

Combining Case-Based Memory with Genetic Algorithm Search for Competent Game AI

Sushil J Louis and Chris Miles

Evolutionary Computing Systems Laboratory
Department of Computer Science and Engineering
University of Nevada, Reno 89557
<http://ecsl.cse.unr.edu>
(sushil,miles@cse.unr.edu)

Abstract. We use case-injected genetic algorithms for learning how to competently play computer strategy games. Case-injected genetic algorithms combine genetic algorithm search with a case-based memory of past problem solving attempts to improve performance on subsequent similar problems. The case-injected genetic algorithm improves performance on later problems in the sequence by learning from cases recorded earlier in the sequence. Since game-play in strategy games usually boils down to optimally allocating resources to achieve in-game mission objectives, we describe how a case-injected genetic algorithm player can play our game by solving the sequence of resource allocation problems generated by opponent moves during game-play. When retrieving and using cases recorded from human game-play, results show that case injection effectively biases the genetic algorithm toward producing plans that contain appropriate elements of plans produced by human players.

1 Introduction

This paper reports on ongoing work at the Evolutionary Computing Systems Lab (ECSL) in using Case-Injected Genetic Algorithms (CIGARs) to learn how to competently play computer strategy games. Case-injected genetic algorithms combine genetic algorithm search with a case-based memory to improve performance on similar problems. In this approach, rather than starting anew with a randomly initialized search population on each problem, we inject a genetic algorithm's evolving population with appropriate intermediate solutions (cases) to similar, previously solved problems. The case-base does what it is best at - memory organization; the genetic algorithm handles what it is best at - adaptation. The resulting combination takes advantage of both paradigms; the genetic algorithm component delivers robustness and adaptive learning while the case-based component speeds up the learning process.

We have developed a 3D strategy game as a research platform for investigating our non-traditional approach to computer game playing. Casting game-play in our strategy game as a resource allocation optimization problem makes it suitable for a genetic algorithm-based approach. However, to play well, a genetic

algorithm player must learn to anticipate opponent moves from game playing experience. This anticipatory knowledge can come from cases recorded during past CIGAR game playing experience or from cases extracted during human game-play. This paper describes the design of a case-injected Genetic Algorithm Player (GAP) that uses CIGAR to play strategy games and attacks the problem of automatically acquiring game playing knowledge from human players. Results show that GAP can play competently and that injecting cases into the evolving population of a genetic algorithm effectively biases GAP toward producing plans that contain important strategic elements used by human players.

We introduce case-injected genetic algorithms in the next section. We then describe our real-time strategy game. Section 6 explains CIGAR’s representation of allocations and routing. The last two sections provide results and conclusions.

2 Case-Injected Genetic Algorithms

Genetic algorithms (GAs) are randomized population-based search algorithms that search from a population of points [1, 2]. GAs use feedback about the utility (fitness) of points in the search space to progress towards a goal (the optimum). Current genetic algorithm-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, 7].¹ CIGAR combines genetic algorithm search with case-based memory to improve performance with experience. Previous results on a variety of problems show 1) that CIGAR, with experience, takes less time to solve new problems and produces better quality solutions and 2) that simple syntactic similarity metrics for case indexing lead to this performance improvement [8].

Search as a problem solving technique has a long history in AI research. Casting game-play as a resource allocation problem means that we can define a search space of possible allocations of assets to targets. Each point in the search space represents a possible allocation. In the absence of admissible heuristics, if we can evaluate the fitness of points in this search space, we can devise search algorithms that use this fitness measure to guide heuristic search. Genetic algorithms were designed to search poorly-understood spaces such as those that arise in combinatorial optimization problems like the resource allocation problem. GAs are the search method of choice in this paper.

Typically, a genetic algorithm randomly initializes its starting population of individuals (candidate solutions to the problem) so that the GA can proceed from an unbiased sample of the search space. However, 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 and many of these problems may be similar. In this case it makes little sense to start a problem solving search attempt from scratch (random initialization) when previous searches

¹ See also the web page at <http://ai-cbr.cs.auckland.ac.nz> for a list of case-based design projects.

may have yielded useful information about the problem domain. Instead, injecting a genetic algorithm’s population with *relevant* solutions 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 new problems [9–11]. In our system, the case-base of problems and their solutions supplies the genetic problem solver with a long term memory.

