Finding Robust Strategies to Defeat Specific Opponents Using Case-Injected Coevolution

Christopher Ballinger and Sushil Louis
University of Nevada, Reno
Reno, Nevada 89503
{caballinger, sushil}@cse.unr.edu

***Abstract*—Finding robust solutions that are also capable of beating specific opponents presents a challenging problem. This paper investigates solving this problem by using case-injection with a coevolutionary algorithm. Specifically, we recorded winning strategies used by a human player against a coevolved strategy and then injected the player's strategies into the coevolutionary teachset. We compare the strategies produced by case-injected coevolution to strategies produced by a genetic algorithm that only evaluated against the player's strategies. In this paper, our results show that genetic algorithms do not work well against sufficiently difficult opponents. However, coevolution eventually learns to defeat these opponents by first bootstrapping strategies that work well in general, which drives the population closer to strategies that can defeat the challenging opponent. This work informs our research on finding robust real-time strategy game players that also defeat specific opponents.**

I. Introduction

Finding effective, robust strategies in Real-Time Strategy (RTS) games presents a challenging problem. RTS game players must manage many problems such as collecting resources, building military forces, gathering information on the opponent's movements, expanding control over the map, and eventually destroying their opponent's base. Managing even one of these problems presents a challenge, but an effective RTS player must manage them all simultaneously. We believe developing RTS game players that solve these problems will advance computational intelligence (CI) significantly, just as developing players for chess and checkers did.

Our overall goal focuses on finding competent RTS game players that defeat specific opponents without overspecializing. Strategies that overspecialize to defeat a single opponent often have easily identifiable weaknesses that many other strategies exploit. We plan to approach this problem by creating a system that continuously coevolves new strategies by playing against and learning from new opponents, as shown in Figure 1. Cases recorded from human players can be injected into coevolution's population and teachset, which will help coevolution find new strategies. These cases are also used to identify the current opponent's strategy, and find strategies that defeat similar opponent strategies. In this paper, we only investigate injection into the teachset, we do not consider injection into the population.

However, several problems make finding effective strategies difficult. Many types of units are available, each with their own costs, prerequisites, and strengths. Choosing which units to build, what order to build them and how soon they should be built leads to a combinatorially explosive number of possible strategies. No single strategy beats every strategy; a strategy that works well against one opponent may do poorly against a different opponent. Intransitive superiority, which Rock-Scissors-Paper exemplifies, relates to this issue [1]. Just because Strategy 1 beats Strategy 2 does not mean Strategy 1 is better overall, since Strategy 2 may beat Strategy 3, which in turn beats Strategy 1. It may also be the case that Strategy 1 only beats Strategy 2, but Strategy 2 defeats many opponents. Intransitive relationships between strategies makes coevolving strategies particularly difficult [2]. All these issues make exhaustive search through the strategy space infeasible, so we rely on heuristic search methods to find robust strategies. Specifically, this paper focuses on comparing a genetic algorithm (GA) to a coevolutionary algorithm (CA) that injects cases recorded from a human player into the teachset. Rather than evolving complete RTS game players, our preliminary work searches for good build-orders, strategies for what units to build and the order in which to build them.

Our previous research indicates a GA that trains against multiple opponents will find strategies that each defeat many of those opponents [3], [4]. Our previous research also indi-

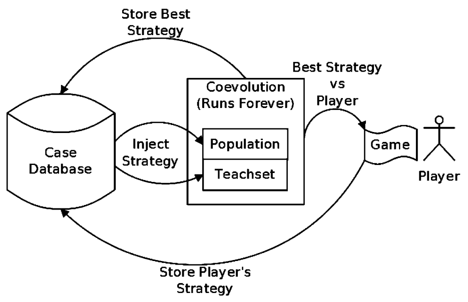

Fig. 1. Case-injected coevolution system.

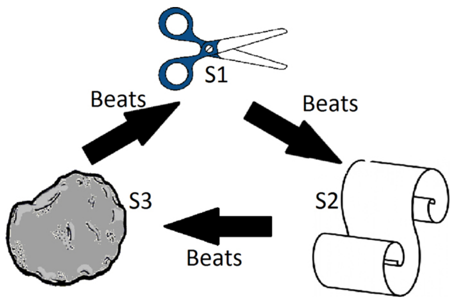

Fig. 2. Three strategies that defeat each other.

cates that a CA finds solutions that are robust against many opponents, but may not find strategies that defeat particular opponents [4], [5]. In this paper, we investigate how injecting cases into the coevolutionary teachset affects the strategies found by the CA. Does the CA find solutions to defeat those specific human strategies? How well do the strategies found by the CA perform against the human strategies? Does the CA find solutions to beat the human strategies slower than a GA that trains against only the human strategies? Are the strategies found by the CA the same strategies found by the GA?

This paper analyzes the results of ten CA runs and ten GA runs to answer the questions posed in the previous paragraph. Our results showed that a GA that trains against an easy human (EH) strategy and a hard human (HH) strategy at the same time, will only find strategies to defeat the EH strategy. Strategies that defeated the EH strategy could be found in the initial population, while strategies that defeated the HH strategy could not. As a result, the GA focused on strategies that could defeat the EH strategy and overspecialized. However, the CA found strategies that could defeat both human strategies after ten generations. The CA found a solution to the HH strategy by bootstrapping a diverse set of opponents to drive the population to more robust solutions closer to defeating the challenging opponent. Training against these different opponents also increased the rate at which the score improved against the player's strategy. These results show that case-injection into the teachset helps our CA find high-quality strategies faster than a GA, while maintaining robustness. This informs our current research on finding robust RTS strategies during a game that also defeat the current opponent.

