The modular genetic algorithm: exploiting regularities in the problem space

Ozlem O. Garibay, Ivan I. Garibay and Annie S. Wu
{ozlem,igaribay,aswu}@cs.ucf.edu

University of Central Florida
School of Electrical Engineering and Computer Science
P.O. Box 162362, Orlando, FL 32816-2362

Abstract. We introduce the modular genetic algorithm (MGA). The modular genetic algorithm is a search algorithm designed for a class of problems pervasive throughout nature and engineering: problems with modularity and regularity in their solutions. We hypothesize that genetic search algorithms with explicit mechanisms to exploit regularity and modularity on the problem space would not only outperform conventional genetic search, but also scale better for this problem class. In this paper we present experimental evidence in support of our hypothesis. In our experiments, we compare a limited version of the modular genetic algorithm with a canonical genetic algorithm (GA) applied to the checkerboard-pattern discovery problem for search spaces of sizes 2^{32}, 2^{128}, and 2^{512}. We observe that the MGA significantly outperforms the GA for high complexities. More importantly, while the performance of the GA drops 22.50% when the complexity of the problem increases, the MGA performance drops only 11.38%. These results indicate that the MGA has a strong scalability property for problems with regularity and modularity in their solutions.

1 Introduction

According to the NFL theorem (Wolpert & Macready 1995), a search algorithm that outperforms every other on every problem does not exist. What does exist, however, are algorithms that are very good for a particular class of problems. Modularity, or the concept of forming and reusing high-level building blocks from lower-level building blocks, is pervasive in nature: atoms form molecules, molecules are the basic components of cellular organization, cells form organs, and so on. Most importantly, modularity and regularity are also present in the way we organize our ideas—books are made of chapters, chapters of paragraphs and paragraphs of sentences—; and the way we conceive solutions—designs are made of components and subcomponents; computational systems are made of programs, programs are made of reusable routines, etc. Due to the pervasiveness of modularity and regularity in the way that nature self-organizes and also in the way that we tend to engineer solutions to problems, we believe that it is important to investigate algorithms that explicitly work on modular and regular search spaces.

The concept of modularity has been studied extensively in complex systems design and analysis in general, and in evolutionary computation in particular (Happel & Murre 1994, O'Reilly 1996, Angeline & Pollack 1993, Boers & Kuiper 1992, Wagner 1995, Angeline & Pollack 1994, DeJong & Oates 2002). Recently, there has been a surge of interest in modularity in the evolutionary computation community as a way to improve the "innovativeness" and the scalability of evolutionary search. For instance see (Garibay & Wu 2003, Koza, Streeter & Keane 2003, Hu, Goodman et al. 2003). In our perspective, modularity not only implies the hierarchical organization of components from one level of complexity into the next, but also the ability to freely reuse components. It is this component reuse that gives rise to regular and repetitive organizational patterns.

Evolutionary algorithms, such as genetic algorithm are thought to be good at processing building blocks to assemble solutions (Holland 1975, Goldberg 1989). In this paper, we introduce an evolutionary algorithm explicitly designed to work in highly regular and modular search spaces. We call this algorithm the *Modular Genetic Algorithm* (MGA). We envision the MGA as an algorithm that extends the GA's ability to work with building blocks by explicitly encouraging the formation of modules and the reuse of these modules. To this end, our MGA design accounts for two important features: the use of a representation that exploits regular and modular search spaces, and the use of mechanisms for automatic module discovery and encapsulation. The MGA implemented for this paper uses only the first feature. We leave the automatic module discovery and encapsulation for future work on the MGA.

