Generation of Unconstrained Looping Programs for Control of Xpilot Agents

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

Abstract— **An unconstrained cyclic genetic algorithm (CGA) is presented as a means to evolve multi-loop behavior for an autonomous agent. The evolved programs control autonomous agents in the network space combat game Xpilot. Ultimately, the agent's goal is survival; it must learn to move within a restricted area, avoiding obstacles while engaged in combat with an opposing agent. The CGA learned multi-loop control programs that significantly improved the agent's survivability in the hostile Xpilot environment.**

I. Introduction

AUTONOMOUS agent control learning, important in the study of robot control, offers agents the ability to adapt and refine their behaviors to changing environments. In this paper we present the use of an unconstrained cyclic genetic algorithm (CGA) to develop a system of controls for a simulated robot (agent) in the space combat game Xpilot. The control system that was developed consists of a single large control program, but is capable of creating multiple loops of varying length through the ability to jump from any one gene to any other. This potentially allows the robot to learn multiple sequences of repeated behavior for different situations; the control system has the ability to "jump" between loops and string several sequences together. Alternatively, the CGA can maintain the entire program as a single loop, but skip portions of the program (and their commands) when certain inputs are presented. The algorithm uses as inputs information such as location and/or velocity of the agent, walls, enemy craft and incoming enemy fire.

Allen Schultz of the Naval Research Lab used a GA to evolve navigation behavior for a simulated robot; the robot learned to navigate a course of obstacles to within a certain radius of a given goal [1]. In the Xpilot program there is no physical goal, but rather the goal is survival in the presence of obstacles and enemy craft. Contact by the agent with an obstacle will result in the termination of the controlling chromosome's life in Xpilot, unlike in the Schulz experiment.

Previous research performed by Schulz did involve decision making for an agent that must avoid obstacles. Schultz used a GA to develop decisions for navigating a simulated autonomous underwater vehicle (AUV) through a mine field [2]. This simulation is more closely related to the experiment described in this paper, as contact with a mine by the AUV resulted in a failure. Though the AUV and the Xpilot robot must both avoid collisions, the Xpilot robot is faced with an aggressive enemy robot in this experiment while the AUV is constrained by a time limit and the need to find a geographic goal.

Beer and Gallagher used a GA to learn weights for a neural network used as a controller for an autonomous agent [3]. They developed a controller that was able to switch between different strategies dependent upon environmental conditions, but these strategies were not cyclical in nature.

Games often provide excellent environments for learning autonomous control. They involve problems that can be addressed in different manners with measurable degrees of success; solutions can be improved and compared to each other. Fogel explored the use of evolutionary computation in developing strategies for board games such as *checkers* [4]. Konidaris, Shell & Oren applied evolutionary computation to the game *Go*, while other researchers have addressed the *prisoner's dilemma* problem [5, 6].

Researchers have explored learning predator strategies for agents in a version of *Pac-Man* [7]. While the researchers concentrated on evolving the behavior of a group of 4 predators, the research presented in this paper aims at learning cyclical behavior for one agent under constant attack from a single enemy. The researchers using *Pac-Man* were interested foremost in what characteristics of a video game opponent would increase excitement and entertainment value; we seek to develop the most successful autonomous robot (agent) possible for the Xpilot environment. Additional examples of the use of evolutionary computation for autonomous agents include applications to decide upon weaponry and learn behaviors in the first person shooter *Counter-Strike* and how to move a light-cycle in the game of *Tron* [8, 9].

Xpilot is a particularly suitable platform for learning autonomous agent control. It simulates a two-dimensional environment, with many of the physical properties of space. Maps can be easily created and modified to introduce obstacles, enemies etc., and the agent is able to restart immediately upon termination.

Parker, Parker and Johnson applied a GA to evolve a fixed neural network (NN) to control an Xpilot bot [10]. Though the NN produced wide ranging results, an analysis of the average of several runs showed that the GA was successful in improving the performance of the NN over the course of 256 generations.

Previously we have used a GA to develop behavior for an agent in Xpilot facing the same problem presented in this paper: survival in an arena in the presence of an aggressive enemy robot [11]. We used the GA to learn the consequents to a rule based system with 16 rule antecedents, as well as to prioritize the relative importance of each rule. The result was a system that substantially improved in navigating the Xpilot environment as the GA learning progressed. A disadvantage of this system was that the robot was limited in its behaviors. Only 16 different actions were possible, and the robot could only perform one action to react to its current state, utilizing the consequent of a rule whose antecedent state was true. There was no potential for the robot to enact a strategy that took place over multiple frames.

The evolving Xpilot agent in this study has the potential for far more complex and adaptive behaviors than in previous designs. The use of a CGA allows for more possible actions by the robot, and also for the execution and repetition of sequences of actions. The enemy robot used in this study was developed by a human to harry the evolving agent; it has a high rate of fire and quickly engages the evolving agent when it appears on the map.

In addition to developing the controls for an autonomous agent, the system we developed outputs the contents of the system translated into English. This allows the agent's behavior to be analyzed; it is possible to follow the loops and branches of the system and see exactly how it behaves in every situation.

