Using Coevolution to Understand and Validate Game Balance in Continuous Games

Ryan Leigh
University of Nevada, Reno
Reno, Nevada, United States
leigh@cse.unr.edu

Justin Schonfeld
University of Nevada, Reno
Reno, Nevada, United States
schonfju@cse.unr.edu

Sushil J. Louis
University of Nevada, Reno
Reno, Nevada, United States
sushil@cse.unr.edu

ABSTRACT

We attack the problem of game balancing by using a coevolutionary algorithm to explore the space of possible game strategies and counter strategies. We define balanced games as games which have no single dominating strategy. Balanced games are more fun and provide a more interesting strategy space for players to explore. However, proving that a game is balanced mathematically may not be possible and industry commonly uses extensive and expensive human testing to balance games. We show how a coevolutionary algorithm can be used to test game balance and use the publicly available continuous state, capture-the-flag CaST game as our testbed. Our results show that we can use coevolution to highlight game imbalances in CaST and provide intuition towards balancing this game. This aids in eliminating dominating strategies, thus making the game more interesting as players must constantly adapt to opponent strategies.

Categories and Subject Descriptors

I.2.1 [**Artificial Intelligence**]: Applications and Expert Systems —*Games*

General Terms

Design

Keywords

Games, Game Balance, Coevolution, Artificial Intelligence

1. INTRODUCTION

Game balance plays a crucial role in game design and development. A balanced game is one in which there are many paths to victory; there is no single dominating strategy. Balanced games provide many benefits. Players have an array of options to choose from, and as a result the game is more interesting and fun. In multi-player games, balance is especially important. The ability for all players to compete on a fair playing field is one of the factors which contributes to making such games enjoyable. Balanced games provide a challenging domain for studying artificially intelligent agents. These domains have no domaining strategy, so players can find counter strategies that beat agents that use non-adaptive strategies.

Unfortunately, determining whether or not a game is balanced requires completely exploring the game's strategy space. For a simple game like Tic-Tac-Toe, this is a relatively straightforward computational task, but for more complex games such as Chess and Go it is nearly impossible to investigate all possible strategies. As a result balancing a game is often a challenging and difficult process requiring a great deal of time and money. This paper demonstrates the effectiveness of coevolution as a tool for understanding and testing game balance.

1.1 Game Balancing

Balanced games are more fun to play. If there is a single dominating strategy, games become a race to implement the dominant strategy first and every match becomes predictable. Worse yet, players may constantly lose against such a strategy with no chance of winning. A balanced game that provides counter strategies means that players are always able to adapt to their opponents and a chance at victory is always possible. If each game is dynamic and the game feels "fair", then they will tend to play the game more than if the game fails to exhibit these qualities [15].

Balanced games also serve as interesting platforms for AI research. Developing an effective artificial agent for such games is challenging because players will often find a counter strategy to static, non-adaptive agent. Even if the agent's programmer encodes many strategies, players will inevitably locate holes in the agent's AI to exploit.

Determining if a game is balanced can be difficult. Sometimes a game is simple enough to make the determination straightforward. For example, Tic-Tac-Toe can be easily analyzed to determine that it is in fact unbalanced. The first player can always win or force a tie. As a game gets more complex, the challenge in determining whether or not the game is balanced also increases.

Discrete games can have large search spaces to find a series of moves that leads to a winning condition. Chess needed a computer like Deep Blue to compete with grandmasters. Analyzing these search spaces for game inbalances is complicated due to their size. If we move from discrete to continuous state games, building a search tree becomes impractical. Searching for the correct sequence of moves in a continu-

ous space requires that the space be discretized resulting in a loss of fidelity. Mathematically proving continuous state games are balanced is often difficult due to the complicated non-linear interactions between their rules.

Currently, games are balanced by hand tuning, but this approach presents several problems. Humans players are expensive in both time and resources, and even human players cannot explore all strategies. It takes time to play games and a human can only explore a relatively small facet of a game each time they play. Commercial games undergo extensive player testing before they are released partly due to game balance issues. In this paper, we explore game balancing with respect to a simple game environment, looking towards making the game a useful platform for designing adaptive agents in the future.

Adaptive Agents

Balanced games provide a platform for researching adaptive agents. We define an adaptive agent to be agent that observes an opponent's play, analyzes it, and responds accordingly. An adaptive agent has four stages: First, the agent "observers" the opponent. Second, the agent builds a model of the opponent. Third, the agent trains against the model in a separate simulation. Fourth, in the next match, the agent uses its new strategy or strategies against the opponent.

Things to note about this procedure: "Observing" can either be done egocentrically or globally, either with limited information or full information. The opponent does not have to be human, but could be another agent. Lastly, the separate training simulation can either be run offline or online while the main game is progressing.

