Evolution of Strategies for Resource Protection Problems

William M. Spears and Diana F. Gordon

Adaptive Systems Group
AI Center - Code 5515
Naval Research Laboratory
Washington, DC 20375
{*E-mail :*} {*spears,gordon*}*@aic.nrl.navy.mil*

Abstract. The objective of this project is to develop effective finite-state machine (FSM) strategies for winning against an adversary in a Competition for Resources simulation. To achieve this goal, we evolve these strategies in a simulated environment and compare a variety of evolutionary methods in this context. Key empirical questions are addressed, such as how many FSM states are optimal, how effective is it to use an evolutionary algorithm that adapts the number of states, and how can one reduce the variance in fitness evaluation? Some of our experimental answers to these questions are quite intriguing. This paper also explores and evaluates novel algorithms for detecting and repairing deleterious cycles in the evolved FSMs.

1 Introduction

We are becoming increasingly dependent on large interconnected networks for the control of our resources, such as the Internet, communications networks, and power grids. The advantage of these networks is the ability to route resources in a reasonably optimal fashion. However, their interconnectivity, coupled with the lack of global view of what is happening in these networks, can lead to tremendous problems in network reliability. For example, small local failures can easily propagate to entire networks, causing loss of service and corruption of data. Also, deliberate attacks (such as "denial of service" attacks) can also easily cause widespread havoc (Denning, 1999).

Thus one important issue is the development of effective network traversal strategies to protect as many resources as possible from failure and/or attacks, i.e., to maximally restrict the number of resources damaged. To address this issue we have decided to create a novel "resource protection" simulation that captures the essential aspects of this problem. A "defender" agent attempts to protect resources before they are damaged by an intentional (or unintentional) "adversary".

Our primary goal then is to create sophisticated reactive strategies for the defender. We use finite-state machines (FSMs) for our strategies, since there are a number of precedents for FSMs being effective strategies for adversarial situations. We use evolutionary algorithms (EAs) to evolve the FSMs, since

there is ample precedent for the effectiveness of this approach (Fogel, Owens, and Walsh, 1966; Fogel, 1995; Fogel, 1999).[1] A secondary goal of our research is to highlight a number of important issues with respect to the evolution of FSMs: their representation, the application of mutation and recombination, methods for adapting the number of states in the FSM, and the removal of unproductive cyclic behavior from the FSMs in the context of resource protection problems.

2 The Competition for Resources Problem

Our current Competition for Resources simulation is a two-player game on a board of squares. Each square corresponds to a resource, and the two players compete for squares on the board. One player is the defender while the other is the adversary. At the beginning of each game the defender occupies one square on the board, the adversary occupies another square on the board, and the remainder of the squares are not occupied. If the board is of size $N \times N$, then the defender will start at square (1,1) and the adversary will start at square (N,N). For example, if the defender is depicted with a circle and the adversary with a square, the beginning of a 5×5 board game would appear as shown in Figure 1.

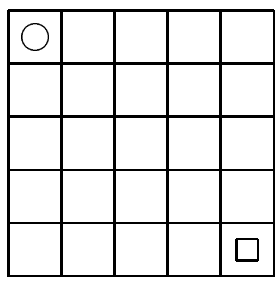

Fig. 1. The board at the beginning of the game.

Since the board grid represents real networks, such as power grids or communication networks, and in the real world networks may be highly interconnected and will have few geophysical boundaries, our board is toroidal (has no edges). Furthermore, no assumption is made about board size, thus allowing us to scale to large problems.

Each player can only perceive a limited amount of information, namely, the status of the squares neighboring the current position of the player. The neighbors consist of the four squares to the north, south, east and west of the current position. The diagonal squares can not be seen. The status of

[1] The evolution of FSMs is generally referred to as "evolutionary programming" in the literature.

each neighboring square will be one of the following: unoccupied, occupied by that player, or occupied by the opponent. It is important to point out that we use the global view provided in Figure 1 only as a visualization tool; the players do not see this global view. Due to the toroidal nature of the board, the defender and the adversary are actually quite close to one another at the beginning of the game. However, because they can not see along diagonals, they can't see one another initially.

In the current simulation, the two players alternate making moves. First a player examines its neighboring squares. Then it moves to one of the neighboring squares. A player can move to an unoccupied square or back to a square that it has previously occupied. It can not move to a square occupied by the opponent. At each time step, each of the players takes one action, which consists of moving to a neighboring resource to control/protect that resource. The player is not allowed to "stand still" and make no move. However, because each player must follow a path of "owned" resources to its current position, it will always be able to make a move at every time step (it can always back up along the path it has taken). Thus a player can not be "trapped", i.e., it can not be completely surrounded by the opponent. Once an agent occupies a resource, it controls/protects that resource forever. A game ends when all squares are occupied or time runs out. The agent with the most resources at the end of the game wins.

To continue the example shown in Figure 1, suppose the defender moves first. Each of the four neighboring squares are unoccupied. Suppose the defender randomly decides to move west (again, remember that the board is toroidal). The board is shown in Figure 2. The open circle represents the current position of the defender. The filled circle represents the fact that the defender has previously occupied this position and now "owns" it.

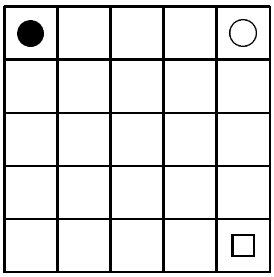

Fig. 2. The board after the defender moves.

Now the adversary moves. In this case only three neighboring squares are unoccupied; the southern neighbor is occupied by the defender. Thus the adversary can only move north, east, or west. If it moves west the board is as shown in Figure 3. The open square represents the current position of the adversary. The filled square represents the fact that the adversary has

previously occupied this position and now "owns" it. As mentioned above, the two players continue to move until all squares are occupied or time expires.

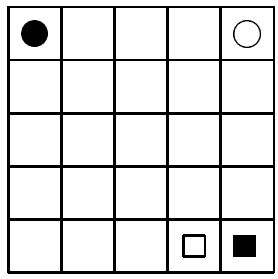

Fig. 3. The board after the adversary moves.

Throughout this chapter the adversary will have a fixed stochastic strategy that the defender must learn to defeat. The strategy we have chosen for the adversary is simple, but is surprisingly hard to beat. If the adversary detects any unoccupied neighboring squares, it uniformly randomly moves to one of them. Otherwise it uniformly randomly backtracks to a neighboring square it has previously occupied. We view this as an opportunistic but stochastic opponent. Given the game and our adversary, we focus on developing effective strategies for the defender. The form of these strategies is finite-state machines (FSMs), as described in the next section.

3 Overview of Finite State Machines

Our choice of FSMs for representing strategies was motivated by three main considerations. First, FSMs have proven to be effective representations of agent plans/strategies, e.g., see Carmel and Markovitch (1996) or Jefferson et al. (1991). Second, unlike classical plans (Dean and Wellman, 1991), FSMs allow for indeterminate-length action sequences. Recall from Hopcroft and Ullman (1979) that the usual acceptance criterion for finite-length strings is termination in a "final" state. Here we assume that there are no final states, i.e., action sequences of any length are allowed. This provides a good model of embedded agents that are continually responsive to their environment. Finally, FSMs divide an overall task into subtasks, represented as internal states. This facilitates the understanding of agent strategies (see, for example, Section 7).

