Journal of Soft Computing manuscript No.
(will be inserted by the editor)

Genetic Learning for Combinational Logic Design

Sushil J. Louis[1]

Genetic Algorithm Systems Laboratory
Dept. of Computer Science
University of Nevada
Reno, NV 89557
sushil@cs.unr.edu
http://www.cs.unr.edu/∼sushil

Received: October 15, 2002 / Revised version: March 15, 2003

Abstract This paper investigates the effect of injection percentage on the performance of a case-injected genetic algorithm for combinational logic design. A case-injected genetic algorithm is a genetic algorithm augmented with a case-based memory of past problem solving attempts which learns to improve performance on sets of similar design problems. In this approach, rather than starting anew on each design, we periodically inject a genetic algorithm's population with *appropriate intermediate design solutions* to similar, previously solved problems. Experimental results on a configuration design problem; the design of a parity checker, demonstrate the performance gains from our approach and show that our system learns to take less time to provide quality solutions to a new design problem as it gains experience from solving other similar design problems.

1 Introduction

Design problems seldom exist in isolation. Any useful design system must expect to confront many related problems over its lifetime and we would like such a system to improve its performance with experience. Such a learning system requires memory; a place for storing past experiences to guide future operations. The storage area may be distributed or localized, but a system without a memory is forced to start from scratch in trying to solve every given problem. Genetic algorithms (GAs) are randomized parallel search algorithms that search from a population of points [11,8]. Current genetic algorithm based machine learning systems use rules to store past experience to improve their performance over time [11, 8] and [29,13,9]. However, many application areas, especially in the design domain, are more suited to a case-based storage of past experience [23,12] and [30,7]. [1] We propose and describe a system that uses a case-base as a long term knowledge store in a new genetic algorithm based design system that learns from experience. Results from the design of parity checkers, applicable to problems from engineering design and architecture, show 1) that our system, with experience, takes less time to solve new problems and produces better quality solutions 2) a simple syntactic similarity metric can result in this performance improvement thus avoiding the indexing problem endemic to purely case-based systems and 3) the system's performance is affected by the strategy for choosing which cases to inject (injection strategy) and by the number of cases to inject (injection percentage).

Typically, a genetic algorithm randomly initializes its starting population so that the GA can proceed from an unbiased sample of the search space. However, as pointed out earlier, problems do not usually exist in isolation and we often confront sets of similar problems. It makes little sense to start a problem solving search attempt from scratch when previous search attempts may have yielded useful information about the search space. Instead, periodically injecting a genetic algorithm's population with *relevant* (we describe what we mean by relevant later) solutions or 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 [26]. 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 learn-

[1] See also the web page at http://www.ai-cbr.org/projects.html for a list of case-based design projects.

ing new cases from the genetic algorithm's attempts at solving a problem.

The **C**ase **I**njected **G**enetic **A**lgo**R**ithm (CIGAR) system presented in this paper can be applied in a number of domains from computational science and engineering to operations research. We use configuration design problems, specifically the design of combinational logic circuits, as an illustration of the performance improvements that are possible.

We define the CIGAR system in the next section. We then delineate the difference between problem and solution similarity and apply our system to two sets of design problems that highlight fundamental issues addressed by this work. The last section presents conclusions and directions for future work.

2 Case Injected Genetic Algorithms

We consider two ways of combining a genetic algorithm with case-based memory. Our first approach worked as follows [18]. When confronted with a (new) problem, the CBR module looked in its case base for *similar problems* and their associated solutions. Note that case-based reasoning research has shown that defining a problem similarity metric is non-trivial [16]. When the system found any similar problems, a small number of their solutions were injected into the *initial* population of the genetic algorithm. The rest of the population was initialized randomly (to maintain diversity) and the GA searched from this combined population. This worked well when we had a good measure of problem similarity and a prototype system was described in [18].

However, if we assume that similar solutions must have come from similar problems, an admittedly strong assumption, CIGAR can operate on the basis of solution similarity and avoid the issue of defining problem similarity metrics. 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. Since we are injecting cases into the population, injected cases must be chromosomes encoding candidate solutions to the problem at hand. To create the case base, during GA search, whenever the fitness of the best individual in the population increases, the new best individual is stored in the case-base. 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 [21]. Reusing old solutions has been a traditional performance improvement procedure. 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 avoiding indexing problems common to case-based systems.

