The Compact Genetic Algorithm

Georges R. Harik
Fernando G. Lobo
David E. Goldberg

University of Illinois at Urbana-Champaign
Urbana, IL 61801

IlliGAL Report No. 97006
August 1997

Illinois Genetic Algorithms Laboratory
Department of General Engineering
University of Illinois at Urbana-Champaign
117 Transportation Building
104 South Mathews Avenue
Urbana, IL 61801

The Compact Genetic Algorithm

Georges R. Harik, Fernando G. Lobo, and David E. Goldberg
Illinois Genetic Algorithms Laboratory
University of Illinois at Urbana-Champaign

Abstract

This paper introduces the compact genetic algorithm (cGA). The cGA represents the population as a probability distribution over the set of solutions, and is operationally equivalent to the order-one behavior of the simple GA with uniform crossover. It processes each gene independently and requires less memory than the simple GA. Therefore, it can be used to give a quick estimate of a problem's difficulty.

In addition, this work raises important questions about the use of information in a genetic algorithm, and its ramifications show us a direction that can lead to the design of more efficient GAs.

1 Introduction

There is a tendency in the community of evolutionary computation to treat the population with almost mystical reverence, and certainly the population deserves our respect as it is the source of all that goes right (or wrong) in a genetic algorithm (GA) with respect to function evaluation, schema evaluation, and partition identification (Kargupta & Goldberg, 1996). But if one lesson is clear from the history of GA analysis and design, it is that genetic algorithms are complex objects and multiple perspectives are useful in understanding what they can and cannot do.

In this paper, we take a minimalist view of the population and create a GA that mimics the order-one behavior of a simple GA using a finite memory bit by bit. Although the resulting compact genetic algorithm (cGA) is not intended to replace population-oriented GAs, it does teach us important lessons regarding GA memory and efficiency. Moreover, as a practical matter, the resulting algorithm can act as a filter to discriminate whether problems are easy or hard. As a matter of design, the cGA shows us an interesting way of getting more information out of a finite set of evaluations.

We start by discussing the inspiration of this work from a random walk model that has been recently proposed. Then, we present the cGA and describe the mapping of the sGA's parameters into those of an equivalent cGA. Along the way, computer simulations compare the two algorithms, both in terms of solution quality and speed. At the end of the paper, important ramifications are outlined concerning the design of more efficient simple GAs.

2 Motivation and related work

This work is primarily inspired by the random walk model introduced by Harik, Cantú-Paz, Goldberg, and Miller (1997). In that work, the authors gave accurate estimates of the GA's convergence on a special class of problems: problems consisting of non-overlapping building blocks (BBs) of

bounded difficulty. Because there are no interactions among BBs, the authors made the assumption that the BBs could be solved independently. Therefore, they focused on one BB (partition) at a time. The next paragraph describes the basic idea of the model.

In the initial population, there will be some instances of the correct BB. Then, during the action of a GA run, the number of instances of the correct BB can increase or decrease. Eventually, the correct BB will spread throughout all the population members or it will get extinct.

This type of process is easily modeled using a random walk as a mathematical tool. Using such a model, Harik et al., were able to accurately predict the GA's convergence. There, the random walk variable represents the number of BBs in the population at a given time. Two absorbing barriers (one at 0 and one at the population size) represent the success or failure in the overall decision of the GA. The transition probability of the random walk is given by the probability that the GA commits a decision error on two competing schemata. This error in decision making occurs because a schema is always evaluated within the context of a larger individual. The GA can make an incorrect decision in a partition because of the noise coming from the remaining partitions. In the model, the population plays the role of a memory to account for a finite number of such decision-errors.

The dynamics of the random walk model suggests that it is possible to directly simulate its behavior for order-1 problems. The idea is to simulate ℓ independent random walks bit by bit. The next section, which introduces the cGA, shows how this is possible. The cGA represents the population as a probability distribution over the set of solutions. By discretizing its probability representation, the proposed algorithm reduces the GA's memory requirements. In addition, the manner in which the cGA manipulates this distribution allows it to mimic the order-1 behavior of the simple genetic algorithm (sGA). But before introducing the cGA, let's review other related works.

Ackley (1987) introduced a learning algorithm that manipulates a gene vector via positive and negative feedback coming from the population members. To describe his algorithm, he used a political metaphor where the voters (population) express their satisfaction or dissatisfaction towards an ℓ-member government (a point in the search space).

Syswerda (1993) introduced an operator called bit-based simulated crossover (BSC) that uses the statistics in the GA's population to generate offspring. BSC does a weighted average of the alleles of the individuals along each bit position (a bit column). By using the fitness of the individuals in this computation, BSC integrates the selection and crossover operators into a single step. A variation of BSC was also discussed by Eshelman and Schaffer (1993) in the context of investigating how GAs differ from population-based hillclimbers.

Population-based incremental learning (PBIL) was introduced by Baluja (1994) (Baluja & Caruana, 1995) As opposed to storing the whole population as in BSC, PBIL uses a probability vector over the chromosome to represent its population. Specifically, it records the proportion of ones (and consequently zeroes) at each gene position. These probabilities are initially set to 0.5 and move towards 0 or 1 as the search progresses. The probability vector is used to generate new solutions, and thus represents the combined experiences of the PBIL algorithm at any one time. Using the probability vector, PBIL generates a certain number of solutions, and updates the vector based on the fitnesses of these solutions. The aim of this update is to move the probability vector towards the fittest of the generated solutions. The update rule is similar to that used in learning vector quantization (Hertz, Krogh, & Palmer, 1993). The number of individuals generated, the number of individuals to update from, the stopping criterion, and the rate of the probability vector's change are all parameters of the algorithm. Attempts were made to relate PBIL's parameters to the simple GA. For instance, the number of samples generated was equated with the GA's population size. These attempts were not successful because the GA manipulates its distributions in a different way.

The following section, which introduces the compact GA, shows how this is possible in a related algorithm.

3 The Compact Genetic Algorithm

Harik, Cantú-Paz, Goldberg, and Miller (1997) analyzed the growth and decay of a particular gene in the population as a one-dimensional random walk. As the GA progresses, genes fight with their competitors, and their number in the population can go up or down, depending on whether the GA makes good or bad decisions. These decisions are made implicitly by the GA when selection takes place. The next section explores the effects of this decision making.

3.1 Selection

Selection gives more copies to better individuals. But it does not always do so for better genes. This is because genes are always evaluated within the context of a larger individual. For example, consider the onemax problem (that of counting ones). Suppose individual a competes with individual b.

individual	chromosome	fitness
a	1011	3
b	0101	2

When these two individuals compete, individual a will win. However, at the level of the gene, a decision error is made on the second position. That is, selection incorrectly prefers the schema *0** to *1**. The role of the population is to buffer against a finite number of such decision errors.

Imagine the following selection scheme: pick two individuals randomly from the population and keep two copies of the better one. This scheme is equivalent to a steady-state binary tournament selection. In a population of size n, the proportion of the winning alleles will increase by $1/n$. For instance, in the previous example the proportion of 1's will increase by $1/n$ at gene positions 1 and 3, and the proportion of 0's will also increase by $1/n$ at gene position 2. At gene position 4, the proportion will remain the same. This thought experiment suggests that an update rule increasing a gene's proportion by $1/n$ simulates a small step in the action of a GA with a population of size n.

