Comparing Coevolution, Genetic Algorithms, and Hill-Climbers for Finding Real-Time Strategy Game Plans

Christopher Ballinger
University of Nevada, Reno
Reno, Nevada 89503
caballinger@cse.unr.edu

Sushil Louis
University of Nevada, Reno
Reno, Nevada 89503
sushil@cse.unr.edu

ABSTRACT

This paper evaluates a co-evolutionary genetic algorithm's performance at generating competitive strategies in the initial stages of real-time strategy games. Specifically, we evaluate co-evolution's performance against an exhaustive search of all possible build orders. Three hand coded strategies outside this exhaustive list provide a quantitative baseline for comparison with other strategy search algorithms. Earlier work had shown that a bit-setting hill-climber only finds the best strategies six percent of the time but takes significantly less time compared to a genetic algorithm that routinely finds the best strategies. Our results here show that co-evolved strategies win or tie against hill-climber and genetic algorithm strategies eighty percent of the time but routinely lose to the three hand coded baselines. This work informs our research on improving coevolutionary approaches to real-time strategy game player design.

Categories and Subject Descriptors

I.2.m [**Artificial Intelligence**]: Evolutionary Computation; I.2.1 [**Applications and Expert Systems**]: Games

General Terms

Algorithms

Keywords

Coevolution, Real-Time Strategy Games, Build-orders

1. INTRODUCTION

Finding effective and robust strategies in Real-Time Strategy (RTS) games presents a challenging problem. RTS game players must compete for resources, build up an economy that is able to support their military force, expand their control over the map, and eventually destroy their opponent's base. Any advances in developing RTS game players will impact planning and execution in competitive industrial settings through the development of smart, realistic opponents.

Several problems make finding effective strategies complicated. Many types of units are available, each with their own costs and dependencies. For example, in StarCraft, building a *marine* requires 50 minerals and takes 25 seconds to build. However, before you can start building *marines* you must build at least one *barracks*, which requires 150 minerals and takes 80 seconds to build. Choosing which units to build and the order in which to build them leads to a combinatorially explosive number of possible strategies. An exhaustive search through the strategy space is generally not feasible and in this paper, we compare coevolutionary algorithm (CA) generate strategies aginst hand-coded baselines and solutions produced by a genetic algorithm (GA) and hill-climber (HC) from prior work [4].

Our overall goal is to create competent RTS game players and this paper compare CAs to GAs and HCs. However, several issues make such comparisons difficult. First, there is no single optimal strategy. RTS games are designed like rock-paper-scissors games and for every possible good strategy there is a counter strategy that can beat it. This means that it is not trivial to compare two strategies because strategy S_1 beating S_2 does not imply that S_1 is better than S_2 since S_1 could be beaten by S_3 which could be beatable by S_2, as shown by Figure 1. Furthermore, S_2 could beat a larger number of strategies and thus be more robust than S_1.

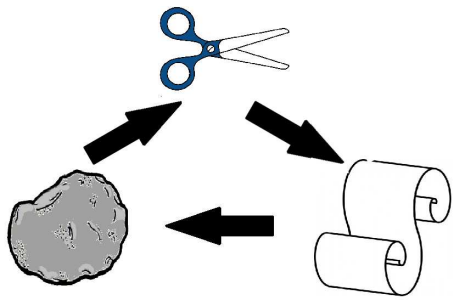

Figure 1: Three strategies that defeat each other.

Once we recognize and overcome the difficulties in comparing strategies, we want to see how well co-evolution does compared to HCs and GAs in searching strategy spaces. Does CA find good strategies, strategies that beat one or more of our three baselines? The GA and HC search to find strategies that beat our three hand-coded baselines. How

GECCO'13, July 6-10, 2013, Amsterdam, The Netherlands.

do the CA found strategies perform against GA and HC generated strategies? Are CAs more robust than GAs or HCs? In order to make such comparisons more meaningful, we also need to look at the space of all possible strategies, that is, we need to exhaustively list all possible strategies and evaluate their performance against our three baselines. We therefore (as in our prior work) restrict our search space enough to make it tractable to exhaustively list and evaluate all strategies on our 400 core cluster. We can then compare CAs against GAs, HCs, and with the results of exhaustive search to answer questions of solution quality, robustness, and search time complexity.

