Learning Robust Build-Orders from Previous Opponents with Coevolution

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

***Abstract*—Learning robust, winning strategies from previous opponents in Real-Time Strategy games presents a challenging problem. In our paper, we investigate this problem by using case-injection into the teachset and population of a coevolutionary algorithm. Specifically, we take several winning build-orders we created through hand-tuning or coevolution and periodically inject them into the coevolutionary population and/or teachset. We compare the build-orders produced by three different case-injection methods to the robustness of build-orders produced without case-injection and measure their similarity to all the injected cases. Our results show that case injection works well with a coevolutionary algorithm. Case injection into the population quickly influences the strategies to play like some of the injected cases, without losing robustness. This work informs our ongoing research on finding robust build-orders for real-time strategy games.**

I. Introduction

Finding robust build-orders in Real Time Strategy (RTS) games that defeat multiple opponents presents a challenging problem. There are several problems that RTS players must manage in order to win: gathering resources, building infrastructure, amassing military forces, scouting concealed areas for intel on their opponent, expanding control over the map, and ultimately destroying their opponent's entire army and base. Each of these problems present a difficult challenge on their own, but a player must effectively manage all these problems simultaneously in order to succeed. In this context, a build-order is the military forces and infrastructure a player chooses to build, and the order in which they player choose to build them. Build-orders that do not choose the most effective military units for a given opponent, or do not produce the units when they are needed most, are doomed to failure. 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.

In this paper, our overall goal is finding effective RTS build-orders that incorporate elements from build-orders used by other players, while remaining robust. Learning to play like other competent players helps us learn new effective combinations and learn winning build-orders faster. We plan to approach this problem by creating a system that continuously coevolves new strategies by playing against and learning from new opponents.

With case injection, winning build-orders produced by human players, coevolution, or other CI methods can be stored in a case base and injected into coevolution's population and teachset at an appropriate time. These cases help coevolution identify the current opponent's build-order, find build-orders that defeat similar opponents, and learn new useful build-orders. In this paper, we investigate case injection into a coevolutionary algorithm's population and teachset as a method to influence coevolving build-orders to learn from other players.

However, several problems make finding effective build-orders difficult. There are many units and structures to choose from, each with their own unique cost, prerequisites, strengths, weaknesses, and special abilities. Deciding which units or structures to build, what order we should build them in, how quickly should we produce them, and how much of our time and resources should we spend on our economy to make that happen creates a combinatorially explosive number of possible strategies. No single "First Order Optimal" strategy exists that beats all other possible strategies; a strategy that works well against one or more opponent will do poorly against one or more different opponents. Strategies may also have intransitive superiority relationships, which the game of "Rock-Scissors-Paper" best illustrates [1]. Strategy 1 (Scissors) beats Strategy 2 (Paper), but this does not mean that Strategy 1 is better overall, since Strategy 2 beats Strategy 3 (Rock), which comes back around to defeating Strategy 1. In RTS games, it may also be the case that while Strategy 1 is the best strategy to defeat Strategy 2, Strategy 1 will not win against any other strategies. Meanwhile, Strategy 2 may defeat many other opponents, making Strategy 2 more robust. Intransitive relationships between strategies makes coevolving strategies particularly difficult [2]. Additionally, a build-order's effectiveness highly depends on a game's parameters and gameplay mechanics; what works well in one game may not work in another game.

With such a large number of possible strategies we could choose from, and an equally large number of opponents we could be facing, exhaustively searching through the strategy space is infeasible. Instead, we rely on heuristic search methods to find robust strategies. Specifically, this paper focuses on comparing four case-injection methods for a coveolutionary algorithm: case injection into only the teachset, case injection into only the population, case injection into both the teachset and population, and no case injection. 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.

