

Case Injected Genetic Algorithms for Learning Across Problems

Sushil J. Louis

Dept. of Computer Science

University of Nevada

Reno, NV 89557

sushil@cs.unr.edu

<http://www.cs.unr.edu/~sushil>

November 21, 2004

Abstract

We use genetic algorithms augmented with a case-based memory of past design problem solving attempts to obtain better performance over time on sets of similar design problems. Rather than starting anew on each design, we periodically inject a genetic algorithm's population with *appropriate intermediate design solutions* to similar, previously solved design problems. Experimental results on configuration design problems; the design of parity checker and adder circuits, 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.

Keywords: Learning, similarity, genetic algorithm

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 (classifier systems) use rules to store past experience to improve their performance over time [11, 8] and [27, 13, 9]. However, many application areas, especially in the design domain, are more suited to a case-based storage of past experience [21, 12] and [28, 7]. 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 and adders, 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 and 2) a simple similarity metric based on the genotype can result in this performance improvement thus avoiding having to come up with a phenotypic (problem specific) similarity metric.

Consider an application setting, say in an engineering design firm or a supply chain management company that uses a genetic algorithm package to do optimization. We expect that the firm uses the package many times and that the package is used to solve many search and optimization problems. This paper assumes that some of these problems are similar. For example, perhaps there was a small change in the engineering specification or weather delayed a particular arrival time - either scenario leads to a new problem that is different from but similar to the original. Not only can firms confront slight variations of a problem, different clients can pose similar problems. For example, consider a client that asks for design of a component that is similar in function to one that the firm has already designed for another client. When designing a component for a new client, the engineer may look up and use an old blueprint corresponding to a similar component that someone at the firm had generated before. This paper describes a technique that allows the genetic algorithm to make use of relevant past problem solving experience to guide its search on a current problem.

Typically, when solving a problem, P_0 , with evaluation function f_0 , we would randomly initialize the genetic algorithm's population in order to proceed from an unbiased sample of the search space. When solving another problem, say P_1 with evaluation function f_1 , we would start the GA again with a randomly initialized population. This is a good approach when P_0 and P_1 are not related. However, if we assume that P_0 is similar to P_1 , we may want to use information gleaned while solving P_0 to solve P_1 . In general, a genetic algorithm based optimization system may be confronted by many problems, P_0 through P_n in sequence over its lifetime. This paper describes a genetic algorithm technique that uses information from solving P_0 to help solve P_1 , information from solving P_0 and P_1 to help solve P_2 , and in general, information from solving $P_0...P_{n-1}$ to help solve P_n . 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 [24]. 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 [24]. 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.

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. An organization equipped with such a system and with case-base exploration and analysis tools would have a useful repository of key design knowledge and experience.

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.

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.

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 design of combinational logic circuits. Specifically, we use the design of combinational logic circuits similar to parity checker and adders as a testbed for the following reasons.

- These problems are well studied in computer science
- The problems are scalable in size
- We can easily generate many different problems
- We have experience with using genetic algorithms for these problems and have published on this topic.

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

2 Case Injected Genetic Algorithms

How do we combine a genetic algorithm with case-based memory? Our first approach worked as follows. 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]. If 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.

However, genetic algorithms are usually applied to poorly-understood problems [11, 8]. We do not expect to easily define a problem similarity metric when we have a poor understanding of the problem domain. This observation led us to define a slightly different genetic algorithm case-based memory system described below.

If we assume that similar solutions must have come from similar problems, 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.

What are cases and how do we save them to 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 ancillary information including fitness, the generation this case was generated, and its parents [19]. 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.

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. As noted earlier, we can choose schemes other than injecting the closest to the best; furthest from the worst and probabilistic versions of both have proven effective.

3 Related Work

One early attempt at reuse 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 [25]. 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 [23, 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 [26]. 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 [19]. Preliminary work in this area solved pairs of similar problems with a clear performance advantage for the combined system [17]. Work that uses case-based reasoning in extracting design patterns, short term memory in genetic programming, Robert Reynolds' cultural algorithms, and circuit design that may be relevant are [22, 14] and [2, 29].

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 toward 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 toward information exchange on related tasks in parallel for genetic based machine learning systems.

Lifelong learning seeks to address the problem of knowledge transfer between related tasks in the context of learning control algorithms for robotics [30, 31]. 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 [18].

What 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.

4 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 [17, 23]. 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* 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. Solution similarity measures provide a domain independent similarity metric thus avoiding having to define a problem specific similarity metric.

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 [20]. 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

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

| Inputs | 3-bit parity problem | 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 |

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 1. The first column in Table 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 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. 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

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 that this simple similarity metric works well for these parity checker design problems. Real number strings could use Euclidean distance.