Adaptive agents have many uses. For players who do not or cannot play games multiplayer, the adaptive agent can provide a challenging opponent. This can even apply to single player games. As the player progresses in the game and learns the agent's tricks, the agent also learns the player's tricks. This can lead to entirely new behavior from the agent during the next game. Additionally, an adaptive agent can meet the needs of players of different skill levels by choosing strategies the player will do better against [3, 4].

We are developing our adaptive agents for real-time continuous state games for three reasons. First, many high-selling and highly-rated commercial games these days are real-time continuous state games. Fifteen of the top twenty rated games of 2007 are real-time continuous state games [7], and of the top ten selling games of 2007 half are real-time continuous state games [11]. Second, we have experience working in the domain of these games [12, 13]. Third, real-time continuous state games have not been explored for research as much as turn-based discrete games [6, 14, 1].

To aid us in developing our adaptive agents, we impose two guidelines on the game we are designing. First, the game must be balanced so that the adaptive agent can formulate a counter strategy. If the game is not balanced, then the player can find a strategy that the agent cannot beat and the purpose of adaptive agents have been defeated. Second, the player's strategies must be able to be determined based on analysis of the player's actions. Analyzing a player's strategies directly from their actions can be difficult because different strategies can share similar moves and no moves may directly correspond to a single strategy. By ensuring that certain actions correspond to certain strategies, then those strategies can be easily observed. This ensures that we can build a model of the player for later training.

1.2 Using coevolution to evaluate game balance

Coevolutionary algorithms are a subset of genetic algorithms which optimize problems with coupled fitness functions such as two player games[9, 8, 10]. In two population coevolution, two populations of players are repeatedly compared and evolved against one another. Each generation, one population tests the other population and successful individuals may pass their successful strategies to the next generation. The two populations form an "arms race" [5], constantly changing and improving to beat the other population.

Evolutionary algorithms tend to converge towards strategies that win all the time. Coevolution has the advantage of being able to play games faster than humans and searching the strategy space implicitly in parallel [10]. As we will demonstrate experimentally for our game CaST, and likely for many similar games, if dominating strategies exist in the game, coevolution will likely find them and converge. If the game is balanced with proper counter strategies, then the two populations will never stop adapting to each other.

Even if a game is unbalanced and there exists an ultimate strategy coevolutionary search is not guaranteed to locate it. Deficiencies in the agent representation may prevent the search from testing all of the strategies available to a human player. Although the portion of the strategy space coevolution can explore is limited by the representation, within that rather large space, coevolution can be used to locate game imbalances. The remainder of the strategy space can be explored via human testing.

In section 2, we describe the CaST game, the strategy space, and the agent artificial intelligence used. We discuss coevolution, how it works, and its relationship to game balancing in section 3. Section 4 and 5 describe our experimental design and results respectively. Lastly, we give our conclusions and future work in sections 6 and 7.

2. THE GAME

2.1 The Initial CaST Game Rules

CaST is a two player, real-time action game where the goal is to be the first to score 50 points. CaST is short for "Capture, Steal, Territory". If neither player scores 50 points in a 5 minute time period, then the player with the most points wins. Each player controls a single ship equipped with a tractor beam.

We developed CaST for two reasons. First, it is a good game for testing game balancing. It is complex enough to require balancing due to its continuous nature but simple enough to find a representation of its rules for use in a GA. Second, it is a good test platform for developing agents because it is easy to tell which strategy they are using when deployed.

2.1.1 Scoring Points

There are two ways a player can score points: 1) by capturing a crate and taking it to their base, or 2) by controlling more territory than their opponent. Points that are acquired through taking a crate to a base are counted as *crate points*.

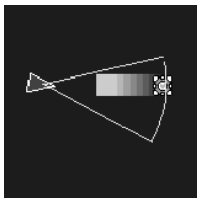

Figure 1: A ship is capturing a crate. The trail extending from the crate represents progress in capturing. When the trail reaches the ship, the crate becomes captured.

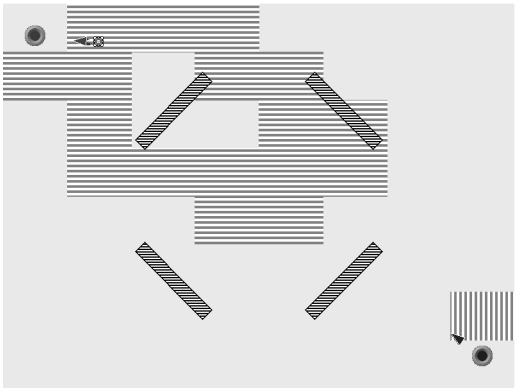

Figure 2: The red player is about to capture the crate upon returning to his base. Along the way, he has flipped several tiles to his color.

Points that are acquired through controlling territory are counted as *territory points*.

