Learning With Case-Injected Genetic Algorithms

Sushil J. Louis, *Member, IEEE,* and John McDonnell, *Member, IEEE*

***Abstract*—This paper presents a new approach to acquiring and using problem specific knowledge during a genetic algorithm (GA) search. A GA augmented with a case-based memory of past problem solving attempts learns to obtain better performance over time on sets of similar problems. Rather than starting anew on each problem, we periodically inject a GA's population with *appropriate intermediate solutions* to similar previously solved problems. Perhaps, counterintuitively, simply injecting *solutions* to previously solved problems does not produce very good results. We provide a framework for evaluating this GA-based machine-learning system and show experimental results on a set of design and optimization problems. These results demonstrate the performance gains from our approach and indicate that our system learns to take less time to provide quality solutions to a new problem as it gains experience from solving other similar problems in design and optimization.**

***Index Terms*—Case-based reasoning, genetic algorithm (GA), optimization.**

I. Introduction

PROBLEMS seldom exist in isolation. Any useful system must expect to confront many related problems over its lifetime and we would like such a system to be able 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 [1], [2]. Current GA-based machine-learning systems, such as classifier systems, use rules to store past experience in order to improve their performance over time [1]–[5]. However, many application areas are more suited to a case-based storage of past experience [6]–[9]. We propose and describe a system that uses a case base as a long-term knowledge store in a new GA-based machine-learning system. Results from three design and optimization problems show: 1) that our system, with experience, takes less time to solve new problems and produces better quality solutions and 2) simple syntactic similarity metrics lead to this performance improvement. In addition, our system mitigates domain-dependent indexing, a difficult issue in purely case-based systems.

Manuscript received February 17, 2003; revised November 20, 2003. This work was supported in part by the National Science Foundation under Grant 9624130 and by the Office of Naval Research under Contract N00014-03-1-0104.

S. J. Louis is with the Evolutionary Computing Systems Laboratory, Department of Computer Science, University of Nevada, Reno, NV 89557 USA (e-mail: sushil@cs.unr.edu).

J. McDonnell is with the Space and Naval Warfare Systems Center, San Diego, CA 92110 USA (e-mail: john.mcdonnell@navy.mil).

Digital Object Identifier 10.1109/TEVC.2004.823466

Typically, a GA randomly initializes its starting population so that the GA can proceed from an unbiased sample of the search space. However, since problems do not usually exist in isolation, we expect a system deployed in an industrial setting to confront a large number of problems over the system's lifetime. Many of these problems may be similar. In this case, it makes little sense to start a problem-solving search attempt from scratch when previous searches may have yielded useful information about the problem domain. Instead, periodically injecting a GA's population with *relevant* solutions (we describe what we mean by relevant later) or partial solutions from previously solved similar problems can provide information (a search bias) that reduces the time taken to find an acceptable 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]–[12]. In our system, the database, 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 GA's attempts at solving a problem.

The case base does what it is best at, i.e., memory organization; the GA handles what it is best at, i.e., adaptation. The resulting combination takes advantage of both paradigms; the GA component delivers robustness and adaptive learning while the case-based component speeds up the learning process.

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 one design problem, the design of combinational logic circuits, and on two optimization problems, strike force asset allocation and job shop scheduling, to use as testbeds in this paper.

It must be pointed out that our system differs significantly from classifier systems [1]. One way that classifier systems differ is that they use rules to represent domain knowledge; our system uses cases. We will describe other differences with classifier systems in Section II.

Although we only consider using GAs and a case base of past experience, we believe that our approach is not limited to either GAs or to case-based memory. We conjecture that properly combining a robust search algorithm with some implementation of an associative memory can result in a learning system that learns, with experience, to solve similar problems quickly. What we mean by similarity and how to identify similar problems and solutions will be discussed at length in the subsequent sections.

We define the CIGAR system and a framework for the learning task in the next section. We then discuss related work and delineate the difference between problem and solution similarity. The subsequent section applies our system to three different problems, combinational circuit design, strike force asset allocation, and job shop scheduling, that are used to highlight issues addressed by this work. The last section presents conclusions and directions for future work.

II. Framework for Learning During Search

Mitchell defines a machine-learning program as follows [13].

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.

Using this notation, we let $T = \{t_1, t_2, \ldots, t_n\}$ denote the set of tasks with solutions found in the search space defined by the set S. Casting our definition of the set of tasks in terms of search, S becomes the search space for the GA for all $t \in T$. Feedback is provided to the GA by the finite set, $O = \{o_1, o_2, \ldots, o_n\}$ of objective functions corresponding to the set of tasks, which map candidate solutions to the set of real numbers. That is

$$\langle O : \{S \to \mathcal{R}\}\rangle$$

where $\mathcal{R}$ is the set of real numbers, the range of the mapping. GAs attempt to maximize a fitness function, a function of the objective function that maps objective function values to the set of nonnegative real numbers $\mathcal{R}_{\geq 0}$. Thus, $F = \{f_1, f_2, \ldots, f_N\}$, the set of corresponding fitness functions, is a set of maps from points in S to $\mathcal{R}_{\geq 0}$

$$\langle F : \{S \to \mathcal{R}_{\geq 0}\}\rangle.$$

Note that the space of possible solutions S remains syntactically (or representationally) the same for all $t \in T$. This is not usually an issue for GAs since solutions tend to be encoded as binary or real strings. CIGAR makes use of this property to avoid domain dependent indexing.

Elements of F define different search landscapes generating search biases which may lead to different points in S as solutions to tasks in T. A task is then the search problem of finding a solution $s_i^* \in S$ that maximizes the fitness function f_i for task t_i to a certain degree of precision. More formally, a task can be described in the following way:

$$\begin{aligned}\text{Given: } & t_i \equiv \langle f_i : S \to \mathcal{R}_{\geq 0}\rangle \\ \text{Find: } & s_i^* \in S : \forall s \in S\,[f_i(s_i^*) \geq f_i(s)]\,.\end{aligned}$$

CIGAR gains experience by attempting to solve tasks in T. Thus, we define a problem-solving experience $e \in E$ to be an attempt to solve a task $t \in T$. Performance can be measured in terms of the time taken to solve a task, denoted P^t, and in terms of the quality of solutions found P^q, with improvement in either or both constituting an increase in performance. Without any loss of generality, we define performance as a function from E to the set of integers

$$\langle P^t : E \to I\rangle \text{ and } \langle P^q : E \to I\rangle$$

where I is the set of integers.

We assume that CIGAR will experience similar tasks over time. In order to exploit this similarity, CIGAR needs a similarity metric $\mathcal{D}$, defined on the set of tasks T or on the set of solutions S. The similarity metric is a function from pairs of tasks to the set of integers, where I denotes the set of integers, or from pairs of solutions to the set of integers

$$\langle \mathcal{D} : T \times T \to I\rangle \text{ or } \langle \mathcal{D} : S \times S \to I\rangle.$$

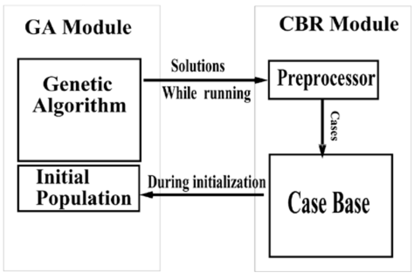


Fig. 1. Conceptual view of CIGAR_p.

Let P_{GA}^t be the time taken to solve a task using a randomly initialized GA and let the corresponding time for CIGAR be denoted by P_{CIGAR}^t. With increasing experience, we would expect $P_{\mathrm{GA}}^t \geq P_{\mathrm{CIGAR}}^t$ for a particular level of solution quality. We would also expect $P_{\mathrm{GA}}^q \leq P_{\mathrm{CIGAR}}^q$ for a given amount of time. That is

$$\lim_{j\to\infty} P_{\mathrm{CIGAR}}^t(t_i, S, f_i, \mathcal{D}, e_j) \leq P_{\mathrm{GA}}^t(t_i, S, f_i, \mathcal{D})$$

and

$$\lim_{j\to\infty} P_{\mathrm{CIGAR}}^q(t_i, S, f_i, \mathcal{D}, e_j) \geq P_{\mathrm{GA}}^q(t_i, S, f_i, \mathcal{D})\,.$$

This mathematical framework allows us to evaluate how the system described in this paper learns with experience.