The next section describes related work in game players using coevolutionary algorithms, our previous related research, and our own RTS game called "WaterCraft". In Section III we describe our GA and CA implementation, and how strategy fitnesses are calculated. Section IV presents our results in detail, and analyzes the differences between strategies found by the GA and CA. The final section provides our conclusion on how the results inform our research, and what steps we will take next to continue towards our overall goal.

II. Related Work

In the past, researchers have investigated coevolutionary approaches to designing players for board games. Chellapilla and Fogel coevolved the weights of an artificial neural network (ANN) to play Checkers [6]. The best ANN produced by their method played competitively against human opponents, winning most games. The ANN player also defeated an opponent who was 24 points away from the "Master" level and was ranked 98th out of 80000 registered players. Cowling also coevolved the weights of a ANN player to win at "The Virus Game" [7]. The resulting ANN player performed well and won against opponents never encountered during training. Davis and Kendall created a player for the game "Awari" by using coevolution to tune the weights of an evaluation function [8]. Using a population of 20 chromosomes, the best player after 250 generations defeated three out of four difficulty settings in the commercial Aware game "Awale". Nitschke used competitive coevolution in a pursuit-evasion game to create pursuer-players that cooperated with each other to capture one of the evader-players [9].

More recently, researchers have turned their attention to computer games where players do not have discrete turns. Cardamone compared cooperative coevolution to a GA for tuning the parameters of a particular AI player in the racing simulator "TORCS" [10]. The parameters found by coevolution performed better than the parameters found by the GA, and the AI player using the parameters found by coevolution placed 4th in the 2009 TORCS Endurance World Championship, against opponents that used human expertise to tune the AI and car configurations. Avery and Louis did work on coevolving team strategies using influence maps, allowing a group of entities to adapt to opponent moves [11]. Keaveney and Riordan used an abstract RTS game to find players that coordinated their movements with their allies [12]. They coevolved one population of players only on one map, and a second population of players on multiple maps. While the players coevolved on one map were able to win on maps not used during training, the players coevolved on multiple maps performed better on the maps not used during training.

In addition to coevolutionary methods, researchers have used other online and offline methods to find good RTS strategies, such as case-based reasoning, genetic algorithms, dynamic scripting, and reinforcement learning [13], [14], [15], [16], [17]. While some research focuses on producing a player to manage an entire RTS game, other research focuses on finding good strategies for specific RTS game elements, such as combat, positioning, navigation, cooperation and resource management [18], [19], [20], [21].

Research into case-injection often involves injecting cases into the population of a GA, which has been shown to help the GA find higher quality solutions in a shorter amount of time [22], [23], [24], [25]. Injecting relevant cases into a GA's population will bias the GA towards finding similar solutions that work well. Previous research also examines the problems of deciding which cases to inject, and when to inject them [22]. However, while these previous works focused on injecting cases into the population, in this paper we only consider injecting cases into the teachset.

Our prior work in this domain investigated bit-setting hill-climbers (HCs), GAs, and CAs for finding robust strategies in RTS games. We started our research by evaluating five-action strategies against three hand-tuned baselines that used eight to thirteen actions, and comparing the five-action strategies found by a GA and HC to exhaustive search [3]. We extended this study to include coevolution, first by coevolving five-action strategies we could compare to the GA strategies and exhaustive search, and then coevolving thirteen-action strategies that were not at a disadvantage against the baselines [4], [5]. We will briefly cover the results of these previous studies in Section IV.

We developed WaterCraft, seen in Figure 3, for researching evolutionary algorithms in RTS games and wrote Watercraft primarily in Python with some C/C++ to speed up physics. WaterCraft uses the popular Python-OGRE graphics engine for the GUI and graphics display [26]. We modeled the game play in WaterCraft around the popular commercial game, StarCraft [27]. While Watercraft lacks some features provided in commercial games, we have implemented some of the core features of all RTS games. Players can build several types of buildings and units, with the objective of destroying their opponent's base. WaterCraft's graphical user interface (GUI) resembles and functions similar to the player GUI in other RTS games. The player also has the option of playing against the AI or another human player over the network. We also model all the game mechanics and units in WaterCraft around StarCraft, but we use a naval-based theme instead of a science-fiction space theme since our research interests also involve military and defense applications.

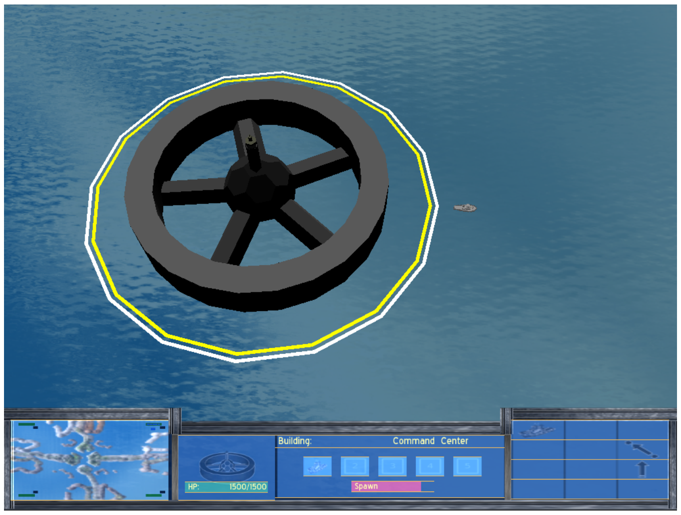

Fig. 3. A Command Center as it produces a new SCV.

