Evolving Flocking Navigation Controllers for Adaptive Swarms

Nicholas Felt and Philip Koonce
Computer Science Department
Swarthmore College
Swarthmore, PA 19081, USA
{nfelt1,pkoonc1}@cs.swarthmore.edu

11 May 2012

Abstract

Flocking behavior in nature provides an example of perfectly graceful collective navigation. Inspired by this ability of the birds and the bees, we attempted to use techniques from artificial intelligence (AI) to "naturally" develop a neural network controller for swarms of robots. We constructed a framework consisting of a parallelized genetic algorithm (GA), a three–layer feed–forward neural network, and a 2D flocking simulator, and used this setup to run experiments searching for controllers that effectively demonstrated flocking behavior, as measured by the properties of compactness and alignment. These experiments used a range of training scenarios and fitness functions, including situations with and without food sources, with and without alignment rewards, and with varying degrees of local awareness of flock neighbors. Though we were ultimately unsuccessful in developing a consistent controller that imitates biological flocking, we gathered some encouraging results and gained valuable experience attempting to solve a complex problem with techniques from AI.

1 Introduction

Flocking, mesmerizing and delightful to watch in nature, is an example of evolution's ability to create an elegant solution to a monumentally complex task. In a starling flock, hundreds — or even hundreds of thousands — of birds decide their motion via local interactions, leading to the emergence of a single large-scale phenomenon: the coordinated dance of the flock as a whole.

Flocking behavior, defined concisely by Craig Reynolds as "polarized, noncolliding, aggregate motion" [9], has long attracted the attention of both biologists and computer science researchers. In computer animation, modeling realistic flocking behavior can completely eliminate the tedious task of scripting the paths of every individual in an animated flock of birds. To this end, graphics researchers have proposed simple models of an individual's behavior in a flock that results in realistic–looking flocking behavior [9, 11]. That fairly simple organisms, such as birds and fish, can perform their individual roles in a flock, reacting perfectly and instantaneously to their environment, suggests that simple yet accurate models of flocking behavior do exist.

At the same time, multi-agent coordination in navigation tasks is still an unsolved problem in robotics. A quick survey of demonstration media by various research groups reveals that graceful and efficient navigation is typically well out of the reach of the systems on display. This is often because these systems rely upon some central controller performing path planning for all the agents and sequentially issuing movement orders to other agents. Researchers have proposed imitating nature's solution to this problem with simple navigation controllers that primarily use local information [1, 7].

In this paper, we apply techniques from the fields of artificial intelligence and adaptive robotics toward producing multi-agent coordination in navigation tasks. Specifically, we use genetic algorithms to evolve weights in a neural network navigation controller for large groups of homogeneous simulated agents. This process involves a computationally expensive simulation of agent behavior in order to evaluate each candidate agent's "fitness", as well as a long evolution time consisting of several hundred or more generations of evolution (refer to Section 4.1 for the details of the genetic algorithm we implemented). Thus, in order to complete a reasonable number of experiments within the time frame of this project, we employ our previously implemented parallel framework for genetic algorithm computation, the details of which can be found in Felt and Koonce [3]. To evolve neural network controllers using this framework, we rely on a lightweight neural network library and a simulator running within a custom fitness function called by the framework. With these tools, we attempt to evolve effective flocking agents, using a variety of fitness functions and evaluation scenarios. Though our experimental results are generally unsuccessful, we do discuss the ways in which they did not match our expectations. Finally, we review lessons learned and various routes for future work that could improve on our approach.

2 Related Work

Relevant prior work falls into two categories. First, biologists and computer scientists at the intersection of computer science and animal behavior have extensively studied the behavior of flocks, herds, and schools of biological organisms [2, 8]. Many emergent properties of the flocks as a whole have been enumerated and described in attempt to reduce the behavior down to simple rules. Conversely, many localized spatial and social interactions have been observed and proposed as factors leading to the emergent flocking behaviors. Both of these approaches to studying biological flocking are useful to a computer scientist implementing simulated flocking. The emergent properties provide starting points for evaluating the performance of artificial models, and the individual interactions provide building blocks for creating these artificial models.