4.4 Configuration Design

Using the parity problem as a basis, we randomly generate 50 problems that are similar in terms of our problem similarity metric by randomly flipping 12% of the output bits of the 6-bit parity checker's output bit string. This gives us 50 different combinational logic problems, $P_0 \dots P_{49}$ to be solved one by one.

In the next subsection, the genetic algorithm 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. CHC elitist selection is used as our elitist selection strategy. Previous work has shown that this set of parameters allows the GA to converge quickly to correct parity circuits [17]. The paper also explains why this particular tradeoff between exploration of the search space (crossover and mutation) and exploitation of high fitness areas of the space (selection) is suitable in such problems. We inject three cases (10% of population size) into the population every five generations. All plots are averages over ten (10) runs.

4.5 Parity

Figure 4(left) compares performance in terms of time taken to find a solution for a Randomly Initialized Genetic Algorithm (RIGA) and CIGAR. 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 time taken is in number of generations which can be easily converted to number of evaluations by multiplying by population size. That is, each point on the horizontal axis represents the problems P_0 through P_{49} that we are attempting to solve. Both algorithms start with a randomly initialized population on each problem. We expect the same performance on P_0 since CIGAR has no case-base to help on P_0 , however, CIGAR saves individuals in its population to the case-base while solving P_0 . When solving P_1 , a different problem from P_0 , CIGAR injects cases from its case-base (cases from P_0) into its population every five generations. If P_0 is similar to P_1 and both share design components we may expect a performance improvement for CIGAR - on the other hand, if P_0 and P_1 require very different circuits, CIGAR may perform worse. The GA without case injection also starts with a randomly initialized population and has no memory of its search on P_0 , thus, if P_0 and P_1 are at about the same level of problem difficulty (for our GA) we expect it to perform about the same on both problems. When solving P_2 , CIGAR can use cases saved from both P_0 and P_1 , to inject into the evolving population of the GA solving P_2 . Figure 4 (left) shows that CIGAR takes decreasing time to find its best designs as it gains and stores experience at solving other related problems. Note, however, that this may be a symptom of premature convergence unless these best designs also happen to be as good as, or better than, the designs found by a GA without case injection.

Figure 4 (right) plots the fitness of the best solution found on the vertical axis and number of problems solved on the horizontal axis. The figure shows that CIGAR finds better solutions than the GA without case-injection, as it solves more problems. Together, the figures show that CIGAR takes less time to produce better quality solutions as it gains experience from attempting different, but related, problems.

We have also begun investigation of a number of injection strategies and have experimented with the following:

1. Closest to the best
2. Probabilistic closest to the best
3. Furthest from the worst
4. Probabilistic furthest from the worst
5. Randomly choosing individuals from the case-base to inject
6. Creating and injecting randomly generated individuals

In all cases, we replace the worst individuals in the population with the individuals chosen (or created) by the injection strategy. Our preliminary results shows that the probabilistic strategies and the strategy of randomly injecting individuals from the case-base result in the largest increase in both our performance measures with experience. As expected, creating and injecting random individuals does not lead to better performance. Both deterministic strategies show intermediate gains in quality.

4.6 Adder

The adder's requirement for a carry bit makes it a harder design problem for the genetic algorithm. We thus used a larger population size of 50 run for 75 generations to design 2-bit adders and adder-like 4-bit input, 3-bit output circuits. The encoding for this problem takes up 100 bits for 4 rows, 5 columns, and 5 bits per gate. As before, we generated 50 problems by randomly changing between 8 and 16 of the $2^4 \times 3 = 48$ bit output boolean strings that define the function for which we want a circuit designed. Results are averages over 10 runs with different random seeds.

Figure 5 compares CIGAR's performance using the probabilistic injection strategies with a randomly initialized GA. Figure 5(left) plots time taken to find the best solution on the vertical axis and the number of problems solved on the horizontal axis. Figure 5 (right) plots the fitness of the best solution found on the vertical axis and number of problems solved on the horizontal axis. Again, the graphs show that CIGAR takes less time to produce better quality solutions as it gains experience from attempting more problems.

In both cases we can see that the performance difference is not as large as with the parity problem. The grouping of 3 output bits for each possible 4-bit input are not respected in our problem generator and this probably reduces CIGAR's ability to learn from past problems. The straight lines on the graphs are linear regression lines of best fit and predict steady performance improvement with experience.

Figure 6 plots the distribution of solutions' similarities produced by CIGAR and RIGA. Although all algorithms peak at around the same number of bits, CIGAR designed solutions tend to have more portions in common as shown by zooming in on the left portion of figure as shown in the inset. Again, CIGAR solutions tend to be more similar indicating that CIGAR is storing and reusing solutions and partial solutions to previously solved problems.

