Using a Queue Genetic Algorithm to Evolve Xpilot Control Strategies on a Distributed System

Matt Parker, *Computer Science, Indiana University,* matparke@indiana.edu
Gary B. Parker, *Computer Science, Connecticut College,* parker@conncoll.edu

***Abstract*— In this paper, we describe a distributed learning system used to evolve a control program for an agent operating in the network game Xpilot. This system, which we refer to as a queue genetic algorithm, is a steady state genetic algorithm that uses stochastic selection and first-in-first-out replacement. We employ it to distribute fitness evaluations over a local network of dissimilar computers. The system made full use of our available computers while evolving successful controller solutions that were comparable to those evolved using a regular generational genetic algorithm.**

I. Introduction

XPILOT is an internet based 2D interactive space combat game, which is a good environment for evolving complex multi-objective control systems. In past work, we used a standard genetic algorithm (GA) to learn the weights for a simple neural network (NN) controller. In our present research, we are using incremental learning to evolve more sophisticated controllers than those of the past. This process has required a continual cycle of design / evolve / evaluate to develop competent components for the control strategies. However, since the evaluations of the controllers are inordinately long, this process is very slow on a single computer. In future research, we hope that evolved agents will be evaluated in the real world, competing against human players in different arenas. In an effort to speed up our current development process and additionally address our future need for real-world evaluation, we implemented a method for evolving Xpilot controllers on a distributed system. The method is based on a steady state genetic algorithm (SSGA). A queue is used to store individuals of the population with new children added to the front and the oldest individual removed from the tail of the queue.

In a SSGA only one or two individuals are replaced in a population at each iteration. These new individuals become part of the population and are now available for selection. This is in contrast to the standard GA where the entire population (with the possible exception of an elite group of individuals carried over from the previous generation) is replaced each iteration (referred to as a generation in this case). Although Holland discussed this idea in his original work [1] and De Jong evaluated the properties in follow-on research [2], it was not until Whitney developed GENITOR [3] that "steady state" became of increased interest in genetic algorithms. GENITOR uses rank-based selection where parents are picked based on their fitness ranking as opposed to their proportional fitness where an individual with a fitness of $2x$ has twice the probability of being selected as an individual with a fitness of x. GENITOR produces one offspring at a time and this offspring replaces the least fit individual in the population. This means of learning was found to be very effective on several problems as this replacement strategy aggressively pursues an optimal solution. However, high selection pressure can result in premature convergence. Large population sizes may be needed to avoid this possibility.

In their paper describing studies on the overlapping generations of SSGAs [4], De Jong and Sarma use a SSGA with a FIFO deletion strategy to better approximate the results of a standard GA. This helped to reduce the variance of the final solutions (high variance results in allele loss [5]) and resulted in a higher best-individual-takeover than SSGAs using uniform replacement (replace an individual selected at random).

SSGAs offer a reasonable method for executing GA learning on a network of computers. They have been used by other researchers for this purpose. Whitney and Starkweather [6] and Takashimar et al. [7] used a SSGA in the implementation of a parallel GA that uses the island model of computation. In the island model, processors are assigned subpopulations, which share genetic information through migration. We did not use an island model; the fitness evaluation time precludes the use of large populations. Runarsson [8] used an asynchronous parallel evolution strategy for multiprocessor machines. Our system is distinct from this work in that we used GAs instead of evolution strategies and we used FIFO replacement instead of tournament selection/replacement. In addition, our implementation is unique in that it is applied to learn controllers for an interactive network computer game.

In this paper, we use a SSGA that uses FIFO replacement to learn control strategies for Xpilot. Since the population is stored on a queue to implement FIFO replacement, we will refer to the algorithm as a queue genetic algorithm (QGA) in this paper.

II. XPILOT

Xpilot is an open-source multi-player networked two dimensional space combat simulator in which a player controls a ship in varieties of free-for-all combat, team combat, and capture-the-flag game-play. Although the main controls are few—turn, shoot, and thrust—the strategies and maneuvers for which these simple controls are employed can be very complex and separate poor from successful playing styles.