Projects similar to WaterCraft have been used in the past for research in RTS games. The BroodWar API (BWAPI) allows developers to use StarCraft for research by providing an interface that allows researchers to retrieve game-state information and interact with units [28]. Stratagus provides a free, cross-platform RTS engine that has been used to create custom RTS games useful for research [29]. Stratagus has been used as the back-end for another popular research-oriented RTS game call Wargus, which uses the entity data from the commercial game Wargus II [30]. Like Wargus, Stargus uses Strategus as the back-end, but uses the entity data from StarCraft [31]. ORTS, another RTS engine, provides a total programming environment for computational and artificial intelligence research, including a graphical client [32]. While these other projects are designed for CI and AI research in general, we created WaterCraft with our research needs specifically in mind. In the next section, we describe the methods we use when evaluating a strategy in WaterCraft and finding new strategies that can beat an opponent in WaterCraft.

III. Methodology

Depending on our needs, we run WaterCraft either with or without graphics. Disabling the graphics allows us to maximize the rate the game plays at. This allows us to quickly evaluate the outcome of two artificial players (game AIs) competing against each other and give them a fitness. Enabling the graphics allows a human player to interact with the game, issues commands, and observe the outcome in real-time. As the player issues commands, the commands are encoded to a bit-string which represents the player's strategy. Matches in WaterCraft are deterministic, meaning a match between two strategies will always have the same outcome, regardless of whether the graphics are enabled or disabled. This allows us to take the strategy recorded from the human player with the graphics enabled, and quickly evolve counter-strategies with the graphics disabled.

TABLE I. Unit Encodings

Bit Sequence	Action	Prerequisites
000-001	Build SCV (Gather Minerals)	None
010	Build Marine	Barracks
011-100	Build Firebat	Barracks, Refinery, Academy
101	Build Vulture	Barracks, Refinery, Factory
110	Build SCV (Gather Gas)	Refinery
111	Attack	N/A

We encode strategies as sequences of commands. GAs and CAs prefer a binary representation, which has the best pattern-to-schema ratio [33]. We use the same binary encoded sequence of commands for GA, CA and human-recorded strategies, to make strategies produced by different methods easier to compare. Every three bits in the chromosome represents an action, as shown in Table I. When the game AI in WaterCraft receives a chromosome, the game AI sequentially decode the chromosome and inserts the encoded actions into a queue. If any prerequisites in Table I are missing for the action, the game AI inserts the missing prerequisites immediately before the encoded action. This means that a single encoded action may cause several actions to be inserted into queue. In our previous work, we limited ourselves to 15-bit (5-action) chromosomes, so we could perform an exhaustive search of strategies and compare them to the strategies found by the GA and CA [3], [4]. In a later study, we investigated how increasing the chromosome length to 39-bits (13-actions) affected our results [5]. In this study, we continue to use 39-bit chromosomes, so we can compare our results to our previous work.

For our research, we allow players to build four different units. We modeled the unit strengths and costs after StarCraft units, which are balanced against each other. Marines are a basic offensive unit that are quick and cheap to build, and are produced from a Barracks. Firebats are stronger than Marines, but cost more to produce and require an Academy to be built in addition to the Barracks. Vultures are our strongest unit, but take the longest to produce. SCVs (Space Construction Vehicles) are support units with limited offensive capabilities, but have the ability to gather resources and build new structures. Since the "Build SCV (Gather Minerals)" and "Build Firebat" actions are encoded twice, the GA is biased towards selecting these actions during crossover and mutation. However, strategies that are biased towards actions will quickly go extinct in the population, since strategies that make more effective use of their resources will have a higher fitness.

All game AIs follow the same rules for managing their units. Game AIs will sequentially issue commands from the queue as quickly as possible. If a command cannot be issued due to a lack of resources, the game AI waits until enough resources are gathered instead of skipping to an affordable action. The game AI may be "dumb" in this respect, and relies on the GA and CA to find strategies that use an effective order for the actions. When the game AI reaches an "Attack" command, all completed Marines, Firebats and Vultures are sent to attack the opponent's Command Center. These units will also attack any nearby opponent units and structures they encounter along the way. When a Command Center comes under attack, SCVs that were gathering resources will defend the Command Center by attacking the aggressors. Once all aggressors are destroyed, the SCVs return to their previous tasks. Players can immediately win and end the game by destroying their opponent's Command Center.

Since we are only looking to find build-orders instead of complete AI players, our game AI does not look at micromanagement. However, when we had our human player compete against the strategies to determine difficulty, we used two different setups. In the first setup we removed micromanagement from the human player. The human could determine what units to build as the game progressed, but the unit behavior and movement automatically followed the same rules as the game AI. In the second setup, we enabled micromanagement and allowed the human player to instruct each individual unit. Since our current representation cannot encode micromanagement, we use the first setup for recording human strategies to evolve against. We use the second setup to further test the difficulty and robustness of the strategies found by our GA and CA.

