subfunctions found by the RKGA and the OmeGA is plotted versus the number of function evaluations for the relative and absolute problem with deflen6 coding. Both algorithms found the global optimum in every run. In all plots the points corresponding to the filtering phases are omitted in the OmeGA curve for simplicity. The number of function evaluations includes evaluations during the initialization and the filtering phases.

Figures 2c and 2d demonstrate the performance of both GAs for the loose coding. The OmeGA found the global optimum of the absolute problem in 19 out of 20 runs. The optimum of the relative problem was found in every run. The maximum number of correct subfunctions discovered by the RKGA was seven in the absolute and five in the relative problem. The global solution was not found at all. Especially figure 2c demonstrates how the RKGA is mislead by the deceptive attractors: while it converges to highly fit solutions, the number of correct building blocks decreases to a low value.

These results clearly show the OmeGA's independence from the underlying problem coding. For all tested codings the OmeGA curves have roughly the same appearance and the problem was completely solved in almost every run, whereas the RKGA succeeded only with tightly coded problems (deflen6 and tight coding). Although this is not an thorough scale-up analysis the results suggest that the OmeGA has a significantly better scalability than the RKGA. It is interesting that the RKGA curve in figure 2d converges to a significantly higher value than the RKGA curve in 2c. This indicates that the absolute problem is less deceptive for random-key based GAs.

10 CONCLUSIONS

In this paper we developed an ordering messy genetic algorithm — the fast messy GA representing its chromosomes with random keys. We introduced three problem codings and compared the performance of RKGA and OmeGA in experiments with length-32 ordering deceptive problems. In result, RKGA was clearly outperformed by its messy competitor. We summarize some benefits of OmeGA that became apparent in the experimental results:

- OmeGA is linkage-friendly and coding independent
- relative and absolute ordering deceptive problems are solved optimally.

For future research work we recommend a more accurate and detailed analysis on the scalability of the two GAs with different problem lengths. It would be also interesting to investigate OmeGA's performance on harder ordering problems with overlapping, differently sized, and differently scaled building blocks that we would expect to appear in real world permutation-based problems like scheduling or vehicle routing problems.

Acknowledgments

The authors would like to thank Franz Rothlauf, who proposed the usage of random keys, Martin Pelikan, Erick Cantu-Paz, Fernando Lobo and Martin Butz for their useful comments and suggestions. The authors would like to give special thanks to Professor Hans-Paul Schwefel for encouraging the first author to spend a portion of his diploma thesis studies with the second author thus enabling this collaboration to take place. Professor Schwefel also provided important guidance and essential suggestions that greatly improved this work. The first author was partially supported by the "Studienstiftung des Deutschen Volkes" (Germany) and the Sigma-Xi Research Society.

The work was sponsored by the Air Force Office of Scientific Research, Air Force Materiel Command, USAF, under grant F49620-97-1-0050. Research funding for

this work was also provided by the National Science Foundation under grant DMI-9908252. Support was also provided by a grant from the U. S. Army Research Laboratory under the Federated Laboratory Program, Cooperative Agreement DAAL01-96-2-0003. The U. S. Government is authorized to reproduce and distribute reprints for Government purposes notwithstanding any copyright notation thereon.

The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the Air Force Office of Scientific Research, the National Science Foundation, the U. S. Army, or the U. S. Government.

References

Bandyopadhyay, S., Kargupta, H., & Wang, G. (1998). Revisiting the GEMGA: Scalable evolutionary optimizaiton through linkage learning. (pp. 603–608). Piscataway, NJ: IEEE Service Center.

Bean, J. C. (1994). Genetic algorithms and random keys for sequencing and optimization. *ORSA Journal on Computing*, *6*(2), 154–160.

Blanton Jr., J. L., & Wainwright, R. L. (1993). Multiple vehicle routing with time and capacity constraints using genetic algorithms. In *Proceedings of the Fifth International Conference on Genetic Algorithms* (pp. 452–459).