In our previous research, we compared the robustness of strategies found by a genetic algorithm (GA) to strategies found by a coevolutionary algorithm (CA). Our research indicates that while the GA learns strategies to defeat a set of known opponents, the GA can be misled to overspecialize to only defeat one opponent [3], [4], [5]. A CA alone learns robust strategies, but the strategies are unlikely to defeat the known set of opponents as well as a GA [4], [6]. However, injecting cases into the teachset of a CA influenced the CA to find strategies that defeat the known opponents and are more robust than strategies produced by the GA [5]. In this paper, we investigate how injecting cases into the coevolutionary teachset and/or population affects strategies found by the CA. Does the CA learn strategies that play like the injected cases? How similar are the strategies we find to the strategies we injected? Does learning from these injected cases positively or negatively impact the robustness of strategies found?

This paper analyzes the results of ten CA runs for each of our possible injection methods to answer these questions. Our results show that case injection into the population quickly biases the population towards finding robust, winning strategies that play like our injected cases. Additionally, the robustness of strategies produced from population injection was not negatively impacted, and had a robustness no less than the robustness of strategies produced without case injection. This informs our current research on finding robust RTS strategies that learn to play like previously encountered opponents.

The following section describes related work in game players using coevolution, our previous related research, and our RTS game called "WaterCraft". In Section III we describe our CA implementation, our four case-injection methods, and how strategy fitnesses are calculated. Section IV presents our results in detail, and analyzes the robustness of our strategies and their similarity to the injected cases. 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, board games were popular research platforms for coevolutionary approaches to designing game players. Chellapilla and Fogel used coevolution to tune the weights of an artificial neural network (ANN) to play Checkers [7]. Their method produced an ANN that played competitively against human opponents, and won most of the games it played. The most challenging opponent the ANN beat was a human player who was 24 points away from the "Master" level and was ranked 98th out of $80,000$ registered players. Cowling also used coevolution to tune the weights of an ANN, but trained the ANN to play "The Virus Game" [8]. The best ANN produced played the game well and won against opponents never encountered during training. Davis and Kendall used coevolution to tune the weights of an evaluation function of a player for the game "Awari" [9]. With a population size of 20, after 250 generations the best player in the population won games on three out of the four difficultly settings provided 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 [10].

More recently, as board games became easier to solve, researchers turned their attention to computer games which provide even greater challenges. Cardamone tuned the parameters of an AI player for the car racing game "TORCS" using a GA and cooperative coevoluton [11]. Cooperative coevolution produced more effective parameters than the GA. In the 2009 TORCS Endurance World Championship, the AI player using the coevolved parameters made it to 4th place, against opponents that used human expertise to tune the AI parameters AND car configurations. Avery and Louis worked on coevolving influence maps for team strategies which allow a group of entities to adapt to opponent moves [12]. Keaveney and Riordan coevolved players that coordinated their movements in an abstract RTS game [13]. They coevolved two populations of players: one population that only coevolved players on one map, and another population that coevolved players on multiple maps. Their results showed that while the players coevolved on only one map won on maps not used during training, the player's coevolved on multiple maps performed better on the same maps.

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 [14], [15], [16], [17], [18]. 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 [19], [20], [21], [22].

In previous research, case injection into a GA's population has been shown to help a GA quickly find high quality solutions [23], [24], [25], [26]. Injecting relevant cases into the population bias' the GA towards finding solutions that work well and resemble the injected cases. Other research on case injection investigates the question of when to inject cases into the population and which cases the population would benefit the most from [23], [27], [28]. While previous work on case injection focus' on injecting cases into the population of a GA, this paper looks at injecting into the population and teachset of a CA.

Our prior work on finding robust strategies in RTS games investigated the use of hill-climbers (HC), GAs and CAs. We started by exhaustively searching the performance all possible five action build-orders against three hand-tune baselines that used eight to thirteen actions. This allowed us to know which build-orders would be the best, and how common the best solutions actually were. We then used a GA and HC to find new strategies, and compared them to the best build-orders from exhaustive search [3]. We extended this study to include coevolution, first by coevolving five action build-orders to compare them to our previous results, and then coevolving thirteen action build-orders to place them on even terms with our baselines [4], [6]. We then extended our coevolutionary study to include case injection into the coevolutionary teachset, and compared the results to the build-orders found by a GA that trained against the same cases [5].