Xpilot uses a client/server model to allow for multiple players. The server generates the map, calculates game physics, receives input from clients, and sends game data to the clients. A client captures information from the player keyboard and mouse input, sends it to the server, and displays information received from the server, such as ships, bullets, walls, and player scores. The server and client are synchronized frame by frame. In recent work, we modified the Xpilot client by adding an interface to control the Xpilot client ship with an artificial intellence agent (AI). We created a module of Xpilot with structures containing useful information about the player's ship, the enemy ships, the bullets, the radar, and the map, which were parsed after being received by the client and stored into variables. In addition, commonly used functions for aiming, calculating distances, finding barriers, etc. were added to the AI module. The AI module uses the input information to determine the AI's behavior (thrust, turn, and shoot) for the next frame.

III. DITRIBUTED SYSTEM

Because Xpilot is a real-time multi-player game, all clients must synchronize with each other and with the server, running the game world at a constant rate. Since a significant amount of time is necessary to evolve an AI, it is desirable to do the fitness computations as fast as possible. However, Xpilot was built for human players, who can only control their ships at a slow game world speed, so it has natural limitations on the speed of the simulation. In past work, we increased the possible speed of the game by running the display in a low color-depth video buffer rather than through a real video card and removed the 100 frames-per-second limit built into the game. However, between each frame we often require enough computation for the ship controller that running the game at a high frame rate often results in dropped frames due to lost coordination between the server and client. The Xpilot server runs at a constant rate, according to the CPU clock of its host computer. If a client does not send any data to control its ship in a frame, due perhaps to unfinished controller computation, the server cannot wait for that client to send the data since other clients may be connected to the server and they must not be forced to wait as well. If a client drops frames, it slightly changes the control of the ship and results in a controller that does not work properly at normal speeds. Due to the fact that the server does not wait for slow clients and as the game play varies the client's required computation time between frames also varies, it is necessary to have a large buffer of excess computational power so that at computational peaks a client will not lose any frames. Therefore, running a single simulation of Xpilot does not take full advantage of the capabilities of the computer on which it runs. The evolution is slow and it takes days to evolve a controller.

To increase the speed of fitness computation for Xpilot we distribute the evolution of a single GA over multiple instances of Xpilot simulations. A single genetic algorithm server distributes chromosomes to clients running an Xpilot simulation (Xpilot server and client), receives from the clients the fitness of the chromosomes, and evolves the population. We are able to run multiple Xpilot simulations locally on one computer and/or distribute them across multiple computers over the network. These clients are able to run their simulations at manageable frame rates according to the capabilities of their local host computers. A single computer distributing multiple simulations is now able to safely utilize more of its processing power. Evolution that once took days can now be accomplished in a few hours, making Xpilot a more feasible platform for testing learning methods.

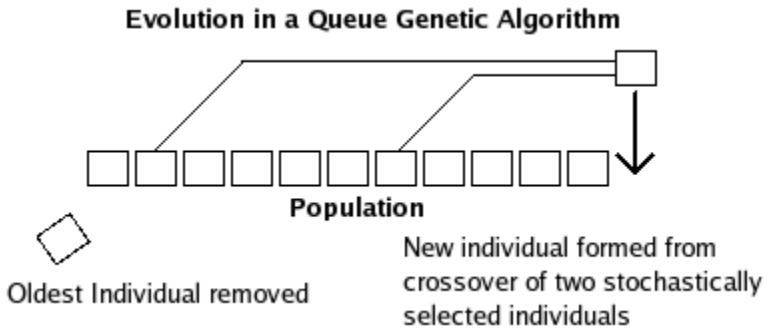

Fig. 1. The evolutionary scheme of a QGA. The oldest individual is removed from the queue as a new individual is formed from crossover of two stochastically (roulette wheel) selected individuals from the population, and placed at the top of the queue.

