Evolving Game-Playing Agents Through Coevolution

Lucas Ellgren
Division of Science and Mathematics
University of Minnesota Morris
Morris, MN 56267
ellgr001@umn.edu

ABSTRACT

The classical approach to creating a game-playing agent is to use search algorithms to find a solution in a space of possible moves, to develop special rules for playing called heuristics, or to use a combination of the two. We explore an alternate approach that uses evolutionary computation to evolve agents that play games at a human-competitive level. Evolutionary algorithms are capable of producing competent game-playing individuals through the process of evolution, instead of being designed entirely by humans. The evolutionary algorithms explored in this paper use coevolution, a method of evolution where agents are evaluated through their interactions with their peers. Agents must win games against other agents in order to survive and pass on their genes. We explore two recent papers in which researchers use coevolutionary algorithms to evolve agents for the games of FreeCell and Othello. We compare the effectiveness of the evolved agents against traditional, human-designed algorithms as well as some expert-level human players. Through this process, we demonstrate how coevolution can be used as a powerful and effective tool for solving complex problems.

Categories and Subject Descriptors

I.2.1 [**Applications and Expert Systems**]: Games; I.2.6 [**Learning**]: Knowledge acquisition

General Terms

Algorithms, Performance, Design

Keywords

evolutionary computation, evolutionary algorithms, coevolution, games, FreeCell, Othello

1. INTRODUCTION

The ability to play games has long been used as a benchmark in artificial intelligence. Early work in the field of artificial intelligence created systems that could play the game of tic-tac-toe and checkers at a near-perfect level. One historic milestone in the field was the victory of IBM's Deep Blue over the world chess champion Garry Kasparov in 1997 [3]. Typically, systems like these make use of search algorithms that attempt to find the best move in a space of all possible moves while also incorporating data from thousands of games played by experts. More recently, work has been done to *evolve* game-playing agents without the aid of archived knowledge or specially tuned search strategies. This is done through the use of evolutionary computation, a field of artificial intelligence which uses the basic principles of biological evolution to evolve solutions to problems.

Programs that use the ideas of evolutionary computation are called evolutionary algorithms (EAs). Traditionally, evolutionary algorithms solve problems that have an objective way to evaluate their solution. For example, an agent can be objectively evaluated on how well it predicts the physics of a bouncing ball. However, game players can only be evaluated subjectively, through comparison with other players. To accommodate for this, the evolutionary algorithms explored in this paper use *coevolution,* wherein the evolving agents play games against each other in order to evaluate their effectiveness. Such algorithms are called coevolutionary algorithms (CEAs). CEAs have been successfully used to create agents that play at human-competitive levels, and are usually more effective at evolving such agents than traditional evolutionary algorithms [9]. In one study, researchers were able to evolve an agent that played checkers at a grandmaster level, without the use of archived knowledge on the subject [8]. This alone demonstrates how useful coevolution can be when creating game-playing agents.

In Section 2 we will first give some background on search-based algorithms, artificial neural networks and evolutionary algorithms. We will then describe how coevolutionary algorithms work in detail in Section 3. Next, we will explore two recent research papers in which coevolution is used to evolve solvers for the games of FreeCell and Othello in Sections 4 and 5. Finally, we will draw conclusions about the effectiveness of CEAs for solving games, and consider some possible avenues for future work in the field in Section 6.

2. BACKGROUND

Before delving into the specifics of coevolutionary algorithms, we will first explain how game-playing agents work and the processes involved in evolutionary algorithms. Search based algorithms are explained in Section 2.1, and artificial neural networks are explained in Section 2.2. We describe EAs in depth in Section 2.3.

UMM CSci Senior Seminar Conference, April 2012 Morris, MN.

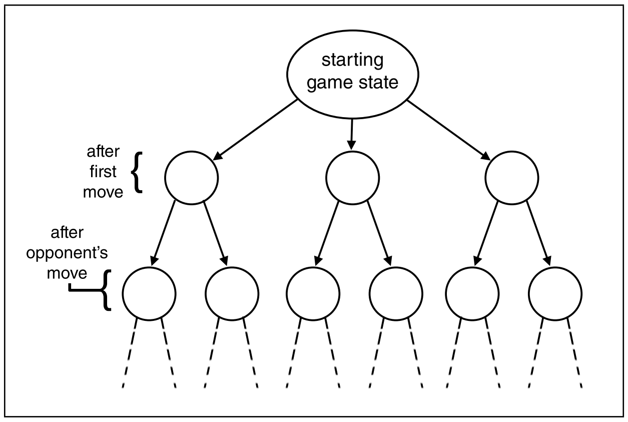

Figure 1: An example game-tree for a two-player game.

