Evolving Autonomous Agent Control in the Xpilot Environment

Gary B. Parker
Computer Science
Connecticut College
New London, CT 06320
parker@conncoll.edu

Matt Parker
Computer Science Department
Indiana University
Bloomington, IN 47405
matparke@indiana.edu

Steven D. Johnson
Computer Science Department
Indiana University
Bloomington, IN 47405
sjohnson@cs.indiana.edu

Abstract- Interactive combat games are useful as test-beds for learning systems employing evolutionary computation. Of particular value are games that can be modified to accommodate differing levels of complexity. In this paper, we present the use of Xpilot as a learning environment that can be used to evolve primitive reactive behaviors, yet can be complex enough to require combat strategies and team cooperation. In addition, we use this environment with a genetic algorithm to learn the weights for an artificial neural network controller that provides both offensive and defensive reactive control for an autonomous agent.

1 Introduction

The study of learning in autonomous agents is important for the development of robots that can operate independently and have the ability to adapt to changes in their capabilities, the environment, and mission. There are many practical applications of these types of robots including military, search and rescue, and exploration. Computer games represent one class of platforms for experimentation with learning in autonomous agents. They can offer an environment that is complex and challenging, often modeling relevant characteristics of the physical world.

The computer game Xpilot offers such benefits while requiring minimal in-game graphics. Xpilot has several levels of complexity that allow researchers to expand their work as their systems grow in their learning capabilities. It is a simple matter to restrict the area of play and/or complexity through the design of maps and restrictions on the player’s capabilities and the number of players involved. In addition, Xpilot is an internet game that can be downloaded for free, allowing several groups to interact through remote agent competitions. In this paper, we present the game Xpilot and discuss a system that we developed for using *evolutionary computation* for learning agent control. A genetic algorithm used to evolve weights for a fixed neural network is used to control an Xpilot bot. The resulting controllers exhibit improved combat capabilities over 256 generations of evolution, with different runs evolving distinct combat strategies.

Using evolutionary computation for learning in games has been approached in several ways. The bulk of this work has been done on board games such as Go and thought games, such as the prisoner’s dilemma problem. Examples of this research include the work done by Fogel to learn checkers [1]; by Konidaris, Shell, and Oren to evolve a neural network to learn to capture in Go [2]; and by Hingston and Kendall in using evolution in the iterated prisoner's dilemma problem [3].

Some research has also been done on learning controllers for interactive computer games. In research by Cole, Louis, and Miles, parameters for the popular first person shooter, Counter-Strike, were evolved to help a computer controlled bot determine when to use various weapons and perform a variety of behaviors [4]. In work by Yannakakis and Hallam, opponents were evolved for the game Pac-Man [5]. Funes and Pollack evolved behavior for light-cycles in his online Java Tron applet [6].

No published research on the use of evolutionary computation for learning controllers for the agents in Xpilot could be found. However, an online paper written by Funes [7] describes an attempt to evolve a combat robot in Xpilot using programmed heuristic behaviors. There were six possible behaviors, such as firing at the nearest enemy and random wandering. It evolved to constantly do one behavior: shoot at the nearest enemy. In our research, we wanted to use more information about the Xpilot gaming environment (inputs to the controller) than was used by Funes and we wanted to avoid predefined behaviors by making the output more primitive directions such as turn direction/rate, fire or not, and thrust or not.

We hope the work described in this paper will convince other researchers of the benefits of using Xpilot for experimentation in using evolutionary computation for learning controllers for autonomous agents. In addition, we present preliminary work using a genetic algorithm to learn the connection weights for a fixed artificial neural network controlling the agent as it engages in combat with a single opponent. Our research shows that the agent learns reactive control allowing both defensive and offensive maneuvers while contending with a free space environment.