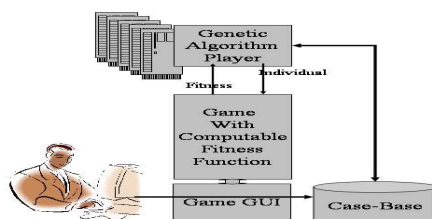


Fig. 1. The CIGAR game playing system

Figure 1 depicts the genetic algorithm and its interaction with the case-base and game. As the genetic algorithm searches through the space of possible allocations and game, it periodically saves high fitness individuals to the case-base. Cases in CIGAR are individuals from a genetic algorithm’s population along with some ancillary information like a unique case identifier, the individual’s fitness, pointers to the individual’s parents, and a problem identifier. These individuals, or cases, encode candidate solutions or partial solutions to the game’s underlying resource allocation problem. The case-base also obtains cases from human game-play - we reverse-engineer human player generated allocations (game-play decisions) into the encoding used by the genetic algorithm and store these cases in the case-base. On subsequent missions in the game, the system injects appropriate cases into the genetic algorithm player’s evolving population thus biasing search towards solutions similar to injected cases. If injected cases come from human players, the knowledge encapsulated in these cases biases GAP. Alternatively, if cases come from previous GAP attempts at playing the game, then the system learns from its own previous game playing experience. The system does not require a case-base to start with and can bootstrap itself by learning new cases from the genetic algorithm’s attempts at solving a problem.

Using CBR terminology, the genetic algorithm adapts retrieved cases to generate solutions to the current in-game mission’s resource allocation problem. We do not need domain specific adaptation operators because domain knowledge encapsulated within the fitness function and encoding suffice to guide genetic algorithm-based domain-independent adaptation of injected cases. The GA takes care of adaptation - indexing for retrieval, however, remains an issue.

Assuming that similar problems have similar solutions, the typical CBR approach would inject cases representing solutions to **similar problems**. When we have problem similarity metrics, this approach works quite well and is described in [12]. However, genetic algorithms are supposed to work in poorly-understood spaces and we may thus find it difficult, if not impossible, to come up with a problem similarity metric in such spaces. Thus, instead of assuming that similar problems have similar solutions, we assume that similar solutions must have come from similar problems. If this assumption is true, because genetic algorithms use simple string representations of solutions, equally simple similarity metrics like hamming distance and euclidean distance should work well for indexing cases. Previous work provides empirical evidence in support of the assumption and approach [8, 13]. This paper uses hamming distance between bit-strings that encode game-playing strategies for its distance metric.²

Figure 2 depicts a case-injected genetic algorithm that uses solution similarity metrics for case-retrieval and injection. Initially the case-base is empty.

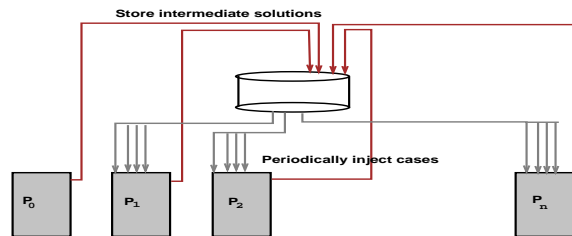


Fig. 2. A case-injected genetic algorithm working on the basis of solution similarity

When confronted with the first problem, P_0 , the system starts with a randomly initialized GA but generates cases to be stored in the case-base. Next, when CIGAR attempts P_1 , it 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.

When CIGAR operates on the basis of solution similarity, we **periodically** retrieve and inject a small number of *solutions* similar to the current best member of the GA population into the *current* population. Injected individuals replace the worst members in the current population and the GA continues searching with this combined population. The idea is to cycle through the following steps. Let the GA make some progress. Next, find cases (individuals) in the case-base that are similar to the current best individual in the population and inject these individuals into the population replacing the worst individuals. If injected

² The hamming distance between two strings is the number of bits that are different.

individuals contain useful cross problem information, the GA's performance will be boosted.

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 retrieved from the case-base contain useful information to guide genetic search.