We have used CIGAR on a number of other problems from design and from optimization. In every case, we observe the same qualitative behavior: CIGAR increases performance with experience.

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. CIGAR thus combines the strengths of GAs and CBR and represents a new kind of genetic algorithm based machine learning system.

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. However, injecting too many individuals often leads to premature convergence. Since we do not usually know problem similarity injecting between 5% – 15% of the population size strikes a good balance.

Our experiments also indicate that we can use one of several ways to choose individuals to be injected. 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 deliberately made the parity checker and the adder that last problems to be attempted out of the 50. Our results support the multi-task learning approach where “practice” at related tasks helps to solve the main task at hand. CIGAR’s performance on the last task (parity checker or adder) is better for its experience in attacking related problems. Thus, our results show that our approach can be used to attack difficult (but not deceptive) problems for GAs.

The experiments indicate the potential of this simple technique in augmenting genetic algorithm performance in an industrial setting. The point is not that we want an algorithm that is “best” at designing adders or parity checkers but that we have described an approach that shows promise in boosting genetic algorithm performance with experience. Enterprises will also benefit from the case-base of solutions and partial solutions obtained from deploying and using such a system.

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 by contract number N00014-03-1-0104 from the Office of Naval Research.

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. Chandrasekaran. 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. Grefenstette 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 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.
- [18] 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.
- [19] 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.
- [20] 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.
- [21] 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.
- [22] 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.
- [23] C. Ramsey and J. Grefenstete. 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.
- [24] C. K. Riesbeck and R. C. Schank. *Inside Case-Based Reasoning*. Lawrence Erlbaum Associates, Cambridge, MA, 1989.
- [25] 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.
- [26] 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.
- [27] D Smith. Bin packing with adaptive search. In *Proceedings of an International Conference on Genetic Algorithms*, pages 202–206. Morgan Kauffman, 1985.

- [28] 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.
- [29] 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.
- [30] S. Thrun. *Explanation-Based Neural Network Learning: A Lifelong Learning Approach*. Kluwer Academic Publisher, Amsterdam, 1996.
- [31] 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.

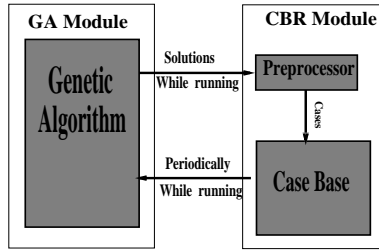


Figure 1: Conceptual view of CIGAR.

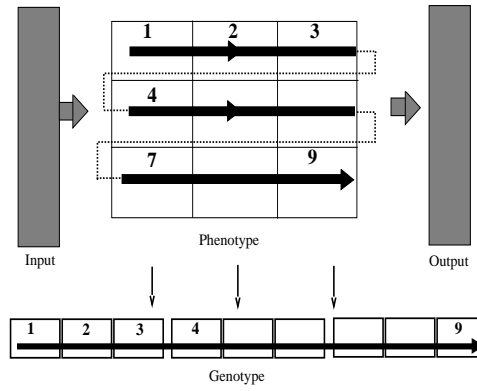


Figure 2: Mapping from 1D chromosome to 2D circuit.

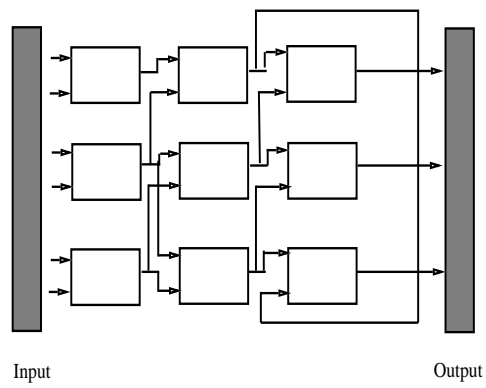


Figure 3: A gate in a 2D template gets its second input from one of two gates in the previous column.