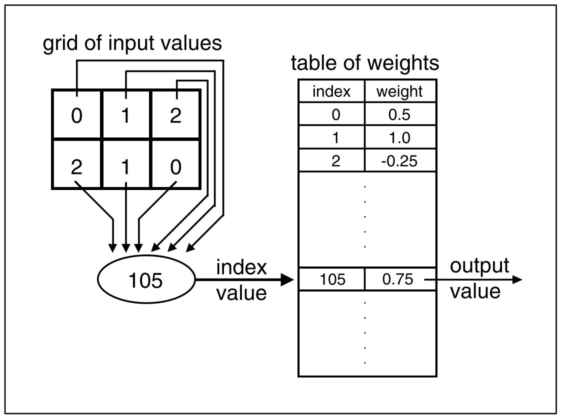

Figure 2: A diagram of an n-tuple neuron. Values from the input grid are used to calculate an index into a table of weights, which is the output value returned.

2.1 Search-Based Algorithms

As previously mentioned, a traditional way to create a game-playing agent is to use a search algorithm to locate the best moves within a space of all possible moves. To start with, the current state of the game and all subsequent possible moves are represented as a tree, called the *game-tree.* Each node in the tree corresponds to a game state, and every edge to another node further down the tree represents a legal move to another state. A diagram of an example game-tree is shown in Figure 1. Search algorithms are able to explore the possible nodes within the tree in an attempt to find the move that will most likely end in a win. How this happens differs from game to game. For example, with the game of checkers it is usually best to select the move that has the most possible victory states further down the tree.

There are several search algorithms that have been used to solve games. Breadth-first search (BFS) and depth-first search (DFS) are well-known search algorithms that can be useful for very simple games. Breadth-first search works by searching all of the possible moves at one level of a tree before moving on to the next. Depth-first search will explore one path to the end of the tree before returning to the beginning and starting again. While these algorithms are easy to implement, they are not very useful for complicated games. An example of a search algorithm used to solve complex two-player games is *depth-first iterative deepening* [11]. Depth-first iterative deepening works by doing depth-first searches iteratively deeper into the tree. It starts by doing DFS to the first level, then restarting and doing DFS to the second level, and so on until it reaches the estimated goal depth. Each time the algorithm starts searching from the root node, so it has a high probability to use a different route every iteration through the tree. Therefore, it will usually explore a large number of nodes without using as much memory as BFS. Depth-first iterative deepening is used to solve FreeCell in the paper by Elyasaf et al. as described in Section 4.2.

2.2 Artificial Neural Networks

Artificial neural networks (ANNs) are a computational model used to simulate the way real-life neurons work [5]. They are commonly used in the field of artificial intelligence, and can be used to create game-playing agents. They are constructed from groups of artificial neurons connected together. In a mathematical sense, these neurons can be represented as nodes in a directed graph. Artificial neural networks process information as it flows from neuron to neuron. The input neurons receive data, which is passed on to the neurons they are connected to. These neurons process the information in some manner, and then pass it along to the other neurons it is connected to. Eventually, the processed data reaches the output neurons where it is collected and used as the output value for the ANN.

ANNs are useful for artificial intelligence because their structure allows them to change and learn. Most ANNs typically use *weights* to affect the computation of their neurons. These are variables that are applied to the data within the neurons in some manner. For example, consider an ANN which takes a set of real numbers as input. The weights could be represented as coefficients which are multiplied with the input numbers. By changing the values of the weight coefficients, the output value of the ANN will change.

N-tuple neural networks are a type of ANN originally designed for optical character recognition [2]. They work by taking input from a grid of squares, and so they are ideal for use in board games. They are specifically used by Manning in his paper, as discussed in Section 5. Each n-tuple neural network is a collection of n-tuple neurons, otherwise known as just *n-tuples.* Each n-tuple contains a table of values refereed to as its *weight values.* When an n-tuple is given input from a grid of data, the values of the cells are used to compute an index into the table of weights. The value returned by the table acts as the output for the n-tuple. The process involved with an n-tuple neuron is shown in Figure 2. The n-tuple neural network collects the return values of all of its n-tuples to decide what to do next. By changing the weight values for specific inputs, it is possible to change how the network reacts to different situations.

