Trap Avoidance in Strategic Computer Game Playing with Case Injected Genetic Algorithms

Chris Miles, Sushil J. Louis, and Rich Drewes

Evolutionary Computing Systems Lab
Department of Computer Science
University of Nevada
Reno - 89557
{miles,sushil,drewes}@cs.unr.edu

Abstract. We use case injected genetic algorithms to learn to competently play computer strategy games. Such games are characterized by player decision in anticipation of opponent moves and imperfect knowledge of game state. Within the broad goal of developing effective and general methods of anticipatory play, this paper investigates anticipation in the context of trap avoidance in an immersive, 3D strike planning game. Case injection allows acquiring player knowledge from experience and incorporating acquired knowledge into future game play. Results show that with an appropriate representation case injection is effective at biasing the genetic algorithm toward producing plans that both avoid traps and carry out the mission effectively.

1 Introduction

The computer gaming industry is now bigger than the movie industry and both gaming and entertainment drive research in graphics and modeling. Although AI research has in the past been interested in games like checkers and chess, popular computer games like Starcraft and counter-strike are very different from chess and checkers. 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 business, training, planning, and scientific problems as games where player decisions determine the final solution. A decision support system for a player in such games corresponds closely with a decision support system in the "real" world.

This paper applies a case-injected genetic algorithm that combines genetic algorithms with case-based reasoning to provide player decision support in the context of domains modeled by computer games [1]. The genetic algorithm "plays" the game by attempting to solve the underlying decision problem. Specifically, we develop and use a strike force asset allocation game, which maps to a broad category of resource allocation problems in industry, as our test problem. Strike force planning consists of allocating a collection of strike assets on flying platforms to a set of targets and threats on the ground. The problem is dynamic; weather and other environmental factors affect asset performance, unknown

K. Deb et al. (Eds.): GECCO 2004, LNCS 3102, pp. 1365–1376, 2004.

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 genetic and evolutionary computing approaches.

The idea behind a case-injected genetic algorithm is that as the genetic algorithm component iterates over a problem it selects members of its population and caches them (in memory) for future storage into a case base. Cases are therefore members of the genetic algorithm's population and represent an encoded candidate solution to the problem at hand. Periodically, the system injects appropriate cases from the case base, containing cases from previous attempts at *other* problems, into the evolving population replacing low fitness population members. When done with the current problem, the system stores the cached population members into the case base for retrieval and use on new problems.

Case injection is used to handle the dynamic nature of the game which places a premium on re-planning or re-allocation of assets when needed. We have shown that case-injected genetic algorithms learn to increase performance with experience at solving similar problems [1,2,3,4,5]. This implies that a case-injected genetic algorithm should quickly produce new plans (a new allocation) in response to changing game dynamics. Beyond purely responding to immediate scenario changes we use case injection in order to produce plans that anticipate opponent moves in the future. Doing this teaches our Genetic Algorithm Player (GAP) where traps are likely to occur, so that GAP acts in anticipation of changing game states. Specifically we try to influence GAP to produce plans that avoid areas similar to those in which it has encountered traps in the past. Our results show that GAP makes an effective Blue player with the ability to quickly replan to deal with changing game dynamics, and that case-injection can bias GAP to produce solutions that are suboptimal with respect to the game simulation's evaluation function but that avoid potential traps.

In the rest of the paper we define the game, the particular scenario being played, and the trap being encountered. We outline GAP's architecture; detailing the use of genetic algorithms, the encoding of strategy, the routing system, and the incorporation of case injection. Section 7 presents results showing that GAP can effectively play the game in the absence of the trap, and that GAP can quickly re-plan in the face of changing game dynamics. Preliminary results also show the effectiveness of case injection in acquiring and using player knowledge in learning to avoid traps. The last section presents our conclusions and directions for future work.

2 The Strike Planning Game

The strike planning game is based on an underlying resource allocation and routing problem and the genetic algorithm plays by solving the underlying problem. Our game involves two sides: Blue and Red, both seeking to allocate their respective resources to minimize damage received while maximizing the effectiveness of the strike.

Blue plays by allocating its resources, a set of assets on aircraft (platforms), to Red's buildings (targets) and defensive installations (threats). Blue determines which targets to attack, which weapons (assets) to use on them, and how to route each platform to minimize risk and maximize effectiveness.