This paper answers the questions posed in the previous paragraph with respect genetic algorithms and a bit setting hill-climber. In addition, we also describe and qualitatively analyze the properties of strategies found by the CA. Our results show that the CA teach set solutions beat GA generated and HC generated solutions 80% of the the time but fare much worse against the three hand code baselines. Since the GA generated solutions do relatively well against the baselines, the CA is tending to find solutions that beat the GA but not the baselines and forms one of the corners of the triangle in Figure 1 above with the GA and baselines making up the other two corners. These results inform our current research in developing a full game AI for WaterCraft, our open-source RTS clone of StarCraft.

The next section describes related work in generating game players with coevolutionary techniques and RTS simulation environments. In Section 3 we describe our representation and the details of our CA implementation. Section 4 presents preliminary results and shows a comparison of solutions produced by our methods. Finally, the last section provides conclusions and discusses possible future work.

2. RELATED WORK

Much work has been done in coevolutionary approaches to designing players for various games, particularly for board games. Chellapilla and Fogel used coevolution to tune the weights of an artificial neural network (ANN) to play Checkers [7]. The best resulting ANN competed against human players and was able to win most games, including a player who was 27 points away from the "Master" level and ranked as 98th out of over 80,0000 registered users on a Checkers website. Cowling used coevolution to tun an ANN to play "The Virus Game", which were able to win against opponent never seen during training [8]. Davis and Kendall coevolved the weights of an evaluation function for the game "Awari" [9]. Using a population of 20 chromosomes over 250 generations, Davis and Kendall coevolved a player that could win three out of the four difficultly settings in the commercial Awari game "Awale". Nitschke used competitive coevolution in a pursuit-evasion game to evolve pursuer-entities that cooperated with each other to capture one of the evader-entities [14].

In addition to board games, computer games without discrete game-states have recently become to target of coevolution research. Cardamone used cooperative coevolution to optimize the parameters of a race car for a specific AI in the racing simulator TORCS [6]. Coevolution increased the cars performance more than a GA, and the final result placed 4th in the 2009 TORCS Endurance World Championship, against AIs and car configurations that were turned with human expertise. Avery and Louis also did work on coevolving team tactics using influence maps, allowing a group of entities to adapt to opponent moves [3]. Keaveney and Riordan used an abstract RTS game to coevolve spatial tactical coordination [12]. They coevolved two populations, a population that only played on one map and a population that played on multiple maps. While the population that only played was not overly specialized to win on that one map, the population that played on multiple maps performed much better.

Our prior work in this domain investigated GAs and HCs for finding robust strategies for RTS games [4]. In that study, we used the same three hand-coded baseline strategies to evaluate the fitness of individuals found by a GA and HC, and used exhaustive search to rank how good the strategies were overall. We will go over these results briefly in Section 4.

We developed WaterCraft 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 [13]. We modeled the game play in WaterCraft around the popular commercial game, StarCraft [10]. 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 player over the network.

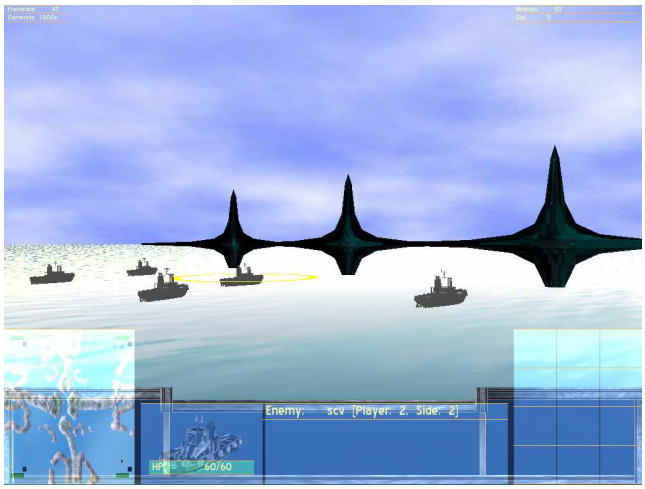

Figure 2: Several SCVs gathering additional resources.

Other similar projects have been used in the past for research in RTS games. The Brood-War API (BWAPI) allows developers to retrieve information about the game state and interact with units in StarCraft [1]. Stratagus provides a free, cross-platform RTS engine that has been used directly in several research projects [15]. WARGUS uses Stratagus as the back-end for game play, but uses the entity data from the commercial game WarCraft II [17]. The same thing has been done with STARGUS, which uses the entity data from StarCraft [2]. ORTS, another RTS engine, provides a total programming environment for computational and ar-

tificial intelligence research, including a graphical client [5]. While other projects do not target a specific method, we designed Watercraft specifically for evolutionary computing approaches. In the next section, we describe how we configured Watercraft for our research as well as our search methods.