The agent used in this research is capable of learning to react in numerous situations involving obstacles, enemy fire and enemy craft. The hope for this project is to advance the use of computational intelligence, and specifically CGAs, in areas of adaptive robotics with multiple sensory inputs. Additionally, we are interested in exploring the potential for an unconstrained CGA with jumping to solve problems that are suitable for multi-loop CGAs. This paper highlights the capabilities of evolutionary computation for video game and simulation technology.

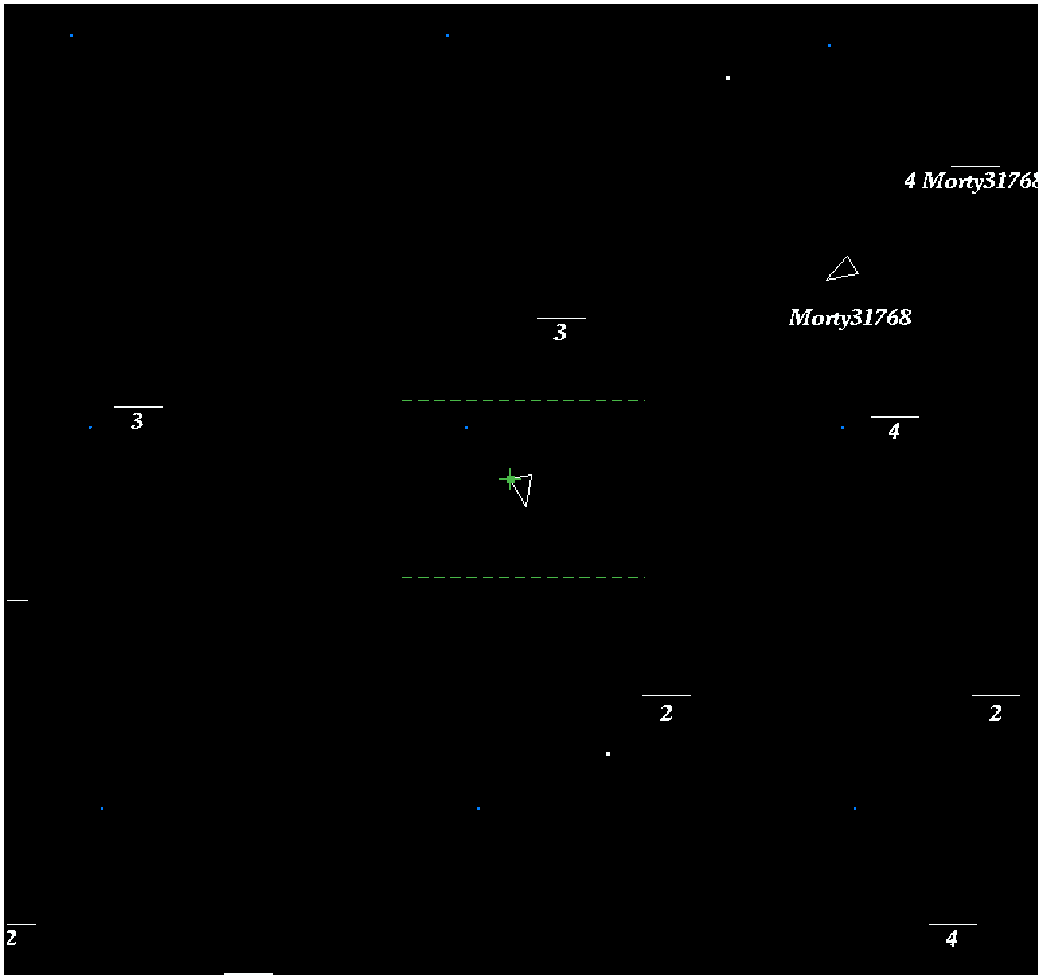

Fig. 1. Screenshot of evolving agent (center) evading enemy robot. Various starting points are shown, labeled with the team to which they belong. One white blip below the agent denotes a fired bullet by one of the robots, while smaller blips represent stars.

II. XPILOT

Xpilot (Fig. 1) is a popular 2-dimensional space-combat flight simulator written in the C programming language and freely available over the internet. A keyboard and mouse are used to control a ship. The game's open source code allows for modification to allow CGA controlled Xpilot ships. The client/server design supports multiple players, and has many variables for weapon and ship upgrades, map settings, team play, etc. Variables used by Xpilot to store information such as ship velocity, enemy and bullet location etc. can be parsed, and through the addition of a CGA, principles of evolution can be used to learn a looping control system for the agent. The control system uses the AI interface to simulate keystrokes and mouse movement to control the agent.

The evolving Xpilot agent was placed in a square arena enclosed by 32 blocks of 35 pixels on each side (the area of one block is roughly three times that of a ship). Placed in the arena with the evolving agent was a predator robot which used a rule based system designed by the authors. This predator robot was quite effective at harrying the agent; it was frustratingly difficult for a human player to combat. Whenever either robot was killed, whether from an enemy bullet, a wall, or crashing into the other agent, it reappeared at its initial start location. In addition to avoiding the enemy agent and its bullets, the evolving agent needed to evolve its flight to avoid the walls of the arena. The aim of the CGA was to evolve the chromosomes to allow the robot to survive as long as possible, without crashing or being shot. This was measured by the number of frames the robot survived. Each chromosome was run three times, and the sum of fitness taken. This was done in an effort to give each chromosome ample chance to display its fitness; depending on the location of the enemy robot when the evolving agent appeared on the map, the CGA-controlled agent might be destroyed before it had a chance to move. By allowing each chromosome to run three times, we hoped to minimize this effect.