2.3 Evolutionary Algorithms

Evolutionary algorithms, as explained in Section 1, use the fundamental concepts of biological evolution to evolve solutions to problems. EAs represent potential solutions to the problem they are trying to solve as *individuals.* Individuals are evaluated based on how good their solution to the given problem is. They are then given a numerical score called their *fitness*, which can be used to easily rank the individual with others in the population. Individuals with a better fitness score have a greater chance of mating with other individuals to pass down the components of their solution to the next generation.

The components that make up an individual's solution can be thought of as genes. Like genes in real world, an individual's genes undergo crossover and mutation as they are passed down to the next generation. *Crossover* takes the genes of two individuals and combines them to create a new genome. This can happen in many ways, and differs from implementation to implementation, but it always involves taking random pieces from both of the parents' genes. *Mutation* will take the resulting genome from a crossover operation and change it in a small way. This change is always random, which allows for some unexpected solutions to come about. Both crossover and mutation allow for new, unseen solutions to be explored by the EA.

Before an individual can even pass on its genes, it must first be selected for reproduction, and receive a partner. The process of selecting individuals which will mate is commonly referred to as *selection*, and it can take many forms, just like crossover and mutation. Unless the selection process is completely random, individuals with a greater fitness score are given a better chance to be selected. One example of a selection process is tournament selection, in which a random group of individual will compare their fitnesses with each other tournament-style in order to determine who has the best fitness score. The individuals with the top fitness score pair up and produce offspring, while the lower ones do not. This means that the good solutions will pass down their genes while the not-so-good solutions die off. Through this cycle, the EA continues to find better and better solutions as it progresses.

3. COEVOLUTION

For some problems, traditional evolutionary algorithms are not very effective. If a problem does not have an objective way of evaluating potential solutions, then it is usually more effective to use a coevolutionary algorithm instead. A coevolutionary algorithm will assign fitness values to individuals based on the interactions between them, instead of using an objective scale. For example, a game-playing individual that wins 5 out of 5 games against its opponents would be assigned a higher fitness value than an individual who only won 3 out of 5 games. Fitness functions like these are called *subjective*, since they are merely comparisons between individuals. With this method, individuals with a high fitness may not actually be good on an objective scale. For each generation in an EA using coevolution, the program will let the best individuals pass on their genes. Like traditional EAs, this will allow the population to steadily improve over time.

EAs attempting to evolve game-playing agents can benefit greatly from the use of coevolution. Games typically have a very large set of possible moves, and no objective way to rank the individuals that play them. Instead, game-playing individuals can be assigned fitness by ranking them against each other. This has been proven to be effective in the past [13].

3.1 Using Coevolution

As previously mentioned, an individual in a CEA is evaluated based on its interactions with other individuals and given a fitness value. An individual must interact with several other individuals in order to be evaluated. CEAs can have a single population of individuals, or multiple populations. With a single population, individuals interact with those in the same population. When there are multiple populations, individuals will only interact with those from a different population. In a multiple-population CEA, individuals within a population do not compete, which can allow for better cooperation between individuals.

Evaluation of individuals can occur in many different ways. In *All vs. Best Evaluation*, the individuals interact with the best individuals from the previous generation [6]. This can work in both single and multiple-population CEAs. For example, in a multiple-population CEA, the individuals of one population will be evaluated against the best previous individuals of the other population. *Tournament Evaluation* is another approach, in which individuals are evaluated in pairs within a tournament. The individuals who make it to the top brackets are given a better fitness than those in the lower brackets. This approach does not work for multiple populations since individuals would not be guaranteed to only interact with members from the other population.

3.2 Potential Issues with Coevolution

CEAs have several unique problems, which may limit or reduce their effectiveness. One such problem is called *disengagement*, which can occur when a CEA can not find a discernible difference between all or most of the individuals in its population. For example, if all the individuals did equally well in their interactions with each other, it would be impossible to rank them from best to worst. When this occurs, the population may stop making evolutionary progress. There are techniques for reducing disengagement in CEAs, as seen in Cartlidge and Bullock's paper [4]. Another problem is *cycling*, which occurs when a population oscillates within a set of solutions that have already been seen and does not make progress to a new solution. The chance of this happening depends on the problem being solved, but can frequently happen when attempting to solve games, as seen in Section 5.

