Robustness of Case-Initialized Genetic Algorithms

Sushil J. Louis
Genetic Adaptive Systems LAB
Department of Computer Science
University of Nevada
Reno - 89557
sushil@cs.unr.edu

Judy Johnson
Genetic Adaptive Systems LAB
Department of Computer Science
University of Nevada
Reno - 89557

Abstract

We investigate the robustness of Case Initialized Genetic AlgoRithm (CIGAR) systems with respect to problem indexing. When confronted with a series of similar problems CIGAR stores potential solutions in a case-base or an associative memory and retrieves and uses these solutions to help improve a genetic algorithm's performance over time. Defining similarity among the problems, or indexing, is key to performance improvement. We study four indexing schemes on a class of simple problems and provide empirical evidence of CIGAR's robustness to imperfect indexing.

Introduction

Genetic algorithms (GAs) are randomized parallel search algorithms that search from a population of points (Holland 1975; Goldberg 1989). We typically randomly initialize the starting population so that a genetic algorithm can proceed from an unbiased sample of the search space. However, we often confront sets of similar problems. It makes little sense to start a problem solving search attempt from scratch with a random initial population when previous search attempts may have yielded useful information about the search space. Instead, seeding a genetic algorithm's initial population with solutions to similar previously solved problems can provide information (a search bias) that can reduce the time taken to find a quality solution. Our approach borrows ideas from case-based reasoning (CBR) in which old problem and solution information, stored as cases in a case-base, help solve a new problem (Riesbeck & Schank 1989).

The case-base does what it is best at — memory organization; the genetic algorithm handles what it is best at — adaptation. The genetic algorithm also provides a ready-made case generating mechanism as the individuals generated during a GA search can be thought of as cases or as parts of cases. CBR systems usually have difficult in finding enough cases; our problem is the opposite. We need to sift through a large number of cases to find potential seeds for the initial population.

CIGAR uses the stored cases to initialize it's population when attempting a new problem. Cases are chosen based on an index of similarity between the new problem and the problems stored in the case-base. As the GA proceeds, these solutions are changed by the GA operators of crossover and mutation to adapt and become solutions to the new problem. We hope that, by storing solutions to similar problems in a case-base, we will be storing building blocks common to solutions to our new problems. When we inject these stored solutions into our initial population for the new problem, we will already have some of the schemata that are needed to build the solution we want.

From the **machine learning** point of view, using cases instead of rules to store information provides another approach to genetic based machine learning. Holland classifier systems (Holland 1975; Goldberg 1989) use simple string rules for long term storage and genetic algorithms as their learning or adaptive mechanism. In our system, the case-base of problems and their solutions supplies the genetic problem solver with a long term memory and leads to improvement over time.

Indexing is a major issue for case retrieval, especially in "poorly-understood" problems. We use a prototype system on a test problem to study the effect of different indexing schemes on performance. Studying a well understood simple problem provides the necessary analysis and intuition that will allow principled application of our system on more complex real-world problems. Preliminary results indicate that CIGAR is resilient with respect to indexing scheme and even improves performance when a misleading noisy indexing scheme is used.

The next section provides some background and defines our set of similar test problems. Section describes our methodology and outlines our baseline indexing scheme. Results from our baseline indexing scheme and variations on this scheme follow. The last section provides conclusions and directions for future research.

Background

Early work in combining genetic algorithms and case-based systems was done by Louis, McGraw (Louis, McGraw, & Wyckoff 1993) and Wyckoff, who used a case-based approach to explain the process of generating solutions using a genetic algorithm. Their approach was more concerned with using the case-base to gather information about the development of the solutions over the intermediate generations of the genetic algorithm. This study also had initial promising results from seeding their populations with promising schema generated early in the GA run and later lost (Louis, McGraw, & Wyckoff 1993). Later work by Louis dealt with the open-shop scheduling problem, circuit design, and a host of other applications providing empirical evidence of the combined system's efficacy (Liu 1996; Xu & Louis 1996; Louis & Li 1997; 1998; Louis & Johnson 1997). The thrust of this work involved research into selecting appropriate cases to inject and the number of cases to use in seeding the population. Again, results were promising, with better solutions being found in a shorter time period.

Other early work in this field was done by Ramsey and Grefenstette (Ramsey & Grefensttete 1993). They used a case-base to initialize populations for a GA finding the best strategies in a tracker/target simulation with a periodically changing environment. Solution strategies for similar environments were used to initialize the population each time the environment changed. Improvements in results with the case-base were observed both when compared to the GA running with no knowledge of the change in environment and when the GA was restarted when the environment was altered.