At the end of a game, players determine how well they performed by calculating a score, which we also use as a strategy's fitness for our search methods. We want strategies that spend as many resources as possible, since the most expensive units and structures tend to be the strongest and most useful. We also want strategies that destroy many of their opponent's units, essentially wasting the opponent's resources. Finally, we want strategies that destroy their opponent's structures, since structures cost many resources to build, take a long time to build, and are required to produce new units. To find strategies that meet these goals, we calculate fitness as show in Equation 1 to reward them for having those qualities.

$$F_{ij} = SR_i + 2\sum_{k \in UD_j} UC_k + 3\sum_{k \in BD_j} BC_k \tag{1}$$

In Equation 1, F_{ij} is the total fitness of individual i against individual j. To encourage our first goal, we calculate SR_i, which is the amount of resources spent by individual i. To meet the requirements of our second goal, we calculate the sum of destroyed opponent unit cost, where UD_j is the set of units owned by individual j that were destroyed, and UC_k is the cost to build unit k. Our third goal is rewarded by taking the sum of destroyed opponent structure cost, where BD_j is the set of buildings owned by individual j that were destroyed, and BC_k is the cost to build structure k.

A. Coevolution

We use several methods defined by Rosin and Belew in our CA to evaluate the fitness of individuals in a population [34]. To calculate the fitness of individuals in the population, every individual plays against eight opponents in a teachset. We populate the teachset with opponents from three different sources: three random opponents from the hall of fame, three from shared sampling, and two from case-injection.

The hall of fame keeps a history of chromosomes that have performed well in past generations. Adding chromosome from the hall of fame to the teachset prevents the population from forgetting how to defeat strategies that have gone extinct in the current population.

Shared sampling allows us to find diverse, challenging opponents so the population avoids overspecializing while minimizing the number of opponents the population must play against [34]. When selecting an individual for the teachset, shared sampling gives less weight to individuals for defeating opponents already defeated by current members of the teachset. Sample fitness must be recalculated each time we select a new chromosome for the teachset, so that remaining members of the population that defeat different opponents are given higher preference.

$$s_i = \sum_{j \in D_i} \frac{1}{1 + j_b} F_{ji} \tag{2}$$

We calculate the sample fitness using Equation 2, where s_i is the sample fitness, D_i is the set of teachset members chromosome i defeated, j_b is the number of times j has been defeated by chromosomes already selected by shared sampling, and F_{ji} is the fitness of teachset member j against chromosome i.

Teachset case-injection takes individuals from an outside source, and places the individual into the teachset for every generation. In our case, we inject winning strategies a human player used to defeat the best strategy found by coevolution. This makes the CA prefer solutions that can defeat the human player's strategy, while the remainder of the teachset prevents the CA from overspecializing against the human strategies.

We limit our teachset to eight opponents, because that was the highest number of opponents our population could evaluate against in a reasonable amount of time. Evaluating the fitness of fifty chromosomes in a population against eight opponents requires 400 evaluations.

Once all individuals in the population have a fitness against all opponents in the teachset, we calculate shared fitness [34]. Usually, only chromosomes that defeat many opponents have a high fitness. However, shared fitness gives a high fitness to individuals that defeat opponents few other individuals can defeat. If an individual defeats only one opponent, but no other individual can defeat that opponent, we want that individual to have a high fitness because it contains important information on winning that no other individual has. We also multiply the shared fitness by the score the individual got against each opponent. If two individuals defeat the same opponents, but one individual does better at winning, we give the individual with the better performance a higher fitness. We calculated fitness sharing as shown in Equation 3. Where f_i^{shared} is the shared fitness of chromosome i, D_i is the set of teachset members chromosome i defeated, j a teachset member in D_i, L_j is the number of times j lost against all chromosomes, and F_{ij} is the fitness of chromosome i against teachset member j.

$$f_i^{shared} = \sum_{j \in D_i} \frac{1}{L_j} F_{ij} \tag{3}$$

L_j is the total number of individuals that teachset member j lost to in the current population. We calculate L_j using

Equation 4, where P is the set of all chromosomes in a population, and i is an element of P. $GameResult$ is a function that returns 1 if teachset member j lost against individual i, and returns 0 if teachset member j won against individual i, as shown by Equation 5. Since Equation 3 only takes the sum of teachset members that were defeated L_j can never be 0, since if a teachset member j was never defeated j would not be in set D_i.

$$L_j = \sum_{i \in P} GameResult(j, i) \tag{4}$$

$$GameResult(j, i) = \left\{ \begin{array}{ll} 0, & F_{ji} >= F_{ij} \\ 1, & F_{ji} < F_{ij} \end{array} \right\} \tag{5}$$

Once we calculate shared fitness for the population, we use linear fitness scaling with a factor of 1.5. This prevents an exceptionally good individual from suddenly taking over the population. Instead, we even out the selection pressure by adjusting the fitness of each individual accordingly, so we produce more diverse children. Strong individuals still are selected more often, but only to a certain threshold.