4. SOLVING FREECELL

In this section, we will explore how Achiya Elyasaf et al. use coevolutionary algorithms to evolve game-playing agents for the card game FreeCell [7]. We start by explaining the game of FreeCell and its rules, and then describe the details of the CEA used by Elyasaf's team, called GA-FreeCell. The last subsection will describe the results found by the researchers, and compare them with human players.

4.1 The Game of FreeCell

FreeCell is a single-player card game similar to the game of Solitaire. The game is played with a standard deck of 52 playing cards which are arranged into eight piles called *cascades*. The object of the game is to arrange the cards into four piles called *foundations*. Cards must be placed in foundations in ascending order, and each pile can only contain cards from one of the four suits. There are also four FreeCells, in which any card can be placed and later moved to a different pile. Figure 3 shows the starting layout for a game of FreeCell. Cards can be moved on to others in cascade piles as long as they are in descending order and have alternating colors. For example, a black 9 can be placed on top of a red 10, but not on top of a black 10 or a red 8. The basic strategy involves organizing cards within the

Name	Description
HSDH	Heineman's staged deepening heuristic (Explained in-depth in his paper [10])
NumberWellPlaced	Number of well-placed cards in cascade piles
NumCardsNotAtFoundations	Number of cards not at foundation piles
FreeCells	Number of free FreeCells and cascades
DifferenceFromTop	Average value of top cards in cascades minus average value of top cards in foundation piles
LowestHomeCard	Highest possible card value minus lowest card value in foundation piles
HighestHomeCard	Highest card value in foundation piles
DifferenceHome	Highest card value in foundation piles minus lowest one
SumOfBottomCards	Highest possible card value multiplied by number of suites, minus sum of cascades' bottom card

Table 1: The basic heuristics used by GA-FreeCell.

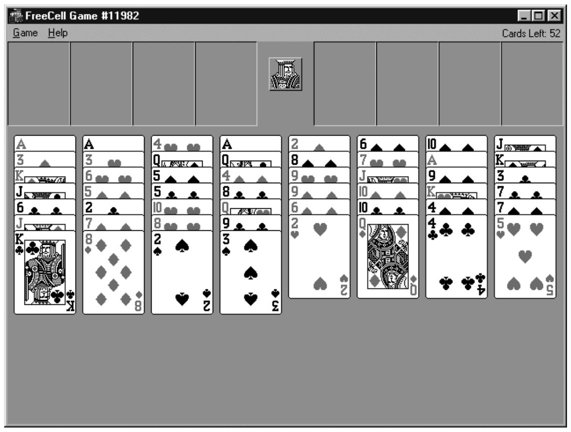

Figure 3: A screenshot of the FreeCell game included with the Windows 95 operating system.

cascades as much as possible by making use of the FreeCells and placing cards into the foundation piles.

FreeCell has existed as a card game for a long time, but it became most popular when it was released as a game along with the Windows 95 operating system. This version of FreeCell came with 32,000 different FreeCell deals, all of which are guaranteed to be solvable except for one (game 11982). These FreeCell deals will be referred to as the *Microsoft 32K*. After its release, it quickly gained popularity as a single-player card game. It is also a difficult problem to solve in general. FreeCell has been proven to be within the NP-hard set of problems, which means that it can't be solved in polynomial time with a traditional, deterministic computer. This makes it nearly impossible to solve with conventional game-tree searching algorithms.

4.2 GA-FreeCell

Elyasaf et al. tried using a few different searching algorithms before they moved on to evolutionary algorithms. They first attempted to solve the game using depth-first iterative deepening, a search technique described in Section 2.1. This proved unsuccessful; the algorithm failed to solve any of the deals in the Microsoft 32K. They then developed another version of the iterative deepening algorithm which used a novel heursitic from an algorithm developed by George Heineman [10]. The Heineman heursitic allowed the iterative deepening search algorithm to more accurately guess how far the current game state is from the winning state, which helped guide the search process. However, the iterative deepening algorithm combined with the Heineman heuristic (called IDA*) failed to meet the expectations of the researchers. They then decided to implement Heineman's staged deepening search algorithm (HSD) in full, for the purposes of comparison. IDA* and the HSD algorithm were tested on the Microsoft 32K along with the CEA developed by the researchers. The results are discussed in Section 4.3.