III. CYCLIC GENETIC ALGORTHMS

Cyclic Genetic Algorithms (CGAs) are a type of genetic algorithm in which the genes represent tasks to be performed rather than traits of a problem solution [12] (see Fig. 2). CGAs were initially developed to generate walking gaits for hexapod robots. They were effective in solving this problem because the movement of a single leg requires a continual repetition of several actions such as lift leg, extend leg, lower leg etc. [13]. CGAs are highly suited to problem domains where sequential behavior is useful or necessary.

The looping structure of a CGA allows an agent to perform different movements in order, and then to repeat those instructions any number of times. A traditional single loop control system, however, may not allow certain complex sequences of behavior that would be useful in navigating the multi-variant and highly dynamic Xpilot environment; different sequences of actions (possibly of differing lengths) are desirable for the variety of tasks the agent may encounter, such as avoiding a wall, or trying to shoot an enemy.

Parker, Parashkevov, Blumenthal and Guildman (2004) applied the use of CGAs to multi-loop controls [14]. A chromosome with four loops was developed to teach a predator to locate a stationary prey. The loops each applied to one of four possible states in which the system could be, and when these states changed the program would jump to the loop corresponding to the new state.

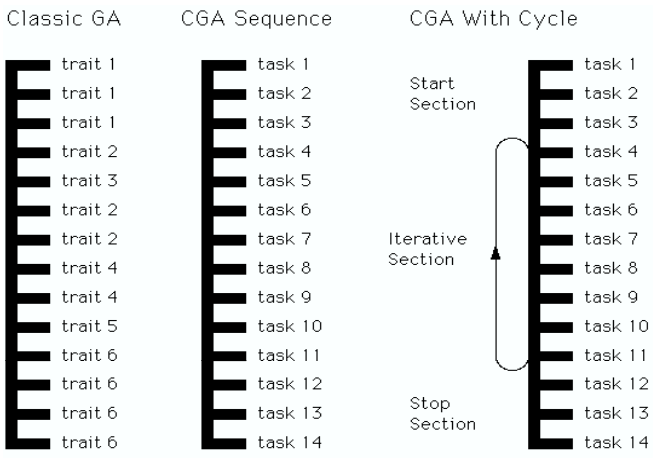

Fig. 2. Comparison of classic GA, a CGA sequence and CGA with single loop.

Although this multi-loop CGA was highly successful for the predator/prey scenario, the researchers found that it becomes less feasible as the number of inputs to the system increased, because this multi-loop CGA must have a segment for each possible state condition. To address this issue, Parker and Georgescu developed a multi-loop CGA with conditional branching to solve a search problem with obstacle avoidance [12]. They used the CGA to learn a control system for a LEGO Mindstorms robot to avoid obstacles while navigating towards a light. In their design, they were able to incorporate new inputs into their system by creating new instructions, rather than by adding exponentially more loops.

While the research performed by Parker and Georgescu used a system with 4 loops of predetermined length, the CGA introduced in this paper is less structured. The Xpilot agent's chromosome consists of one single loop, but due to the ability to jump from any one gene to any other, the system is capable of creating looping sequences of behavior for situations that combine different values of the sensory inputs. Loops of varying length can be created, and the conditional branching allows sequences to be incorporated into different loops. The Xpilot environment has varied sensory inputs of great importance: the distance and direction of walls, distance and velocity of enemy craft, bullet collision information, the agent's own track, heading

and velocity, etc. It is the aim of this paper to research the efficacy of an unconstrained CGA with jumping and conditional branching in the Xpilot problem domain.

A. *Learning the Controller*

A set of 15 conditions (rule antecedents) important for reasonable play were developed (Fig. 3). These included conditions dealing with the ship's position relative to walls, the enemy ship, and hostile fire, as well as the relationship between the ship's heading and its track. A binary coding was developed to represent the possible responses that were to be learned by the GA.

A population of 256 random chromosomes was generated initially. The controller is represented by a chromosome of 64 genes, with each chromosome being one large loop. Each gene is represented as a segment of 11 bits, so a chromosome is comprised of 704 bits. Each 11-bit gene either contains information for how the robot should act in a given frame, or an instruction to jump to another gene to look for control information.

The 15 different conditions were identified as potentially critical situations for which the robot would need to react. The system has the ability to check if a given condition is true, and to begin a different series of actions depending on whether it is true or false. In addition there is a sixteenth conditional which is always true; the system will jump to the specified gene without consulting any input information when this conditional is checked. This allows a gene to always jump to another gene, no matter the state of the inputs. Although we are using 16 possible conditionals in this experiment, the unconstrained CGA does not have a limit to the number used; to have more than 16, however, we would have needed to add bits to each gene in order to represent the additional possibilities.