3. METHODOLOGY

We made several changes to the default Watercraft configuration. Since we evaluate the fitness of an individual by decoding the individual into a build-order and simulating the result in Watercraft, we need to run Watercraft for every evaluation. With thousands of evaluations necessary, we turned off the graphics display and GUI during evaluation to simplify our simulations and reduce the amount of wall clock time a game takes to run. By disabling the OGRE graphics engine we greatly reduced the amount of processing time required to run a simulation. The ability to easily turn off the graphics makes WaterCraft especially suited for evolutionary computing approaches.

In this preliminary research we worked with only four types of units and the buildings needed for their production. Marines can be built quickly and cheaply, with barracks being the only infrastructure required beforehand, but with low offensive capabilities. Firebats are stronger than Marines, but require more infrastructure and cost more to produce. Vultures are the strongest unit, but cost the most and take the longest to produce. SCVs are used for gathering resources and building new structures, but have little offensive or defensive value. When building a structure, the SCV must move to the build location of the structure, and becomes unavailable until the structure is complete. When gathering resources, the SCV must move to the source of the resource, gather a small portion, and deliver the resource to a Command Center. Once the SCV delivers the resource to the Command Center, the resource is added to a bank that players use to pay for additional units and structures. When we initialize a game, each player starts with five SCVs and a Command Center. We place these units near sources of additional resources to decrease the amount of time spent gathering and delivering resources.

We created three hand-tuned "baseline" build-orders to compare against potential solutions generated by the CA, GA and HC. These baselines take 30 bits to encode, compared to the 15 bit solutions we develop in covolution. This allows the baselines to present a challenging opponent that coevolution will have never encountered before, in contrast to the 15 bit solutions produced by the GA and HC, which the CA may still independently discover. We designed the first baseline to attack quickly with a small force. The baseline builds three additional SCVs for gathering minerals, followed by constructing five Marines, then attacking the player. This strategy challenges the player by quickly building enough units to attack before the player builds much. The second baseline does the same, but builds ten Marines instead of five. While much slower than the first strategy, the second baseline builds a much harder force to overcome if left uninterrupted. The third baseline builds two SCVs for gathering minerals, three SCVs for gathering gas, five Vultures, then attacks. This build-order also takes a long time to complete, but focuses on building fewer units that are individually stronger.

In our earlier work we found that the strategies found by the GA and HC were focused entirely on defense. This trend was due to being evaluated against the above baselines, which were too overwhelmingly powerful for any attack to succeed. Instead, the strategies produced by the GA and HC build the most powerful and expensive units possible before the baseline units arrive to attack the base. Against the baselines that attack quickly, the solutions tended to build a one or two SCVs, followed by Marines and Firebats. Against the slower baselines, solutions tended to build Firebats and a few Vultures, with no extra SCVs to help get resources quicker. Building the Firebats and Vultures takes longer, but provides a better defense and maximizes the amount of resources spent.

Table 1: 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

Players typically determine how well they played by using the score at the end of a game. In the context of our search methods, the score also acts as a strategy's fitness. Our fitness calculation encourages players to find ways to destroy enemy units and structures, as shown in Equation 1. Where F_{ij} is the fitness of individual i against individual j, SR_i is the amount of resources spent by individual i, UD_j is the set of units owned by individual j that were destroyed, UC_k is the cost to build unit k, BD_j is the set of buildings owned by individual j that were destroyed, and BC_k is the cost to build building k.

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

We represented a build-order as a sequence of commands. Since GAs prefer a binary encoding, the CA, GA and the HC all worked with the same binary encoded sequence of commands - using GA terminology, this is our chromosome. We encoded the chromosome as a 15-length binary string. Every three bits of the binary string encodes an action, as shown in Table 1. When WaterCraft receives a chromosome, the game component that deals with the artificial player (Game AI or just AI), sequentially decodes the binary string and inserts a request for each encoded action into a queue. While decoding the binary string, the game AI verifies that all the prerequisites shown in Table 1 for the next encoded action were previously inserted into the queue. If a prerequisite was not inserted into the queue already, the decoder will insert a request for the prerequisite immediately before the encoded action. Our game AI will attempt to issue actions from the queue as quickly as possible. When the AI fails to execute the current action because of a lack of resources or pending prerequisite being built, the AI waits a short duration and reattempts to execute the action until the action succeeds.