IV. QUEUE GENETIC ALGORITHM

A regular genetic algorithm (RGA), which tests an entire population, assigns fitnesses, and recombines (crossover and mutation) the individuals to form a new population, is effective for evolution but is often difficult to distribute over several processors. If the fitness functions take varying amounts of time to evaluate, managing the returning evaluations often means either waiting for the last individuals of a population to evaluate, or sending out more chromosomes than necessary and wasting unfinished evaluations when enough have returned. The server which controls the GA must often remember the number of clients connected and juggle the timing of the computations. Because the length of fitness evaluations in Xpilot can vary by several seconds, it is not desirable to use an RGA as a distributed server. As an alternative, we use a steady-state first-in-first-out genetic algorithm, which we refer to as a queue genetic algorithm (QGA). This solution allows us greater versatility in our distribution and offers the advantage of an evolution scheme that is equal in effect to an RGA.

The QGA's structure (Figure 1) is a queue in which new individuals are formed from crossovers of stochastically (roulette wheel) selected individuals from the population while the oldest individuals are removed. The queue is the size of the desired population and consists entirely of individuals who have already been tested for fitness. When a client is available, two individuals are chosen stochastically according to fitness from the population, their chromosomes are crossed over to form the new child. The child is sent to the client where it is tested for fitness and placed at the beginning of the queue. The oldest individual is removed, and each individual moves down one slot closer to the end where it will eventually be removed. If the size of the population is p, each individual has p chances to be chosen as a parent of a new child. Because each individual has p chances to reproduce, the QGA is very similar in evolutionary behavior to an RGA, which performs p recombinations to form a new population after every individual in the generation is evaluated.

To begin a population, as in an RGA, a QGA population is filled with random individuals. Because a QGA only places an individual on the queue after its fitness has been evaluated, the QGA must form the random individuals and test them to build the queue to the size of the population. If the QGA builds a population entirely of random individuals and commences with its standard evolutionary cycle of recombining new children and removing the oldest in the queue, then the oldest individual will have only one opportunity to be selected as a parent for a new child. The second oldest will have only two opportunities, the third, three, and so on, and the last randomly created individual will have the most opportunities to reproduce. The evolution would favor the later formed individuals with little regard to their fitnesses. However, if while the random population is formed, the early individuals are given opportunities to reproduce offspring in the place of producing a random individual, the initial population will lack diversity.

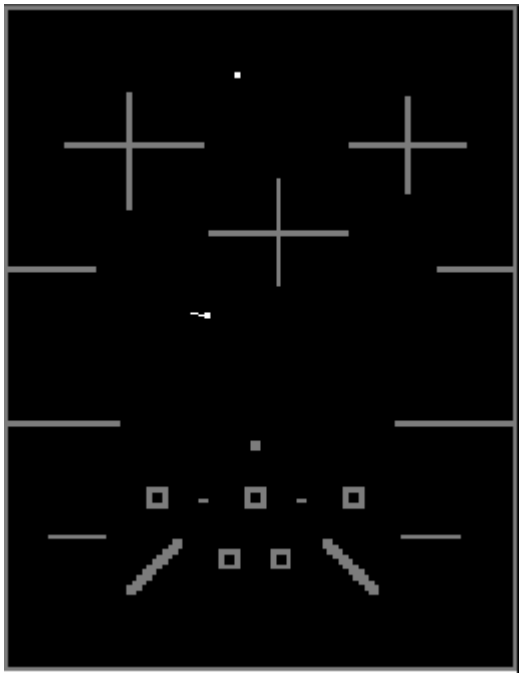

Fig. 2. Map which the evolving ship must learn to navigate to get close to target ship. The evolved ship is the white dot in the center of the screen with a white directional line. The target ship, which does not move, is at the top of the map. It cycles its position between the top of the map and the bottom. The other lines are walls or obstacles.

To create a population with diversity in which there is negligible advantage for an individual according to the time of its creation, the initial queue is filled to at least double the target population. In addition, we use probabilities to determine how each new individual will be produced. Given a target population size of p, a current population size of n, and an initial population size of $2p$, while $n < 2p$, there is a $n/2p$ chance that the newest formed individual will be a recombination of stochastically selected parents from the current n sized population. Otherwise, a new random individual will be created. This gives roughly equal opportunity to all newly formed individuals to reproduce, while still forming approximately p random individuals. When $n=2p$, any new individuals formed will be children formed from crossover, and in order to reduce the size of the population back to p, two rather than one of the oldest individuals are removed for every new individual formed.