To capture an unclaimed crate, the player must use their tractor beam on the crate for a short period of time ($\frac{1}{2}$ second). As long as the crate lies in the arc described by the tractor beam capturing progress is made (Figure 1). Capturing progress is indicated by a yellow line extending from the crate to the player's ship. A captured crate can be *stolen* by the opposing player if they focus their tractor beam on the crate for a period of 1 second. After capturing a crate, a player can score points by returning to their base with the crate in tow. Once it has been taken to a base the crate then reappears at it's starting position. If a player is towing a crate then they are unable to take control of tiles.

Scoring points through territory requires that a player controls more tiles than their opponent. A tile is controlled by the last player to have physically occupied the tile. Periodically, the game awards points to the player with the most tiles based on the following equation:

$$points = \Delta Tiles \cdot \Delta Time \cdot C \quad (1)$$

where $\Delta Tiles$ is the difference in tiles between players, $\Delta Time$ is the time since the last check, and C is a constant representing the rate of tiles per second. Initially, C had a value of $\frac{1}{100}$.

Ship Physics

The ships move according to a simple physics model. A ship's speed changes each tick by $t \times accelConst \times \Delta Time \times s/s_{max}$, where t is a throttle set from -1 to 1, s is the current speed, s_{max} is the max speed, and *accelCost* is an acceleration constant. A ships rotation changes each tick by $r \times turnConstTime \times s/s_{max}$, where r is a rudder set from -1 to 1 and *turnConst* is a turning constant. The tractor beam can be used to capture unclaimed crates and to steal crates from the opponent.

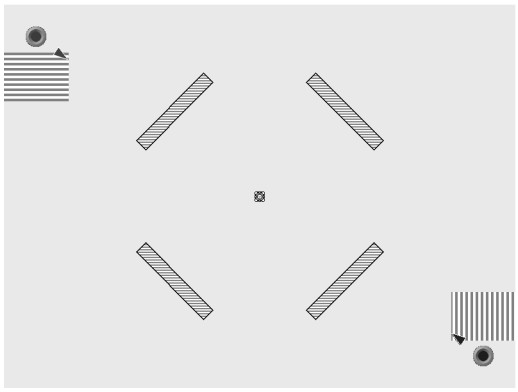

Figure 3: This is CaST at the start of a match. Each player starts in the corner near his base. The flag is in the middle. The thing diagonal rectangles are obstacles that players must drive around.

The Game Board

CaST is played in a two-dimensional arena of 800 by 600 pixels surrounded by walls (see Figure 3). Four additional walls are placed in the center of the arena blocking access to the crate. The map is divided evenly into 64 tiles, called *territory tiles*. Each territory tile can be in one of three states: controlled by player 1, controlled by player 2, or uncontrolled. At the center of the board is a crate. At the upper left corner of the board is player 1's base and in the lower right corner is player 2's base.

2.2 The Strategy Space

Since one of the goals of **CaST** is to serve as a platform for researching opponent modeling it needs a well defined strategy space. **CaST** was designed to have three core strategies, each of which cancels out one of the others.

Crate Running The player takes the quickest route to the crate, grabs it, and returns it to their base. This process is repeated as many times as possible.

Grab Territory The player tries to convert as many tiles to their side as they can. Priority is given to converting territories controlled by their opponent.

Stealing Crates The player waits around the crate until his opponent takes the crate. Then the player steals the crate and returns it to their own base.

Each of these strategies beats one of the other core strategies and loses to the third, much like Rock-Paper-Scissors. **Crate Running** beats **Grab Territory** as it will earn more points at the end of a match than **Grab Territory**. **Grab Territory** beats **Stealing Crates** because grabbing territory generates points that **Stealing Crates** cannot block. **Stealing Crates** beats **Crate Running** because stealing crates prevents the opponent from taking the crate to their

base and scoring. The nature of these core strategies ensures that for any strategy a player chooses, there is another strategy to counter it. Also, each strategy should tie when played against itself, as their point outputs will be the same.

2.3 Agent AI: Representing the Strategies

Each player is represented as a state-machine with three states, one for each core strategy. Each state will run for a set amount of time after which a new state will be chosen. The runtime for each state is determined by a parameter specific to that state. Each state has a probability of getting chosen such that $P_{crate} + P_{grab} + P_{steal} = 1$. Once a state is chosen, it is queried every 0.5 seconds to determine the next action. Ship navigation is handled by a combination of A* and a low-level reactive navigator to avoid their opponent.

The A* algorithm works on an adjacency map generated by dividing the game arena into a 16 by 16 grid. Each grid intersection is a vertex in the adjacency map and the edges connect neighboring vertices including diagonals. Each edge is tested to see that it does not intersect with fixed obstacles in the map. If an edge does intersect with an obstacle, it is removed from the map.

