

Learning for Evolutionary Design

Sushil J. Louis
Genetic Algorithm Systems Laboratory
Department of Computer Science
University of Nevada, Reno
sushil@cs.unr.edu
<http://www.cs.unr.edu/~sushil>

Abstract

This paper describes a technique for evolving similar solutions to similar configuration design problems. Using the configuration design of combination logic circuits as a testbed, the paper shows that combining genetic algorithms with a case-based memory leads to improved performance on sets of similar design problems. In this approach, rather than starting from scratch on each design, we periodically inject a genetic algorithm's population with appropriate partial solutions to similar previously attempted problems. Experimental results on the combinational logic design of parity checkers and adders shows that this system takes less time to provide better quality solutions to new design problems as it gains experience from solving other similar design problems. The designs generated by the combined system also tend to be more similar than those generated by a randomly initialized genetic algorithm. This implies that the system can be used for quick, high quality re-design so that when components fail or deteriorate, we can quickly regain lost or deteriorating functionality.

1. Introduction

Genetic algorithms (GAs) are randomized parallel search algorithms that search from an initially randomly generated population of points [4, 3, 2]. Current genetic algorithm based machine learning systems use rules to store past experience to improve their performance over time, however, many application areas, especially in the design domain, are more suited to a case-based storage of past experience. We propose and describe a system that uses a case-base (or a case-library) as a long term knowledge store in a new genetic algorithm based design system that learns from experience. Results from the configuration design of parity checkers and adders applicable to problems from engineering design and architecture, show 1) that our

system, with experience, takes less time to evolve solutions to new design problems, produces better quality designs, 2) the evolved designs are more similar to each other, and 3) a simple syntactic similarity metric can result in this performance improvement thus avoiding the problem of defining a domain dependent indexing scheme.

Typically, a genetic algorithm randomly initializes its starting population so that the GA can proceed from an unbiased sample of the search space. Instead, periodically injecting a genetic algorithm's population with *relevant* partial solutions to similar previously solved problems can provide information (a search bias) that reduces 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, helps solve a new problem [10]. In our system, the data-base, or case-base, of problems and their solutions supplies the genetic problem solver with a long term memory. The system does not require a case-base to start with and can bootstrap itself by learning new cases from the genetic algorithm's attempts at solving a problem.

The case-base does what it is best at – memory organization; the genetic algorithm handles what it is best at – adaptation. The resulting combination takes advantage of both paradigms; the genetic algorithm component delivers robustness and adaptive learning while the case-based component speeds up the system and provides a means for biasing the structure of generated solutions.

These performance gains imply that fewer evaluations are needed to reach a specified design solution quality and that the organization deploying this system builds a reusable knowledge base of designs. Not only do these stored designs represent captured intellectual capital and design knowledge, they can be used to help perform sensitivity analysis, and improve the performance of a genetic algorithm based design tool [7]. The tendency to evolve more similar designs has implications for the reuse of existing components and the quick design of new hardware function-

ality from the reuse of existing components. For example, if we lose some functionality (component failure) or require new functionality from existing hardware, we can quickly generate a design that achieves the desired functionality using (remaining) working components.

The Case Injected Genetic AlgoRithm (CIGAR) system presented in this paper can be applied in a number of domains from computational science and engineering to operations research. We concentrate on configuration design problems, specifically the evolutionary design of combinational-logic parity and adder circuits.

The next section describes CIGAR, then point out the difference between problem and solution similarity. Section 4 applies the system to the parity and adder problems highlighting issues addressed by this work. The last section presents conclusions and directions for further research. Due to space limitations we do not discuss related work but refer the interested reader to [5].

2. The system

Similarity between problems and their solutions can be defined over the space of problems or the space of solutions. Classic CBR systems operate by defining similarity over the space of problems and hence are problem or domain dependent. Coming up with a (problem) similarity metric is therefore an important, usually difficult, issue in such systems. Genetic algorithms are usually applied to “poorly-understood” problems where domain knowledge is scarce and we would therefore find it even more difficult to find a good problem similarity metric. Other work describes systems that combine genetic algorithm’s and case-based memory for domains where a similarity metric exists [6, 5].