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 towards a solution. CIGAR is thus robust.

Note also that every individual in the population is a potential case. For a population of size 50, run for 50 generations, we could end up with 2500 potential cases on every new problem attempted. We reduce the number of cases stored in the case-base by only storing individuals that represent an increase in the maximum fitness seen thus far in the evolving population.

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 proved effective.

3 The Game

We have designed and built OPS, an air strike attack planning and defense game. OPS maps to a broad category of resource allocation problems in industry and the military. The game's objective is to plan an air attack against defended targets (Blue player) or to defend against such an air attack (Red player). Blue's goal is thus allocating weapons on the air platforms to targets on the ground and Red's goal is to allocate threats to defend targets. Target priorities and the varying effectiveness of Blue's weapon types against target types further complicate planning. Potential new threats and targets can also "pop-up" on Red's command in the middle of a mission, requiring Blue to be able to respond to changing game dynamics. Both players seek to minimize the damage they receive while maximizing the damage dealt to their opponent. Red plays by organizing

defenses in order to best protect its targets and Red’s ability to play popups can affect this planning. For example, feigning vulnerability can lure Blue into a pop-up trap, or keep Blue from exploiting a weakness out of fear of such a trap. Blue plays by allocating platforms and their assets (weapons) as efficiently as possible in order to destroy targets while minimizing risk. Many factors determine risk: the platform’s route, the effect of accompanying wingmen, weather, and the presence of threats around chosen targets. Using a case-injected genetic algorithm, GAP develops strategies for the attacking strike force, including flight plans and weapon targeting for all available aircraft. When confronted with pop-ups, GAP responds by re-planning with the case injected genetic algorithm to produce a new plan of action.

The left side of Figure 3 shows a screen shot of our game while the right side shows a top-level view of a game mission. We have implemented this game

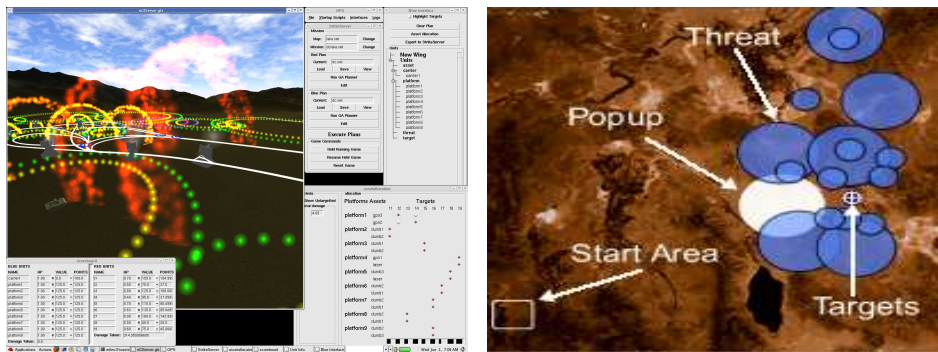


Fig. 3. Left: Game screen-shot. Right: Example mission

on top of an open-source game engine making it more interesting than a pure optimization problem. An attractive, “fun,” game would entice more people to play and allow us to acquire player knowledge during game-play. After all, beyond producing near optimal strategies, we would like to bias CIGAR toward producing solutions similar to those used by humans playing Blue in the past. We do this so that GAP can be a better and more versatile player by learning strategies from human experts. Specifically, we want to show that GAP more frequently produces plans similar to those a human has played in the past on the same mission. When that information from the human is used by GAP playing a different mission, we show that GAP continues to play strategies closer to the style of the human than GAP’s non injected counterpart.

3.1 Related Work

A large body of work exists in which evolutionary methods have been applied to games [14–18]. However the majority of this work has been applied to board,

card, and other well defined games which have many differences from popular real time strategy (RTS) computer games such as Starcraft, Total Annihilation, and Homeworld [19–21]. Entities in our game exist and interact over time in continuous three-dimensional space, they are not directly controlled by players but instead sets of algorithms (Game AIs) control them in order to meet strategic goals outlined by players. This adds a level of abstraction not found in traditional games. In addition, computer game players usually have incomplete knowledge of the game state. John Laird [22–24] surveys the state of research in using Artificial Intelligence (AI) techniques in interactive computers games. He describes the importance of such research and provides a taxonomy of games.