In the other category are prior attempts to develop mathematical, artificial models of biological flocking [6, 7, 9]. The models presented in these papers draw constraints and rules from the aforementioned work by biologists. Olfati-Saber [7] concisely summarizes the rules that most mathematical models of flocking incorporate:

- Stay close to nearby flockmates
- Avoid collisions with nearby flockmates
- Match the alignment of nearby flockmates
- Match the velocity of nearby flockmates

These rules have resulted in models that function by exerting a turning and acceleration force on each individual as a function of distance from and alignment with its neighbors. Olfati-Saber [7] also presents some shortcomings in this prior work, discussed in detail in Section 3.1, some of which we hope to address in our solution.

A few other researchers have attempted to use adaptive techniques like genetic algorithms to attempt to evolve controllers for flocking agents [5, 12]. Zaera et al. [12] attempted to evolve neural network controllers for simulated fish, and it was after this paper that we most closely modeled our approach. The authors were not able to produce a controller resulting in behavior that resembled schooling. There were three major differences between our approach and the approach in this paper that we hoped would allow us to succeed where the authors failed.

First, the flexibility of the neural networks in Zaera et al. [12] was limited — only 5 hidden nodes. Furthermore, though the authors do not specify what the activation function used in the neural networks is, weights were constrained to the interval $[0, 1.0)$. Especially if the standard logistic function was used, this may have severely limited flexibility of the networks produced. The neural networks used in our experiment have unbounded weights and a different mutation method (presented in Section 4.2) that eliminates this constraint. Second, evolution was only run

for 100 generations. Because the authors employed a sequential evolution on a sequential simulator, their experiments were constrained by time. Reflecting on experiments we have run in our AI and Adaptive Robotics courses, we concluded that 100 generations is not nearly long enough to solve a complex task like flocking

Finally, the simulator and simulated fish in this work were very complex. In an attempt to maximize biological plausibility, the authors simulated fish in 3D with realistic Newtonian kinematics (including mass, moments of inertia, and drag). Furthermore, each agent's inputs were based on 3D simulated vision. In our implementation (see Section 4.2 for details), we simulate agents in 2D and discard most of the physics simulation, only limiting the speeds of each agent. We do not simulate vision and give individuals perfect information about their nearest neighbors. While less biologically plausible, this eliminates a huge amount of complexity that could prevent the GA from finding a solution. As a last note, Zaera et al. [12] conclude with an important point in their paper: developing a fitness function to encourage flocking behavior is at least as difficult as creating a hand–tuned model. Perhaps we should have better heeded this warning!

3 Hard–Coded Models

Before attempting to evolve a neural network flocking controller, we attempted to replicate the results of prior work in the modeling of flocking behavior in order to better gauge the complexity of the task we were setting the genetic algorithm. Particularly, we wished to imitate the work of Hodgin [4], who created an absolutely mesmerizing simulation of a real behavior in fish schools know as the "bait ball". In this simulation, Hodgin uses only the 4 main rules listed above (and an addition rule to avoid predators) and achieves stunning results. While we lack Hodgin's graphics experience, we hoped to develop simulated agents with similar behavior. We quickly realized that this task could easily be a project on its own (perhaps in a graphics course — Hodgin built his simulation in the framework of a particle effects rendering engine), but we did encounter many of the pitfalls predicted by Olfati-Saber [7], which we will now discuss.

3.1 Pitfalls of Force–Based Models

In the simulations presented in this paper, individuals started out placed randomly in a 640×480 window, with 0 velocity. The first experiment was to simply subject the individuals to forces from neighbors according to a function of their distance function. Intuitively, this function should a) be very negative at close distances, b) be around zero for ideal following distances, and c) be positive when an individual should move closer to its neighbors. Originally, the function in Equation 1 was used. This function accounts for avoiding collisions with the $-1/x$ term, and gives a positive attractive force starting around $x = 40$. However, this led to the first of the pitfalls: group collapse.