Instead of relying on just one heuristic from Heineman's HSD algorithm, the researchers decided to evolve their own through the use of coevolutionary algorithms. They represented their individuals as heuristics that would guide a staged deepening search algorithm. Specifically, the individuals were a collection of basic heuristic algorithms that combine input from the game state mathematically to form an estimate of the distance to the winning state of the game. The list of heuristics used and a description of their output values is shown in Table 1. The basic heuristics output an integer which is then normalized to a floating point number between 0 and 1. The value is then multiplied by the weight coefficient specified by the genome. The weights applied to the heuristics influence how much effect the heuristic has on the search algorithm's decisions. The total heuristic value of an individual is the sum of all of the values of the basic heuristics after being multiplied by their weight coefficients. In mathematical notation, this is: $H = \sum_{i=1}^{9} w_i h_i$ where h_i is the value from the ith heuristic and w_i is its corresponding weight coefficient. This is the value used to guide the search algorithm towards the winning game state.

To evolve the heuristics for their search algorithm, they used a coevolutionary algorithm that uses *Hillis-Style coevolution*. In Hillis-Style coevolution, the population of solutions evolves along with a population of problems. The fitness of the solutions depends on how well they solve the problems, and the fitness of the problems depends on how well they avoid being solved by the solutions. This means that both populations, the problem and the solutions, are in constant competition with each other, and the fitness of one population is inversely proportional to the fitness of the other. This allows the populations to evolve steadily and reduces the likelihood of disengagement. The problem of cycling, however, is still present.

The population of problems was made up of individuals that represented sets of FreeCell deals. Each individual had a set of six FreeCell deals from the Microsoft 32K. The evolved heuristics (representing the solutions) are evaluated against all six deals in order to decide their fitness. As the population evolves, the sets of FreeCell deals find the deals that are harder to solve, while the heuristics get better at solving them.

For their runs of the coevolutionary algorithm, the researchers had two populations containing between 40 to 60 individuals which ran for 300 to 400 generations. The indi-

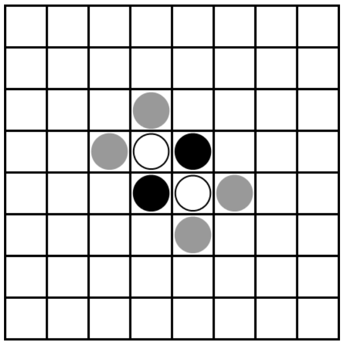

Figure 4: The starting position for the game of Othello, with the next potential moves for the black player shown in gray.

viduals had a 20% chance of reproduction, a 70% chance of crossover, and a 10% chance of mutation. Mutation happened similar to bitwise mutation, meaning that weights were randomly replaced by a randomly generated value between 0 and 1. The crossover operator used was one-point crossover. This operator works as follows: First the operator divides each parent's set of values at a randomly selected point. Then, the first part of the one parent and the second part of the other parent are combined to make a new set of values.

4.3 GA-FreeCell Results

The results reported by the researchers were very impressive. In comparison with HSD, GA-FreeCell reduced the amount of search by 87%, the time to find a solution by 93%, and the number of moves required to find a solution by 41%. GA-FreeCell solved 98% of the Microsoft 32K, beating HSD's previous record of 96%. These results are summarized in Table 2. These results show a vast improvement over traditional hand-crafted search heuristics, and make a convincing argument for the usefulness of coevolution.

5. SOLVING OTHELLO

In his paper, Manning employs coevolution to evolve n-tuple neural networks for the game of Othello [12]. We first explain the game of Othello in Section 5.1. We then explain the ideas of Resource Limited Nash Memory used in this CEA in Section 5.2. The CEA itself is explained next in Section 5.3. Finally, we explore the results of Manning's work in Section 5.4.

5.1 The Game of Othello

The game of Othello, also known as Reversi, is a two-player game played on a 8×8 grid. The game starts with two pieces of each color, white and black, arranged in the center of the grid in alternating order. Figure 4 shows an example of an Othello game board in its starting position.

Algorithm	Average Time to Solve	Deals Solved
HSD	709 seconds	30,859
GA-FreeCell	150 seconds	31,475

Table 2: A comparison between HSD and GA-FreeCell in terms of effectiveness and average time required to solve a deal.

