Playing to Learn: Case-Injected Genetic Algorithms for Learning to Play Computer Games

Sushil J. Louis, *Member, IEEE,* and Chris Miles

***Abstract*—We use case-injected genetic algorithms (CIGARs) to learn to competently play computer strategy games. CIGARs periodically inject individuals that were successful in past games into the population of the GA working on the current game, biasing search toward known successful strategies. Computer strategy games are fundamentally resource allocation games characterized by complex long-term dynamics and by imperfect knowledge of the game state. CIGAR plays by extracting and solving the game's underlying resource allocation problems. We show how case injection can be used to learn to play better from a human's or system's game-playing experience and our approach to acquiring experience from human players showcases an elegant solution to the knowledge acquisition bottleneck in this domain. Results show that with an appropriate representation, case injection effectively biases the GA toward producing plans that contain important strategic elements from previously successful strategies.**

***Index Terms*—Computer games, genetic algorithms, real-time strategy.**

I. Introduction

THE COMPUTER gaming industry is now almost as big as the movie industry and both gaming and entertainment drive research in graphics, modeling, and many other computer fields. Although AI and evolutionary computing research has been interested in games like checkers and chess [1]–[6], popular computer games such as Starcraft and Counter-Strike are very different and have not received much attention. These games are situated in a virtual world, involve both long-term and reactive planning, and provide an immersive, fun experience. At the same time, we can pose many training, planning, and scientific problems as games in which player decisions bias or determine the final solution.

Developers of computer players (game AI) for popular first-person shooters (FPS) and real-time strategy (RTS) games tend to acquire and encode human-expert knowledge in finite state machines or rule-based systems [7], [8]. This works well, until a human player learns the game AI's weaknesses, and requires significant player and developer time to create competent players. Development of game AI, thus, suffers from the knowledge acquisition bottleneck that is well known to AI researchers.

This paper, in contrast, describes and uses a case-injected genetic algorithm (CIGAR) that combines genetic algorithms (GAs) with case-based reasoning to competently play a computer strategy game. The main task in such a strategy game is to continuously allocate (and reallocate) resources to counter opponent moves. Since RTS games are fundamentally about solving a sequence of resource allocation problems, the GA plays by attempting to solve these underlying resource allocation problems. Note that the GA (or human) is **attempting** to solve resource allocation problems with no guarantee that the GA (or human) will find the optimal solution to the current resource allocation problem—quickly finding a good solution is usually enough to get good game-play.

Case injection improves the GA's performance (quality and speed) by periodically seeding the evolving population with individuals containing good building blocks from a case-based repository of individuals that have performed well on previously confronted problems. Think of this case-base as a repository of past experience. Our past work describes how to choose appropriate cases from the case-base for injection, how to define similarity, and how often to inject chosen cases to maximize performance [9].

This paper reports on results from ongoing work that seeks to develop competent game opponents for tactical and strategic games. We are particularly interested in automated methods for modeling human strategic and tactical game play in order to develop competent opponents and to model a particular doctrine or "style" of human game-play. Our long-term goal is to show that evolutionary computing techniques can lead to robust, flexible, challenging opponents that learn from human game-play. In this paper, we develop and use a strike force planning RTS game as a testbed (see Fig. 1) and show that CIGAR can: 1) play the game; 2) learn from experience to play better; and 3) learn trap avoidance from a human player's game play.

The significance of learning trap avoidance from human game-play arises from the system having to learn a concept that is external to the evaluation function used by CIGAR. Initially, the system has no concept of a trap (the concept) and has no way of learning about traps through feedback from the evaluation function. Therefore, the problem is for the system to acquire knowledge about traps and trap-avoidance from humans and then to learn to avoid traps. This paper shows how the system "plays to learn." That is, we show how CIGAR uses cases acquired from human (or system) game-play to learn to avoid traps without changing the game and the evaluation function.

Section II introduces the strike force planning game and CIGARs. Section III then describes previous work in this area. Section IV describes the specific strike scenarios used for testing, the evaluation computation, the system's architecture,

Manuscript received September 23, 2004; revised February 19, 2005. This work was supported in part by the Office of Naval Research under Contract N00014-03-1-0104.

The authors are with the Department of Computer Science, University of Nevada, Reno, NV 89557-0148 USA (e-mail: sushil@cse.unr.edu; miles@cse.unr.edu).

Digital Object Identifier 10.1109/TEVC.2005.856209

1089-778X/$20.00 © 2005 IEEE

Fig. 1. Game screenshot.

and the encoding. Sections V and VI describe the test setup and results with using CIGAR to play the game and to learn trap-avoidance from humans. Section VII provides conclusions and directions for future research.

II. Strike Force Planning

Strike force asset allocation maps to a broad category of resource allocation problems in industry and, thus, makes a suitable test problem for our work. We want to allocate a collection of assets on platforms to a set of targets and threats on the ground. The problem is dynamic; weather and other environmental factors affect asset performance, unknown threats can "popup," and new targets can be assigned. These complications as well as the varying effectiveness of assets on targets make the problem suitable for evolutionary computing approaches.

Our game involves two sides: Blue and Red, both seeking to allocate their respective resources to minimize damage received, while maximizing the effectiveness of their assets in damaging the opponent. Blue plays by allocating a set of assets on aircraft (platforms), to attack Red's buildings (targets) and defensive installations (threats). Blue determines which targets to attack, which weapons (assets) to use on them, as well as how to route platforms to targets, trying to minimize risk presented, while maximizing weapon effectiveness.

Red has defensive installations (threats) that protect targets by attacking Blue platforms that come within range. Red plays by placing these threats to best protect targets. Potential threats and targets can also pop up on Red's command in the middle of a mission, allowing a range of strategic options. By cleverly locating threats, Red can feign vulnerability and lure Blue into a deviously located popup trap, or keep Blue from exploiting such a weakness out of fear of a trap. The scenario in this paper involves Red presenting Blue with a trapped corridor of seemingly easy access to targets.

In this paper, a human plays Red, while a genetic algorithm player (GAP) plays Blue. GAP develops strategies for the attacking strike force, including flight plans and weapons targeting for all available aircraft. When confronted with popups, GAP responds by replanning in order to produce a new plan of action that responds to the changes. Beyond purely responding to immediate scenario changes we use case injection in order to produce plans that anticipate opponent moves. We provide a short introduction to CIGAR next.

A. Case-Injected Genetic Algorithms (CIGARs)

A CIGAR works differently than a typical GA. A GA randomly initializes its starting population so that it can proceed from an unbiased sample of the search space. We believe that it makes less sense to start a problem solving search attempt from scratch when previous search attempts (on similar problems) may have yielded useful information about the search space. Instead, periodically injecting a GA's population with *relevant* solutions or partial solutions to similar previously solved problems can provide information (a search bias) that reduces the time taken to find a quality solution. Our approach borrows ideas from case-based reasoning (CBR) in which old problem and solution information, stored as cases in a case-base, helps solve a new problem [10]–[12]. 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 GA's attempts at solving a problem.

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 percentage (say 10%–15%) of the initial population. A case is a member of the population (a candidate solution) together with other information including its fitness and the generation at which this case was generated [13]. During GA search, whenever the fitness of the best individual in the population increases, the new best individual is stored in the case-base.

Like CIGAR, human players playing the game are also solving resource allocation and routing problems. A human player's asset allocation and routing strategy is automatically reverse engineered into CIGAR's chromosomal representation and stored as a case into the case-base. Such cases embody domain knowledge acquired from human players.

The case-base does what it is best at—memory organization; the GA handles what it is best at—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 system.