More current work by Louis and Johnson shows that combined genetic algorithm case-based systems improve their performance with experience on a simple test problem and on problems in combinational circuit design, traveling salesman, and scheduling (Louis & Johnson 1997). Other problems, however, may not have the properties of our test problem set that allowed us to use a simple problem indexing scheme. To test our system on other types of problem distances we changed the method we used to choose cases for injection and studied performance on our test problem reviewed below.

Our problem set is based on the number of one's in a bit string. We considered problems where a string of ones was followed by a string of zeros. Consider a string of length 10. Choose a position M in this string. Let n_1 be the number of ones to the left of M and n_0 the number of zeros to the right of M. The fitness of this string is fitness $= n_1 + n_0$ We let M be the index of a problem in this class. The maximum fitness of any of the problems in this class is l, the length of the string. Indexing is easy since the index of problem i, P_i is simply M the position that defines the start of zeros in the string. The distance between two problems P_i and P_j is $\mid P_i - P_j \mid$. When the distance between two problems is small the solution strings are similar; if two problems are within distance one of each other, the difference in the solution is one bit position. The number of problems in the class is equal to the length of the string used for the solution.

Having constructed a problem with a simple known effective indexing scheme we tested CIGAR's performance on a 50 problem set to establish a baseline for comparison. We then perturbed our indexing scheme to test CIGAR's robustness with respect to indexing.

Case-Based Initialization

Choosing the correct cases to inject is an important parameter. Liu found the following general trends in case selection (Liu 1996). 1) As problem distance increases, injecting cases with lower fitness results in better solutions. 2) This trend is emphasized with larger problem sizes. 3) A quicker flattening out of the performance curve (average or maximum fitness versus time) is seen when higher fitness individuals are injected. Keeping these trends in mind, we wanted an indexing strategy that would chose lower fitness individuals when problem distance was high and higher fitness individuals when problem distance was low. For indexing, we set a threshold value and calculated the linear distance, $\text{dist}(P_i, P_j)$, of the new problem to a problem stored in the case-base.

$$\text{dist}(P_i, P_j) = \mid P_i - P_j \mid$$

where P_i is the new problem and P_j is a problem from the case-base. If this distance was less than the threshold value, the solutions to problem P_j were put in the usable list. Seed individuals were chosen from the strings in the usable list using ***distance proportional selection***. The probability Prob_{P_j} that solutions to problem P_j would be chosen for injection was:

$$\text{Prob}_{P_j} = 1 - \frac{\text{dist}(P_i, P_j)}{\sum_{P_j \in CB} \text{dist}(P_i, P_j)}$$

where CB denotes the case-base and $\text{dist}(P_i, P_j)$ designates the linear distance between problems P_i and P_j. This ***distance proportional*** selection is similar to roulette selection used for the classical GA. More strings are chosen from those solutions in the case-base that are solutions to a problem close to the new problem. If no previous solutions are within the threshold distance, problems are randomly chosen from the case-base and solutions to those problems are placed in the usable list. In either case, individuals are chosen from generations based on the problem distance, with the probability of choosing an individual from a particular generation inversely proportional to the problem distance. When P_j is close to the new problem P_i, solutions are chosen from later generations or from the best solutions which are also stored in the case-base. If P_j is not close to the new problem, P_i, the GA is seeded with solutions from earlier generations which are presumed to be of lower fitness in solving the P_j, the problem stored in the case-base.

Elitist selection provided results that were much better than roulette selection, and we used it for the rest of the work presented in this paper. In the next section we study the time taken to find the best solution to the problem.