We developed WaterCraft, seen in Fig. 1, for researching evolutionary algorithms in RTS games. We wrote WaterCraft primarily in Python with some C/C++ to speed up physics, and made use of the popular Python-OGRE graphics engine for the graphical user interface (GUI) and display output [29]. WaterCraft's gameplay and unit properties were modeled around Blizzard's popular commercial RTS game "StarCraft" [30]. While WaterCraft lacks some features provided in commercial games, we have implemented the core elements of an RTS game. Two players can gather two types of resources (minerals and oil) in order to build units and structures (with several different types of units/structures to choose from), with the objective of destroying their opponent's base. The two players in the game can either be AI vs AI, Human vs AI, or Human vs Human. We made WaterCraft's GUI resemble and function like a typical RTS game GUI, so that playing the game ourselves would be intuitive. We gave WaterCraft a navel theme instead of StarCraft's science-fiction theme since our research interests also involve military and defense applications.

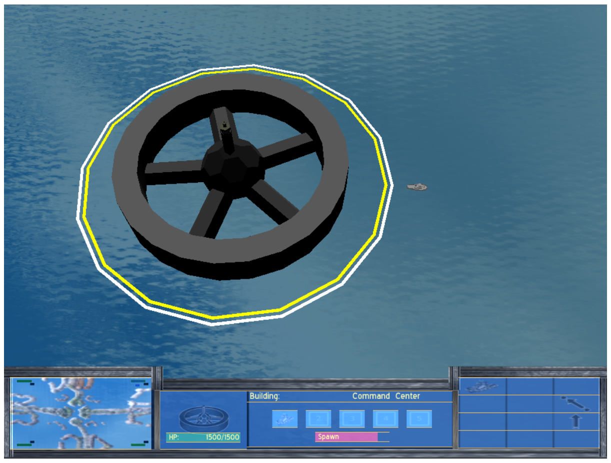

Fig. 1. A Command Center as it produces a new SCV.

In the past, projects similar to WaterCraft have been used for research involving RTS games. The BroodWar API (BWAPI) provides an interface to StarCraft that researchers use to retrieve game-state information and allows game players to interact with units in real time [31] Stratagus provides a free, cross-platform RTS engine that researchers use to create custom RTS games [32]. Wargus, which uses the Stratagus engine, is a research-oriented RTS game that uses entity data from the commercial game "Wargus II" [33]. Stargus follows in Wargus' footsteps in using the Stratagus engine, but uses the entity data from StarCraft[34]. Orts, another RTS engine, provides a graphical client and programming environment targeted towards CI and AI research. While these other projects are designed for CI and AI research in general, WaterCraft was designed with our research needs specifically in mind. In the following section, we describe how we use WaterCraft to evaluate the quality of a strategy and our methods for finding new winning strategies.

III. Methodology

WaterCraft can run either with or without graphics, depending on our needs. Disabling the graphics allows us to maximize the rate the game plays at, and thus minimize the time required to evaluate the outcome between two artificial players (game AI) competing against each other. Enabling the graphics limits the rate at which the game will execute, but allows a human player to interact with the game, issue commands and compete against other humans or game AIs in real time, or simply observe the outcome between two game AIs. If a human is issuing commands, the commands are encoded to a bit-string which represents the player's strategy, allowing us to store and use their strategy in a case base. Matches in WaterCraft are deterministic, meaning that a match between two specific strategies will always have the same outcome, regardless of if the graphics are enabled or not. This allows us to take strategies recorded from a human and quickly coevolve counter-strategies with the graphics disabled.