Although we cater to the definition of machine learning in Mitchell's book, unlike classifier systems and many other machine-learning algorithms, we do not explicitly induce patterns (generalize) from experiencing data nor do we expect to obtain concept descriptions from exposure to exemplars, or to learn weights, or rules. Instead, think of repeated searches using the system as exploring a domain, gleaning useful information about the domain, storing this in a long-term memory, then retrieving and using this information to bias future searches in the domain. More specifically, we use cases to store domain information in a case base, then retrieve a subset of these cases from the case base and inject them into the GA's population to bias future searches in the domain. Note that cases stored in long-term memory may, in fact, implicitly embody a domain model.

A. *Problem Similarity*

How do we combine a GA with case-based memory? Our first approach worked as follows. When confronted with a (new) problem, the CBR module looks in its case base for *similar problems* and their associated solutions. Note that CBR research has shown that defining a problem similarity metric is nontrivial [12]. If the system finds any similar problems, a small number of their solutions gets injected into the *initial* population of the GA. This is case-based initialization. The rest of the population is initialized randomly (to maintain diversity) and the GA searches from this combined population. This works well if we have a good measure of problem similarity. Fig. 1 shows a conceptual

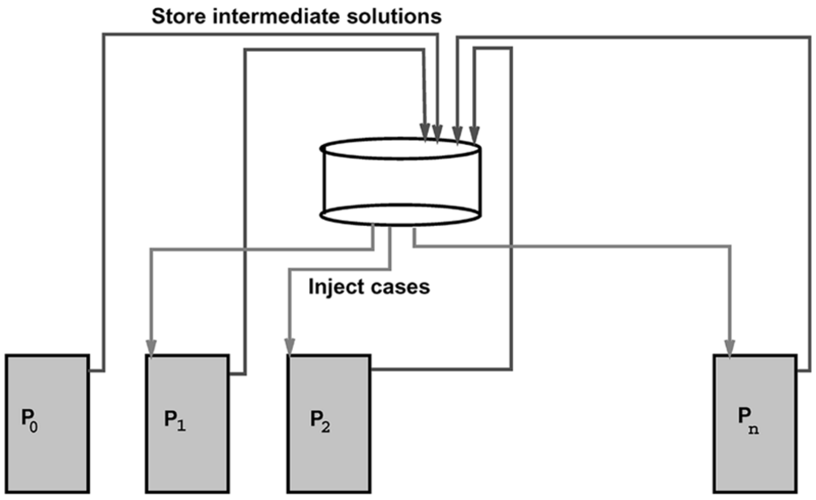


Fig. 2. Solving problems in sequence.

view of this CIGAR$_p$ working on the basis of problem similarity. As the figure shows, while the GA works on a problem, promising members of the population are stored into the case base through a preprocessor. Subsequently, when starting work on a new problem, suitable cases are retrieved from the case base and are used to populate a small fraction (say 10%) of the initial population. In this paper, a case is a member of the population (a candidate solution) together with ancillary information including its fitness and the timestep at which this case was generated [14]. During a GA search, whenever the fitness of the best individual in the population increases, the new best individual is stored in the case base.

This problem similarity-based CIGAR system has been reported in [15] and [16] and led to several principles for working with CIGAR$_p$. We elucidate these principles in the context of the following simple experiment. Let a GA be applied to P_{old}. During a genetic search on P_{old}, the system saves individuals in the population to the case base as described above. Note that we are saving partial solutions with differing fitnesses to the case base. P_{old} is changed slightly to P_{new}, and we look at the effect of injecting individuals saved at different generations of P_{old} into the initial population of the GA trying to solve P_{new}. If cases from P_{old} contain good building blocks or partial solutions, the GA can use these building blocks or schemas to quickly approach a solution to P_{new}.

Fig. 2 generalizes the situation to the case of n problems being solved sequentially. Initially, the case base is empty. When confronted with P_0, the system starts with a randomly initialized GA but generates cases to be stored in the case base. The CIGAR attempting P_1 injects cases from P_0 and generates cases from P_1 for the case base. When attempting P_2, CIGAR injects cases from the case base which now contains cases from both P_0 and P_1. In addition, while working on P_2, CIGAR generates cases from P_2 to be added to the case base, and so on. Thus, when attempting P_n, CIGAR can potentially use cases generated by P_0 through P_{n-1} for injection into P_n's population.

Using this methodology with just two problems, P_{old} and P_{new}, we use a CIGAR$_p$ to design a parity checker and a circuit similar to the parity checker. Three observations surface [15], [16].

1) As the distance between problems increases, injecting or seeding P_{new} with cases that had lower fitness in P_{old} results in better performance on P_{new} than when seeding P_{new} with cases that had higher fitness in P_{old}.
2) The above tendency is amplified with increasing problem size.
3) Injecting cases with higher fitness in P_{old} tends to lead to a quicker flattening out of the performance curve (average or maximum fitness) versus time than when injecting cases with lower fitness individuals in P_{old}.

The interested reader can access the references above for more details but we can summarize as follows. Problem similarity measures tend to be domain (or problem) specific and we will need to devise a different similarity metric for every domain. One way to deal with this issue is by abstracting the similarity metric computation to a separate module, thus making it easier to customize our system to difference domains. Note, however, that even an inexact problem similarity measure can be accommodated if we cover our bases and save and inject a set of *individuals with different fitnesses saved at different generations* of the GA's run on P_{old}. If the solutions to the problems are "further" apart, the individuals from earlier generations[1] will probably help more than if the solutions to the problems are "closer." In this situation, individuals from later generations will help more. Since we are injecting a variety of individuals from P_{old} saved at different points during the evolution of P_{old}'s solution, we are covering our bases and letting the GA decide which of these injected cases are useful and which are not. The GA takes care of culling those individuals that do not contribute to the solution of the problem being solved. If none of the injected individuals are useful, selection quickly removes them from the population and the (relatively large) randomly initialized component is used as the starting point for the search and the system reverts to a GA. If any injected individuals contribute to the problem being solved, the GA's performance increases.

However, GAs are applied in "poorly understood" domains where domain-dependent problem similarity metrics may be hard to design. A case-injected GA that did not use a

[1]Lower fitness is associated with earlier generations.

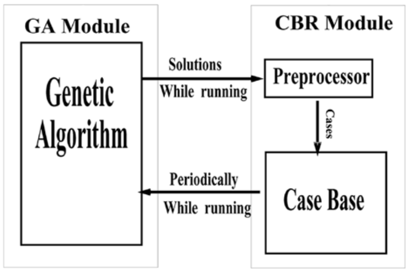


Fig. 3. Conceptual view of CIGAR_s.

problem-dependent similarity measure for indexing cases in its case base would be more widely (and perhaps more easily) applicable than the system described above. CIGAR_p assumes that "similar problems have similar solutions." This is a strong assumption but it seems to work well on the many problem domains we have considered. If we turn the statement around and consider that "similar solutions must solve similar problems," we have the foundation for a CIGAR system that works on the basis of solution similarity.

B. Solution Similarity

When CIGAR operates on the basis of solution similarity (CIGAR_s), we periodically inject a small number of *solutions* similar to the current best member of the GA population into the *current* population, replacing the worst members. The GA continues searching with this combined population. Note that Fig. 3 shows cases being *periodically* injected into the evolving population. The idea is to cycle through the following steps. Let the GA make some progress. Next, find solutions in the case base that are similar to the current best solution in the population and inject these solutions into the population. If injected solutions contain useful cross-problem information, the GA's performance will be significantly boosted. Fig. 4 shows this situation for CIGAR_s when it is solving a sequence of problems. In contrast to Fig. 2, each $P_i, 0 < i \leq n$ undergoes a periodic injection of cases. We have described one particular implementation of such a system. Other less elitist approaches for choosing population members to replace are possible, as are different strategies for choosing individuals from the case base. We can also vary the injection percentage, the fraction of the population replaced by chosen injected cases.

We have to periodically inject solutions based on the makeup of the current population because we do not know which previously solved problems are similar to the current one. We do not have a problem similarity metric. By finding and injecting cases in the case base that are similar to the current best individual in the population, we are assuming that similar solutions must have come from similar problems and that these similar solutions retreived from the case base contain useful information to guide genetic search.

An advantage of using solution similarity arises from the string representations typically used by GAs. A chromosome is, after all, a string of symbols. String similarity metrics are relatively easy to come by, and, furthermore, are domain independent. For example, in this paper, we use hamming distance between binary encoded chromosomes for the similarity metric whether our system is designing a circuit or optimizing resource allocations.