The objective of this paper is to compare the performance and scalability of evolutionary algorithms with and without modularity-aware representations when applied to a modular and regular problem: the checkerboard-pattern discovery problem. We hypothesize that for a regular and modular problem space, evolutionary algorithms with explicit mechanism to handle regularity and modularity should outperform evolutionary algorithms not explicitly designed to do so. More importantly, we also expect that for these kinds of problems, modularity-aware algorithms scale to higher complexities better than algorithms with no modularity-awareness. As an initial test of our hypothesis, we compare the simple GA with no explicit modularity with an MGA with a representation explicitly designed to exploit regularities on the problem space. We compare these algorithms on three instances of the checkerboard-pattern discovery problem of increasing complexity. Experimental studies show that our hypothesis holds true for this problem. In effect, we observe that the MGA significantly outperforms the GA for higher complexity intances of our test problem. While the performance of the GA drops 22.50% when the complexity of the problem increases, the MGA performance drops only 11.38%. These results seem to indicate that the MGA has a strong scalability property for problems with regularity and modularity on their solutions.

2 The checkerboard-pattern discovery problem

After a brief survey in the evolutionary computation community, we were unable to find a widely accepted benchmark problem for algorithms that exploit regularity and modularity. As a result we design our own: the *scalable checkerboard-pattern discovery problem*. The objective of this problem is to discover the target pattern of white and black squares arranged in a checkerboard of size $N \times M$. The fitness evaluation for a candidate solution is simply the number of correctly matched (white or black) squares. A solution with a perfect match achieves a maximum fitness of $N \times M$ while a solution with no matches to the target has a fitness of zero. In Figure 1 (A) and (B), we see an example of a target and a candidate solution with 20 wrongly placed squares and a sub-optimal fitness of $8 \times 16 - 20 = 108$. . Figure 1 (C) and (D) shows and example of a target and a candidate solution with a perfect match. We designed this simple problem to explicitly provide an environment rich in regularity and modularity (handcrafted patterns) to be our scalable initial test problem for the Modular GA as seen in Figure 1. We can scale up the complexity of these problems by simply increasing

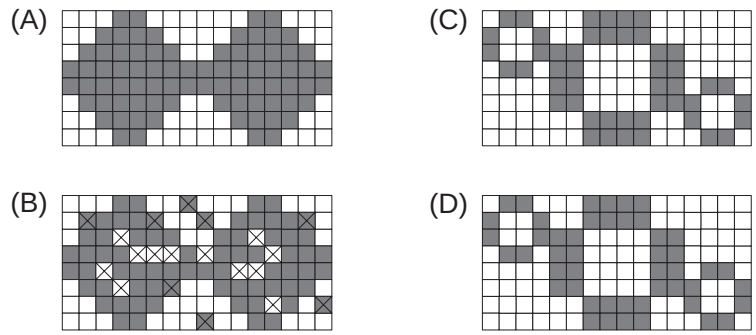

Fig. 1. Examples of target **(A)** and candidate **(B)** solutions for a checkerboard-pattern discovery problem of size 8×16. The fitness of the candidate solution is 108 because there are 20 un-matched squares (marked with an X). The maximum fitness possible for this instance is $8 \times 16 = 128$. Target **(C)** and candidate with optimal solution **(D)** for a checkerboard-pattern discovery problem of size 8×16.

the size of the checkerboard. We can increase the complexity but keep constant the amount of regularity by keeping the number of components in each pattern constant. Figure 2 shows the three instances of the checkerboard-pattern discovery problem used on this paper: 4x8(A), 8x16(B), and 16x32(C). The search space for this three problems increase exponentially in size 2^{32}(A), 2^{128}(B), and 2^{512}(C), while the amount of regularity on the pattern is kept constant: four black and four white square areas.

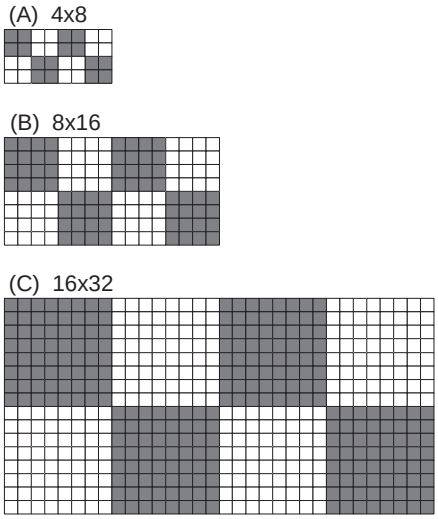

Fig. 2. The target pattern for the checkerboard-pattern discovery problem used in all experiments on this paper. This pattern is scaled up in size 4x8(A), 8x16(B), and 16x32(C), to increase the complexity of the problem while keeping the pattern constant: four black and four white big squares. The search space's sizes increase exponentially: 2^{32}(A), 2^{128}(B), and 2^{512}(C).