2 Xpilot

Xpilot is an open-source 2-dimensional multiplayer space combat game (Figure 1), playable over the internet. A player controls a ship using the keyboard or mouse and must destroy enemy ships while avoiding being killed by the enemy. There are different maps that vary in size and objectives, with team play, capture the flag, or free-for-all

combat. There are often ship upgrades made available that enhance a ship's abilities with added weapons like lasers or triple fire, or with ship upgrades such as cloaking or increased fuel. The game uses a client/server approach to support multiple players. It has realistic physics and solid networking code.

Figure 1: A player's ship in combat with another ship. This is from the player's perspective, so only the other ships (in this case Sel) are labeled. The grid lines at the top of are parts of a wall. The faint dots behind the ships show thrust and the dot to the right of the player's ship is a bullet.

2.1 Modifications to Accommodate Learning

Because of its predetermined networking protocol and simplicity, Xpilot makes a good platform for developing Artificial Intelligence (AI). The Xpilot server and client are synchronized frame by frame. For each frame, the server sends packets to the client with information about what to display. The client receives this information, parses it, and displays it graphically on the screen. The client sends back to the server information about mouse movement and what keys the player is pressing on the keyboard. Making Xpilot usable for AI is just a matter of parsing the information received by the client and storing it in easily accessible variables. We have created our own module of Xpilot with structures containing useful information about the player's ship, the enemy ships, the bullets, the radar, and the map. The structures contain information like X and Y coordinates, velocities, accelerations, distances, etc. The AI module also includes commonly used functions for aiming, calculating distances, finding barriers, and so forth.

The AI module controls the ship by simulating keyboard key presses and mouse movement. To turn the ship, it sends a packet to the server saying the mouse has moved a certain amount. To thrust, shoot, and do other controls it sends a packet saying a particular key has been pressed on the keyboard. Between the sending and receiving, an AI function can be inserted that takes as inputs environment information and outputs the control for every frame. Because all this is done in the client, any number of AI controlled clients may be connected to play against one another or against real people.

2.2 Issues for this Research

There is a large variety of AI behaviors that can be learned using Xpilot, from team combat to hunting and path-finding. For this project we chose to have an AI develop effective combat skills for a simple one-on-one combat environment. We wanted the AI to learn to avoid flying into walls, to dodge an opponent's bullets, and to effectively attack the enemy. The normal AI module that we built had all variables and coordinates relative to the origin, the bottom left corner of the map. The angles (heading, bearing, etc.) were relative to a compass rose that had 0 as pointing right, parallel to the top of the screen (this is how it is defined in Xpilot).

For the controller, inputs relative to the origin cause problems in learning. The ship may learn only to do certain things in certain spots on a map, or learn to attack the enemy when the enemy is directly south of it, and it may somehow become familiar with the map on which it is learning, not learning to really avoid walls, but only learning to stay in particular coordinates on that particular map. Because of these problems, we decided to make a new module for this research that normalizes coordinates relative to the AI controlled ship (self ship). The angles are all recalculated with 0 degrees as the direction of velocity of the self ship. In this way we were able to not only greatly simplify the input to the AI, but also allow for the AI to learn behavior that is effective in universal positions on various maps with various enemy locations.

The next step was to design the environment in which the AI would learn. We decided to use a simple square map enclosed by walls. So that the AI would learn to avoid walls, wall contact at any but a very small velocity is lethal. Though it is possible to have a wide variety of weapons and ship upgrades, for this research we chose to have no upgrades and to limit the firing rate to approximately one second between shots. This results in fewer bullets to contend with and makes aiming more important.

The opponent of the AI controlled ship is a rule-based bot named "Sel" that we programmed using our AI module on another client. It is programmed to have a very good aiming function and the ability to avoid walls and bullets. Because Sel is the opponent for each in the population of AI controlled ships, it is important that Sel performs no random behavior and that it plays exactly the same from one individual to the next.