Red's defensive installations (threats) protect targets by threatening platforms that come within range. Red plays by placing these threats in space and time 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 game-playing 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 corridor of easy access to unprotected targets, a corridor containing a popup threat.

In this paper, a human player scripts Red's play while a Genetic Algorithm Player (GAP) plays Blue. The fitness of an individual in GAP's population solving the underlying allocation problem is evaluated by running the game. We explain our fitness evaluation in more detail in a later section. GAP develops strategies for the attacking strike force, including flight plans and weapon targeting for all available aircraft. When confronted with popups, GAP responds by replanning with the genetic algorithm in order to produce a new plan of action that responds to changes. Beyond purely responding to immediate scenario changes we use case injection in order to produce plans that anticipate opponent moves in the future.

2.1 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 pre-mission planning. Griggs [6] 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 [7] converts a nonlinear programming formulation into a MIP problem. Yost [8] provides a survey of the work that has been conducted to address the optimization of strike allocation assets. Louis [9] applied case injected genetic algorithms to strike force asset allocation. A large body of work exists in which evolutionary methods have been applied to games [10,11,12,13,14]. However the majority of this work has been applied to board, card, and other well defined games. Such games have many differences from popular real time strategy (RTS) games such as Starcraft, Total Annihilation, and Homeworld[15,16,17]. 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. 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 directly controlled by players but instead sets of parametrized algorithms control them in order to meet goals out-

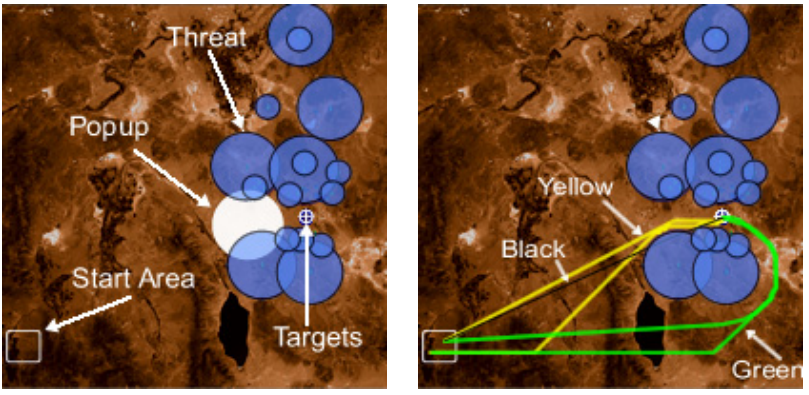

Fig. 1. Left: The Scenario Right: Route Categories

lined 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. John Laird [18,19,20] 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. Several military simulations share some of our game's properties [21,22,23], however these attempt to model reality while ours is designed to provide a platform for research in strategic planning, knowledge acquisition and re-use, and to have fun. The next section describes the scenario (or mission) used in our experiments.

3 The Scenario

Figure 1-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, Lake Tahoe is 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 search-able, but more complex scenarios quickly become intractable.

In this scenario, GAP's router can produce the three broad types of routes shown in Figure 1-Right.

1. Black - Flies inside the perimeter of known threats.
2. Yellow - Flies through the corridor in order to reach the targets.
3. Green - Flies around the threats, attacking the targets from behind.

Black routes expose platforms to unnecessary risk from threats and thus receive low fitness. The naively optimal strategy contains yellow routes which are the most direct routes to the target that still manage to avoid known threats. However in the presence of the popup, Green routes become optimal although they are longer then yellow routes. The evaluator looks only at known threats, so plans containing green routes receive lower fitness then those containing yellow routes. With experience GAP should learn to anticipate traps and to prefer green routes even though green routes have lower fitness then yellow 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 representing this state is rather unusual so we next detail it.

4 Probabilistic Health Metrics

In many games, entities (platforms, threats and targets in our game) posses hit-points which represents their ability to take damage. Each attack removes a number of hit-points and when reduced to zero hit-points the entity is destroyed and cannot participate further. However, weapons have a more hit or miss effect, entirely destroying an entity or leaving it functional. A single attack may be effective while multiple attacks may have no effect. Although more realistic, this introduces a degree of stochastic error into the game. In the worst case, evaluating a 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 becomes an effective means to judge 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 many-fold.