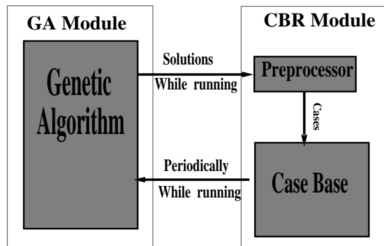

Fig. 1 Conceptual view of CIGAR.

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. CIGAR is robust.

The next section provides a summary of related work in evolutionary computing and case-based systems. We then delineate the difference between problem and solution similarity using combinational logic design, then discuss our representation, provide results, and conclude the paper.

3 Related Work

One early attempt at reuse with search can be found in Ackley's work with SIGH [1]. Ackley periodically restarts a search in an attempt to avoid local optima and increase the quality of solutions. Eshelman's CHC algorithm, a genetic algorithm with elitist selection and cataclysmic mutation, also restarts search when the population diversity drops below a threshold [6]. Other related work includes Koza's automatically defined functions [15] and Schoenauer's constraint satisfaction method [27]. More recently, Ramsey and Grefenstette come closest to our approach and use previously stored solutions to initialize a genetic algorithm's initial population and thus increase a genetic algorithm's performance in an anytime learning environment that changes with time [25,10]. Automatic injection of the best solutions to previously encountered problems biases the search toward relevant areas of the search space and results in the reported consistent performance improvements. Sheppard and Salzburg combined memory-based learning with genetic algorithms in finding better plans for the pursuer-evader game within a reinforcement learning framework [28]. Their experiments indicate that the combined approach performed better than either approach alone. To our knowledge, the earliest work in combining genetic algorithms and case-based reasoning was done by Louis, McGraw, and Wyckoff who used case-based reasoning principles to explain solutions found by genetic algorithm search [21].

Preliminary work in this area solved pairs of similar problems with a clear performance advantage for the combined system [18]. Work that uses case-based reasoning in extracting design patterns, short term memory in genetic programming, Robert Reynold's cultural algorithms, and circuit design that may be relevant are [24,14] and [2,31].

These approaches only attack a single problem not a related *class* of problems. Moreover, these approaches do not relate problem/solution similarity to the quality of injected solutions and performance – a fundamental result of this paper.

Work on multitask learning suggests, as we do, that it is easier to learn many tasks at once, rather than to learn these same tasks separately [4,5] and [3]. Multitask learning can be applied to clusters of related tasks in parallel or in sequence and provides an inductive bias that often leads to better generalization performance on the tasks. While MTL addresses generalization, we lean towards improving search performance – in design domains, a mapping from function to structure – for sequential (design) tasks. Parallel genetic algorithms using the island model, provide one possible avenue towards information exchange on related tasks in parallel for genetic algorithm based machine learning systems. This is an area to be further explored.

Lifelong learning seeks to address the problem of knowledge transfer between related tasks in the context of learning control algorithms for robotics [32,33]. The authors believe that knowledge transfer is essential for scaling robot learning algorithms to more realistic and complex domains. Lifelong learning differs from our approach in that they are interested in an incremental learning situation seeking to build behavioral complexity with experience. Work by Louis provides strong support for the CIGAR approach to control problems in robotics thus indirectly and independently providing more evidence for the utility of lifelong learning [20].

What do we mean by problem similarity? What is the difference between problem similarity and solution similarity? The next section defines what we mean by problem similarity and solution similarity in the context of combinational logic design.

4 Indexing and similarity

Indexing, or how we define similarity, is a basic issue in case-based systems. Previous work has dealt with the simpler case where a similarity metric exists in the problem space [19,25]. Simply put, when we know that two *problems* are similar, the system can use information from attempting one problem to improve performance on the other. However, a problem similarity metric is not easy to come by and remains a major issue for case-based reasoning systems. In this paper, we study the more realistic case where we do *not* have (or need) a similarity measure on the problem space. Since genetic algorithms solutions are encoded as binary or real strings, a purely syntactic similarity measure, such as hamming distance, on these binary strings leads to good performance – a needed property for application to poorly-understood systems. Solution similarity measures avoid CBR's indexing problems.