Fitness tests of the AI bot runs until it dies. If it kills Sel, both bots are reset to run again from the same location. The behavior of both Sel and the AI controlled bots lack enough randomness that endless repetitive behavioral loops became a problem. The same AI individual would kill Sel over and over again, simply because they were each flying exactly the same path every time. To fix this, we have Sel turn and thrust at the start of each run at an angle determined by the number of times in a row that he

has been killed by that individual. In this way, with every individual Sel starts out the same, but varies as he is killed by that same individual, so that the evolving individual never enters an endless loop of killing, except by great skill.

Another problem, particularly with the graphics-driven Xpilot, is the preprogrammed limit to game speed. A normal Xpilot server runs at 16 frames per second (FPS), which accommodates human vision and reaction times. The AI controlled clients can play at a speed faster than humans can play, but are still limited by the server to 100 FPS. The Xpilot clients also load the graphics for the game, which can make the game sluggish. On our Sunblade 100s, the clients can only run consistently at 32 FPS, only about double the speed of normal play. This means that the evolution process takes many days. In the future we plan to make an option to disable the client graphics and to remove the 100 FPS limit on the server.

3 The Controller

The AI ship uses a single layer feedforward neural network to control its movements (Figure 2). There are twenty-two inputs and three outputs. The inputs are:

- Self Heading: degrees from velocity direction.
- Self Velocity
- Self Reload Time: frames left until self can fire a shot.
- 8 Wall Sensor Inputs: the AI ship has 8 sensors to check the distance to the nearest wall at the angle of the sensor. Each is sensing at evenly spaced angles, with the ship's velocity direction being North, and from that there is NW, NE, W, E, SW, SE, and S.
- Enemy Distance from Self
- Enemy Direction from Self
- Enemy Heading: direction enemy ship is pointing.
- Enemy Change in Distance from Self
- Enemy Change in Direction from Self
- Enemy Reload Time
- MDB (Most Dangerous Bullet) Distance from Self
- MDB Direction from Self
- MDB Change in Distance from Self
- MDB Change in Direction from Self
- Threshold: Inputs always fully enabled

All inputs are between -1 and 1. In the case of an angular input, 0 degrees is represented as 0.0, 179 degrees as 1.0, and 181 degrees as -1.0, with all other angles falling between -1.0 and 1.0, proportionately. The distance, time, velocity, and change values are calculated by dividing the actual input by the maximum possible value. For instance, a wall sensor input is divided by the maximum distance the sensor can check for walls, so that distant wall input is almost 1.0, but closer is near 0.0. The threshold node always has an input of 1.

Between these twenty two inputs and each of the outputs is an evolved floating point weight, valued between or equal to -1.0 and 1.0. The sum of each input multiplied by the weight is used to determine the output of three output control nodes:

- Thrust: either true or false. If the input to this node is above 0, true, otherwise, false.
- Shoot: either true or false, just like Thrust.
- Turn: for the sake of realism, we decided to limit turning to 15 degrees or below per frame. To determine the turn rate between -15 and 15, the input sum is translated to a floating point value between -1.0 and 1.0, which is then multiplied by 15. The equation to determine the floating value from the sum of inputs is shown in Equation (1). If *output* is greater than 1.0, it is set to 1.0, and if less than -1.0, to -1.0.

$$output = \tfrac{1}{4} \times input_sum \qquad (1)$$

This function reduces the input sum by ¼ so that a broad domain of *input_sum* values, between -4.0 and 4.0, will turn the ship less than maximum, yielding output between -1.0 and 1.0, allowing for finer precision in turning.

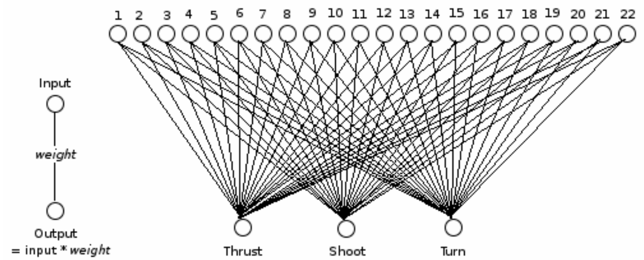

Figure 2: Artificial Neural Network with 22 inputs discussed in Section 5 and three outputs. Thrust and Shoot are either on or off, while turn can have a value between -1 and 1, which corresponds to a turn of -15 and 15.