Baseline Results

We chose a string length of 100 and the GA was run 10 times with different random seeds, solving 50 randomly chosen problems with indices between 25 and 75. Results were averaged over these 10 runs. Each run of the GA generated it's own set of 50 problems, and during each run of 50 problems it was possible that the same problem could be chosen more than once. Figure 1 (top) shows the number of generations taken to find the best solution (y-axis) for each problem attempt (x-axis). Using injected cases, we see a decrease in the number of generations taken to arrive at the best solution. Without injected cases the time taken to the best solution remains approximately the same. By the time approximately 1/2 of the problems have been attempted, there is a statistically significant decrease in the time take to solve a new problem. When we look

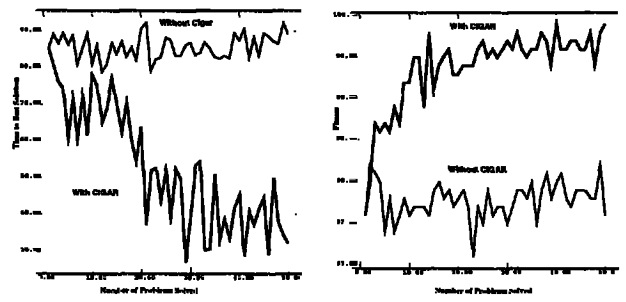

Figure 1: Solving 50 randomly generated problems with possible repetitions. Left: Time to best solution Right: Best fitness

at the quality of solutions generated, it can be seen in figure 1 (bottom) that the best fitness found with injected cases increases with experience. Without injection, the maximum fitness found is fairly constant. Using CIGAR the maximum fitness increases as more problems are solved.

We next ran the GA for 50 problems using the same set of problems in the same order for both the case-initialized GA and the random GA and for each of the 10 runs of the GA. It was still possible to repeat a problem during any of the 10 runs. Once again, the time taken to the best solution was shorter with our case-base injection than without.(Figure 2 left). The best fitness found was also better with the injected cases than without. (Figure 2 right).

Looking at the set of 50 problems where each problem is only attempted once and the same problems are evaluated in the same order in each of the 10 runs, we get similar results again. Time taken to best solution decreases (Figure 3 (top)) and best fitness found increases (Figure 3 (bottom)). In this case the quality of the solutions found using CIGAR was approximately the same as when repetition of problems was allowed. The time taken to the best solutions was slightly longer on average without repetition than when repetition was

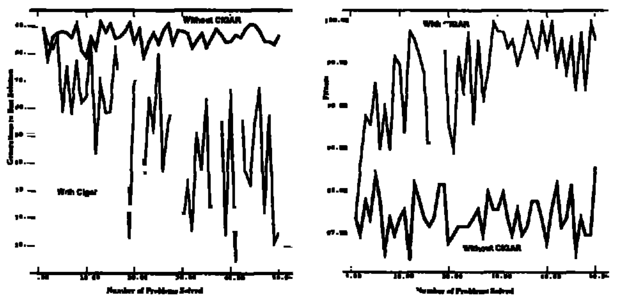

Figure 2: Solving 50 randomly ordered problems with possible repetitions. Identical order for all ten runs. Left: Time to best solution Right: Best fitness

allowed, but once again there was a statistically significant decrease in time to best solution once approximately 1/2 of the problems were solved.

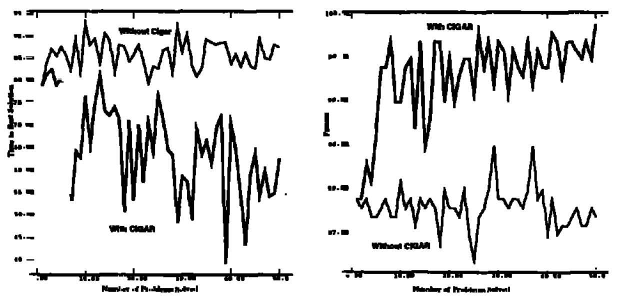

Figure 3: Solving 50 randomly ordered problems without repetitions. Identical order for all ten runs. Left: Time to best solution Right: Best fitness

These are the results we want from a learning system; the system improves with experience, taking less time to arrive at better solutions. Our system conforms to Mitchell's definition of a learning system (Mitchell 1997). Mitchell defines a machine learning program as follows (Mitchell 1997):

> A computer program is said to **learn** from experience E with respect to some class of tasks T and performance measure P, if its performance at tasks in T, as measured by P, improves with experience E.

Each problem confronted by the system is a task and our performance measure has two facets, time and quality. A decrease in time taken **or** an increase in solution quality result in a performance measure improvement. Our results so far, show that both performance measures improve but we will be satisfied with an improvement in either and expect to find real-world problems where only one performance measure improves.

Results with other Indexing Schemes

Our system does generate better results in less time when we have a good indexing system. Real world problems, however, don't usually have the properties of our problem set that allowed us to use a simple linear problem distance indexing scheme. We therefore perturbed the simple linear indexing scheme (shortened to linear scheme in the rest of the paper) to study CIGAR performance under more realistic conditions.