1. Agent's velocity is greater than 10 pixels/second.
2. Agent's velocity is greater than 20 pixels/second.
3. Bullet is incoming and less than 60 pixels away.
4. Bullet is incoming and less than 130 pixels away.
5. Enemy ship is detected.
6. Nearest enemy ship is moving greater than 12 pixels/second.
7. Nearest enemy ship is closer than 200 pixels away.
8. Nearest enemy ships is closer than 450 pixels away.
9. Distance to nearest enemy ship is decreasing.
10. Distance to nearest enemy ship is decreasing by more than 5 pixels/second.
11. Distance to enemy ship is increasing by less than 5 pixels/second.
12. Distance to nearest wall is greater than 40 pixels.
13. Distance to nearest wall is greater than 132 pixels.
14. Enemy ship is pointed within 10 degrees of agent.
15. Enemy ship is pointed within 20 degrees of agent.

Fig. 3. Conditions for branching. The conditionals contain specific values (e.g., 40 pixels) that were determined by the authors to be effective values in previous research through trial and error. If the condition is true, the conditional jumps control to a designated gene.

B. *Xpilot Chromosome*

The chromosome is a single loop of 64 genes, with each gene consisting of 11 bits (Fig. 4). The first bit of the gene determines whether the gene is a control gene, which provides instruction for moving the agent, or a conditional jump gene, which causes the system to move to another gene if the condition is true. If the gene is a control gene, it executes the command (thrust, shoot, turn) and in the next frame it executes the next gene in the sequence; if it is a conditional jump gene it jumps to the specified gene. There it will either find a control gene to execute, or it will find another jump gene with instructions to conditionally jump. If the jump condition is not true, the system moves to the next gene in the loop and again can either execute a control action or a jump. The system executes a control gene once each frame, regardless of the number of jumps. With the ability to jump to any gene, sequences of instructions may be created where the final instruction commands a jump back to the first instruction; in this way variable length loops may be created.

If the gene is a control gene (designated by the first bit), the next 2 bits of the gene determine whether or not the robot will thrust. The binary value of the bits, from 0-3 is compared to a random number that is generated; only if the value of the bits is less than the random number will the robot thrust. This allows the system to evolve different movement speeds when spread over multiple frames. The next two bits are used in the same manner to determine whether or not the agent fires a bullet (thus giving the robot different possible rates of fire). The 3 bits next in sequence determine on which of 8 different criteria the agent will base its turn; it may turn towards or away from enemy bullets, enemy craft, walls, or the agent's own trajectory (the agent's trajectory and heading may not match).

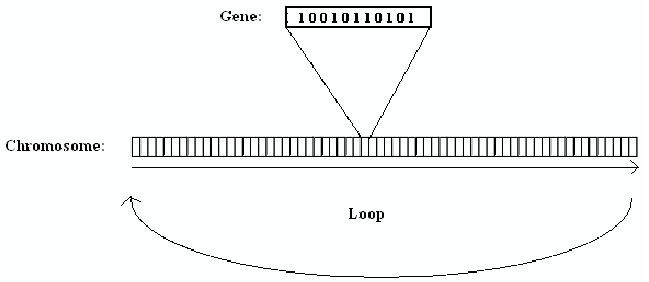

Fig. 4. Sample chromosome used to encode control and jump commands.

The agent always reacts to the nearest wall, the nearest enemy, and the most dangerous bullet etc. If we were to label the agent's track as north, then it searches each of the eight primary directions (north, northeast, east etc.) to identify the nearest wall. When avoiding bullets, the agent reacts to the location in space where the bullet will pass closest to the agents own track, not to the actual position of the bullet.

The final 3 bits of the gene determine the magnitude of the agent's turn; it may turn 0, 2, 4, 6, 8, 10, 12 or 14 degrees in the chosen direction. In the following frame, the next gene in the loop will be executed, and if the final gene in the chromosome is executed the system will return to the first gene in the chromosome.

If the first bit in a gene indicates that the gene is a jump gene rather than a control gene, then the remaining bits in the gene have a very different purpose. The four bits following the control bit will give the number of a condition to check, from 0-15. If that condition is true, then the system will jump to one of the 64 genes in the chromosome, designated by the final 6 bits of the current gene. For example, the gene may call condition 5 which asks if an enemy ship is detected. If there is an enemy detected, the system will jump to the gene designated by the remainder of the current gene. If the condition is not true, then the system moves to the next gene in the loop and executes the command there, whether it is a control gene or another jump gene. In the event that the gene checked is the final gene in the loop, it will return to the first gene in the loop. Several jump genes can be strung together to check more complex states; e.g. enemy is close, but moving away.

C. Genetic Operators

The total fitness of the population is stored so that it can be accessed up to any point in the trial of that population. Roulette wheel selection is used to choose parents for crossover [15]. Chromosomes are tested three times, and the number of frames they survive each trial is summed and stored. This number is squared and stored as the individual's fitness.

When two individuals are chosen for crossover, there is an even chance that they will be combined using one of two possible crossover methods. If inter-gene crossover is utilized, then a random crossover point is chosen; the chromosome of one parent is copied up to that point, and the remainder is copied from the other parent. This allows genes and even whole loops to be left intact. If intra-gene crossover is selected, then for each bit of a new gene, the bit at that spot is copied at random from one of the two parents. This allows for greater diversification. Each bit also has a chance of mutation; there is a one in three hundred chance of each bit being flipped, regardless of the type of crossover used.