4 The Learning System

The learning task to evolve weights for the NN is done with a Genetic Algorithm [8]. Each weight is represented by a six bit binary number in the chromosome. The value for the weight is derived by Equation (2).

$$weight = 2.0 \times \left(\frac{gene}{64.0} - 0.5 \right) \qquad (2)$$

The weight is a value in the range [-1.0, 1.0]. There are 22 inputs and 3 outputs (Figure 2), with a unique weight between each input and output, so that there are 66 total weights yielding a 396-bit chromosome (Figure 3).

We use a standard GA where each individual of the population is tested, assigned a fitness, and selected using the *roulette wheel* method [8]. Two point crossover is used after two unique individuals are selected. Bit by bit mutation is used with a 1/600 chance of flipping for each bit. We use a large population of 512 individuals because smaller populations sometimes suffer from premature convergence.

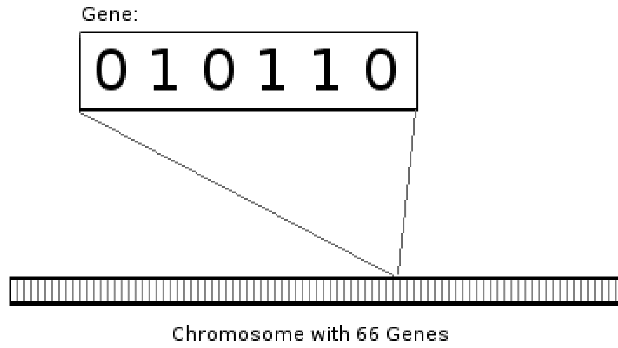

Figure 3: The chromosomes used to represent neural network weights during evolutionary computation.

Many attempts were made to find a near-optimal fitness function.

- First we chose to judge the fitness by how many frames the bot survived, giving no bonus points for killing the enemy bot. This evolved pacifist bots who became quite good at dodging the enemy's bullets and avoiding walls, but made little attempt at aggression.
- We tried giving a large bonus to the individuals who killed the enemy. This evolved a population that did not bother to move or respond to enemy bullets, but rather shot as desperately as possible at the enemy bot, so that they could gain the precious bonus.
- In the end we settled on giving a small bonus of 20 points to the fitness for every kill. This is not much, considering a typical span of life for an individual can be over 200 frames. To help differentiate between fitnesses of individuals, we also squared the fitness for selection.

5 Tests and Results

We ran five tests on separate populations of 512 individuals, for 256 generations each. After each generation, the starting location was switched to a different location on the map (one of 10 starting positions defined to be distributed throughout the map). This appeared to provide sufficient variety in the starting distances and relative bearings of the enemy ship and surrounding walls. Because some starting locations are closer to the enemy's starting location, or closer to the walls, and some are in safer locations, there is variation in the average fitness of the population from generation to generation. The difficulty of the starting location of a particular generation significantly influences the performance of all individuals. However, the starting locations are recorded for every generation so that the change in fitness over time can be determined.

Even in the early generations, due to the numerous factors influencing survival time, there was a good chance for at least one individual out of the 512 to be “lucky” and survive for a long period despite having no useful combat skills. The genes of such lucky but unskillful individuals are eventually weeded out by having such a large population and changing starting positions every generation, but because they did often exist in a brief moment of prolonged glory, graphing the best fitness for each generation yields a random looking data plot with no discernable trend lines. Therefore, we use the average fitness of the populations in successive generations to gauge the overall change in health and fitness of the populations. The graph (Figure 4) of the average fitnesses of the five populations still has a random appearance, but it shows an overall trend in increasing fitness.