Formally, we define the machine M to be a six-tuple $(Q, \Sigma, \Delta, \delta, \lambda, q_0)$. Q is the set of vertices (states) of M, Σ is the alphabet of input symbols (which are agent sensory inputs), and Δ is the alphabet of output symbols (which are agent actions). δ is the transition function from a state and an input to a next state, i.e., $\delta(q_i, x_i) = q_{i+1}$ where q_i, $q_{i+1} \in Q$ and $x_i \in \Sigma$ is

a sensory input. An edge is denoted as (q_i, q_{i+1}). For example, the edge from STATE1 to STATE2 in Figure 4 is (STATE1, STATE2). λ is the transition function from a state and an input to an output, i.e., $\lambda(q_i, x_i) = a_i$ where $q_i \in Q$, $x_i \in \Sigma$, and $a_i \in \Delta$ is an action. Finally, q_0 is the initial state, where all behavior begins.[2]

We assume that the FSMs are *deterministic* and *complete*. The FSMs are deterministic because δ and λ are functions, i.e., for every state and input there is a unique next state and action. The FSMs are complete because there exists a next state and action for every state and input, i.e., δ and λ are fully defined. Deterministic and complete FSMs are strategies that tell the agent precisely what to do in every situation it perceives.

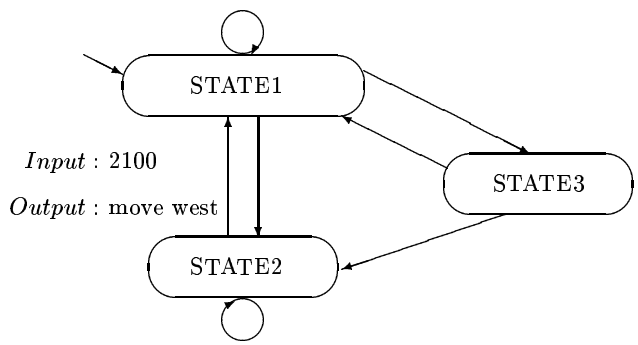

Fig. 4. An FSM agent plan for the Competition for Resources simulation, where the Input and Output are shown for only one of the edges.

Consider Figure 4, which is an example of an FSM strategy for the Competition for Resources simulation. Each sensory input x_i in this case shows the status of the neighboring resource immediately to the north, east, south, and west of the agent. The status of each resource can be 0 (unoccupied), 1 (occupied by the defensive agent), or 2 (occupied by the adversary). Thus an input of "2100" specifies that the north resource is owned by the adversary, the east resource is owned by the defensive agent, and that the south and west resources are unoccupied. Thus, according to the FSM depicted in Figure 4, if the defensive agent is in STATE2 and it sees an input of "2100" it will move west (λ (STATE2 , 2100) = west) and it will transition to STATE1 (δ (STATE2 , 2100) = STATE1). The initial state for this FSM is STATE1.

4 Evolution of Finite State Machines

As mentioned earlier, the goal of this chapter is to develop effective strategies for the defender in the context of the Competition for Resources simulation

[2] This definition of FSM M corresponds to what is called a *Mealy machine* in Hopcroft and Ullman (1979).

```
procedure EA;
t = 0; /* Initial Generation */
initialize_population(t);
evaluate(t);
until (done) {
        t = t + 1; /* Next Generation */
        selection(t);
        recombine(t);
        mutate(t);
        evaluate(t);
}
```

Fig. 5. The outline of an evolutionary algorithm.

outlined above. The defender will be modeled with an FSM. We focus on the application of evolutionary algorithms (EAs) to evolve the FSMs, since there is ample precedent for the effectiveness of this approach (Fogel, Owens, and Walsh, 1966; Fogel, 1995; Fogel, 1999). This section outlines precisely how we evolve the FSMs, by describing the representation used and the operators applied. First, however, a brief overview of EAs is in order.

4.1 A brief overview of EAs

EAs are population-based search algorithms. They maintain a population of individual structures that evolve according to rules of Darwinian selection and other operators, such as recombination and mutation. Each individual in the population is evaluated, receiving a measure of its fitness in the environment. Selection focuses attention on high-fitness individuals, thus exploiting the available fitness information. Recombination and mutation perturb those individuals, providing general heuristics for exploration. Although simplistic from a biologist's viewpoint, these algorithms are sufficiently complex to provide robust and powerful adaptive search mechanisms.

Figure 5 outlines a typical EA. A population of P individual structures is initialized and then evolved from generation t to generation $t+1$ by repeated applications of fitness evaluation, selection, recombination, and mutation. The population size P is generally constant in an EA, although there is no *a priori* reason (other than convenience) to make this assumption.

An EA typically initializes its population randomly, although domain-specific knowledge can also be used to bias the search. Evaluation measures the fitness of each individual according to its worth in some environment. Selection chooses the best individuals (those with high fitness) for survival and allows them to have children. Children are created via recombination, which exchanges information between parents, and mutation, which further perturbs the children. The children are then evaluated.

In the context of the Competition for Resources simulation, each individual in the population is an FSM. Each FSM is evaluated by playing the game numerous times, to obtain an estimate of how well that FSM is defending the resources against the adversary. Those FSMs that perform the task better are allowed to have more children, which are created through the processes of mutation and recombination. This process continues generation by generation, until the search is terminated.

The application of an EA to any problem involves a number of design decisions. For example, one has to choose a reasonable representation for the individual. The method of mutation and recombination must be precisely defined, as must be the form of selection and the termination criterion. These processes may depend on domain-specific knowledge, as mentioned above. Finally, the evaluation of the fitness of the individuals must be clearly described.

4.2 Representation of FSMs

The first design decision is the choice of internal representation for an FSM. Although at a high level all reasonable choices are semantically equivalent, a good data structure promotes efficiency. For this reason we chose a simple tabular representation, as exemplified in Table 1, which shows a simple FSM with three states ($Q = \{q_0, q_1, q_2\}$), three inputs ($\Sigma = \{x_0, x_1, x_2\}$), and two actions ($\Delta = \{a_0, a_1\}$).

	x_0	x_1	x_2
q_0	q_1/a_1	q_2/a_0	q_1/a_1
q_1	q_2/a_0	q_0/a_1	q_1/a_0
q_2	q_0/a_0	q_1/a_1	q_2/a_1

Table 1. The transition table for a simple three-state FSM. Rows are states and columns are inputs. Table entries are denoted as a next-state/action pair.

Rows in Table 1 correspond to states, and columns correspond to inputs. For each state q_i and input x_j, the corresponding table entry is a pair. The first element of this table entry pair is the next state, i.e., it is $\delta(q_i, x_j)$. The second element is the action to take given the agent is in state q_i and sees input x_j, i.e., it is $\lambda(q_i, x_j)$. Again, q_0 is the initial state (although this is not shown in the table).