The CGA was executed as a steady state genetic algorithm (SSGA) with first in first out deletion (FIFO), as discussed by De Jong and Sarma [16]. The FIFO SSGA genetic algorithm removes the oldest individual from the population as it puts in the newest individual. The new individuals are a recombination with mutation of two individuals picked with roulette wheel selection from the current population.

The best chromosome from each generation is saved to a data file, while its fitness and the average fitness of the generation are stored in a second file. Any of the stored chromosomes can be loaded into Xpilot, so that the behavior of the most highly fit chromosomes can be viewed over several robot lifetimes.

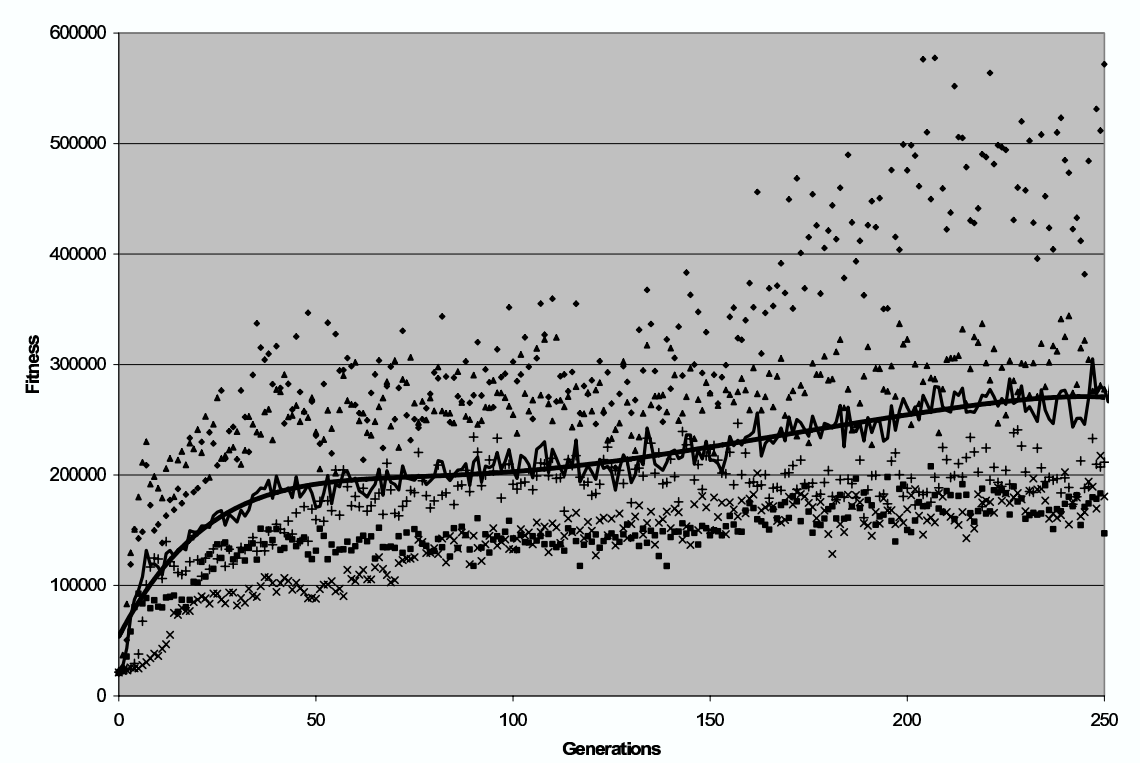

Fig. 5. The results of evolving controller for Xpilot using a CGA. The average population fitness is shown for each of 5 test runs. The line is the average of the 5 runs. The bold line is a 6th order polynomial trend line positioned by least squares.

