Evolutionary Artificial Intelligence in Video Games

Reed Simpson
University of Minnesota, Morris
Division of Science and Math
Computer Science Dept.
simps148@morris.umn.edu

ABSTRACT

Evolutionary algorithms are becoming increasingly useful in numerous applications as computational resources improve with time. As computational resources have increased, demand for better quality in all aspects of video games has likewise increased. In this paper we will provide an overview of evolutionary algorithms, with particular attention to the context of video game artificial intelligence. Then we will discuss a specific type of artificial intelligence, often created by evolutionary algorithms, called an artificial neural network. Finally, we will present a number of examples in which evolutionary algorithms have been applied to video game artificial intelligence, as well as the strengths and weaknesses of evolutionary algorithms in this context.

Categories and Subject Descriptors

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

General Terms

Algorithms, Experimentation

Keywords

computer games, video games, artificial intelligence, game AI, evolutionary algorithms, genetic algorithms, artificial neural networks, neuroevolution, games, evolutionary computation

1. INTRODUCTION

In the context of video games, artificial intelligence (AI) refers to a system which controls agents in a game in a way that makes the player feel like he or she is interacting with an intelligent entity. Game AIs are an important component in most video games, and their complexity can vary considerably. This paper will discuss the use of Evolutionary Algorithms (EAs) in the development of game AIs, an approach which is relatively underutilized.

Although relatively simple approaches to game AI have so far been sufficiently sophisticated, as computational power increases the players of such games have begun to expect more sophisticated AIs just as they have come to expect sophisticated visuals and sound. Complex AIs are difficult to code by hand due to the increased mental load on the programmer, which leads to developers searching for other avenues of managing complexity. Evolutionary Algorithms (which will be described in Section 3) offer a potential solution to this problem by shifting the bulk of the work to the computer. The programmer initially sets up the EA, after which the programmer need not be concerned with the inner workings of the algorithm, only the results. This has the effect of reducing programmer effort at the expense of computational resources.

Throughout the history of video games, the industry has usually put the biggest emphasis on graphics [2]. Sophisticated graphics entice people to purchase the game. In recent years, high-definition, three-dimensional graphics have quickly become commonplace, and many game developers are thus leaning towards game AIs as their next area of focus. The majority of modern game AIs do not make use of evolutionary algorithms. Game designers prefer to use hand-coded AIs in their games, so why use EAs? What sort of advantages could we get from an evolutionary algorithm?

2. MOTIVATION

2.1 Better Game Playing

Some games have proven too complex for simple AI methods. For example, only recently have hand-coded AIs even attained average human playing level in the game Go (a board game known for the rich strategies that stem from its simple rules) [4]. In Xin Yao's paper [10] he says that EAs are especially useful for complex problems because they are less likely to get caught in local maxima (i.e. a solution that is good, but not the best).

Even though the gaming industry does not use evolutionary algorithms, they recognize the need for more advanced AI techniques. In their paper, McFarlin *et al.* state that, although there is some dissension about the direction game AI will take in the near future, it is generally agreed upon that better AI techniques are needed, and EAs may be the solution.

2.2 Humanlike Behavior

A video game with enemies that exhibit strategies with obvious and exploitable flaws is less fun than one with enemies that behave more intelligently. While human players may exhibit such flaws, they will often be repaired when the player realizes their mistake, and differ from player to player. When an AI exhibits such a flaw, it is usually repeated over and over again, even from different entities in the game.

A term often used to describe a quality of a video game is "immersive". An immersive video game allows the player to believe that the video game world is in some sense real, tricking the mind into becoming emotionally invested in the world in much the same way as we become emotionally invested in fictional stories in other mediums, such as books and movies.

While graphics have long been thought to be the most important component in making a video game immersive, as graphics have rapidly improved it can be jarring to see a highly realistic depiction of a human behaving like an automaton. An enemy player that is able to recognize a loophole in its strategy and repair it, or modify its strategy over time, would be one step closer to the goal of creating a believable illusion of an intelligent opponent. [9, 2]

2.3 Programmer Cost

Last but not least, one of the most important reasons to use an EA is that it trades programmer work for computational work. Computational work is less valuable than programmer work, since a programmer is paid tens of thousands of dollars per year, while a computer costs several hundred dollars and will probably last several years. Since computational work is much cheaper than programmer work, this is often a very good trade even if it is not especially efficient. For example, trading several hours of programmer time for several days of computation time could easily be a good trade, if you have one or two extra computers that are not otherwise being used.