QGAs are naturally easy to distribute. The population of the QGA is steady state and always made of the most current individuals, so the QGA server may form any number of new chromosomes on demand from its single population without keeping a saved history of an older population, as may be necessary in an RGA. The QGA server can create and send new chromosomes to the QGA clients, receive

back the chromosomes with the fitnesses, add them to the population, and drop the oldest individuals. This can be done at any rate and to any number of clients; all without remembering the number of individuals sent out or the number received. It is even possible to have more clients testing individuals than there are individuals in the population. Each returned individual still has the same number of chances to mate, no matter the rate at which individuals are being created, dropped, or added.

The QGA server is implemented in such a way that after starting it waits for QGA clients to connect. When the server first starts, it either loads a previously saved population or creates a new one. If a client connects, the server sends to the client a chromosome; either a randomly created chromosome, or a crossover from two selected individuals in the population. Once the server sends away the new chromosome it forgets it, erasing it from memory. After the client has tested the chromosome to attain fitness, which in our case is done in an Xpilot simulation (made up of an Xpilot server and client), it sends back the chromosome and its fitness to the QGA server. The server adds the new individual to the queue and drops the oldest individual. It then immediately forms a new chromosome and sends it out to the client from which it received the latest individual with fitness. Because the server does not add the chromosome to a population until after it has been tested, there is no problem if a client must quit and never returns the chromosome. All that is lost is a new chromosome, which can easily be generated again.

Because of this great versatility and simplicity, we are able to easily distribute our Xpilot evolution. We can add and drop clients to a QGA server whenever we wish. We can run as many clients as we would like, increasing the speed of evolution numerous times over. We can also run the clients at different frame rates, depending on the capabilities of the clients' local computers. Connected to the same QGA server, we can run 30 clients in the background on our cluster running at 100 frames per second, 5 clients on a dual processor Xeon running at 64 frames per second, and a single client on a desktop computer in the lab running at 32 frames per second with a graphics display to view the progress of the evolution. Our QGA distribution allows great speed with versatility, and the simple design makes it robust and easy to use.