3.1 Coevolution

When evaluating an individual for genetic operations such as selection, we used a teach set, scaled fitness, hall of fame, shared sampling, and fitness sharing as described by Rosin and Belew [16]. Usually, only chromosomes that can defeat many teach set members have a high fitness. However, shared fitness also gives chromosomes that only defeat a few teach set members few other chromosomes defeat, a high fitness. Chromosomes that defeat teach set members few others can, probably contain new and important innovations for winning. Giving these unique chromosomes a higher fitness prevents diverse niches from going extinct too quickly. We also multiply the shared fitness by the score the chromosome received against each defeated opponent. This allows us to identify chromosomes that not only win, but perform significantly better in the game. We calculated fitness sharing as shown in Equation 2. Where f_i^{shared} is the shared fitness of chromosome i, D_i is the set of teach set members chromosome i defeated, j a teach set member in D_i, j_l is the number of times j lost against all chromosomes, and F_{ij} is the fitness of chromosome i against teach set member j.

$$f_i^{shared} = \sum_{j \in D_i} \frac{1}{j_l} F_{ij} \tag{2}$$

j_l is the total number of individuals that baseline j lost to in the current population. We calculate j_l using Equation 3, 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 baseline j lost against individual i, and returns 0 if baseline j won against individual i, as show by Equation 4. Since Equation 2 only takes the sum of baselines that were defeated j_l can never be 0, since if a baseline j was never defeated j would not be in set D_i.

$$j_l = \sum_{i \in P} GameResult(j, i) \tag{3}$$

$$GameResult(j, i) = \left\{ \begin{array}{ll} 0, & F_{ji} >= F_{ij} \\ 1, & F_{ji} < F_{ij} \end{array} \right\} \tag{4}$$

When all population members have a shared fitness, we use linear fitness scaling with a factor of 1.5. At times there may be a population member who's shared fitness dwarfs all the other chromosome shared fitnessâĂŹs, or where all population members have shared fitnessâĂŹs that are very close. Fitness scaling helps even out selection pressure, making the children produced more diverse.

The teach set that every chromosome plays against contains eight chromosomes selected from two sources: four chromosomes from the hall of fame and four chromosomes from shared sampling. We limited our teach set size to eight, so that with our population size of 50, our entire cluster of 400 nodes would be full. The hall of fame is simply a list of chromosomes that have performed well in the past. At the end of each generation, the chromosome with the highest shared fitness is added to the hall of fame. Adding chromosomes from the hall of fame to the teach set prevents solutions in the current population from forgetting how to beat opponents from previous generations that may have gone extinct.

Shared sampling is a method that selects a opponents that offer diverse challenges. Shared sampling works by selecting chromosomes that beat the most teach set members, but giving less weight to teach set members already defeated by chromosomes previously selected by shared sampling, as shown in Algorithm 1. The shared fitness is given by Equation 5 where s_i is the sample fitness, D_i is the set of teach set members chromosome i defeated, j_b is the number of time j has been defeated by chromosomes selected by shared sampling, and F_{ji} is the fitness of teach set member j against chromosome i. Shared fitness is recalculated each time a chromosome is sampled, so that chromosomes that defeat teach set members not defeated by the currently sampled chromosomes are given higher preference. This allows us to maintain a diverse teach set, while keeping the number of opponents needed to a minimum.

Algorithm 1 Shared Sampling

```
unsampled = current population
sampled = empty list
while size(sampled) < samples wanted do
  for all s ∈ unsampled do
    calculate s_i
  end for
  best = s ∈ unsampled with highest s_i
  unsampled.remove(best)
  sampled.append(best)
  for all j ∈ best.defeated do
    j_b += 1
  end for
end while
```

$$s_i = \sum_{j \in D_i} \frac{1}{1 + j_b} F_{ji} \tag{5}$$

For the basic CA operations and parameters, we used uniform crossover with a 95% chance of crossover occurring and bit-mutation with a .1% chance of each individual bit changing value. The chromosomes are selected for crossover by using standard roulette wheel selection. Once we have created all the children chromosomes, we evaluate them against the same teach set, then use CHC selection to select chromosomes for the next population [11]. This prevents valuable information from being lost if our CA produces many unfit children.