3 Comparing representations with and without explicit modularity awareness

3.1 No modularity: Genetic Algorithm

As an example of an evolutionary algorithm with no explicit mechanisms to exploit modularity on the search space we use a simple binary genetic algorithm. The encoding of the problem is shown in Figure 3 (A). We represent each square of the board with one bit: one for black and zero for white.

3.2 Repetitive modularity: Modular Genetic Algorithm

We consider two features to be key to for the MGA's success: it needs a representation that exploits regular and modular problem spaces, and it needs mechanisms for automatic module discovery and encapsulation. In this paper we study an MGA with only the first feature. A simple implementation of such representation is described below. A MGA genome is a string of genes: MGA-Genome = $\text{gene}_1\text{gene}_2 \ldots \text{gene}_i$ with each gene being of the form: $gene_i = \langle \text{number-of-repetitions}_i, \text{function}_i() \rangle$ and gene_i being interpreted as: $gene_i = \underbrace{\text{function}_i()\text{function}_i() \ldots \text{function}_i()}_{\text{number-of-repetitions}_i}$ random This is a functional representation

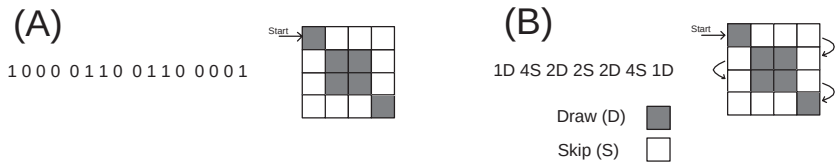

Fig. 3. (A) GA encoding for the checkerboard problem. Ones indicate black squares; zeros indicate white squares.**(B)** MGA encoding for the checkerboard problem. Two functional predefined primitives are used: Draw "D" and Skip "S". They are preceded by their repetition number. The path to draw the board is fixed as indicated by the arrows.

with each gene being prefixed by its gene expression value. The gene expression value specifies the number of times a gene is expressed or executed. This representation requires a set of initial predefined functions. For the checkerboard-pattern discovery problem, the set of initial functions is $\{D(), S()\}$. The *draw* function, D(), paints the current square black then moves to the next square. The *skip* function, S(), skips to the next square without painting any square. Functions are prefixed by their number of repetitions. For example the string $2D4S1D2S$ is interpreted as $DDSSSSDSS$, where 2D = DD, 4S = SSSS, and so on. The path to follow in the checkerboard is predefined, hence there is no need for directional primitives such as right() or left(). As a result, the draw() and skip() primitives are sound and complete for our checkerboard-pattern discovery problem. A MGA representation of the checkerboard problem is shown in Figure 3 (B).

4 Experiments

4.1 Objectives

We compare the GA with the Modular GA and observe if the simple MGA implementation has a positive impact on the search performance and scalability for the checkerboard-pattern discovery problem. We perform three experiments comparing this two algorithms with the following problem sizes: 4×8, 8×16, and 16×32. The target pattern for all the runs is the same and it is shown in Figure 2. The following parameters are kept constant throughout all the experiments: 20 runs, two-point crossover, crossover rate of 0.7, mutation rate of 0.001, standard fitness proportional selection, and a population size of 1000. Each experiment was run until population convergence: 500 generations for the 4×8 board size, 1000 generations for the 8×16, and 2000 generations for the 16×32. Initialization is performed at random: a random binary string for the GA and random string of genes for the MGA. The GA uses a standard binary alphabet. Each random gene for the MGA is a repetition number randomly picked from the interval $[0, N/2]$ (for a problem of size $N \times 2N$). This random repetition number is paired with a

symbol randomly picked from the MGA alphabet $\{S, D\}$. The GA uses bit-flip mutation. Mutation for the MGA changes both parts of the gene: the number of repetitions and the symbol from its alphabet. Crossover for the GA is canonical, for the MGA crossover is allowed only to act between genes with no gene splitting allowed.