This paper concentrates on the more realistic case where domain knowledge is lacking and describes a system that uses similarity metrics over the solution space. For the problems in this paper with genetic algorithms solutions encoded as bit strings, we show that simple hamming distance suffices as our solution similarity metric for case-injected genetic algorithms.

When operating on the basis of solution similarity, CIGAR makes the assumption that similar solutions must have come from similar problems (This is admittedly a strong assumption). In this scenario, shown in Figure 1, we **periodically inject** a small number of solutions *similar to the current best member (candidate solution)* of the GA population into the *current* population, replacing the worst members. We call this the “closest to the best” strategy, one of several possible injection strategies. The GA continues searching with this combined population. In our system, every individual generated during genetic search can be a case. However, we choose to save the best individuals in the population as cases. That is, whenever the fitness of the

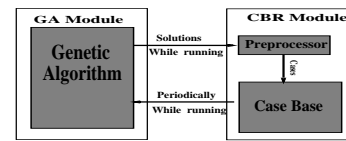


Figure 1. Conceptual view of a case injected genetic algorithm.

best individual in the population increases (the GA makes progress), the new best individual is stored in the case-base. Specifically, a case is a member of the population (a candidate solution) together with auxiliary information including fitness, the generation this case was generated, and its parents [7]. Reusing old solutions has been a traditional performance improvement procedure in heuristic search. Our work differs in that 1) we attack a set of tasks, 2) store and reuse **intermediate** candidate solutions, and 3) do not depend on the existence of a problem similarity metric. The rest of this paper only deals with CIGAR working on the basis of solution similarity.

As stated earlier, for the (traditional) binary genetic algorithm encoding of possible circuits, we use hamming distance as our similarity metric. What happens if our similarity measure is noisy and/or leads to unsuitable retrieved solutions? By definition, unsuitable solutions will have low fitness and will quickly be eliminated from the GA’s population. CIGAR may suffer from a slight performance hit but will not break or fail – the genetic search component will continue making progress toward a solution.

What exactly do we mean by problem similarity? What is the difference between problem similarity and solution similarity? The next section provides concrete answers to these questions in the context of combinational logic design.

3. Indexing and similarity

Indexing, or how we define similarity, is a basic issue in all the systems described above. Previous work has dealt with the simpler case where a similarity metric exists in the problem space [6, 9]. In this paper, we study the more realistic case where we do *not* need a similarity measure on the problem space. Since genetic algorithms solutions are encoded as binary or real strings, purely syntactic similarity measures lead to surprisingly good performance – a needed property for applications to poorly-understood systems.

Combinational circuit design is an example of a configuration design problem. The general problem can be stated as: Given a function and a target technology to work within, design an artifact that performs the function subject to constraints. For adders, the function is addition and the target technology is combinational logic gates such as boolean

AND and OR gates. For parity checkers the function is parity checking and it uses the same set of logic gates. A genetic algorithm can be used to design combinational circuits as described in [8].

A five-bit parity checker is one of $2^{2^5} = 2^{32} = 4294967296$ different five-input one-output combinational circuits (boolean functions). If we are trying to solve this class of problems, one way of indexing (defining similarity of problems) can be as follows: Concatenate the output bit for all possible 5-bit input combinations counting from 0 through 31 in binary. This results in a binary string of output bits, S_o , of length 32. Strings that are one bit different from S_o define a set of boolean functions, as do strings that are two bits different and so on. This way of naming boolean functions provides a distance metric and indexing mechanism for the combinational circuit design problems that we consider in this paper. That is, we have a metric for measuring *problem similarity*.

Problems are constructed from the parity problem by randomly changing bits in the output bit string. The fitness of a candidate circuit is the number of correct output bits. Thus 5-bit candidate parity checkers would have a maximum fitness of $2^5 = 32$.