4.1 Problem Similarity

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 parity checkers, the function is parity checking and the target technology is combinational logic gates such as boolean AND and OR gates. A genetic algorithm can be used to design combinational circuits as described in [22]. 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: 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 simple 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 problems would have a maximum fitness of $2^5 = 32$. As a simple example, consider the 3-bit parity checker as shown in Table 4.1. The first column in Table 4.1 shows all possible inputs for a 3-bit parity checker, the second column shows the correct output for each input. We construct a 3-input, 1-output problem that is similar to the 3-bit parity checker by randomly choosing and flipping one of the output bits of the 3-bit parity checker as shown by the boxed 0 in the table. The correct output bits for the new problem (1 bit different from the 3-bit parity checker) are shown in the third column. This new problem is distance 1 away from the parity problem in terms of our problem similarity metric.

4.2 Representation

Since a solution similarity metric depends on how solutions/circuits are encoded, we describe our representation. An individual in the GA's population encoded as

Table 1 Outputs of the 3-bit parity checker (a 3-0) and a 3-1 problem.

Inputs	3-bit parity	3-1 problem
000	0	0
001	1	0
010	1	1
011	0	0
100	1	1
101	0	0
110	0	0
111	1	1

a bit string maps to a two-dimensional circuit as shown in figures 2 and 3. We use four (4) bits to encode all $2^{2^1} = 16$ possible two-input, one-output gates. An additional bit specifies the location of the second input. A gate $S_{i,j}$ gets its first input from $S_{i,j-1}$ and its second, from one of $S_{i+1,j-1}$ or $S_{i-1,j-1}$ as shown in figure 3.

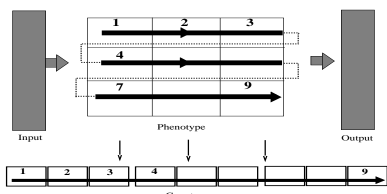

Fig. 2 Mapping from 1D chromosome to 2D circuit.

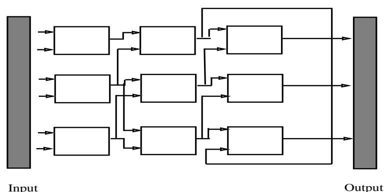

Fig. 3 A gate in a 2D template gets its second input from one of two gates in the previous column.

If the gate is in the first or last rows, the row number for the second input is calculated modulo the number of rows. The gates in the first column, $S_{i,0}$ receive the input to the circuit.

4.3 Solution Similarity Metric

The solution similarity metric needs to measure the distance between encoded circuits (solutions). We are assuming that similar circuits have similar function. 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. We show empirically that this simple similarity metric works well for these combinational logic design problems.

4.4 Configuration Design

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 0 to 10, 10 - 20, and 20 - 30 output bits of the 6-bit parity checker's output bit string.

In this paper, CIGAR uses a population of size 30 run for a maximum of 30 generations to solve 6-bit combinational circuit design problems (similar to parity checkers). This results in a 150 length chromosome (6 rows, 5 columns, 5 bits per gate) with a corresponding search space of 2^{150}. The probability of crossover is 0.95 and the probability of mutation is 0.05. All plots are averages over ten (10) runs.

In all cases, we replace the worst individuals in the population with the individuals chosen by the injection strategy. 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. We inject differing numbers of cases into the population every $\lceil log_2(30) \rceil = 5$ generations.

Figure 4 compares performance in terms of time taken to find a solution for CIGAR and a randomly initialized genetic algorithm. CIGAR used 10% injection percentage, and the probabilistic closest to the best injection strategy. The figure plots time taken to find the best solution on the vertical axis and the number of problems solved on the horizontal axis. The Randomly Initialized Genetic Algorithm (RIGA) uses the same set of genetic algorithm parameters.

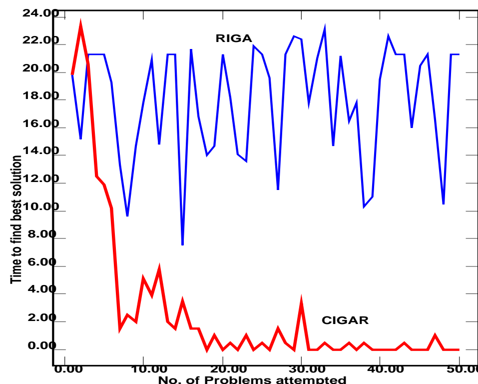

Fig. 4 Convergence speed. As more problems are attempted, CIGAR reduces the time to convergence.