The CIGAR used in this paper operates on the basis of solution similarity. CIGAR **periodically injects** 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. Apart from using solution similarity, note that one other distinguishing feature from the "problem-similarity" metric CIGAR is that cases are periodically injected. 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. Then, let the GA make some progress, and repeat the previous steps. The detailed algorithm can be found in [9]. If injected solutions contain useful cross

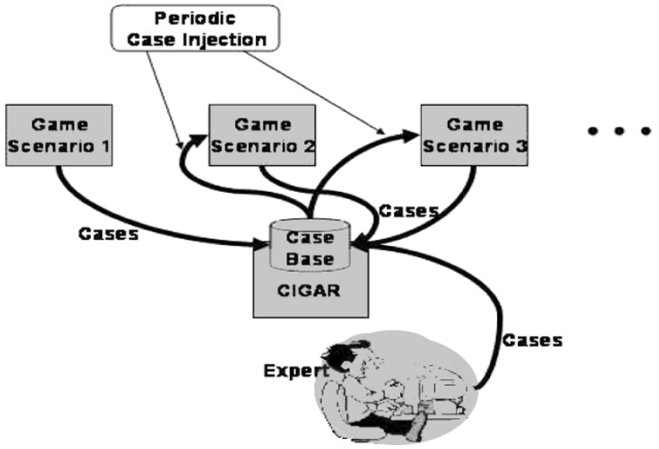

Fig. 2. Solving problems in sequence with CIGAR. Note the multiple periodic injections in the population as CIGAR attempts problem $P_i, 0<i \leq n$.

problem information, the GA's performance will be significantly boosted. Fig. 2 shows this situation for CIGAR when it is solving a sequence of problems $P_i, 0 < i \leq n$, each of which undergoes 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.

CIGAR has to periodically inject cases because we do not know which previously solved problems are similar to the current one. That is, we do not have a problem similarity metric. However, the Hamming distance between binary encoded chromosomes provides a simple and remarkably effective solution similarity metric. We, thus, find and inject cases that are similar (close in Hamming distance) to the current best individual. Since the current best individual changes, we have to find and inject the closest cases into the evolving population. We are assuming that similar solutions must have come from similar problems and that these similar solutions retrieved from the case-base contain useful information to guide genetic search. Although this is an assumption, results on design, scheduling, and allocation problems show the efficacy of this similarity metric and, therefore, of CIGAR [9].

An advantage of using solution similarity arises from the string representations typically used by GAs. A chromosome is a string of symbols. String similarity metrics are relatively easy to create and compute, and furthermore, are domain independent.

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 [14]" 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. The system that we have described injects individuals from 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. The CIGAR approach 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. CIGAR pseudocode and more details are provided in [9].

B. CIGAR for RTS Games

Within the context of RTS games as resource allocation problems, GAs can usually robustly search for effective strategies. These strategies usually approach static game optimal strategies but they do not necessarily approach optima in the real world as the game is an imperfect reflection of reality. For example, in complex games, humans with past experience "playing" the real-world counterpart of the game tend to include external knowledge when producing strategies for the simulated game. Incorporating knowledge from the way these humans play (through case injection) should allow us to carry over some of this external knowledge into GAP's game play. Since GAP can gain experience (cases) from observing and recording human Blue-players' decisions as well as from playing against human or computer opponents, case injection allows GAP to use current game-state information as well as acquired knowledge to play better. Our game is designed to record all player decisions (moves) on a central server for later storage into a case-base. The algorithm does not consider whether cases come from humans or from past game-playing episodes and can use cases from a variety of sources. We are particularly interested in acquiring and using cases from humans in order to learn to play with a specific human style and to be able to acquire knowledge external to the game.

Specifically, we seek to produce a GAP that can play on a strategic level and learn to emulate aspects of strategies used by human players. Our goals in learning to play like humans are the following.

- We want to use GAP for decision support, whereby GAP provides suggestions and alternative strategies to humans actively playing the game. Strategies more compatible with those being considered by the humans should be more likely to have a positive effect on the decision-making process.
- We would like to make GAP a challenging opponent to play against.
- We would like to use GAP for training. GAP plays not just to win but to teach its opponent how to better play the game, in particular to prepare them for future play against human opponents. This would allow us to use GAP for acquiring knowledge from human experts and transferring that knowledge to human players without the expense of individual training with experts.

These roles require GAP to play with objectives in mind besides that of winning—these objectives would be difficult to quantify inside the evaluator. As humans can function effectively in these regards, learning from them should help GAP better fulfill these responsibilities.

C. Playing the Game

A GA can generate an initial resource allocation (a plan) to start the game. However, no initial plan survives contact with the enemy.[1] The dynamic nature of the game requires replanning in response to opponent decisions (moves) and changing game-state. This replanning has to be fast enough to not interfere with the flow of the game and the new plan has to be good enough to win the game, or at least, not lose. Can GAs satisfy these speed and quality constraints? Initial results on small scenarios with tens of units showed that a parallelized GA on a small ten-node cluster runs fast enough to satisfy both speed and quality requirements. For the CIGAR, replanning is simply solving a similar planning problem. We have shown that CIGAR learns to increase performance with experience at solving similar problems [9], [15], [16]–[18]. This implies that when used for replanning, CIGAR should quickly produce better new plans in response to changing game dynamics. In our game, aircraft break off to attack newly discovered targets, reroute to avoid new threats, and reprioritize to deal with changes to the game state.

Beyond speeding up GAP's response to scenario changes through replanning, we use case injection in order to produce plans that anticipate opponent moves. This teaches GAP to act in *anticipation* of changing game states and leads to the avoidance of likely traps and better capitalization on opponent vulnerabilities. GAP learns to avoid areas likely to contain traps from two sources.

- Humans: As humans play the game, GAP adds their play to the case-base gaining some of their strategic knowledge. Specifically, whenever the human player makes a game move, the system records this move for later storage into the case-base. The system, thus, acquires knowledge from humans simply by recording their game-play. We do not need to conduct interviews, deploy concept maps, or use other expensive, error-prone, and lengthy knowledge-acquisition techniques.
- Experience: As GAP plays games it builds a case-base with knowledge of how it should play. Since the system does not distinguish between human players and GAP, it acquires knowledge from GAP's game-play exactly as described above.

Our results indicate GAP's potential in making an effective Blue player with the ability to quickly replan in response to changing game dynamics, and that case injection can bias GAP to produce good solutions that are suboptimal with respect to the game simulation's evaluation function but that avoid potential traps. Instead of changing evaluation function parameters or code, GAP changes its behavior by acquiring and reusing knowledge, stored as cases in a case-base. Case injection also biases the GA toward producing strategies similar to those learned from a human player. Furthermore, our novel representation allows the GA to reuse learned strategic knowledge across a range of similar scenarios independent of geographic location.

[1]We paraphrase from a quote attributed to Helmuth von Moltke.

III. Previous Work

Previous work in strike force asset allocation has been done in optimizing the allocation of assets to targets, the majority of it focusing on static premission planning. Griggs [19] formulated a mixed-integer problem (MIP) to allocate platforms and assets for each objective. The MIP is augmented with a decision tree that determines the best plan based upon weather data. Li [20] converts a nonlinear programming formulation into a MIP problem. Yost [21] provides a survey of the work that has been conducted to address the optimization of strike allocation assets. Louis [22] applied CIGARs to strike force asset allocation.

From the computer gaming side, a large body of work exists in which evolutionary methods have been applied to games [2]–[4], [23], [24]. However, the majority of this work has been applied to board, card, and other well-defined games. Such games have many differences from popular RTS games such as Starcraft, Total Annihilation, and Homeworld [25]–[27]. Chess, checkers and many others use entities (pieces) that have a limited space of positions (such as on a board) and restricted sets of actions (defined moves). Players in these games also have well-defined roles and the domain of knowledge available to each player is well identified. These characteristics make the game state easier to specify and analyze. In contrast, entities in our game exist and interact over time in continuous three-dimensional space. Entities are not controlled directly by players but instead sets of parametrized algorithms control them in order to meet goals outlined by players. This adds a level of abstraction not found in more traditional games. In most such computer games, players have incomplete knowledge of the game state and even the domain of this incomplete knowledge is difficult to determine. Laird [7], [8], [28] surveys the state of research in using AI techniques in interactive computers games. He describes the importance of such research and provides a taxonomy of games. Several military simulations share some of our game's properties [29], [30], however, these attempt to model reality, while ours is designed to provide a platform for research in strategic planning, knowledge acquisition and reuse, and to have fun. The next section describes the scenario being played.