Our build-order strategies are encoded as a sequence of commands. We represent the strategies as a bit string, since GAs and CAs prefer binary representations because they have the best pattern-to-schema ratio [35]. We use the same binary representation for our GA, CA, and baseline strategies so we can compare strategies produced by different methods and reuse them in our case injection case base. Every three bits in a chromosome represents a command, as shown in Table I. When we send a chromosome to WaterCraft for evaluation, the game AI will sequentially decode the chromosome and insert the encoded commands into a First-In, First-Out (FIFO) queue. When the game AI decodes a command, if any prerequisites in Table I have not been previously inserted into the queue before, then the game AI will insert the missing prerequisites into the queue before the encoded command. Otherwise, if the prerequisites were inserted into the queue at least once beforehand, then the game AI will only insert the encoded command into the queue. We do not allow prerequisites to be explicitly encoded in order to keep our representation shorter, which allowed us to exhaustively search build-orders which encode more combat units. However, we also have some preliminary research which explicitly encode prerequisites and does not automatically insert missing prerequisites. These preliminary results show that a GA and CA will evolve solutions which encode the required prerequisites before the desired units.

In our previous work we limited ourselves to 15-bit (5-command) chromosomes to make comparing our results to exhaustive search feasible [3], [4]. In a later study, we increased the chromosome length to 39-bits (13-commands) to make them even with our longest baseline [6]. This means our search space has 2^{39} possible solutions. 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 select from four different units. For each of these units, we use the same

TABLE I
UNIT ENCODINGS

Bit Sequence	Command	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

strengths and costs as units of the same name in StarCraft, where the units are balanced against each other. Marines are the basic offensive unit, and while Marines are not the strongest unit, they are cheap to build, build quickly, and only require a Barracks structure to produce. Firebats are offensive units that are stronger than Marines, but cost more to produce and require at least one Academy structure in addition to the Barracks. Vultures are our strongest unit, but take the longest to produce. SCVs (Space Construction Vehicles) offer very little in terms of offensive and defensive value, but are essential for gathering resources and building structures. There are a total of six actions we can encode, which means we need at least three bits to encode them. However, since three bits encodes eight values, we arbitrarily gave two actions multiple encodings so all three-bit strings encode valid actions. Since the "Build SCV (Gather Minerals)" and "Build Firebat" each have two ways of being encoded, our CA's crossover and mutation operators are biased towards selecting them more often than other commands. However, strategies that are biased towards these commands will quickly go extinct in the population, since strategies that use their resources more effectively will have a higher shared fitness.

All game AIs follow the same rules for creating and managing their units. Game AIs sequentially issue commands from the queue in the order they were inserted, as quickly as they possible can. If the game AI fails to issue a command due to lack of resources, the game AI will wait until enough resources are gathered to issue the command, instead of skipping to an affordable command. The game AI acts "dumb" in this respect, and relies on the CA to find an effective order for the commands. When the game AI reaches an "Attack" command, all completed Marines, Firebats and Vultures are sent to attack the opponent's Command Center. SCVs remain behind to guard their own Command Center and continue gathering resources. The units sent to attack the Command Center will also attack any of the opponent's units or structures encountered along the way. If one or more units damages the Command Center, the SCVs allied with the Command Center will stop gathering resources to attack the aggressors. If the SCVs successfully destroy all units attacking the Command Center, they return to gathering resources. Player's immediately win and end the game by destroying their opponent's Command Center.

Once a game has ended, players determine how well they performed by calculating their score, which we also use as the strategy's fitness against a single opponent. We calculate the score by rewarding features we think are important in our strategies. We want strategies that spend as many resources as possible, since the most expensive units and structures tend to be the most useful in the long term. We also want strategies that destroy many of their opponent's units, thereby wasting the resources their opponent spent on those units and leaving their army in a weaker condition. Strategies that destroy their opponent's structures are truly devastating, since structures cost more resources and take longer to build than units, and are required in order to produce new units. In order to find strategies that contain these features, we calculate fitness as show in Equation 1.

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

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 coevolutionary population [36]. Every individual in the population must play against all eight of the opponents in a teachset. We populate the teachset with opponents from two different sources: four opponents from the "Hall of Fame" (HoF) and four opponents from shared sampling.