For the 2-bit adders considered in this paper, we can define an equivalent similarity metric. Concatenate the output bits for all possible pairs of 2-bit input combinations. This is equivalent to the 16 possible combinations of $2+2 = 4$ bits. Since each of these input combinations has a 3-bit output, we get a total of $16 \times 3 = 48$ length output string, S_o . Once we define the correct 48-bit output string for a 2-bit adder, we can construct similar problems by randomly changing bits in the output bit string. A 2-bit adder is therefore one of 2^{48} different 4-bit input, 3-bit output combinational logic circuits. The fitness of a candidate circuit is also the number of correct output bits – if a circuit gets the right output for all possible inputs it is a correct circuit. Thus 4-bit input, 3-bit output problems (including the 2-bit adder) would have a maximum fitness of $2^4 \times 3 = 48$.

A **solution similarity** metric depends on how solutions/circuits are encoded. The phenotype is a two dimensional array of logic gates with inputs being fed into the first column and outputs being measured on the last column. [8] gives details on our representation. We use four (4) bits to encode each of $2^{2^1} = 16$ possible two-input, one-output gates. An additional bit specifies the location of the second input.

The solution similarity metric needs to measure the distance between encoded circuits (solutions). Since we use a binary string to encode combinational circuits, the hamming distance between these bit strings suffices as our similarity metric on this problem and our results show that this simple similarity metric works well. Real number strings could use Euclidean distance.

4. Results

For both problems we developed problem generators to produce problems that were similar - the degree of similarity was configurable. We configured these generators to produce a sequence of 50 similar problems - this number was at the limit of our computational capacity in obtaining reasonable turnaround times.

Using the parity problem as a basis, we generate 50 problems that are similar in terms of our problem similarity metric by randomly choosing and flipping 10% (uniform distribution) of the output bits of the 6-bit parity checker's $2^6 = 64$ -bit output bit string. Similarly, using the adder problem as a basis, we randomly generate 50 problems that are similar in terms of our problem similarity metric by randomly flipping 8 – 16 (16 - 33 percent) of the output bits of the 2-bit adder's 48-bit output string.

In all cases, we replace the worst individuals in the population with the individuals chosen by the injection strategy. This paper uses the probabilistic closest to the best injection strategy where the probability of being chosen for injection is directly proportional to similarity (inversely proportional to hamming distance). We use the roulette wheel method described in Goldberg to implement this probabilistic strategy [3]. We chose the injection interval, the number of generations between injections, to be $\lceil \log_2(N) \rceil$ where N is the population size. This formula reflects the takeover time when using CHC selection [1]. We inject three cases into the population (10% of the population size) every $\lceil \log_2(30) \rceil = 5$ generations. Previous work explains these parameter values [6]. All plots are averages over ten (10) runs.

CIGAR uses a population of size 100 run for a maximum of 150 generations to solve 4-bit input, 3-bit output combinational circuit design problems that are similar to adders. We use a 100 length chromosome (4 rows, 5 columns, 5 bits per gate) with a corresponding search space of 2^{100} . We inject 10 cases (10% of population size) into the population every eight generations.

Figure 2 (left) compares performance in terms of time taken to find a solution for CIGAR and a randomly initialized genetic algorithm. The figure plots time taken to find the best solution on the vertical axis and the number of problems solved on the horizontal axis. Figure 2 (right) plots the fitness of the best solution found on the vertical axis and number of problems solved on the horizontal axis. These figures show that CIGAR takes less time to produce better quality solutions as it gains experience from attempting more problems - CIGAR learns from experience.

Figure 3 (left and right) plot the distribution of similarities among solutions and along chromosome positions for the chromosomes produced by CIGAR and the randomly initialized GA. The graphs show that CIGAR produces sig-

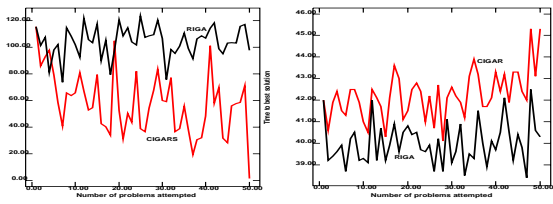


Figure 2. Left: Time to evolve best design. Right: Solution quality. As more problems are attempted, CIGAR reduces the time to convergence for adder-like circuits