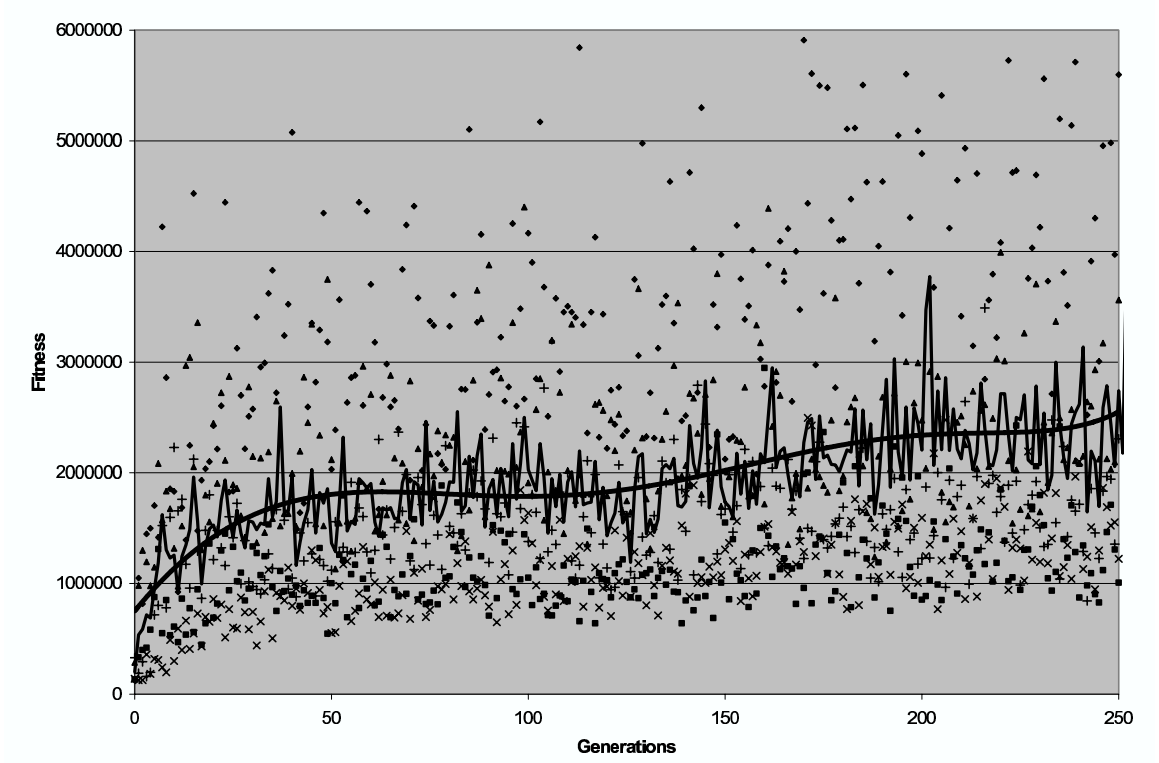

Fig. 6. The best individual chromosomes yielded by the evolution of a multi-loop controller for Xpilot using a CGA. The best individual fitness is shown for each of 5 test runs. The line is the average of the 5 runs. The bold line is a 6th order polynomial trend line positioned by least squares.

IV. Results

The results of the trials showed that control learning with the CGA was highly successful. Five test runs were started with randomly generated populations and were performed for 250 generations. The average fitness for each generation of each run is shown in Figure 5.

The enemy-harried robot evolved from an average fitness of 21,676.4 to a maximum average fitness of 305,118.8 in the 247th generation, averaged across all runs. The best individual chromosome from the 5 runs had a fitness of 5,909,761 in the 170th generation of run 3 (Fig. 6).

The agent evolved several different strategies across the runs. The agents in each run would sometimes exhibit behaviors visible in other runs, but each evolved one main strategy for surviving in its environment. In one run, the robot learned to fly in straight lines, then pivot and thrust as it approached walls. It would then fly in a straight line in a different direction. It would fire as the enemy crossed its path, but it did not seem to alter its flight drastically to seek or elude its enemy. This strategy was fairly successful for avoiding bullets, as the enemy robot had trouble tracking the agent on the long legs of its flight across the map.

Another evolving agent learned similar behavior, but it flew on shorter routes before altering its course; it reacted to the walls from a greater distance, and thus tended to use more of the central portion of the map. This agent also drifted along its heading, adjusting its track to fire at the enemy robot. It was the most successful agent in killing the enemy robot, but its ability to evade bullets suffered greatly.

In one of the runs, the agent seemed to evolve to solely try to evade bullets. It would fly to the middle of the map whenever it came too close to a wall, but would then turn and slow its motion to try and remain in the center. As it drifted, the agent would make miniscule adjustments to its flight path with short bursts of thrust accompanied by turning. In this way, the robot seemed to dance between the bullets that were fired at it. Generally, it made no efforts to fire upon the enemy robot. The success of this behavior was varied, but at times it was highly successful.

A different evolving agent learned an interesting strategy, but one that was initiated only when fired upon; otherwise it would sit in place. For this reason, it was often killed quickly. When the first few enemy bullets were detected by the agent, it would thrust; if the agent moved quickly enough, or the first few shots missed, it would begin to make large swooping circles around the map. As it moved, it would adjust its aim to fire upon the enemy when the enemy was detected, but would maintain its circular flight.

Occasionally it would break out of this and fly in a straight line, or make a figure eight, until more bullets would put it in danger. This robot would often be followed by a hail of enemy bullets that continually passed just behind it.

The behavior was successful both for evading enemy attacks and returning fire.

A different evolving agent learned an interesting strategy, but one that was initiated only when fired upon; otherwise it would sit in place. For this reason, it was often killed quickly. When the first few enemy bullets were detected by the agent, it would thrust; if the agent moved quickly enough, or the first few shots missed, it would begin to make large swooping circles around the map. As it moved, it would adjust its aim to fire upon the enemy when the enemy was detected, but would maintain its circular flight. Occasionally it would break out of this and fly in a straight line, or make a figure eight, until more bullets would put it in danger. This robot would often be followed by a hail of enemy bullets that continually passed just behind it. The behavior was successful both for evading enemy attacks and returning fire.

The final agent would sit in place but turn to fire upon the enemy robot. When bullets were detected, however, it would immediately thrust and attempt to flee the bullets. Unfortunately, the robot would often flee directly into a wall, and so be terminated in that manner. Even when this did not happen, it was often cut down quickly by the enemy robot's bullets.