The Hall of Fame keeps a history of chromosomes that had that highest shared fitness in each generation. Adding chromosomes from HoF to the teachset prevents the population from forgetting how to defeat strategies that no longer exist 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 [36]. Shared sampling accomplishes this by seeing which chromosomes in the current population have been selected so far for the next teachset and tracking which opponents in the current teachset those chromosome defeat. The more often an opponent in the current teachset has been defeated by chromosomes selected for the next teachset, the less the remaining chromosomes in the current population have their "Sample Fitness" rewarded for defeating those opponents, making shared sampling less likely to select them for the next 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.

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 all fifty chromosomes in a population against all eight opponents in the teachset requires 400 evaluations per generation, which rises to 800 evaluations with CHC selection.

Once all individuals in the population have a fitness against all opponents in the teachset, we calculate shared fitness [36]. Usually, only chromosomes that defeat many opponents have a high fitness. However, fitness sharing gives a high shared fitness to individuals that defeat opponents few other individuals in the same population can defeat. If an individual defeats only one opponent, and no other chromosome in the same population defeats that opponent, then that individual contains important new information on how to win that we want to share with the population. We also multiply the shared fitness by the score the individual got against each opponent. If two individuals defeat the same opponents, but one of the individuals gets a higher score against at least one of the opponents, we give the individual with the better performance a higher shared 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 entire population, we use linear fitness scaling with a factor of 1.5. This prevents an exceptionally good individual from suddenly taking over the population. We even out the selection pressure by adjusting the shared fitness of each individual accordingly, so we produce more diverse children. The strongest individuals still are selected more often, but only to a certain threshold.

With the shared 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 [37]. This prevents valuable information in the parent population from being lost if our CA produces many unfit children.

B. Case Injection

We use four different case injection methods with our CA: case injection into only teachset, case injection into only the population, case injection into both the teachset and population, and no case injection. Our no case-injected method serves as a baseline to compare the influence of the other case injection methods on the coevolutionary population. We use the same case base as the source of the injected cases for all case injection methods. The case base contains five strategies from our previous work that were either hand-tuned or evolved. Some of the selected strategies were robust and defeated many possible opponents, while the other selected strategies were specialized and defeated only a few of the "best" robust strategies we had previously evolved.

Teachset case injection randomly selects two individuals from the case base every generation and injects them into the teachset. The two injected cases replace the last chromosome selected from shared selection and one randomly selected HoF chromosome, keeping the teachset size to eight. This influences the CA to prefer strategies that defeat the injected cases, while the remainder of the teachset keeps the strategies robust and prevents them from overspecializing for defeating the injected cases.

Population case injection randomly selects two individuals from the case base every five generations, which replace the two lowest shared fitness chromosomes in the population. This influences the population to play like the injected cases, while the teachset continues to encourage the strategies to remain robust. We limit ourselves to injecting only two cases to prevent the CA from being overly biased towards the injected cases, and narrowing the search space explored by the CA too

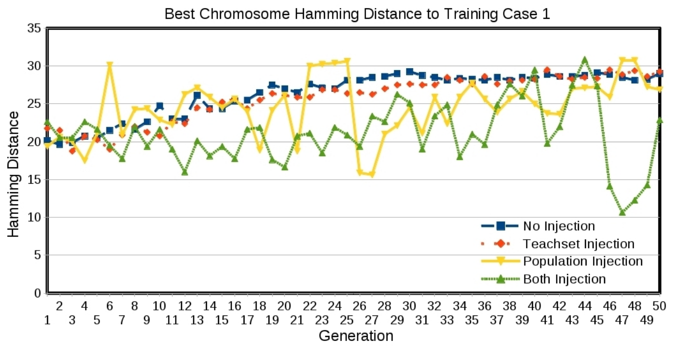

Fig. 2. Avg. Hamming Distance of best chromosome to Training Case #1.