The players take turns placing pieces on the grid and capturing their opponent's pieces. The objective is to end the game with the most pieces of your color on the board. A player can only place their pieces on a square that makes a straight line with one of their own pieces with one or more of their opponent's pieces "sandwiched" in between. Once the piece is played, all of the opponent's pieces in between get switched to the opposite color. This can happen horizontally, vertically or diagonally. This is demonstrated in Figure 4; the gray pieces indicate legal positions for the black player's next move.

While deceptively simple to play, Othello is an exceedingly complex game to master. It is estimated that there are at most 10^{28} legal positions for Othello games, with a game-tree size of approximately 10^{58} nodes [1]. To this day, Othello has not been solved by any algorithm.

5.2 Resource Limited Nash Memory

Resource limited Nash memory is the main idea behind Manning's paper and his CEA. This concept allows the algorithm to avoid cycling by maintaining a constant equilibrium of strategies in the population of game-playing agents. This method is based on the idea of *Nash equilibrium*, which comes from game theory. A Nash equilibrium involves having a set of players, all with different strategies, pitted against each other in a way where their strategies make an even playing field for all. In this equilibrium, all players are encouraged to keep their strategies constant, since they allow for the highest payoff if everyone else does the same. The set of strategies in the equilibrium is called the *support set*. A Nash equilibrium will continue until another viable strategy surfaces which beats all of the strategies in the support set. Then the Nash equilibrium is reformed by incorporating this new strategy into the support set.

Since the support set of strategies will keep an accurate record of all strategies seen so far, it can be applied to CEAs in order to reduce cycling. By pitting game-playing agents against those in the support set, the CEA will avoid rediscovering strategies that have already been seen in the past, and allow the CEA to focus on finding better strategies overall. The details of how Nash memory is incorporated into the CEA are examined in the next section.

5.3 Coevolutionary Algorithm

The Coevolutionary algorithm used by Manning uses n-tuple neural networks as its game-playing agents, as explained in Section 2.2. When it is an agent's turn to make a move, it considers all possible valid moves that it can perform at the current game state. Each potential move is represented as a game state and used as input for the network. The network actually consists of 12 6-tuples, which take input from different, 6-cell sections of the board. The 6-tuples use the location of pieces on their section of the game board to look up an index into their table of weight values. The weight values have a range from -1 to +1, and correspond to how good the 6-tuple believes the move to be. The return values of all the 6-tuples are collected and used to decide the next move. The N-tuple neural network will always select the move with the highest total return value.

For the coevolutionary algorithm, Manning used a population of 100 individuals, with randomly generated weight values. The n-tuple neural networks are evaluated by playing 40 games against random peers and then each member of the support set. If the individual does not do well against

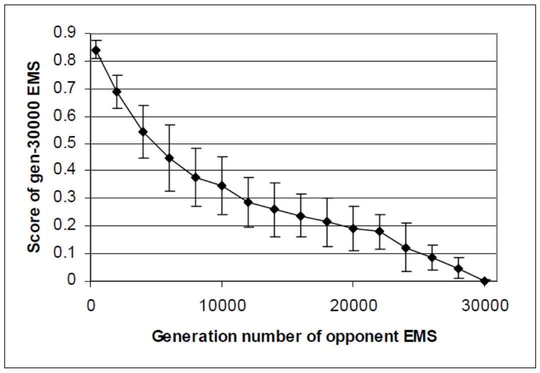

Figure 5: Mean fitness score of the EMS from generation 30,0000 against previous generations. The EMS scores best against opponents from the distant past. Error bars show one standard deviation of the sample.

its peers, it will not play against the support set and will be replaced by a new individual for the next generation. If an individual does well against its peers and the support set, it will be selected to become part of the new support set to maintain the Nash equilibrium. New individuals are created through a process of crossover and mutation similar to most CEAs. Crossover occurs by selecting 6 random 6-tuples from each parent, and mutation happens by randomly changing a weight value in a table with a 0.01% chance. For each generation, the support set and all individuals selected for reproduction survive while the rest are replaced.

5.4 Results for Solving Othello

Manning performed five experimental runs with his CEA. Each run lasted 30,000 generations, with the best individuals from each generation being periodically saved for comparison purposes. The best individuals were composed of a combination of the individuals in the support set, and called the *equilibrium mixed strategy* (EMS) for that generation. Manning found a steady increase in the competence of the EMS individuals. The EMS from generation 30,000 was able to beat the EMS from generation 400 92% of the time, and had a mean score that was 0.84 better. These results are summarized in Figure 5.