What happens if our similarity measure is noisy and/or leads to unsuitable retrieved cases? By definition, unsuitable cases will have low fitness and will quickly be eliminated from the GA's population. CIGAR may suffer from a slight performance hit in this situation but will not break or fail; the genetic search component will continue making progress toward a solution. In addition, note that diversity in the population, "the grist for the mill of genetic search [2]," can be supplied by the genetic operators **and** by injection from the case base. Even if the injected cases are unsuitable, variation is still injected. CIGAR is robust.

The system that we have described injects individuals in the case base that are deterministically closest, in hamming distance, to the current best individual in the population. We can also choose schemes other than injecting the closest to the best. For example, we have experimented with injecting cases that are the furthest (in the case base) from the current worst member of the population. Probabilistic versions of both have also proven effective.

Reusing old solutions has been a traditional performance improvement procedure. This work differs in that we: 1) 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.

Pseudocode for the CIGAR_s algorithm is given.

```
t = 0;
Initialize $P(t)$;
While (not termination condition)
Begin
 if((t% injectPeriod) == 0)
  InjectFromCaseBase(P(t), CaseBase);
Select P(t + 1) from P(t);
 Crossover P(t + 1);
 Mutate P(t + 1);
t = t + 1;
 if (NewBest(P(t)))
  CacheNewBestIndividual(P(t), Cache);
End
SaveCacheToCaseBase(Cache, CaseBase).
```

The code in boldface differentiates CIGAR_s code from that of the canonical GA. In every injectPeriod iteration, we choose and inject cases from the case base into the GA's population. These injected cases replace the worst members of the current population. Whenever there is an increase in the GA population's maximum fitness (a new best individual has been found), we set aside (or cache) this new best individual to be later stored into the case base. Just before we exit, we store these cached individuals into the case base. This pseudocode shows that changes that need to be made to the canonical GA are quite small and our results show that these changes lead to significant improvements in performance. The rest of this paper only deals with this CIGAR_s.

III. Related Work

One early attempt at reuse can be found in Ackley's work with SIGH [17]. Ackley periodically restarts a search in an

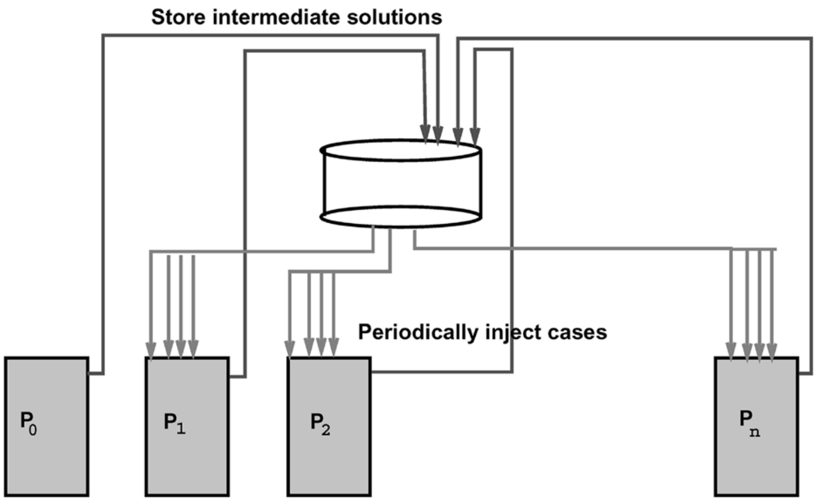


Fig. 4. Solving problems in sequence with CIGAR_s. Note the muliple periodic injections in the population as CIGAR attempts problem $P_i, 0 < i \leq n$.

attempt to avoid local optima and increase the quality of solutions. Eshelman's algorithm, a GA with elitist selection and cataclysmic mutation, also restarts a search when the population diversity drops below a threshold [18]. Other related work includes Angeline's compression and expansion operators [19], Koza's automatically defined functions [20], and Schoenauer's constraint satisfaction method [21]. More recently, Ramsey and Grefenstette come closest to our approach by using previously stored solutions to initialize a GA's initial population and thus increase a GA's performance in an anytime learning environment that changes with time [22], [23]. 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. Reynolds' cultural algorithms have also extracted and reused domain knowledge, represented in various forms including version spaces, to solve problems [24]. Sheppard and Salzburg combined memory-based learning with GAs in finding better plans for the pursuer–evader game within a reinforcement learning framework [25]. Their experiments indicate that the combined approach performed better than either approach alone. To our knowledge, the earliest work in combining GAs and case-based reasoning was done by Louis *et al.* who used case-based reasoning principles to explain solutions found by a GA search [14]. Preliminary work in this area solved pairs of similar problems with a clear performance advantage for the combined system [16]. These approaches, except for Reynolds' cultural algorithms, 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.

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 [26]–[28]. 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 for sequential tasks. Parallel GAs using the island model provide one possible avenue toward information exchange on related tasks in parallel for GA-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 [29], [30]. 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 [31].

CIGAR makes the assumption that similar solutions must have come from similar problems. Here, similarity means different things in different contexts. What do we mean by problem similarity? What is the difference between problem similarity and solution similarity? The next few sections address these questions.

IV. 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 [32]. 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 GA applications, the problem is compounded because we usually apply GAs to poorly understood problems where domain specific similarity metrics may be harder to find. In this paper, we study the more realistic case where we do *not* need a similarity measure on the problem space. Since GA solutions are encoded as binary or real strings, purely syntactic similarity measures lead to surprisingly good performance, a necessary property for applications to poorly understood systems. Solution similarity measures avoid the traditional CBR's issue of coming up with a domain dependent similarity metric.

The next three sections describe our three testbed problems. We start with combinational logic design, an example of the more general configuration design problem. We use this problem to delineate the difference between problem and solution similarity. Subsequently, we describe an asset allocation problem which uses a similar positional representation as the first problem. Finally, the job shop scheduling problem (JSSP) offers an example of a problem that is suited to an order-based encoding where where the order of the allelles matters. Order-based encodings, as we shall see, require a different similarity metric than do the positional representations used in the first two problems.

V. Combinational Circuit Design

The general problem can be stated as follows: Given a function and a target technology to work within, design an artifact that performs the specified 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 GA can be used to design combinational circuits as described in [33]. A five-bit parity checker is one of $2^{2^5} = 2^{32} = 4\,294\,967\,296$ 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) is given.

First, concatenate the output bit for all possible five-bit input combinations counting from 0 through 31 in binary. This results in a binary string of output bits S_0 of length 32. We call this binary string of output bits the output string. Strings that are one bit different from S_0 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*. Specifically, let P_i and P_j be two n-bit Boolean functions and let O_i and O_j be their corresponding output strings. Then, the **problem** distance between P_i and P_j is the hamming distance between O_i and O_j. That is, the distance between P_i and P_j is

$$\mathcal{D}(P_i, P_j) = \sum_{k=0}^{k=2^n} O_i^k \oplus O_j^k$$

where the $\oplus$ represents the exclusive or operator and O_i^k represents the kth bit of the output string. The equation above counts the number of differences in bit values at corresponding positions in O_i and O_j.

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, five-bit problems would have a maximum fitness of $2^5 = 32$.

As a simple example, consider the three-bit parity checker as shown in Table I. The first column in Table I shows all possible inputs for a three-bit parity checker, the second column shows the correct output for each input. We construct a three-input one-output problem that is similar to the three-bit parity checker by randomly choosing and flipping one of the output bits of the three-bit parity checker as shown by the boxed 0 in the table. The correct output bits for the new problem (one bit different from the three-bit parity checker) are shown in the third column. This new problem is a distance of 1 away from the parity problem in terms of our problem similarity metric.

TABLE I
Outputs of the Three-bit Parity Checker (a 3–0) and (a 3–1) Problem

Inputs	3-bit parity problem	3-1 problem
000	0	0
001	1	$\boxed{0}$
010	1	1
011	0	0
100	1	1
101	0	0
110	0	0
111	1	1

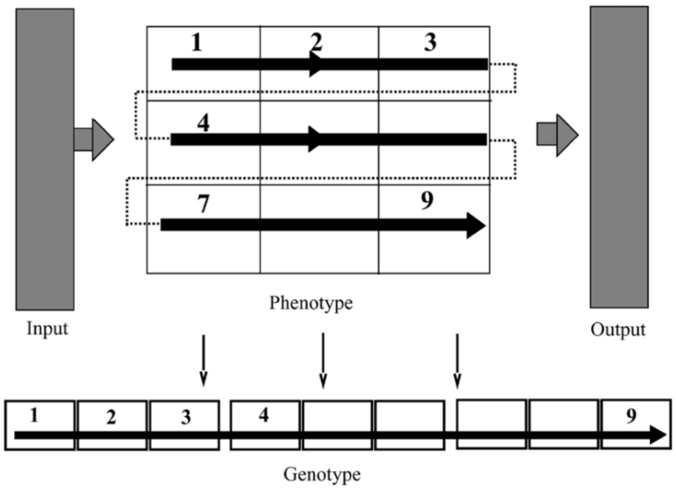


Fig. 5. Mapping from 1-D chromosome to 2-D circuit.