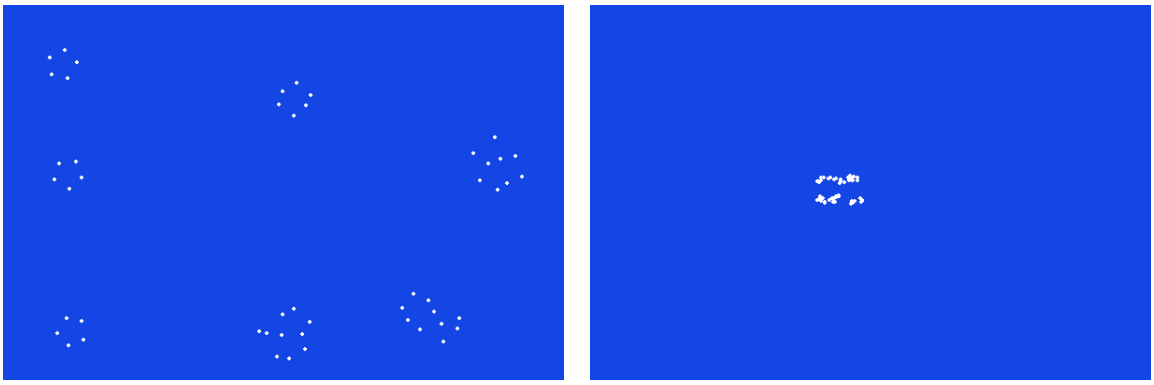
$$-\frac{1}{x} + 10 * \frac{1}{1 + e^{-x+40}} \tag{1}$$

Olfati-Saber [7] proposes that force–based models can lead to group fragmentation and group collapse. With a constant, positive, attractive force above a certain distance threshold, all individuals are pulled toward the centroid of the group, and even with a strong negative force at close distances, the force function acts like gravity and crushes individuals inward. The resulting simulation looks completely inorganic, with individuals effectively bouncing off one another close to the centroid like excited particles. A screen capture is shown in Figure 1b.

Another experiment led to the other extreme of Olfati's pitfall. Using Equation 2, which includes a term to decay the force exerted by neighbors with distance, we saw group fragmentation, as shown in Figure 1a, because there was no significant force pulling disparate groups back together as there likely is in biological flocks.

$$-\frac{100}{x} + x * e^{-\frac{x}{100}} * \frac{1}{1 + e^{-x+40}} \tag{2}$$

By rewarding our evolved agents for simply being at reasonable distances from one another and evolv-



(a) Fragmented Group (b) Collapsed Group

Figure 1: Two of the failures explained by Olfati, frequently encountered in hard–coded force–based models.

ing the parameters of the controller, we hoped to avoid both of these problems entirely. Another pitfall was that individuals had no knowledge of and therefore no capability to react to global flock location. In our hard–coded model we combined a function of nearest neighbor distance with similar force functions for distance from the flock's centroid and distance from a target (a food source, to give a real–life example), and achieved behavior that vaguely resembled flocking in our hand–tuned agents. This required with considerable tuning of both the individual functions (adding and tweaking terms) and their relative weights. Our hope was to replace what was sure to be endless tweaking of parameters and function terms with an adaptive search technique: genetic algorithms.

4 Implementation

4.1 Genetic Algorithms

To perform the experiments presented in this paper, we implemented a parallel genetic algorithm framework in C++ using MPI, a message-passing interface standard providing high scalability. The details of this framework and its API appear in Felt and Koonce [3]; the parts relevant to this work are the specific details of the genetic algorithm this framework implements. It is a standard genetic algorithm, which consists of a population of individuals — each represented by an array of numeric values, the chromosome, typically encoding a solution to a problem — and a fitness function with which to evaluate individuals. Each generation of evolution consists of evaluation of each individual, selection of pairs of new individuals with probability proportional to their fitness using the roulette selection method, single–point crossover of every two new individuals performed with a probability P_c, and then mutation of each value in the chromosome of the new individuals with probability P_m. In our framework, individuals are subclasses from a generic `ArrayIndividual` class, templated on the type of data in the chromosome and the chromosome's size, and override a `compute_fitness()` fitness function and optionally the `mutate()` method. This allows for our framework to easily be applied to many problems.