4.2 Results

The performance is measured as the best fitness obtained by an algorithm on its final generation averaged over 20 runs. Fitness is computed as described on Section 2. Complexity is measured in terms of the size of the space being searched by the algorithms. Figure 4 shows that the GA and the MGA perform similarly for problems of size $4\times8, 8\times16$, but the MGA significantly outperforms the GA on the high complexity case: problem of size 16×32. Figure 5 plots the performance versus complexity. We observe that, for the GA, this curve drops sharply in the last increase in complexity, while, for the MGA, the curve decreases more steadily. This plot indicates that the MGA scales better than the GA for this problem. In fact, while GA performance decreases by 22.50% from the 4×8 to the 16×32 problem, the MGA performance decreases by only half that amount, 11.38%, as shown in Table 1.

	Performance Drop (% of fitness decrease)	
Complexity increase (in search space size)	**GA**	**MGA**
from $2^{4\times8}$ to $2^{8\times16}$	3.48%	3.68%
from $2^{4\times8}$ to $2^{16\times32}$	22.50%	11.38%

Table 1. Percentages of performance drops for the GA and MGA as the complexity of the problem is increased. GA and MGA both drop their performances around 3.5% on a increase of search space size from $2^{4\times8}$ to $2^{8\times16}$. Interestingly, for the increase of search space size from $2^{4\times8}$ to $2^{16\times32}$ the GA drops in performance a substantial 22% while the MGA drop only around half of that amount: 11%.

5 Conclusions

We introduce the *Modular Genetic Algorithm* in this paper. This algorithm is designed for problem spaces with a high degree of modularity and regularity. We hypothesize that the MGA will outperform conventional genetic search and scale better for the class of problems with significant degree of modularity and regularity on their solutions. We provide experimental results that support our

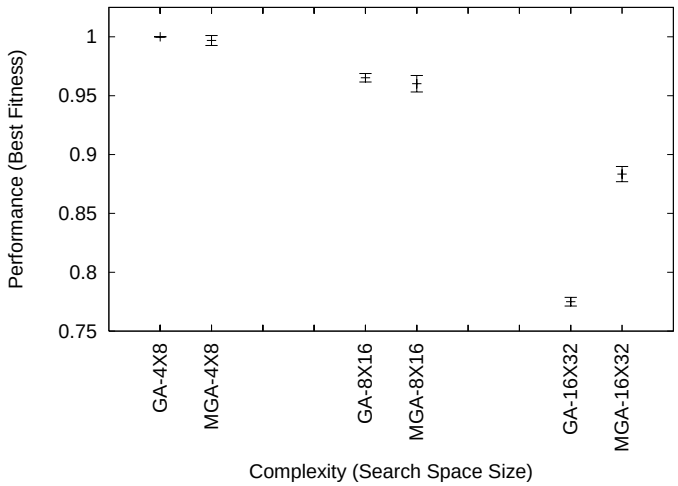

Fig. 4. GA vs. MGA performance comparison I: performance is measured as the best fitness of the last generation averaged over 20 runs and shown with 95% confidence intervals. (Left) GA vs. MGA on the checkerboard-pattern discovery problem of size 4×8. (Center) GA vs. MGA for the same problem size of 8×16 and (Right) for size 16×32. MGA and GA have equivalent performance for low and medium complexity, but MGA significantly outperform GA on the high complexity case.