3.2 Genetic Algorithm

Our GA implementation was simple. We modified the concept of the teach set we used for coevolution; instead of creating a new teach set each generation, we populate the teach set solely with our baseline strategies. Other than the change to the teach set, we used all the same parameters and methods that we used for coevolution.

3.3 Hill-climber

We use the bit-setting optimization HC shown in Algorithm 2, which attempts to find an effective solution by sequentially flipping each bit and keeping the value with the highest fitness.

We determine the fitness by playing a chromosome against all three baselines and taking the sum of the differences in scores, as shown in Equation 6 [18]. Where f_i is the fitness of chromosome i, j is a baseline in the set of all baselines B, F_{ij} is the fitness chromosome i received against baseline j, and F_{ji} is the fitness baseline j received against chromosome

Algorithm 2 Bit Setting Optimization Hill-climber

```
chromosome = initialize()
select first bit
evaluate(chromosome)
while not end of chromosome do
  flip current bit
  evaluate(chromosome)
  if fitness decreased then
    flip current bit back
  end if
  select next bit
end while
```

i.

$$f_i = \sum_{j \in B} F_{ij} - F_{ji} \quad (6)$$

HC performance depends on the initial seed. We initialize this HC with thirty-two different seeds: a chromosome set to all 0's, a chromosome set to all 1's, and thirty randomly generated chromosomes.

3.4 Exhaustive Search

Exhaustive search evaluates all 2^{15} possible build-orders against all three of our baselines. We limited ourselves to 15-bit solutions since that was the maximum build-order length we could exhaustively search in a reasonable amount of time. Exhaustive search enables us to rank all possible 15-bit solutions, and compare the effectiveness of solutions against the baselines found by CAs, GAs, HCs, as well as other search methods in the future.

In our previous work, we used a GA and HC to produce solutions that could beat our baselines. The baselines we used posed varying levels of difficultly and majority of possible 15 bit solution lose to all three baselines. These prior results showed that GA's always found the best strategies that could defeat two baselines (no strategies in the solution space could defeat all three, as shown in Figure 3), while the HC did note perform as well, as shown in Figure 4.

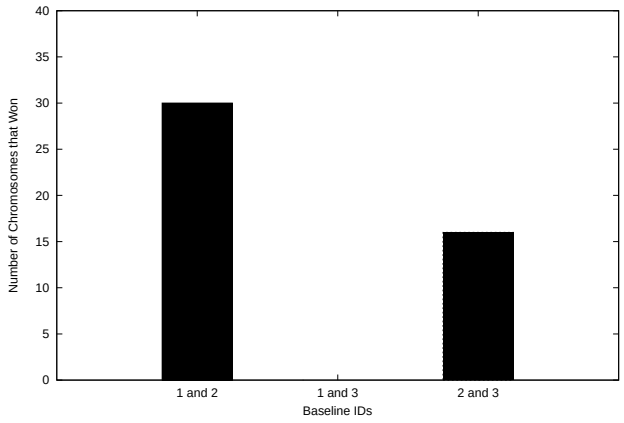

Figure 3: The number of strategies from exhaustive search that defeat only two baselines, and the baselines they defeat.

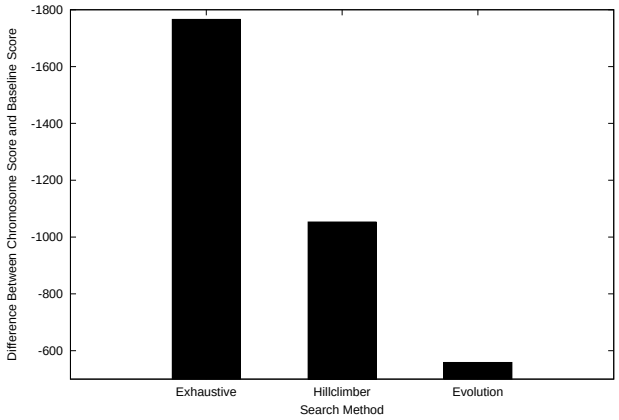

Figure 4: The average difference in score against the baselines our previous search methods.

