Chapter 6

Evolving Finite State Automata

In this chapter, we will evolve finite state automata. (For the benefit of those trained in computer science we note the finite state automata used here are, strictly speaking, finite state transducers: they produce an output for each input.) Finite state automata (or FSAs) are a staple of computer science. They are used to encode computations, recognize events, or as a data structure for holding strategies for playing games. In Section 6.1, we start off with a very simple task: learning to predict a periodic stream of zeros and ones. In Section 6.2, we apply the techniques of artificial life to perform some experiments on the Iterated Prisoner's Dilemma. In Section 6.3, we use the same technology to explore other games. We need a bit of notation from computer science.

Definition 6.1 *If A is an alphabet, e.g., $A = \{0, 1\}$ or $A = \{L, R, F\}$, then we denote by A^n the set of strings of length n over the alphabet A.*

Definition 6.2 *A* **sequence** *over an alphabet A is an infinite string of characters from A.*

Definition 6.3 *By A^* we mean the set of all finite length strings over A.*

Example 6.1

$$\{0, 1\}^3 = \{000, 001, 010, 011, 100, 101, 110, 111\}$$

$$\{a, b\}^* = \{\lambda, a, b, aa, ab, ba, bb, aaa, aab, aba, abb, \ldots\}$$

.

Definition 6.4 *The symbol λ denotes the empty string, a string with no characters in it.*

Definition 6.5 *For a string s we denote by $|s|$ the length of s (i.e., the number of characters in s).*

Example 6.2

$$|\lambda| = 0$$

$$|heyHeyHEY| = 9$$

.

6.1 Finite State Predictors

A finite state automaton requires an input alphabet, an output alphabet, a collection of states (including a distinguished initial state), a transition function, and a response function (possibly including an initial response used before the automaton has processed any input). The states are internal markers used as memory - like the tumblers of a combination lock that "remember" if the user is currently dialing in the second or third number in the combination. The transition function encodes how the automaton moves from one state to another. The response function encodes the outputs produced by the automaton, depending on the current state and input.

An example may help make some of this clear. Consider a thermostat. The thermostat makes a decision every little while and must not change abruptly from running the furnace to running the air-conditioner and vice-versa. The input alphabet for the thermostat is $\{hot, okay, cold\}$. The output alphabet of a thermostat is $\{air\text{–}conditioner, do\text{–}nothing, furnace\}$. The states are $\{ready, heating, cooling, just\text{–}heated, just\text{–}cooled\}$. The initial state, transition function and response function are shown in Figure 6.1.

The thermostat uses the "just-cooled" and "just-heated" states to avoid going from running the air-conditioner to the furnace (or the reverse) abruptly. As an added benefit, the furnace and air-conditioner don't pop on and off; the "just" states slow the electronics down to where they don't hurt the poor machinery. If this delay were *not* needed we might be able to confuse the states and actions. Formally, you let the states be the set of actions and "do" whatever state you're in. A finite state automaton that does this is called a *Moore* machine. The more usual type of finite state automaton, with an explicitly separate response function, is termed a *Mealey* machine. In general, we will use the Mealey architecture.

Notice that the transition function t (shown in the second column of Figure 6.1), is a function from the set of ordered pairs of states and inputs to the set of states, i.e., $t(state, input)$ is a member of the set of states. The response function r (in the third column), is a function from the set of ordered pairs of states and inputs to the set of outputs, i.e., $r(state, input)$ is a member of the output alphabet.

Colloquially speaking, the automaton sits in a state until an input comes. When an input comes, the automaton then generates an output (with its response function) and moves to a new state (which is found by consulting the transition function). The initial response, not

Initial State: ready		
When current state and input are	make a transition to state	and respond by
(hot,ready)	cooling	air-conditioner
(hot,heating)	just-heated	do-nothing
(hot,cooling)	cooling	air-conditioner
(hot,just-heated)	ready	do-nothing
(hot,just-cooled)	ready	do-nothing
(okay,ready)	ready	do-nothing
(okay,heating)	just-heated	do-nothing
(okay,cooling)	just-cooled	do-nothing
(okay,just-heated)	ready	do-nothing
(okay,just-cooled)	ready	do-nothing
(cold,ready)	heating	furnace
(cold,heating)	heating	furnace
(cold,cooling)	just-cooled	do-nothing
(cold,just-heated)	ready	do-nothing
(cold,just-cooled)	ready	do-nothing

Figure 6.1: A thermostat as a finite state automaton

present in the thermostat, is used if the automaton must have some output even before it has an input to work with (a good initial response for the thermostat would be do-nothing).

For the remainder of this section, the input and output alphabets will both be $\{0, 1\}$ and the task will be to learn to predict the next bit of an input stream of bits.

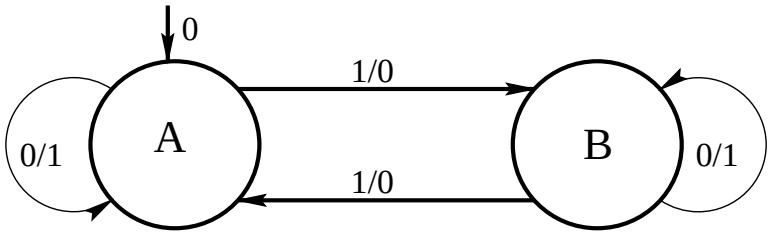

Figure 6.2: A finite state automaton diagram

A finite state automaton of this type is shown in Figure 6.2. It has two states, state "A" and state "B". The transition function is specified by the arrows in the diagram and the arrow labels are of the form *input/output*. The initial response is on an arrow that does not start at a state and which points to the initial state. This sort of diagram is handy for

representing automata on paper. Formally: the finite state automaton's response function is $r(A,0) = 1$, $r(A,1) = 0$, $r(B,0) = 1$, $r(B,1) = 0$ and the initial response is 0. Its transition function is $t(A,0) = A$, $t(A,1) = B$, $t(B,0) = B$, $t(B,1) = A$. The initial state is A.