Randomized Indexing

We studied cases where the distance was not linear. First we looked at a randomized case, where the problems were randomly inserted into an array and the array indices were used to calculate problem distance. Distance was calculated as

$$\text{Dist}_{calc} = | \text{i} - \text{j} |$$

where i and j are indices into the array of randomly ordered problem numbers. The actual distance is

$$\text{Dist}_{actual} = | \text{ProbArray[i]} - \text{ProbArray[j]} |$$

and, therefore,

$$\text{Dist}_{actual} \neq \text{Dist}_{Calc}$$

As can be seen from Figure 4 (top) the linear array arrived at the best fitness faster than the random array based injection, but, the random injection still reaches it's best fitness faster than a purely random initialization (canonical GA). Figure 4 (bottom) shows the best fitness found is higher with a linear distance calculation than with random distance, but both are better than a purely random initialization. This is the expected behavior, since using the random array to choose cases

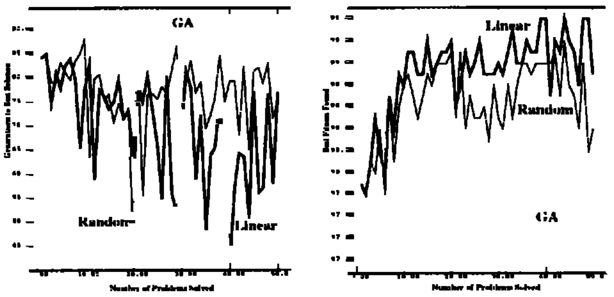

Figure 4: Solving 50 randomly ordered problems without repetitions. Identical order for all ten runs. Linear and random injection versus canonical GA. Left: Time to best solution Right: Best fitness

for injection creates a situation where cases are chosen randomly with no respect to problem distance. We expect that using the true measure of problem distance to choose cases for injection will result in selecting more appropriate cases than a system which merely chooses them at random. The random case injection still has better performance than random initialization because even randomly chosen cases are still solutions to a problem which requires a block of ones in a row and then a block of zeros. These blocks of ones and zeros become building blocks to solutions for problems which require similar blocks of ones and zeros, therefore they help the GA to find a solution faster than a random initial population.

Quadratic and Exponential Indexing

We also studied CIGAR's robustness with respect to indexing when the index used is a quadratic or exponential function of problem distance. Problem distance was therefore calculated using

$$\text{Dist}_{calc} = \text{problem distance}^2$$

and

$$\text{Dist}_{calc} = 2^{problemdistance}$$

for quadratic and exponential indexing respectively. We used a population of 50 and ran the GA for 50 generations. In Figure 5 (top) it can be seen that the quadratic distance gets to it's best fitness in the least time, with linear indexing (baseline) producing similar results. Exponential distances get poor improvement in the time to best solution, producing results similar to random initialization. Looking at Figure 5 (bottom),

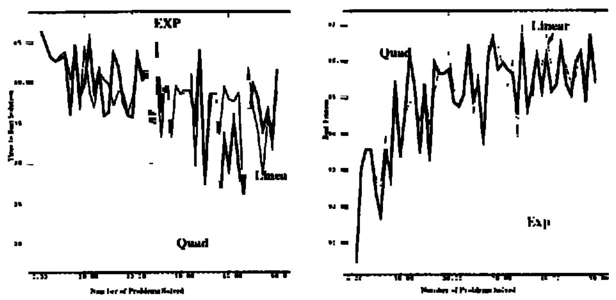

Figure 5: Solving 50 randomly ordered problems without repetitions. Identical order for all ten runs. Quadratic, Exponential, and Linear initialization. GA. Left: Time to best solution Right: Best fitness

both linear and quadratic indexing achieve similar best fitnesses with exponential indexing generating lower fitness solutions. Changing the distance function to a quadratic rather than a linear function, places more emphasis on choosing cases from those problems in the case-base that are close to the new problem. This additional emphasis helps with this problem set, and similar results are achieved in less time than with a linear distance function. However, when we increase this emphasis on closeness even more by using an exponential distance function, we decrease the fitness of the solutions we generate and increase the time taken to arrive at those solutions. For this problem, exponential distances place too much emphasis on the cases closer to the new problem, causing too much exploitation and not enough exploration.