Figure 5 plots the fitness of the best solution found on the vertical axis and number of problems solved on the horizontal axis. The figures clearly show that CIGAR takes less time to produce better quality solutions as it gains experience from attempting more problems.

Previous work has looked at the effect of injection strategy on performance [17]. In this study, we investigate the effect of injection percentage on performance.

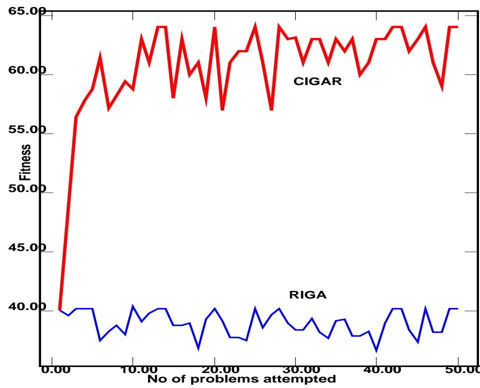

Fig. 5 Solution quality. As more problems are solved CIGAR produces better solutions

Injection percentage defines the number of cases injected into the genetic algorithm population as a percentage of the genetic algorithm's population size.

Figure 6 plots the solution quality as a function of experience for differing injection percentages when using the "closest to the best" injection strategy. The figure shows that higher injection percentages result in better quality solutions as the system gains experience. There must, however, exist a maximum injection percentage beyond which performance suffers because of premature convergence. Figure 7 shows a similar trend for the time to solution measure of performance. In these plots the system is attempting problems that have on average a difference of between 5 and 10 on the output bits, according to our problem similarity measure.

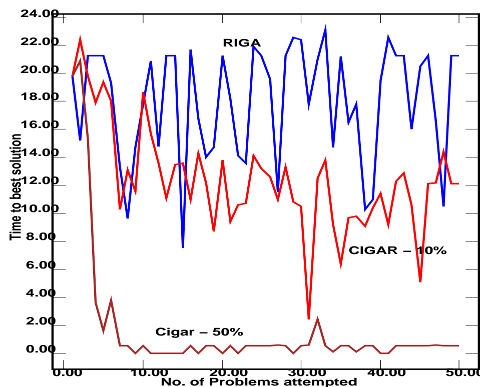

Fig. 6 Comparing injection percentages. Average time to find the best design versus number of problems attempted.

Figures 8 and 9 shows what happens as we attempt to solve problems that are on average between 10 to 20 bits different in output. All other parameters remain the same. It appears that you need more cases injected to achieve a performance boost. Keep in mind however, that these figures are affected to a large extent by the search landscape and will not quantitatively generalize.

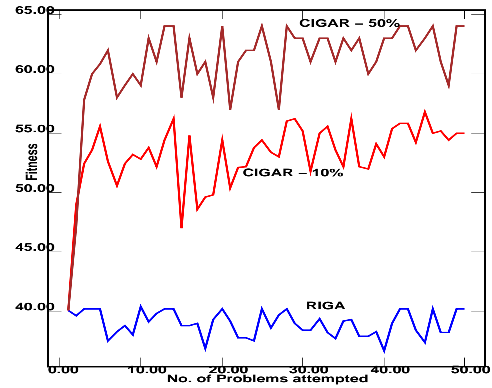

Fig. 7 Comparing injection percentages. Quality of design solution versus number of problems attempted

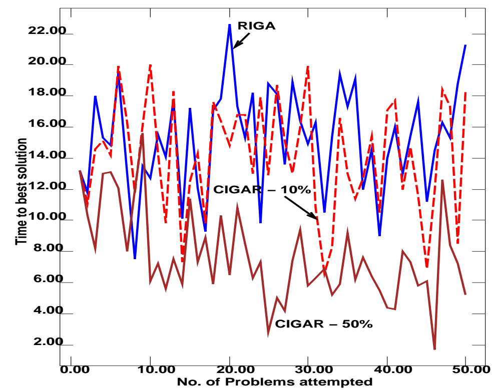

Fig. 8 Time to solution: Comparing injection percentages on problems that are less similar. These problems are between 20 and 30 bits different.