IV. The Scenario

Fig. 3 shows an overview of our test scenario. We chose the scenario to be simple and easy to analyze but to still encapsulate the dynamics of traps and anticipation.

The translucent gray hemispheres show the effective radii of Red's threats placed on the game map. The scenario takes place in Northern Nevada, Walker Lake is visible near the bottom of the map covered by the largest gray hemisphere. Red has eight targets on the right-hand side of the map with their locations denoted by the cross-hairs. Red has a number of threats placed to

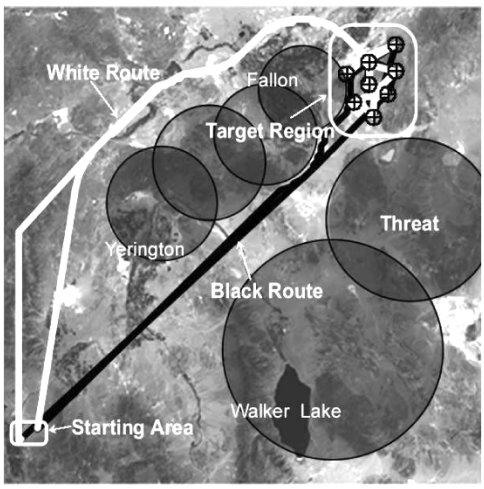

Fig. 3. The scenario.

defend the targets and the translucent gray hemispheres show the effective radii of some of these threats. Red has the potential to play a popup threat to trap platforms venturing into the corridor formed by the threats.

Blue has eight platforms, all of which start in the lower left-hand corner. Each platform has one weapon, with three classes of weapons being distributed among the platforms. Each of the eight weapons can be allocated to any of the four targets, giving $4^8 = 2^{16} = 65\,536$ allocations. This space can be exhaustively searched but more complex scenarios quickly become intractable.

In this scenario, GAP's router can produce three broad types of routes that we have named black, white, and gray (see Fig. 3).

1) Black—Flies through the corridor in order to reach the targets.
2) White—Flies around the threats, attacking the targets from behind.
3) Gray—Flies inside the perimeter of known threats (not shown in the figure).

Gray routes expose platforms to unnecessary risk from threats and, thus, receive low fitness. Ignoring popup threats, the optimal strategy contains black routes, which are the most direct routes to the target that still manage to avoid known threats. However, in the presence of the popup threat and our risk averse evaluation function, aircraft following the black route are vulnerable and white routes become optimal although they are longer than black routes. The evaluator looks only at known threats, so plans containing white routes receive lower fitness then those containing black routes. GAP should learn to anticipate traps and to prefer white trap-avoiding routes even though white routes have lower fitness than black routes.

In order to search for good routes and allocations, GAP must be able to compute and compare their fitnesses. Computing this fitness is dependent on the representation of entities' states inside the game, and our way of computing fitness and representing this state is described next.

A. Fitness

We evaluate the fitness of an individual in GAP's population by running the game and checking the game outcome. Blue's goals are to maximize damage done to red targets, while minimizing damage done to its platforms. Shorter simpler routes are also desirable, so we include a penalty in the fitness function based on the total distance traveled. This gives the fitness calculated, as shown in (1)

$$\text{Fitness(plan)} = \text{Damage(Red)} - \text{Damage(Blue)} - d * c \quad (1)$$

d is the total distance traveled by Blue's platforms and c is chosen such that $d * c$ has a 10%–20% effect on the fitness of a plan. Total damage done is calculated below

$$\text{Damage(Player)} = \sum_{E \in F} E_v * (1 - E_s)$$

E is an entity in the game and F is the set of all forces belonging to that side. E_v is the value of E, while E_s is the probability of survival for entity E. We use probabilistic health metrics to evaluate entity damage keeping careful track of time to ensure that the probabilities are calculated at appropriate times during game-play.

B. Probabilistic Health Metrics

In many games, entities (platforms, threats, and targets in our game) possess hit-points that represent their ability to take damage. Each attack removes a number of hit-points and the entity is destroyed when the number of hit-points is reduced to zero. In reality, weapons have a more hit or miss effect, destroying entities or leaving them functional. A single attack may be effective, while multiple attacks may have no effect. Although more realistic, this introduces a large degree of stochastic error into the game. Evaluating an individual plan can result in outcomes ranging from total failure to perfect success, making it difficult to compare two plans based on a single evaluation. Lacking a good comparison, it is difficult to search for an optimal strategy. By taking a statistical analysis of survival we can achieve better results.

Consider the state of each entity at the end of the mission as a random variable. Comparing the expected values for those variables allows judging the effectiveness of a plan. These expected values can then be estimated by executing each plan a number of times and averaging the results. However, doing multiple runs to determine a single evaluation increases the computational expense manyfold.

We use a different approach based on probabilistic health metrics. Instead of monitoring whether or not an object has been destroyed, we monitor the probability of its survival. Being attacked no longer destroys objects and removes them from the game, it just reduces their probability of survival according to (2)

$$S(E) = S_{t_0}(E) * (1 - D(E)) \quad (2)$$

E is the entity being considered, a platform, target, or threat. $S(E)$ is the probability of survival of entity E after the attack. $S_{t_0}(E)$ is probability of survival of E up until the attack and $D(E)$ is the probability of that platform being destroyed by the attack and is given by (3)

$$D(E) = S(A) * E(W). \quad (3)$$

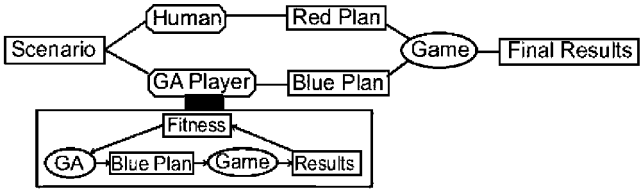

Fig. 4. System architecture.

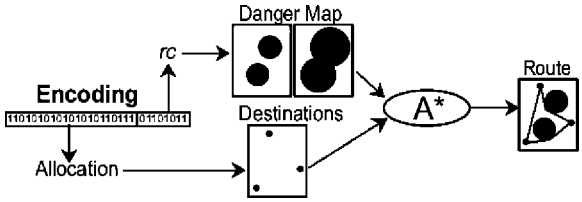

Fig. 5. How routes are built from an encoding.

Here, $S(A)$ is the attacker's probability of survival up until the time of the attack and $E(W)$ is the effectiveness of the attacker's weapon as given in the weapon-entity effectiveness matrix. Our method provides the expected values of survival for all entities in the game within one run of the game, thereby producing a representative evaluation of the value of a plan. As a side effect, we also gain a smoother gradient for the GA to search as well as consistently reproducible evaluations. We expect that this approach will work for games where a continuous approximation to discontinuous events (like death) does not affect game outcomes. Note that this approach does not yet consider: 1) ensuring that performance lies above a minimum acceptable threshold and 2) a plan's tolerance to small perturbations. Incorporating additional constraints is ongoing work, but for this paper the evaluation function described above provides an efficient approach to evaluating a plan's effectiveness for the GA.

The strike force game uses this approach to compute damage sustained by entities in the game. The gaming system's architecture reflects the flow of action in the game and is described next.

C. System Architecture

Fig. 4 outlines our system's architecture. Starting at the left, Red and Blue, human and GAP, respectively, see the scenario and have some initialization time to prepare strategy. GAP applies the CIGAR to the underlying resource allocation and routing problem and chooses the best plan to play against Red. The game then begins. During the game, Red can activate popup threats that GAP can detect upon activation. GAP then runs the CIGAR producing a new plan of action, and so on.

To play the game, GAP must produce routing data for each of Blue's platforms. Fig. 5 shows how routes are built using the A$*$ algorithm [31]. A$*$ builds routes between locations that platforms wish to visit, generally, the starting airbase and targets they are attacking. The A$*$ router finds the cheapest route, where cost here is a function of length and risk and leads to a preference for the shortest routes that avoid threats.