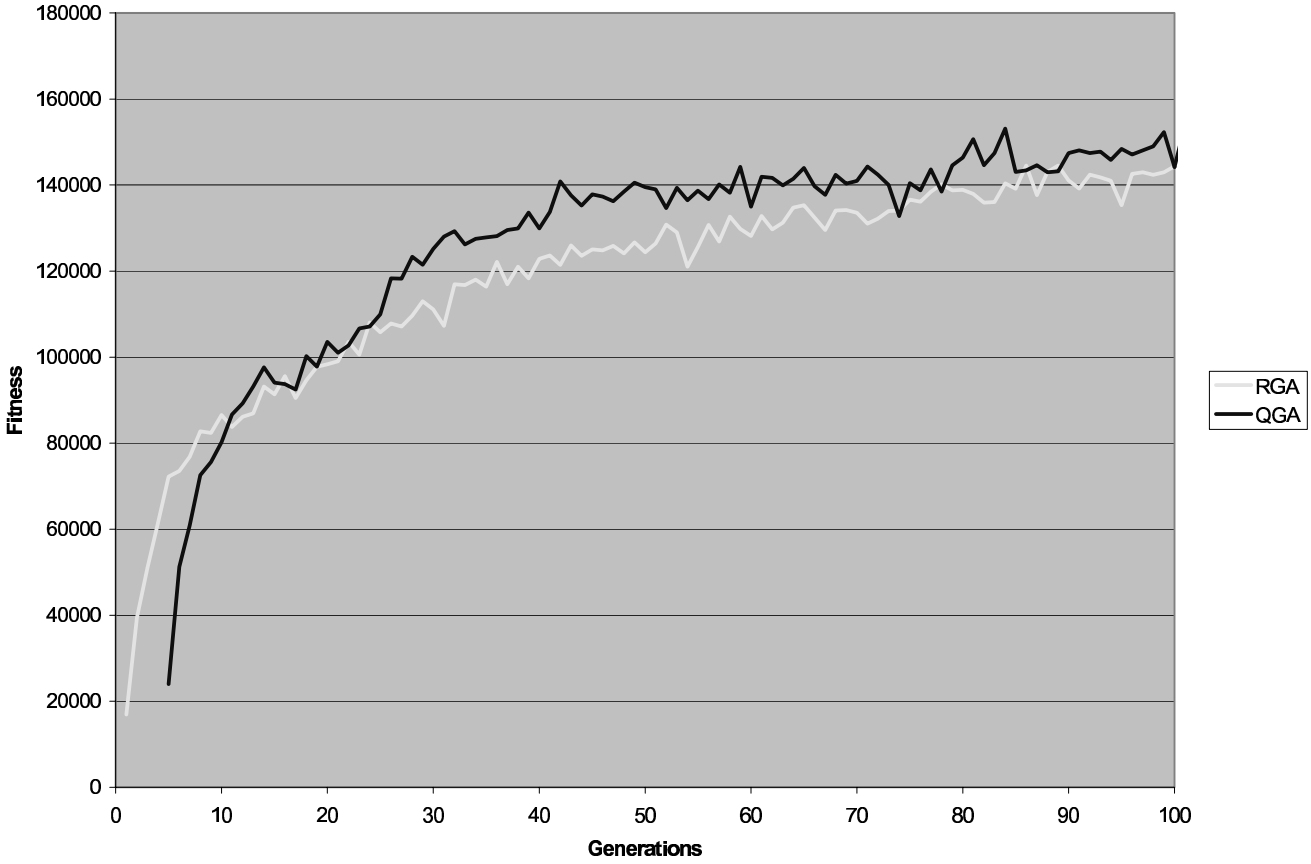

Fig. 3. Results showing similar improvement between the distributed Queue GA and the Regular GA. This is the average of the average fitness of the five tests for each evolutionary method.