In the context of the Competition for Resources simulation, the number of states is user defined, and will vary from one to S. The number of possible inputs is $3^4 = 81$, since the status of each of the four neighboring squares may have three values (0 = unoccupied, 1 = occupied by the defensive agent, 2 = occupied by the adversary). It should be noted that the input "2222" will never occur, since that implies that the defender is surrounded by the

adversary. This is impossible, since the defender must have been able to get to the square it currently occupies. Thus there are 80 possible inputs and we require a table of size $S \times 80$. Each entry in the table is an "allele" that represents a next-state/action pair.[3] The states are labeled from 1 to S and in this chapter the initial state is always STATE1.

4.3 Initialization and the use of domain knowledge

As mentioned above, the EA population is of size P. Thus, each of the P FSMs at generation zero must be initialized either randomly or by using domain-specific knowledge. Initialization defines the functions δ and λ for each FSM. For any given state (row) and input (column) the next state (given by δ) is chosen uniformly randomly from the set of all states Q.

The choice of action (given by λ) is somewhat more complex. The number of possible actions is maximally four, since the defender may potentially move north, east, south, or west. However, in practice, some of these moves might be impossible, if the adversary owns the neighboring squares.

	2100	2101	2102
STATE1	STATE2/south	STATE1/south	STATE3/south
STATE2	STATE1/west	STATE2/south	STATE1/south
STATE3	STATE2/west	STATE3/south	STATE2/south

Table 2. Example of a portion of an FSM transition table at initialization, using only preferred actions. Only one action (south) is preferred for inputs 2101 and 2102.

For example, recall the prior example where the input was "2100". In this case the north resource is owned by the adversary, the east resource is owned by the defensive agent, and the south and west resources are unoccupied. In this case there are only three *legal* moves: east, south, and west. The move to the north is *illegal*, since the adversary owns that square. Naturally, we must always restrict actions to those that are legal, and every input has a set of legal moves Δ_l that are possible. However, since the goal of the game is to capture resources, we also found it useful to define *preferred* moves – those that capture previously unoccupied squares. In the prior example where the input is "2100", moves to the south or west will capture new territory and are thus preferable. During initialization actions are always chosen uniformly randomly from the set of preferred actions Δ_p, if there are any. If there are no preferred actions, then a legal action is randomly chosen.[4] As an example,

[3] We use the term "allele" because individuals in EAs are often considered to be chromosomes of alleles, in an analogy to genetics.

[4] Preliminary experiments indicated that the emphasis on preferred actions enormously helps the initial search of the EA.

Table 2 shows three columns of an initial FSM with three states, under the inputs "2100", "2101", and "2102".

4.4 Mutation and recombination of FSMs

Mutation and recombination are operators that can alter the functions δ and λ. The mutation of an FSM is reasonably straightforward. Each allele (next-state/action pair) in the FSM is mutated with probability p_m. Once an allele is chosen for mutation a coin is flipped to see whether the action or the next state is mutated. With probability p the next state is mutated by uniformly randomly choosing a state from the set of all states Q. This could result in no change, with probability $1/S$. With probability $1-p$ the action is mutated. If there are any preferred actions Δ_p the algorithm uniformly randomly chooses one of those. If there are no preferred actions the algorithm uniformly chooses any legal action Δ_l. Note that this also could result in no change (e.g., if there is only one legal action then there *will* be no change). In this chapter we have set $p = 0.5$. In the future we plan to test whether there is any advantage to having both the next state and the action be mutated independently.

Recombination is also straightforward. A proportion p_r of pairs of parents in the population are chosen for recombination (p_r is referred to as the "recombination rate"). For each pair of parents, a coin is flipped for each of the $S \times 80$ alleles. The allele at the table location (i,j) in the first FSM is swapped with the corresponding allele in the second FSM, with probability P_0.[5] Note that since only corresponding alleles are swapped, there is no need to worry about possible illegal actions. If an action is legal for one FSM at location (i,j) it must be legal for any other FSM at location (i,j), since the input j is the same. Note also that if alleles are swapped, both the next state and the action are swapped. In the future we plan to test whether there is any advantage to having the next state and action be swapped independently.

4.5 The method of selection and termination

For this chapter we have chosen a standard fitness-proportional selection mechanism (Holland, 1975). We also made sure that the population contained a copy of the best individual that has ever been seen. This is referred to as "elitism" and it often helps to bias the search towards promising areas of the search space. For a termination criterion we simply ran the EA for a user-defined number of generations.

4.6 Evaluation of FSMs

Since the adversary in the Competition for Resources simulation is stochastic, each defender FSM will have to play the game multiple times in order to

[5] This is simply the well-known parameterized uniform recombination often used in the literature (Spears and De Jong, 1991).

obtain an estimate of how well it defends the resources. As stated earlier, the defender wins the game if it controls more resources than the adversary at the end of the game. Otherwise the adversary wins. The adversary also wins if both players control the same number of resources at the end of the game (i.e., there is a "tie").

The fitness function used by the EA is simple. Given G games, the fitness is the fraction of games that the defender wins. Thus this function returns values from 0.0 to 1.0, with 1.0 representing an FSM that won all the games it played. A major issue is the setting of G. If G is small, the fitness function will return a relatively poor estimate of the true worth of the defender. If G is large, the estimate will be much better, but at the expense of valuable computational cycles that might be better spent on running the EA for more generations. Prior work by Grefenstette and Fitzpatrick (1985) concluded that in some cases the overall efficiency of the EA may be improved by reducing G and by running for more generations.

In order to test this hypothesis we tried a wide range of values of G, ranging from 100 to 10,000. Due to high variance, G had to be quite large (on the order of 10,000) in order to obtain good estimates of the true fitness of an FSM. Unfortunately, setting G to 10,000 slowed down the EA so much that it took too long to run. With lower G the EA ran much faster, but there was a high probability that an FSM would do well purely due to stochastic noise, thus biasing the search towards FSMs that actually had poorer fitness. Equally unfortunately, running the EA for more generations did nothing to solve this difficulty and we were unable to confirm the hypothesis made by Grefenstette and Fitzpatrick.

Thus, on one hand a low value of G made for a fast fitness computation but produced FSMs that actually had inferior fitness. On the other hand a high value of G produced an accurate measure of fitness, but we were unable to run the EA for an adequate number of generations to produce good results. We were also unable to balance these two constraints adequately by picking some medium value of G. Instead we took another approach. Each individual would get a quick evaluation by using a low value of G. If that individual was promising (it did better than the best individual seen thus far), it was re-evaluated using a high value of G. If it still beat the best individual thus far, it became the new best individual. The idea was to carefully evaluate only those individuals that appeared promising. This approach worked quite well. We used a value of $G = 500$ for the initial evaluation and $G = 10{,}000$ for the subsequent re-evaluation (if it was performed). Since most individuals were unable to beat the best individual seen thus far, they were not re-evaluated. As a result the algorithm ran much more quickly, but the fitnesses of the best individuals were quite accurate, biasing the search in promising directions.[6]

[6] This is similar in spirit to performing a "secondary search" of promising nodes in minimax search (Rich and Knight, 1991).