The complete interpreted instructions for one of the more successful chromosomes are displayed below in order to demonstrate how the system functions.

0: Thrust 3/4 Shot 1/4 Turn 8 degrees Away from AIself.track
1: Check Rule: *Closing rate < -5* If true, jump to 25
2: Check Rule: *Shot is closer than 60 pixels* If true, jump to 25
3: Check Rule: *No Rule, Automatic jump* If true, jump to 21
4: Check Rule: *Wall > 40 pixels away* If true, jump to 41
5: Check Rule: *Closing rate < -5* If true, jump to 63
6: Check Rule: *Enemy velocity greater than 0* If true, jump to 26
7: Thrust 1/4 Shot 0 Turn 8 degrees Away from AIself.track
8: Check Rule: *Enemy closer than 200 pixels* If true, jump to 14
9: Thrust 3/4 Shot 0 Turn 8 degrees Towards Bullet
10: Check Rule: *Enemy velocity greater than 12* If true, jump to 38
11: Thrust 1/4 Shot 1/4 Turn -10 degrees Away from Wall
12: Thrust 1/4 Shot 0 Turn -10 degrees Away from Wall
13: Check Rule: *Enemy closer than 450 pixels* If true, jump to 7
14: Check Rule: *Wall > 40 pixels away* If true, jump to 23
15: Thrust 0 Shot 3/4 Turn 14 degrees Towards Bullet
16: Check Rule: *Enemy velocity greater than 12* If true, jump to 37
17: Thrust 1/4 Shot 0 Turn -10 degrees Away from Bullet
18: Thrust 0 Shot 1/2 Turn -12 degrees Away from Bullet
19: Check Rule: *Enemy aimed w/in 10 deg of ship* If true, jump to 18
20: Check Rule: *Wall > 40 pixels away* If true, jump to 46
21: Thrust 1/2 Shot 1/4 Turn -14 degrees Away from Wall
22: Thrust 3/4 Shot 3/4 Turn -14 degrees Away from Wall
23: Thrust 3/4 Shot 3/4 Turn -12 degrees Away from Bullet
24: Check Rule: Agent velocity < 20* If true, jump to 5
25: Thrust 0 Shot 0 Turn 12 degrees Away from AIself.track
26: Check Rule: *Wall > 40 pixels away* If true, jump to 1
27: Check Rule: *Closing rate < 5* If true, jump to 48
28: Thrust 3/4 Shot 3/4 Turn -6 degrees Away from Bullet
29: Check Rule: *Closing rate < 0* If true, jump to 22
30: Check Rule: *Enemy velocity greater than 12* If true, jump to 61
31: Check Rule: *Wall > 132 pixels away* If true, jump to 48
32: Check Rule: *No Rule, Automatic jump* If true, jump to 0
33: Check Rule: *Enemy closer than 450 pixels* If true, jump to 6
34: Check Rule: *Wall > 132 pixels away* If true, jump to 22
35: Check Rule: *Wall > 132 pixels away* If true, jump to 13
36: Check Rule: Agent velocity < 20* If true, jump to 63
37: Check Rule: *Closing rate < -5* If true, jump to 36
38: Thrust 1/4 Shot 0 Turn -10 degrees Towards AIself.track
39: Thrust 3/4 Shot 3/4 Turn 12 degrees Towards Wall
40: Check Rule: *Enemy velocity greater than 12* If true, jump to 1
41: Check Rule: *Wall > 40 pixels away* If true, jump to 63
42: Check Rule: *No Rule, Automatic jump* If true, jump to 29
43: Check Rule: *Enemy closer than 450 pixels* If true, jump to 13
44: Check Rule: *Closing rate < 5* If true, jump to 37
45: Thrust 3/4 Shot 1/4 Turn 0 degrees Away from Enemy ship
46: Check Rule: *Enemy velocity greater than 12* If true, jump to 5
47: Check Rule: *No Rule, Automatic jump* If true, jump to 3
48: Check Rule: Agent velocity < 20* If true, jump to 23
49: Check Rule: *Enemy velocity greater than 12* If true, jump to 25
50: Thrust 1/4 Shot 0 Turn -8 degrees Away from Bullet
51: Check Rule: *Enemy closer than 200 pixels* If true, jump to 43
52: Thrust 3/4 Shot 0 Turn 0 degrees Towards Enemy ship
53: Thrust 1/2 Shot 1/2 Turn 0 degrees Towards AIself.track
54: Check Rule: *Enemy closer than 200 pixels* If true, jump to 57
55: Check Rule: Agent velocity < 20* If true, jump to 5
56: Check Rule: *Enemy velocity greater than 12* If true, jump to 51
57: Check Rule: *Closing rate < -5* If true, jump to 59
58: Check Rule: *Enemy velocity greater than 12* If true, jump to 18
59: Check Rule: *Closing rate < 5* If true, jump to 6
60: Check Rule: *Enemy aimed w/in 20 deg of ship* If true, jump to 54
61: Thrust 1/2 Shot 1/2 Turn -12 degrees Away from Enemy ship
62: Thrust 3/4 Shot 1/2 Turn -8 degrees Away from Enemy ship
63: Thrust 0 Shot 3/4 Turn -4 degrees Away from Wall

By beginning with the first gene of the system denoted by "0", we can follow the system's jumps, most of which are conditional upon inputs from the environment (several demand jumps regardless of the inputs). The system begins by instructing the robot away from its own track with a high chance of thrusting. After this initial step, the system begins to act according to the state the robot is in. It first checks if the enemy agent is closing at a fast rate (gene 1). If it is, the system jumps to gene 25 and turns the agent away from its current track. The system then checks if the agent is farther than 40 pixels from a wall (gene 26), and if it is safe from walls it returns to gene 1.