Initial response:0		
Initial state:A		
State	If 0	If 1
A	1→A	0→B
B	1→B	0→A

Figure 6.3: A finite state automaton table

If we were to specify the finite state automaton shown in Figure 6.2 in a tabular format, the result would be as shown in Figure 6.3. This is not identical to the tabular format used in Figure 6.1. It is less explicit about the identity of the functions it is specifying and much easier to read. The table starts by giving the initial response and initial state of the finite state automaton. The rest of the table is a matrix with rows indexed by states and columns indexed by inputs. The entries of this matrix are of the form $response \rightarrow state$. This means that when the automaton is in the state indexing the row and sees the action indexing the column, it will make the response given at the tail of the arrow and then make a transition to the state at the arrow's head.

You may want to develop a computer data structure for representing finite state automata. You should definitely build a routine that can print an FSA in roughly the tabular form given in Figure 6.3; it will be an invaluable aid in debugging experiments.

So that we can perform crossover with finite state automata, we will describe them as a string of integers and then use the usual crossover operators for strings. We can either group the integers describing the transition and response functions together, termed *functional* grouping, or we can group the integers describing individual states together, termed *structural* grouping. In Example 6.3, both these techniques are shown. Functional grouping places the integers describing the transition function and those describing the response function in contiguous blocks, making it easy for crossover to preserve large parts of their individual structure. Structural groupings place descriptions of individual states of an FSA into contiguous blocks making their preservation easy. Which sort of grouping is better depends entirely on the problem being studied.

Example 6.3 *We will change the finite state automaton from Figure 6.3 into an array of integers in the structural and functional manners. First we strip a finite state automaton down to the integers that describe it (setting $A = 0$, $B = 1$) as follows:*

Initial response:0		
Initial state:A		
State	*If 0*	*If 1*
A	*1→A*	*0→B*
B	*1→B*	*0→A*

0		
0		
	1 0	*0 1*
	1 1	*0 0*

To get the structural grouping gene we simply read the stripped table from left to right, assembling the the integers into the array:

$$0010011100 \tag{6.1}$$

To get the functional gene we note the pairs of integers in the stripped version of the table, above, are of the form:

response transition

We thus take the first integer (the response) in each pair from left to right, and then the second integer (the transition) in each pair from left to right to obtain the gene:

$$0010100110. \tag{6.2}$$

Note that in both the functional and structural genes the initial response and initial state are the first two integers in the gene.

We also want a definition of point mutation for a finite state automaton. This turns out to be much easier than crossover. Pick at random any one of: the initial action, the initial state, any transition, or any response; replace it with a randomly chosen valid value.

Now we know how to do crossover and mutation, we can run an evolutionary algorithm on a population of finite state automata. For our first such evolutionary algorithm, we will use a task inspired by a Computer Recreations column in *Scientific American*. Somewhat reminiscent of the string evolver, this task starts with a reference string. We will evolve a population of finite state automata that can predict the next bit of the reference string as that string is fed to them one bit at a time.

We need to define the alphabets for this task, and the fitness function. The reference string is over the alphabet $\{0, 1\}$ which is also the input alphabet and the output alphabet of the automaton. The fitness function is called the *String Prediction fitness function*, computed as follows. Pick a reference string in $\{0, 1\}^*$ and a number of bits to feed the automaton. Bits beyond the length of the string are obtained by cycling back though the string again. Initialize fitness to zero. If the first bit of the string matches the initial response of the FSA, fitness is $+1$. After this, we use the input bits as inputs to the FSA, checking the output of the FSA against the next bit from the string; each time they match fitness is $+1$. The finite state automaton is being scored on its ability to correctly guess the next bit of the input.

Example 6.4 *Compute the String Prediction fitness of the finite state automaton in Figure 6.2 on the string 011 with 6 bits.*

Step	*FSM guess*	*String bit*	*State after guess*	*Fitness*
0	*0*	*0*	*A*	*+1*
1	*1*	*1*	*A*	*+1*
2	*0*	*1*	*B*	-
3	*0*	*0*	*A*	*+1*
4	*1*	*1*	*A*	*+1*
5	*0*	*1*	*B*	-
			Total fitness:	*4*

The String Prediction fitness function gives us the last piece needed to run our first evolutionary algorithm on finite state automata.

Experiment 6.1 *Write or obtain software for randomly generating, printing, and handling file input/output of finite state automata as well as the variation operators described above. Create an evolutionary algorithm using size 4 tournament selection, two point crossover, single point mutation, and String Prediction fitness. Use the structural grouping for your crossover. Run 30 populations for up to 1000 generations, recording time-to-solution (or the fact of failure), for populations of 100 finite state automata with,*

(i) Reference string 001, 6 bits, 4 state FSA,

(ii) Reference string 001111, 12 bits, 4 state FSA,

(iii) Reference string 001111, 12 bits, 8 state FSA.

Define "solution" to consist of having at least one creature whose fitness equals the number of bits used. Graph the fraction of populations that have succeeded as a function of the number of generations for all 3 sets of runs on the same set of axes.

Experiment 6.2 *Redo Experiment 6.1 with functional grouping used to represent the automaton for crossover. Does this make a difference?*

Let's try another fitness function. The *Self-Driving Length function* is computed as follows. Start with the finite state automaton in its initial state with its initial response. Thereafter, use the last response as the current input; use the automaton's output to drive its input. Eventually the automaton must simultaneously repeat both a state and response. The number of steps it takes to do this is its Self-Driving Length fitness.