The next section explores how the generation of individuals from a probability distribution mimics the effects of crossover.

3.2 Crossover

The role of crossover in the GA is to combine bits and pieces from fit solutions. A repeated application of most commonly used crossover operators eventually leads to a decorrelation of the population's genes. In this decorrelated state, the population is more compactly represented as a probability vector. Thus the generation of individuals from this vector can be seen as a shortcut to the eventual aim of crossover. Figure 1 gives pseudocode of the compact GA.

```
1) initialize probability vector
   for i := 1 to ℓ do p[i] := 0.5;

2) generate two individuals from the vector
   a := generate(p);
   b := generate(p);

3) let them compete
   winner, loser := evaluate(a, b);

4) update the probability vector towards the better one
   for i := 1 to ℓ do
      if winner[i] ≠ loser[i] then
         if winner[i] = 1 then p[i] := p[i] + 1/n
                          else  p[i] := p[i] − 1/n;

5) check if the vector has converged
   for i := 1 to ℓ do
      if p[i] > 0 and p[i] < 1 then
         return to step 2;

6) p represents the final solution


compact GA parameters
   n: population size.
   ℓ: chromosome length.
```

Figure 1: pseudocode of the compact GA

3.3 Two main differences from PBIL

The proposed algorithm differs from PBIL in two ways: (1) it can simulate a GA with a given population size, and (2) it reduces the memory requirements of the GA.

The update step of the compact GA has a constant size of $1/n$. While the simple GA needs to store n bits for each gene position, the compact GA only needs to keep the proportion of ones (and zeros), a finite set of n numbers that can be stored with $\log_2 n$ bits. With PBIL's update rule (for details see Baluja and Caruana (1995)), an element in the probability vector can have any arbitrary precision, and the number of values that can be stored in an element of the vector is not finite. Therefore, PBIL can't achieve the same level of memory compression as the cGA.

PBIL typically generates a large number of individuals from the probability vector. According to Baluja and Caruana (1995) that number was something analogous to the population size. In the compact GA, the size of the update step is the "thing" that is analogous to the population size.

4 Experimental results

This section presents simulation results and compare the compact GA with the simple GA, both in terms of solution quality and in the number of function evaluations taken. All experiments are averaged over 50 runs. The simple GA uses binary tournament selection without replacement, and uniform crossover with exchange probability 0.5. Mutation is not used, and crossover is applied all the time. All runs end when the population fully converges—that is—when all the individuals have the same alleles at each gene position. Figures 2 and 3 show the results of the experiments on a 100-bit onemax problem (the counting ones problem). Figure 2 plots the solution quality (number of correct bits at the end of the run) for different population sizes. Figure 3 plots the number of function evaluations taken until convergence for the various population sizes. On both graphs, the solid line is for the simple GA, and the dashed line is for the compact GA. Additional simulations were performed with the binary integer function, and with De Jong's test functions (De Jong, 1975). The results obtained were similar to these, and are collected in appendix A. The match between the two algorithms seems accurate, and gives evidence that the two are doing roughly the same thing and that they are somehow "equivalent". Note however that while the sGA has a memory requirement of $n * \ell$ bits, the cGA requires only $\log_2 n * \ell$ bits.

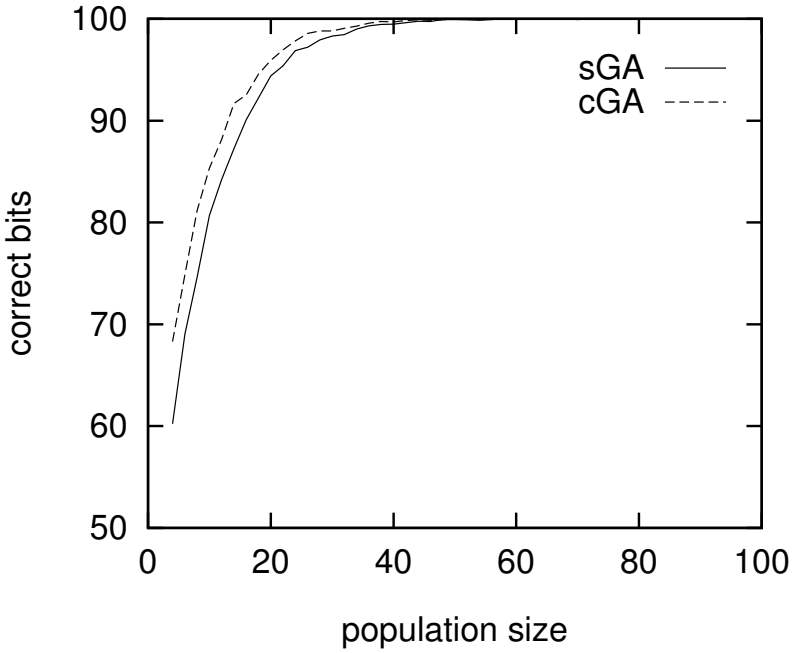

Figure 2: Comparison of the solution quality (number of correct bits at the end of the run) achieved by the compact GA and the simple GA on a 100-bit onemax problem. The solid line is for the simple GA. The dashed line is for the compact GA.

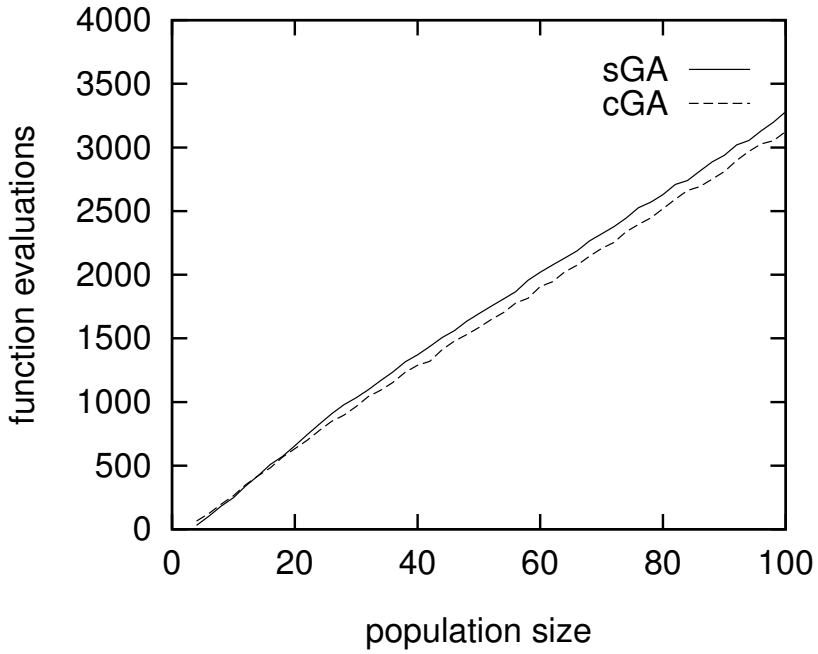

Figure 3: Comparison of the compact GA and the simple GA in the number of function evaluations needed to achieve convergence on a 100-bit onemax problem. The solid line is for the simple GA. The dashed line is for the compact GA.