4. RESULTS

We ran coevolution with a population size of 50 on our 400 node cluster for 98 generations, in order to match the number of generations performed by our GA. We measured the progress coevolution makes by playing all members of the population and teach set at each generation against three sets of opponents, taken from our previous study: three of the best solutions produced by the GA, 32 solutions produced by the HC, and the three baselines used to evaluate the GA and HC solutions.

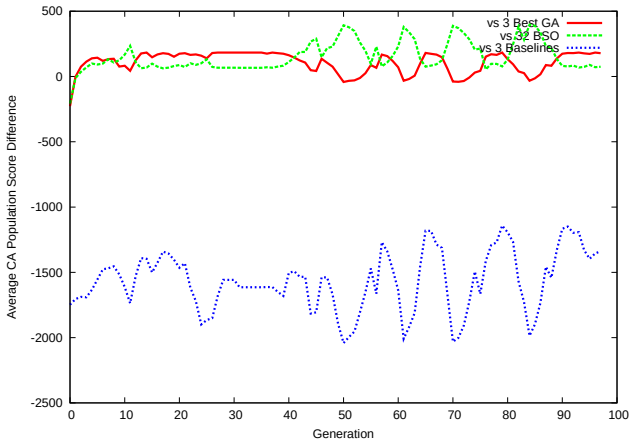

Figure 5: Average Score Difference of Coevolutionary Population.

Figure 5 shows that coevolution very quickly moves to increase the average score of the population, but not by very much. However, this slight increase in average score has a huge affect on the number of wins the solutions achieve, as shown by Figure 6. During the first ten generations, the increase in score leads to an increased number of wins against the GA and HC, but a decreased number of wins against the baselines. In later generations, an increase in score correlates to an increase in wins for the GA and baselines, but a decrease in wins for the HC. This pattern also holds true for

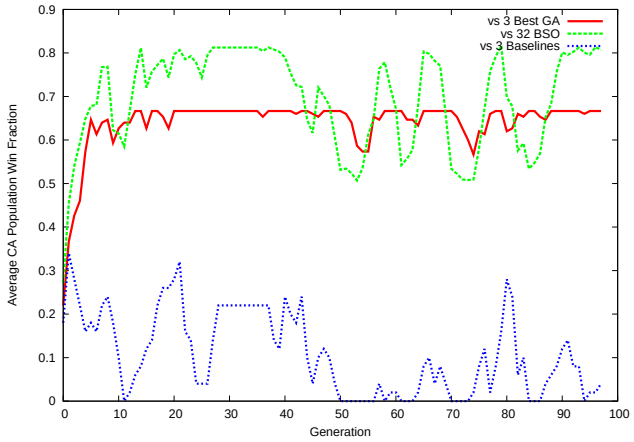

Figure 6: Average Number of Wins of Coevolutionary Population.

the number of ties, as shown in Figure 7. All of these figures show that a cyclic pattern emerges around generation 40.

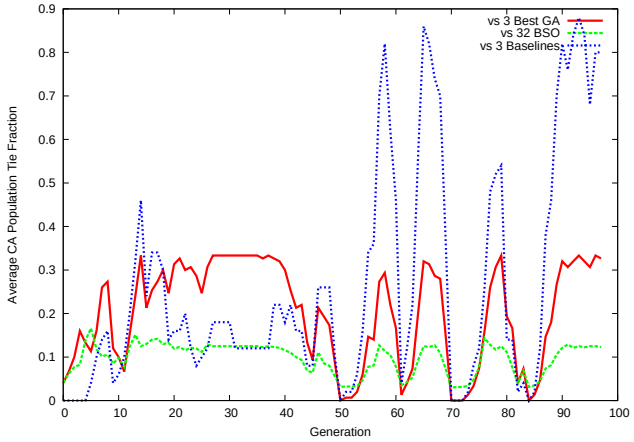

Figure 7: Average Number of Ties of Coevolutionary Population.

While Figure 5 and Figure 6 shows that our coevolved solutions are not as good against the baselines as the GA results, all three figures seem to indicate that an increase/decrease in performance against the GA and HC solutions correlates to an increase/decrease in performance against the baselines.

The CA teachset average show progress similar to the CA population average when compared to the same three sets. However, the cyclic pattern and correlation between the increasing/decrease scores, wins, and ties are no longer as pronounced, as shown in Figure 8, Figure 9, and Figure 10 respectively. These features are most likely muted, due to half the teachset being constructed from the hall of fame. Chromosomes injected from the hall of fame may be very old solutions that were not very effective in the long run, and heavily influences our averages.