Example 6.5 *For the following FSAs with input and output alphabet* $\{0, 1\}$*, find the Self-Driving Length fitness.*

Initial response:1		
Initial state:D		
State	*If 0*	*If 1*
A	*1*→*B*	*0*→*B*
B	*1*→*A*	*0*→*B*
C	*1*→*C*	*0*→*D*
D	*0*→*A*	*0*→*C*

Time-step by time-step:

Step	*Response*	*State*
1	*1*	*D*
2	*0*	*C*
3	*1*	*C*
4	*0*	*D*
5	*0*	*A*
6	*1*	*B*
7	*0*	*B*
8	*1*	*A*
9	*0*	*B*

So in time-step 9 the automaton finally repeats the response/state pair "0", "B". We therefore put its Self-Driving Length fitness at 8.

Notice that in our example we have *all* possible pairs of states and responses. We can do no better. This implies that success in the Self-Driving Length fitness function is a score of twice the number of states (at least over the alphabet $\{0, 1\}$).

Experiment 6.3 *Rewrite the software for Experiment 6.1 to use the Self-Driving Length fitness function. Run 30 populations of 100 finite state automata, recording time to success and cutting the automata off after 2000 generations. Graph the fraction of populations that succeeded after k generations, showing the fraction of failures on the left side of the graph as the distance below one. Do this experiment for automata with 4, 6, and 8 states. Also report the successful string, in those automata that do succeed.*

It is easy to write a finite state automaton description that does not use some of its states. The Self-Driving Length fitness function encourages the finite state automaton to

use as many transitions as possible. In Experiment 6.1, the string 001111, while possible for a 4-state automaton to predict, was difficult. The string 111110 would prove entirely impossible for a 4-state automaton (why?) and very difficult for a 6-state automaton.

There is a very large local optimum in Experiment 6.1 for an automaton that predicts the string 111110; automata that just churn out 1s get relatively high fitness in this environment. If we look at all automata that churn out only 1s, we see that they are likely to use few states. The more transitions involved, the easier to have one that is associated with a response of 0, either initially or by a mutation. A moment's thought shows, in fact, that 1-making automata that do use a large number of transitions are more likely to have children that don't, and so there is substantial evolutionary pressure to stay in the local optimum associated with a population of FSAs generating 1s, and using a small number of states to do so. This leaves only extremely low probability evolutionary paths to an automaton that predicts 111110.

Where possible, when handed lemons, make lemonade. In Chapter 5, we introduced the lexical product of fitness functions. When attempting to optimize for the String Prediction fitness function in difficult cases like 111110, the Self-Driving Length fitness function is a natural candidate for a lexical product; it lends much greater weight to the paths out of the local optimum described above. Let us test this intuition experimentally.

Experiment 6.4 *Modify the software from Experiment 6.1 to optionally use either the String Prediction fitness function or the lexical product of String Prediction with Self-Driving Length, with String Prediction dominant. Report the same data as in Experiment 6.1, but running 6- and 8-state automata with both the plain and lexical fitness functions on the reference string 111110 using 12 bits. In your write up, document the differences in performance and give all reasons you can imagine for the differences, not just the one suggested in the text.*

Experiment 6.4 is an example of an evolutionary algorithm in which lexical products yield a substantial gain in performance. Would having more states cause more of a gain? To work out the exact interaction between additional states and the solutions present in a randomly generated population, you would need a couple of stiff courses in finite state automata or combinatorics. In the next section, we will leave aside optimization of finite state automata and proceed with co-evolving finite state automata.

Problems

Problem 6.1 *Suppose that A is an alphabet of size n. Compute the size of the set $\{s \in A^* : |s| \leq k\}$ for any non-negative integer k.*

Problem 6.2 *How many strings are there in $\{0,1\}^{2m}$ with exactly m ones?*

Problem 6.3 *Notice that in Experiment 6.1 the number of bits used is twice the string length. What difference would it make if the number of bits were equal to the string length?*

Problem 6.4 *If we adopt the definition of success given in Experiment 6.1 for a finite state automaton on a string, is there any limit to the length of a string on which a finite state automaton with n states can succeed?*

Problem 6.5 *Give the structural and functional grouping genes for the following FSAs with input and output alphabet* $\{0, 1\}$.

(i)

Initial response:1		
Initial state:B		
State	If 0	If 1
A	1→A	1→C
B	1→B	0→A
C	0→C	0→A

(ii)

Initial response:1		
Initial state:A		
State	If 0	If 1
A	1→A	0→B
B	1→C	1→A
C	1→B	0→C

(iii)

Initial response:0		
Initial state:D		
State	If 0	If 1
A	1→B	1→D
B	1→C	0→A
C	0→D	1→B
D	0→A	0→C

(iv)

Initial response:0		
Initial state:D		
State	If 0	If 1
A	0→B	0→D
B	0→C	1→A
C	1→D	0→B
D	1→A	1→C

Problem 6.6 *For each of the finite state automata in Problem 6.5, give the set of all strings the automaton in question would count as a success, if the string were used in Experiment 6.1 with a number of bits equaling twice its length.*

Problem 6.7 *Prove that the maximum possible value for the Self-Driving Length fitness function of an FSA with input and output alphabet* $\{0, 1\}$ *is twice the number of states in the automaton.*

Problem 6.8 *Given an example that shows that Problem 6.7 does not imply that the longest string a finite state automaton can succeed on in the String Prediction fitness function is of length* $2n$ *for an* n *state finite state automaton.*

Problem 6.9 *In the text it was stated that a 4-state automaton* cannot *succeed, in the sense of Experiment 6.1, on the string 111110. Explain irrefutably why this is so.*

Problem 6.10 *Problems 6.7, 6.8, and 6.9 all dance around an issue. How do you tell if a string is too "complex" for an n state finite state automaton to completely predict? Do your level best to answer this question, over the input and output alphabet* $\{0, 1\}$.