We parameterize A$*$ in order to represent and produce routes that are not dependent on geographical location and that have specific characteristics. For example, to avoid traps, GAP must

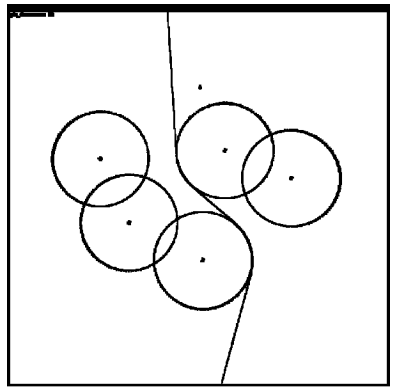

Fig. 6. Routing with $rc = 1.0$.

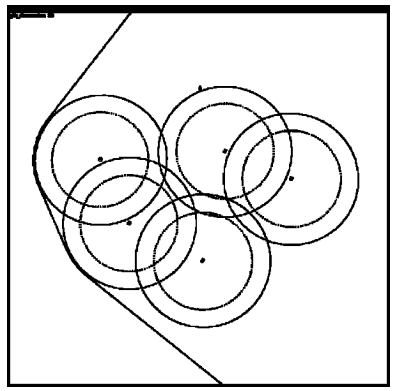

Fig. 7. Routing with $rc = 1.3$.

be able to specify that it wants to avoid areas of potential danger. In our game, traps are most effective in areas confined by other threats. If we artificially inflate threat radii, threats expand to fill in potential trap corridors and A$*$ produces routes that go around these expanded threats. We, thus, introduce a parameter, rc that encodes threats' effective radii. Larger rc's expand threats and fill in confined areas, smaller rc's lead to more direct routes. Figs. 6 and 7 show rc's effect on routing, as rc increases, A$*$ produces routes that avoid the confined area. In our scenarios, values of $rc < 1.0$ produce gray routes, values with $1.0 < rc < 1.35$ produce direct black routes, and values of $rc > 1.35$ produce white trap-avoiding routes. rc is limited currently to the range $[0, 3]$ and encoded with 8 bits at the end of our chromosome. We encoded a single rc for each plan but are investigating the encoding of rc's for each section of a route. Note that this representation of routes is location independent. We can store and reuse values of rc that have worked in different terrains and different locations to produce more direct or indirect routes.

D. Encoding

Most of the encoding specifies the asset-to-target allocation with rc encoded at the end as detailed earlier. Fig. 8 shows how we represent the allocation data as an enumeration of assets to targets. The scenario involves two platforms (P1, P2), each with a pair of assets, attacking four targets. The left box illustrates the allocation of asset A1 on platform P1 to target T3, asset A2 to target T1, and so on. Tabulating the asset-to-target allocation

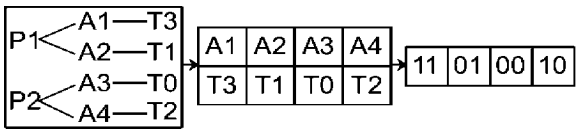

Fig. 8. Allocation encoding.

gives the table in the center. Letting the position denote the asset and reducing the target id to binary then produces a binary string representation for the allocation.

Earlier work has shown how we can use CIGAR to learn to increase asset allocation performance with experience [9] and we, therefore, focus more on rc and routing in this paper.

V. Learning to Avoid Traps

We address the problem of learning from experience to avoid traps using a two-part approach. First, from experience, we learn where traps are likely to be, then we apply that acquired knowledge and avoid potential traps in the future. Case injection provides an implementation of these steps: building a case-base of individuals from past games stores important knowledge. The injection of those individuals applies the knowledge toward future search.

GAP records games played against opponents and runs offline to determine the optimal way to win the previously played game. If the game contains a popup trap, genetic search progresses toward the optimal strategy in the presence of the popup and GAP saves individuals from this search into the case-base, building a case-base with routes that go around the popup trap—white routes. When faced with other opponents, GAP then injects individuals from the case-base, biasing the current search toward containing this learned anticipatory knowledge.

Specifically, GAP first plays our test scenario, likely picking a black route and falling into Red's trap. Afterward GAP replays the game, while including Red's trap. At this stage, black routes receive poor fitness and GAP prefers white trap-avoiding routes. Saving individuals to the case-base from this search stores a cross-section of plans containing "trap avoiding" knowledge.

The process produces a case-base of individuals that contain important knowledge about how we should play, but how can we use that knowledge in order to play smarter in the future? We use case injection when playing the game and periodically inject a number of individuals from the case-base into the population, biasing our current search toward information from those individuals. Injection replaces the worst members of the population with individuals chosen from the case-base through a "probabilistic closest to the best" strategy [9]. These new individuals bring their "trap avoiding" knowledge into the population, increasing the likelihood of that knowledge being used in the final solution and, therefore, increasing GAP's ability to avoid the trap.

A. Knowledge Acquisition and Application

Imagine playing a game and seeing your opponents do something you had not considered but that worked out to great effect. Seeing something new, you are likely to try to learn some of the dynamics of that move so you can incorporate it into your own play and become a better player. Ideally, you would have perfect understanding of when and where this move is effective and ineffective, and how to best execute it under effective circumstances. Whether the move is using a combination of chess pieces in a particular way, bluffing in poker, or doing a reaver drop in Starcraft, the general idea remains. In order to imitate this process, we use a two-step approach with case injection. First, we learn knowledge from human players by saving their decision making during game play and encoding it for storage in the case-base. Second, we apply this knowledge by periodically injecting these stored cases into GAP's evolving population.

B. Knowledge Acquisition

Knowledge acquisition is a significant problem in rule-based systems. GAP acquires knowledge from human Blue players by recording player plans, reverse engineering these plans into GA chromosomes, and storing these chromosomes as cases in our case-base. In the strike force game, we can easily encode the human player's asset allocation. Finding an rc that closely matches the route chosen by the human player may require a search but note that this reverse-engineering is done offline. When a person plays the game, we store all human moves (solutions) into the case-base. Injecting appropriate cases from a particular person's case-base biases the GA to generate candidate solutions that are similar to those from the player. Instead of interviewing an expert game player, deriving rules that govern the player's strategy and style, and then encoding them into a finite-state machine or a rule-base, our approach simply and automatically records player interactions, while playing the game, automatically transforms player solutions into cases, and uses these cases to bias search toward producing strategies similar to those used by the player. We believe that our approach is less expensive in that we do not need a knowledge engineer. In addition, we gain flexibility and robustness. For example, consider what happens when a system is confronted with an unexpected situation. In rule-based systems, default rules that may or may not be appropriate to the situation control game play. With CIGARs, if no appropriate cases are available, the "default" GA finds near-optimal player strategies.

C. Knowledge Application

Consider learning from a human who played a white trap-avoiding route, but had a nonoptimal allocation. The GA should keep the white route, but optimize the allocation unless the allocation itself was based on some external knowledge (a particular target might seem like a trap), in which case the GA should maintain that knowledge. Identifying which knowledge to maintain and which to replace is a difficult task even for human players. In this research, we thus use GAP to reproduce a simple but useful and *easily identifiable* aspect of human strategy: avoidance of confined areas.

VI. Results

We designed test scenarios that were nontrivial but tractable. In each of the test scenarios, we know the optimum solution and can, thus, evaluate GAP's performance against this known optimum. This allows us to evaluate our approach on a well-understood (known) problem. For learning trap-avoidance in the

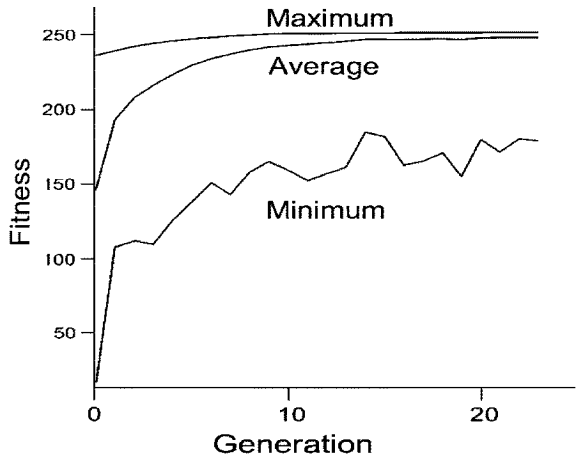

Fig. 9. Best/worst/average individual fitness as a function of generation—averaged over 50 runs.