Evolutionary algorithms can already produce results similar to a programmer in a comparable amount of time in some instances [5]. Additionally, computers are projected to continue to increase in power in the foreseeable future, which means the trade off will become more and more efficient, and will allow us to apply EAs to more complex problems. Furthermore, as AIs become more complex the cost of a programmer coding them by hand will scale poorly compared to the cost of that same programmer setting up an EA. Ultimately, the cost of setting up and running an EA will become cheaper than coding that AI by hand.

3. EVOLUTIONARY ALGORITHMS

This paper is primarily about evolutionary algorithms applied to AI development, so some background on EAs will be relevant to some readers. An EA is a type of algorithm which is engineered to solve a given problem. It does this by creating a population of "individuals", where each individual represents a solution to the problem[1]. The EA then varies this population over time by removing poor solutions from the population and replacing them with new individuals derived from the more successful solutions. Randomness involved in the introduction of new individuals creates and maintains diversity in the population, allowing it to gravitate towards better solutions to the original problem over time.

An EA starts with an initially random population. Usually, EAs are organized into "generations" where each generation has a new population of individuals produced from individuals in the previous generation. To produce the next generation, each individual in the old generation is assigned a number representing how good of a solution it is, i.e. its "fitness score." Then, individuals with high fitness (better solutions) are selected to produce new individuals in the new population. New individuals are produced from old individuals in a way in which some of the old individual's traits and characteristics are passed down, often by means of the *mutation* operation (which modifies a random part of the individual) and the *crossover* operation (which combines the traits of two individuals). The individuals in the new population will be similar to the best individuals from the previous population, but the random modifications have a chance of producing individuals that are better solutions. In this way, the population tends towards individuals with high fitness.

3.1 Fitness Evaluation

The most important part of an EA is determining which individuals in a population are better or worse solutions to the problem that the EA is supposed to solve. Naturally, we will want to keep the most fit solutions (the solutions that best solve the problem) while abandoning the least fit. This is done via a fitness function, which is intended to measure the relative success of an individual at solving the problem. In the context of game AIs, this is often not a "function" in the traditional sense of the word, but rather an implementation of the individual, either on some data or in competition against another individual, to see how well it does. This can easily become the most computationally expensive part of an EA, and the care put into its implementation can easily determine how quickly an EA reaches a sufficiently good solution (or if it does at all).

The fitness function does not have to be perfect, and often includes some noise (randomness). For example, if a set of algorithms is produced for playing a two-player board game, their fitness might be determined via a round-robin style series of matches. However, if the game involves any amount of randomness you are not guaranteed to get exact results. Often, to simplify the calculation, instead of a round-robin style series of matches, individuals will be paired randomly for a smaller number of matches. This will produce more noise but will speed the evaluation of fitness. Where the happy medium will lie depends on the given problem, which is one of the reasons why implementing a good EA is nontrivial.

3.2 Selection and Reproduction

Once the fitness of each individual is calculated, it is time to produce the next generation of individuals. This is usually done via one or more operations which pass some traits on to the offspring while modifying others.

Individuals are selected from the population based on fitness score, as described in the previous section. The most fit individuals are copied (with or without modifications) into the next generation. Copying the most fit individuals directly from one generation to the next safeguards against

[1]Aside: the terms "genotype" and "phenotype" are often used to refer to the individual that is evolved, and the solution that individual represents, respectively. Some readers may be more familiar with these terms from other resources on evolutionary algorithms, or from the biological terms they are inspired from.