Problem 6.11 *Work Problem 6.7 over assuming the finite state automaton uses the input and output alphabets* $\{0, 1, \ldots, n-1\}$. *You will have to conjecture what to prove and then prove it.*

6.2 The Prisoner's Dilemma I

The work in this section is based on a famous experiment of Robert Axelrod's concerning the Prisoner's Dilemma. The original Prisoner's Dilemma was a dilemma experienced by two accomplices, accused of a burglary. The local minions of the law are sure of the guilt of the two suspects they have in custody, but have only sufficient evidence to convict them of criminal trespass, a much less serious crime than burglary. In an attempt to get better evidence, the minions of the law separate the accomplices and make the same offer to both. The state will drop the criminal trespass charges and give immunity from any self-incriminating statements made, if the suspect will implicate his accomplice. There are 4 possible outcomes to this situation.

1 Both suspects remain mum, serve their short sentence for criminal trespass, and divide the loot.

2,3 One suspect testifies against the other, going off scot-free and keeping all the loot for himself. The other serves a long sentence as an unrepentant burglar.

4 Both suspects offer to testify against the other and receive moderate sentences because they are repentant and cooperative burglars. Each also keeps some chance at getting the loot.

In order to analyze the Prisoner's Dilemma, it is convenient to arithmetize these outcomes as numerical payoffs. We characterize the action of maintaining silence as *cooperation* and the action of testifying against one's accomplice as *defection*. Abbreviating these actions as C and D we obtain the payoff matrix for the Prisoner's Dilemma shown in Figure 6.4. Mutual cooperation yields a payoff of 3, mutual defection a payoff of 1, and stabbing the other player in the back yields a payoff of 5 for the stabber and 0 for the stabbee. These represent only one possible set of values in a payoff matrix for the Prisoner's Dilemma. Discussion of this and other related issues are saved for Section 6.3.

The Prisoner's Dilemma is an example of a *game* of the sort treated by the field of *game theory*. Game theory was invented by John von Neumann and Oskar Morgenstern. Their

		Player 2	
		C	D
Player 1	C	(3,3)	(0,5)
	D	(5,0)	(1,1)

Figure 6.4: Payoff matrix for the Prisoner's Dilemma

foundational text, *The Theory of Games and Economic Behavior*, appeared in 1953. Game theory has been widely applied to economics, politics, and even evolutionary biology. One of the earliest conclusions drawn from the paradigm of the Prisoner's Dilemma was somewhat shocking. To appreciate the conclusion von Neumann drew from the Prisoner's Dilemma, we must first perform the standard analysis of the game.

Imagine you are a suspect in the story we used to introduce the Prisoner's Dilemma. Sitting in the small, hot interrogation room you reflect on your options. If the other suspect has already stabbed you in the back, you get the lightest sentence for stabbing him in the back as well. If, on the other hand, he is maintaining honor among thieves and refusing to testify against you, then you get the lightest sentence (and all the loot) by stabbing him in the back. It seems that your highest payoff comes, in all cases, from stabbing your accomplice in the back. Unless you are altruistic, that is what you'll do.

At the time he and Morgenstern were developing game theory, von Neumann was advising the U.S. government on national security issues. A central European refugee from the Second World War, Von Neumann was a bit hawkish and concluded that the game theoretic analysis of the Prisoner's Dilemma indicated a nuclear first strike against the Soviet Union was the only rational course of action. It is, perhaps, a good thing that politicians are not especially respectful of reason. In any case, there is a flaw in von Neumann's reasoning. This flaw comes from viewing the "game" the U.S. and U.S.S.R. were playing as being exactly like the one the two convicts were playing. Consider a similar situation, again presented as a story, with an important difference. It was inspired by observing a parking lot across from the apartment the author lived in during graduate school.

Once upon a time in California, the police could not search a suspected drug dealer standing in a parking lot where drugs were frequently sold. The law required that they see the suspected drug dealer exchange something, presumably money and drugs, with a suspected customer. The drug dealers and their customers found a way to prevent the police from interfering in their business. The dealer would drop a plastic bag of white powder in the ornamental ivy beside the parking lot in a usual spot. The customer would, at the same time, hide an envelope full of money in a drain pipe on the other side of the lot. These actions were performed when the police were not looking. Both then walked with their best

"I'm not up to anything" stride, exchanged positions, and picked up their respective goods. This is quite a clever system as long as the drug dealer and the customer are both able to trust each other.

In order to cast this system into a Prisoner's Dilemma format, we must decide what constitutes a defection and a cooperation by each player. For the drug dealer, cooperation consists of dropping a bag containing drugs into the ivy, while defection consists of dropping a bag of cornstarch or baking soda. The customer cooperates by leaving an envelope of Federal Reserve Notes in the drain pipe and defects by supplying phony money or, perhaps, insufficiently many real bills. The arithmetization of the payoffs given in Figure 6.4 is still sensible for this situation. In spite of that, this is a new and different situation from the one faced by the two suspects accused of burglary.

Suppose the dealer and customer both think through the situation. Will they conclude that ripping off the other party is the only rational choice? No, in all probability, they will not. The reason for this is obvious. The dealer wants the customer to come back and buy again, tomorrow, and the customer would likewise like to have a dealer willing to supply him with drugs. The two players play the game many times. A situation in which two players play a game over and over is said to be *iterated*. The one-shot Prisoner's Dilemma is entirely unlike the Iterated Prisoner's Dilemma, as we will see in the experiments done in this section.

The Iterated Prisoner's Dilemma is the core of the excellent book *The Evolution of Cooperation* by Robert Axelrod. The book goes through many real life examples that are explained by the iterated game and gives an accessible mathematical treatment.