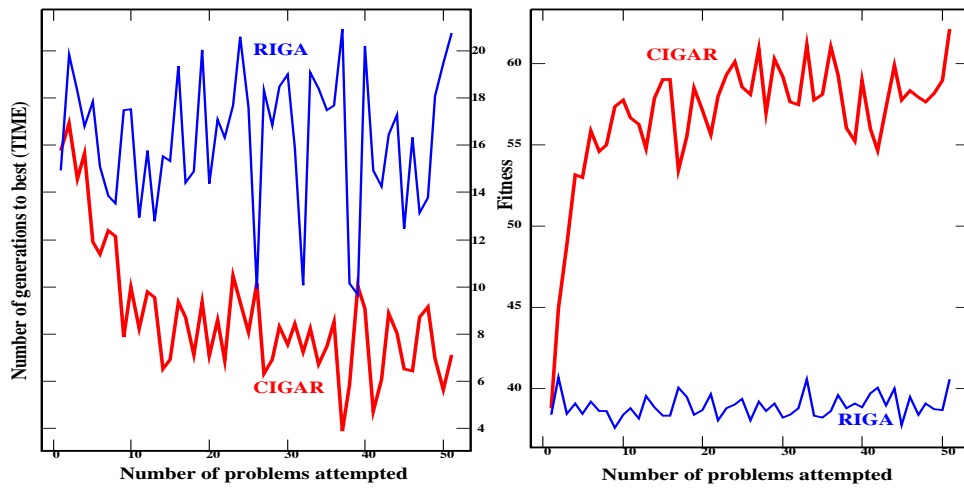


Figure 4: Parity. Left: Convergence speed. As more problems are attempted, CIGAR reduces the time to convergence.

Right: Solution quality. As more problems are solved CIGAR produces better solutions

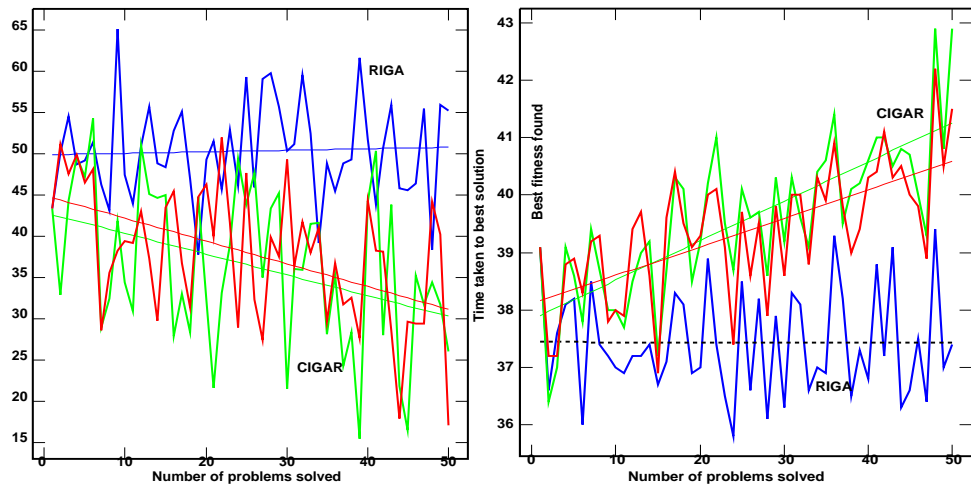


Figure 5: Adder. Left: Convergence speed. As more problems are attempted, CIGAR reduces the time to convergence. Right: Solution quality. As more problems are solved CIGAR produces better solutions

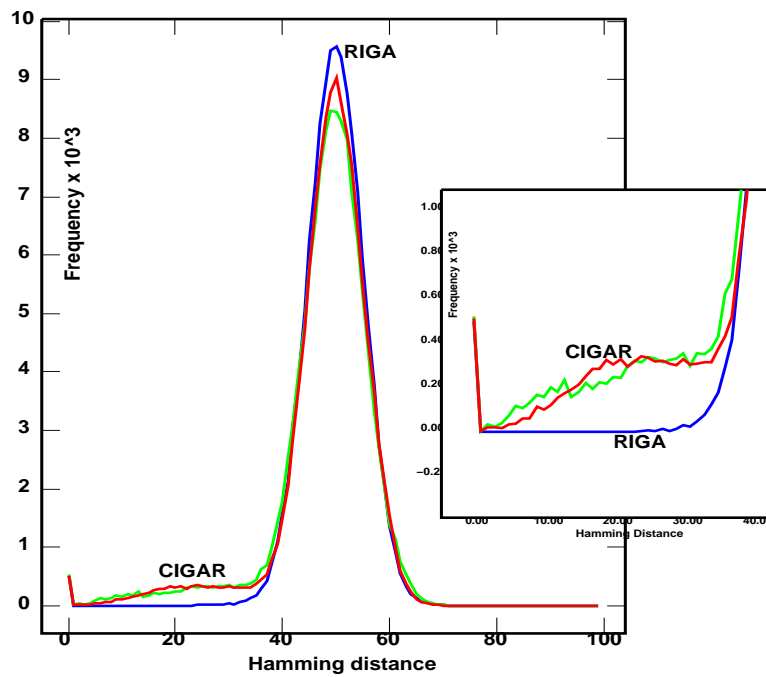


Figure 6: Adder solution distribution