Davis, L. (1985). Job shop scheduling with genetic algorithms. In *Proceedings of an International Conference on Genetic Algorithms and Their Applications* (pp. 136–140).

Davis, L. (1991). A genetic algorithms tutorial. In *Handbook of Genetic Algorithms* (pp. 1–101).

Goldberg, D. (1999). Using time efficiently: Genetic-evolutionary algorithms and the continuation problem. In Banzhaf, W., Daida, J., Eiben, A. E., Garzon, M. H., Honavar, V., Jakiela, M., & Smith, R. E. (Eds.), *Proceedings of the Genetic and Evolutionary Computation Conference GECCO-99*, Volume I (pp. 212–219). Orlando, FL: Morgan Kaufmann Publishers, San Fransisco, CA.

Goldberg, D. E. (1989). Genetic algorithms and Walsh functions: Part I, a gentle introduction. *Complex Systems*, *3*(2), 129–152. (Also TCGA Report 88006).

Goldberg, D. E. (1993). Making genetic algorithms fly: A lesson from the Wright Brothers. *Advanced Technology for Developers*, *2*, 1–8.

Goldberg, D. E., Deb, K., Kargupta, H., & Harik, G. (1993). Rapid, accurate optimization of difficult problems using fast messy genetic algorithms. *Proceedings of the Fifth International Conference on Genetic Algorithms*, 56–64.

Goldberg, D. E., Deb, K., & Korb, B. (1990). Messy genetic algorithms revisited: Studies in mixed size and scale. *Complex Systems*, *4*, 415–444.

Goldberg, D. E., Korb, B., & Deb, K. (1989). Messy genetic algorithms: Motivation, analysis, and first results. *Complex Systems*, *3*(5), 493–530. (Also TCGA Report 89003).

Goldberg, D. E., & Lingle, Jr., R. (1985). Alleles, loci, and the traveling salesman problem. In *Proceedings of an International Conference on Genetic Algorithms and Their Applications* (pp. 154–159).

Harik, G. R. (1997). *Learning gene linkage to efficiently solve problems of bounded difficulty using genetic algorithms.* Unpublished doctoral dissertation, University of Michigan, Ann Arbor. Also IlliGAL Report No. 97005.

Janikow, C. Z., & Michalewicz, Z. (1991). An experimental comparison of binary and floating point representations in genetic algorithms. In *Proceedings of the Fourth International Conference on Genetic Algorithms* (pp. 31–36).

Kargupta, H. (1995). *SEARCH, polynomial complexity, and the fast messy genetic algorithm* (Technical Report 95008). Urbana, IL: University of Illinois at Urbana-Champaign.

Kargupta, H., Deb, K., & Goldberg, D. E. (1992). Ordering genetic algorithms and deception. In *Parallel Problem Solving from Nature - PPSN II* (pp. 47–56).

Louis, S. J., & Rawlins, G. J. E. (1991). Designer genetic algorithms: Genetic algorithms in structure design. In *Proceedings of the Fourth International Conference on Genetic Algorithms* (pp. 53–60).

Norman, B., & Bean, J. (1997). A random keys genetic algorithm for job shop scheduling. *Engineering Design and Automation*, *3*, 145–156.

Pelikan, M., Goldberg, D. E., & Cantú-Paz, E. (1999). BOA: The Bayesian optimization algorithm. In *GECCO-99: Proceedings of 1999 Genetic and Evolutionary Computation Conference*, Volume 1 (pp. 525–532).

Schwefel, H. P. (1995). *Evolution and optimum seeking.* Sixth-Generation Computer Technology Series. New York: John Wiley & Sons, Inc.

Thierens, D., & Goldberg, D. E. (1993). Mixing in genetic algorithms. In *Proceedings of the Fifth International Conference on Genetic Algorithms* (pp. 38–45).

Whitley, D., & Yoo, N.-W. (1994). Modeling simple genetic algorithms for permutation problems. In Whitley, L. D., & Vose, M. D. (Eds.), *Foundations of Genetic Algorithms 3* (pp. 163–184). San Francisco, California: Morgan Kaufmann Publishers, Inc.