In simulation games like SimCity, the decision maker controls many non-linearly related variables (inhabitants and environment) and makes decisions that affect the health of her world (in this case a city). Fasciano’s work using a case-based approach to build the mayor for the original SimCity game stands out as an example of using computer gaming to study decision support and planning issues [25–27].

Several military simulations share some of our game’s properties [28–30] however, not only do we want to provide a platform for research in strategic planning - we also want to have fun.

4 System Architecture and Encoding

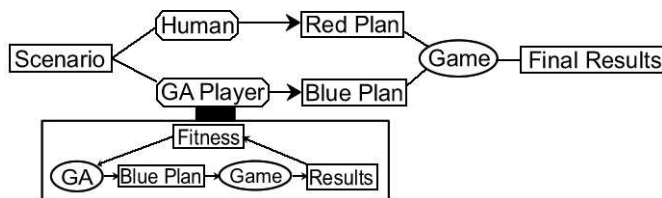


Fig. 4. System architecture.

Figure 4 outlines our system’s architecture. Starting at the left, the Red team (human) and the Blue team (GAP) get the scenario and have time to prepare their strategy. GAP works by applying the genetic algorithm to the underlying resource allocation and routing problem. We chose the best plan produced by the GA in the time available, to play against Red. These plans then execute and during execution, Red can script the emergence of a popup threat. When GAP detects the popup, the genetic algorithm re-plans and begins execution of the new plan.

To play the game GAP must produce routing data for each of Blue’s platforms. Figure 5 shows how routes are built using the A* algorithm [31]. A* builds routes from current platform locations to target locations and back and tends to prefer short routes that avoid threats while seeking targets.

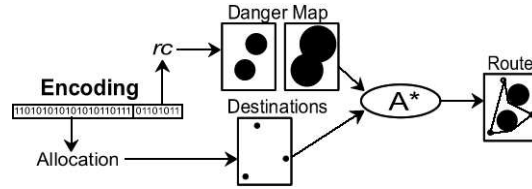


Fig. 5. How routes are built from an encoding.

In order to avoid traps the routing system must be somehow parameterized to avoid areas with particular characteristics. Note that traps are most effective in areas confined by other threats. If we artificially inflate threat radii, threats will expand to fill in potential trap corridors and A* will find routes that go around these expanded threats. We therefore add a multiplier parameter rc that modifies threats' effective radii. Larger rc 's expand threats and fill in confined areas. A* then routes around those confined areas. Combined with case injection, rc allows GAP to learn coefficients that avoid traps and re-use them in new scenarios. In our scenario $rc < 1.0$ produce shorter, more direct routes, that might go through a threat's area of influence, $1.0 < rc < 1.35$ produce direct routes that avoid threats, and $rc > 1.35$ produce roundabout routes. rc is limited to the range $[0, 3]$ and encoded with eight (8) bits at the end of our chromosome.

4.1 Encoding

Most of the encoding specifies the asset to target allocation with rc encoded at the end as detailed above. Figure 6 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 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.

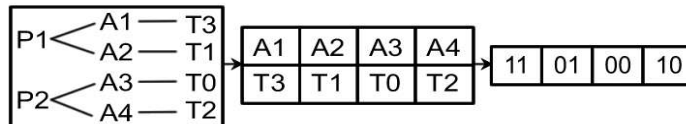


Fig. 6. Allocation encoding

4.2 Fitness

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 shown in Equation 1

$$fit(plan) = TotalDamage(Red) - TotalDamage(Blue) - d * c \quad (1)$$

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 ($fit(plan)$). Total damage done is computed below.

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

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 the formulas above for computing the fitness of a particular individual in the genetic algorithm's population. Evaluating an individual thus means that we have to run the game (simulation). Although our game can run much faster than real-time, evaluating a population of, say 50, individuals takes far too long for interactive game-play. However, parallelizing CIGAR by simultaneously evaluating multiple individuals on our cluster significantly speeds up evaluations. At the current time, we can respond competently to a human opponent's moves in less than one minute.

