Genetic Algorithms for Approximating Solutions to POMDPs

Chris Wells
Computer Science Division
University of California, Berkeley
387 Soda Hall #1776
Berkeley, CA 94720-1776

Christopher Lusena and **Judy Goldsmith**
Computer Science Department
University of Kentucky
773 Anderson Hall
Lexington, KY 40506-0046

Abstract

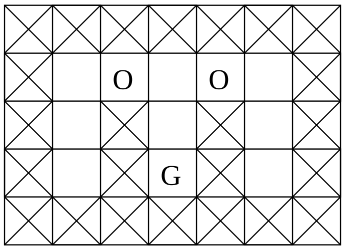
We use genetic algorithms (GAs) to find good finite horizon policies for POMDPs, where the search is limited to policies with a fixed finite amount of policy memory. Initial results were presented in (Lusena *et al.* 1999) with one GA. In this paper, different cross-over and mutation rates are compared. Initializing the population of the genetic algorithm is done using smaller genetic algorithms. The selection and termination criteria are altered and tested. Reordering and uniform cross-over are implemented and compared against the original GA. And finally, a steady-state GA is implemented and used as a control for tests with a steady-state GA whose operator rates are dynamic.

Introduction to POMDPs

A Markov Decision Process is a discrete stochastic representation of a particular environment or problem that is convenient for describing a computerized agent. An MDP is a tuple $M = (\mathcal{S}, \mathcal{A}, t, r)$: $\mathcal{S}$ is the state space, which contains every possible state the agent can be in; $\mathcal{A}$ is the action space, from which the agent must choose an action at each decision epoch; t maps $\mathcal{S} \times \mathcal{A} \times \mathcal{S}$ to [0,1], meaning that while in state s, if action a is taken, then s' will be the new state with probability t(s,a,s'); and r, the reward table, maps $\mathcal{S} \times \mathcal{A}$ to the set of rational numbers, indicating that taking action a while in state s will yield the specified numeric reward. An agent chooses an action based on its current state according to some policy.

When the agent is unsure of its position in the state space, however, the problem of finding an optimal policy becomes more difficult. These problems are represented by Partially Observable Markov Decision Processes. Similar to MDPs', POMDPs' tuples also include $\mathcal{O}$ and o: $\mathcal{O}$, the observation space, is the set of all possible observations; and o is a mapping from $\mathcal{S}$ to $\mathcal{O}$, so that when the agent is in state s, it will be given observation o. Because in this POMDP model a particular state will only ever yield the same observation, it is called deterministically observable; this is the only model considered in this paper.

Figure 1: McCallum's Maze



Policies for POMDPs which determine actions based solely on the current observation are known as stationary policies. Finding optimal stationary policies was shown to be NP-hard (Papadimitriou & Tsitsiklis 1987). In many cases, however, a stationary policy is not an optimal policy. (McCallum 1993) provided a simple example of such POMDPs (see Figure 1). This POMDP is a very simple model of an animal in a small environment. The animal–or agent–at any given moment, is in a single place or square. It cannot occupy any square with an "X" in it. It desires to reach the goal state, marked by "G", but it can only see the squares immediately above, below, to the left or to the right of its current location. Based on its surroundings, the agent therefore makes one of seven possible observations. In the squares marked by "O," however, the agent makes the same observation, but the best actions are different. So there does not exist a stationary, memoryless policy which will take the agent to the goal state from any square in the grid world.

A policy which will take the agent from any square to the goal state must employ some form of memory. In some POMDPs, a full history is required to yield the optimal policy for a problem. A full history policy, however, will grow exponentially with the number of decision epochs considered for the problem. In many problems, however, a small finite amount of memory is sufficient for an optimal policy. The concept of finite memory was first discussed in (Sondik 1971). The policies we consider here are limited free finite memory policies, meaning that a policy is given a predetermined number, k, of memory states among which it moves at each decision epoch. So formally, we are looking for an optimal policy $\pi_f{}^* : \mathcal{O} \times \mathcal{M} \to \mathcal{A} \times \mathcal{M}$ for a POMDP $M = (\mathcal{S}, \mathcal{A}, \mathcal{O}, t, r, o)$. Because the number of memory states is predetermined in our model, the POMDP and memory set can be reduced to a POMDP whose state space is $\mathcal{S}' = \mathcal{S} \times \mathcal{M}$, and therefore solving the POMDP is still only NP-hard (Lusena *et al.* 1999).

Related Work

Much recent work has been done on finding optimal policies for POMDPs. (Hansen 1998) proposed using finite-state controllers to find ε-optimal polices. Hansen's algorithm is a form of policy iteration. His finite-state controller represents actions as nodes in the controller, and observations as transitions among the controller's states. The policy is improved by modifying the finite state controller.