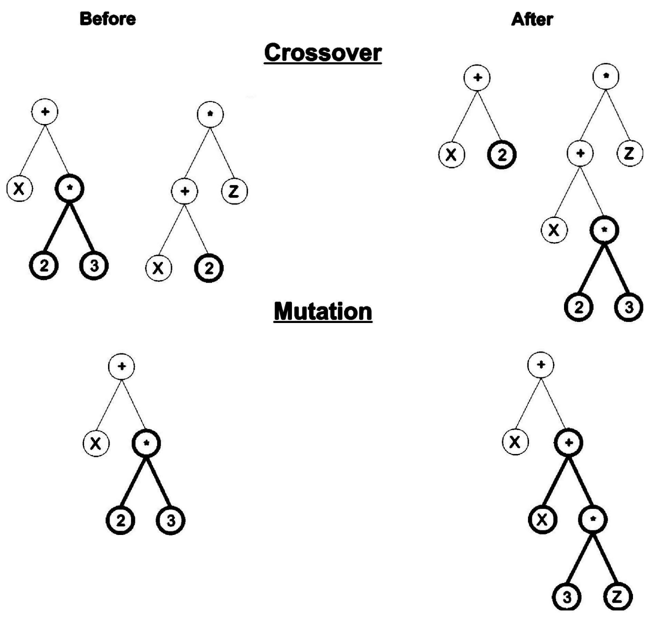

Figure 1: A diagram depicting the two basic mechanisms of evolutionary algorithms [3].

backtracking to worse solutions. Some EAs will allow individuals with higher fitness to produce many offspring, with the number of offspring each individual produces proportional to its relative fitness.

Mutation is the most basic mechanism of EAs, and also the most vital. Some EAs are implemented using only mutation, especially when computational power is limited. Essentially, some part of the individual is randomly modified. In Figure 1, one can see an example of mutation being performed on a tree structure in the lower half of the figure. Trees like this are often used to represent numerical expressions. In this case, the tree on the left represents the expression $x + (2 * 3)$. The letter x in this context might represent a constant, a variable, or a function that returns a number. When mutation is performed on this tree, a random subtree is selected (depicted in bold) and then replaced with a new randomly grown subtree (shown in bold in the tree on the right). The end result of the mutation, the tree on the right, represents the expression $x + (x + (3 * z))$.

Crossover is the process of combining parts of two different individuals into a new individual with traits from both parents. In Figure 1, an example of crossover is depicted in the upper half of the figure. The two trees on the left, representing the numerical expressions $x + (2 * 3)$ and $(x + 2) * z$ respectively, create two new trees each of which contains parts of both parents. In this case, the subtrees outlined in bold have been swapped, so each child has the main tree section from one parent and the bolded subtree from the other. This particular crossover operation produces two offspring, but many crossover operations produce only one child, or randomly selects one to add to the population.

Crossover can be difficult to implement in a reasonable way, since the two individuals may be wildly different. One way of avoiding this problem is by implementing markers of some kind which identify analogous sections of each individual, so they can be interchanged more easily.

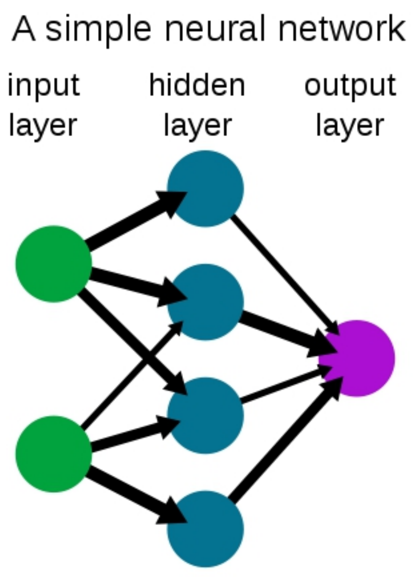

Figure 2: An example of the ANNs that control each individual robot in NERO[9].

4. ARTIFICIAL NEURAL NETWORKS

One common type of EA is one that produces an Artificial Neural Network (ANN). The principles of ANNs are inspired by biological neural networks, or brains, in the sense that ANNs are made of interconnected nodes (neurons) which produce results based on the weighted connections between the nodes.