4.2 Neural Networks

To evolve neural networks for flocking controllers, it was necessary to either incorporate an existing C++ neural network library into our framework or implement our own. Since we did not foresee requiring recurrent networks or requiring back–propagation, we implemented our own lightweight neural network class, and restricted our solutions to three–layer,

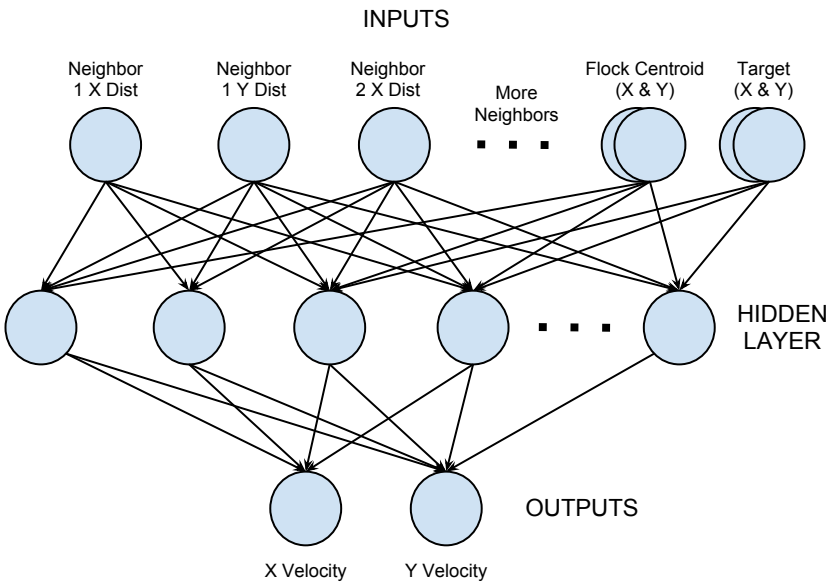


Figure 2: The topology of our three–layer feed–forward neural network controller, showing the inputs to the network (X and Y distances to some number of neighbors, the flock centroid, and the target location) and the network outputs (the agent's X and Y velocity).

feed–forward, fully–connected networks. The general topology for these networks appears in Figure 2. Inputs to the network included the X and Y distances to some preset number of the agent's nearest neighbors (we varied this exact number between experiments, from as little as 3 to as high as 10). Other inputs that we used in some experiments included the centroid of the flock and the target (i.e. food source) location, again both represented as X and Y distance offsets from the individual agent.

We also implemented an agent class, consisting of little more than a wrapper around an instance of the neural network class with additional state such as location and velocity, along with a method to construct the neural network from a chromosome of weights.

4.3 Simulator

Neural network controlled agents reside in a simulator object, which allows agents to move and communicate location information with each other in an unbounded 2D world. The simulator runs an update function for a specified number of time steps (which we varied between experiments). At each time step, the simulator updates the inputs of all agents, runs each agent's feed–forward method, updates the velocities of all agents, and then updates their positions. The simulator class can register a function pointer to an arbitrary fitness function to perform analysis of the group as a whole (to give fitness for compactness or group alignment, for example), which can be run every N time steps for a set value of N, or once at the end of simulation.

4.4 Tying into the GA Framework

All these pieces tie together into a powerful system to search for effective flocking controllers. We created a subclass of the `ArrayIndividual` templated on doubles and the size of a compile–time determined network topology, which is configured by a series of compiler directives. This flocking individ-

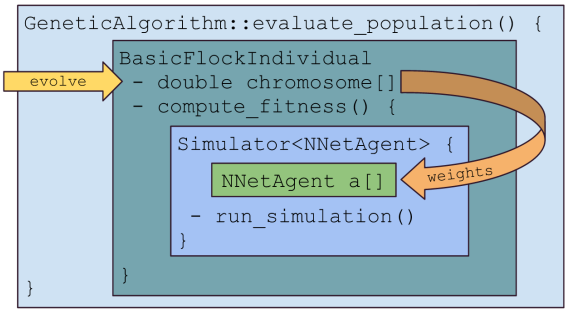


Figure 3: `GeneticAlgorithm` selects, recombines, and mutates an array of individuals, using the individuals' evaluation function, which instantiates a `Simulator`. This simulator contains an array of `NNetAgent`s, which all move according to the outputs of their neural networks.