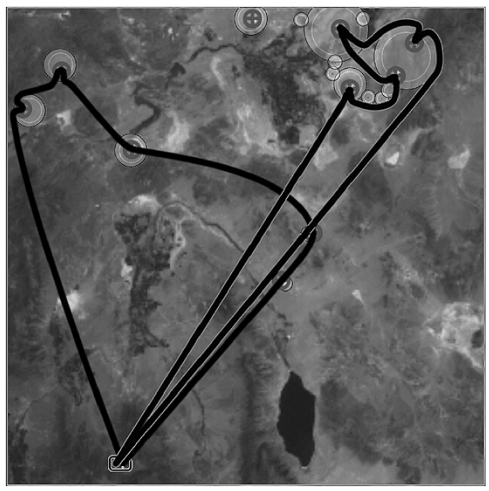

Fig. 10. Complex mission.

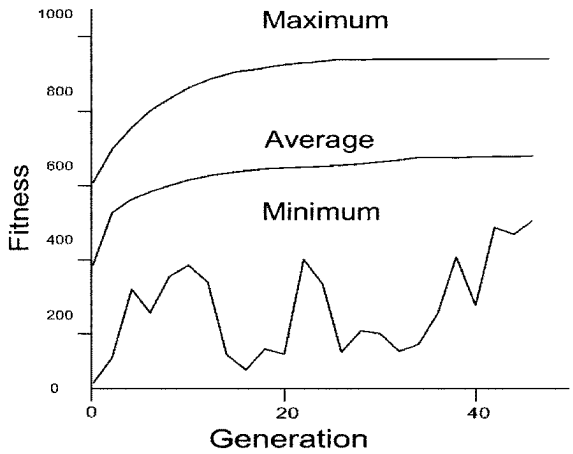

Fig. 11. Best/worst/average individual fitness as a function of generation—averaged over 50 runs on the complex mission.

presence of "popup" traps, human experts (the authors) played optimally and chose white trap-avoiding routes with an optimal asset allocation.

Plans consist of an allocation of assets to targets and a parameter to A$*$ (rc) that determines the route taken. For the scenarios considered, reverse-engineering a human plan into a chromosome is nontrivial but manageable. The human asset allocation can be easily reverse-engineered, but we have to search through rc values (offline) to find the closest chromosomal representation of the human route. We present results showing the following.

1) GAP can play the strike force asset allocation game effectively.
2) Replanning can effectively react to popups.
3) GAP can use case injection to learn to avoid traps.
4) GAP can use knowledge acquired from human players.
5) With our representation, acquired knowledge can be generalized to different scenarios.
6) Fitness biasing can maintain injected information in the search.

Unless stated otherwise, GAP uses a population size of 25, two-point crossover with a probability of 0.95, and point mutation with a probability of 0.01. We use elitist selection, where offspring and parents compete for population slots in the next generation [32]. Experimentation showed that these parameter values satisfied our time and quality constraints. Results are averages over 50 runs and are statistically significant at the 0.05 level of significance or below.

A. GAP Plays the Game

We first show that GAP can generate good strategies. GAP runs 50 times against our test scenario, and we graph the minimum, maximum, and average population fitness against the number of generations in Fig. 9. We designed the test scenario to have an optimum fitness of 250 and the graph in Fig. 9 shows a strong approach toward the optimum—in more the 95% of runs the final solution is within 5% of the optimum. This indicates that GAP can form effective strategies for playing the game.

B. Playing the Game—Complex Mission

Testing GAP on a more realistic/complex mission (in Fig. 10) leads to a similar effect shown in Fig. 11. This mission has a wider array of defenses, which are often placed directly on top of targets. Note that the first generation best is now much farther from the optimum compared with Fig. 9, but that the GA quickly makes progress. Sample routes generated by GAP to reach targets in the two widely separated clusters are shown and there are no popups in this mission.

C. Replanning

To analyze GAP's ability to deal with the dynamic nature of the game, we look at the effects of replanning. Fig. 12 illustrates the effect of replanning by showing the final route followed inside a game. A black (direct) route was chosen, and when the popup occurred, trapping the platforms, GAP redirected the strike force to retreat and attack from the rear. Replanning allows GAP to rebuild its routing information, as well as modify its allocation to compensate for damaged platforms. The A$*$ routing algorithm's cost function found that the lowest cost route was to retreat and go around the threats rather than simply fly through the popup. Using a different cost function

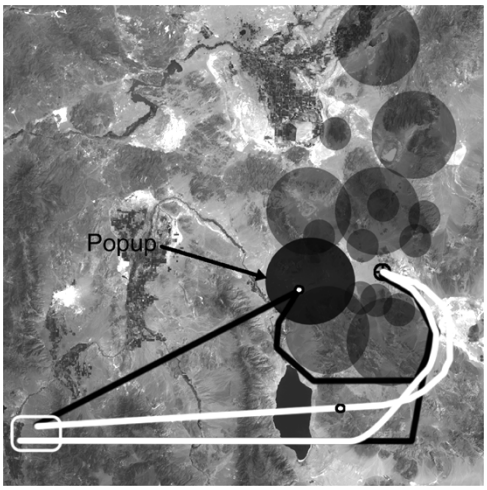

Fig. 12. Final routes used during a mission involving replanning.

may have allowed the mission to keep flying through the popup even in the new plan. The white route shown in the figure is explained next.

D. Learned Trap Avoidance

GAP learns to avoid traps through playing games offline. Specifically, GAP plays (or replays) games that it lost in order to learn how to avoid losing. In our scenario, during GAP's offline play, the popup was included as part of the scenario and cases corresponding to solutions that avoided the popup threat were stored in the case-base. GAP learns to avoid the popup trap through injection of these cases obtained from offline play. This is also shown in Fig. 12, where GAP, having learned from past experience, prefers the white trap-avoiding route.

GAP's ability to learn to avoid the trap can also be seen by looking at the numbers of black and white routes produced with and without case injection, as shown in Fig. 13. The figures compare the histograms of rc values produced by GAP with and without case injection. Case injection leads to a strong shift in the kinds of rc's produced, biasing the population toward using white routes. The effect of this bias is a large and statistically significant increase in the frequency at which strategies containing white routes were produced (2% to 42%). These results were based on 50 independent runs of the system and show that case injection does bias the search toward avoiding the trap.

E. Case Injection's Effect on Fitness

Fig. 14 compares the fitnesses with and without case injection. Without case injection the search shows a strong approach toward the optimal black-route plan; with injection the population quickly converges toward the white-route plan.

Case injection applies a bias toward white routes, however, the GA has a tendency to act in opposition to this bias, trying to search toward ever-shorter routes. GAP's ability to overcome the bias through manipulation of injected material depends on the size of the population and the number of generations run. We will come back to this later in the section.

Instead of gaining experience by replaying past games offline, we can also gain experience by acquiring knowledge from good players. Since we control the game's interface, it is a simple

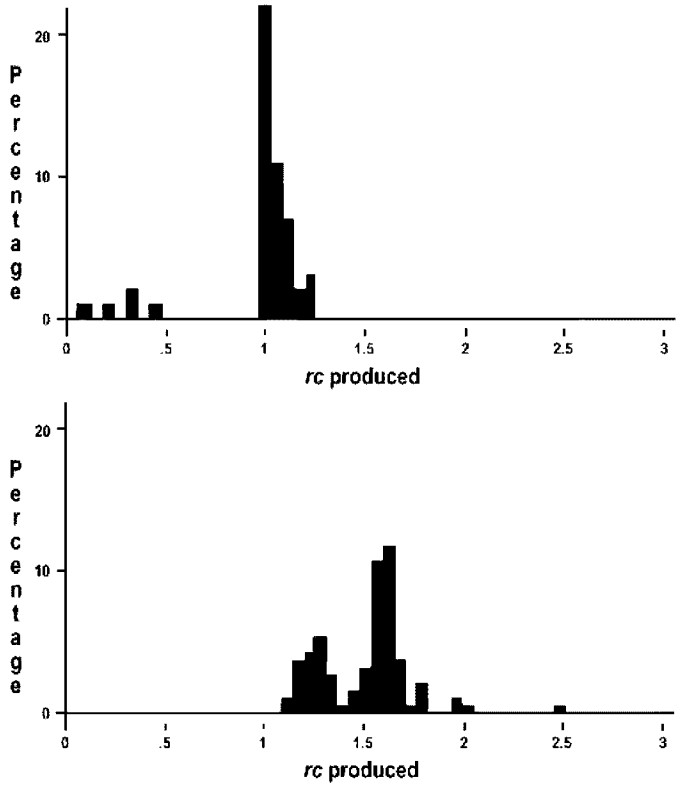

Fig. 13. Histogram of routing parameters produced (top) without case injection and (bottom) with case injection from offline play.