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

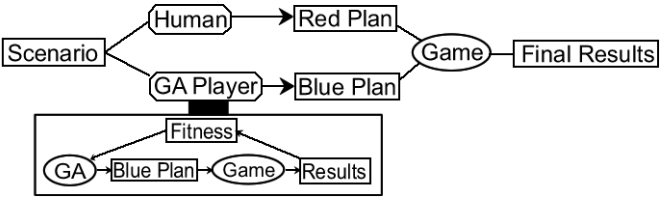

Fig. 2. System Architecture.

them from the game, it just reduces their probability of survival according to Equation 1 below.

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

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 equation 2 below.

$$D(E) = S(A) * E(W) \tag{2}$$

Here, $S(A)$ is the attackers probability of survival up until the time of the attack and $E(W)$ is the effectiveness of the attackers weapon as given in the weapon-entity effectiveness matrix. This method gives us the true 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. This technique is impractical when applied to more complicated relationships, but is effective at this stage of research.

The gaming system's architecture reflects the flow of action in the game and is described next.

5 System Architecture

Figure 2 outlines our system's architecture. Starting at the left, Red and Blue, human and GAP, are presented with the scenario and given 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 the popup is detected by GAP, 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 3 shows how routes are built using the A* algorithm [24]. A* builds routes from current platform locations to target locations and back and tends to prefer short routes that avoid threats while seeking targets.

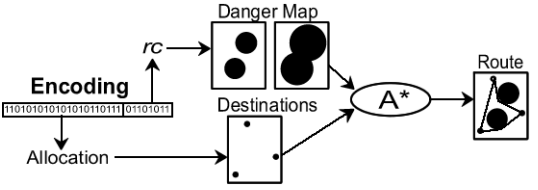

Fig. 3. How Routes are Built From an Encoding.

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 increases 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 black routes, $1.0 < rc < 1.35$ produce yellow routes and $rc > 1.35$ produce green routes. rc is limited to the range $[0, 3]$ and encoded with eight (8) bits at the end of our chromosome. We are encoding a single rc for each plan, future work may include rc's for each section of routing contained in the plan.

5.1 Encoding

Most of the encoding specifies the asset to target allocation with rc encoded at the end as detailed above. Figure 4 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.

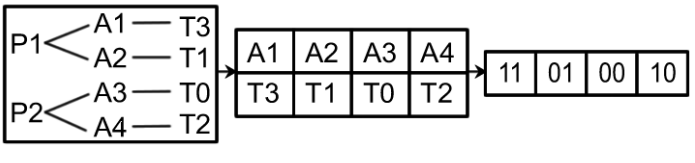

Fig. 4. Allocation Encoding

5.2 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 calculated as shown in Equation 3

$$fit(plan) = TotalDamage(Red) - TotalDamage(Blue) - d * c \qquad (3)$$

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 calculated 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.

6 Avoiding Traps with Case-Injection

We address the problem of learning from experience to avoid traps with a two part approach. First we learn from experience where traps are likely to be, then we apply that 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 towards future search.

GAP records games played against opponents and runs offline after a game playing episode in order to determine the optimal way to win that game. The simulation now contains knowledge about opponents moves, in our case, the game contains the popup trap. Allowing the search to progress towards the optimal strategy in the presence of the popup, GAP saves individuals from this search into the case-base, building a case-base with routes that go around the popup trap – green routes. When faced with other opponents, GAP then injects individuals from the case-base, biasing the current search towards containing this learned anticipatory knowledge.

In this paper GAP first plays the scenario, likely picking a yellow route and falling into Red's trap. Afterward GAP replays the game, including Red's trap into the evaluator. Yellow routes then receive poor fitness, and GAP searches towards the optimal green route. 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 containing important knowledge about how we should play, but how can we use that knowledge in order to play smarter in the future? Case Injection has been shown [2] to increase the search speed and the quality of the final solution produced by a GA working on a similar problem. It also tends to produce answers similar to old ones by biasing the search to look in areas that were previously successful – exploiting this effect

gives our GA its learning behavior. When playing the game we periodically inject a number of individuals from the case-base into the population, biasing our current search towards information from those individuals. Injection occurs by replacing the worst members of the population with individuals chosen from the case database through a "Probabilistic Closest to the Best" strategy [1]. Those 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.

7 Results

We present results showing

1. GAP can play the game effectively.
2. Replanning can effectively react to popups.
3. We can use case injection to learn to avoid the trap.

We also analyze the effect of altering the population size and number of generations on the strength of the biasing provided by case injection.