Each point in the path that A* finds is passed into the low-level navigator. First, it finds the heading to the desired point (dh) and sweeps across an array of angles. The length of the array is $2 \times sweepWindow$ and stores the vote of each angle as a possible candidate direction. The value of each vote (v) for each angle (a) is based on the following equation:

$$v(a) = \begin{cases} \frac{C \times turnBias \times (1 - |dh - a|)}{2 \times sweepWindow} & \text{if } a < dh \\ \frac{C(1 - |dh - a|)}{2 \times sweepWindow} & \text{if } a >= dh \end{cases} \quad (2)$$

where C is a scaling constant and $turnBias$ applies a scalar to angles left of the desired heading. This value, if not 1.0, gives a preference to move right or left depending on whether or not $turnBias$ is less than or greater than 1.0 respectively. Second, the navigator then does coarse ray traces around the perimeter of the ship within a certain distance ($distThresh$). If a ray trace intersects with land, angles in the voting array around that ray trace angle receive a penalty of $p(a) = distToLand/distThresh$. The ship then turns towards the angle with the highest value of $v(a) \times p(a)$.

2.4 Evolving the Agent

Genotype: The genotype is comprised of 11 distinct parameters and is stored as a bit string of length 72. Each state is represented by two parameters: the probability of transitioning to the state from either of the other states, and the amount of time to stay in the state. The last five parameters tune the low-level navigator.

Phenotype: The first set of decoded parameters determines the probabilities for transitioning from one state to another. The three parameters are summed and a random number is chosen between 0 and the total. The parameter that corresponds to the random number is the next state. Each parameter is 6 bits long and decodes to a value between 0 and 63.

The second set of parameters determine how long each state runs. Each parameter is 6 bits long and decodes to an amount of time ranging from $\frac{1}{8}$th of a second to 8 seconds in increments of an $\frac{1}{8}$th of a second.

The last set of parameters tune the low-level navigator as shown in Table 1.

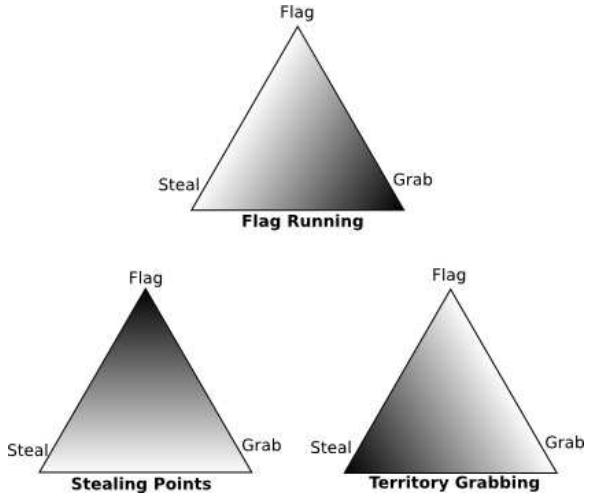

Figure 4: The simplex for each strategy shows wins against the space of mixed strategies with dark grays meaning more wins.

A single player's actions during a run of the game can be described as the amount of time spent in each of the three states. Mixed strategies such as grabbing the crate and taking an indirect path to the base in order to flip additional tiles can expressed as time spent with **Crate Run** and time spent with **Territory Grab**. The agent representation can replicate this time breakdown through the use of appropriate parameters. Although players can employ more subtlety when using a mixed strategy than the agent, the agent can still mimic those decisions with parameters that result in a similar time breakdown. Fidelity may be lost, but anything the player does can be mapped into the space of possible strategies for analysis and replication.

3. BALANCING WITH COEVOLUTION

Ideally, any core strategy will beat one of the other core strategies and lose to the third. In the space of mixed strategies, the strength of a core strategy will weaken as the level of the strategy it is good against in the mixed strategy lessens. This balance is depicted in Figure 4 where each triangle is a simplex graph. A simplex is a three-dimensional point where the sum of the components equals one. Black points in the graph correspond to high levels of success against that mixed strategy. The ability for a core strategy to win against a mixed strategy lessens as it moves away from the corner of the core strategy it does well against. The line between the core strategy and the dominating strategy is all white, because any mixed strategy on the line will result in either a tie or a loss. Realistically, unless the rules are set just right, these gradients will not arise. We use coevolution to test if the game is balanced and to suggest which mixed strategies are too weak or too strong.

3.1 Testing the Game

Competitive coevolution creates an arms race between two populations, where each population is trying to beat the other [16]. Through the course of a coevolutionary run, the strategy space is sampled and searched many times faster than human players can do by themselves. In our experi-

Table 1: Parameters Used by the GA for Obstacle Avoidance

Name	Bits	Values	Description
sweepWindow	7	$0-127$	Numbers of angles to sweep left and right
voteScalar (C)	8	$0-2.56$	Scalar C used in Eq. 2
turnBias	7	$-0.64-0.63$	Give preference to turn left of right
distThresh	9	$200-711$	Range to search for land
landSweepRes	5	$1-32$	Angle distance between land raytrace

ment, each population takes the role of one of the two players. In a game with good balancing, if one population converges on one strategy, then the other population will find an appropriate counter strategy. The first population will then need to change its strategy to beat the second population again. This will form cycles in the population, as they both constantly try to out do each other. For example, if the first population converges on the **Crate Running** strategy, then the second population will have to move to the **Territory Grab** strategy. To counter this move, the first population must then adopt the **Steal Crates** strategy. This cycle will continue until the end of the coevolutionary run.