Before we dive into coding and experimentation, a word about altruism is in order. The game theory of the Prisoner's Dilemma, iterated or not, assumes that the players are not altruistic - that they are acting for their own self-interest. This is done for a number of reasons, foremost of which is the mathematical intractability of altruism. One of the major results of research on the Iterated Prisoner's Dilemma is that cooperation can arise in the absence of altruism. None of this is meant to denigrate altruism or imply it is irrelevant to the social or biological sciences. It is simply beyond the scope of this text.

In the following experiment we will explore the effect of iteration on play. A population of finite state automata will play Prisoner's Dilemma once, a small number of times, and a large number of times. A *round robin tournament* is a tournament in which each possible pair of contestants meet.

Experiment 6.5 *This experiment is similar to one done by John Miller. Write or obtain software for an evolutionary algorithm that operates on 4-state finite state automata with an initial response. Use $\{C, D\}$ for the input and output alphabets. The algorithm should use the same variation operators as in Experiment 6.1. Generate your initial populations by filling the tables of the finite state automata with uniformly distributed valid values.*

Fitness will be computed by playing a Prisoner's Dilemma round robin tournament. To

play, a finite state automata uses its current response as the current play, and the last response of the opposing automaton as its input. Its first play is thus its initial response. Each pair of distinct automata should play n rounds of Prisoner's Dilemma. The fitness of an automaton is its total score in the tournament. Start the automata over in their initial states with each new partner. Do not save state information between generations.

On a population of 36 automata, use roulette selection and absolute fitness replacement, replacing 12 automata in each generation for 100 generations. This is a strongly elitist algorithm with $\frac{2}{3}$ of the automata surviving in each generation. Save the average fitness of each population divided by $35n$ (the number of games played) in each generation of each of 30 runs.

Plot the average of the averages in each generation versus the generations. Optionally, plot the individual population averages. Do this for $n = 1$, $n = 20$, and $n = 150$. For which of the runs does the average plot most closely approach cooperativeness (a score of 3)? Also, save the finite state automata in the final generations of the runs with $n = 1$ and $n = 150$ for later use.

There are a number of strategies for playing the Prisoner's Dilemma that are important in analyzing the game and aid in discussion. Figure 6.5 lists several such strategies, and Figure 6.6 describes 5 as finite state automata. The strategies, Random, Always Cooperate, and Always Defect, represent extreme behaviors, useful in analysis. Pavlov is special for reasons we will see later.

The strategy, Tit-for-Tat, has a special place in the folklore of the Prisoner's Dilemma. In two computer tournaments, Robert Axelrod solicited computer strategies for playing the Prisoner's Dilemma from game theorists in a number of academic disciplines. In both tournaments, Tit-for-Tat, submitted by Professor Anatole Rapoport, won the tournament. The details of this tournament are reported in the second chapter of Axelrod's book, *The Evolution of Cooperation*.

The success of Tit-for-Tat is, in Axelrod's view, the result of four qualities. Tit-for-Tat is *nice*; it never defects first. Tit-for-Tat is *vengeful*; it responds to defection with defection. Tit-for-Tat is *forgiving*; given an attempt at cooperation by the other player it reciprocates. Finally, Tit-for-Tat is *simple*; its behavior is predicated only on the last move its opponent made and hence other strategies can adapt to it easily. Note that not all these qualities are advantageous in and of themselves, but rather they form a good group. Always Cooperate has three of these four qualities, and yet it is a miserable strategy. Tit-for-Two-Tats is like Tit-for-Tat, but nicer.

Before we do the next experiment, we need a definition that will help cut down the work involved. The *self-play string* of a finite state automaton with initial response is the string of responses the automaton makes playing against itself. This string is very much like the string of responses used for computing the Self-Driving Length fitness, but the string is not cut off at the first repetition of a state and input. The self-play string is infinite.

Random	The Random strategy simply flips a coin to decide how to play.
Always Cooperate	The Always Cooperate strategy always cooperates.
Always Defect	The Always Defect strategy always defects.
Tit-for-Tat	The strategy Tit-for-Tat cooperates as its initial response and then repeats its opponent's last action.
Tit-for-Two-Tats	The strategy Tit-for-Two-Tats cooperates for its initial response and then cooperates on any action in which its opponent's last two actions have not been cooperation.
Pavlov	The strategy Pavlov cooperates on its first action and then cooperates if its action and its opponent's actions matched last time.

Figure 6.5: Some common strategies for the Prisoner's Dilemma

Thinking about how finite state automata work, we see that the automaton might never repeat its first few responses and states. For any finite state automaton, the self-play string will be a (possibly empty) string of responses associated with state/input pairs that never happen again followed by a string of actions associated with a repeating sequence of states and responses. For notational simplicity, we write the self-play string in the form $string1 : string2$ where $string1$ contains the actions associated with unrepeated state/response pairs and $string2$ contains the actions associated with repeated state/action pairs. Examine Example 6.6 to increase your understanding.

Example 6.6 *Examine the automaton:*

Initial response:C		
Initial state:4		
State	*If D*	*If C*
1	*D→2*	*C→2*
2	*C→1*	*D→2*
3	*D→3*	*D→4*
4	*C→1*	*C→3*

Always Cooperate		
Initial response:C		
Initial state:1		
State	If D	If C
1	C→1	C→1

Always Defect		
Initial response:D		
Initial state:1		
State	If D	If C
1	D→1	D→1

Tit-for-Tat		
Initial response:C		
Initial state:1		
State	If D	If C
1	D→1	C→1

Tit-for-Two-Tats		
Initial response:C		
Initial state:1		
State	If D	If C
1	C→2	C→1
2	D→2	C→1

Pavlov		
Initial response:C		
Initial state:1		
State	If D	If C
1	D→2	C→1
2	C→1	D→2

Figure 6.6: Finite state automaton tables for common Prisoner's Dilemma strategies

The sequence of plays of this automaton against itself is:

Step	*Response*	*State*
1	*C*	*4*
2	*C*	*3*
3	*D*	*4*
4	*C*	*1*
5	*C*	*2*
6	*D*	*2*
7	*C*	*1*
...	...	...