5 Simulating higher selection pressures

This section introduces a modification to the compact GA that allows it to simulate higher selection pressures. We would like to simulate a tournament of size s. The following mechanism produces such an effect: (1) generate s individuals from the probability vector and find out the best one. (2) let the best individual compete with the other $s-1$ individuals, updating the probability vector along the way. Clearly, the best individual wins all the competitions, thus the above procedure simulates something like a tournament of size s. Experiments on the onemax problem with $s = 2, 4$ and 8 are shown in figure 4 confirming our expectations. Once more, the graphs show the solution quality and also the number of function evaluations needed to reach convergence. The runs were done for different population sizes and the results agree with the predictions of the population sizing theory (Harik, Cantú-Paz, Goldberg, & Miller, 1997). The top graphs are for $s = 2$, the middle ones for $s = 4$, and the bottom ones are for $s = 8$. In all of them, the solid line is for the simple GA, and the dashed line is for the compact GA.

Being able to simulate higher selection rates should allow the compact GA to solve problems with higher order building blocks in approximately the same way that a simple GA with uniform crossover does. It is known that in order to solve such problems, high selection rates are needed to compensate for the highly disruptive effects of crossover. Moreover, the population size required to solve such problems grows exponentially with the problem size (Thierens & Goldberg, 1993). To test the compact GA on problems with higher order building blocks, 10 copies of a a 3-bit deceptive sub-function are concatenated to form a 30-bit problem. Each sub-function is a 3-bit trap function with deceptive-to-optimal ratio of 0.7 (Ackley, 1987), (Deb & Goldberg, 1993). The results are presented in figure 5.

In this case there is a discrepancy between the two algorithms. This can be explained on schema theorem grounds. Using uniform crossover, an order-k BB has a survival probability of 2^{1-k}. According to the schema theorem, the simple GA should be able to propagate these BBs as long as the selection rate is high enough to compensate for the crossover disruption. For an order-3 schema, the survival probability is 1/4, so the sGA should start to work well when the selection rate is greater than 4. In the case of the cGA, we can think of a global schema theorem. The survival probability of a schema H under the cGA would then be given by:

$$P(survival\ of\ H) = \prod_{i \in H} p_i$$

In the cGA, all the p_i start with 1/2. This means that initially the survival probability of an order-3 BB is 1/8. Therefore, the BB should grow when the selection rate is greater than 8. This argumentation explains the results obtained in figure 5 (see the cases $s = 4$ and $s = 8$). Observe that a selection rate of $s = 2$ is not enough to combat the disruptive effects of crossover. No matter what population size is used, the compact GA (and also the simple GA) with $s = 2$ will fail to solve this problem. This is an indication that the problem has higher-order building blocks and that it can only be solved with these kind of algorithms by raising the selection pressure.

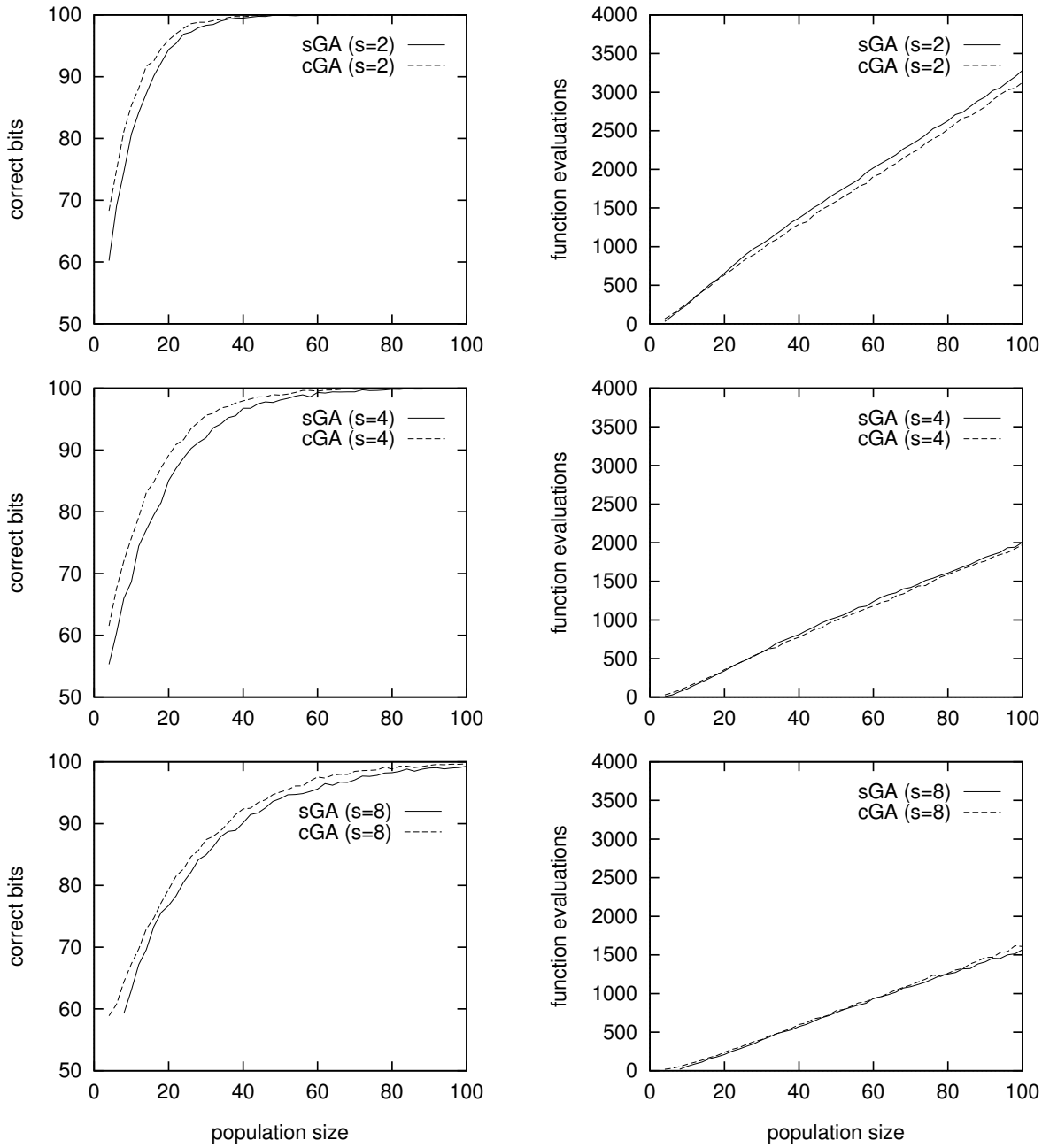

Figure 4: The plots illustrate the mapping of the selection rate of the compact GA into that of the simple GA using the onemax function. Selection rates are 2, 4, and 8. On the left side, the graphs plot the number of correct bits at the end of the run for the various population sizes. On the right side, the graphs plot the number of function evaluations taken. Selection rates are $s = 2$ (top), $s = 4$ (middle), and $s = 8$ (bottom). The solid lines are for the simple GA, and the dashed lines are for the compact GA.