The input into a Neural Network comes from a set of "input" nodes. In the context of game AIs, the input nodes take data directly from the game environment, in the form of numerical data such as distance to a wall, distance to an enemy, or current health. The output from an ANN comes from one or more output nodes, which are used to determine a course of action. For example, in Figure 2 the input nodes are shown on the left of the image and the output nodes are shown on the right.

Each node takes values from one or more inputs, performs some operation with them, and sends a value to that node's output. Any number of nodes can take one of their inputs from a node's output. In this way, values propagate through the ANN until they reach the output nodes.

4.1 Transfer Function

Once a node has inputs, it must calculate its output value. It does this by means of a special function called a transfer function. This function is usually the same across all nodes, but takes into account both the inputs of the node and a set of weights specific to that node (depicted in Figure 3). An example of a basic function is given below:

$$output = f(\sum_{j=1}^{n} weight_j * input_j - threshold)$$

Where the function f is some function predefined by the programmers. Usually, f is a sigmoid function. A sigmoid function is a type of function which converts any real number to a number within a set range, as seen in Figure 4. In the

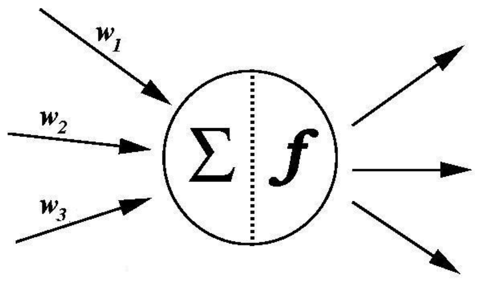

Figure 3: A diagram of a typical node in a Neural Network

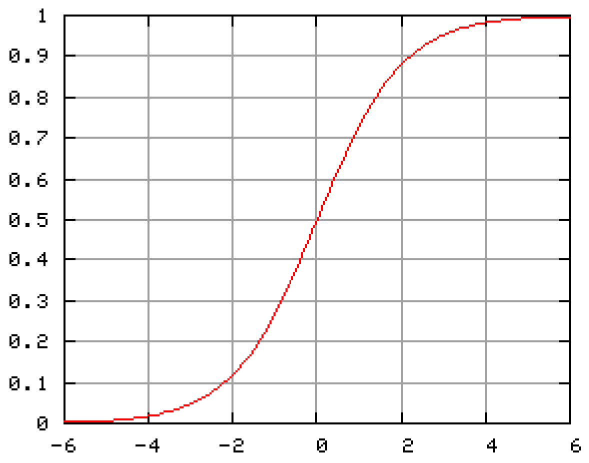

Figure 4: A standard sigmoid function (Wikipedia).

example in Figure 4, the range is between 0 and 1. The function f may also be a gaussian function (popularly known as a bell curve). The function can be evolved, but is usually fixed by the programmer, as are all of the examples discussed later in this paper.

4.2 Topology and Weights

Often, both the topology (i.e. structure, number of nodes and node connections) and the weights of an ANN are evolved, but it is possible to use a fixed topology and just evolve the weights. ANNs can often change their weights dynamically, a process which falls under the domain of machine learning rather than EAs. If the topology of the ANN is evolved, evolution and learning can be combined. In the examples presented in this paper that incorporate ANNS (namely, NERO and Wolfenstein 3D) both the topology and the weights are evolved together, so further discussion of machine learning is beyond the scope of this paper. ANNs are often very effective, but can be computationally expensive, especially for more complex networks [10, 3].

Each input into a node has a weight, and the input is multiplied by this weight before the modified inputs are added together. Essentially, an input with a higher weight has a larger influence on the output of the node.

5. EXAMPLES

In the effort to explore the potential abilities of evolutionary algorithms, many researchers have implemented this technique in various ways. In this section, we will explore three interesting applications of evolutionary algorithms to video game AI. Whether the evolution occurs in game at play time, as in the first example, or the AI is evolved beforehand and tested in the game, as in the second and third examples, all of the following examples are successful implementations of EAs to the task of making a game AI.

5.1 NERO

NERO is a game designed from the ground up to incorporate evolutionary ANNs. In it, the player trains a team of robots to fight opponents autonomously. Each robot is controlled by an ANN, the topology of which is evolved based on event weights provided by the player (not to be confused with the edge weights evolved by the ANN, which are also evolved) .