5 Experimental Methodology

One topic that we have not discussed thus far is whether the EA will also be effective at evolving the number of states in an FSM. Traditional approaches (Fogel, Owens, and Walsh, 1966; Fogel, 1995; Fogel, 1999) to the evolution of FSMs have always included operators for adding and deleting states in an FSM. Although we were primarily motivated to generate good defenders for the Competition for Resources task, we found that the issue of how to properly add and delete states raised a number of interesting questions that we considered worth some investigation.

The first question is that when a state is deleted, should one actually erase the state from the table, or should it simply be made inaccessible? Secondly, once a state has been deleted (or has been made inaccessible), the portions of the FSM that point to this state must now be "repaired". How is this repair done? Third, if a state has simply been made inaccessible, is it still subject to mutation and recombination? Finally, when a state is added, is this done by making a former state accessible again, or is a new state randomly initialized?

Although we realized that we could not hope to address all of these questions adequately in this chapter, what became clear is that in order to even deal with these issues we needed a way to evaluate the efficacy of different mechanisms for evolving the number of states in an FSM. Thus we decided that a control study was needed, in which we fixed the number of states. Once we could see the performance of the EA with a fixed number of states, we would be better equipped to judge how well the EA was performing when it adapted the number of states.

5.1 Fixed number of states

For our experiments we had to select a reasonable board size N for the Competition for Resources simulation. The evaluation of the fitness function is the most expensive component of the EA, and the computation goes up as N^2, since the goal of the agents is to occupy all N^2 resources. On the other hand, if N is small the game is not terribly interesting. For the experiments reported in this chapter, we use a board size N of ten. We found that this was large enough to provide a challenging problem for the defender, while still keeping the computational overhead within reason.

Unlike a purely reactive agent that has no memory, states in an FSM provide a mechanism for "counting", i.e., given the same input the FSM can perform a certain action for some number of time steps and then perform a different action. Visualization of this game indicated that there might be merit in having the defender change direction after a certain number of steps, even if the input hadn't changed. A reasonable upper bound on the amount of counting that might be necessary appeared to be the size of the board, i.e., the FSM could easily recognize via counting that it had crossed the board from one end to another. Thus we allowed up to ten states in the FSMs,

since N was ten. Ten experiments were performed – during experiment i the number of states was held constant at i.

We ran some preliminary experiments to determine reasonable values for the population size P, the mutation rate p_m, the recombination rate p_r, and the number of generations to run the EA before termination. We found that a small population size was not sufficient, and settled on a population size of $P = 100$. As for the mutation rate, we followed the heuristic that the mutation rate should be inversely proportional to the number of alleles in the individual (Bäck and Schwefel, 1993). For one state FSMs p_m was set to 0.001, and p_m was decreased linearly with the number of states until it was 0.0001 for ten states. Thus, regardless of the number of states, on average the same number of individuals were mutated per generation. In general, we also found that the EA was relatively insensitive to changes in p_m, although higher values near 0.01 were detrimental to performance. In all cases we used a fixed mutation rate which did not adapt during the course of a run.[7]

Recombination performed quite differently. We varied the recombination rate p_r and the probability of swapping alleles P_0. The EA was quite sensitive to these values, and performance was optimized when p_r was 1.0 (all individuals are recombined) and P_0 was at its maximum of 0.5. In general, we found that recombination was quite useful for successful search. Interestingly, these results are counter to observations made by Fogel (personal communication). The differences in results may be due to differences in the mutation used (fixed rate versus adaptive rate), the form of recombination, or even the form of selection. Unfortunately, a thorough investigation of the cause of this difference is beyond the scope of the current chapter, and we instead intend to explore this issue in detail in future work.

As for the termination criterion, we found that there was a very low probability of continued improvement in the search after 2,500 generations, so the search was terminated at that point. Ten independent runs were performed for each experiment (fixed number of states) and the performance of the best individual seen thus far throughout the search was monitored. We wished to answer two experimental questions. First, is there an increase in performance when there is more than one state (i.e., does the defender actually need multiple states to play this game well)? Second, what is the optimal number of states?

	Number of States									
	1	2	3	4	5	6	7	8	9	10
Fitness	0.806	0.867	0.893	0.888	0.901	0.903	0.899	0.905	0.896	0.883

Table 3. The fitness of the best individual at the end of the experiment, averaged over ten runs per experiment. One state isn't enough for good performance.

[7] This differs from the evolutionary programming community, which typically uses an adaptive mutation rate.

The results are shown in Table 3 and represent the fitness of the best individual seen by the end of experiment (averaged over the ten runs per experiment). As can be seen, there is clearly a need for more than one state. Two states is not quite adequate either. However, it isn't clear that there is an optimal number of states. The optimal range of states appears to be roughly from five to eight, although anything from three to ten states performs roughly equivalently.[8]

Table 3 shows the results only at the end of the experiments, and it is important to see how performance improved over the course of the 2,500 generations. Figure 6 shows a graph of the fitness of the best individuals (these are referred to as "best-so-far" curves), as the population evolves. The vertical axis is the fitness, while the horizontal axis is the number of individuals that have been evaluated. Since the population size P is 100, 250,000 individuals will be evaluated during the course of a run. We use a log plot to emphasize the behavior of the EA during the early generations. To avoid clutter, we show only the graphs for one state and the even number of states. The remaining graphs for the odd number of states are similar. Again, each curve is averaged over ten runs.

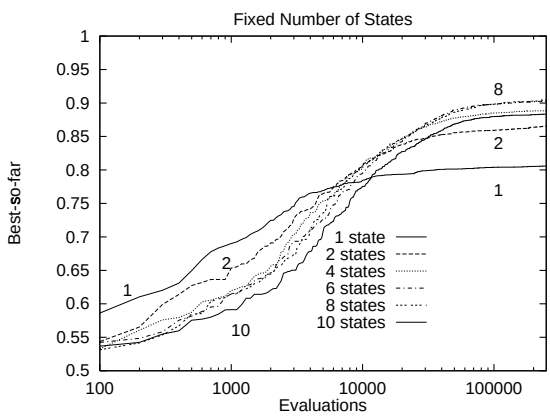

Fig. 6. "Best-so-far" curves for different numbers of states. We use a log plot to emphasize the behavior of the EA during the early generations. Having a small number of states enhances early performance but hurts later performance. Having a larger number of states degrades early performance but helps later.

Figure 6 shows quite clearly that increasing the number of states hurts performance at the beginning of the run. This is quite intuitively plausible, since the EA is searching a larger search space when the number of states increases. Thus, having fewer states helps search in the beginning. However,

[8] The increase in performance from one to two states and two to three states is significant ($p < 0.003$ and $p < 0.04$). The other differences are not significant.

as also can be seen, having only one or two states hinders search later in the experiment (around 20,000 evaluations). Having three or more states appears to be crucial for achieving the best performance at the end of the runs.