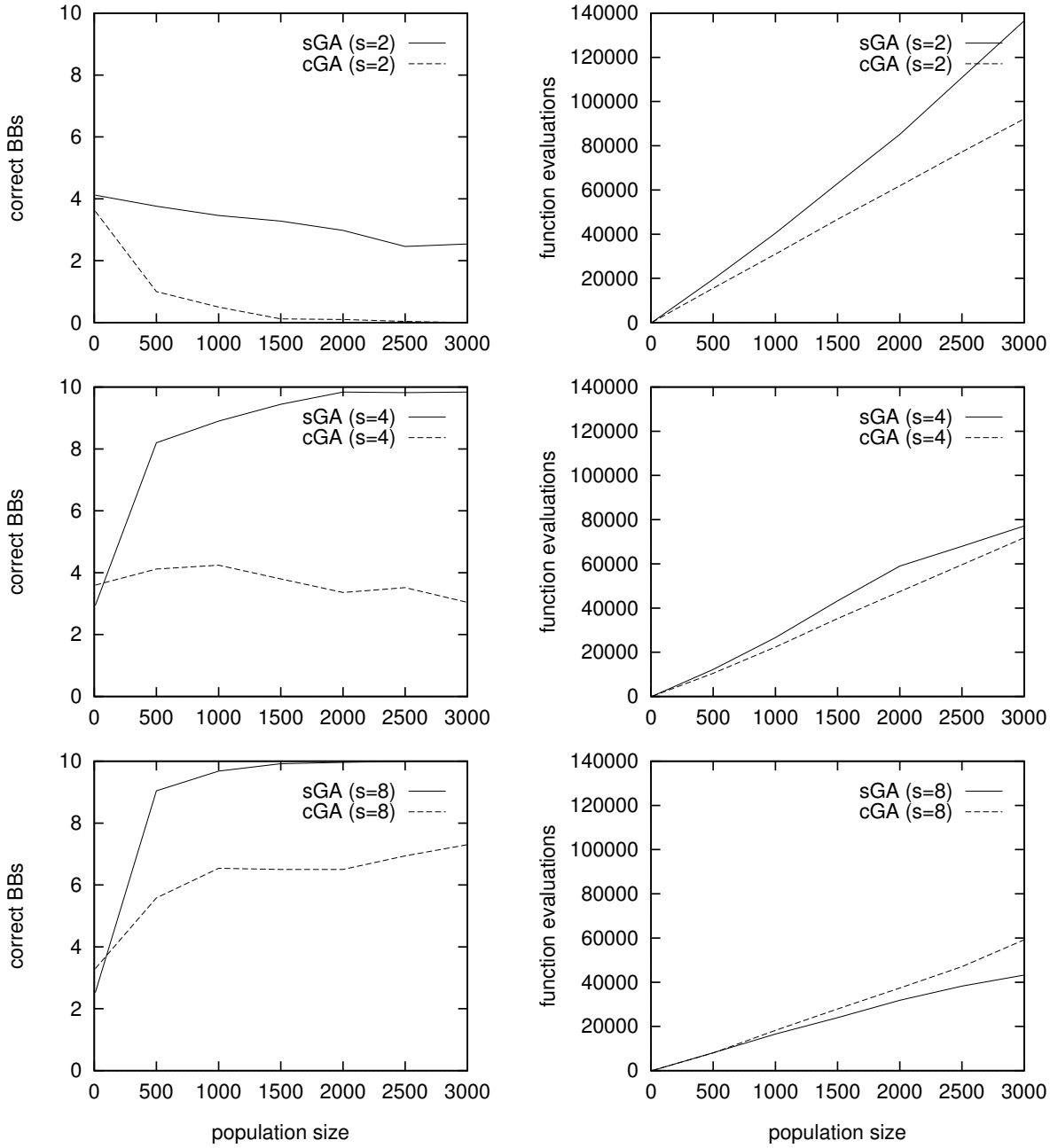

Figure 5: These plots compare the compact GA and the simple GA on the 10 copies of a 3-bit trap function, using selection rates of 2, 4, and 8. On the left side, the graphs plot the number of correct building-blocks (sub-functions) at the end of the run for the various population sizes. On the right side, the graphs plot the number function evaluations taken. Selection rates are $s = 2$ (top), $s = 4$ (middle), and $s = 4$ (bottom). The solid lines are for the simple GA, and the dashed lines are for the compact GA.

6 Getting more with less

This section introduces a concept that is unusual in terms of standard GA practice. To motivate the discussion, let's start with an analogy between the selection operator of a GA and a tennis (or soccer) competition.

In tennis there are two kinds of tournaments: elimination and round-robin. In both cases, the players are matched in pairs. In the elimination case, the losers get out of the tournament, and the winners proceed to next round. In the round-robin variation, everybody plays with everybody. It's also possible to have competitions that are something in between these two. An example is the soccer world cup. There, the teams are divided in groups, and within each group the teams play round-robin. Then, the top-k within each group proceed to the next phase.

After this brief detour, let's shift back to our discussion on genetic algorithms. Typically, a GA using binary tournament selection is very much like an elimination tennis competition. The only difference is that in the GA, each individual participates in two tournaments. This is because we don't want the population to be chopped by a half after each generation. Round-robin competitions are not usually done in GAs, because this would make the population size grow after each generation.

The remaining of this section shows how it is possible to have round-robin like competitions within the compact GA while maintaining the population size fixed. In order to implement it, we do the following: instead of generating two individuals, generate m individuals and make a round-robin tournament among them, updating the probability vector along the way. Steps 2, 3, and 4 of the cGA's pseudocode (figure 1) would have to be replaced by the ones shown in figure 6.

This results in a faster search, because $O(m^2)$ binary tournaments are made using only $O(m)$ function evaluations. On the other hand, this scheme takes bigger steps in the probability vector, and therefore more decision-making mistakes are made. When $m = 2$ the tournaments are played using elimination. When $m = n$ the tournament is played in a round-robin fashion among all the population members. When m is between 2 and n we get something that is neither a pure elimination scheme, nor a pure round-robin scheme.

```
2) generate m individuals from the vector and store them in M
   for i := 1 to m do
       M[i] := generate(p);

3) do the round-robin tournament
   for j := 1 to m − 1 do
       for k := j + 1 to m do
       begin
           winner, loser := evaluate(M[j], M[k]);
           update probability vector (step 4 of cGA code)
       end
```

Figure 6: modification of the compact GA that implements a round-robin tournament. This would replace steps 2, 3, and 4 of the cGA code.

Experiments of the cGA with a selection rate of $s = 2$ are performed again, but this time using different values of m. Plots for the onemax problem are shown in figures 7, 8 and 9. Figure 7 shows the solution quality (number of correct bits at the end of the run) of the compact GA with $m = 2, 4, 8, 40$, for different population sizes. Figure 8 shows the number of function evaluations taken by the compact GA with $m = 2, 4, 8, 40$, for the different population sizes. Figure 9 is a combination of figures 7 and 8. It shows that a given solution quality can be obtained faster by using $m = 4$ or $m = 8$ instead of $m = 2$. In other words, although using higher values of m reduces the solution quality, the corresponding increase in speed makes it worth its while. Observe that after a certain point, it's not worth to increase m. In this example, using $m = 40$ is worse than using $m = 4$ or $m = 8$. This shows that there must be an optimal m and raises important questions concerning GA efficiency. Specifically, how to develop a theory of an "optimal m", and most important, how to use such a theory in order to design more efficient simple GAs.