4.3 Sample Mission

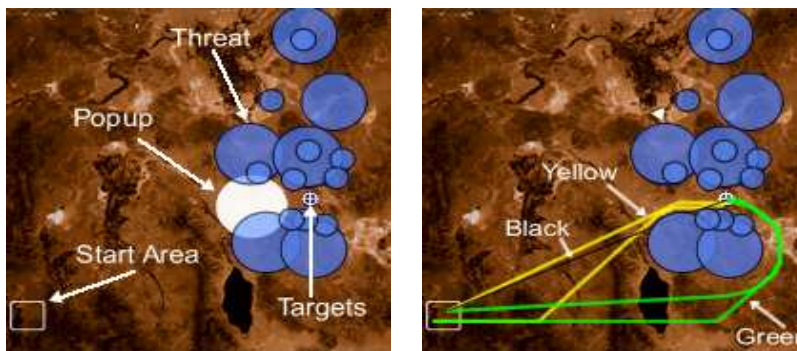


Fig. 7. Left: The scenario Right: Route categories

Figure 7-Left shows an overview of our test scenario - chosen to be simple, easily analyzable but to still encapsulate the dynamics of traps and anticipation.

The scenario takes place in Northern Nevada and California with Walker Lake visible below (south) of the popup on the bottom of the map. Red has four targets on the right-hand side of the map with their locations denoted by the white cross-hair. As the targets represent different buildings comprising a larger facility, they appear as a single cross-hair from our point of view which is at a significant distance. Red has twenty two (22) threats placed to defend the targets and the translucent blue hemispheres show the effective radii of these threats. Red has the potential to play a popup threat to trap platforms venturing into the corridor formed by the threats and this trap is displayed as the solid white circle near the middle.

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. A weapon-target effectiveness table determines the effectiveness of each weapon against each target. Each of the eight weapons can be allocated to any of the four targets, giving $4^8 = 2^{16} = 64k$ allocations. This space is exhaustively searchable, but more complex scenarios quickly become intractable. The second stage in Blue’s planning involves finding routes for each of the platforms to follow during their mission. These routes should be short and simple but still minimize exposure to risk. We categorize Blue’s possible routes into two categories: yellow routes fly through the corridor between the threats, while green routes fly around. The evaluator has no direct knowledge of potential danger presented to platforms inside the corridor area. Because of this, the evaluator optimal solution is the yellow route, since it is the shortest. The human expert however, understands the potential for danger as the corridor provides the greatest potential for a pop-up trap. Knowing this, the green route is the human optimal solution. Our goal is now to bias the GA to produce the green route, while still optimizing the allocation. Teaching GAP to learn from a human and produce green strategies even though yellow strategies have higher fitness is a goal of this research.

5 Results

We first present results showing that GAP can play the game effectively. GAP runs against our test scenario 50 times, and we plot the minimum, maximum, and average population fitness over the number of genetic algorithm iterations in Figure 8-Left. The graph shows a strong approach toward the optimum and in more the 95% of runs it gets within 5% of the optimum. This indicates that GAP can form effective strategies for playing the game.

To deal with opponent moves and the dynamic nature of the game we look at the effects of re-planning. Figure 8-Middle illustrates the effect of re-planning by showing the final route followed inside a game. A yellow route was chosen, and when the popup occurred, trapping the platforms, GAP redirected the strike force to retreat and attack from the rear. Re-planning allows GAP to rebuild its routing information as well as modify its allocation to compensate for damaged platforms. Figure 8-Right compares fitnesses with and without case injection for

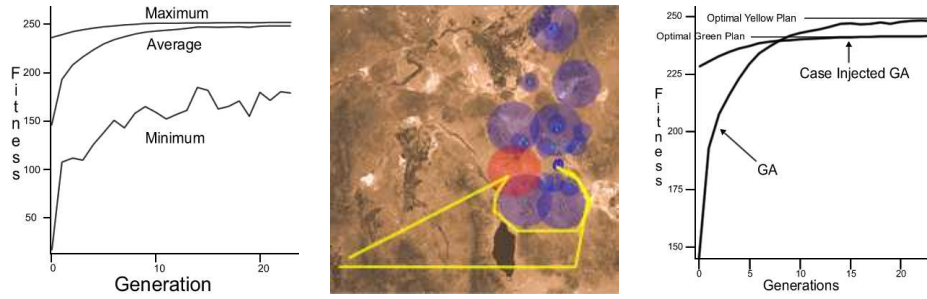


Fig. 8. Left: Best, worst, and average individual fitness as a function of generation - averaged over 50 runs. Middle: Final routes used during a mission involving re-planning. Right: Effect of case injection on fitness over time.