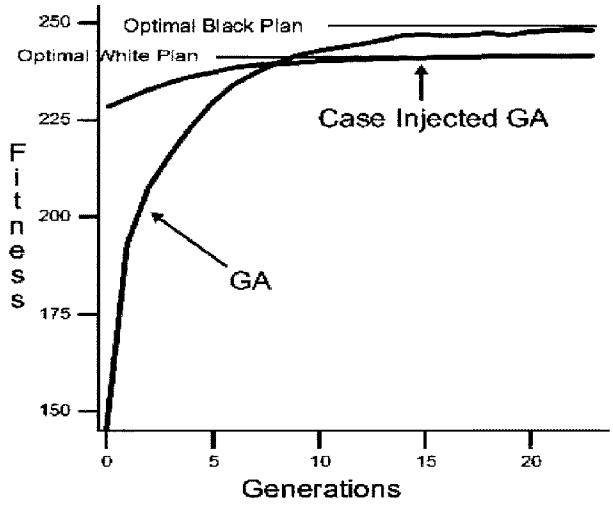

Fig. 14. Effect of case injection on fitness inside the GA over time.

matter to capture all human player decisions during the course of playing the game. We can then convert these decisions into our plan encoding and store them in the case-base for later injection. Using this methodology, we reverse engineer the human route (shown in black in Fig. 15) into our chromosome encoding. The closest encoding gives the route shown in white in Fig. 15. The plans are not identical because the chromosome does not contain exact routes—it contains the routing parameter rc. The overall fitness difference between these two plans is less then 2%.

The rc values determine the route category produced and GAP's ability to generate the human route depends on the values of rc found by the GA. Fig. 16 shows the distribution of rc

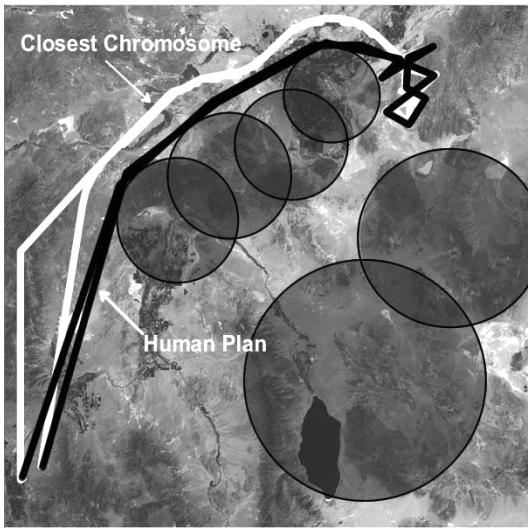

Fig. 15. Plans produced by the human and GAP.

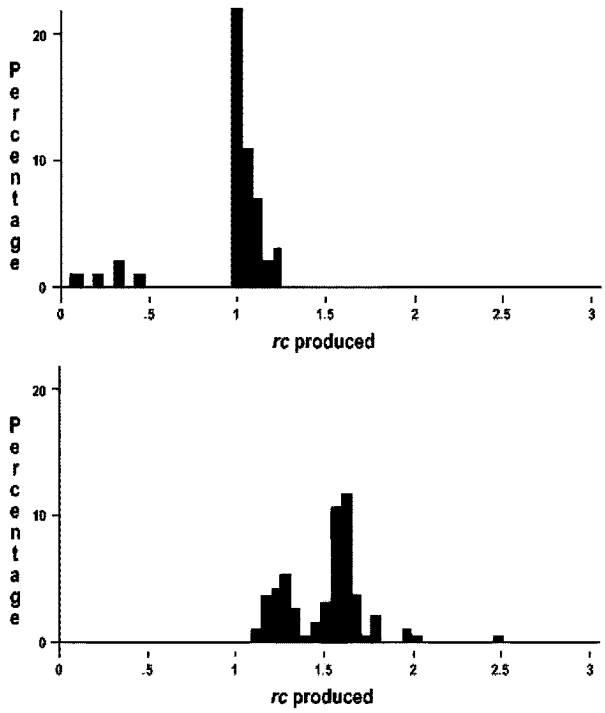

Fig. 16. Histogram of routing parameters produced (top) without injection and (bottom) with injection of human cases.

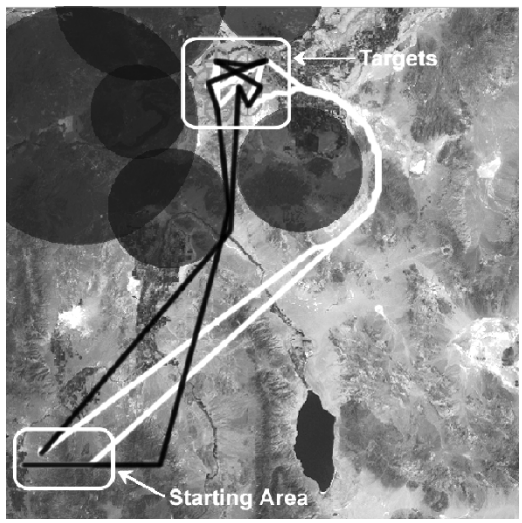

Fig. 17. Alternate mission.

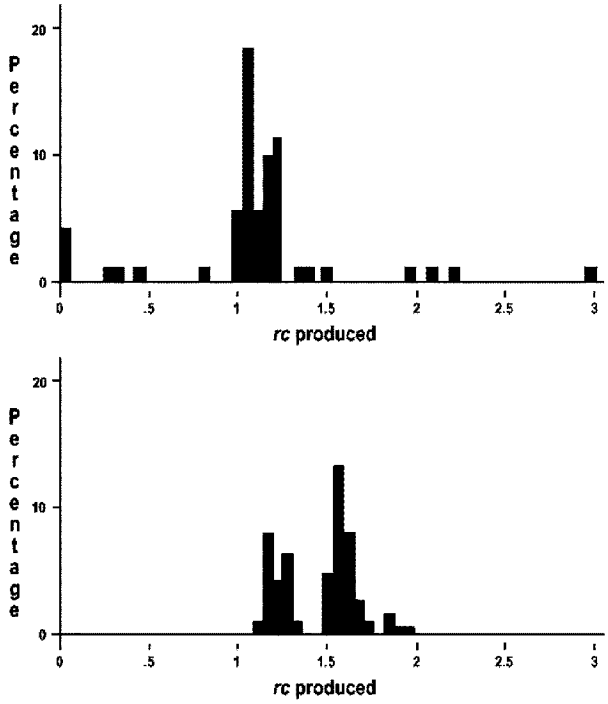

Fig. 18. Histogram of routing parameters on the alternate mission (top) without case injection and (bottom) with case injection.

produced by the noninjected GA and CIGAR. Comparing the figures shows a significant shift in the rc's produced. This shift corresponds to a large increase in the number of white routes generated by CIGAR. Without case injection, GAP produced no (0%) white trap-avoiding routes, but using case injection, 64% of the routes produced by GAP were white trap-avoiding routes. This difference is statistically significant and based on 50 different runs of the system with different random seeds. The figures indicate that case injection does bias the search toward the human strategy.