If the game is not properly balanced, then many things may happen. One strategy may be weak, causing the populations to cycle inside a smaller area of the search space. One strategy may dominate all the other player's strategies, causing one population to converge and the other to drift around looking for a counter strategy. By looking at the patterns of the populations as they change over time, weaknesses or strengths can be discovered providing insight on changes to bring game balance.

3.2 Coevolutionary Algorithm

We used a competitive two population coevolutionary algorithm to evolve and test game playing agents for **CaST**. The algorithm we used is a variation of the one described by Rosin and Belew ([16]). An outline of the algorithm is provided below:

```
Initialize two populations
Designate populations as hosts and parasites
do
    Choose individuals from parasites to test hosts
    Evaluate hosts against the parasite sampling
    Save best individual of hosts to hall of fame
    Copy best of last hosts to new population
    Create children for remainder of new population
    Exchange roles of two populations
while(Ending condition not met)
```

This algorithm used shared sampling when choosing parasites, fitness sharing, and a hall of fame. Shared sampling ensured that the small set of parasites chosen maximally test the host population. The shared sampling procedure chooses parasites that performed well against individuals that the other parasites in the current sample did not perform well against.

Fitness sharing rewards individuals that competed well against a parasite that other individuals did not compete well against. This ensures that standout individuals will receive higher fitness for wins against hard parasites than an individual with an equal number of wins against easier parasites. Fitness sharing also encourages niching, allowing for multiple counter strategies in one population. The fitness calculation also included a second component designed to minimize the number of collisions the ships were involved in. As the simulation progressed, we logged all collisions of the host individual that last longer than a half second. We ignored shorter collisions as this game is quick paced and short collisions are often unavoidable. The number of collisions between 0 and 100 are normalized from 1.0 to 0 and is saved for each of the individuals matches. If the number of collisions is over a hundred, we set the value to 0. After fitness sharing is calculated for each individual, we scale this fitness by the average of these normalized collision counts. This way, we prefer individuals with high fitness that crash the least.

Lastly, the hall of fame ensures that individuals not only perform well against the current parasite population, but also against good parasites from earlier generations in the coevolutionary run.

Coevolution Parameters

Each run of the coevolutionary algorithm was done with a population size of 30 for 100 generations. For testing the host population, three parasites were chosen using shared sampling and three members of the hall of fame were chosen at random. For reproduction, we used roulette selection, 1-point crossover with a probability of 70%, single bitwise mutation applied to each bit with a probability of 5%, and the top two members of the last generation were passed to the next generation.

4. EXPERIMENTAL DESIGN

We propose a game balancing process consisting of three stages:

1. Coevolve two populations of game playing agents
2. Analyze the dynamics of the coevolutionary process
3. If the game is unbalanced, tune the game rules and parameters

Although we repeated this process more than five times we will only show the analysis for five key iterations of the game balancing process: the initial game, three midway through the game balancing process, and the final balanced game. For each iteration, we describe the rules used, show graphs (see below) resulting from those rules.

4.1 Analysis

During each coevolutionary iteration the algorithm was run five times. The combined results of all five runs were displayed as a heat map combined with a simplex graph. The simplex heat map is our primary form of visualization and takes a simplex-value mapping and displays it in a range of grayscale values. We generate our graphs in three phases. First, the parameters for state selection are converted into

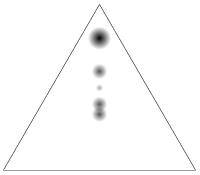

Figure 5: A simplex heat map with example data points. Each data point from top to bottom is max fitness, $\frac{2}{3}$ max fitness, $\frac{1}{3}$ max fitness, and lastly two overlapping points each at $\frac{2}{3}$ max fitness.