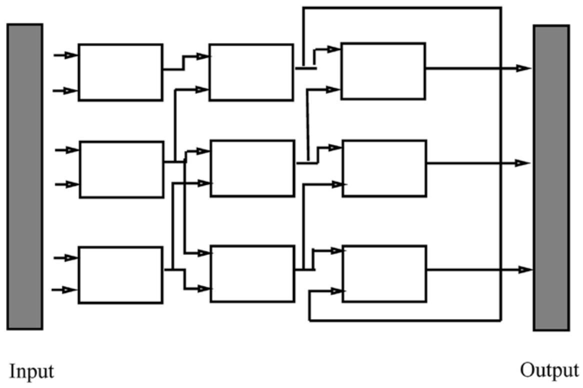


Fig. 6. Gate in a 2-D template gets its second input from one of two gates in the previous column.

A. 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 (2-D) circuit as shown in Figs. 5 and 6. We use four (4) bits to encode all 16 possible two-input one-output gates. An additional bit specifies the location of the second input. A gate $G_{i,j}$ gets its first input from $G_{i,j-1}$ and its second from one of $G_{i+1,j-1}$ or $G_{i-1,j-1}$, as shown in Fig. 6. 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 $G_{i,0}$ receive the input to the circuit.

B. Solution Similarity

The solution similarity metric needs to measure the distance between encoded circuits (solutions). We are assuming that similar circuits have a similar function. Whether this is a reasonable assumption for our representation of a circuit will be seen in Section VIII. Since we use a binary string to encode combinational circuits, the *hamming distance*, the number of bits that are different, between these bit strings suffices as our similarity metric on this problem. Real number strings could use Euclidean distance. That is, let A and B be two encoded binary chromosomes representing candidate solutions (circuits). Then, if l is the length of the chromosome, our solution similarity distance metric is given by

$$\mathcal{D}(A,B) = \sum_{i=0}^{i=l} A_i \oplus B_i \tag{1}$$

where the $\oplus$ represents the exclusive or operator and A_i represents the ith bit of chromosome A. That is, the hamming distance is simply the number of positions at which the chromosomes differ. We explain how we use this solution similarity distance metric in Section VIII. In the next section, we describe the asset allocation problem and its encoding. Although this domain is very different from configuration design, we use the same solution similarity measure, *hamming distance* between binary encoded representations of an allocation.

VI. Strike Force Asset Allocation

The problem essentially is to allocate a collection of strike force assets to a set of targets. An air-strike package typically consists of attack, fighter support, suppression of enemy air defenses (SEAD), and command and control (C^2) platforms. The problem is dynamic and various unforseen factors may require a quick reallocation of assets. GAs usually require many objective function evaluations in order to come up with a promising solution, thus implying that the GA may be unable to provide a viable solution within stringent time limits. CIGAR learns from experience, either offline or online, and reduces the time needed to obtain a quality solution. Under battlefield conditions with strict time limits, a CIGAR system, thus, provides a promising alternate approach to pure GAs for this problem.

An air-strike package consists of a variety of platforms, each of which has a specific role in the mission. A platform's effectiveness depends on the assets loaded on the platform and proficiency of the pilot. Our mathematical formulation of the problem is based on the work done by Abrahams [34]. We modified the problem formulation to accommodate pilot (or smart munitions) performance capabilities and relax the single platform-to-objective constraint that had been imposed. The objective function now consists of three terms. $U(X)$ measures an allocation's effectiveness, $V(X)$ measures the marginal benefit of allocating multiple assets to one target, and $Y(X)$ measures the risk to the platform in achieving the objective. The objective function to be maximized is a weighted sum of these measures.

The optimizer's goal is to select the most effective platforms $P_1,\ldots,P_m$ to achieve the targeting objectives $T_1,\ldots,T_n$. The platform allocation strategy is represented as an $m \times n$ matrix X where $x_{ij} = 1$ if platform P_i is assigned to objective T_j and zero otherwise. A platform can be assigned to more than one objective up to a maximum of M_i objectives that a platform P_i can accommodate.

Each platform has a load-out, the set of loaded assets, that is configured prior to launch. The total number of assets in the overall strike package is represented as a set of assets $A = A_1,\ldots,Aq$. The binary variable Γ_{ij} represents the assignment of the jth asset to the ith platform. Once the platforms are allocated to targeting objectives, the optimizer assigns the most effective weaponry from the existing load-out to achieve the desired effectiveness. This assignment of resources may be represented as a binary $n \times q$ matrix Ω that indicates asset A_j has been allocated to objective T_i.

The proficiency of the pilot of platform P_i in using an on-board asset A_j is represented as δ_{ij}. This term can also accommodate the usage of smart munitions that have characteristics independent of a particular pilot's capabilities. If w_{ij} represents the effectiveness (probability) of asset A_j achieving objective T_i, then, assuming statistical independence between onboard assets in achieving the targeting objectives, the probability of platform P_i achieving objective T_k is

$$p_{ik} = 1 - \prod_{j=1}^{q}(1 - \Gamma_{ij}\delta_{ij}\Omega_{kj}w_{jk})$$

where q is the total number of assets in the strike package.

The overall probability of an objective T_k being achieved by all platforms in the strike package is given by

$$s_k = 1 - \prod_{i=1}^{m}(1 - x_{ik}p_{ik})$$

where m is the number of platforms. We also incorporate a priority term ρ_i for each objective T_i and include it in the objective function. Thus, the effectiveness $U(X)$ of an allocation is given by

$$U(X) = \sum_{j=1}^{n} \rho_j s_j$$

where n is the number of objectives.

The coupling of assets to jointly achieve an objective can provide a marginal benefit and is described by the term $V(X)$

$$V(X) = \sum_{j=1}^{n}\sum_{k1=1}^{q}\sum_{k2=1}^{q} \Phi_{jk1k2}\Omega_{jk1}\Omega_{jk2}$$

where Φ_{jk1k2} defines joint effectiveness obtained by simultaneously assigning assets A_{k1} and A_{k2} to the same objective, T_i. It is important to note that Φ can be negative as well as positive.

The risk $Y(X)$ to the platforms, as well as their value, should also be addressed in the objective function. If each platform P_i

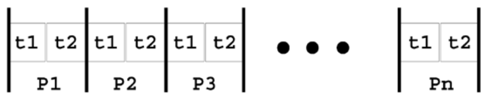


Fig. 7. Individual represents an allocation of platforms to targets.

has an associated value v_i, and the risk associated with allocating platform P_i to objective T_j is r_{ij}, then the cost term

$$Y(X) = \sum_{i=1}^{m} v_i \sum_{j=1}^{n} r_{ij} x_{ij}$$

needs to be included in our objective function.

Our combined objective function measures the effectiveness of the strike as well as the risks and costs entailed in carrying out the mission. We want to maximize effectiveness and minimize risk. Combining the effectiveness terms $U(X)$ and $V(X)$ with the risk/cost component $Y(X)$ yields the function to be optimized

$$J(X) = \alpha U(X) + \beta V(X) - \gamma Y(X). \tag{2}$$

$J(X)$ should be maximized in order to maximize the effectiveness while minimizing the risk. In our work, we set α, β, and γ to 1.0.

A. Representation

Each individual was encoded in a binary string composed from substrings corresponding to each platform (see Fig. 7). Each substring encodes the targets to which the platform has been allocated and the concatenation of these platform substrings encodes the allocation of platforms to targets for the strike force. As shown in Fig. 7, each platform P_i was allocated to two targets t_1 and t_2.

B. Solution Similarity

The solution similarity metric that CIGAR used for this problem was once again the *hamming distance*. We expect hamming distance (or euclidean for real encodings) to suffice for the positional encodings used both for circuit design and for strike force asset allocation. Letting A and B be two binary chromosomes representing candidate solutions to the strike force asset allocation problem, our solution similarity distance metric is given by (1) and is reproduced as

$$\mathcal{D}(A, B) = \sum_{i=0}^{i=l} A_i \oplus B_i$$

where l is the chromosome length for this problem. That these two very different problems can use the same similarity metric strongly indicates the domain-independent nature of our approach. In other words, both problems have binary positional encodings and can therefore use the same distance metric; other than the evaluation function, no changes are needed to use CIGAR on this problem.