5.2 Adaptive number of states

With the fixed-state experiments providing a control study, we were prepared to evaluate the efficacy of having the EA adapt the number of states as it ran. As mentioned earlier, the exact implementation of state adaptation involves a sequence of design decisions. To aid us in making these decisions we relied upon two basic EA design principles: "exploitation" and "exploration". Exploitation refers to the continued use of knowledge already learned by the EA, while exploration refers to the continued search for new knowledge. In general increased exploitation results in decreased exploration (and vice versa), and successful EA implementations must strike a balance between these two design principles.

Let us address the questions mentioned earlier. When a state is deleted, should one actually erase the state from the table, or should it simply be made inaccessible? Our prior experience in similar areas has shown that it is often best to make the information inaccessible (De Jong, Spears, Gordon, 1993). Since information has been learned, keeping the information stored serves as a useful memory, which can be re-activated at a later time (if the state is added back to the FSM). Memory is a form of exploitation. Thus we decided to add a "tag" to each row of the FSM table. If the tag is 1 the state is accessible. If the tag is 0 the state is inaccessible, but is not destroyed. Thus, when a state is added, it is accomplished simply by turning on the tag. In our implementation these tags are subject to an independent mutation operation, that flips the tags (from 1 to 0 and vice versa) with probability 0.001. As mentioned earlier, STATE1 is always the initial state, so that is the only state that can not be made inaccessible.

Once a state has been made inaccessible, how should the remainder of the FSM (that points to that state) be "repaired"? Two solutions come to mind. Suppose state s is no longer accessible. If state s_1 points to s, change the pointer so that it points back to s_1. We refer to this form of repair as "self-reference", since the state will point back to itself. An alternative is to change the pointer to point to any state that is accessible, chosen uniformly randomly. In this situation the principle of exploration appears most germane, and we took the latter approach. We also modified mutation and recombination in a similar fashion. When mutation changes the next state of an allele, it can only choose accessible states (uniformly randomly).[9] Since parents may have different sets of accessible states, recombination may swap alleles in such a fashion that a next state that was accessible in one child FSM is now

[9] For these experiments we chose an intermediary mutation rate p_m of 0.0005.

inaccessible in the other FSM. In this situation a new next state is chosen uniformly randomly from the set of accessible states (in that FSM).

If a state has simply been made inaccessible, is it still subject to mutation and recombination? The principle of exploitation would indicate that one should not mutate and recombine inaccessible states. But the principle of exploration would indicate that one should. We tested both versions, and found very little difference in the results. For the results presented here, mutation and recombination work on all states, regardless of whether they are accessible or not.

We performed several experiments to judge the efficacy of this implementation of adaptation. We were interested in answering two questions. First, does the adaptive mechanism find the optimal range of accessible states? Second, how many states should be accessible initially? To address these questions we initialized each FSM individual with ten states. In the first experiment all ten states are initially accessible. In the second experiment the first five states were initially accessible. In the third experiment only the first state was initially accessible. The only mechanism for changing the number of accessible states is via the independent mutation operation mentioned above, which flips the accessibility tags. Although we have no "penalty" function per se (that would penalize the FSMs for having more accessible states), the mutation operator will provide a slight bias towards having five accessible states (in the same way that continuously flipping ten coins tends to result in roughly five heads and five tails).

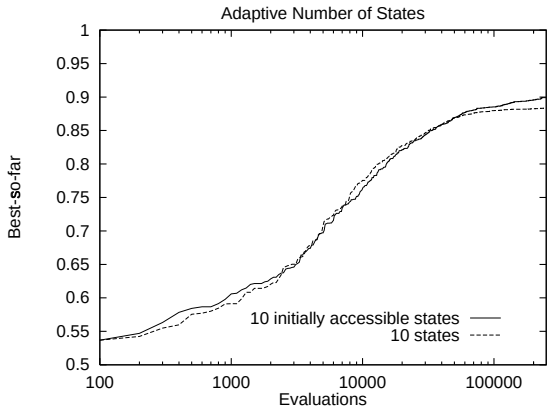

Fig. 7. "Best-so-far" curves for the EA with ten fixed states and the adaptive-state EA with ten initially accessible states. There is very little difference.

Figure 7 shows the best-so-far curve for the adaptive-state EA when it is initialized with ten accessible states. The results are again averaged over ten runs. On average the adaptive mechanism decreased the number of accessible

states in the FSMs, but only down to nine. Performance was reasonable, given that the final fitness was 0.899. Figure 7 also includes the best-so-far curve for the EA with ten fixed states. What is surprising is how similar the two curves are. There is little evidence that the adaptive mechanism is actually having much effect, from a performance or accessibility point of view.

When the EA was initialized with five accessible states, the adaptive mechanism increased the number of accessible states in the FSMs to roughly seven. Performance was reasonable, given that the final fitness was 0.898. The best-so-far curve is almost identical to that for the EA with five fixed states. Finally, when the EA was initialized with one accessible state, the adaptive mechanism increased the number of accessible states in the FSMs to roughly five. However, performance did not reflect that increase. The final fitness was 0.841, which is between the performance of the one and two fixed-state experiments. This provides evidence that the EA is having difficulty taking full advantage of the newly accessible states.

In summary, we have had mixed results with using the EA to adapt the number of accessible states in the FSM. Starting with only one accessible state resulted in inferior performance. Starting with five or ten accessible states provided much better performance, but there is little evidence that the adaptive mechanism is having much effect. The evidence does suggest that if one is going to use adaptation, it might be best to err on the side of initially having at least as many accessible states as the problem requires. We explore this issue further in the next section.

6 Behavior of the Evolved FSMs

As mentioned in the previous section, the EA appears to have some difficulty in adapting the number of accessible states successfully. We address this further in this section by examining the internal dynamics of the FSMs. We also examine the external behavior of the FSMs, which provides some important insights into how to evolve better FSMs for the Competition for Resources simulation.

6.1 Internal behavior

Although the number of accessible states is an interesting metric in itself, it obscures information about how often accessible states are actually used during the execution of the defender FSM. For example, during the fixed-state experiments, are all states used equally? Or does the FSM depend on some states more than others? Similar questions can be asked about the adaptive-state experiments. For example, in the experiment with one initially accessible state, the number of accessible states increased to roughly five. But a graph of performance indicates that all five states are probably not being used. When the EA is started with ten initially accessible states, performance

is good and the EA ends up using nine accessible states (on average). But are all of these accessible states used equally?