With the fitness of each individual appropriately scaled, we produce a population of child chromosomes using roulette wheel selection, uniform crossover with a 95% occurrence, and a .1% chance of each individual bit in a chromosome mutating. Our experimental results showed that these parameters gave us the best results. These parameters allow our CA to explore many new strategies, while exploiting the progress made by current strategies. We repeat this process until the child population contains the same number of individuals as the parent population. We then play the child population against the same opponents the parent population played against, and give each child a shared fitness. Finally, we use CHC selection to select the best chromosomes from the child and parent population, to produce our new population for the next generation [35]. This prevents valuable information in the parent population from being lost if our CA produces many unfit children.

B. Genetic Algorithm

Implementing a GA to find build-orders was simple, as we use a slightly modified version of our CA. We removed shared sampling and the hall of fame from the CA, so the teach set only contains the two human strategies in every generation. All other methods and parameters in our GA are identical to our CA.

IV. Results

In our previous work, we compared 39-bit strategies produced by our CA to strategies produced by a GA evaluating against three baseline strategies [5]. The baselines used to evaluate the GA strategies were designed to provide a diverse set of challenges. The first baseline built five Marines and attacked, which could quickly defeat an opponent with a weak attack force. The second baseline built ten Marines and attacked, taking a little longer than the first baseline, but using a much stronger attack force. The third baseline built five Vultures and attacked, which took a very long time to build but provided the strongest attack force.

Our GA produced a strategy that builds two SCVs, several Firebats, another SCV, several more Firebats, two Vultures, then attacks. Interestingly, this strategy builds the third SCV seconds after SCVs are destroyed by the second baseline, showing that this strategy may be slightly overspecialized. The CA produced three effective strategies to defeat different opponents. The "Single Attack" strategy builds two SCVs, followed by mostly Vultures, then two Firebats and attacks. The "Double Attack" strategy builds one SCV, five Firebats, attacks, builds five more Firebats, and attacks again. The "Defensive" strategy focused purely on defense by building a few Firebats followed by Vultures. We put our strategies into five groups: the three baselines were in the Baseline group, the single best GA strategy was put in the GA group, the three best strategies in the CA population were put in the CA group, the three best strategies in the CA teachset were put in the Teachset group, and ten randomly created strategies were put into the Random group. To evaluate the quality of our strategies, we had every group compete against every other group and took the average score of all strategies in a group against all their opponents.

Our results from this previous study showed that as expected, the GA defeated all three baselines and got the highest average score. The GA also performed well in general, and got a higher average score than the baseline or random players against all the groups.The CA never trained against the baselines, but still produced solutions that did well against two of the baselines. However, the CA solutions got a higher average score and more wins against all other opponents than any other group, including the GA. The CA was robust enough that the Random group could not beat any strategies in the CA group, while the GA group was slightly more exploitable, showing that the CA produces more robust strategies.

For our current work, one of the authors acted as the human player and competed against the strategies produced by a GA and CA from our previous study. Our human player is "Gold" rank in StarCraft II and found that the CA Single Attack strategy was the most challenging to defeat. We plan to more formally compare strategies against other human players later. We recorded the player's actions as they competed against the CA strategy, and found two significantly different strategies that could win. The first winning strategy used by the human player, which we call the EH strategy, was to build two Marines, attack, and repeat six times. This strategy slowly chipped away at the CA strategy's Command Center, while also destroying some of the CA strategy's SCVs early on and slowing down how quickly structures were built. The second winning strategy used by the human player, which we call the HH strategy, was to build nine SCVs, then build seven Firebats and seven Vultures in parallel. This strategy builds a strong defense and waits to destroy the CA strategy's attack force. Once the human player destroyed the CA strategy's attack force, the human player builds a few more units and destroys the CA strategy's Command Center. Both human strategies required 75-bits to encode, compared to the 39-bits the CA strategy used. We then reinitialized the CA population to random chromosomes and injected the two human strategies into the CA's teachset. We also removed the baselines from our GA, and instead ran the GA against only the human player's strategies. We ran our GA and CA ten times, and used the average score of the entire population at each generation for

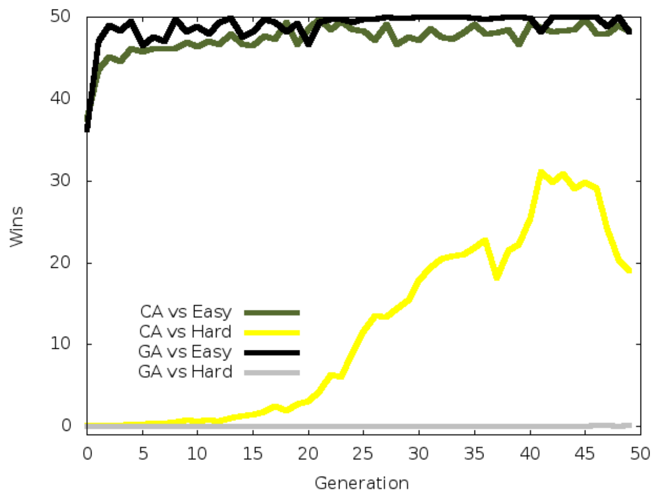

Fig. 4. Average Number of Wins Against Human Strategies.

our results.