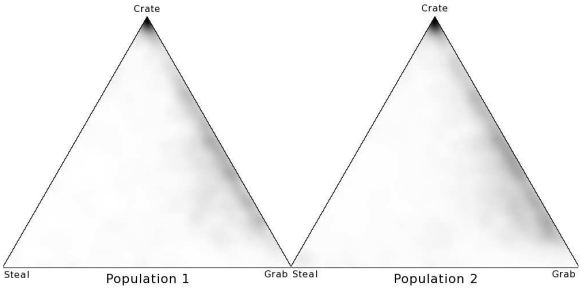

Figure 6: The simplex heat maps for both populations of all five coevolved runs under the initial game rules. Dark spots indicate that those portions of the strategy space were better represented in the runs. Stealing Crates is the strongly preferred strategy.

simplex values. Second, we weight the simplex values by the time each state runs. For example, if each state has an equal chance of running, but **Crate Running** and **Territory Grab** run for one second and **Steal Points** runs for two seconds, then overall **Steal Points** will run for half of the simulation and the other two states will run a quarter of the time each. In other words, this value is the predicted fraction of time that each behavior will run. The third phase converts the simplex values into two dimensional coordinates. The value for the simplex point equals the normalized fitness times a scalar (30 for the graphs we present). This value becomes an impact radius and all points within a distance of this radius receives an value increase equal to the impact radius minus the distance between points. Finally, the whole graph is normalized such that the highest value is black and the lowest white and then graphed.

Figure 5 shows a graph with a few sample points: one at max fitness, one at $\frac{2}{3}$ max fitness, one at $\frac{1}{3}$ max fitness, and lastly two overlapping points at $\frac{2}{3}$ max fitness each. These graphs aim to show where high fitness individuals tend to converge. Convergence looks like dark black spots while a lack of convergence would be a cloudy gray.

5. GAME BALANCING

5.1 Initial Configuration

In the initial configuration, capturing an unclaimed crate took $\frac{1}{2}$ second, stealing the crate took 1 second, returning the crate earned 5 points and holding territory earned $\frac{1}{100}$ points per difference in tiles per second. The heat map produced by these parameters are shown in Figure 6.

Figure 6 shows that individuals tended to converge on

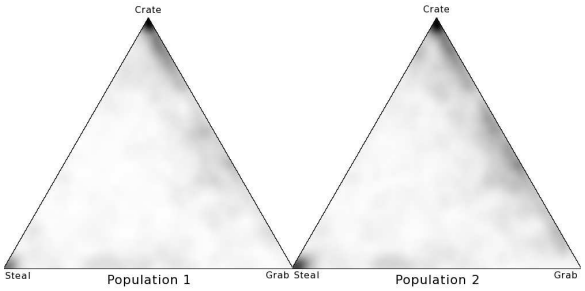

Figure 7: Simplex heat map of the first iteration. Notice that stealing is now highly effective as well.

Crate Running heavy strategies but also spent time along the **Crate Running** and **Territory Grab** edge. The figure also shows that individuals using the **Crate Stealing** strategy were not heavily represented in the population. This suggested that either the evolved agent was unable to effectively implement stealing or that stealing was unbalanced.

5.2 Iteration 1

In response to the results of the initial coevolutionary analysis we altered the difficulty of the stealing strategy by reducing the time it took to steal a crate. The time required to steal a crate was changed to 0.75 seconds from 1 second. The results of the change are shown in Figure 7. As can be seen in the figure, reducing the time caused **Stealing Crates** to become a preferred strategy along with **Crate Running**. The edge between **Crate Running** and **Territory Grab** continued to show some activity but **Territory Grab** itself remains relatively unrepresented.

5.3 Iteration 2

In order to address the weakness of the **Territory Grab** strategy the constant C was increased from $\frac{1}{100}$ to $\frac{1}{75}$. Figure 8 shows that this small change caused both populations to almost completely converge on **Stealing Crates**. Since the only change in the rules was to change the effectiveness of **Territory Grab**, the convergence on **Crate Stealing** was nonintuitive. Further investigation into the actual behavior of these strategies, however, revealed a flaw in the rules which could be exploited by the agent or player. Agents in the stealing state followed their opponents until there was a crate to steal. This meant that if the opponent was currently grabbing territory the agent would simply follow the opponent around and take control of every tile right after their opponent did, making stealing effective against both **Crate Running** and **Territory Grab**. Inadvertently, the alterations had turned **Crate Stealing** into an unbeatable strategy. In order to correct this imbalance the stealing mechanic was redesigned so that it was ineffective against **Territory Grab**.

5.4 Iteration 3

In order to rebalance **CaST** the ability to steal a crate currently held by the other player was removed, and the ability to steal *crate points* was added. In this third iteration, instead of stealing crates from their opponents, player had the ability to steal up to 10 points from their opponent's base by focusing the tractor beam on it. The number 10 was selected because a player using the steal strategy has to move

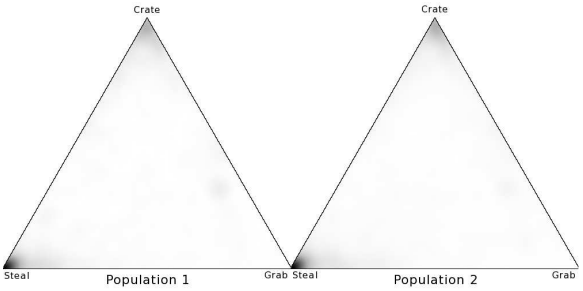

Figure 8: Simplex heat map of the second iteration. In all five runs both populations converged on Stealing Crates as the most effective strategy.