CIGAR's resistance to noise was tested by adding a randomly chosen number between −5 and 5 to the problem distance and using this for indexing. This was studied for a 20, 50 or 80 percent probability of adding the random factor. With respect to the time taken to the best solution, basing injection on a noisy distance calculation produced results similar to linear distance injection. There was a downward trend in the time taken as more problems were solved.

When we looked at the best fitness found, noisy indexing generated better fitness solutions than both the random initialization and the strictly linear distance indexing. The different probabilities did not make a great difference in results. Figure 6 shows random initialization, linear injection and noisy injection with a probability of 80%. In this figure *CIGAR* is linear indexing injection, $CIGAR_{80}$ is noisy injection with 80% probability of noise being added to the indexing (distance) calculation, and *RANDOM* is the randomly initialized GA. Here again we see a tradeoff between exploration and exploitation. Linear injection is able to exploit the closeness of the injected solutions to the new problem solution and arrive at the best fitness in less time than the GA using the noisy injection technique. Noisy injection, however, was able to achieve better fitness solutions because of increased exploration of the search space.

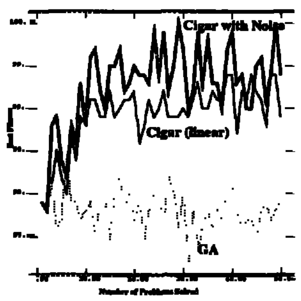

Figure 6: Misleading vs Linear Initialization, best fitness found

Conclusions

The results demonstrate that CIGAR is robust with respect to a variety of indexing schemes one our test problem, for our selection strategy. Thus, these preliminary results provide evidence that we may expect CIGAR to learn to improve performance in poorly understood domains where we do not have enough information to design a good indexing scheme. These results help explain the increased performance on a number of CIGAR applications in poorly-understood domains (Liu 1996; Xu & Louis 1996; Louis & Li 1997; 1998; Louis & Johnson 1997). Simply injecting randomly selected solutions to similar problems results in a performance increase while noise may actually benefit solution quality. Our current results provide evidence of CIGAR's robustness and point out areas for further investigation.

We need to investigate the effects of different selection strategies and crossover operators and are working on quantifying our results based on a mathematical model of our system. Another area for research is in using an indexing scheme that is based on candidate solution similarity rather than problem similarity. This reduces our domain information requirements and decrease the application dependence of indexing schemes for CIGAR systems.

Acknowledgments

This material is based upon work supported by the National Science Foundation under Grant No. 9624130.

References

Goldberg, D. E. 1989. *Genetic Algorithms in Search, Optimization, and Machine Learning.* Addison-Wesley.

Holland, J. 1975. *Adaptation In Natural and Artificial Systems.* Ann Arbour: The University of Michigan Press.

Liu, X. 1996. *Combining Genetic Algorithm and Case-based Reasoning for Structure Design.* University of Nevada, Reno. M.S. Thesis, Department of Computer Science.

Louis, S. J., and Johnson, J. 1997. Solving similar problems using genetic algorithms and case-based memory. In *Proceedings of Seventh International Conference on Genetic Algorithms*, 101–127.

Louis, S. J., and Li, G. 1997. Augmenting genetic algorithms with memory to solve traveling salesman problems. In *Proceedings of the Third Joint Conference on Information Sciences*, 108–111.

Louis, S. J., and Li, G. 1998. Combining robot control strategies using genetic algorithms with memory. In *Proceedings of the Sixth Annual Conference on Evolutionary Programming*, 431–442. Springer-Verlag.

Louis, S. J.; McGraw, G.; and Wyckoff, R. 1993. Case-based reasoning assisted explanation of genetic algorithm results. *Journal of Experimental and Theoretical Artificial Intelligence* 5:21–37.

Mitchell, T. M. 1997. *Machine Learning.* Boston, MA: WCB McGraw-Hill.

Ramsey, C., and Grefensttete, J. 1993. Case-based initialization of genetic algorithms. In Forrest, S., ed., *Proceedings of the Fifth International Conference on Genetic Algorithms*, 84–91. San Mateo, California: Morgan Kauffman.

Riesbeck, C. K., and Schank, R. C. 1989. *Inside Case-Based Reasoning.* Cambridge, MA: Lawrence Erlbaum Associates.

Xu, Z., and Louis, S. J. 1996. Genetic algorithms for open shop scheduling and re-scheduling. In *Proceedings of the ISCA 11th International Conference on Computers and Their Applications.*, 99–102. Raleigh, NC, USA: ISCA.