We first show that GAP can form efficient strategies. GAP is run against our test scenario 50 times, and we graph the min, max, and average population fitness against generation in Figure 5-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 5-Right illustrates the effect of replanning 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

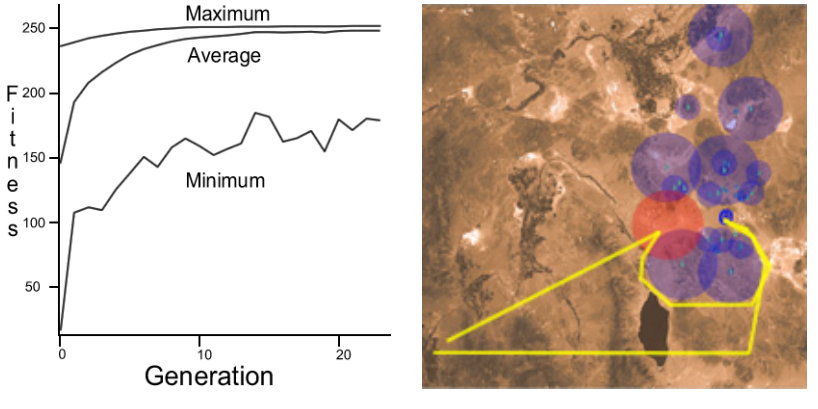

Fig. 5. Left: Best/Worst/Average Individual Fitness as a function of Generation - Averaged over 50 runs. Right: Final routes used during a mission involving replanning.

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.

GAP's ability to learn to avoid the trap is shown in Figure 6. The figure compares 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 towards using green routes. The effect of this bias being a large and statistically significant increase in the frequency at which strategies containing green routes were produced ($2\% - > 42\%$). These results were based on 50 independent runs of the system and show that case injection does bias the search toward avoiding the trap.

Figure 7-left compares the fitnesses with and without case injection. Without case injection the search shows a strong approach toward the optimal yellow plan; with injection the population quickly converges toward the optimal green plan. Case injection applies a bias towards green routes, however the GA has a tendency to act in opposition of this bias, trying to search towards ever shorter routes. The ability of the GA to overcome the bias through manipulation of injected material is dependent on the size of the population and the number of generations it runs. Figure 7-Right illustrates this effect. As the number of evaluations alloted to the GA is increased, the frequency of green routes being produced as a final solution decrease. Counteracting this tendency requires a careful balance of GA and case-injection parameters.

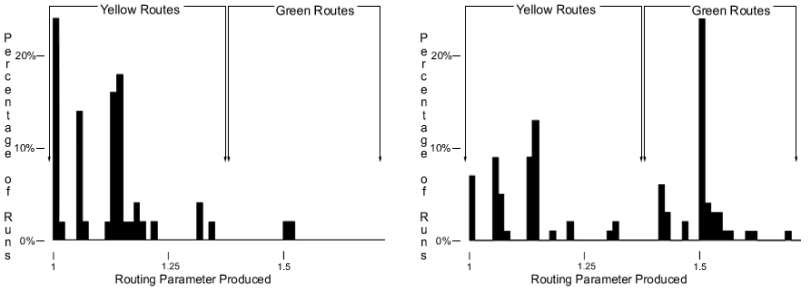

Fig. 6. Histogram of Routing Parameters produced without Case Injection.

8 Conclusions and Future Work

Results show that GAP is able to play the game, and that case injection can be used to to bias the search to incorporate knowledge from past game playing experience. We had expected difficulty in biasing the search, but we had underestimated the GA's resilience towards searching away from the optimum. We expected a stronger bias from case-injection - while 50% green is a significant

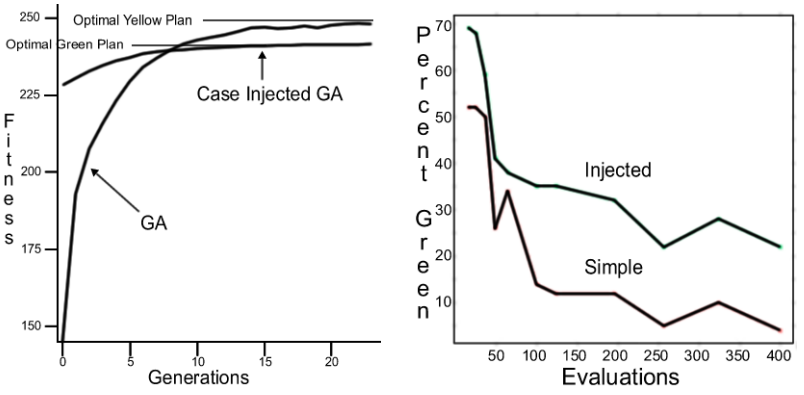

Fig. 7. Left: Effect of Case Injection on Fitness Inside the GA over time Right: Effect of Population Size and the Number of Generations on Percentage Green routes Produced