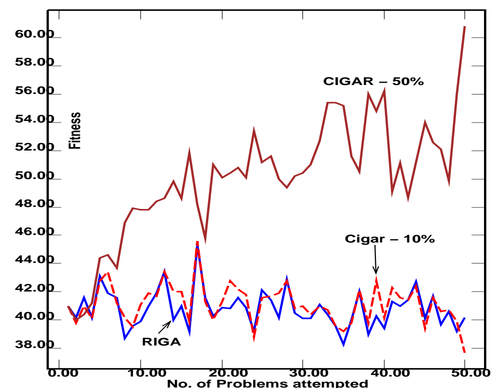

Fig. 9 Solution Quality: Comparing injection percentages on problems that are less similar. These problems are between 20 and 30 bits different.

5 Discussion and Conclusions

CIGAR makes the assumption that similar problems have similar solutions. According to the schema theorem, genetic algorithms process syntactic string similarities and we have shown that storing and injecting syntactically similar solutions leads to increased performance with experience. We expect this qualitative behavior to generalize to other design problems.

Our sample results indicate that CIGAR learns to increase performance at related tasks as it gains experience. We show that the simple syntactic similarity measure of hamming distance works well on combinational circuit design problems. It is important to note that we need to save (and re-use) to the case-base, not just the best or final solution to the problem at hand, but also intermediate solutions that the GA generates while working on a problem. We also find that the more similar the problems the larger the difference in performance between CIGAR and RIGA and the more individuals that can be injected without loss of performance.

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 knowledge base of cases. When deployed, all similar problems attempted by an organization using this system will contribute to the deployed system's case-base. With the right browsing and case analysis tools, an organization can develop and persist its problem solving intellectual property.

Our experiments also indicate that we can use one of several ways to choose individuals to be injected with some difference in CIGAR performance. We can inject the closest (individuals in the case-base) to the best individual in the population, the farthest from the worst, and probabilistic versions of both. The probabilistic versions seem to result in greater performance improvements and we are currently working on confirming these initial impressions.

We are applying CIGAR to problems in object identification, spectroscopic analysis of dense plasmas, and real-time targeting and re-targeting. Because these problems are computationally intensive or require real-time responses, we will be building and investigating a parallel CIGAR implementation, different selection schemes, and different case-base implementations. On the theoretical side we are modeling and quantifying our qualitatively derived parameter values for CIGAR.

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 the Office of Naval Research under Grant No. N00014-03-1-0104.

References

1. David A. Ackley. *A Connectionist Machine for Genetic Hillclimbing*. Kluwer Academic Publishers, 1987.
2. K. Bearpark and A. J. Keane. Short term memory in gp. In *Proceedings of fourth Adaptive Computing in Design and Manufacturing*, pages 309 – 320, 2000.
3. Rich Caruana. *Multitask Learning*. PhD thesis, School of Computer Science, Carnegie Mellon University, 1997. Pittsburg, PA.
4. Rich Caruana. Multitask learning: A knowledge-based source of inductive bias. *Machine Learning*, 28:41 – 75, 1997.
5. Rich Caruana, Shumeet Baluja, and Tom Mitchell. Using the future to "Sort Out" the present, rankprop and multitask learning for medical risk prediction. In *Advances in Neural Information Processing Systems 8, (Proceedings of NIPS-95)*, pages 959–965, 1996.
6. Larry J. Eshelman. The CHC adaptive search algorithm: How to have safe search when engaging in nontraditional genetic recombination. In Gregory J. E. Rawlins, editor, *Foundations of Genetic Algorithms-1*, pages 265–283. Morgan Kauffman, 1991.
7. A. Goel and B. Chandresekaran. Case-based design: A task analysis. In Christopher Tong and Duvvuru Sriram, editors, *Artificial Intelligence in Engineering Design, Vol II*, pages 165–184. Academic Press, Inc., 1992.
8. David E. Goldberg. *Genetic Algorithms in Search, Optimization, and Machine Learning*. Addison-Wesley, 1989.
9. J. Grefenstette, C. Ramsey, and A. Shultz. Learning sequential decision rules using simulation models and competition. *Machine Learning*, 5:355–381, 1990.
10. J. Grefensttete and C. Ramsey. An approach to anytime learning. In *Proceedings of the Ninth International Conference on Machine Learning*, pages 189–195, San Mateo, California, 1992. Morgan Kauffman.
11. John Holland. *Adaptation In Natural and Artificial Systems*. The University of Michigan Press, Ann Arbour, 1975.
12. M. Huhns and R. Acosta. Argo: An analogical reasoning system for solving design problems. In Christopher Tong and Duvvuru Sriram, editors, *Artificial Intelligence in Engineering Design, Vol II*, pages 105–144. Academic Press, Inc., 1992.
13. Cezary Z. Janikow. A knowledge-intensive genetic algorithm for supervised learning. *Machine Learning*, 13:189–228, 1993.
14. X. Jin and Robert Reynolds. Using knowledge based evolutionary computation to solve machine constraint optimization problems: A cultural algorithm approach. In *Proceedings of the Congress on Evolutionary Computation*, pages 672 – 678, 1999.
15. J. R. Koza. *Genetic Programming*. MIT Press, 1993.
16. David B. Leake. *Case-Based Reasoning: Experiences, Lessons, and Future Directions*. AAAI/MIT Press, Menlo Park, CA, 1996.
17. Sushil J. Louis. Learning from experience: Case injected genetic algorithm design of combinational logic circuits. In *Proceedings of the Fifth International Conference on Adaptive Computing in Design and Manufacturing*, page to appear. Springer-Verlag, 2002.
18. Sushil 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 Kauffman, San Mateo, CA, 1997.
19. Sushil J. Louis and J. Johnson. Solving similar problems using genetic algorithms and case-based memory. In *Proceedings of Seventh International Conference on Genetic Algorithms*, pages 101–127, 1997.
20. Sushil J. Louis and Gan Li. Combining robot control strategies using genetic algorithms with memory. *Lecture*