Finally, Figure 11 shows how well on average the CA, GA, HC, and exhaustive search performed against the all three baselines. While the CA was usually able to do better than the average of exhaustive search, the CA does not perform

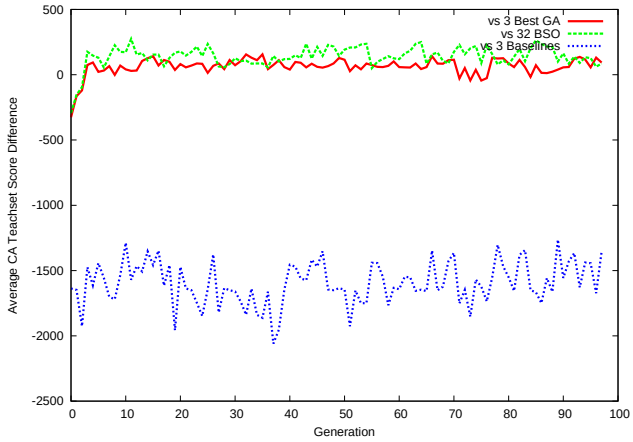

Figure 8: Average Score Difference of Coevolutionary Teachset.

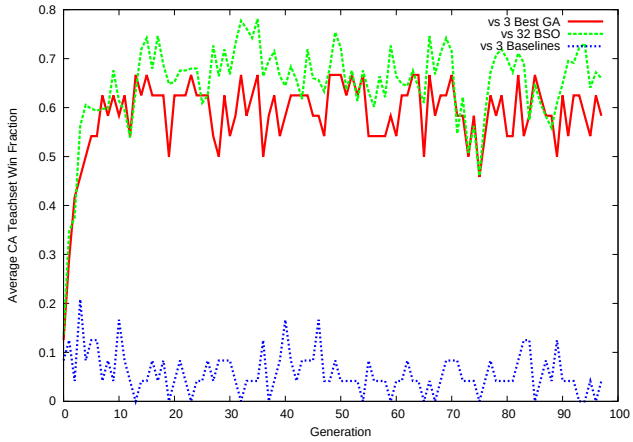

Figure 9: Average Number of Wins of Coevolutionary Teachset.

as well as the GA or HC solutions, which were tuned using the baselines.

Analysis of the populations at generations with high average scores showed that the favored strategy was to build mostly Vultures followed by a one or two Firebats. One of the solutions found by the GA was the opposite of this, preferring to build mostly Firebats followed by the Vultures. This two strategies cost the same to construct, however the strategy found by coevolution provides a stronger defense in exchange for taking longer to complete.

Populations at generations with low average scores had similar strategies, but issued an attack command as the final action. Attacking with multiple Vultures and Firebats can inflict heavy losses, before the attacking units are destroyed by the defending units. However, against opponents that are faster to attack or build up an equally strong defense, such as with our baselines, GA produced solutions and HC produced solutions, the attacking force is wiped out to quickly to benefit.

5. CONCLUSION AND FUTURE WORK

This paper compares co-evolutinary genetic algorithsm

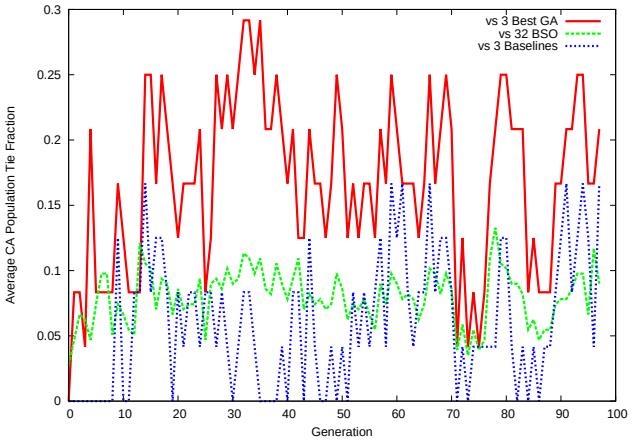

Figure 10: Average Number of Ties of Coevolutionary Teachset.