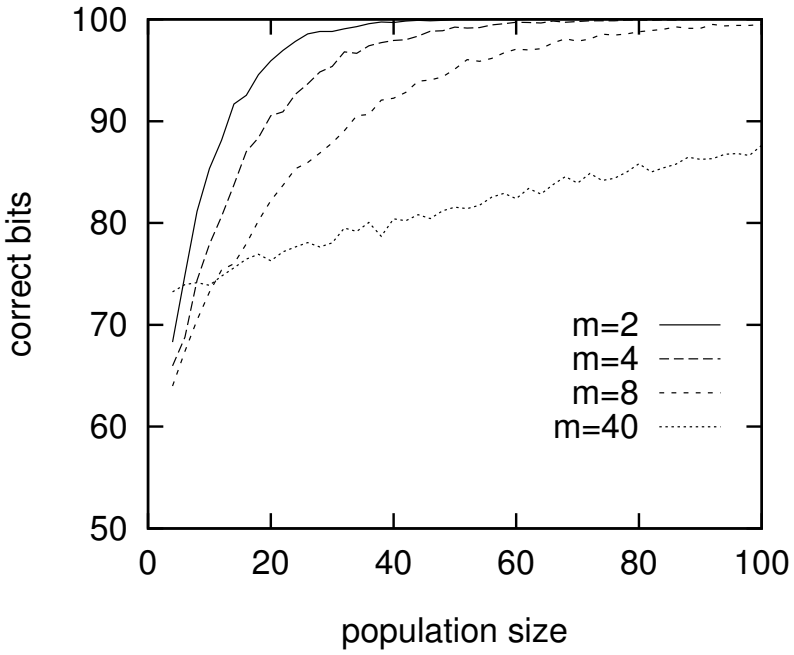

Figure 7: This graph shows the solution quality on a 100-bit onemax problem for various population sizes, using different values of m. Observe that the solution quality decreases as m increases.

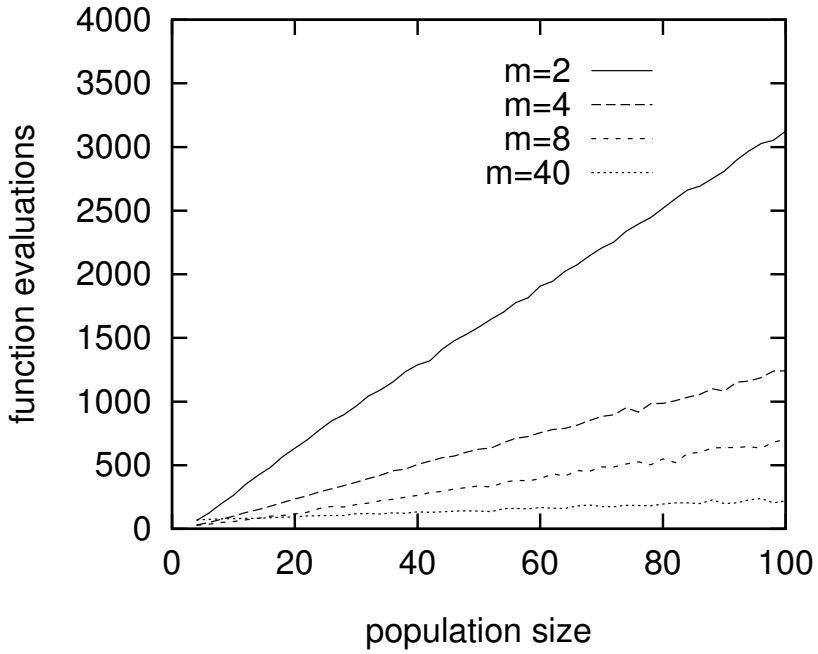

Figure 8: This graph shows the number of function evaluations needed reach convergence on a 100-bit onemax problem, using various population sizes, and different values of m. Observe that the speed increases as m increases.

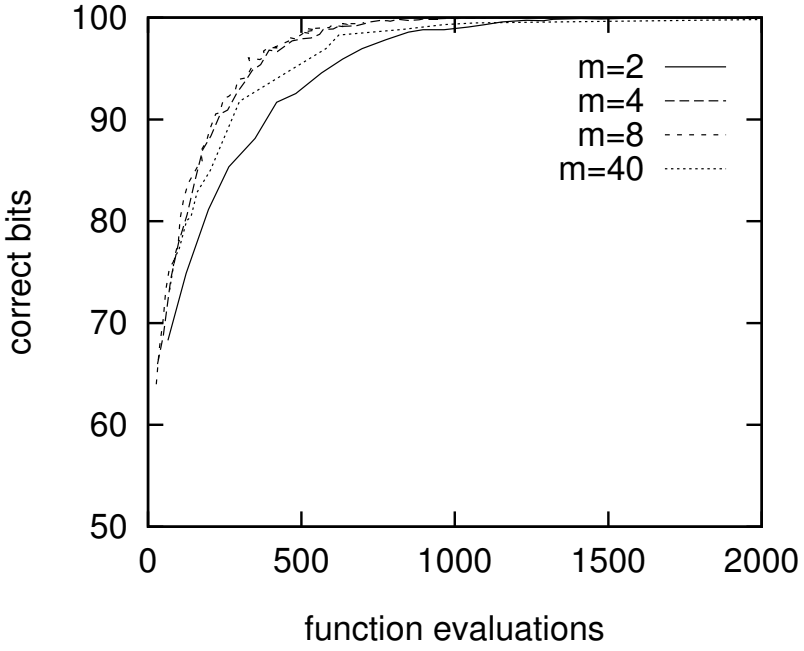

Figure 9: This is a combination of the previous two graphs. It shows that to achieve a given solution quality, it is better to use $m = 4$ or $m = 8$ instead of $m = 2$ or $m = 40$. In other words, the best strategy is neither to use a pure elimination tournament, nor a pure round-robin tournament.

7 Extensions

Two extensions are proposed for this work: (1) investigate extensions of the cGA for order-k problems, (2) investigate how to maximize the information contained in a finite set of evaluations in order to design more efficient GAs.

The compact GA is basically a 1-bit optimizer and ignores the interactions among the genes. The set of problems that can be solved efficiently with such schemes, are problems that are somehow easy. The representation of the population in the compact GA explicitly stores all the order-1 schemata contained in the population. It is possible to have a similar scheme that is also capable of storing higher order schemata in a compact way. Work has already begun on this topic and will be presented in a later report.

Another direction is to investigate more deeply the results discussed in section 6, and find out their implications for the design of more efficient simple GAs. Our preliminary work has shown that it's possible to extract more information from a set of n function evaluations, than the usual information extracted by the simple GA. But how to use this additional information in the context of a simple GA is still an open question and deserves further research.

8 Conclusions

This paper presented the compact GA, an algorithm that mimics the order-one behavior of a simple GA with a given population size and selection rate, but that reduces its memory requirements. The design of the compact GA was explained, and computational experiments illustrated the approximate equivalence of the compact GA with a simple GA using uniform crossover.

Although the compact GA approximately mimics the order-one behavior of the simple GA with uniform crossover, it is not a replacement for the simple GA. Simple GAs perform quite well when the user has some knowledge about the non-linearities in the problem. In that case, the building blocks can be tightly coded, and they can be propagated throughout the population through the repeated action of selection and recombination. Note that in general, this linkage information is not known. However, in most applications the GA user has some knowledge about the problem's domain, and tends to code together in the chromosome, features that are somehow spatially related in the original problem. In a way, the GA user has partial knowledge about the linkage. This is probably one of the main reasons why simple GAs have had so much success in real-world applications.