The JSSP, however, is an example of a problem that uses an order-based encoding. Hamming distance does not pay attention to order and thus will not be a suitable distance metric for this problem. The next section uses the JSSP to propose and evaluate a solution similarity distance metric for order-based encodings.

VII. Job Shop Scheduling (JSSP)

The JSSP has been well studied elsewhere in the GA literature [35]–[38]. In the general $j \times m$ JSSP, there are j jobs and m machines. Each job contains an ordered set of tasks each of which must be done on a different machine for a different amount of time. Our objective function is to minimize makespan, the total time from the beginning of the first task until the end of the last task.

A. Representation

We encode a candidate schedule according to the scheme proposed by Fang [36] and our chromosome is a string of integers. Fang's method envisions a chromosome as a series of operations to be scheduled, where each gene ranges from one to the number of jobs. A separate list is maintained of uncompleted operations for each job. A gene's value i is interpreted by the scheduler as: schedule the next uncompleted task of the ith job. This task is then removed from the list. The scheduler views the ith uncompleted job modulo the list size, so as jobs are completed and the length of the list shrinks, the ith job will vary. Thus, no gene represents, by its value, any specific job, and no chromosome is ever illegal.

Hamming distance suffices when using positional encodings. Our JSSP encoding is order based since the effect of a decoded allelle depends on the preceding sequence of allelle values. Hamming distance does not capture this ordering information. What similarity metric does?

B. Solution Similarity

We use the longest common subsequence (LCS) similarity measure described in [39]. A subsequence of a given sequence is just the sequence with some elements (possible none) missing. Given two sequences X and Y, we say Z is a common subsequence of X and Y if Z is a subsequence of both X and Y. For example, if $X = ABCDABDB$ and $Y = ADAABCDB$, then $Z = ADABD$ is a common subsequence of X and Y, although it is not the longest common subsequence. We use the dynamic programming solution to the problem of finding the longest such subsequence given in the book. This measure runs in $O(n^2)$ time where n in this case is the length of the chromosome. The longer the LCS, the more similar the two encoded solutions. We find LCS to be a suitable similarity metric since it pays attention to the sequence (or order) of alleles in our order-based encoding of a schedule.

Recently, Hart and Ross [37] proposed a new heuristic combination method approach that they referred to as a heuristically guided GA (HGA). HGA evolves not only the selection heuristic but also the algorithmic method used to generate the conflict set. The two algorithms they used were the Giffler and Thompson (GT) and nondelay (ND), a variant of GT based on the set of nondelay schedules that creates a test set of the earliest possible operations. They reported very good results on a set of dynamic JSSPs when compared to previous work. Although we are aware of this work, we decided to use an order-based encoding (like Fangs') for the JSSP because our objective in this paper is to define and evaluate similarity metrics related to the goal of investigating CIGAR's learning behavior. We have also looked at order-based encodings and CIGAR in the context of the traveling salesperson problem and obtained results qualitatively like those reported here.

VIII. Results

For each of the three 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 which was at the limit of our computational capacity in obtaining reasonable turnaround times. We start by describing our expectations, as guided by our mathematical framework, then characterize problem generation for each problem and our results with CIGAR.

A. Expectations

According to our theoretical framework, for 50 problems, we need to show

$$\lim_{j \to 50} P^q_{\text{CIGAR}}(t_i, S, f_i, \mathcal{D}, e_j) \geq P^q_{\text{GA}}(t_i, S, f_i, \mathcal{D}) \qquad (3)$$

and

$$\lim_{j \to 50} P^t_{\text{CIGAR}}(t_i, S, f_i, \mathcal{D}, e_j) \leq P^t_{\text{GA}}(t_i, S, f_i, \mathcal{D}) \qquad (4)$$

for the 50 problems in our set to provide evidence that CIGAR is learning from experience. Our graphs show this by the following.

1) Plotting the number of problems solved on the x axis versus the best fitness found by the algorithm on the y axis. We would expect that P^q_{CIGAR} over the 50 problems attempted would be as good as or better (fitness is greater than or equal to) for all problems $P_i \in \{P_0 \dots P_{50}\}$ than P^q_{GA} over the same set of problems.
2) Plotting the number of problems solved on the x axis versus the time needed to find this best solution (item 1) from above) on the y axis. In this case, we would expect that P^t_{CIGAR} over the 50 problems attempted would be lower or the same as (less or equal time taken) for all problems $P_i \in \{P_0 \dots P_{50}\}$ than P^t_{GA} over the same set of problems.

We use a "probabilistic closest to the best" injection strategy to choose individuals from the case base to be injected into CIGAR's population. Here, the probability of a case being chosen for injection into the population is proportional to its similarity to the current best individual in the population. More formally, let I be the current best individual in the population, then the probability of a case C in the case base being selected for injection is

$$\text{Prob}(C) = \frac{l - \mathcal{D}(I, C)}{\sum_{i=1}^{i=n} (l - \mathcal{D}(I, J))}$$

where l is the chromosome length, n is the number of cases in the case base, and $\mathcal{D}$ is given by (1) for binary encoded chromosomes. Since $\mathcal{D}$ is the hamming distance between two chromosomes, $(l - \mathcal{D})$ provides the similarity between the two chromosomes.

The results provide empirical evidence concerning the above relations [(3) and (4)] from the domains of combinational logic design, asset allocation, and job shop scheduling.

B. Combinational Circuit Design

Using the parity problem as a basis, we generate 50 problems that are similar in terms to our problem similarity metric by randomly choosing and flipping between 10 – 20 (uniform distribution) output bits of the six-bit parity checker's $2^6 = 64$-bit output bit string. We use all $2^4 = 16$ two-input one-output logic gates as possible design elements and therefore need four bits to represent a gate in our encoding. An additional bit locates the specified logic gate's second input and we therefore use five bits to represent each logic gate.

The set of tasks T, defined in Section II for this problem, is the set of six-bit input one-bit output combinational logic circuits. There are $2^{2^6} = 18\,446\,744\,073\,709\,551\,616$ possible Boolean functions (or fitness functions, F) that can be defined for six-input and one-output circuits. The 50 Boolean functions for which CIGAR attempts to design circuits belong to this much larger set.

CIGAR uses a population of size 30 runs for a maximum of 30 generations to solve these six-bit combinational circuit design problems (similar to parity checkers). This results in a 150 length chromosome (six rows, five columns, five bits per gate) with a corresponding search space of 2^{150}. Thus, for each $t \in T$, the search space S's size is 2^{150}. The probability of crossover is 0.95 and the probability of mutation is 0.05. All plots are averages over ten runs.

In all cases, we replace the three (10% of the population size) 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 three cases into the population every $\lceil \log_2(30) \rceil = 5$ generations to replace the three worst individuals.

The injection percentage, which determines the number of injected cases (and the number of lowest fitness individuals replaced), and the injection intervals are adjustable parameters to the case-injected GA. Injection percentage controls the influence of injected cases; the higher the injection percentage the more genetic search is driven by past experience. We also expect less diversity and faster convergence with higher injection percentages.

Fig. 8 compares performance in terms of time taken to find a solution for CIGAR and a randomly initialized GA that uses the same GA parameters. 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 parameters. Fig. 9 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 (3) and (4) hold

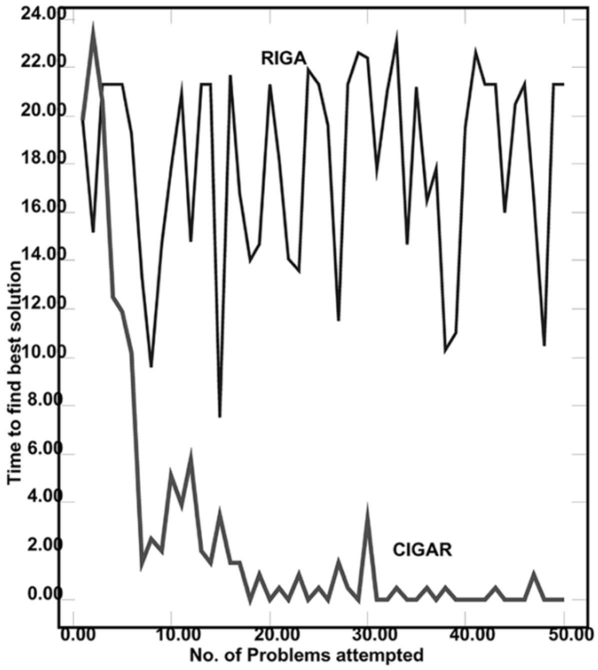


Fig. 8. Time to best design found. As more problems are solved, CIGAR takes less time.