To address this issue we took the FSMs at the end of every run and reran each over 100,000 games. Each FSM has a set of $n \leq 10$ accessible states (in the case of the fixed-state experiments all states are always accessible). While the FSM was executing we counted the number of times that each of the n states was actually the next state of a transition. After execution the accessible states were sorted by their transition counts. For example, suppose that a FSM with four accessible states has made 100 transitions. It could be the case that 40 transitions were to one state, 30 to a second state, 20 to a third state, and 10 to the remaining state. We then translated this to the percentage of time that the FSM spent in each state. Finally, we computed the cumulative distribution of these percentages. Using the example above, 40% of time is spent in the most popular state, 70% of time is spent in the two most popular states, and 90% of time is spent in the three most popular states. Clearly 100% of time is spent in all accessible states.

# of States	State Usage									
	1	2	3	4	5	6	7	8	9	10
2	0.617	1.000	-	-	-	-	-	-	-	-
3	0.460	0.777	1.000	-	-	-	-	-	-	-
4	0.418	0.661	0.862	1.000	-	-	-	-	-	-
5	0.353	0.579	0.748	0.897	1.000	-	-	-	-	-
6	0.314	0.521	0.698	0.833	0.930	1.000	-	-	-	-
7	0.283	0.462	0.609	0.731	0.837	0.928	1.000	-	-	-
8	0.233	0.415	0.561	0.688	0.788	0.873	0.942	1.000	-	-
9	0.226	0.393	0.522	0.638	0.739	0.820	0.891	0.953	1.000	-
10	0.194	0.340	0.464	0.571	0.670	0.753	0.825	0.892	0.953	1.000

Table 4. Cumulative distributions of state usage for the fixed-state experiments. Note that although each experiment with i fixed states did not use all i states uniformly, it did make reasonable use of all i states.

For each experiment conducted in this chapter we performed ten runs. Thus for each experiment we averaged the cumulative distributions over those ten runs. The results for the fixed-state experiments are shown in Table 4. If the FSMs use all states uniformly the cumulative distribution will be linear. This is not the case, indicating that the FSMs are making more use of some states than others. For example, if all states were being used uniformly in the fixed-state experiment with ten states, each state would account for roughly 10% of the transitions. However, one can note that in fact the most popular state accounted for roughly 20% of the transitions. It is clear, though, that all states play a role in adding to the dynamics of the FSM.

Table 5 provides a similar table for the adaptive-state experiments. The data for the experiment with one initially accessible state (where the number

Initial # of States	State Usage									
	1	2	3	4	5	6	7	8	9	10
1	0.946	0.997	0.999	1.000	-	-	-	-	-	-
5	0.448	0.730	0.924	0.983	1.000	-	-	-	-	-
10	0.346	0.578	0.776	0.899	0.963	0.990	0.999	1.000	-	-

Table 5. Cumulative distributions of state usage for the adaptive-state experiments. In this case not all accessible states are well used.

of accessible states increased to roughly five) indicates quite clearly that the FSMs never actually make any real use of the newly accessible states. One can see from the table that one state accounts for about 95% of the transitions, while a second state accounts for the remainder. Similar results can be seen in the table when there are five and ten initially accessible states. The FSM appears to have difficulty making use of newly accessible states that have never been seen before. This confirms our prior conclusion that it appears best to err on the side of initially having more accessible states than the problem requires.

The results of our analyses suggest a few intriguing modifications to the EA. First, suppose FSMs are routinely monitored in this dynamic fashion. If states are not accessed, then increase the probability that a mutation will provide a transition to an under-utilized state. This increases exploration. Second, instead of making states inaccessible at random, it might be better to make poorly used states inaccessible, which would increase exploitation. We intend to explore these possibilities in the future.

Another possibility for modifying the EA is in response to the concern that addition and deletion of states are highly disruptive operations. We are currently investigating the application of "gentler" operators that could perform the same role. Simply deleting states (or turning them off) is too disruptive, due to the repair that must be performed afterwards. However, merging two similar states could remove a state in a fashion less deleterious to evolution. This process would be analogous to generalization. Similarly, as opposed to adding states (or turning them on), an alternative operator would clone an existing row of the tabular representation. Accessing this new state would not be deleterious and evolution could proceed to modify it slowly. This provides a process of specialization.

6.2 External behavior

Although a study of the internal behavior of the FSMs is useful, it can not replace the striking visual impression gained from simply watching the strategy execution in action. By watching the defender agent play numerous games against the adversary, we have detected fascinating patterns.

Let us begin by describing the desirable behaviors that evolve. We have observed two key successful types of strategies. The first consists of encapsu-

lating a relatively large rectangular region, and then filling it in. For example, the defender might capture all squares along the perimeter of a 4-square by 5-square rectangle, and then capture all the internal squares. The effectiveness of this strategy results from the fact that the adversary can not cross the perimeter, once captured, since it is now owned by the defender. The second successful strategy consists of spiraling through alternate rows of the board, then returning to capture squares in the unowned rows in between. The success of this second strategy is due to reasons similar to the success of the first strategy. In particular, blocking off rows of squares sets up borders that inhibit the adversary's movement.

Watching the agents play has also led to the observation of undesirable behaviors. The most striking and deleterious of these is the presence of unproductive cycling behaviors by the defender. Although it is guaranteed that FSMs must eventually cycle, many of our cycles were surprisingly short, e.g., of length less than N (the board size). For example, the defender might encapsulate a region, fill it in, and appear to be winning the game. But much to our dismay it will then get caught in an infinite cycle going back and forth between the same two or more squares. Meanwhile, the adversary continues capturing squares in its opportunistic random fashion and wins the game! Having seen this problem of cycles, our immediate concern was to remove these cycles from the FSM strategies, as described in section 8. But first, let us examine an example evolved FSM.

7 An Example FSM

To gain some understanding of how the FSM encodes the encapsulation behavior, we carefully examined the best FSM that evolved during the fixed-state experiments. This was an FSM with four states that won 95% of the games. Although the number of states was small, the large number of inputs (80) inherent in this task makes it very difficult to fully understand the actual FSM. In addition, we were unable to find any way to generalize the FSM, since it performed different actions for slightly different inputs. However, we did find it useful to consider the situation in which the defender has not yet observed the adversary. The resulting behavior is deterministic, but also holds in the more general situation where the adversary has been observed.

Figure 8 illustrates the portion of the FSM that encodes the encapsulation behavior. STATE1 is the initial state and the labels on the edges are input/action pairs. In this case, since the adversary hasn't been seen, the input indicates whether the defender is occupying any of the four neighbors of its current position. Figure 9 shows the sequence of moves that occur on the board.

At the beginning of the game the defender is at resource (1,1) and has been nowhere else; the input is null ($\emptyset$). The beginning of the game is indicated by time step '0' in square (1,1). The FSM indicates that the defender moves

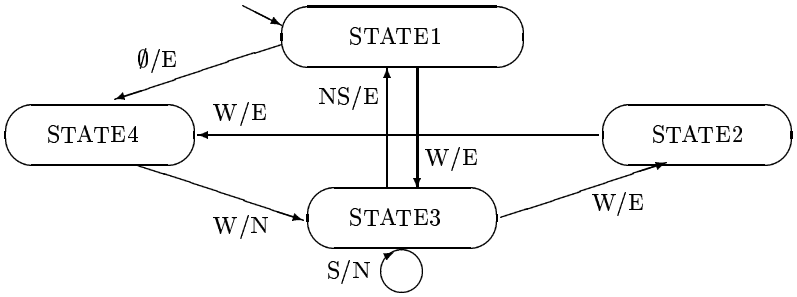

Fig. 8. A portion of an FSM agent plan that performed well. The edges are labeled as input/action, e.g., NS/E indicates that the agent observes that it is occupying the North and South resources (from its current position) and it moves East.