Notes in Computer Science, Evolutionary Programming VI, 1213:431 – 442, 1997.
21. Sushil J. Louis, Gary McGraw, and Richard Wyckoff. Case-based reasoning assisted explanation of genetic algorithm results. *Journal of Experimental and Theoretical Artificial Intelligence*, 5:21–37, 1993.
22. Sushil J. Louis and Gregory 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 Kauffman, San Mateo, CA, 1991.
23. J. Mostow, M. Barley, and T. Weinrich. Automated reuse of design plans in bogart. In Christopher Tong and Duvvuru Sriram, editors, *Artificial Intelligence in Engineering Design, Vol II*, pages 57–104. Academic Press, Inc., 1992.
24. E. Islas Perez, Carlos Cuello Coello, and A. Hernandez Aguirre. Extraction of design patterns patterns from evolutionary algorithms using case-based reasoning. In Y. Liu et al, editor, *Evolvable Systems: From Biology to Hardware (ICES 2001), Lecture Notes in Computer Science No. 2210*, pages 244 – 255. Springer, 2001.
25. C. Ramsey and J. Grefensttete. Case-based initialization of genetic algorithms. In Stephanie Forrest, editor, *Proceedings of the Fifth International Conference on Genetic Algorithms*, pages 84–91, San Mateo, California, 1993. Morgan Kauffman.
26. C. K. Riesbeck and R. C. Schank. *Inside Case-Based Reasoning*. Lawrence Erlbaum Associates, Cambridge, MA, 1989.
27. Marc Schoenauer and Spyros Xanthakis. Constrained ga optimization. In *Proceedings of the Fifth International Conference on Genetic Algorithms*, pages 573–580. Morgan Kauffman, San Mateo, CA, 1993.
28. J. W. Sheppard and S. L. Salzburg. Combining genetic algorithms with memory based reasoning. In Stephanie Forrest, editor, *Proceedings of the Sixth International Conference on Genetic Algorithms*, pages 452–459, San Mateo, California, 1995. Morgan Kauffman.
29. D Smith. Bin packing with adaptive search. In *Proceedings of an International Conference on Genetic Algorithms*, pages 202–206. Morgan Kauffman, 1985.
30. K. Sycara and D. Navinchandra. Retrieval strategies in case-based design system. In Christopher Tong and Duvvuru Sriram, editors, *Artificial Intelligence in Engineering Design, Vol II*, pages 145–164. Academic Press, Inc., 1992.
31. P. Thomson. Circuit evolution and visualization. In J. Miller et al, editor, *Evolvable Systems: From Biology to Hardware (ICES 2000), Lecture Notes in Computer Science*, pages 229 – 240. Springer, 2000.
32. S. Thrun. *Explanation-Based Neural Network Learning: A Lifelong Learning Approach*. Kluwer Academic Publisher, Amsterdam, 1996.
33. S. Thrun. Is learning the n-th thing any easier than learning the first. In *Advances in Neural Information Processing Systems 8, (Proceedings of the NIPS-95)*, pages 640–646, 1996.