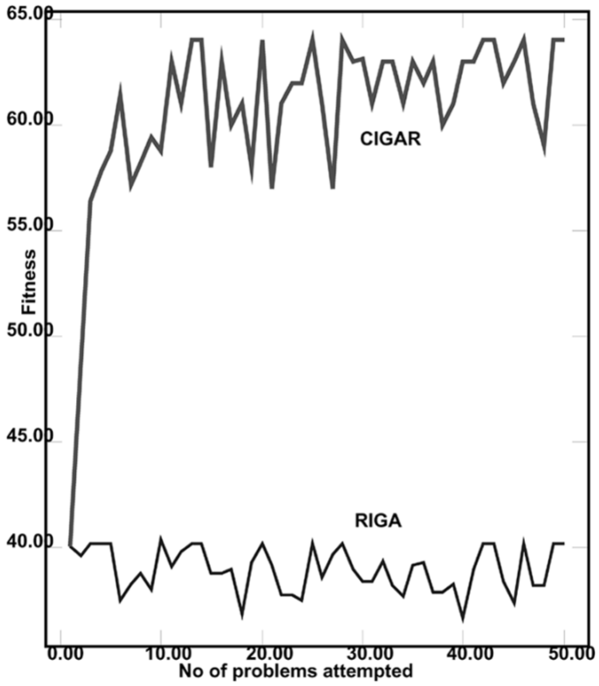


Fig. 9. Solution quality. As more problems are solved, CIGAR produces better solutions.

for this problem. That is, the figures provide clear evidence that CIGAR takes less time then the GA to produce better quality solutions as it gains experience from attempting more problems; thus, CIGAR learns from experience.

C. Asset Allocation

The problem configuration that we constructed here had ten platforms, ten targets, and 40 assets. Each platform could be allocated to two targets. We need four bits to associate a unique id for each of ten targets and thus each platform's substrings consists of $4 \times 2 = 8$ bits and for ten targets we end up with a chromosome length of $8 \times 10 = 80$ bits (Fig. 7).

To generate problems, we start by constructing a baseline asset allocation problem A_0 using fixed values for all platform, pilot, target, and asset properties. Risk was the only factor that varied. We generated a risk matrix by initializing the diagonal elements of the matrix to 0.0 and all other elements to 1.0. Thus, the optimal allocation for this initial problem was platform P_i to target T_i. We generate A_1 by randomly choosing two rows or two columns and swapping them and repeating this process $(mn)^{0.5}$ times. This ensures that there is exactly one 0.0 entry in every row and column and all other entries are 1.0, thus keeping the problem simple. Continuing in this way, using A_1 as a basis for A_2, and A_2 as a basis for A_3, and so on, we generate 50 problems A_0 through A_{49}. Note that problems close to each other in the sequence are probabilistically more similar than problems farther apart in the sequence.

The set of task T, defined in Section II for this problem, is the set of 10×10 asset allocation problems. In the general case, there is an infinite number of possible problems, since the risk matrix is a matrix of probabilities, but restricting risk (r_{ij}) to be between 0.00 and 0.99 inclusive (a precision of two decimal places) gives us the size for S to be 100^{10}.

As with the design problem, we used the CHC selection algorithm with a crossover rate of 1.0 coupled with point mutations with a mutation rate of 0.05. A population size of 80 was used and the GA ran for 80 iterations; CHC converges fast without a highly disruptive crossover operator and 80 generations was enough for convergence with two-point crossover. The randomly initialized GA used exactly the same parameters as CIGAR.

CIGAR used the "probabilistic closest to the best" strategy for case injection and we injected 10% of the population with individuals from the case base every nine generations.

Fig. 10 compares the time to convergence of a GA versus CIGAR while Fig. 11 compares the quality of the converged solution produced by a GA versus that produced by CIGAR. Specifically, Fig. 10 plots average time taken to find the best solution on the vertical axis and the number of problems solved on the horizontal axis. Fig. 11 plots average objective function value on the vertical axis and the number of problems solved on the horizontal axis.

The thin straight lines on both graphs are the linear regression lines for the distributions represented by our performance metrics. These regression lines provide a simple model to predict performance trends. The figures show that, once again, CIGAR takes less time to produce better quality solutions as its problem solving experience increases. We have similar results from using other injection strategies and other sizes of problems.

Compared with the circuit design problem, we see less of an increase in performance for both of our criteria. The structure of the search spaces are different and we should therefore expect to see such differences across problem domains.

D. Job Shop Scheduling

We first construct a JSSP J_1 with randomly generated task durations. Each of the 50 similar problems are then generated by modifiying a randomly picked set of tasks (40% of the total number of tasks of the problem) and then changing the picked

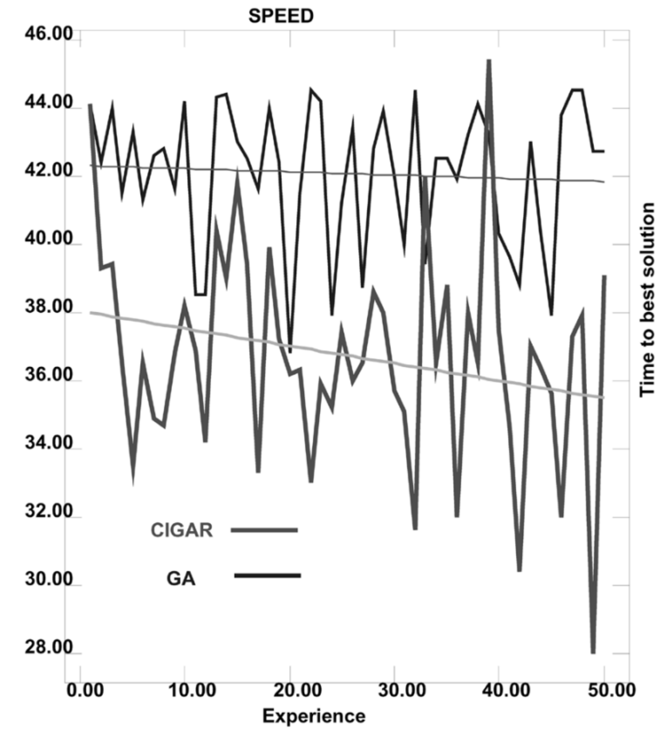


Fig. 10. Time to best solutions: GA versus CIGAR.

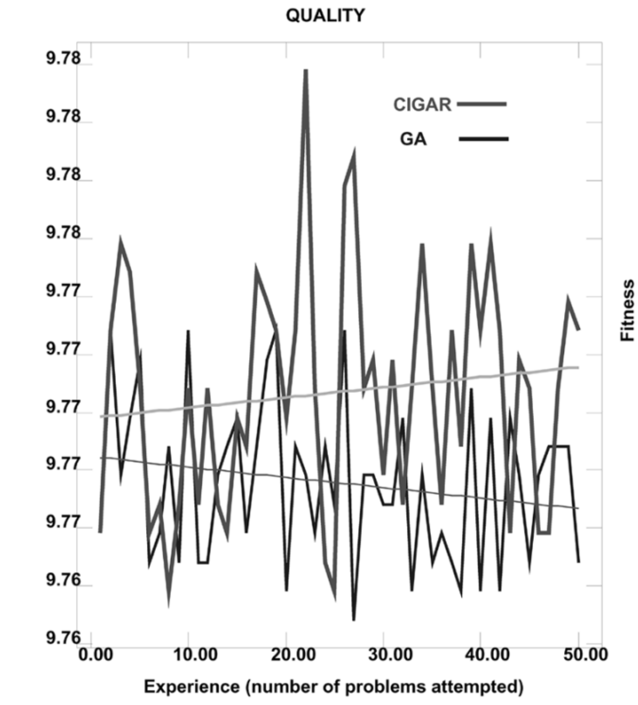


Fig. 11. Solution quality: GA versus CIGAR.

tasks' durations by a randomly picked amount upto 20% of the maximum duration of a task. Each of the 50 problems is generated by modifying J_1 as described above.

The set of tasks T, according to our formulation, is the set of 15×15 JSSP problems. As with asset allocation, in the general case there is an infinite number of possible problems, since task durations can be real numbers. Restricting task durations to d leads to us to estimate the cardinality of T to be $d^{15\times 15}$.