Each potential event, such as hitting an enemy or being hit by an enemy, has a weight provided by the player. The fitness score is calculated by multiplying the number of occurrences of each event by the weight of that event. For example, if the player chooses to reward robots who charge straight at an enemy, then robots will receive a higher fitness score the longer they stay near an enemy. By combining this with a negative weight for being hit by an enemy, a robot may develop a preference for remaining at a certain distance from the enemy, as this will provide a balance between the two weights.

Unlike the other two examples in this section, NERO does not divide its populations into generations. Instead, evolution occurs in real-time. Every n seconds, a parameter set by the player, two individuals with high fitness are selected. Then, a new individual is created by performing both the crossover and mutation operations (as described in Section 3.2). The new individual replaces an old individual with low fitness. The process of selecting two individuals and creating a new individual is depicted in Figure 5. Because the fitness score is calculated in real-time, the new individual is protected for a short time from being replaced so its fitness score can stabilize.

As evidence of the success of the experiment, the authors present the speed with which the robots adapt to weights presented by the user. Starting from a random neural network (the default) it takes the robots an average of 99.7 seconds to develop seeking behavior with the appropriate weights. This is a relatively simple behavior, but the authors also demonstrated the ability for the game to quickly evolve complex behaviors by manually controlling a hostile robot and training the robots to find alternative paths to avoid being shot. This took only 2 minutes, again with the proper choice of weights. This successfully demonstrates that neural networks can be evolved quickly and in real-time, a result that could be used to make games more dynamic and less predictable. [9]

5.2 Robocode

Robocode is a competition where contestants submit a java program to control virtual tanks which fight each other autonomously. One of the first, and most successful, contestants not programmed directly by a human is described by Sipper *et al.* in *GP Robocode: Using Genetic Programming*

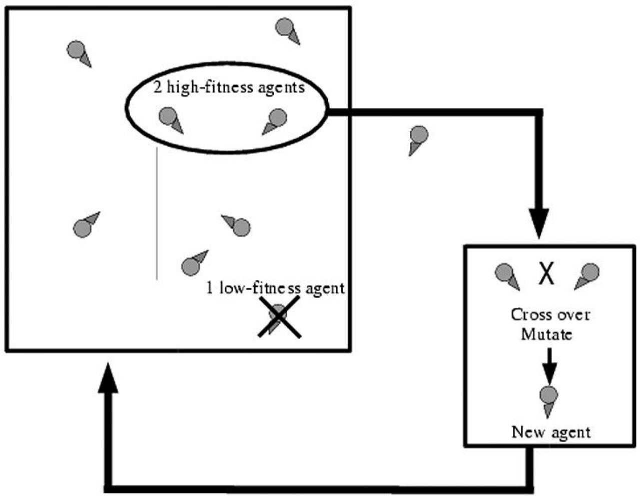

Figure 5: The reproduction model for the autonomous robots in NERO[9]

```
while (true) {
    TurnGunRight(INFINITY); //main code loop
}
 OnScannedRobot() {
    MoveTank(< EA#1 >);
    TurnTankRight(< EA#2 >);
    TurnGunRight(< EA#3 >);
}
```

Figure 6: Robocode player's code layout

to Evolve Robocode Players [8]. This player was developed for the Haiku-bot challenge, a division of the competition restricting code size to four instances of a semicolon (which can be interpreted as four lines of code). The code consisted of a main loop which rotated the gun of the tank. When an enemy tank was spotted, the code triggered an *OnScannedRobot* event, which caused the tank to execute a series of commands: move tank forward, rotate tank, and rotate gun (see Figure 6). The input values for each of these commands were numerical expressions like those described in Figure 1. Because the *fire* command was implemented as a function with a numerical return value, it could be placed at any point in any of the three evolved numerical expressions just like a numerical constant.

Sipper *et al.* chose to carry a population size of 256. They chose three high-scoring, hand coded players from previous competitions. In each generation, each individual plays three rounds against each of the three benchmark players. Three competitors were chosen because previous experiments indicated that evolutionary algorithms developed against a single opponent did not generalize to other opponents[7, 8].