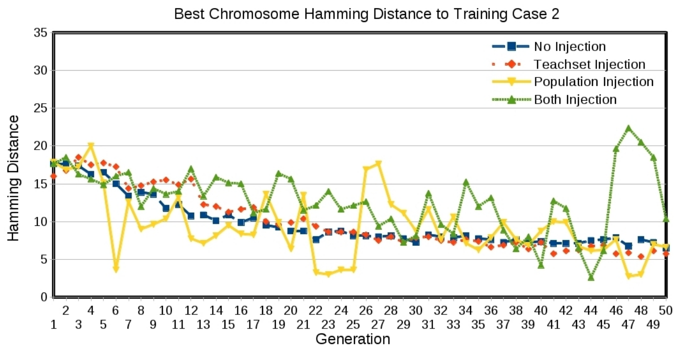

Fig. 3. Avg. Hamming Distance of best chromosome to Training Case #2.

quickly. Opponents in the teachset change as coevolution finds new strategies, which means the effectiveness of an injected solution changes depending on when we inject the case. We do periodic case injection every five generations to allow the injected cases a chance to demonstrate their effectiveness against different solutions in the teachset.

When we inject into both the population and teachset, we use both of the above methods without any modifications, since they do not interfere with each other. We selected the frequency and number of injections based on what worked experimentally well, but future work may look at finding optimal values.

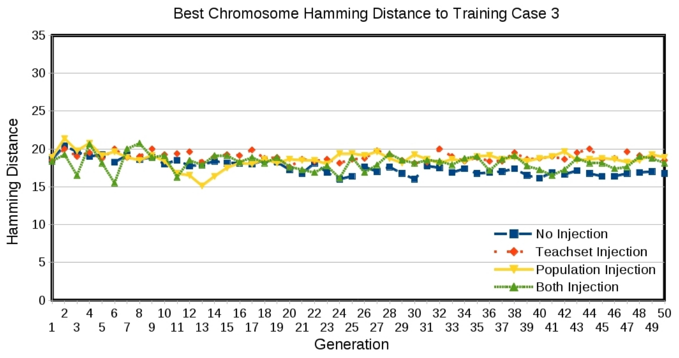

Fig. 4. Avg. Hamming Distance of best chromosome to Training Case #3.

IV. Results

For our current work, we ran each of our four injection cases ten times with a population size of 50 for 50 generations with a chromosome length of 39-bits(13-commands). We measure the robustness of the strategies produced by comparing them to five test cases that are never seen during training. These test cases are different from the cases used for case injection, but were also produced from hand-tuning or coevolution in our previous studys and were either robust or defeated our best evolved strategies. Measuring the robustness of every chromosome in every population would take excessively long, so we limit ourselves to testing the “best” chromosome (the chromosome with the highest shared fitness) in the population for every generation. We measure robustness by examining the average score and number of wins against all five testing cases. We also measure the Hamming Distance of the best chromosome to each of the five injected cases [38]. We compare the all bits of two chromosomes, and for each bit-position if the bit-value in one chromosome does not match the bit-value in the other chromosome, we increase the Hamming Distance by one. This means that the higher the Hamming Distance between two chromosomes, the less similar they are to each other. Conversely, the lower the Hamming Distance between between two chromosomes, the more similar they are to each other. This tells us how much case injection has influenced the population to play like the injected cases. Finally, our results are calculated by taking the average of the best chromosome for each generation across all ten runs.