We used a population of size 100 and ran the algorithms for 100 generations. We needed larger population sizes to get good

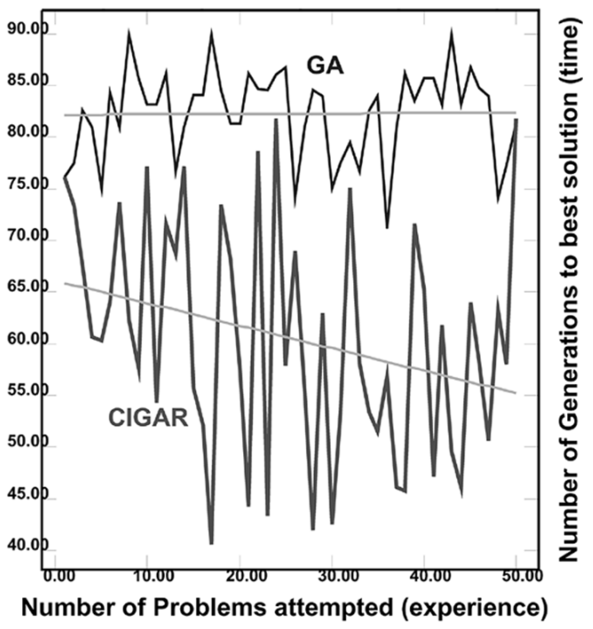


Fig. 12. Convergence speed for JSSP. As more problems are attempted, CIGAR reduces the time taken to achieve convergence.

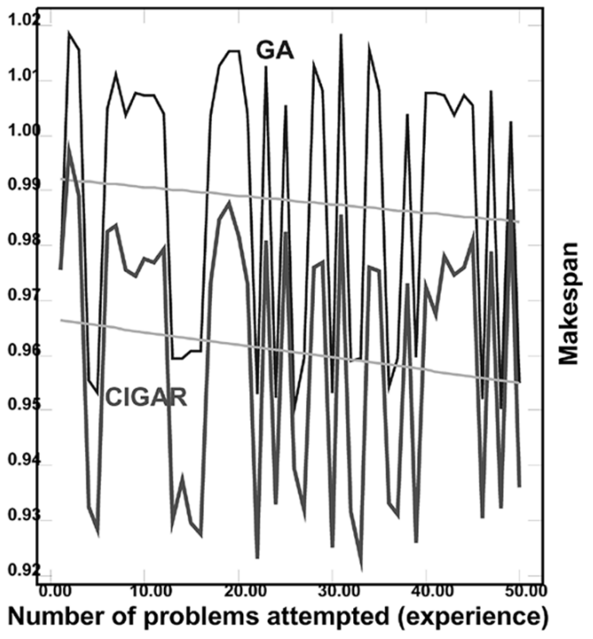


Fig. 13. Convergence quality for JSSP. At the same time, as more problems are attempted, CIGAR produces better quality solutions.

performance. Again, we used the CHC selection strategy [18] of greedy crossover [40] with a probability of 1.0 and swap mutation with probability 0.5 that an individual would be mutated round out the parameters used.

CIGAR used the "probabilistic closest to the best" strategy for case injection in the following results and we injected 10% of the population with individuals from the case base every ten generations.

Figs. 12 and 13 compare a randomly injected GA with CIGAR on 15×15 JSSPs. Fig. 12 plots the time taken to find the best solution on the vertical axis and the number of problems solved on the horizontal axis. Fig. 13 plots the makespan of the best solution found on the vertical axis and number of problems solved on the horizontal axis.

The thin straight lines on both graphs are the linear regression lines for the distributions represented by our performance metrics, time to convergence, and quality of solution at convergence.

E. Summary of Results

The mathematical framework developed in Section II provided two indicators of learning performance. Measuring these indications, empiricial results from circuit design, asset allocation, and job shop scheduling indicate that CIGAR learns to increase performance with experience. Furthermore, although an increase in solution quality or time to best solution would have sufficed to show an advantage, CIGAR does better than a randomly initialized GA on **both** criteria for these three problems.

Given our encodings, we believe that combinational logic design is an easier problem compared to job shop scheduling or asset allocation for the GA since low population sizes (30) run for a small number of generations (30) lead to acceptable performance. When we look at performance across problems we thus see a marked difference in the graphs for combinational logic circuit design versus those for strike force asset allocation and job shop scheduling. The difference in performance between CIGAR and the GA is greater for combinational logic design than for the other two problems. It appears that, in this domain, CIGAR finds it easier to obtain and reuse domain information to quickly improve performance over the GA.

The graphs indicate that although the trend for increasing performance may be clear, for example, from the regression lines, it is difficult to predict CIGAR's relative difference in performance between problems. The performance on a particular problem depends on past experience (what problems have been solved in the past) and the relationship of those problems to the current problem. If CIGAR is able to retreive and store relevant cases from past problems, we expect a performance boost on the current problem relative to a GA starting from scratch. We can also expect an increase in performance if the cases that CIGAR injects add useful diversity to the population.

These results show that whether a problem uses positional or order-based encodings to represent solutions, we can use simple domain independent similarity metrics to effectively guide CIGAR's case injection.

IX. Discussion and Conclusion

This paper makes three contributions. We defined a framework for machine learning in the context of search algorithms, described the algorithm (CIGAR) that combines genetic search with case-based memory to learn to increase performance with experience, and defined similarity metrics for indexing cases for positional and order-based encodings. Empirical evidence from three different application areas show the viability of our approach when compared to a GA that starts from scratch on every new problem in an application domain. The results show that CIGAR performance improves with experience in related problems as defined by our framework.

We conjecture that, like CIGAR, properly combining a robust search algorithm with some implementation of an associative memory can result in a learning system that learns, with experience, to solve similar problems quickly and robustly. In our implementation, members of the population form the cases stored in the case base and provide CIGAR with a long-term associative memory of problem-solving experience. CIGAR thus combines the strengths of GAs and case-based reasoning and uses domain-independent indexing schemes to do so. The cases chosen by the indexing scheme and injected into the evolving GA population provide random variation and variation based on past experience. This proves useful in increasing CIGAR performance relative to a GA on similar problems.

We defined two similarity metrics for indexing: hamming distance for positional encodings and the LCS for order-based encodings. Empirical evidence from three problems using these metrics indicates their relative effectiveness as domain-independent similarity metrics for GA applications. In our implementation, you can set flags to determine the distance metric (hamming distance or longest common substring) and injection strategy to be used by the system. The results indicate that these distance metrics work well with the probabilistic closest to the best injections strategy on the problems addressed in this paper. We have similar results on other problems as well [41], [42].

CIGAR's performance depends on two parameters. For injection percentage, injecting between 5%–15% of the population size with individuals from the case base strikes a good balance between exploration of the search space and concentrating search in promising areas defined by injected individuals. This injection percentage parameter can be tuned to suit a particular domain. There are several injection strategies that can be used to choose cases from the case base for injection. 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. This parameter too can be tuned to a particular domain. Work that we have done indicates that the probabilistic versions tend to get better performance and that injecting too many individuals into an evolving population tends to quickly lead to lower quality solutions [16], [43].

The case-injected GA technique described and evaluated in this paper provides a simple effective method for increasing GA performance across similar problems. Our results have shown the potential of combining GAs and case-based memory but much work remains to be done. We are beginning investigation of problem structure, population distibutions, operator effects, and injection parameters to characterize problems and predict their suitability to the CIGAR approach. We are also interested in investigating similarity metrics for variable length encodings. Finally, we are applying and evaluating case-injected Gas on several real-world applications and investigating whether problem sequences in the real-world result in performance improvements for case-injected GAs.

Acknowledgment

The authors would like to thank the reviewers for their insightful feedback.

References

[1] J. Holland, *Adaptation in Natural and Artificial Systems*. Ann Arbor, MI: Univ. of Michigan Press, 1975.

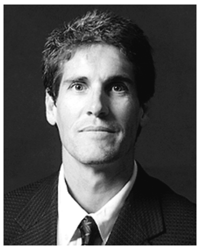
[2] D. E. Goldberg, *Genetic Algorithms in Search, Optimization, and Machine Learning*. New York: Addison-Wesley, 1989.