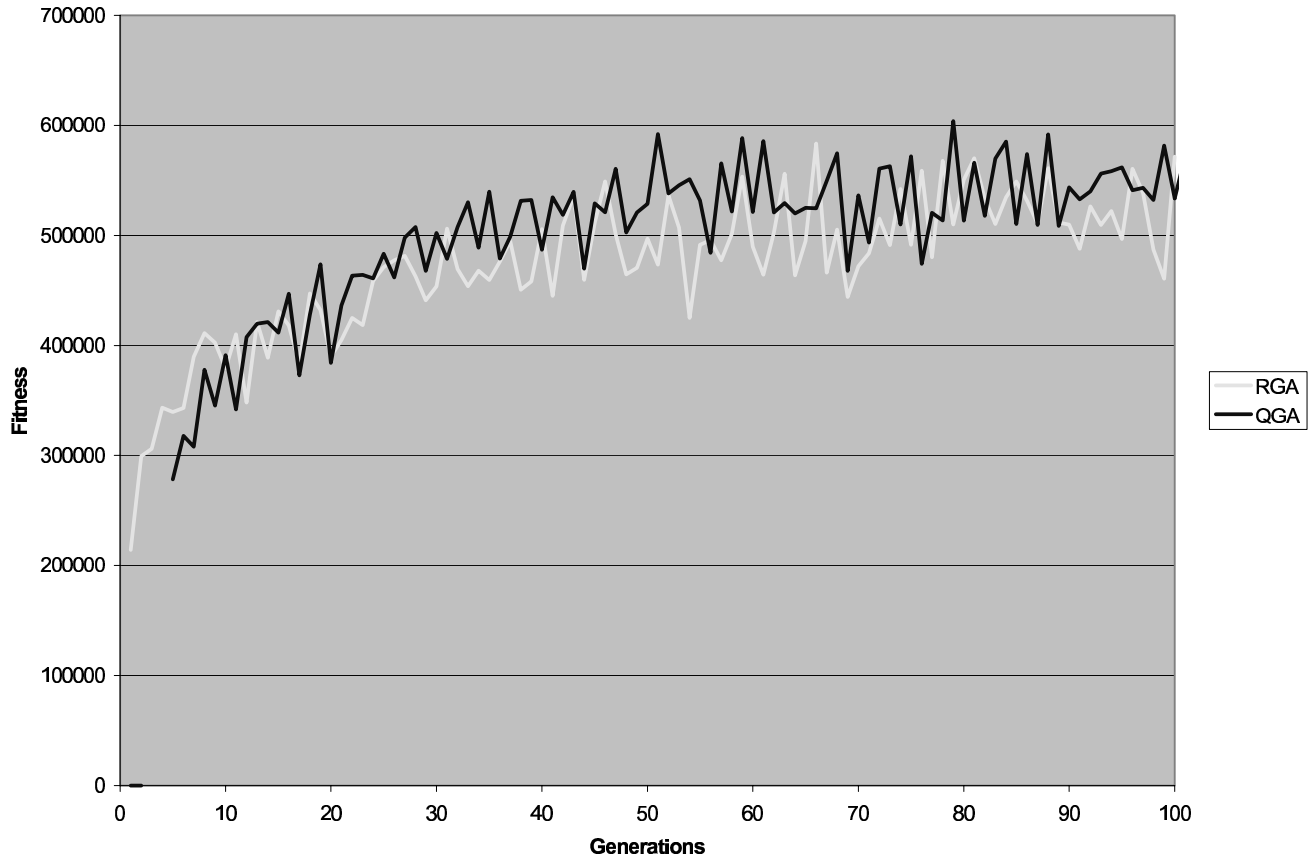

Fig. 4. Graph of the average of the best fitness over the generations for each of the five tests for each evolutionary method.

V. TESTS AND RESULTS

For this paper we tested a particular problem in Xpilot with both a distributed QGA and an RGA, in order see if the QGA evolution was as effective as that of the RGA.

We chose to evolve a controller for a ship that was to navigate the ship through obstacles to fly as near as possible to a target ship. The controller is a single-layer neural network with 16 inputs going to two outputs, with 32 weights in between. Each weight is a real number with a value between -1.0 and 1.0 and is derived from an 8 bit gene from the chromosome. Since there are 32 weights, the length of the chromosome is 256 bits. The inputs consist of information about the distance of walls surrounding the ship, the ship's own velocity and heading, the direction to the target ship, etc. One of the outputs is for thrust; the ship will thrust if the output is greater than or equal to zero. The other output controls the turn of the ship, which can be anywhere from -15 to 15 degrees per frame. The output of the turn node is mapped onto this range.

The evolving ship starts at a random location in the center of the map (Figure 2) and the target ship starts either at the top or the bottom of the map. Each individual in the population lives two lives, once where the target ship is located to the north, and once where it is to the south. The walls are lethal if crashed into at anything but a very slow speed. The layout of the walls differs from north and south in order to make the agent learn a more robust navigational controller. The agent is given a reasonable amount of time to fly to the target ship. If the target ship is reached, it will fly a little ways away to make the agent learn to follow. Fitness is awarded the agent according to how close it flew to the target ship from its original location, summed over every frame of game-play. It is also given bonus points for every frame that it is located right next to the target ship, and to encourage it to fly carefully its score is doubled if it stays alive during the whole allotted run time-limit. The fitness is summed over the two runs, north and south.

We ran five tests with a distributed QGA and five tests with an RGA, each at 64 frames per second with a population of 256. We set the QGA to build up an initial population that was four times (rather than two times) its target population of 256 to increase initial diversity and then decreased it down to 256 (removing two oldest instead of one). Figure 3 shows the average of the average fitnesses of the five test populations for the RGA and the QGA and Figure 4 shows the average of the best fitnesses for each generation of each of the five tests for the QGA and RGA. Because there are no discrete "generations" in the QGA, we determine them by each time p number of individuals have

been tested, where p is the actual size of the population. Because p first expands out to four times the target population, the generations are first read at 1024 and then 512 individuals, after which every generation is made of 256 individuals. For this reason, we start the graphs of the QGA 4 generations later than the RGA.

With both the RGA and the QGA the agents evolved similarly successful strategies to fly close to the target ship. Their best and average fitnesses are very similar, with the QGA's average fitness being slightly higher, which could be luck or because at the start the QGA built up to a larger initial population, allowing for greater diversity. The real advantage with using the distributed QGA is that the evolution was finished in a matter of hours rather than a few days, as it took the single RGA tests. By using a distributed RGA we could have sped up the time of evolution, but not to the extent of the QGA and not with the same flexibility for distribution.