Our results show that the influence of case injection on a population varies based on the cases and injection methods. Due to space limitations, we limit ourselves to showing the similarity to only three of the five injected cases. We select these three cases because they exemplified the results of the remaining two. Our figures show that without case injection, that the best chromosomes tends to become less similar to Training Case 1 (Fig. 2), more similar to Training Case 2 (Fig. 3), and have little change in Hamming Distance to Training Case 3 (Fig. 4). Injecting cases into only the teachset has little to no effect on influencing the best chromosome to play like the injected cases.you All three figures show that the best chromosomes produced from teachset injection have almost the same Hamming Distance as chromosomes produced without case injection. When we inject chromosomes into only the population or into both the population and teachset, the results vary a bit more. Fig. 2 shows that after we inject our first chromosomes into the population at generation 5, over time the best chromosomes tend to become more similar to Training Case 1 than the chromosomes produced without case injection. On the other hand, Fig. 3 shows the opposite effect. While the best cases produced without case injection tend to become more similar to Training Case 2, the best cases produced from population injection tend to become less similar over time. However, when we look at Training Case 1 and Training Case 2, we see that with a Hamming Distance of 32 (out of a maximum of 39) from each other these two cases are dissimilar. As one of these cases begins to influence the

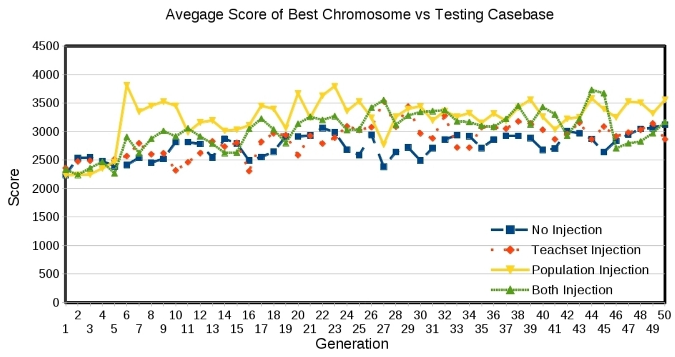

Fig. 5. Avg. Score of best chromosome versus all Testing Cases.

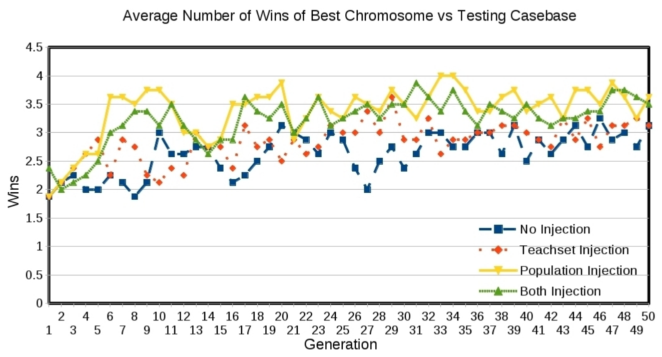

Fig. 6. Avg. # of Wins of best chromosome versus all Testing Cases.

population to play in a similar manner, then naturally that means the population begins to play less like the other case. We see happening in Fig. 2 and Fig. 3 , since the results in each almost perfectly mirror each other. Training Case 1 builds two SCVs, ten Marine, then attacks, while Training Case 2 builds ten Vultures instead of Marines. The coevolved solutions that are similar to Training Case 1 replaces three of the Marines with an additional two SCVs at the beginning and a Firebat towards the end, allowing for an earlier attack that is successful against opponents that produce units slowly. Coevolved solutions similar to Training Case 2 replaced one SCV and the attack command with a Firebat and additional Vulture, allowing for a stronger defense force that takes longer to produce. Finally, while Training Case 1 influences the best cases to play in a similar manner, Fig. 4 shows this does not affect the best cases' similarity to Training Case 3, which has a Hamming Distance of 19 to Training Cases 1 and 2.

While our CA manages to learn new strategies from injecting cases into the population, we also want our coevolved strategies to defeat a wide variety of opponents. We test the robustness of the strategies the CA produces by playing them against five chromosomes we had previously hand-tuned or coevolved and that were never seen during training. Our results show that while case injection into the population increases the score against unknown opponents in early generations, strategies produced in the long term score only slightly higher than our other injection methods, as shown in Fig. 5. Fig. 6 also shows that over time our coevolved strategies win against more of the opponents. While Fig. 2 shows us that population injection influences our best chromosomes to play like some of the injected cases, Fig. 5 and Fig. 6 show us that our best solutions continue to be at least as robust as coevolution without case injection.