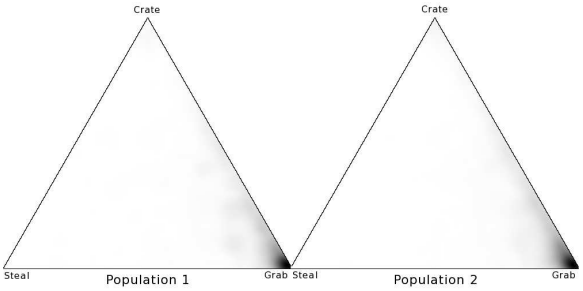

Figure 9: Simplex heat map of the third iteration. The coevolved populations were dominated primarily by Territory Grab.

twice as far as a player implementing **Crate Running**. In order to complete the stealing of points, the player has to the place their tractor beam on their own base. This change in the steal mechanic reduced the effectiveness of steal against **Territory Grab**, but maintained its effectiveness against **Crate Running**.

The results of the new mechanic are shown in Figure 9. As a result of the rule change **Territory Grab** took over as the most effective strategy. Since the **Territory Grab** constant had been increased in an earlier iteration we decreased it for the next iteration.

5.5 Iteration 4

The **Territory Grab** constant was reduced from $\frac{1}{75}$ to $\frac{1}{200}$. Figure 10 shows that the results of these changes are much more promising. The individuals still spend the majority of their time on territory grabbing strategies but they also implement more mixed strategies than ever before. Strategies appear to become more ineffective as stealing becomes more involved. Since **Territory Grab** now seems to be functioning well, rather than altering C any further, the other two strategies are strengthened.

5.6 Iteration 5

To improve crate running and stealing strategies, the point values for each strategy were doubled. Taking a crate to a base earned 10 points and a player could steal up to 20 points at one time. Figure 11 shows that as a result of this final round of changes the game appears to be well balanced with respect to the evolvable agent. Individuals are spread all over the space. They still cluster a bit around **Territory Grab** and not much around **Stealing Points**, but this is acceptable when considering the performance of the game as a whole.

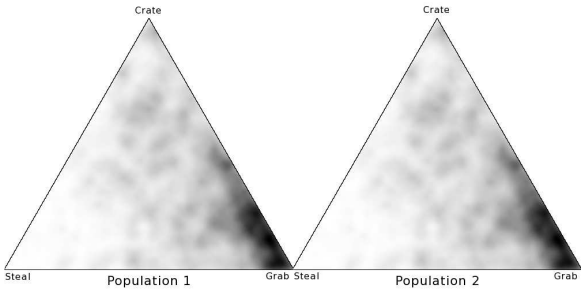

Figure 10: Simplex heat map of the fourth iteration. For the first time mixed strategies started to appear on the map.

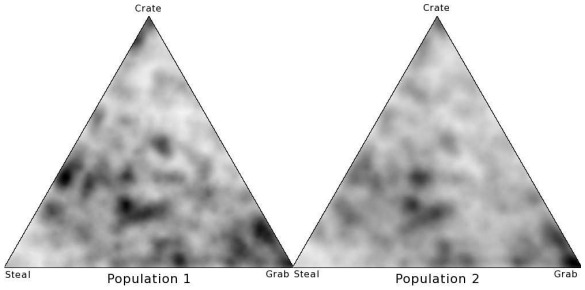

Figure 11: Simplex heat map of the fifth iteration with the final set of rule changes. The populations are fairly diverse over the strategy space.

6. CONCLUSIONS

By combining coevolution, an evolvable agent, and a clever visualization technique we were able to effectively balance the game of **CaST**. After each round of coevolution, the simplex heat map displayed not only whether or not the populations had converged, but if they had, which phenotype they had converged to.

6.1 Discussion

There are two main challenges to generalizing this balancing procedure for other games: selecting an appropriate representation and developing an effective visualization technique. Picking the correct representation is quite difficult. The representation needs to be easy to code and evolve; yet at the same time it needs to be capable of searching the same set of strategies as a human player. Each representation needs to be weighed in terms of the portion of the strategy space it can represent as well as how easy it is to search the strategy space with that representation. Neural nets, for example, when combined with a neuroevolution technique can represent a larger subset of the strategy space than the representation used here. On the other hand, the simpler representation when combined with the right agent can search the same sorts of strategies that humans use much more quickly.

The other main challenge in using coevolution to balance a game is in choosing the right visualization technique. **CaST**

has only three core strategies, so the simplex can effectively display a useful portion of the strategy space. For more complex games such as Chess, Checkers, Go, or Supreme Commander this visualization would be insufficient. While the lack of a visualization wouldn't prevent the coevolution from detecting game imbalances, it would make it more challenging to identify the source of the imbalance. It should be noted, however, that even with the simplex heat map visualization, choosing the appropriate changes to make to the game rules can be difficult.