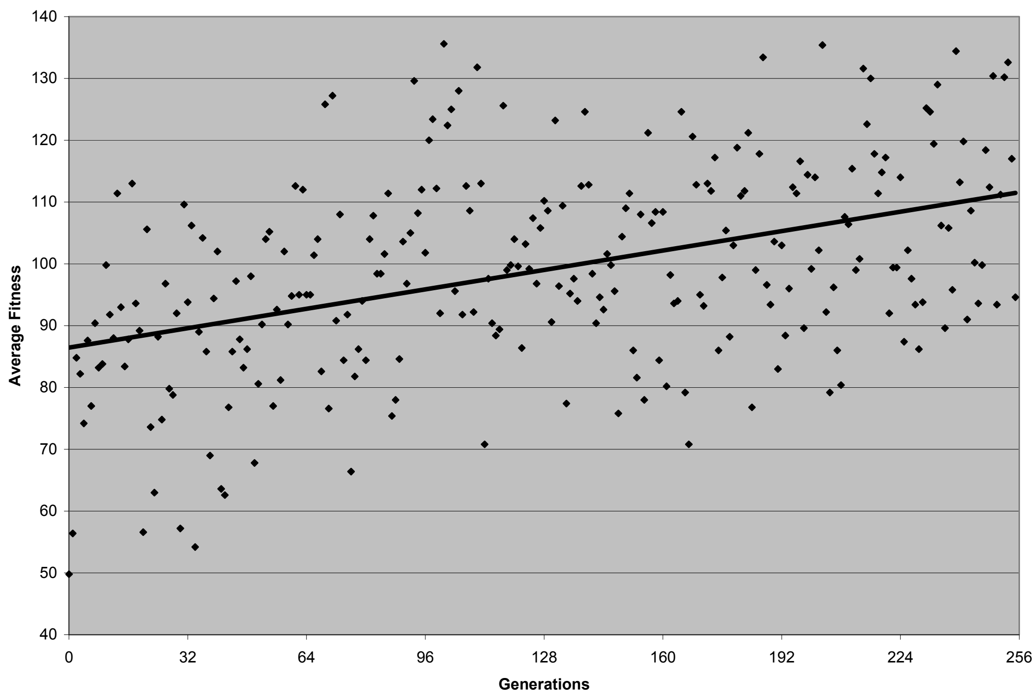

Figure 4: Graph showing average fitness of the five populations over the generations, with a trendline (least squares) showing statistical improvement.