V. Conclusion and Future Work

In this paper we want to find robust, winning strategies for Real-Time Strategy games. We also want these strategies to incorporate knowledge from winning strategies other players have used. We believe we can accomplish these goals by using case injection with a coevolutionary algorithm. Case injection takes strategies contained in a case base, and injects them into our coevolutionary algorithm's population or teachset. There are four case injection methods that we examine: case injection into only the population, case injection into only the teachset, case injection into both the population and teachset, and no case injection. We used each of these methods ten times with a population size of 50 for 50 generations, and took the average across all ten runs for the best chromosome at each generation. When we do case injection into the teachset, every generation we select two random strategies from our case base and put them in the teachset. With case injection into the population, every five generations two random strategies are selected from our case base to replace the two lowest shared fitness chromosomes in the population. There are five winning strategies contained in our case base that were produced from hand-tuning or coevolution.

Our results show that while injecting into only the teachset does not affect the similarity to the injected cases, injecting cases into the population had different effects. We measured the similarity by calculating the Hamming Distance from chromosomes in the population to each of the five cases in our case base [38]. Some injected cases affected the population more than others, influencing the chromosomes to play like some injected cases while also influencing to play unlike other injected cases. For other cases, there seems to be no change in Hamming Distance to the population. This shows that a coevolutionary population can be influenced to play like injected cases, but some injected cases may have more influence than others. However, we do not want chromosomes that learn to play like someone else, to also inherit their flaws and lose robustness.

We tested the robustness our results by playing the best chromosomes against five chromosomes never seen during training. These testing chromosomes are different from the chromosomes used for case injection, but were also produced from hand-tuning and coevolution. Our results showed that although injecting cases into the population influenced the best chromosomes in the population to play like the injected cases, these new chromosomes did not lose robustness, and defeated at least as many opponents as strategies produced from coevolution without case injection.

These results indicate case injection into a coevolutionary population will influence coevolution to quickly produce similar winning strategies, while maintaining robustness. This informs our research into adapting new strategies from previously encountered opponents. In our future research, we would like to investigate new methods for determining which cases in a case base would be the most beneficial to inject into a coevolutionary population and/or teachset and at what time/frequency we should do the injection. We also intend to research methods for examining an opponent's actions during a game, comparing them to cases in our case base, and determining if our game AI should change strategies to something else in our case base. Finally, in the long-term we plan to maintain a case base of strategies used by human players and coevolution, and inject these cases into a perpetually running coevolutionary algorithm that finds new effective strategies as more people play.

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, 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] ——, "Finding robust strategies to defeat specific opponents using case-injected coevolution," in *Computational Intelligence in Games (CIG), 2013 IEEE Conference on*. IEEE, 2013, pp. 1–8.

[6] ——, "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.

[7] 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.

[8] 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.

[9] 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.

[10] 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.

[11] 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.

[12] 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

[13] 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.

[14] 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.

[15] 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.

[16] 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

[17] 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

[18] 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.

[19] 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.

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

[21] 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.

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

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

[24] 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

[25] 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.

[26] 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.

[27] P. M. Avery and Z. Michalewicz, "Static experts and dynamic enemies in coevolutionary games," in *Evolutionary Computation, 2007. CEC 2007. IEEE Congress on*. IEEE, 2007, pp. 4035–4042.

[28] M. N. Collazo, C. Cotta, and A. J. Fernández-Leiva, "Virtual player design using self-learning via competitive coevolutionary algorithms," *Natural Computing*, pp. 1–14.

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

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

[31] "BWAPI: An api for interacting with StarCraft: Broodwar." [Online]. Available: http://code.google.com/p/bwapi/

[32] 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.

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

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

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

[36] 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

[37] 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/

[38] R. W. Hamming, "Error detecting and error correcting codes," *Bell System technical journal*, vol. 29, no. 2, pp. 147–160, 1950.