learning to avoid confined areas. Injecting cases that use green routes acquired from human players causes GAP to learn to start preferring green routes. We can see that GAP starts out with higher quality solutions and quickly converges on the trap-avoiding (green) plan. However, we find that running CIGAR longer and with larger population sizes often causes convergence to yellow routes.

How do we obtain green routes that avoid traps despite CIGAR’s eventual preference for shorter more direct routes? CIGAR’s preference for shorter routes is driven by our evaluation function - knowledge acquired from human players must change this preference. We thus artificially inflate the fitness of individuals descended from injected cases to simulate the change in the evaluation function. With fitness inflation we get strong CIGAR convergence to trap-avoiding green routes. Figure 9 shows the value of the routing parameter, rc , on the x-axis and the number of runs of the GA or CIGAR that converged to a particular value of rc on the y-axis. The left side of the figure shows the distribution of the routing

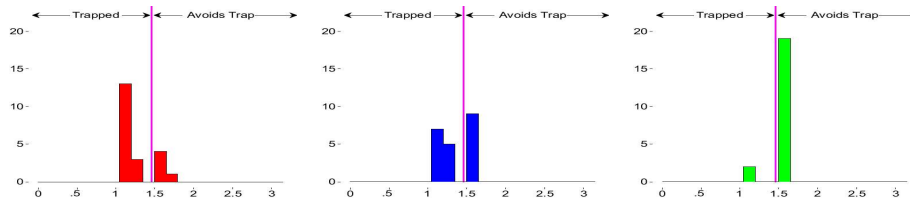


Fig. 9. Distribution of RC values. Left: GA alone. Middle: With case injection. Right: Case injection and fitness biasing.

parameter values without case-injection. More routes lead through the suspicious (to humans) corridor that is ripe for Red’s trap. With case-injection (middle), there are more routes that avoid the potential trap although the difference is not significant. The rightmost graph shows the distribution of RC values with

both case injection and increasing the fitness of injected individuals and their descendants (proportional to the amount of injected material). In this case, there is clearly a strong preference for routes avoiding traps.

6 Conclusions and Future Work

This paper considered two broad issues in developing competent opponents for computer gaming - using a genetic algorithm for playing strategic computer games and knowledge acquisition from expert players to bias genetic search. We showed that our genetic algorithm player can play the game competently. We also showed that GAP learns to prefer solutions similar to injected cases acquired from humans and to deal with the external-to-the-evaluator concept of a trap. Our preliminary results thus indicate that case injection seems to be a promising approach toward acquiring knowledge and biasing genetic search toward human preferred solutions. We are currently using these techniques in developing a training simulation (computer game) for strike planning.

Acknowledgment

This material is based upon work supported by the Office of Naval Research under contract number N00014-03-1-0104.