ual class overrides two important methods. First, `mutate()` adjusts each randomly-mutated gene in the chromosome by a random scaling factor in the interval $[-2, 2)$. This allows random mutations to both tweak values slightly, but also to grow or shrink them exponentially. Second, `compute_fitness()` instantiates a simulator for this individual, populates it with a neural network agents with weights from this individual's chromosome, registers a fitness function for the group as a whole, and simulates a fixed number of time steps, aggregating fitness as appropriate. A schematic diagram of our system, showing its consituent pieces, appears in Figure 3.

5 Experimental Results and Discussion

Most experiments in this paper were run using groups of 10 agents, simulations that lasted 1000 timesteps, using populations of 300 individuals, run for 1000 generations. Using a non–parallelized framework, each experiment would have taken over 26 hours, which would have prevented us from iteratively tweaking the fitness function and simulator parameters, looking for positive results. With our parallel framework, we could run the same experiments on over 300 cores in parallel, achieving nearly linear speedup in evolution run time.

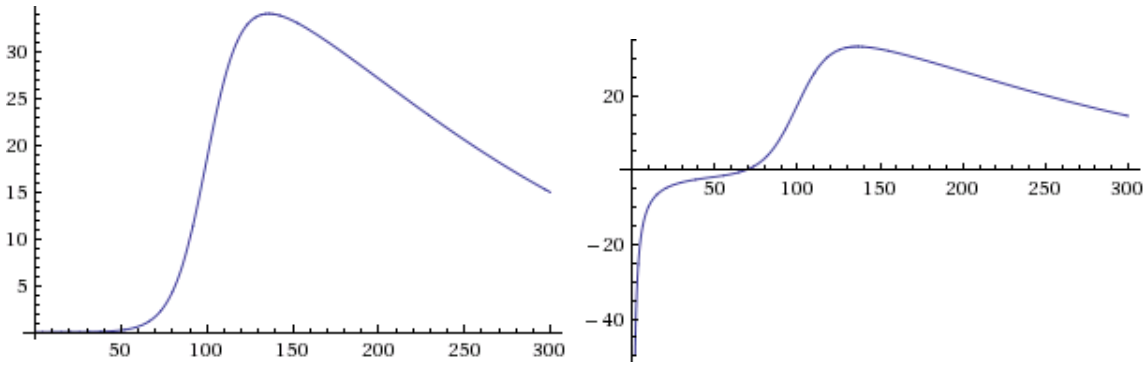
We began by simply rewarding agents based on their distance from all other individuals according to the function in Equation 3, a plot of which is shown in Figure 4a. This resulted in group collapse, just as the hand–coded experiments showed. We hoped to achieve flock–like clustering by rewarding individuals positioned a certain distance from its neighbors, and giving zero (or near–zero) fitness for being too close, because in roulette selection, fitness values must all be positive real numbers. With this always–positive function, we could simply add the reward for each neighboring agent in the simulator and return this value to the GA as the fitness for the individual from which the agents were instantiated.

$$xe^{\frac{-x}{100}} \frac{1}{1+e^{-.1x+10}} \tag{3}$$

The results of simulating with more agents (50, 100), for more generations (5000, 10000), and longer simulations (5000, 10000 time steps) were all similar. We next tried changes to the fitness function, adding a $-100/x$ term to punish agents that ended up too close to one another. The function and its graph are shown in Equation 4 and Figure 4b. In order to use this with the roulette method of selection, we clipped all negative fitnesses to 0. We tried the same amount of tweaking as we did in the previous experiments, and we were met with considerably more success. As shown in Figure 5a, agents quickly organize into a cluster that resembles a flock, moving slowly as a group in a random direction. This was very encouraging, but our excitement was short–lived. In longer simulations, agents controlled by these networks consistently moved farther and farther apart, as shown in Figure 5b.