The self-play string of this finite state automaton is:

CCD:CCD.

Notice that the state/action pairs $(4, C)$*,* $(3, C)$*, and* $(4, D)$ *happen exactly once while the state/action pairs* $(2, C)$*,* $(2, C)$*, and* $(1, D)$ *repeat over and over as we drive the automaton's input with its output. It is possible for two automata with different self-play strings to produce the same output stream when self-driven.*

In Experiment 6.6, the self-play string can be used as a way to distinguish strategies. Before doing Experiment 6.6, do Problems 6.19 and 6.20.

Experiment 6.6 *Take the final populations you saved in Experiment 6.5 and look through them for strategies like those described in in Figures 6.5 and 6.6. Keep in mind that states that are not used or that cannot be used are unimportant in this experiment. Do the following:*

(i) For each of the strategies in Figure 6.5, classify the strategy (or one very like it) as occurring often, occasionally, or never.

(ii) Call a self-play string dominant *if at least $\frac{2}{3}$ of the population in a single run has that self-play string. Find which fraction of the populations have a dominant strategy.*

(iii) Plot the histogram giving the number of self-play strings of each length, across all 30 populations evolved with $n = 150$.

(iv) Plot the histogram as in part (iii) for 1080 randomly generated automata.

In your write up, explain what happened. Document exactly which software tools you used to do the analyses above (don't, for goodness sake, do them by hand).

Experiment 6.6 is very different from the other experiments so far in Chapter 6. Instead of creating or modifying an evolutionary algorithm, we are sorting through the debris left after an evolutionary algorithm has been run. It is usually much harder to analyze an evolutionary algorithm's output than it is to write the thing in the first place. You should carefully document and save any tools you write for sorting through the output of an evolutionary algorithm so you can use them again.

We now want to look at the effect of models of evolution on the emergence of cooperation in the Iterated Prisoner's Dilemma.

Experiment 6.7 *Take the software from Experiment 6.5 and modify it so the the model of evolution is tournament selection with tournament size 4. Rerun the experiment for $n = 150$ and give the average of averages plot. Now do this all over again for tournament size 6. Explain any differences and also compare the two data sets with the data set from Experiment 6.5. Which of the two tournament selection runs is most like the run from Experiment 6.5?*

A strategy for playing a game is said to be *evolutionarily stable* if a large population playing that strategy cannot be invaded by a single new strategy mixed into the population. The notion of invasion is relative to the exact mechanics of play. If the population is playing round robin, for example, the new strategy would invade by getting a higher score in the round robin tournament.

The notion of evolutionarily stable strategies is very important in game theory research. The location of such strategies for various games is a topic of many research papers. The intuition is that the stable strategies represent attracting states of the evolutionary process. This means you would expect an evolving system to become evolutionarily stable with high probability once it had been going for a sufficient amount of time. In the next experiment, we will investigate this notion.

Both Tit-for-Tat and Always Defect are evolutionarily stable strategies for the Iterated Prisoner's Dilemma in many different situations. Certainly, it is intuitive that a group

playing one or the other of these strategies would be very difficult for a single invader to beat. It turns out that neither of these strategies is in fact stable under the type of evolution that takes place in an evolutionary algorithm.

Define the *mean failure time* of a strategy to be the average amount of time (in generations) it takes a population composed entirely of that strategy, undergoing evolution by an evolutionary algorithm, to be invaded. This number exists relative to the type of evolution taking place and is not ordinarily something you can compute. In the next experiment, we will instead approximate it.

Experiment 6.8 *Take the software from Experiment 6.7, for size 4 tournaments, and modify it as follows. Have the evolutionary algorithm take a single automaton and initialize the entire population to be copies of that automaton. Compute the average score per play that automaton gets when playing itself, calling the result the* baseline score*. Run the evolutionary algorithm until the average score in a generation differs from the baseline by 0.3 or more (our test for successful invasion) or until 500 generations have passed. Report the time-to-invasion and fraction of populations that resisted invasion for at least 500 generations for 30 runs for each of the following strategies:*

(i) Tit-for-Two-Tats,

(ii) Tit-for-Tat,

(iii) Always Defect.

Are any of these strategies stable under evolution? Keeping in mind that Tit-for-Two-Tats is not evolutionarily stable in the formal sense, also comment on the comparative decay rates of those strategies that are not stable.

One quite implausible feature of the Prisoner's Dilemma as presented in this chapter so far is the perfect understanding the finite state automata have of one another. In international relations or a drug deal there is plenty of room to mistake cooperation for defection or the reverse. We will conclude this section with an experiment that explores the effect of error on the Iterated Prisoner's Dilemma. We will also finally discover why Pavlov, not a classic strategy, is included in our list of interesting strategies. Pavlov is an example of an *error correcting* strategy. We say a strategy is error correcting if it avoids taking too much revenge for defections caused by error. Do Problem 6.15 by way of preparation.

Experiment 6.9 *Modify the software for Experiment 6.5 with* $n = 150$ *so that actions are transformed into their opposite with probability* α*. Run 30 populations for* $\alpha = 0.05$ *and* $\alpha = 0.01$*. Compare the cooperation in these populations with the* $n = 150$ *population from Experiment 6.5. Save the finite state automata from the final generation of the evolutionary algorithm and answer the following questions. Are there error correcting strategies in any*

of the populations? Did Pavlov arise in any of the populations? Did Tit-for-Tat? Detail carefully the method you used to identify these strategies.