In response to Figure 7, for example, the constant C was increased. The simplex head map suggested that making **Territory Grab** stronger was the best option, when in fact it was stealing that needed changing. The visualization was effective, however, when it came to determing how close the game was to being balanced.

7. FUTURE WORK

Automating the Balancing Procedure

In the current implementation a developer has to hand code any changes to the game between each iteration of coevolution. In order to automate the game balancing process we could tie the coevolutionary algorithm to a GA evolving a parameterized version of the game rules. The convergence time of the coevolutionary algorithm could be used as the fitness function for the GA. The faster the coevolution converges the less balanced the game, and the lower the fitness of the game parameters. While fully automated this approach would be limited by the ability to parameterize the game rules.

Human Testing

Human testing will eventually figure in to any game balancing process. With current AI techniques, it is unlikely we will be able to perfectly capture just what a human player's strategy is, and we may not be able to completely test the game space. Humans can be very good at search, taking into account many factors that may escape a fitness function, and can find unbalanced strategies quickly. Human testing can also provide a sanity check on any game balance changes that we implement. We would like to study hybrid game balancing processes that combine human testing with coevolutionary search.

Visualization

Finally, we need to explore the role visualization techniques play in diagnosing game balance problems. Using techniques such as nonlinear projection and fitness webs we can produce 2D plots of the high dimensional landscapes common in game strategy spaces. These plots can be used to provide intuition about individuals in a fitness landscape [2] and guide us towards producing more balanced games.

8. ACKNOWLEDGMENTS

This material is based upon work supported by the Office of Naval Research under contract number N00014-05-0709.

9. REFERENCES

[1] P. J. Angeline and J. B. Pollack. Competitive environments evolve better solutions for complex tasks. In *Proceedings of the 5th International Conference on Genetic Algorithms*, pages 264–270, San Francisco, CA, USA, 1993. Morgan Kaufmann Publishers Inc.

[2] J. Ashlock, D.; Schonfeld. Nonlinear projection for the display of high dimensional distance data. *Evolutionary Computation, 2005. The 2005 IEEE Congress on*, 3:2776–2783 Vol. 3, 2-5 Sept. 2005.

[3] D. Charles and M. Black. Dynamic player modeling: A framework for player-centered digital games. In *International Conference on Computer Games: Artificial Intelligence, Design and Education*, November 2004.

[4] D. Charles, M. McNeill, M. McAlister, M. Black, A. Moore, K. Stringer, J. KÃijcklich, and A. Kerr. Player-centered game design: Player modeling and adaptive digital games. In *DiGRA 2005 Conference: Changing Views - Worlds in Play*, 2005.

[5] R. Dawkins and J. R. Krebs. Arms races between and within species. In *Proceedings of the Royal Society of London*, number 205 in Series B, pages 489–511, 1979.

[6] D. B. Fogel. *Blondie24: Playing at the Edge of AI*. Morgan Kaufmann, 2001.

[7] http://www.gamerankings.com/, January 28 2008.

[8] D. E. Goldberg. *Genetic Algorithms in Search, Optimization, and Machine Learning*. Addison Wesley Longman, Inc., 1989.

[9] W. D. Hillis. *Artificial Life II*, chapter Co-evolving parasites improves simulated evolution as an optimization technique, pages 313–384. Addison-Wesley, 1997.

[10] J. H. Holland. *Adaptation in natural and artificial systems*. MIT Press, Cambridge, MA, USA, 1992.

[11] P. Klepek. Npd fallout: Best selling games of 2007. http://www.1up.com/do/newsstory?cid=3165505.

[12] R. E. Leigh, T. Morelli, S. J. Louis, M. Nicolescu, and C. Miles. Finding attack strategies for predator swarms using genetic algorithms. In *Proceedings of the 2005 Congress of Evolutionary Computation*, September 2005.

[13] C. Miles and S. J. Louis. Co-evolving influence map tree based strategy game players. In *2007 IEEE Symposium on Computational Intelligence in Games*. IEEE Press, 2007.

[14] J. B. Pollack, A. D. Blair, and M. Land. Co-evolution of a backgammon player. In C. G. Langton and K. Shimohara, editors, *Artificial Life V: Proc. of the Fifth Int. Workshop on the Synthesis and Simulation of Living Systems*, pages 92–98, Cambridge, MA, 1997. The MIT Press.

[15] J. Portnow. The fine art of balance. http://gamecareerguide.com/features/478/the_fine_art_of_.php.

[16] C. D. Rosin and R. K. Belew. New methods for competitive co-evolution. *Evolutionary Computation*, 5(1):1–29, 1997.