Moving to the mission shown in Fig. 17 and repeating the process produces the histograms shown in Fig. 18. The same effect on rc can be observed even though the missions are significantly different in location and in optimal allocation, and even though we use cases from the previous mission. Case injection and the general routing representation allows GAP to generalize and learn to avoid confined areas from play by the human expert.

F. Fitness Biasing

Case injection applies a bias to the GA search, while the number and frequency of individuals injected determines the strength of this bias. However, the fitness function also contains a term that biases *against* producing longer routes. Thus, we would expect that as the number of evaluations allotted to the GA increases, the bias against longer routes outweighs the bias toward white trap-avoiding longer routes and fewer white routes are produced. The effect is shown in Fig. 19. We use

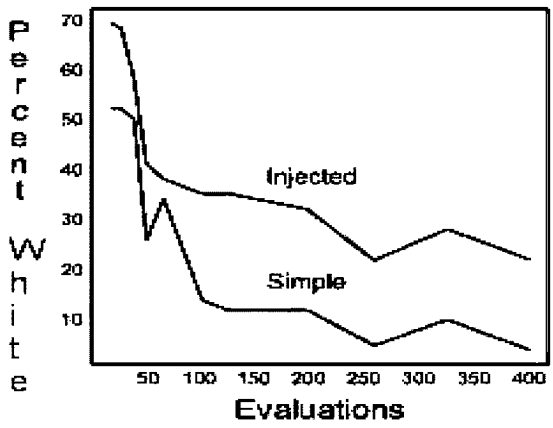

Fig. 19. Percentage of white trap-avoiding routes produced over time.

fitness biasing to change this behavior. Fitness biasing effectively changes the fitness landscape of the underlying problem by changing the fitness of an individual.

One possible approach to changing the fitness landscape is to change the fitness function. This would either involve rewriting code, or parameterizing the fitness function and using some algorithm to set parameters to produce desired behavior. Either way, changing the fitness function is equivalent to changing the strike force game and is domain dependent. However, we want to bias fitness in a domain-independent way without changing the game.

We propose a relatively domain independent way to use information from human derived cases to bias fitness. An individual's fitness is now the sum of two terms: 1) the fitness returned from evaluation and 2) a bonus term that is directly proportional to the number of injected bits in the individual. Let f_b be the biased fitness and let f_e be the fitness returned by evaluation. Then, the equation below computes the biased fitness.

$$f_b = f_e(1 + g(n_b, l))$$

where n_b is the number of injected bits in this individual, l is the chromosome length, and

$$g(n_b, l) = an_b + b \text{ if } n_b < l/c$$
$$= l/c \text{ otherwise}$$

where a, b, and c are constants. In our work, we used $a = 1, b = 0$, and $c = 5$ resulting the simple bias function below

$$g(n_b, l) = n_b \text{ if } n_b < l/5$$
$$= l/5 \text{ otherwise.}$$

Since the change in fitness depends on the genotype (a bit string) not on the domain dependent phenotype, we do not expect to have to significantly change this fitness biasing equation for other domains.

With fitness biasing, there is a significant increase in the number of white trap-avoiding routes produced, regardless of the number of evaluations permitted. Fig. 20 compares the number of white trap-avoiding routes produced by the GA, by CIGAR, and by CIGAR with fitness biasing. Clearly, fitness biasing increases the number of white routes.

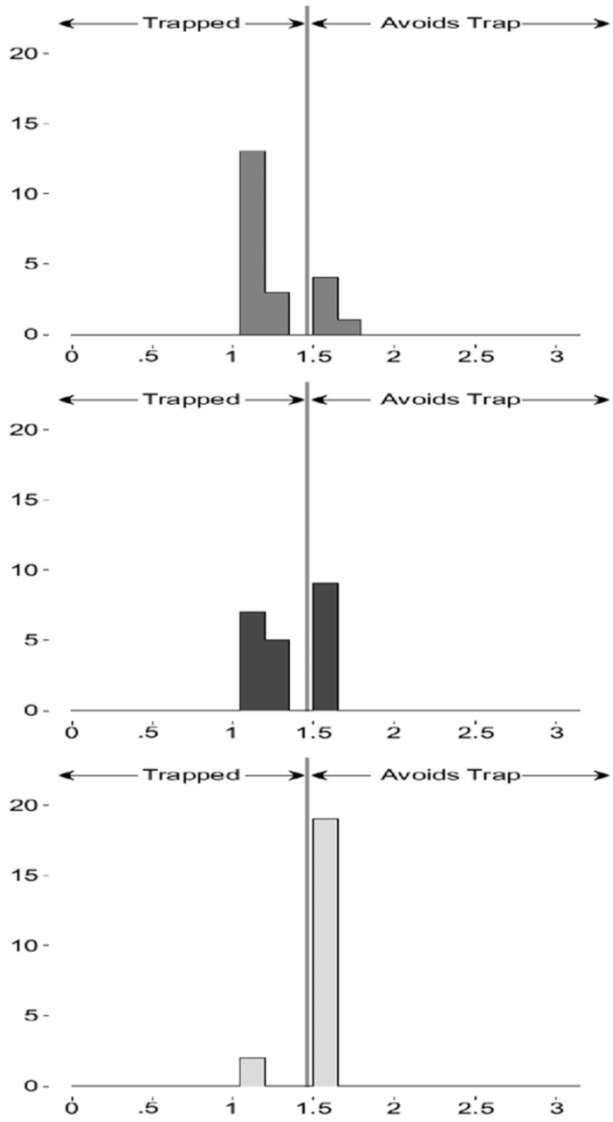

Fig. 20. (Top) Times trapped: (Middle) Without injection: With injection—(Bottom) No fitness biasing: With fitness biasing.

Fitness biasing's long-term behavior is depicted in Fig. 21. The figure shows that as the number of evaluations increases, the number of white routes produced with fitness biasing remains relatively constant and that this number decreases otherwise.

Summarizing, the results indicate that CIGAR can produce competent players for RTS games. GAP can learn through experience gained from human Blue players and from playing against Red opponents. Fitness biasing changes the fitness landscape in response to acquired knowledge and leads to better performance in learning to avoid traps. Finally, our novel route representation allows GAP to generalize acquired knowledge to other geographic locations and scenarios.

VII. Summary, Conclusion, and Future Work

In this paper, we developed and used a strike force planning RTS game to show that CIGAR can: 1) play the game; 2) learn from experience to play better; and 3) learn trap avoidance from a human player's game play. We cast our RTS game play as

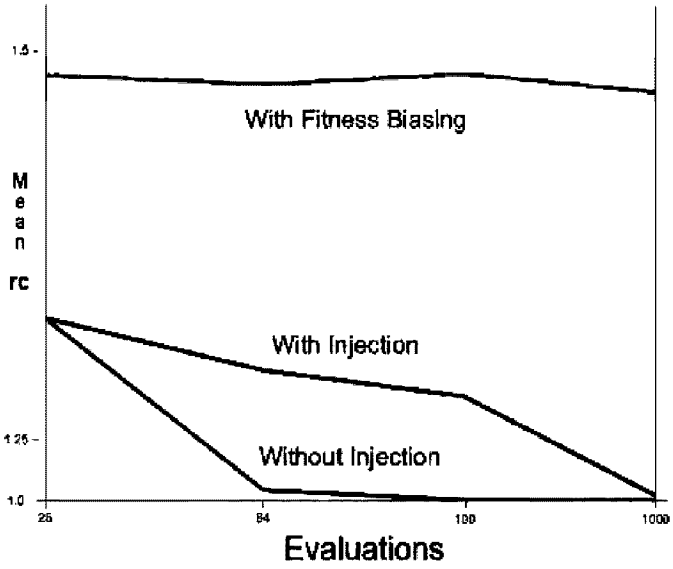

Fig. 21. Fitness biasing's effect over time.

the solving of resource allocation problems and showed that a parallel GA running on a ten-node cluster can efficiently solve the problems considered in this paper. Thus, the GA can play the strike force RTS game by solving the sequence of resource allocation problems that arise during game play.