[3] D. Smith, "Bin packing with adaptive search," in *Proc. Int.Conf. Genetic Algorithms*, 1985, pp. 202–206.
[4] C. Z. Janikow, "A knowledge-intensive genetic algorithm for supervised learning," *Machine Learning*, vol. 13, pp. 189–228, 1993.
[5] J. Grefenstette, C. Ramsey, and A. Shultz, "Learning sequential decision rules using simulation models and competition," *Mach. Learn.*, vol. 5, pp. 355–381, 1990.
[6] J. Mostow, M. Barley, and T. Weinrich, "Automated reuse of design plans in Bogart," in *Artificial Intelligence in Engineering Design, Vol II*, C. Tong and D. Sriram, Eds. New York: Academic, 1992, pp. 57–104.
[7] M. Huhns and R. Acosta, "Argo: An analogical reasoning system for solving design problems," in *Artificial Intelligence in Engineering Design, Vol II*, C. Tong and D. Sriram, Eds. New York: Academic, 1992, pp. 105–144.
[8] K. Sycara and D. Navinchandra, "Retrieval strategies in case-based design system," in *Artificial Intelligence in Engineering Design, Vol II*, C. Tong and D. Sriram, Eds. New York: Academic, 1992, pp. 145–164.
[9] A. Goel and B. Chandresekaran, "Case-based design: A task analysis," in *Artificial Intelligence in Engineering Design, Vol II*, C. Tong and D. Sriram, Eds. New York: Academic, 1992, pp. 165–184.
[10] C. K. Riesbeck and R. C. Schank, *Inside Case-Based Reasoning*. Cambridge, MA: Lawrence Erlbaum, 1989.
[11] R. C. Schank, *Dynamic Memory: A Theory of Reminding and Learning in Computers and People*. Cambridge, MA: Cambridge Univ. Press, 1982.
[12] D. B. Leake, *Case-Based Reasoning: Experiences, Lessons, and Future Directions*. Menlo Park, CA: AAAII/MIT Press, 1996.
[13] T. M. Mitchell, *Machine Learning*. Boston, MA: WCB McGraw-Hill, 1997.
[14] S. J. Louis, G. McGraw, and R. Wyckoff, "Case-based reasoning assisted explanation of genetic algorithm results," *J. Exper. Theoret. Artif. Intell.*, vol. 5, pp. 21–37, 1993.
[15] X. Liu, "Combining genetic algorithm and case-based reasoning for structure design," M.S. dissertation, Univ. Nevada, Reno, 1996.
[16] S. J. Louis and J. Johnson, "Solving similar problems using genetic algorithms and case-based memory," in *Proc. 7th Int. Conf. Genetic Algorithms*, San Mateo, CA, 1997, pp. 283–290.
[17] D. A. Ackley, *A Connectionist Machine for Genetic Hillclimbing*. New York: Kluwer Academic, 1987.
[18] L. J. Eshelman, "The CHC adaptive search algorithm: How to have safe search when engaging in nontraditional genetic recombination," in *Foundations of Genetic Algorithms-1*, G. J. E. Rawlins, Ed. San Diego, CA: Morgan Kauffman, 1991, pp. 265–283.
[19] P. J. Angeline and J. B. Pollack, "Coevolving high-level representations," in *Artificial Life III*, C. G. Langton, Ed. Santa Fe, NM: Addison-Wesley, 1994, vol. XVII, pp. 55–71.
[20] J. R. Koza, *Genetic Programming*. Cambridge, MA: MIT Press, 1993.
[21] M. Schoenauer and S. Xanthakis, "Constrained ga optimization," in *Proc. 5th Int. Conf. Genetic Algorithms*, San Mateo, CA, 1993, pp. 573–580.
[22] C. Ramsey and J. Grefensttete, "Case-based initialization of genetic algorithms," in *Proc. 5th Int. Conf. Genetic Algorithms*, S. Forrest, Ed., San Mateo, CA, 1993, pp. 84–91.
[23] J. Grefensttete and C. Ramsey, "An approach to anytime learning," in *Proc. 9th Int. Conf. Machine Learning*, San Mateo, CA, 1992, pp. 189–195.
[24] C.-J. Chung and R. G. Reynolds, "A testbed for solving optimization problems using cultural algorithms," *Evol. Programm.*, pp. 225–236, 1996.
[25] J. W. Sheppard and S. L. Salzburg, "Combining genetic algorithms with memory based reasoning," in *Proc. 6th Int. Conf. Genetic Algorithms*, S. Forrest, Ed., San Mateo, CA, 1995, pp. 452–459.
[26] R. Caruana, "Multitask learning: A knowledge-based source of inductive bias," *Mach. Learn.*, vol. 28, pp. 41–75, 1997.
[27] R. Caruana, S. Baluja, and T. Mitchell, "Using the future to sort out the present, rankprop and multitask learning for medical risk prediction," in *Proc. Advances Neural Information Processing Systems 8*, 1996, pp. 959–965.
[28] R. Caruana, "Multitask learning," Ph.D. dissertation, Carnegie Mellon Univ., Pittsburgh, PA, 1997.
[29] S. Thrun, *Explanation-Based Neural Network Learning: A Lifelong Learning Approach*. Amsterdam, The Netherlands: Kluwer, 1996.
[30] ——, "Is learning the n-th thing any easier than learning the first," in *Proc. Advances in Neural Information Processing Systems 8*, 1996, pp. 640–646.
[31] S. J. Louis and G. Li, "Combining robot control strategies using genetic algorithms with memory," *Lecture Notes Computer Science, Evolutionary Programming VI*, vol. 1213, pp. 431–442, 1997.
[32] S. J. Louis and J. Johnson, "Solving similar problems using genetic algorithms and case-based memory," in *Proc. 7th Int. Conf. Genetic Algorithms*, 1997, pp. 101–127.
[33] S. J. Louis and G. J. E. Rawlins, "Designer genetic algorithms: Genetic algorithms in structure design," in *Proc. 4th Int. Conf. Genetic Algorithms*, San Mateo, CA, 1991, pp. 53–60.
[34] P. Abrahams, R. Balart, J. S. Byrnes, D. Cochran, M. J. Larkin, W. Moran, G. Ostheimer, and A. Pollington, "Maap: The military aircraft allocation planner," in *Evolutionary Computation Proc. IEEE World Congr. Computational Intelligence*, 1998, pp. 336–341.
[35] R. Nakano, "Conventional genetic algorithms for job shop problems," in *Proc. 4th Int.l Conf. Genetic Algorithms*, R. K. Belew and L. Booker, Eds., San Mateo, CA, 1991, pp. 474–479.
[36] H.-L Fang, P. Ross, and D. Corne, "A promising genetic algorithm approach to job shop scheduling, rescheduling, and open shop scheduling problems," in *Proc. 5th Int. Conf. Genetic Algorithms*, S. Forrest, Ed., San Mateo, CA, 1993, pp. 375–382.
[37] E. Hart and P. M. Ross, "A heuristic combination method for solving jop-shop scheduling problems," in *Parallel Problem Solving from Nature V*, A. E. Eiben, T. Back, M. Schoenauer, and H. Schwefel, Eds. New York: Springer–Verlag, 1998, vol. 1498, pp. 845–854.
[38] Z. Xu and S. J. Louis, "Genetic algorithms for open shop scheduling and re-scheduling," in *Proc. ISCA 11th Int. Conf. Computers Their Applications*, 1996, pp. 99–102.
[39] T. H. Cormen, C. E. Leiserson, and R. L. Rivest, *Introduction to Algorithms*. Cambridge, MA: MIT Press, 1990.
[40] J. Grefenstette, R. Gopal, R. Rosmaita, and D. Gucht, "Genetic algorithms for the traveling salesman problem," in *Proc. Int. Conf. Genetic Algorithms*, Mahwah, NJ, 1985.
[41] S. J. Louis and G. Li, "Augmenting genetic algorithms with memory to solve traveling salesman problems," in *Proc. 3rd Joint Conf. Information Sciences*, P. P. Wang, Ed., 1997, pp. 108–111.
[42] S. J. Louis and Y. Zhang, "A sequential similarity metric for case injected genetic algorithms applied to tsps," in *Proc. Genetic Evolutionary Computation Conf.*, vol. 1, W. Banzhaf, J. Daida, A. E. Eiben, M. H. Garzon, V. Honavar, M. Jakiela, and R. E. Smith, Eds., Orlando, FL, 13–17, 1999, pp. 670–677.
[43] S. J. Louis, "Genetic learning for combinational logic design," *J. Soft Computing*, to be published.

Sushil J. Louis (M'01) received the Ph.D. degree from Indiana University, Bloomington, in 1993.

He is an Associate Professor and Director of the Evolutionary Computing Systems Laboratory, Department of Computer Science, University of Nevada, Reno.

Dr. Louis is a member of the ACM.



John McDonnell (M'89) received the Ph.D. degree from Texas A&M University, College Station, in 1985.

He is a Program Manager at the Space and Naval Warfare Systems Center, San Diego, CA. He is currently developing decision support tools for dynamic resource allocation and other applications.