As a practical consequence of this work, the compact GA can be used as a quick way to check if a given problem is easy or not. A problem is easy if the compact GA can solve it using a low selection rate. As the problem's difficulty increase, higher selection rates must be used.

Finally, this study has introduced new ideas that have important ramifications for GA design. By looking at the simple GA from a different perspective, we learned more about its complex dynamics and opened new doors towards the goal of having more efficient GAs.

Acknowledgments

We thank the comments of Dirk Thierens, Erick Cantú-Paz, and Brad Miller.

This study was sponsored by the Air Force Office of Scientific Research, Air Force Materiel Command, USAF, under grants F49620-94-1-0103, F49620-95-1-0338, and F49620-97-1-0050. The U.S. Government is authorized to reproduce and distribute reprints for Governmental 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 or the U. S. Government.

Appendix A

This appendix presents simulation results comparing the compact GA and the simple GA on the binary integer function, and on De Jong's test functions (De Jong, 1975). All experiments are averaged over 50 runs. The simple GA uses binary tournament selection without replacement, and uniform crossover with exchange probability 0.5. Mutation is not used, and crossover is applied all the time. All runs end when the population fully converges—that is—when all the individuals have the same alleles at each gene position.

The binary integer function is defined as $f = \sum_{i=1}^{\ell} 2^i x_i$. The GA solves this problem in a sequential way (domino-like). First it pays attention to the most significant bits and then, once those bits have converged, it will move on to next most significant bits. For this function, the solution quality is measured by the number of consecutive bits solved correctly.

For De Jong's test functions, the solution quality is measured by the objective function value obtained at the end of the run. Each parameter is coded with the same precision as described in his dissertation (De Jong, 1975). The functions F1-F5 are shown below:

$$f_1(X) = \sum_{i=1}^{3} x_i^2, \qquad -5.12 \leq x_i \leq 5.12$$

$$f_2(X) = 100(x_1^2 - x_2)^2 + (1 - x_1)^2, \qquad -2.048 \leq x_i \leq 2.048$$

$$f_3(X) = \sum_{i=1}^{5} integer(x_i), \qquad -5.12 \leq x_i \leq 5.12$$

$$f_4(X) = \sum_{i=1}^{30} i x_i^4 + Gauss(0,1), \qquad -1.28 \leq x_i \leq 1.28$$

$$f_5(X) = \left[0.002 + \sum_{j=1}^{25} \frac{1}{j + \sum_{i=1}^{2} (x_i - a_{ij})^6)} \right]^{-1}, \quad -65.536 \leq x_i \leq 65.536$$

Binary integer

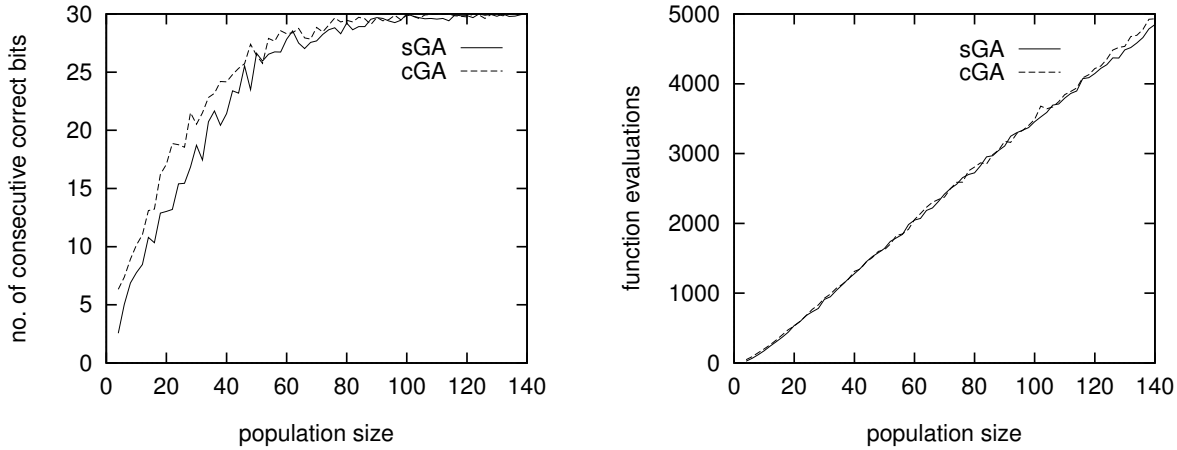

Figure 10: Comparison of the simple GA and the compact GA on a 30-bit binary integer function. The figure on the left shows the solution quality obtained at the end of the runs. On the right, the figure shows the number of function evaluations taken. The solid line is for the simple GA and the dashed line is for the compact GA.

De Jong's F1

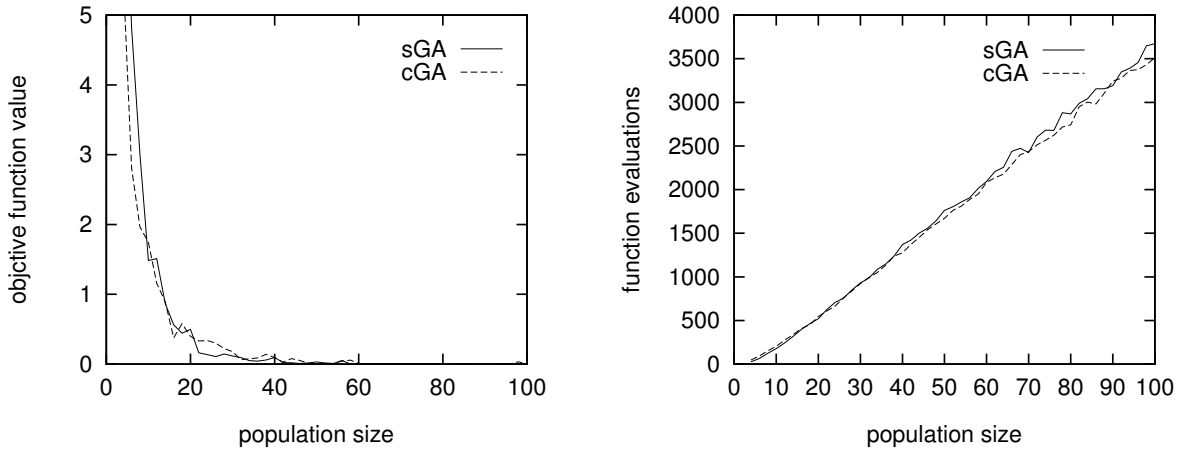

Figure 11: Comparison of the simple GA and the compact GA on function F1. The figure on the left shows the solution quality obtained at the end of the runs. On the right, the figure shows the number of function evaluations taken. The solid line is for the simple GA and the dashed line is for the compact GA.