Case injection allows GAP to learn from past experience and leads to better and quicker response to opponent game play. This past experience can come from previous game play or from expert human game play. To show that CIGARs can acquire and use knowledge from human game play, we first defined a structured scenario involving laying and avoiding traps as a testbed. We then showed how CIGARs use cases saved from human game play in learning to avoid confined areas (potential traps). Although the system has no concept of traps or how to avoid them, we showed that the system acquired and used trap-avoiding knowledge from automatically generated cases that represented human moves (decisions) during game play. Specifically, the system works by automatically recording human player moves during game play. Next, it automatically generates cases for storage into a case-base from these recorded moves. Finally, the system periodically injects relevant cases into the evolving population of the GA. Since humans recognize and avoid confined areas that have high potential for traps, cases generated from human play implicitly contain trap-avoiding knowledge. When injected, these cases bring trap-avoiding knowledge into the evolving GA population.

However, the evaluation function does not model the knowledge being acquired from human players: Trap-avoiding knowledge in our scenarios. GAP may, therefore, prematurely lose these low-fitness injected individuals. To ensure that GAP does not lose acquired knowledge, we proposed a new method, fitness biasing, for more effectively retaining and using acquired knowledge. Fitness biasing is a domain independent method for changing the fitness landscape by changing the value returned from the evaluation function by a factor that depends on the amount of acquired knowledge. This amount of acquired knowledge is measured (domain independently) by the number of bits that were inherited from injected cases in the individual being evaluated.

We parameterized the A* search algorithm in order to define a representation for routes that allows trap-avoidance knowledge to generalize to new game scenarios and locations. Specifically, this new representation allows using cases acquired during game play in one scenario (or map) to bias system play in other scenarios. Recent work in adding more parameters to the routing system has shown that GAP can effectively emulate many attack strategies, from pincer attacks to combined assaults.

We plan to build on these results to further develop the game. We would like to make the game more interesting, allow multiple players to play, and to develop the code for distribution. In the next phase of our research we will be developing a GAP for the Red side. Coevolving competence has a long history in evolutionary computing approaches to game playing, and we would like to explore this area for RTS games.

REFERENCES

[1] P. J. Angeline and J. B. Pollack, "Competitive environments evolve better solutions for complex tasks,," in *Proc. 5th Int. Conf. Genetic Algorithms*, 1993, pp. 264–270. [Online]. Available: citeseer.ist.psu.edu/angeline93competitive.html.

[2] D. B. Fogel, *Blondie24: Playing at the Edge of AI*. San Mateo, CA: Morgan Kauffman, 2001.

[3] A. L. Samuel, "Some studies in machine learning using the game of checkers," *IBM J. Res. Develop.*, vol. 3, pp. 210–229, 1959.

[4] J. B. Pollack, A. D. Blair, and M. Land, "Coevolution of a backgammon player," in *Proc. 5th Int. Workshop Synthesis Simulation of Living Syst. (Artificial Life V)*, C. G. Langton and K. Shimohara, Eds. Cambridge, MA, 1997, pp. 92–98.

[5] G. Tesauro, "Temporal difference learning and TD-gammon," *Commun. ACM*, vol. 38, no. 3, 1995.

[6] D. B. Fogel, T. J. Hays, S. L. Hahn, and J. Quon, "A self-learning evolutionary chess program," *Proc. IEEE*, vol. 92, no. 12, pp. 1947–1954, Dec. 2004.

[7] J. E. Laird, "Research in human-level RAI using computer games," *Commun. ACM*, vol. 45, no. 1, pp. 32–35, 2002.

[8] J. E. Laird and M. van Lent, *The Role of AI in Computer Game Genres*, 2000.

[9] S. J. Louis and J. McDonnell, "Learning with case injected genetic algorithms," *IEEE Trans. Evol. Comput.*, vol. 8, no. 4, pp. 316–328, Aug. 2004.

[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, U.K.: Cambridge Univ. Press, 1982.

[12] D. B. Leake, *Case-Based Reasoning: Experiences, Lessons, and Future Directions*. Menlo Park, CA: AAAI, 1996.

[13] S. J. Louis, G. McGraw, and R. Wyckoff, "Case-based reasoning assisted explanation of genetic algorithm results," *J. Exper. Theor. Artif. Intell.*, vol. 5, pp. 21–37, 1993.

[14] D. E. Goldberg, *Genetic Algorithms in Search, Optimization, and Machine Learning*. Reading, MA: Addison-Wesley, 1989.

[15] S. J. Louis, "Evolutionary learning from experience," *J. Eng. Opt.*, vol. 26, no. 2, pp. 237–247, 2004.

[16] ——, "Genetic learning for combinational logic design," *J. Soft Comput.*, vol. 9, no. 1, pp. 38–43, 2004.

[17] ——, "Learning from experience: Case injected genetic algorithm design of combinational logic circuits," in *Proc. 5th Int. Conf. Adapt. Comput. Design Manuf.*, 2002, pp. 295–306.

[18] 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. 283–290.

[19] B. J. Griggs, G. S. Parnell, and L. J. Lemkuhl, "An air mission planning algorithm using decision analysis and mixed integer programming," *Oper. Res.*, vol. 45, no. 5, pp. 662–676, Sept.–Oct. 1997.

[20] V. C.-W. Li, G. L. Curry, and E. A. Boyd, "Strike force allocation with defender suppression," Ind. Eng. Dept.t, Texas A&M Univ., College Station, TX, Tech. Rep., 1997.
[21] K. A. Yost, "A survey and description of usaf conventional munitions allocation models," Office of Aerospace Studies, Kirtland AFB, Albuquerque, NM, Tech. Rep., Feb. 1995.
[22] S. J. Louis, J. McDonnell, and N. Gizzi, "Dynamic strike force asset allocation using genetic algorithms and case-based reasoning," in *Proc. 6th Conf. Syst., Cybern. Inf.*, Orlando, FL, 2002, pp. 855–861.
[23] C. D. Rosin and R. K. Belew, "Methods for competitive co-evolution: Finding opponents worth beating," in *Proc. 6th Int. Conf. Genetic Algorithms*, L. Eshelman, Ed., 1995, pp. 373–380.
[24] G. Kendall and M. Willdig. An investigation of an adaptive poker player. presented at Proc Australian Joint Conf. Artif. Intell.
[25] Blizzard, Starcraft. (1998). [Online]. Available: www.blizzard.com/starcraft. www.blizzard.com/starcraft
[26] Cavedog, Total Annihilation. [Online]. Available: www.cavedog com/totala. www.cavedog.com/totala
[27] R. E. Inc, Homeworld. (1999). [Online]. Available: homeworld. sierra.com/hw
[28] J. E. Laird and M. van Lent, "Human-level AI's killer application: Interactive computer games," Invited Talk at the AAAI-2000 Conf., [Online]. Available: http://ai.eecs.umich.edu/people/laird/papers/AAAI-00.pdf, 2000.
[29] G. Tidhar, C. Heinze, and M. C. Selvestrel. (1998) Flying together: Modeling air mission teams. *Appl. Intell.* [Online], vol (3), pp. 195–218
[30] D. McIlroy and C. Heinze, "Air combat tactics implementation in the smart whole air mission model," in *Proc. 1st Int. SimTecT Conf.*, Melbourne, Australia, 1996, [Online]. Available: citeseer. nj.nec.com/mcilroy96air.html.
[31] B. Stout, "The basics of A* for path planning," *Game Programming Gems*, pp. 254–262, 2000.
[32] 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 Mateo, CA: Morgan Kaufmann, 1991, pp. 265–283.

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 and Engineering, University of Nevada, Reno.

Dr. Louis and is a member of the Association for Computing Machinery (ACM). He is an Associate Editor of the IEEE TRANSACTIONS ON EVOLUTIONARY COMPUTATION, and he is also Co-General Chairman of the 2006 IEEE Symposium on Computational Intelligence in Games to be held in Reno, NV, May 22-24, 2006.

Chris Miles is currently working towards the Ph.D. degree in the Evolutionary Computing Systems Laboratory, University of Nevada, Reno.

He is working on using evolutionary computing techniques for real-time strategy games.