As with all evolutionary algorithms, choice of fitness algorithm is crucial to determining the success of the evolutionary algorithm. As Sipper *et al.* explain in their paper [7], their goal was to succeed in the scored Robocode tournament, so they chose to adopt the scoring function used in the tournament itself:

$$FractionalScore = \frac{PlayerScore}{PlayerScore + AdversaryScore}$$

This fitness function proved unhelpful early in the evolutionary run, as very poor players would not score any points and thus earn a score of 0 (zero). This was fixed by adding a small positive constant ϵ to the function:

$$FractionalScore = \frac{\epsilon + PlayerScore}{\epsilon + PlayerScore + AdversaryScore}$$

This solved the problem by giving a higher score (non-zero) to the players that best avoided their opponent (and thus prevented their opponent from getting a higher score). This is an excellent example of how the choice of a fitness function can have a dramatic effect on the speed of an evolutionary algorithm.

The most successful AI, after 400 or so generations, was submitted to the competition. It placed third of 27 contestants. Sipper *et al.* make the argument that placing third fulfills their initial goal of making an algorithm that is human-competitive by holding its own against human players. They estimated their algorithm took around 256 hours of computation per evolutionary run, a time which they split between 20 computers to save time. [7, 8]

5.3 Wolfenstein 3D

Wolfenstein 3D is a game released in 1992 for DOS. The game itself features a rather simplistic state-based AI. In *Evolving a Better Adversary: A Case Study in a German Castle*, McFarlin *et al.* describe their evolutionary AIs designed to be used in place of the default AI[5]. In effect, they use Wolfenstein 3D as a testing ground for evolved ANN-based game AI.

To determine the fitness of an individual, McFarlin *et al.* simply kept track of the amount of damage inflicted by the individual. This has the benefit of rewarding individuals that remain alive long enough to inflict more damage.

The average time for a human player to complete the level with the default AI was 117.20 seconds, with a standard deviation of 13.28. To determine the merit of the final ANN, this number was compared with the time for a human player to complete a level populated with enemies controlled by the ANN. The average time to complete the level for the best evolved ANN was 141.04, with a standard deviation of 23.68.

The goal of this experiment was not to produce AI models that were more difficult to beat than the default AI. Rather, the goal was to demonstrate that the process of creating such AIs could be automated. In fact, the authors made a point of suggesting that future research in this field investigate methods for automating the creation of AIs that fall within a certain skill range, such as "good for beginners."

6. CURRENT ISSUES

Historically, game developers have rarely made use of experimental techniques such as evolutionary AIs. Although evolutionary AIs have much potential, they have yet to see much use in the video game industry because they can be problematic in several ways. Techniques used in coding AIs by hand have been honed by many programmers over the years, while EAs are relatively untested. In this section, we present a number of problems that can occur when implementing an EA, or concerns a developer might have about using an EA. In addition, we also present some potential solutions to these problems and concerns.

6.1 Computational Cost

Although EAs trade programmer effort for computational effort, this trade is not very efficient. Setting up an EA may take less work than coding an AI by hand, but it still takes a nontrivial amount of work. Moreover, even a relatively simple EA can take days to compute on a standard computer, and if the programmer does not have something else to do in the meantime nothing is gained. The usefulness of EAs is impeded by the fact that they are computationally expensive, and the problems we wish to solve with them are already complex. For game AI, this means that simple AI can already be made efficiently using hand coding techniques, and complex AI is very time consuming.

This problem is the easiest to solve: we must simply use faster computers. As computer performance is scheduled to continue to increase, this problem will essentially solve itself. One could argue that as computers increase in power, we will also want to implement more complex EAs. If this is the case, then more research will have to be done to make EAs more efficient.