We have barely scratched the surface of the ways we could explore the Iterated Prisoner's Dilemma with artificial life. You are encouraged to think up your own experiments. As we learn more techniques in later chapters, we will revisit the Prisoner's Dilemma and do more experiments.

Problems

Problem 6.12 *Explain why the average score over some set of pairs of automata that play Iterated Prisoner's Dilemma with one another is in the range* $1 \leq \mu \leq 3$.

Problem 6.13 Essay. *Examine the following finite state automaton. We have named the strategy encoded by this finite state automaton* Ripoff. *It is functionally equivalent to an automaton that appeared in a population containing immortal Tit-for-Two-Tat automata. Describe its behavior colloquially and explain how it interacts with Tit-for-Two-Tats. Does this strategy say anything about Tit-for-Two-Tats as an evolutionarilly stable strategy?*

Initial response:D		
Initial state:1		
State	*If D*	*If C*
1	*C→3*	*C→2*
2	*C→3*	*D→1*
3	*D→3*	*C→3*

Problem 6.14 *Give the expected (when the random player is involved) or exact score for 1000 rounds of play for each pair of players drawn from the set:*

{Always Cooperate, Always Defect, Tit-for-Tat, Tit-for-Two-Tats, Random, Ripoff}.

Ripoff is described in Problem 6.13. Include the pair of a player with itself.

Problem 6.15 *Assume we have a population of strategies for playing Prisoner's Dilemma consisting of Tit-for-Tats and Pavlovs. For all possible pairs of strategies in the population, give the sequence of the first 10 plays, assuming the first player's action on round 3 is accidentally reversed. This requires investigating 4 pairs since it matters which type of player is first.*

Problem 6.16 *Find an error correcting strategy other than Pavlov.*

Problem 6.17 *Assume there is a 0.01 chance of an action being the opposite of what was intended. Give the expected score for 1000 rounds of play for each pair of players drawn from the set {Always Cooperate, Always Defect, Tit-for-Tat, Tit-for-Two-Tats, Pavlov, Ripoff}. Ripoff is described in Problem 6.13. Include the pair of a player with itself.*

Problem 6.18 *Give a finite state automaton with each of the following self-play strings.*

(i) :C,

(ii) D:C,

(iii) C:C,

(iv) CDC:DDCCDC.

Problem 6.19 *Show that if two finite state automata have the same self-play string then the self-play string contains the moves they will use when playing one another.*

Problem 6.20 *Give an example of 3 automata such that the first 2 automata have the same self-play string, but the sequences of play of each of the first 2 automata against the 3rd differ.*

Problem 6.21 *In Problem 6.13, we describe a strategy called Ripoff. Suppose we have a group of 6 players playing round robin with 100 plays per pair. If players do not play themselves, compute the scores of the players for each possible mix of Ripoff, Tit-for-Tat, and Tit-for-Two-Tats containing at least one of all 3 player types. There are 10 such groupings.*

Problem 6.22 Essay. *Outline an evolutionary algorithm that evolves Prisoner's Dilemma strategies that does not involve finite state automata. You may wish to use a string based gene, a neural net, or some exotic structure.*

Problem 6.23 *For each of the finite state automata given in Figure 6.6 together with the automaton* Ripoff *given in Problem 6.13, state which of the following properties the strategy encoded by the automaton has: niceness, vengefulness, forgiveness, simplicity. These are the properties to which Axelrod attributes the success of the strategy Tit-for-Tat (see page 149).*

6.3 Other Games

In this section, we will touch briefly on several other games that are easily programmable as artificial life systems. Two are standard modifications of the Prisoner's Dilemma, the third is a very different game called *Divide the Dollar*.

The payoff matrix we used in Section 6.2 is the classic matrix appearing on page 8 of *The Evolution of Cooperation*. It is not the only one that game theorists allow. Any payoff

matrix of the form given in Figure 6.7 for which $S < Y < X < R$ and $S + R < 2X$ is said to be a payoff matrix for the Prisoner's Dilemma. The ordering of the 4 payoffs is intuitive. The second condition is required to make alternation of cooperation and defection worth less than sustained cooperation. We will begin this section by exploring the violation of that second constraint.

The *Graduate School* game is one like Prisoner's Dilemma, save that alternating cooperation and defection scores higher, on average, than sustained cooperation. The name is intended to suggest a married couple, both of whom wish to go to graduate school. The payoff for going to school is higher then the payoff for not going, but attending at the same time causes hardship. For the iterated version of this game, think of two preschoolers with a tricycle. It is more fun to take turns than it is to share the tricycle, and both those options are better than fighting over who gets to ride. We will use the payoff matrix given in Figure 6.8.

For the Graduate School game, we must redefine out terms. Complete cooperation consists of two players alternating cooperation and defection. Partial cooperation is exhibited when players both make the cooperate play together. Defection describes two players defecting.

Experiment 6.10 *Take the software from Experiment 6.7 and change the payoff matrix to play the Graduate School game. As in Experiment 6.5, save the final ecologies. Also, count the number of generations in which an ecology has a score above 3; these are generations in which it is clear there is complete cooperation taking place. Answer the following questions.*

(i) Is complete cooperation rare, occasional, or common?

(ii) Is the self-play string histogram materially different from that in Experiment 6.6?

(iii) What is the fraction of the populations which have a dominant strategy?

A game is said to be *optional* if the players may decide if they will or will not play. Let us construct an optional game built upon the Iterated Prisoner's Dilemma by adding a third

		Player 2	
		C	D
Player 1	C	(X,X)	(S,R)
	D	(R,S)	(Y,Y)

Figure 6.7: General payoff matrix for Prisoner's Dilemma (Prisoner's Dilemma requires that $S < Y < X < R$ and $S + R < 2X$.)