De Jong's F2

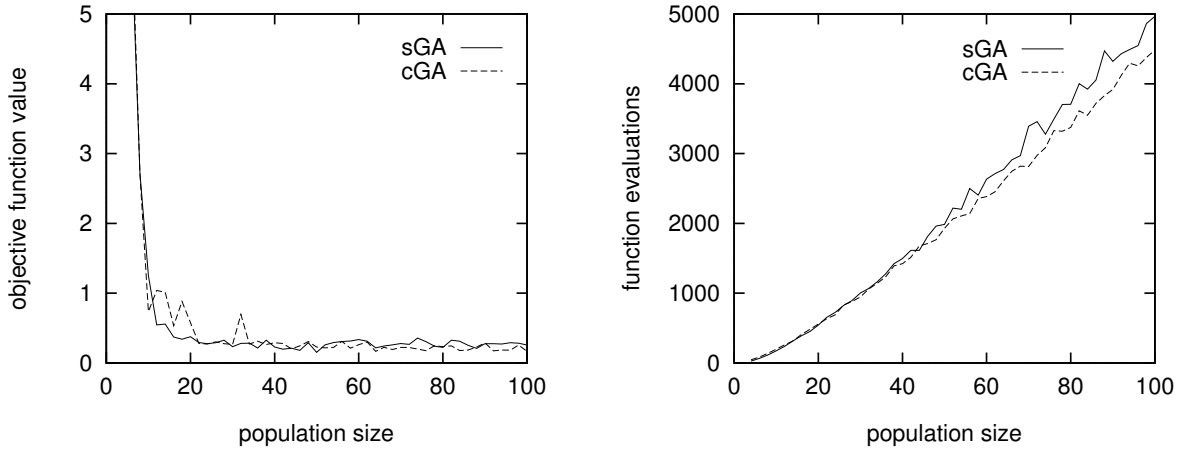

Figure 12: Comparison of the simple GA and the compact GA on function F2. The figure on the left shows the solution quality obtained at the end of the runs. On the right, the figure shows the number of function evaluations taken. The solid line is for the simple GA and the dashed line is for the compact GA.

De Jong's F3

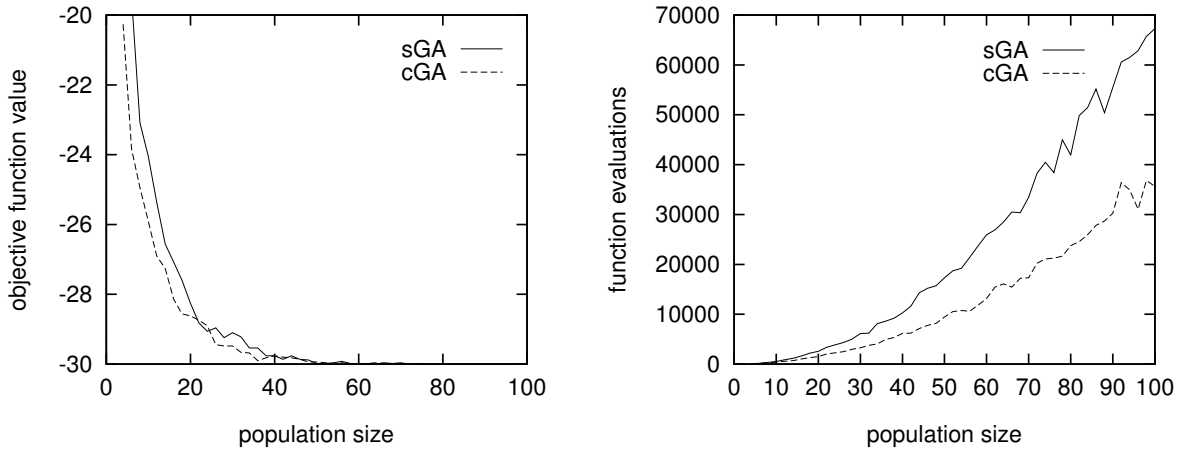

Figure 13: Comparison of the simple GA and the compact GA on function F3. The figure on the left shows the solution quality obtained at the end of the runs. On the right, the figure shows the number of function evaluations taken. The solid line is for the simple GA and the dashed line is for the compact GA.

De Jong's F4

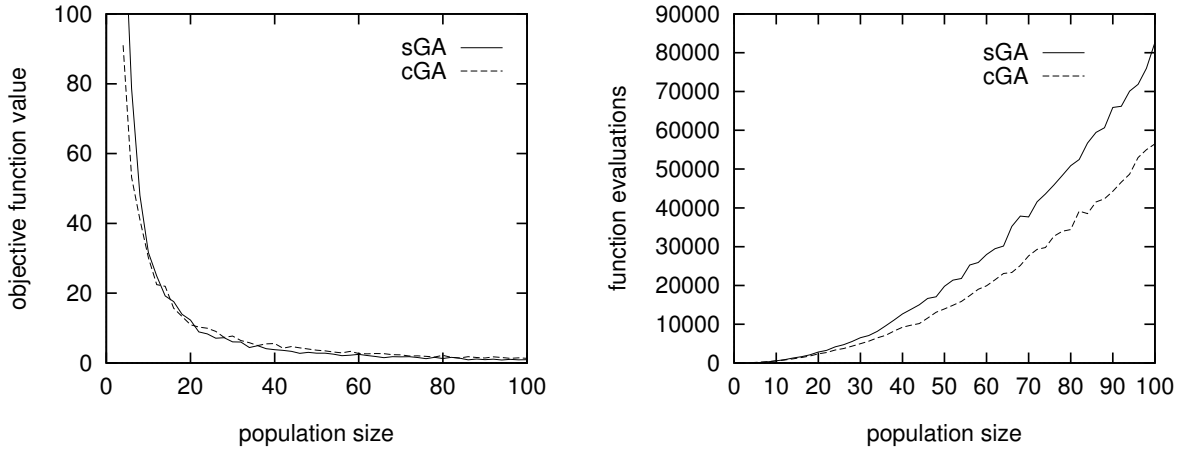

Figure 14: Comparison of the simple GA and the compact GA on function F4. The figure on the left shows the solution quality obtained at the end of the runs. On the right, the figure shows the number of function evaluations taken. The solid line is for the simple GA and the dashed line is for the compact GA.

De Jong's F5

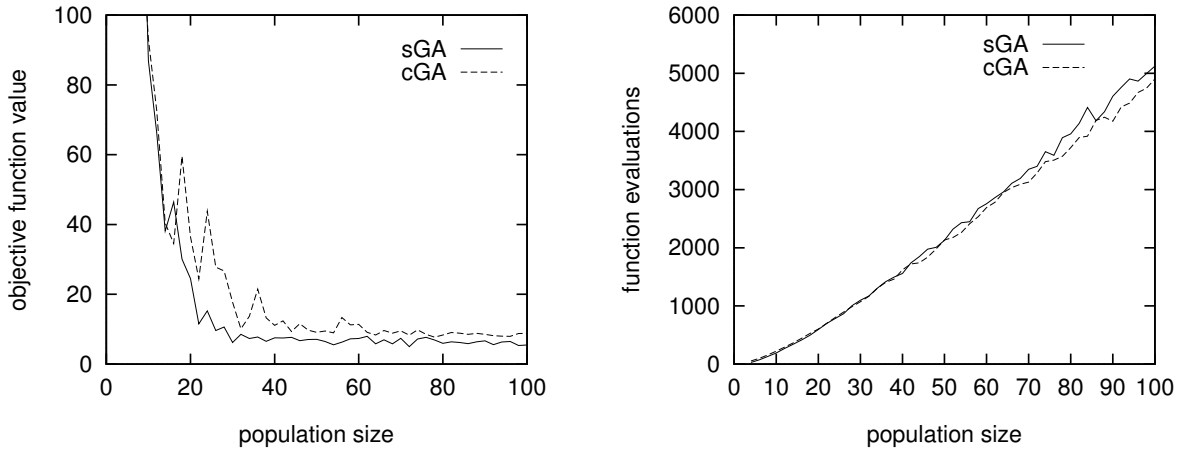

Figure 15: Comparison of the simple GA and the compact GA on function F5. The figure on the left shows the solution quality obtained at the end of the runs. On the right, the figure shows the number of function evaluations taken. The solid line is for the simple GA and the dashed line is for the compact GA.