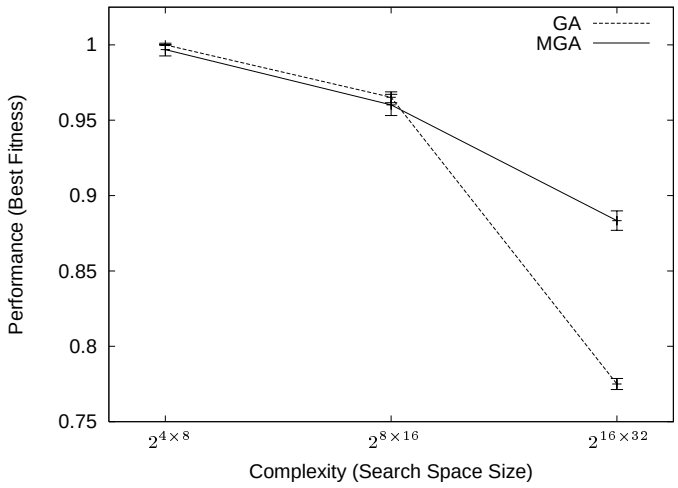

Fig. 5. GA vs. MGA performance comparison II: performance is measured as the best fitness of the last generation averaged over 20 runs and shown with 95% confidence intervals. The performance of the GA decreases sharply as complexity increases. The performance of the MGA decreases less dramatically. This graphic suggests that the MGA scales better than the GA for the checkerboard-pattern discovery problem.

hypothesis on a well defined problem: the checkerboard-pattern discovery problem. Our results show that the MGA significantly outperforms a basic GA for the highest complexity case tested (16×32). For the lower complexity cases (4×8 and 8×16) their performance are comparable. These results indicate that the MGA scales much better than the GA for this problem class. We find these first results very encouraging and revealing. A simple change from a binary representation that exhaustively enumerates every element of the solution being described, to a more functional representation geared towards creating modules and exploiting repetitions can bring a large benefit in terms of both performance and scalability to the evolutionary search.

References

Angeline, P. J. & Pollack, J. (1993), Evolutionary module adquisition, *in* 'Proceedings of the second annual conference on evolutionary programming', pp. 154–163.

Angeline, P. J. & Pollack, J. (1994), Coevolving high-level representations, *in* 'Proceedings of Artificial Life III', pp. 55–71.

Boers, E. J. W. & Kuiper, H. (1992), Biological metaphors and the design of modular artificial neural networks, Master's thesis, Leiden University.

DeJong, E. D. & Oates, T. (2002), A coevolutionary approach to representation development, *in* 'Proc. of the ICML-2002 WS on develop. of repres.', p. ??

Garibay, I. I. & Wu, A. S. (2003), Cross-fertilization between proteomics and computational synthesis, *in* '2003 AAAI Spring Symposium Series', pp. 67–74.

Goldberg, D. E. (1989), *Genetic algorithms in search, optimization, and machine learning*, Addison Wesley.

Happel, B. L. M. & Murre, J. M. J. (1994), 'The design and evaluation of modular neural network architectures', *Neural Networks* **7**, 985–1004.

Holland, J. H. (1975), *Adaptation in Natural and Artificial Systems*, University of Michigan Press, Ann Arbor, MI.

Hu, J., Goodman, E. et al. (2003), HFC: a continuing EA framework for scalable evolutionary synthesis, *in* '2003 AAAI Spring Symposium Series', pp. 106–113.

Koza, J., Streeter, M. & Keane, M. (2003), Automated synthesis by means of genetic programming, *in* '2003 AAAI Spring Symposium Series', pp. 138–145.

O'Reilly, U. (1996), Investigating the generality of automatically defined functions, *in* 'Proceedings of the first annual conference on genetic programming', pp. 351–356.

Wagner, G. P. (1995), Adaptation and the modular design of organisms, *in* 'Proceedings of the european conference on artificial intelligence', pp. 317–328.

Wolpert, D. H. & Macready, W. G. (1995), No free lunch theorems for search, Technical Report SFI-TR-95-02-010, The Santa Fe Institute, Santa Fe, NM.