If the robot is close to a wall and the distance to the enemy is increasing by 5+ pixels/second (gene 27), the robot will thrust away from any dangerous incoming bullets (gene 28).

If the enemy is now closing on the robot (gene 29), the system will jump and the robot will finally turn and thrust from the nearby wall (gene 22) detected by gene 26. If other branches are taken, it will be several more frames before a turn is made from a wall.

The above sequence suggests that the learned controller considers a closing enemy agent to be the most important input, even more important perhaps than incoming bullets. This makes sense when one observes that the enemy agent has a high rate of fire, but is extremely inaccurate as the distance increases from the evolving agent. The sequence also helps explain why wall avoidance behavior can still be imperfect in highly evolved agents. (It is important to recognize that the above steps took place in just a few frames; that is, fractions of a second).

V. CONCLUSIONS

Based upon the results of the experiment, an unconstrained CGA can be used to develop a branching controller with multiple loops for an autonomous robot operating in the Xpilot environment. The control system supports the potential for unconstrained CGAs to evolve multiple loops of variable length to solve problems that are suited to traditional multi-loop CGAs. In the later portions of the runs, the GA driven robot was often more successful at dispatching its enemy than was the enemy robot. If given a larger number of possible behaviors or a predictive aiming function, we speculate that the robot could survive significantly longer in the arena; ongoing research is being conducted to explore these hypotheses. With the increased difficulty in facing the enemy robot we designed taken into account, the unconstrained CGA controller is a significant improvement over our previous rule-based system for Xpilot control.

REFERENCES

[1] Schultz, A. C. (1994). Learning robot behavior using genetic algorithms. Intelligent Automation and Soft Computing: Trends in Research, Development, and Applications.

[2] Schultz. A. C. (1991). Using a genetic algorithm to learn strategies for collision avoidance and local navigation. Proceedings of the Seventh International Symposium on Unmanned, Untethered Submersible Technology.

[3] Beer, R., & Gallagher, J. (1992). Evolving dynamical neural networks for adaptive behavior. Adaptive Behavior, 3, 469-509.

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

[5] Konidaris, G., Shell, D., and Oren, N. (2002). Evolving neural networks for the capture game. Proceedings of the SAICSIT Postgraduate Symposium.

[6] Hingston, P. and Kendall, G. (2004). Learning versus evolution in iterated prisoner's dilemma. Proceedings of the International Congress on Evolutionary Computation 2004 (CEC'04).

[7] Yannakakis, G. and Hallam, J. (2004) Evolving opponents for interesting interactive computer games. Proceedings of the 8th International Conference on the Simulation of Adaptive Behavior (SAB'04).

[8] Cole, N., Louis, S., and Miles, C. (2004). Using a genetic algorithm to tune first-person shooter bots. Proceedings of the International Congress on Evolutionary Computation 2004 (CEC'04).

[9] Funes, P., & Pollack, J. (2000). Measuring progress in coevolutionary competition. From Animals to Animats 6: Proceedings of the Sixth International Conference on Simulation of Adaptive Behavior.

[10] Parker, G., Parker, M., & Johnson, S. (2005). Evolving autonomous agent control in the Xpilot environment. Proceedings of the 2005 Congress on Evolutionary Computation (CEC 2005). Edinburgh, UK.

[11] Parker, G., Doherty, T., & Parker, M. (2005). Evolution and prioritization of survival strategies for simulated robots in Xpilot. Proceedings of the 2005 Congress on Evolutionary Computation (CEC 2005). Edinburgh, UK.

[12] Parker, G., & Georgescu, R. (2005). Using cyclic genetic algorithms to evolve multi-loop control programs. Proceedings of the 2005 IEEE International Conference on Mechatronics and Automation (IMCA 2005). Niagra Falls, Ontario, Canada.

[13] Parker, G., Rawlins, G. (1996). Cyclic genetic algorithms for the locomotion of hexapod robots. Proceedings of the World Automation Congress (WAC 1996), 3, Robotic and Manufacturing Systems.

[14] Parker, G., Parashkevov, I., Blumenthal, H., & Guildman, T. (2004). Cyclic genetic algorithms for evolving multi-loop control programs. Proceedings of the 2004 World Automation Congress (WAC 2004). Seville, Spain.

[15] Goldberg, D. (1989) Genetic algorithms in search optimization and machine learning. Addison-Wesley, Reading MA.

[16] De Jong, K., & Sarma, J. (1992). Generation gaps revisited. Proceedings of the Second Workshop on Foundations of Genetic Algorithms (FOGA 1992). Vail, Colorado, USA