Our results show that the EH strategy was trivial to beat, and often could be beaten by random chromosomes, as shown by Figure 4. On the other hand, the HH strategy proved to be overwhelmingly difficult, and was never defeated in the initial generation. This balance issue caused a problem for the GA. Although improving the score against the EH strategy also slightly improved the score against the HH strategy, as shown in Figure 5, the improvement did not lead to successful strategies against the HH strategy. As a result, the GA produced strategies that were overspecialized to defeat the EH strategy. The strategies found by the GA quickly build two SCVs and a couple Firebats to ward off the Marines while minimizing casualties, followed by a mix of Firebats and Vultures used to attack the opponent's base. The GA finds the best strategy to defeat the EH strategy after about thirty generations.

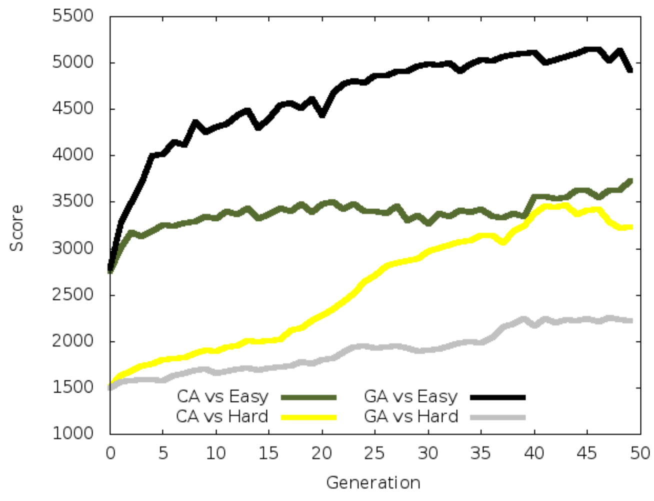

Fig. 5. Average Score Against Human Strategies.

However, our CA was able to quickly find solutions to defeat both human strategies, as shown in Figure 4. The CA typically found at least one strategy to defeat the HH strategy by generation ten. We also see that the average score against the EH strategy peaks almost immediately, while the GA continues to improve and overspecialize over all fifty generations. While the CA also immediately finds strategies to defeat the EH strategy, the other opponents in the teachset force the CA to explore more possible solutions and prevent the CA from overspecializing at the beginning. This leads to enough diverse strategies that work well in general that the CA finds solutions that also work against the HH strategy. These strategies build two SCVs, followed by Vultures, then attack. This strategy loses a few more units to the EH strategy than the GA does, but enables the strategy to build a strong enough defense to defend against the HH strategy. As Figure 5 shows, this compromise lowers the score against the EH strategy significantly compared to the strategies found by the GA but dramatically increases the score against the HH strategy.

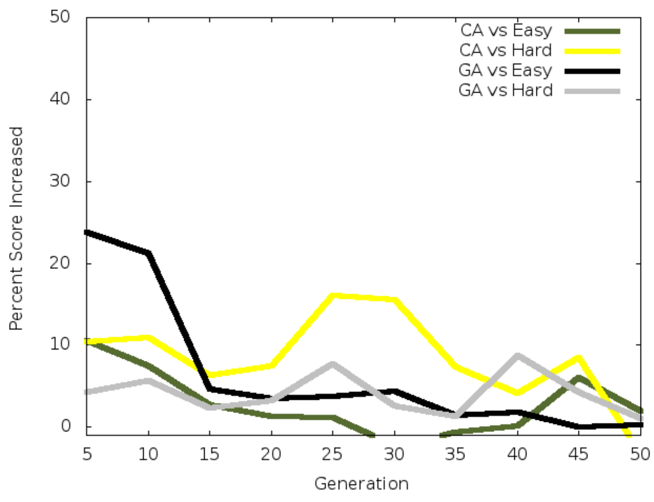

Fig. 6. Average Percent Score Increased Against Human Strategies.

Although the initial scores for the GA and CA against the human strategies start off the same in Figure 5, the average rate at which the scores increase are immediately different. Figure 6 shows the average increase in score the entire population received against the human strategies. We see that although the initial GA population cannot defeat the HH strategy, the fitness against the HH strategy increases by 5% for the first ten generations. The initial CA population also cannot defeat the HH strategy, but increases the score against the HH strategy by 10%. The GA population increases the score against the EH strategy much faster, since all strategies in the population are evolving to beat only the EH strategy. The CA population not only evolves strategies to defeat the EH strategy, but also strategies that defeat other opponents at the cost of lowering the score against the EH strategy. Both the CA and GA do the most learning in the first fifteen generations, at which point the amount the scores increase each generation becomes much smaller.

V. Conclusion and Future Work

In this paper, we want to find robust strategies that also defeat specific opponents. In our previous work, we compared strategies found by a genetic algorithm training against three baselines to strategies found by a coevolutionary algorithm. We then had a human player compete against the strategies produced by the genetic algorithm and coevolutionary algorithm. The human player found that the strategies produced by the coevolutionary algorithm were the most challenging to defeat.