References

1. Holland, J.: *Adaptation In Natural and Artificial Systems*. The University of Michigan Press, Ann Arbour (1975)
2. Goldberg, D.E.: *Genetic Algorithms in Search, Optimization, and Machine Learning*. Addison-Wesley (1989)
3. Smith, D.: Bin packing with adaptive search. In: *Proceedings of an International Conference on Genetic Algorithms*, Morgan Kaufman (1985) 202–206
4. Janikow, C.Z.: A knowledge-intensive genetic algorithm for supervised learning. *Machine Learning* **13** (1993) 189–228
5. Grefenstette, J., Ramsey, C., Shultz, A.: Learning sequential decision rules using simulation models and competition. *Machine Learning* **5** (1990) 355–381
6. Huhns, M., Acosta, R.: Argo: An analogical reasoning system for solving design problems. In Tong, C., Sriram, D., eds.: *Artificial Intelligence in Engineering Design*, Vol II. Academic Press, Inc. (1992) 105–144
7. Sycara, K., Navinchandra, D.: Retrieval strategies in case-based design system. In Tong, C., Sriram, D., eds.: *Artificial Intelligence in Engineering Design*, Vol II. Academic Press, Inc. (1992) 145–164
8. Louis, S.J., McDonnell, J.: Learning with case injected genetic algorithms. *IEEE Transactions on Evolutionary Computation* **8** (2004) 316 – 328
9. Riesbeck, C.K., Schank, R.C.: *Inside Case-Based Reasoning*. Lawrence Erlbaum Associates, Cambridge, MA (1989)
10. Schank, R.C.: *Dynamic Memory: A Theory of Reminding and Learning in Computers and People*. Cambridge University Press, Cambridge, MA (1982)

11. Leake, D.B.: Case-Based Reasoning: Experiences, Lessons, and Future Directions. AAAI/MIT Press, Menlo Park, CA (1996)
12. Johnson, J., Louis, S.J.: Case-initialized genetic algorithms for knowledge extraction and incorporation. In: Knowledge Incorporation in Evolutionary Computation. (2004) 57–80
13. Miles, C., Louis, S.J., Cole, N., McDonnell, J.: Learning to play like a human: Case injected genetic algorithms for strategic computer gaming. In: Proceedings of the International Congress on Evolutionary Computation, Portland, Oregon, IEEE Press (2004) 1441–1448
14. Fogel, D.B.: *Blondie24: Playing at the Edge of AI*. Morgan Kaufman (2001)
15. Rosin, C.D., Belew, R.K.: Methods for competitive co-evolution: Finding opponents worth beating. In Eshelman, L., ed.: Proceedings of the Sixth International Conference on Genetic Algorithms, San Francisco, CA, Morgan Kaufmann (1995) 373–380
16. Pollack, J.B., Blair, A.D., Land, M.: Coevolution of a backgammon player. In Langton, C.G., Shimohara, K., eds.: *Artificial Life V: Proc. of the Fifth Int. Workshop on the Synthesis and Simulation of Living Systems*, Cambridge, MA, The MIT Press (1997) 92–98
17. Kendall, G., Willdig, M.: An investigation of an adaptive poker player. In: Australian Joint Conference on Artificial Intelligence. (2001) 189–200
18. Samuel, A.L.: Some studies in machine learning using the game of checkers. *IBM Journal of Research and Development* **3** (1959) 210–229
19. Blizzard: *Starcraft* (1998, www.blizzard.com/starcraft)
20. Cavedog: *Total annihilation* (1997, www.cavedog.com/totala)
21. Inc., R.E.: *Homeworld* (1999, homeworld.sierra.com/hw)
22. Laird, J.E.: Research in human-level ai using computer games. *Communications of the ACM* **45** (2002) 32–35
23. Laird, J.E., van Lent, M.: The role of ai in computer game genres (2000)
24. Laird, J.E., van Lent, M.: Human-level ai's killer application: Interactive computer games (2000)
25. Fasciano, M.J.: Real-time case-based reasoning in a complex world. Technical Report TR-96-05, Department of Computer Science, The University of Chicago (1996)
26. Hammond, K.J., Fasciano, M.J., Fu, D.D., Converse, T.: Actualized intelligence: Case-based agency in practice. Technical Report TR-96-06, Department of Computer Science, The University of Chicago (1996)
27. Fasciano, M.J.: Everyday-world plan use. Technical Report TR-96-07, Department of Computer Science, The University of Chicago (1996)
28. Tidhar, G., Heinze, C., Selvestrel, M.C.: Flying together: Modelling air mission teams. *Applied Intelligence* **8** (1998) 195–218
29. Serena, G.M.: The challenge of whole air mission modeling (1995)
30. McIlroy, D., Heinze, C.: Air combat tactics implementation in the smart whole air mission model. In: Proceedings of the First International SimTecT Conference, Melbourne, Australia, 1996. (1996)
31. Stout, B.: The basics of A* for path planning. In: *Game Programming Gems*, Charles River media (2000) 254–262