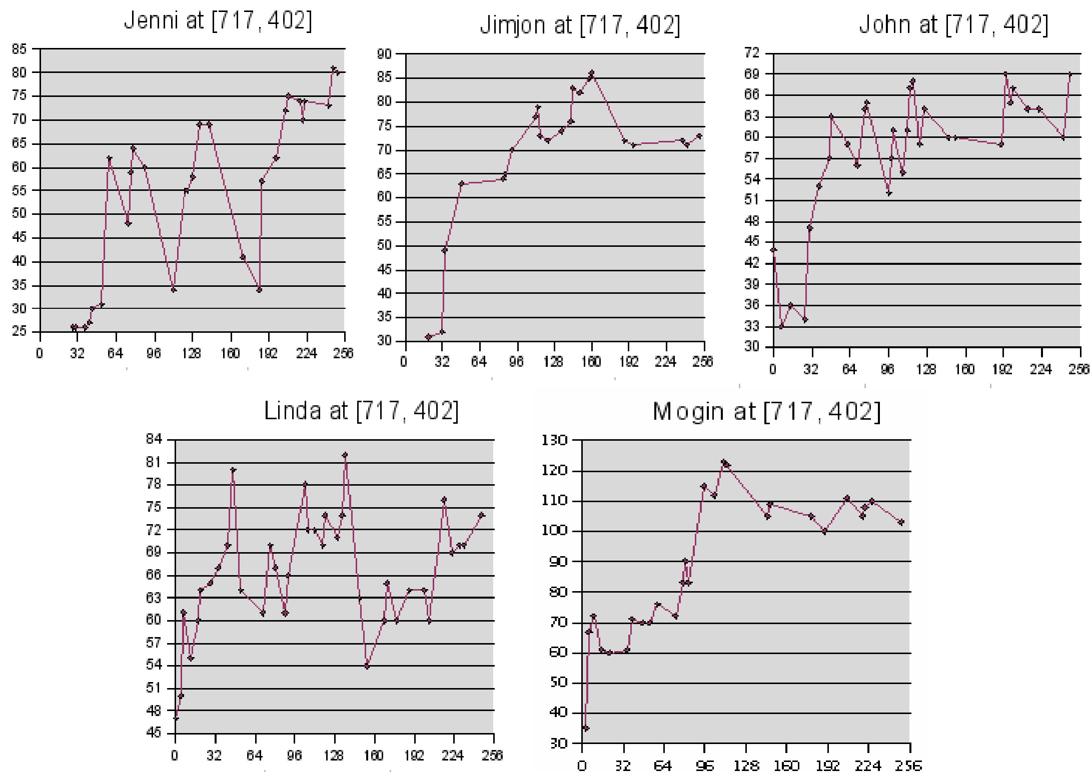

Figure 5: Average fitness (Y axis) of Jenni, Jimjon, John, Linda, and Mogin at the same starting position over the 256 generations (X axis).

To clearly see how the GA affects the fitness of the NN controllers we look at graphs of the average fitness over one particular starting position (Figure 5). These graphs show the average fitnesses of trials at the [717, 402] starting location for each of the 5 test runs. There are still many dips and rises in the fitness for individual locations. This is because between one generation at a particular starting location and the next generation at that same starting location there are on average 9 generations that have had trial runs at different starting locations. During these between generations, the population may adopt strategies that improve survival at intermediate starting locations, but lead to an early death at the original starting position. Consequently, the average fitness may decrease when the generation is run from that same starting location again.

If the populations are developing general combat skill, they should display a trend in increasing average fitness from the majority of starting locations. In the graphs of starting location [717, 402], seen in Figure 5, all populations have a clear trend in increasing average fitness. Although Linda's learning trend is somewhat questionable from this location, she does show increasing fitness trends from other starting locations. This trade-off tendency is common among all individuals. In other words, each bot clearly improved in average fitness over the 256 generations, both in comparison of fitnesses and also as seen by visual observance of their behavior.

Each of the five tests evolved unique strategies and behavior. For convenience, we named each bot. Here are descriptions of the observed behavior developed by each of the five tests:

Mogin developed a very optimal solution for survival. He immediately begins turning to the right and thrusting, so that he flies around in small circular loops. By turning faster or slower at certain parts of the loops he is able to move toward or away from things, mainly away from the deadly walls, and either towards or away from the enemy bot, and away from bullets. By thrusting constantly in a circle, it is very difficult for the enemy bot to aim a shot to hit him. Mogin *appears* to just shoot as often as possible, as if not aiming, but his ability to survive longer and kill the enemy bot more often increased to the last generation, so it appears likely that he did acquire some skill in aiming.

John, **Linda**, and **Jimjon** all learned similar strategies. They spin nearly constantly, slowing down their turn speed occasionally and rarely thrusting, but shooting nearly as often as possible. They appear to have some ability to dodge bullets, thrusting sometimes to move out of a bullet's path. They each seem to have gained some ability to aim.

Jenni developed an interesting swooping attack behavior. She will often swoop around the back of the charging enemy bot and shoot at it. On her best runs, she will fly loops around the enemy bot in wide circles shooting inwardly at it. Figure 6 shows two images displaying her swooping behavior.