For the sake of comparison, Manning also performed a control experiment using a version of the CEA without Nash memory. These included five runs that lasted 10,000 generations for both the CEA with Nash memory and the control version. As a result, the EMS from the version with Nash memory scored 0.33 higher on average than the EMS from the control. These individuals also had a win rate of 66.5% over those in the control group. These results show how big of an impact Nash memory had in producing competent game-playing agents, and proves that the issue of cycling can be counteracted within a CEA.

6. CONCLUSIONS

We have now finished examining two papers which make use of coevolutionary algorithms to evolve highly competent game-playing agents. With proper use, CEAs can evolve agents that play complicated games without relying on human experience, which stands as an impressive example of how artificial intelligences can be constructed. CEAs could be applied to other aspects of artificial intelligence as well as solving games. It could be possible to evolve algorithms for a multitude of tasks, such as spatial navigation, computer vision or controlling robotic limbs. Many of these tasks need to be evaluated in a subjective way, which means that CEAs would be vastly more effective than traditional EAs. In the future, we hope to see coevolutionary algorithms used to expand the limits of what artificial intelligence can accomplish.

7. ACKNOWLEDGMENTS

I thank Elena Machkasova for her support in proofreading and lectures on writing. Nic McPhee, for his advice on writing, help with selecting sources and feedback on drafts. Ellery Crane, for additional proofreading and suggestions. I also thank my fellow students of my Computer Science Senior Seminar course for their assistance with proofreading and moral support.

8. REFERENCES

[1] V. L. Allis. *Searching for Solutions in Games and Artificial Intelligence*. PhD thesis, University of Limburg, 1994.

[2] W. W. Bledsoe and I. Browning. Pattern recognition and reading by machine. In *Papers presented at the December 1-3, 1959, eastern joint IRE-AIEE-ACM computer conference*, IRE-AIEE-ACM '59 (Eastern), pages 225–232, New York, NY, USA, 1959. ACM.

[3] M. Campbell, A. H. Jr., and F. hsiung Hsu. Deep Blue. *Artificial Intelligence*, 134:57 – 83, 2002.

[4] J. Cartlidge and S. Bullock. Combating coevolutionary disengagement by reducing parasite virulence. *Evol. Comput.*, 12(2):193–222, jun 2004.

[5] J. E. Dayhoff and J. M. DeLeo. Artificial neural networks: Opening the black box. *Cancer*, 91:1615–1635, 2001.

[6] E. D. de Jong, K. O. Stanley, and R. P. Wiegand. Introductory tutorial on coevolution. In *Proceedings of the 2007 GECCO conference companion on Genetic and evolutionary computation*, GECCO '07, pages 3133–3157, New York, NY, USA, 2007. ACM.

[7] A. Elyasaf, A. Hauptman, and M. Sipper. GA-FreeCell: evolving solvers for the game of FreeCell. In *Proceedings of the 13th annual conference on Genetic and evolutionary computation*, GECCO '11, pages 1931–1938, New York, NY, USA, 2011. ACM.

[8] D. B. Fogel. Evolving a checkers player without relying on human experience. *Intelligence*, 11:20–27, June 2000.

[9] N. Franken and A. P. Engelbrecht. Evolving intelligent game-playing agents. In *Proceedings of the 2003 annual research conference of the South African Institute for Computer Scientists and Information Technologists*, SAICSIT '03, pages 102–110, , Republic of South Africa, 2003. South African Institute for Computer Scientists and Information Technologists.

[10] G. T. Heineman. Algorithm to solve FreeCell solitaire games. `http://broadcast.oreilly.com/2009/01/january-column-graph-algorithm.html`, Janurary 2009.

[11] R. E. Korf. Depth-first Iterative-Deepening: An optimal admissible tree search. *Artificial Intelligence*, 27:97–109, 1985.

[12] E. P. Manning. Coevolution in a large search space using resource-limited Nash memory. In *Proceedings of the 12th annual conference on Genetic and evolutionary computation*, GECCO '10, pages 999–1006, New York, NY, USA, 2010. ACM.

[13] M. Shi. An empirical comparison of evolution and coevolution for designing artificial neural network game players. In *Proceedings of the 10th annual conference on Genetic and evolutionary computation*, GECCO '08, pages 379–386, New York, NY, USA, 2008. ACM.