$$\frac{-100}{x} + xe^{\frac{-x}{100}} \frac{1}{1+e^{-.1x+10}} \tag{4}$$

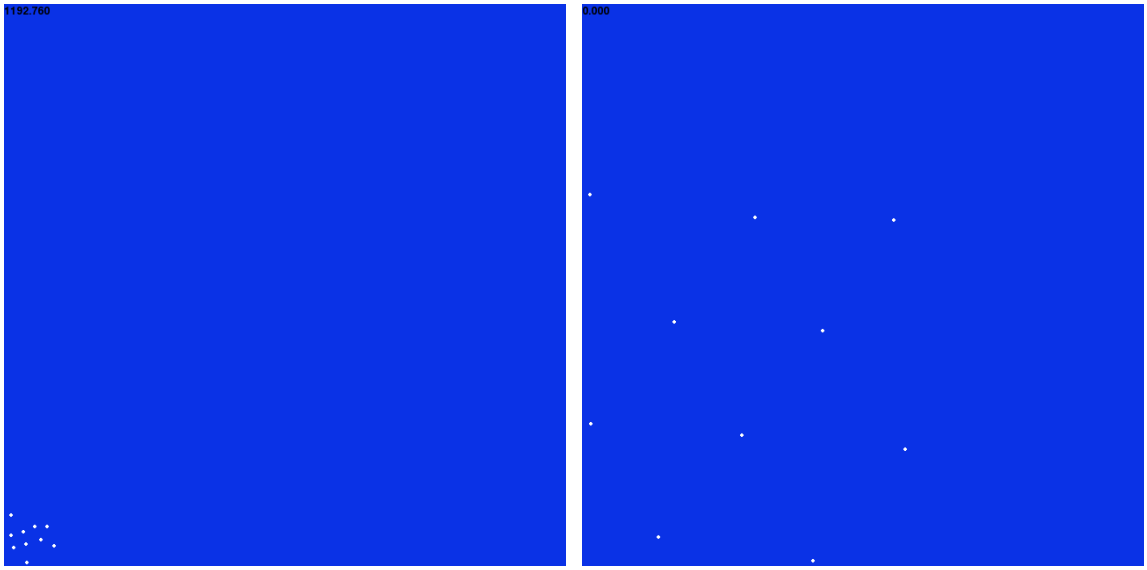
Other attempts to encourage flocking with different fitness functions were not much more successful. Among other things, we added inputs for sensing neighbor alignment, a target (simulating a food source), and the flock centroid. We tried different



(a) Initial Reward Function

(b) Reward Function with Collision Punishment

Figure 4: The initial and modified reward functions. The x-axis shows squared distance from a neighbor, the y-axis shows a scale-independent reward.



(a) Correct Clustering

(b) Drifting Apart

Figure 5: Our more successful experiments led to the group forming a well–spaced cluster, but then in longer simulations, the group drifted ever farther apart.

combinations of fitness functions that rewarded individuals for aligning with the aggregate motion of the group, with its neighbors, for being near the flock centroid, and for being near food, and we also varied the number of neighbors for which a reward involving nearest neighbors was applied. For each combination of sensory capabilities and additions to the fitness function, we found in the best case a new variation on not–quite flocking behavior, and in the worst case something entirely implausible (like group collapse).

6 Conclusion and Future Work

We originally hoped to replace the tedious process of tweaking parameters in a hand–tuned controller for swarm agents with adaptive search using genetic algorithms. We hypothesized that with a fitness function that rewarded individuals for being well–spaced and aligned with their neighbors, genetic algorithms would find weights for a simple neural network controller that would imitate biological flocking behavior. Ultimately, however, we discovered that developing a good fitness function for the genetic algorithm is equally difficult, if not more difficult, than the task of hand–coding a controller that imitates flocking with force–based functions.

We still believe that this method for developing neural network controllers is viable, given a good fitness function. The problem we encountered in our research is a problem encountered by nearly every computer science researcher who has employed genetic algorithms. That said, many researchers have overcome the hurdle of finding a fitness function without any "loopholes" or inconsistencies while encouraging all the behaviors desired in the evolved agents. A few of the controllers we evolved behaved briefly very much like biological organisms in flocks, but they always went on to perform unrealistically later on. We expect that with more time and more experiments, we could achieve more thoroughly satisfying results.