improvement on 2% we had hoped for numbers in the range of 80 to 90%. Even after extensive testing with the parameters involved we were unable to bias the search towards consistently producing plans containing green routes. However, preliminary results from new work show that artificially inflating the fitness of individuals in the population that contain injected material is an effective way of maintaining the preferred bias. This method appears to consistently produce green routes while maintaining an effective search across a range of problems without the need for parameter tuning.

There are a large number of interesting avenues in which to continue this research. Fitness inflation appears to solve one of our major problem in using case injection, further exploration of this technique is underway. We are also interested in capturing information from human players in order to better emulate their style of play. The game itself is also under major expansion, the next phase of research should involve a symmetric game involving aspects of resource management and much deeper strategies than those seen at the current level.

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

References

1. Louis, S.J., McDonnell, J.: Learning with case injected genetic algorithms. IEEE Transactions on Evolutionary Computation (To Appear in 2004)
2. Louis, S.J.: Evolutionary learning from experience. Journal of Engineering Optimization (To Appear in 2004)
3. Louis, S.J.: Genetic learning for combinational logic design. Journal of Soft Computing (To Appear in 2004)

4. Louis, S.J.: Learning from experience: Case injected genetic algorithm design of combinational logic circuits. In: Proceedings of the Fifth International Conference on Adaptive Computing in Design and Manufacturing, Springer-Verlag (2002) to appear
5. Louis, S.J., Johnson, J.: Solving similar problems using genetic algorithms and case-based memory. In: Proceedings of the Seventh International Conference on Genetic Algorithms, Morgan Kauffman, San Mateo, CA (1997) 283–290
6. Griggs, B.J., Parnell, G.S., Lemkuhl, L.J.: An air mission planning algorithm using decision analysis and mixed integer programming. Operations Research **45** (Sep-Oct 1997) 662–676
7. Li, V.C.W., Curry, G.L., Boyd, E.A.: Strike force allocation with defender suppression. Technical report, Industrial Engineering Department, Texas A&M University (1997)
8. Yost, K.A.: A survey and description of usaf conventional munitions allocation models. Technical report, Office of Aerospace Studies, Kirtland AFB (Feb 1995)
9. Louis, S.J., McDonnell, J., Gizzi, N.: Dynamic strike force asset allocation using genetic algorithms and case-based reasoning. In: Proceedings of the Sixth Conference on Systemics, Cybernetics, and Informatics. Orlando. (2002) 855–861
10. Fogel, D.B.: Blondie24: Playing at the Edge of AI. Morgan Kauffman (2001)
11. 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
12. 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
13. Kendall, G., Willdig, M.: An investigation of an adaptive poker player. In: Australian Joint Conference on Artificial Intelligence. (2001) 189–200
14. Samuel, A.L.: Some studies in machine learning using the game of checkers. IBM Journal of Research and Development **3** (1959) 210–229
15. Blizzard: Starcraft (1998, www.blizzard.com/starcraft)
16. Cavedog: Total annihilation (1997, www.cavedog.com/totala)
17. Inc., R.E.: Homeworld (1999, homeworld.sierra.com/hw)
18. Laird, J.E.: Research in human-level ai using computer games. Communications of the ACM **45** (2002) 32–35
19. Laird, J.E., van Lent, M.: The role of ai in computer game genres (2000)
20. Laird, J.E., van Lent, M.: Human-level ai's killer application: Interactive computer games (2000)
21. Tidhar, G., Heinze, C., Selvestrel, M.C.: Flying together: Modelling air mission teams. Applied Intelligence **8** (1998) 195–218
22. Serena, G.M.: The challenge of whole air mission modeling (1995)
23. McIlroy, D., Heinze, C.: Air combat tactics implementation in the smart whole air mission model. In: Proceedings of the First Internation SimTecT Conference, Melbourne, Australia, 1996. (1996)
24. Stout, B.: The basics of a* for path planning. In: Game Programming Gems, Charles River media (2000) 254–262