VI. CONCLUSIONS

The Queue Genetic Algorithm has proved to be an effective evolutionary method, with results very similar to the evolution of a Regular Genetic Algorithm. The design of a QGA is such that it may be easily distributed amongst numerous computers, and its simplicity makes it robust enough to handle varied client capabilities and network limitations. When evolving agents in Xpilot, a game that can not be run fast enough to utilize the full processing capabilities of a computer and that takes varied time to test agents, a distributed solution is necessary for timely evolution results. A distributed QGA has been ideal for evolving one population over multiple Xpilot simulations and in our test to evolve the weights in a neural network to control an agent to navigate towards a target ship, the QGA yielded behavior and fitness extremely similar to our same test performed with an RGA.

In the future we plan to continue using the QGA to evolve tests, as well as to create "chains" of Xpilot simulations, where individuals will be tested in several different maps and arenas. We hope to be able to eventually send individuals out into a public Xpilot server to play against human players. In further research, we intend to test the QGA on other problems requiring simulated evolution. A distributed QGA is an effective learning method that significantly increases the speed of evolution and can be used in several evolutionary computation applications.

REFERENCES

[1] Holland, J. (1975). *Adaptation in Natural and Artificial Systems*, The University of Michigan Press.

[2] De Jong, K. (1975). *An Analysis of the Behavior of a Class of Genetic Adaptive Systems*, Doctoral Thesis, Department of Computer and Communication Sciences, University of Michigan, Ann Arbor.

[3] Whitley, D. (1989). "The GENITOR Algorithm and Selection Pressure: Why Rank-Based Allocation of Reproductive Trials is Best," *Proceedings of the Third International Conference on Genetic Algorithms*.

[4] De Jong, K. and Sarma, J. (1993). "Generation Gaps Revisited," *Foundations of Genetic Algorithms 2*, San Mateo, CA.

[5] Sarma,J. and De Jong K. (1997). "Generation Gap Method," *Handbook of Evolutionary Computation,* Oxford University Press, New York Oxford, C2.7.

[6] Whitley, D. and Starkweather, T. (1990). "Genitor II: a Distributed Genetic Algorithm," *Journal of Experimental & Theoretical Artificial Intelligence* Volume 2 , Issue 3, July/Sept. 1990, pp. 189 – 214.

[7] Takashima, E., Murata, Y., Shibata, N. and Ito, M. (2004). "Techniques to Improve Exploration Efficiency of Parallel Self Adaptive Genetic Algorithms by Dispensing Synchronization," *Proceedings of the Fifth International Conference on Simulated Evolution And Learning (SEAL2004)*, October 2004.

[8] Runarsson, T. (2003). "An Asynchronous Parallel Evolution Strategy," *International Journal of Computational Intelligence and Applications*, Volume 3, Number 4, pp 1-14, 2003.

[9] Parker, G. and Rawlins, G. (1996). "Cyclic Genetic Algorithms for the Locomotion of Hexapod Robots," *Proceedings of the World Automation Congress (WAC '96), Volume 3, Robotic and Manufacturing Systems*. May 1996.

[10] Parker, G. and Georgescu, R. (2005). "Using Cyclic Genetic Algorithms to Evolve Multi-Loop Control Programs," *Proceedings of the 2005 IEEE International Conference on Mechatronics and Automation (ICMA 2005),* Niagara Falls, Ontario, Canada, July 2005.

[11] Parker, G., Parker, M., and Johnson, S. (2005). "Evolving Autonomous Agent Control in the Xpilot Environment," *Proceedings of the 2005 IEEE Congress on Evolutionary Computation (CEC 2005),* Edinburgh, UK., September 2005.

[12] Parker, G., Doherty, T., and Parker, M. (2005). "Evolution and Prioritization of Survival Strategies for a Simulated Robot in Xpilot," *Proceedings of the 2005 IEEE Congress on Evolutionary Computation (CEC 2005)*, Edinburgh, UK., September 2005.