		Player 2 C	Player 2 D
Player 1	C	(3,3)	(0,7)
	D	(7,0)	(1,1)

Figure 6.8: Payoff Matrix for the Graduate School game

move called "Pass." If either player makes the play "Pass," both score 0, and we count that round of the game as not played. Call this game the Optional Prisoner's Dilemma. The option of refusing to play has a profound effect on the Prisoner's Dilemma as we will see in the next experiment.

Experiment 6.11 *Modify the software from Experiment 6.5 with $n = 150$ to work on finite state automata with initial response and input and output alphabets $\{C, D, P\}$. Scoring is as in the Prisoner's Dilemma, save that if either player makes the P move, then both score zero. In addition to a player's score, save the number of times he actually played instead of passing or being passed by the other player. First, run the evolutionary algorithm as before, with fitness equal to total score. Next, change the fitness function to be score divided by number of plays. Comment on the total level of cooperation as compared to the non-optional game and also comment on the differences between the two types of runs in this experiment.*

At this point, we will depart radically from the Iterated Prisoner's Dilemma to games with a continuous set of moves. The game *Divide the Dollar* is played as follows. An infinitely wealthy referee asks two players to write down what fraction of a dollar they would like to have for their very own. Each player writes a bid down on a piece of paper and hands the paper to the referee. If the bids total at most one dollar, the referee pays both players the amount they bid. If the bids total more than a dollar, both players receive nothing.

For now, we will keep the data structure for playing Divide the Dollar simple. A player will have a gene containing 6 real numbers (yes, we will allow fractional cents). The first is the initial bid. The next 5 are the amount to bid if the last pay out p (in cents) from the referee was 0, $0 < p \leq 25$, $25 < p \leq 50$, $50 < p \leq 75$, or $p > 75$, respectively.

Experiment 6.12 *Build an evolutionary algorithm by modifying the software from Experiment 3.1 to work on the 6 number genome for Divide the Dollar given above. Set the maximum mutation size to be* 3.0*. Take the population size to be 36. Replace the fitness function with the total cash a player gets in a round robin tournament with each pair playing 50 times. Run 50 populations, saving the average fitness and the low and high bid accepted*

in each generation of each population, for 60 generations. Graph the average, over the populations, of the per generation fitness and the high and low bids.

One could argue that high bids in Divide the Dollar are a form of defection and that bids of 50 (or not far below) are a form of cooperation. Low bids, however, are a form of capitulation and somewhat akin to cooperating with a defector. From this discussion, it seems that single moves of divide the dollar do not map well onto single moves of Prisoner's Dilemma. If we define cooperation to be making bids that result in a referee payout we can draw one parallel, however.

Experiment 6.13 *Following Experiment 6.7, modify the software from Experiment 6.12 so that it also saves the fraction of bids with payouts in each generation. Run 30 populations as before and graph the average fraction of acceptance of bids per generation over all the populations. Modify the software to use tournament selection with tournament size 6 and do the experiment again. What were the effects of changing the tournament size? Did they parallel Experiment 6.7?*

There are an infinite number of games we could explore, but we have done enough for now. We will return to game theory in future chapters once we have developed more artificial life machinery. If you have already studied game theory, you will notice that the treatment of the subject in this chapter differers violently from the presentation in a traditional game theory course. The approach is experimental (an avenue only recently opened to students by large, cheap digital computers) and avoids lengthy and difficult mathematical analyses. If you found this chapter interesting or entertaining, you should consider taking a mathematical course in game theory. Such a course is sometimes found in a math department, occasionally in a biology department, but most often in an economics department.

Problems

Problem 6.24 *In the Graduate School game, is it possible for a finite state automaton to completely cooperate with a copy of itself? Prove your answer. Write a paragraph about the effect this might have on population diversity as compared to the Prisoner's Dilemma.*

Problem 6.25 *Suppose we have a pair of finite state automata of the sort we used to play Prisoner's Dilemma or the Graduate School game. If the automata have n states, what is the longest they can continue to play before they repeat a set of states and actions they were both in before. If we were to view the pair of automata as a single finite state device engaged in self play, how many states would it have and what would be its input and output alphabet?*

Problem 6.26 *Take all of the one-state finite state automata with initial response and input and output alphabets $\{C, D\}$, and discuss their quality as strategies for playing the Graduate School game. Which pairs work well together? Hint: there are 8 such automata.*

Problem 6.27 Essay. *Explain why it is silly to speak of a single finite state automaton as coding a good strategy for the Graduate School game.*

Problem 6.28 *Find an error correcting strategy for the Graduate School game.*

Problem 6.29 Essay. *Find a real life situation to which the Optional Prisoner's Dilemma would apply and write up the situation in a fashion like the story of the drug dealer and his customer in Section 6.2.*

Problem 6.30 *Are the data structures used in Experiments 6.12 and 6.13 finite state automata? If so, how many states do they have and what are their input and output alphabets.*

Problem 6.31 *Is a pair of the data structures used in Experiments 6.12 and 6.13 a finite state automaton? Justify your answer carefully.*

Problem 6.32 Essay. *Describe a method of using finite state automata to play Divide the Dollar. Do* not *change the set of moves in the game to a discrete set, e.g., the integers 1-100, and then use that as the automaton's input and output alphabet. Such a finite state automaton would be quite cumbersome, and more elegant methods are available. It is just fine to have the real numbers in the range 0-100 as your output alphabet, you just cannot use them directly as input.*

Problem 6.33 *To do this problem you must first do Problem 6.32. Assume that misunderstanding a bid in Divide the Dollar consists of replacing the bid b with $(100 - b)$. Using the finite state system you developed in Problem 6.32 explain what an error correcting strategy is and give an example of one.*