0	1				15				
	10	11	12	13	14				
	9				23	24			
	8				22				
	7				21				
	6				20				
	5				19				
	4				18				
	3				17				
	2				16				

Fig. 9. The sequence of moves made by the FSM in Figure 8, assuming that it has not encountered the opponent yet. Note how the agent encapsulates one half of the resources (which will eventually be filled in). The behavior of the actual agent under simulation looks similar.

east and transitions to STATE4 (step '1'). The defender notices that it is occupying the resource to the west of its current position, moves north, and transitions to STATE3 (step '2'). At this point the defender continues moving north, staying in STATE3. At step '10' the defender notices it can no longer move north, so it moves east and transitions to STATE1.

Now the defender continues moving east, transitioning from STATE3 to STATE2, and then STATE4. However, at step '14' the agent changes direction and moves north (transitioning to STATE3), although the input has not changed. This is a classic example of how FSMs can use states to count – the

defender is able to detect that it has crossed one half of the board's width. The defender continues moving north (staying in STATE3) until step '23'. At this point one half of the resources have been encapsulated.

If the defender has still not encountered the adversary it will move east and attempt to repeat the pattern. Usually this will not happen and the defender will be forced to respond to the adversary. The response will generally be an attempt to block the adversary, followed by a systematic filling in of the encapsulated region. One behavior that we have not shown is the deleterious cycling behavior. For the example FSM that we analyzed here, evolution removed most of the short cycles. However, this rarely occurs, and we needed a more systematic method for detecting and removing cycles.

8 Removing Cycles from the Defender's Strategy

As mentioned above, the EA itself can remove many cycles from FSMs, since their removal (through mutation or recombination) will increase performance. Unfortunately, we found that this simply takes too long to accomplish. Therefore we decided to supplement the EA with another method for cycle removal to speed up the process. We investigated two methods: *model checking* and *behavior checking*. Model checking (Clarke and Wing, 1996) examines an FSM before execution, while behavior checking (Kim et al., 1999) examines the FSM behavior during execution. We investigated model checking first, since it is known to be highly effective for enforcing behavioral constraints and will allow the FSM to be repaired before it is even executed.

Model checking consists of constructing a model of the system and then determining whether a property (constraint) holds strictly for the model, i.e., whether it is guaranteed that there will never be any violations of the property. Since the model is usually in the form of an FSM, model checking is immediately applicable to the defender's strategy. Traditional model checking consists of brute-force search through the entire set of all possible FSM transitions to verify whether the property holds absolutely. In this case we are clearly interested in having a property that "avoids cycles". Unfortunately, this property can not hold absolutely for FSMs, since cycles are an inherent property of an FSM (Fogel, 1999). However, as noted earlier, a large number of the cycles that the defender gets into are actually quite short, e.g., many consist of repeated visits to the same two or three squares. Furthermore, model checking can be computationally expensive; checking for long cycles is far more expensive than checking for short cycles. Therefore we focused instead on the property "avoid short cycles", where "short" is defined to be two- or three-cycles. We felt that avoiding short cycles would significantly improve performance while being reasonably cost-effective. In addition to model checking for cycles, we also implemented repair mechanisms that chose an alternative to the first action in a cycle in order to break the cycle.

To address the question of whether our cycle checking and repair algorithms improve behavior and, if so, how much, we ran experiments to determine their effectiveness. Unfortunately, the outcome of these experiments was that the addition of these algorithms results in very little performance improvement. The primary reason appears to be that the removal of two- or three-cycles usually results in the creation of slightly longer cycles, thus not really solving the cycle problem. We considered the computational expense of running model checking for these longer cycles to be unacceptable.

Given the failure with model checking, we next explored the use of behavior checking. Behavior checking gives no formal guarantee of behavior, but can be much quicker to perform than model checking. Behavior checking examines the dynamic run-time behavior of the agent, rather than the model (FSM) used by the agent. In particular, model checking explores numerous *possible* action sequences that the agent could take. Behavior checking, on the other hand, only tests the property for those action sequences *actually* taken at run-time, which is typically a very small subset of the possible action sequences. Run-time checking of system behavior is a very new topic in the verification community, but some of the results already appear promising (e.g., Gordon et al., 1999). Here, we present the first algorithm of which we are aware that does a run-time check for an FSM agent's cyclic behaviors.

Our behavior checking algorithm is executed while the agents play a game. For a sliding window of t time steps, the defender agent saves its current state and location on the board (the agent now consists of an FSM and auxiliary memory). Once cycle checking is turned on, the defender uses this auxiliary memory to make a cycle check before every move. If all four immediate neighbors are occupied, then the defender checks whether its current state and location are equal to any other in its window of memory. If yes, a cycle has been identified and a random alternative action is taken to the one recommended by the FSM (because the action recommended by the FSM would perpetuate the cycle). Of course, this alternative action could also create a cycle. But the behavior checking algorithm would immediately detect that cycle, after that move. We have found that a window size of $2 \times N = 20$ time steps, which will identify cycles up to length 20, works well. We are also exploring larger window sizes. In our experiments we ran cycle checking as soon as the window memory was filled, although in practice one might want to initiate it later in the game to save the expense – since cycles generally show up later in the game.

There is one more issue that we explored with behavior checking. If a cycle is broken with an alternative action, should that action be remembered, i.e., does the FSM get changed? This would be a form of Lamarckian (1984) learning. We tried this and unfortunately found that it hurt performance, so it is not used in the current version of the algorithm.

Let us consider the results of an experiment where behavior checking is added to the adaptive-state EAs. In this experiment, we added behav-

Initial Number of Accessible States in Adaptive-State EA	5	10
Fitness without behavior checking and repair	0.898	0.899
Fitness with behavior checking and repair	0.948	0.945

Table 6. The fitness of the best individual at the end of the experiment, averaged over ten runs per experiment, with and without behavior checking and repair.