This paper expands upon our previous work, which can be found on the author's website, by introducing case-injection to our coevolutionary algorithm's teachset, and comparing strategies found by the coevolutionary algorithm to the strategies found by a genetic algorithm. We had our human player compete several times against the most challenging strategy found by coevolution in our previous study. As the human player competed, we recorded their actions to a bit-string, allowing us to replicate their actions and outcome against the opponent. We recorded two successful and vastly different strategies the human player used against coevolution's strategy. We then used the human player's strategies to evaluate strategies using a genetic algorithm, and injected the same two player strategies into coevolution's teachset. We only consider case-injection into the coevolutionary teachset for this paper, and not case-injection into the coevolutionary population.

Our results showed that when an opponent poses a significant challenge, the genetic algorithm will not find strategies to defeat the opponent, and will instead over specialize against the easier opponent. This was unexpected, since genetic algorithms have been shown to produce good solutions against the problems used in training. However, when there are multiple opponents with a large difference in difficulty, the genetic algorithm may converge too quickly on solutions for the easy opponents, which does not lead to solutions for the harder opponents. On the other hand, coevolution finds strategies to defeat both opponents after ten generations. Coevolution finds solutions to beat the challenging opponent because coevolution uses a diverse teachset that gradually increases in difficulty. This prevents coevolution from converging to quickly, and allows coevolution to move towards strategies more capable of defeating the challenging opponent, even if no strategies currently beat that opponent. We also show that coevolution increases the population's average score against the challenging opponent much faster than the genetic algorithm does. These results are interesting compared to our previous studies, where our genetic algorithm produced better strategies than coevolution for defeating specific opponents, despite the genetic algorithm only competing against the same three opponents while coevolution competed against eight opponents that changed over time. This shows that while genetic algorithms can produce good strategies to defeat specific opponents, genetic algorithms can also be mislead if only a few opponents are used for evaluation, or if the opponents have a large gap in difficulty.

These results indicate that teachset case-injection helps coevolution to quickly find strategies that defeat specific opponents, while maintaining robustness against other opponents. This informs our research into adapting new strategies during a game to defeat the current opponent. In our future research, it would be interesting to increase our strategy length far beyond 13 actions and see how this affects the strategies produced by our GA and CA. We also plan investigate alternative strategy representations that allow us to model human strategies more accurately, such as determining where to move units or which opponent units to attack. This would allow us to encode better human strategies as injectable cases. We also believe this will allow us to find more robust strategies that can react to the current opponent's actions. We also intend to research how to identify the possible strategies the opponent might be planning and what strategy the game AI should use to defeat the strategies available to the opponent. Finally, we plan to maintain a case-base of strategies used by players and coevolution, and inject cases into coevolution's population and teachset while perpetually running coevolution and finding new effective strategies.

Acknowledgment

This research is supported by ONR grant N000014-12-C-0522.

References

[1] R. A. Watson and J. B. Pollack, "Coevolutionary dynamics in a minimal substrate." Morgan Kaufmann, 2001, pp. 702–709.

[2] S. Samothrakis, S. Lucas, T. Runarsson, and D. Robles, "Coevolving game-playing agents: Measuring performance and intransitivities," *Evolutionary Computation, IEEE Transactions on*, vol. 17, no. 2, pp. 213–226, 2013.

[3] C. Ballinger and S. Louis, "Comparing heuristic search methods for finding effective real-time strategy game plans," in *2013 IEEE Symposium Series on Computational Intelligence*, April 2013.

[4] ——, "Comparing coevolution, genetic algorithms, and hill-climbers for finding real-time strategy game plans," in *Genetic and Evolutionary Computation Conference, GECCO 2013*, July 2013.

[5] ——, "Robustness of real-time strategy game build-orders produced by coevolution (in consideration)," in *Evolutionary Computation, 2013. Proceedings of the 2001 Congress on*. IEEE, 2013.

[6] K. Chellapilla and D. Fogel, "Evolving an expert checkers playing program without using human expertise," *Evolutionary Computation, IEEE Transactions on*, vol. 5, no. 4, pp. 422 –428, aug 2001.

[7] P. Cowling, M. Naveed, and M. Hossain, "A coevolutionary model for the virus game," in *Computational Intelligence and Games, 2006 IEEE Symposium on*, may 2006, pp. 45 –51.

[8] J. Davis and G. Kendall, "An investigation, using co-evolution, to evolve an awari player," in *Evolutionary Computation, 2002. CEC '02. Proceedings of the 2002 Congress on*, vol. 2, 2002, pp. 1408 –1413.

[9] G. Nitschke, "Co-evolution of cooperation in a pursuit evasion game," in *Intelligent Robots and Systems, 2003. (IROS 2003). Proceedings. 2003 IEEE/RSJ International Conference on*, vol. 2, oct. 2003, pp. 2037 – 2042 vol.2.

[10] L. Cardamone, D. Loiacono, and P. Lanzi, "Applying cooperative coevolution to compete in the 2009 torcs endurance world championship," in *Evolutionary Computation (CEC), 2010 IEEE Congress on*, july 2010, pp. 1 –8.

[11] P. Avery and S. Louis, "Coevolving influence maps for spatial team tactics in a rts game," in *Proceedings of the 12th annual conference on Genetic and evolutionary computation*, ser. GECCO '10. New York, NY, USA: ACM, 2010, pp. 783–790. [Online]. Available: http://doi.acm.org/10.1145/1830483.1830621

[12] D. Keaveney and C. O'Riordan, "Evolving robust strategies for an abstract real-time strategy game," in *Computational Intelligence and Games, 2009. CIG 2009. IEEE Symposium on*, sept. 2009, pp. 371 – 378.