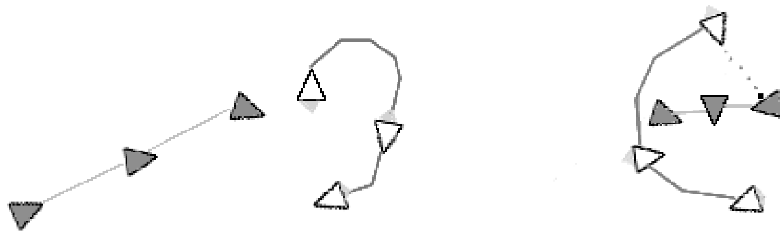

Figure 6: Two frames of an attack by the bot Jenni. The first frame shows the enemy (filled) charging her as she maneuvers to avoid being shot. In the second frame, she completes the loop and fires a deadly shot at the enemy.

6 Conclusions

Xpilot has proven to be an excellent platform for developing an AI system in this experiment. It can be configured so that there are minimal control choices with a surrounding environment that is not complex, yet competent play requires intelligent behavior. Its frame-synchronized client/server multiplayer design is ideal for allowing a great variety of experiments in artificial intelligence. Although our current implementation is limited to 100 FPS by the server and limited by the client's in-game graphics, future development plans include eliminating these limits.

Through the use of a simple neural network controller and a genetic algorithm to evolve the connection weights, we were able to develop a successful learning system for autonomous agents engaged in combat. This was not a system designed for learning the best behavior for a specific starting position, but one that learned general strategies for offense and defense. To this end, we changed starting locations for each generation. This method was successful, but made plotting the change in fitness difficult because of the large variance in the difficulty of starting locations. However, the increase in overall fitness is clearly seen by looking at the trend in the graphs of average fitness of all five populations and at the graphs of fitness changes related to a specific starting location, plus can be observed by simple visual observation of each bot's behavior.

There are many possibilities for future research in using Xpilot for autonomous agent learning. Other learning paradigms and/or control systems can be investigated in this environment. An extension of our current system through incremental learning can be used to solve problems of increased complexity. Once we are able to run evolution without the game speed limitations, we can use punctuated anytime learning, where the learning is done on a simulator with periodic tests on the actual system, to improve learning for changing enemies and environments. This could lead to using the internet to allow our bots to learn against human opponents. We have started research using competitive co-evolution, where one GA learner evolves against another GA learner, but much more needs to be done in this area. Finally, we hope to use methods developed in previous research to evolve cooperative behavior for a team of Xpilot bots to learn strategies in defeating an opponent team.

Bibliography

[1] D. Fogel, (2002). *Blondie24: Playing at the Edge of AI,* Morgan Kaufmann Publishers, Inc., San Francisco, CA.

[2] G. Konidaris, D. Shell, and N. Oren, (2002). "Evolving Neural Networks for the Capture Game," *Proceedings of the SAICSIT Postgraduate Symposium,* Port Elizabeth, South Africa, September 2002.

[3] P. Hingston and G. Kendall, (2004). "Learning versus Evolution in Iterated Prisoner's Dilemma," *Proceedings of the International Congress on Evolutionary Computation 2004* (CEC'04), Portland, Oregon, 20-23 June 2004, pp 364-372.

[4] N. Cole, S. Louis, and C. Miles, (2004). "Using a Genetic Algorithm to Tune First-Person Shooter Bots," *Proceedings of the International Congress on Evolutionary Computation 2004* (CEC'04), Portland, Oregon, pp 139–145.

[5] G. Yannakakis and J. Hallam, (2004). "Evolving Opponents for Interesting Interactive Computer Games," *Proceedings of the 8th International Conference on the Simulation of Adaptive Behavior (SAB'04); From Animals to Animats 8*, pp 499-508.

[6] P. Funes and J. Pollack, (2000). "Measuring Progress in Coevolutionary Competition," *From Animals to Animats 6: Proceedings of the Sixth International Conference on Simulation of Adaptive Behavior.* pp 450-459.

[7] P. Funes. "XPILOT, A Differential Game and a Learning Game Bot."

http://www.cs.brandeis.edu/~pablo/xpilot2.ps

[8] D. Goldberg (1989). *Genetic Algorithms in Search, Optimization and Machine Learning*, Addison-Wesley, Reading, Ma.