Appendix B - Physical interpretation

An analogy with a potential field can be made to explain the search process of the compact GA, and is easily visualized for 2-bit problems. Similar results were obtained by Höhfeld and Rudolph (1997) in the context of studying the convergence behavior of the PBIL algorithm. For completeness, they are presented again.

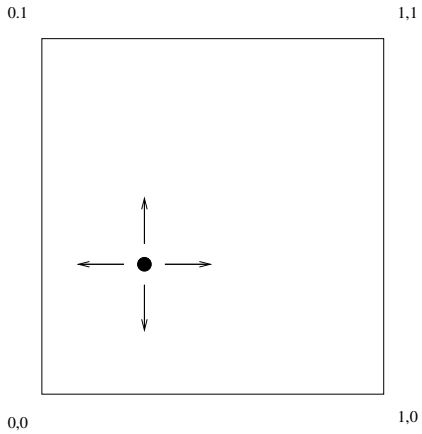

Figure 16: The black circle represents the population. Its coordinates are p and q, the proportion of 1's in the first and second gene positions. The four corners are the points in the search space.

Imagine the particle in the figure above. Its position is given by p,q, which represents the proportion of 1's in the first and second genes. The particle (population) is submitted to a potential field on the search space, seeking its minimum. As the search progresses, the particle (population) moves up or down, left or right (the proportions of 1's in each gene increase or decrease by $1/n$) and eventually, one of the corners will capture the particle (the population converges). Let's illustrate this with a 2-bit onemax problem and with the minimal deceptive problem (MDP) (Goldberg, 1987).

Onemax

Let p and q be the proportion of 1's at the first and second genes respectively. The search space, the potential field, and a graphical interpretation is shown below:

point	fitness
00	0
01	1
10	1
11	2

The potential at position p, q is: $2 - (0(1-p)(1-q) + 1(1-p)q + 1p(1-q) + 2pq) \;=\; 2 - p - q$

MDP

Likewise, for the minimal deceptive problem, the search space is:

point	fitness
00	2
01	1
10	0
11	3

and the potential at position p, q is: $3-(2(1-p)(1-q)+1(1-p)q+0p(1-q)+3pq) = -4pq+q+2p+1$

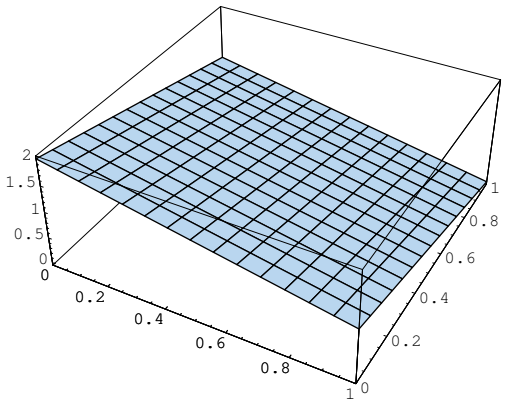

Figure 17: Potential field for the 2-bit onemax problem.

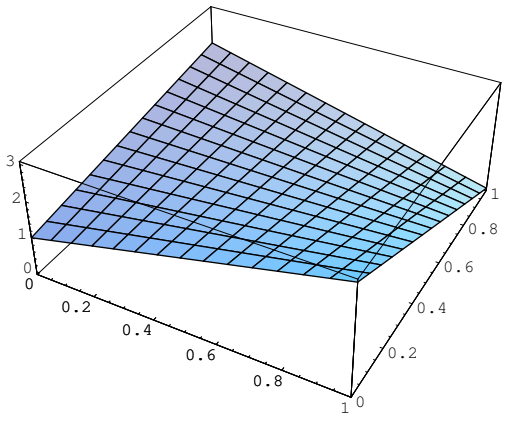

Figure 18: Potencial field vector for the MDP.

Figure 17 shows that the onemax is an easy function. Figure 18 gives a visual representation of Goldberg's observation (Goldberg, 1987) that on the MDP, the GA could converge to the deceptive attractor given certain initial conditions (high proportion of 00 in the initial population).

References

Ackley, D. H. (1987). *A connectionist machine for genetic hill climbing.* Boston: Kluwer Academic.

Baluja, S. (1994). *Population-based incremental learning: A method for integrating genetic search based function optimization and competitive learning* (Tech. Rep. No. CMU-CS-94-163). Pittsburgh, PA: Carnegie Mellon University.

Baluja, S., & Caruana, R. (1995). *Removing the genetics from the standard genetic algorithm* (Tech Rep. No. CMU-CS-95-141). Pittsburgh, Pennsylvania: Carnegie Mellon University.

De Jong, K. A. (1975). *An analysis of the behavior of a class of genetic adaptive systems.* Doctoral dissertation, University of Michigan, Ann Arbor. (University Microfilms No. 76-9381).

Deb, K., & Goldberg, D. E. (1993). Analyzing deception in trap functions. In Whitley, L. D. (Ed.), *Foundations of Genetic Algorithms 2* (pp. 93–108). San Mateo, CA: Morgan Kaufmann.

Eshelman, L. J., & Schaffer, J. D. (1993). Crossover's niche. In Forrest, S. (Ed.), *Proceedings of the Fifth International Conference on Genetic Algorithms* (pp. 9–14). San Mateo, CA: Morgan Kaufmann.

Goldberg, D. E. (1987). Simple genetic algorithms and the minimal, deceptive problem. In Davis, L. (Ed.), *Genetic Algorithms and Simulated Annealing* (pp. 74–88). San Mateo, CA: Morgan Kaufmann. (Also TCGA Report 86003).

Harik, G., Cantú-Paz, E., Goldberg, D., & Miller, B. (1997). The gambler's ruin problem, genetic algorithms, and the sizing of populations. In Bäck, T. (Ed.), *Proceedings of the Fourth International Conference on Evolutionary Computation* (pp. 7–12). New York: IEEE Press.

Hertz, J., Krogh, A., & Palmer, G. (1993). *Introduction to the theory of neural computation.* Addison-Wesley.

Höhfeld, M., & Rudolph, G. (1997). Towards a theory of population-based incremental learning. In Bäck, T. (Ed.), *Proceedings of the Fourth International Conference on Evolutionary Computation* (pp. 1–5). New York: IEEE Press.

Kargupta, H., & Goldberg, D. E. (1996). SEARCH, blackbox optimization, and sample complexity. In Belew, R. K. and Vose M. D. (Ed.), *Foundations of Genetic Algorithms 4* (pp. 291–324). San Francisco, CA: Morgan Kaufmann.

Syswerda, G. (1993). Simulated crossover in genetic algorithms. In Whitley, L. D. (Ed.), *Foundations of Genetic Algorithms 2* (pp. 239–255). San Mateo, CA: Morgan Kaufmann.

Thierens, D., & Goldberg, D. E. (1993). Mixing in genetic algorithms. In Forrest, S. (Ed.), *Proceedings of the Fifth International Conference on Genetic Algorithms* (pp. 38–45). San Mateo, CA: Morgan Kaufmann.