6.2 Speed and Reliability

For simple applications, it's easier to have a human programmer hand code an AI than implement an EA. For complex applications, the EA may not terminate in a reasonable amount of time. In addition, even though programmer time is more expensive than processor time, the current state of EAs is that they are often not reliable enough to let them run completely unsupervised. This problem, compounded with the amount of time it takes for an EA to terminate, means that it is often more efficient to code the AI by hand. [5, 6, 1]

Many speed and reliability issues can be improved, if not entirely remedied, by changing how the EA is set up initially. For example, the choice of fitness function can have dramatic effects on the speed of an EA, and a poor fitness function can easily cause an EA to spin out and not go anywhere useful. Fundamentally, this problem will only be improved by a better understanding of EAs that comes from more research on this topic.

6.3 Transparency

Game developers like to be able to see how their AIs work internally to make sure they won't do anything unexpected. ANNs, in particular, are difficult to understand by direct observation. Essentially, the results of an EA are defined by what they do, not how they do it. This makes game developers uncomfortable, probably because they cannot predict how an AI will behave in an untested situation. [5, 1]

This may improve with our understanding of EAs, but it is unlikely to ever be perfect. It is fair to point out, however, that it is difficult to completely avoid unpredictable behavior with traditional AI techniques, especially as complexity increases. The fear that an AI might encounter an unexpected situation and perform erratically could be mitigated by making the AI be adaptable. If the expectation is that the AI acts as a human would, it would be understandable if it behaved poorly in a new situation but learned from its mistakes over time.

7. CONCLUSION

Evolutionary AIs are a useful tool in game AI. They are comparable, and in some ways superior, to human programmed AI. This technique has been very useful in developing AIs to play complex games, and will likely continue to be useful in this regard.

Despite huge computational advances, EAs still suffer from insufficient computational resources. On the other hand, as computational resources increase, the effectiveness of EAs will scale extremely well compared to other methods of algorithm development [5, 4]. Researchers have already found many useful applications for EAs in game AI, and we expect the utility of this approach to increase in proportion to computational power.

8. REFERENCES

[1] S. Bakkes, P. Spronck, and J. van den Herik. Rapid and reliable adaptation of video game AI. *Computational Intelligence and AI in Games, IEEE Transactions on*, 1(2):93 –104, june 2009.

[2] S. Cass. Mind games [computer game AI]. *Spectrum, IEEE*, 39(12):40 – 44, dec 2002.

[3] N. Kohl and R. Miikkulainen. 2009 special issue: Evolving neural networks for strategic decision-making problems. *Neural Netw.*, 22(3):326–337, Apr. 2009.

[4] S. M. Lucas, P. Rohlfshagen, and D. Perez. Towards more intelligent adaptive video game agents: a computational intelligence perspective. In *Proceedings of the 9th conference on Computing Frontiers*, CF '12, pages 293–298, New York, NY, USA, 2012. ACM.

[5] D. McFarlin and P. Todd. Evolving a better adversary: A case study in a german castle. In *Artificial Life, 2007. ALIFE '07. IEEE Symposium on*, pages 229 –235, april 2007.

[6] S. Rahnamayan, H. Tizhoosh, and M. Salama. Opposition-based differential evolution. *Evolutionary Computation, IEEE Transactions on*, 12(1):64 –79, feb. 2008.

[7] Y. Shichel, E. Ziserman, and M. Sipper. GP-Robocode: Using genetic programming to evolve robocode players. In M. Keijzer, A. Tettamanzi, P. Collet, J. van Hemert, and M. Tomassini, editors, *Genetic Programming*, volume 3447 of *Lecture Notes in Computer Science*, pages 143–143. Springer Berlin / Heidelberg, 2005.

[8] M. Sipper, Y. Azaria, A. Hauptman, and Y. Shichel. Designing an evolutionary strategizing machine for game playing and beyond. *Systems, Man, and Cybernetics, Part C: Applications and Reviews, IEEE Transactions on*, 37(4):583 –593, july 2007.

[9] K. Stanley, B. Bryant, and R. Miikkulainen. Real-time neuroevolution in the NERO video game. *Evolutionary Computation, IEEE Transactions on*, 9(6):653 – 668, dec. 2005.

[10] X. Yao. Evolving artificial neural networks. *Proceedings of the IEEE*, 87(9):1423 –1447, sep 1999.