[13] J. Young, F. Smith, C. Atkinson, K. Poyner, and T. Chothia, "Scail: An integrated starcraft ai system," in *Computational Intelligence and Games (CIG), 2012 IEEE Conference on*, 2012, pp. 438–445.

[14] B. G. Weber, M. Mateas, and A. Jhala, "Building human-level ai for real-time strategy games," in *Proceedings of the AAAI Fall Symposium on Advances in Cognitive Systems*, AAAI Press. San Francisco, California: AAAI Press, 2011.

[15] P. Spronck, M. Ponsen, I. Sprinkhuizen-Kuyper, and E. Postma, "Adaptive game ai with dynamic scripting," *Mach. Learn.*, vol. 63, no. 3, pp. 217–248, Jun. 2006. [Online]. Available: http://dx.doi.org/10.1007/s10994-006-6205-6

[16] S. Ontañón, K. Mishra, N. Sugandh, and A. Ram, "Case-based planning and execution for real-time strategy games," in *Proceedings of the 7th international conference on Case-Based Reasoning: Case-Based Reasoning Research and Development*, ser. ICCBR '07. Berlin, Heidelberg: Springer-Verlag, 2007, pp. 164–178. [Online]. Available: http://dx.doi.org/10.1007/978-3-540-74141-1_12

[17] C. S. Huat and J. Teo, "Online evolution of offensive strategies in real-time strategy gaming," in *Evolutionary Computation (CEC), 2012 IEEE Congress on*, 2012, pp. 1–8.

[18] S. Wender and I. Watson, "Applying reinforcement learning to small scale combat in the real-time strategy game starcraft:broodwar," in *Computational Intelligence and Games (CIG), 2012 IEEE Conference on*, 2012, pp. 402–408.

[19] J. Hagelback, "Potential-field based navigation in starcraft," in *Computational Intelligence and Games (CIG), 2012 IEEE Conference on*, 2012, pp. 388–393.

[20] G. Synnaeve and P. Bessiere, "Special tactics: A bayesian approach to tactical decision-making," in *Computational Intelligence and Games (CIG), 2012 IEEE Conference on*, 2012, pp. 409–416.

[21] D. Churchill and M. Buro, "Build order optimization in starcraft," in *Artificial Intelligence and Interactive Digital Entertainment (AIIDE 2011)*, 10/2011 2011.

[22] S. Louis and J. McDonnell, "Learning with case-injected genetic algorithms," *Evolutionary Computation, IEEE Transactions on*, vol. 8, no. 4, pp. 316–328, 2004.

[23] J. Amunrud and B. A. Julstrom, "Instance similarity and the effectiveness of case injection in a genetic algorithm for binary quadratic programming," in *Proceedings of the 8th annual conference on Genetic and evolutionary computation*, ser. GECCO '06. New York, NY, USA: ACM, 2006, pp. 1395–1396. [Online]. Available: http://doi.acm.org/10.1145/1143997.1144212

[24] R. Drewes, S. J. Louis, C. Miles, J. McDonnell, and N. Gizzi, "Use of case injection to bias genetic algorithm solutions of similar problems," in *Evolutionary Computation, 2003. CEC'03. The 2003 Congress on*, vol. 2. IEEE, 2003, pp. 1170–1177.

[25] S. Louis and C. Miles, "Playing to learn: case-injected genetic algorithms for learning to play computer games," *Evolutionary Computation, IEEE Transactions on*, vol. 9, no. 6, pp. 669 – 681, dec. 2005.

[26] Torus Knot Software Ltd, "Ogre - open source 3d graphics engine," February 2005. [Online]. Available: http://www.ogre3d.org/

[27] Blizzard Entertainment, "Starcraft," March 1998. [Online]. Available: http://us.blizzard.com/en-us/games/sc/

[28] "Bwapi: An api for interacting with starcraft: Broodwar." [Online]. Available: http://code.google.com/p/bwapi/

[29] M. J. V. Ponsen, S. Lee-urban, H. Muoz-avila, D. W. Aha, and M. Molineaux, "Stratagus: An open-source game engine for research in real-time strategy games," Naval Research Laboratory, Navy Center for, Tech. Rep., 2005.

[30] The Wargus Team, "Wargus," 2011. [Online]. Available: http://wargus.sourceforge.net

[31] "Stargus," 2009. [Online]. Available: http://stargus.sourceforge.net/

[32] M. Buro, "Orts - a free software rts game engine," 2005. [Online]. Available: http://skatgame.net/mburo/orts/

[33] D. E. Goldberg, "Genetic algorithms in search, optimization, and machine learning," 1989.

[34] C. D. Rosin and R. K. Belew, "New methods for competitive coevolution," *Evol. Comput.*, vol. 5, no. 1, pp. 1–29, Mar. 1997. [Online]. Available: http://dx.doi.org/10.1162/evco.1997.5.1.1

[35] L. J. Eshelman, "The chc adaptive search algorithm : How to have safe search when engaging in nontraditional genetic recombination," *Foundations of Genetic Algorithms*, pp. 265–283, 1991. [Online]. Available: http://ci.nii.ac.jp/naid/10000024547/en/