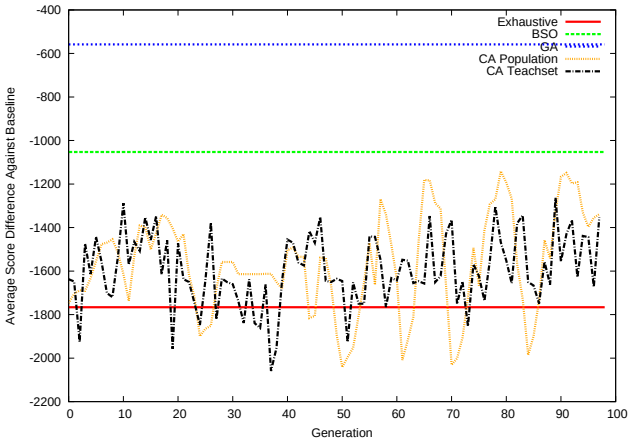

Figure 11: Average Score Difference Against Baselines.

aith with genetic algorithms and hill-climbers to generate strategies for beating opponents in RTS games. We used our three hand-coded baselines and solutions produced by a GA and HC from a previous study upon which to make our comparisons. The strategies produced by the GA and HC were tuned specifically to defeat the three hand-coded baselines. Since the baselines are encoded with a longer bitstring than the chromsomes used in coevolution, we can be assured that coevolution will not have encountered the baselines before.

Our results show that coevolution can find strategies that defeat or tie the GA and HC solutions approximately 80% of the time, showing that the coevolutionary strategies can defeat solutions previously shown to be robust and capable of defeating challenging baselines. When compared directly against those same baselines, our coevolutionary strategies increased their performance over time, and defeated or tied the baselines only 20% of the time without being trained against the baselines beforehand. Clearly, our game exhibits the rock-paper-scissors balance. While our results show coevolved solutions can beat challenging 15 bit opponents, we do not yet know how well these solutions would rank in an exhaustive list of all 2^{15} strategies played against all 2^{15} strategies.

These results help specify the trade-offs between injecting human expertise into our evolutionary algorithms. Using a GA against a set of hand-tuned opponents produces strategies that are robust against those opponents. However, our CA uses a different method to produce a teach set without human expertise. Solutions found by the CA could beat the GA solutions, but as is the case with Rock-Scissors-Paper, this advantage did not translate to defeating the same opponents that the GA solutions could defeat.

We are interested in finding human competitive RTS game players that are robust, specifically that they are successful against many different opponents. This paper's results indicate that while progress towards previously unseen opponents is possible, there is much room to improve the overall performance. Given the similarity of our GA and CA, injecting our hand-tuned baselines into the teachset occasionally may be one method of improving the robustness of our solutions. In our future work, we will focus on finding methods that help increase coevolution's capability of finding robust strategies.

6. ACKNOWLEDGMENTS

This research is supported by ONR grant N00014-09-1-1121.

7. REFERENCES

[1] Bwapi: An api for interacting with starcraft: Broodwar.

[2] Stargus, 2009.

[3] 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*, GECCO '10, pages 783–790, New York, NY, USA, 2010. ACM.

[4] 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.

[5] M. Buro. Orts - a free software rts game engine, 2005.

[6] 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*, pages 1 –8, july 2010.

[7] K. Chellapilla and D. Fogel. Evolving an expert checkers playing program without using human expertise. *Evolutionary Computation, IEEE Transactions on*, 5(4):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*, pages 45 –51, may 2006.

[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*, volume 2, pages 1408 –1413, 2002.

[10] B. Entertainment. Starcraft, March 1998.

[11] L. J. Eshelman. The chc adaptive search algorithm : How to have safe search when engaging in

nontraditional genetic recombination. *Foundations of Genetic Algorithms*, pages 265–283, 1991.

[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*, pages 371 –378, sept. 2009.

[13] T. K. S. Ltd. Ogre ï£¡ open source 3d graphics engine, February 2005.

[14] 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*, volume 2, pages 2037 – 2042 vol.2, oct. 2003.

[15] M. J. V. Ponsen, S. Lee-urban, H. Muï£¡oz-avila, D. W. Aha, and M. Molineaux. Stratagus: An open-source game engine for research in real-time strategy games. Technical report, Naval Research Laboratory, Navy Center for, 2005.

[16] C. D. Rosin and R. K. Belew. New methods for competitive coevolution. *Evol. Comput.*, 5(1):1–29, Mar. 1997.

[17] T. W. Team. Wargus, 2011.

[18] S. W. Wilson, S. W. W. Ga-easy, and S. W. W. Ga-easy. Ga-easy doe not imply steepest-ascent optimizable, 1991.