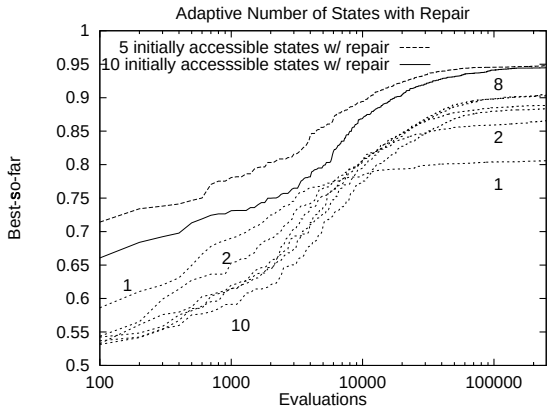

Fig. 10. "Best-so-far" curves for the adaptive-state EA with five and ten initially accessible states with cycle repair. The results for the fixed number of states are repeated for comparison. Note that cycle repair greatly enhances performance.

ior checking to the adaptive-state EA that begins with ten initially accessible states, and the one that begins with five initially accessible states, since these were two of our most successful EAs. Other than the addition of cycle checking and repair, each EA was run identically to the experiments in Section 5.2. Our hypothesis was that the addition of behavior checking and cycle repair would improve performance. This hypothesis is confirmed, as shown in Table 6 and Figure 10. Table 6 gives the fitness of the best individual seen by the end of the experiment, averaged over ten runs per experiment. Clearly, there is a large advantage to adding behavior checking and cycle repair.[10] Our best performing defender wins close to 97% of the games!

9 Summary and Future Work

To summarize, this chapter has empirically explored a variety of issues related to evolving FSMs in the context of the Competition for Resources problem. Our experiments yielded some rather surprising and potentially useful results. For example, the best method we found for evaluating FSM fitness was to perform an initial quick evaluation over a small number of games; then,

[10] The improvement using cycle checking and repair is significant, $p < 0.01$.

for those individuals identified as promising, a more extensive evaluation is performed over a larger number of games. We were unable to predict the optimal number of FSM states for winning this game (other than we expected it to be greater than one); thus it was interesting to discover that the best range is between five and eight states for a board of size ten. Deeper insights into the reasons for this result would help us make this representational decision for other board sizes and problems. Alternatively, one can use an EA with an adaptive number of states. We tried this and were surprised at how difficult it is to do this effectively. An in-depth analysis of dynamic state usage revealed that when starting with a minimum number of states, the EA appears to have difficulty making use of newly accessible states. This is less of a problem for an EA beginning with a maximum number of states – because rather than starting with new states, it is re-activating states that were formerly accessible and still retain valuable information. The hypothesis, then, is that it appears to be preferable to start with more states than needed for an adaptive-state EA. Finally, the chapter concluded with novel algorithms for cycle checking and repair. The algorithm for behavior checking proved to be far more effective than model checking. Because the behavior checking algorithm is restricted to only those cycles that actually appear while playing the game, considerably longer cycles can be detected and repaired with this method.

A possible direction for future research is to explore alternative strategy learning methods. Here we applied EAs because they are the most effective method we could find for learning FSMs from scratch. One option we could explore in the future would be to consider stochastic FSMs. By making the FSMs stochastic, it is possible that our cycle problem could be avoided or reduced. Unfortunately, the evolution of stochastic FSMs seems computationally intensive. On the other hand, there are alternative learning methods for stochastic FSMs that might be explored (Mars et al., 1996). We could also try more widely used reinforcement learning methods, such as *q-learning* (Watkins, 1989), although as indicated by the experiments in this paper, multiple internal states would be required for good performance.

Our main focus for the future will be to make the Competition for Resources simulation more realistic, and to continue our empirical investigations in the context of the newer versions of the game. For example, in the current game resources are all treated equally. In the spirit of game theory, we would like to consider resources having different numeric values, and perhaps have the value of a resource differ for each of the agents. Another possibility is to allow one agent to (with some small probability) "steal" a resource owned by the other agent. In other words, the rule that once a resource is owned by an agent, this ownership is permanent, would be altered. Another possibility is to include multiple agents and co-evolution. What is most interesting about this game is how easy it is to be changed to represent a wide variety of problems. For example, with minor modifications we have extended the game to

represent the epidemiology of virus versus anti-virus spread. In the virus version of the game, each square represents an agent with the virus, anti-virus, or neither. At each time step, an agent having the virus or anti-virus can spread it to one of its neighbors. What one sees on the board when watching this version of the game looks like a "spreading activation". Further pursuit of the virus version both in simulation and in a corresponding mathematical model are currently in progress.

References

1. Bäck, T. and Schwefel, H.-P. (1993) An overview of evolutionary algorithms for parameter optimization. Evolutionary Computation, **1 1**:1–23.
2. Carmel, D. and Markovitch, S. (1996) Learning models of intelligent agents. Proceedings of the Thirteenth National Conference on Artificial Intelligence.
3. Clarke, E. and Wing, J. (1996) Formal methods: State of the art and future directions. ACM Computing Surveys, **28 4**:626–643.
4. Dean, T., and Wellman, M. (1991) Planning and Control. Morgan Kaufmann, San Mateo.
5. De Jong, K., Spears, W. and Gordon, D. (1993) Using genetic algorithms for concept learning. Machine Learning Journal, **13 2/3**:161–188.
6. Denning, D. (1999) Information Warfare and Security. Addison-Wesley, New York.
7. Fogel, L. (1999) Intelligence Through Simulated Evolution: Forty Years of Evolutionary Programming. Wiley Series on Intelligent Systems, New York.
8. Fogel, D. (1995) Evolutionary Computation. IEEE Press, New York.
9. Fogel, L., Owens, A., and Walsh, M. (1966) Artificial Intelligence Through Simulated Evolution. John Wiley and Sons, Inc., New York.
10. Gordon, D., Spears, W., Sokolsky, O., and Lee, I. (1999) Distributed spatial control, global monitoring and steering of mobile physical agents. Proceedings of the IEEE International Conference on Information, Intelligence, and Systems.
11. Grefenstette, J. and Fitzpatrick, J. (1985) Genetic search with approximate function evaluations. Proceedings of the International Conference on Genetic Algorithms.
12. Holland, J. (1975) Adaptation in Natural and Artificial Systems. University of Michigan Press, Ann Arbor.
13. Hopcroft, J. and Ullman, J. (1979) Introduction to Automata Theory, Languages, and Computation. Addison-Wesley, Menlo Park.
14. Jefferson, D., Collins, R., Cooper, C., Dyer, M., Flowers, M., Korf, R., Taylor, C. and Wang, A. (1991) Evolution as a theme in artificial life: The Genesys/Tracker system. Proceedings of Artificial Life II.
15. Kim, M., Viswanathan, M., Ben-Abdallah, H., Kannan, S., Lee, I., and Sokolsky, O. (1999) Formally specified monitoring of temporal properties. Proceedings of the Euromicro Conference on Real-Time Systems.
16. Lamarck, J.B. (1984) Philosophie Zoologique. English Translation, University of Chicago.

17. Mars, P., Chen, J., and Nambiar, R. (1996) Learning Algorithms: Theory and Applications in Signal Processing, Control and Communications. CRC Press, New York.
18. Rich, E. and Knight, K. (1981) Artificial Intelligence. McGraw-Hill, New York.
19. Spears, W. and De Jong, K. (1991) On the virtues of parameterized uniform crossover. Proceedings of the International Conference on Genetic Algorithms.
20. Watkins, C. (1989) Learning from delayed rewards. Ph.D. thesis, University of Cambridge, England.