Among the ideas we have for improving the experimental strategy presented in this work, one of the most promising would be adopting a more gradual approach to evolving flocking controllers. One way to accomplish this would be to use the NEAT algorithm for neural network evolution, first introduced by Stanley and Miikkulainen [10]. Since NEAT starts with a very simple network and gradually adds complexity when needed to adapt to more complex tasks, it might perform well at developing minimal flocking controllers. For example, we could evolve a NEAT-based network by first evaluating it as a controller for very small flocks (e.g. of size 2) and then for flocks of increasing size. Another potential advantage of this approach is that NEAT provides built-in methods for ensuring that crossover of chromosomes respects the topological structure of the neural network, which our general genetic algorithm implementation does not. This aspect could make NEAT more effective at combining successful parts of parent individuals into new candidate individuals.

One other approach we considered originally (that is more attractive in retrospect) is to train neural network controllers for flocking agents with back–propagation. This would require training examples of actual flocking behavior, but such do datasets exist. Couzin et al. [2] have published research with golden shiners, a species of fish that forms schools in shallow water (i.e., in an approximatation of a 2D plane). They have produced frame-by-frame datasets of flocking motion derived from top-down videos of golden shiners schooling in a tank, with location and velocity information over time for each individual. This dataset would lend itself to use in training a neural network to school using back–propagation. Agents could be exposed to examples of the turning and acceleration vectors of individual fish paired with the relative locations of their nearest neighbors. Such an approach might provide an alternative, potentially more direct way of achieving simple and realistic flocking behavior.

This project has shown us the depth of difficulty inherent in solving a complex task using AI techniques. Still, considering our progress in the context of progress made previously by other researchers and current biological research in collective behavior, we hope and expect that the problem of creating simple models for nature-inspired multi-agent navigation will soon be solved.

References

[1] F. Bullo, J. Cortés, and S. Martinez. *Distributed control of robotic networks: a mathematical approach to motion coordination algorithms*. Princeton University Press, 2009.

[2] I.D. Couzin, J. Krause, N.R. Franks, and S.A. Levin. Effective leadership and decision-making in animal groups on the move. *Nature*, 433 (7025):513–516, 2005.

[3] N. Felt and P. Koonce. A scalable framework for genetic algorithms using MPI. *Swarthmore College CS87 Final Projects*, 2012.

[4] Robert Hodgin. Bait ball, 2012. URL `http://roberthodgin.com/bait-ball/`.

[5] H. Kwasnicka, U. Markowska-Kaczmar, and M. Mikosik. Open-ended evolution in flocking behaviour simulation. 2007.

[6] M.J. Mataric. Designing emergent behaviors: From local interactions to collective intelligence. In *Proceedings of the Second International Conference on Simulation of Adaptive Behavior*, pages 432–441, 1993.

[7] R. Olfati-Saber. Flocking for multi-agent dynamic systems: Algorithms and theory. *Automatic Control, IEEE Transactions on*, 51(3): 401–420, 2006.

[8] J.K. Parrish, S.V. Viscido, and D. Grünbaum. Self-organized fish schools: an examination of emergent properties. *The biological bulletin*, 202 (3):296–305, 2002.

[9] C.W. Reynolds. Flocks, herds and schools: A distributed behavioral model. In *ACM SIGGRAPH Computer Graphics*, volume 21, pages 25–34. ACM, 1987.

[10] K. O. Stanley and R. Miikkulainen. Evolving neural networks through augmenting topologies. *Evolutionary Computation*, 10(2):99–127, 2002.

[11] H.G. Tanner, A. Jadbabaie, and G.J. Pappas. Stable flocking of mobile agents, part i: Fixed topology. In *Decision and Control, 2003. Proceedings. 42nd IEEE Conference on*, volume 2, pages 2010–2015. IEEE, 2003.

[12] N. Zaera, D. Cliff, et al. (Not) Evolving collective behaviours in synthetic fish. In *In Proceedings of International Conference on the Simulation of Adaptive Behavior*. Citeseer, 1996.