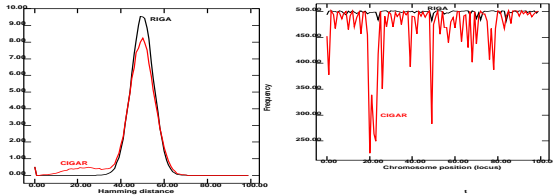


Figure 3. Left: Frequency versus hamming distance. Right: Hamming distance versus chromosome locus for CIGAR and RIGA adder-like circuits.

nificantly more solutions that are closer in hamming distance (more similar) than the randomly initialized genetic algorithm and that the hamming distances at many positions along the chromosome are much lower for CIGAR. CIGAR solutions tend to be more similar.

We have qualitatively similar results with the parity-like design problems but cannot show them here because of space limitations.

5. Conclusions and Future Work

We described a system that combines a genetic algorithm with case-based reasoning and showed that the case-injected genetic algorithm (CIGAR) learns from experience to improve performance. We used the combinational logic design of circuits as a scalable, well-understood, test problem.

These results lead to several implications for fault tolerance, evolvable hardware, and component re-use especially for very long duration space missions. The basic idea would be to use an offline problem generator with knowledge of component failure properties to generate design problems for CIGAR. Before being put to production use (or sent on a mission) the CIGAR based design system would “practice” on these problems and collect a case-library of useful complete and partial designs. In production (during a mission)

when components fail or deteriorate, our system would use the case-library to quickly evolve designs that compensate for component failure or deterioration, or that achieve new required functionality.

We are currently investigating the effect of different injection strategies, analyzing the similarities and differences in evolved circuits, and in parallelizing the code.

Acknowledgments

This material is based in part upon work supported by the National Science Foundation under Grant No. 9624130 and in part upon work supported by contract number N00014-03-1-0104 from the Office of Naval Research.

References

- [1] L. J. Eshelman. The CHC adaptive search algorithm: How to have safe search when engaging in nontraditional genetic recombination. In G. J. E. Rawlins, editor, *Foundations of Genetic Algorithms-1*, pages 265–283. Morgan Kaufman, 1991.
- [2] D. B. Fogel. *Evolutionary Computation, Toward a New Philosophy of Machine Intelligence*. IEEE Press, New York, NY, 1995.
- [3] D. E. Goldberg. *Genetic Algorithms in Search, Optimization, and Machine Learning*. Addison-Wesley, 1989.
- [4] J. Holland. *Adaptation In Natural and Artificial Systems*. The University of Michigan Press, Ann Arbor, 1975.
- [5] S. J. Louis. Learning from experience: Case injected genetic algorithm design of combinational logic circuits. In I. C. Parmee, editor, *Proceedings of the Fifth International Conference on Adaptive Computing in Design and Manufacturing*, pages 296 – 306. Springer-Verlag, 2002.
- [6] S. J. Louis and J. Johnson. Solving similar problems using genetic algorithms and case-based memory. In *Proceedings of the Seventh International Conference on Genetic Algorithms*, pages 283–290. Morgan Kaufman, San Mateo, CA, 1997.
- [7] S. J. Louis, G. McGraw, and R. Wyckoff. Case-based reasoning assisted explanation of genetic algorithm results. *Journal of Experimental and Theoretical Artificial Intelligence*, 5:21–37, 1993.
- [8] S. J. Louis and G. J. E. Rawlins. Designer genetic algorithms: Genetic algorithms in structure design. In *Proceedings of the Fourth International Conference on Genetic Algorithms*, pages 53–60. Morgan Kaufman, San Mateo, CA, 1991.
- [9] C. Ramsey and J. Grefenstette. Case-based initialization of genetic algorithms. In S. Forrest, editor, *Proceedings of the Fifth International Conference on Genetic Algorithms*, pages 84–91, San Mateo, California, 1993. Morgan Kaufman.
- [10] C. K. Riesbeck and R. C. Schank. *Inside Case-Based Reasoning*. Lawrence Erlbaum Associates, Cambridge, MA, 1989.