(Meuleau *et al.* 1999) also solve POMDPs by searching the policy space. Like Hansen, they use finite-state controllers to represent policies. Like (Lusena *et al.* 1999), they also restrict the amount of memory used, as we do. They present a branch and bound algorithm which, because of the constraints they place on the controller, can find optimal policies for POMDPs more quickly than other methods. They also describe a local search method for approximating an optimal solution.

(Lusena *et al.* 1999) also use approximation techniques to find good policies for POMDPs. They present three approximation algorithms: local search, simulated annealing, and a primitive genetic algorithm. This paper expands on the genetic algorithm presented in (Lusena *et al.* 1999).

Introduction to Genetic Algorithms

Genetic algorithms were first proposed by (Holland 1975) as a technique for approximating solutions to intractable problems. We apply them here to finding optimal or near-optimal policies for POMDPs with limited free finite memory. Before our initial genetic algorithm is described, however, we need to review the theories of schemata and encoding in genetic algorithms.

Schema Theory

In this section we consider sets of genes in the individuals' chromosomes. "Sets" refers to all possible combinations of all the genes in an individual. When creating the next generation of individuals, a given set of genes may be created by combining sets from individuals in the previous generation, by modifying an existing set, or by survival of an existing set from the old generation to the new one. These sets are referred to in (Holland 1975) as schemata. More formally, when a chromosome has length n, and each symbol (element) in a chromosome is taken from alphabet Σ, a schema is a pattern of length n whose symbols are taken from $\Sigma \cup \{\#\}$, where '#' matches any character in Σ. Thus, if our policy-chromosome (whose symbols are the numeric labels of actions and new memory-states) is "2,3,1,0,2", it matches schemata "$2, \#, 1, \#, \#$", "$\#, 3, 1, \#, 2$", and "$\#, 3, \#, 0, \#$", as well as many other schemata.

Figure 2: Cardinality in aloha.10 per generation

Generation	Mean Cardinality	Standard Deviation
1	18.6222	1.07624
2	15.1111	0.970059
3	12.7889	1.01626
4	10.6556	1.20407
5	9.16667	0.950417
6	8.13333	1.00178
7	6.81111	1.16893
8	6.20000	1.31552
9	5.55556	1.17822
10	4.83333	1.26713

Given a generic POMDP, we cannot be sure how certain loci in a chromosome interact in the solution space without a lot of analysis and computation. For instance, in McCallum's Maze, if the observation of the square just above the goal state is mapped to an "up" action in a policy, the rest of the policy doesn't matter; no matter what actions we assign to any other observation the policy's value will never improve. Other genes are similarly interdependent, though not necessarily in this precise relationship. These genes, however, may not be located closely together on the chromosome, even if they are highly interdependent.

Encoding in Genetic Algorithms

In order to use a genetic algorithm to evolve an optimal or near-optimal policy, we must first determine an encoding for a policy. Although research in genetic algorithms has shown that binary encodings (where each gene is a single bit, and bits combine in various ways in the translation of a solution) offer a higher number of schemata for combination, research has been done with encodings that use symbols other than bits as genes; genes can have a cardinality higher than two.

(Goldberg 1990) explains in his theory of virtual alphabets that through the process of selection, genes with high cardinality converge to a few symbols within a few generations, so that each gene effectively has a low cardinality. We therefore encode a policy as a chromosome with size $|O| \times |\mathcal{M}|$. Each gene in the chromosome has $|A| \times |\mathcal{M}|$ possible values, a high cardinality relative to a binary encoding. We monitored the cardinality of each gene in 10 runs of 10 generations each on aloha.10 (a POMDP from Cassandra's database (Cassandra 1997 99)) with three memory states, so that the number of genes was 3 observations $\times$ 3 memory states = 9 genes and each gene had 9 actions $\times$ 3 memory states = 27 possible symbols.

Figure 2 summarizes our findings. By the fifth generation, the number of symbols per gene was reduced to one third of the possible cardinality for each gene. This supports Goldberg's theory (Goldberg 1990). With a low cardinality per gene, more schemata can be preserved and combined than if the cardinality were high.

Initial Implementation

The initial genetic algorithm used in this paper is described in (Lusena *et al.* 1999). It is described again here for completeness. We are trying to evolve good policies; a chromosome corresponds to a policy. A chromosome represents a $|O| \times |\mathcal{M}|$ table. Each cell in this table contains a reference to a single action and a single memory state, indicating that when the agent makes the current observation and is in the current memory state, that it should take the indicated action and move to the new memory state.

To enable this genetic algorithm to be competitive with local search, simulated annealing, or other approximation algorithms– each of which take time approximately polynomial in the size of the state space– we make the population size logarithmic in terms of the size of the search space. The size of the search space is simply the number of different policies that are possible, which is $(|A| \times |\mathcal{M}|)^{|\mathcal{O}| \times |\mathcal{M}|}$. Because we expect more memory to help us converge to the optimal policy more often (based on the experiments reported in (Lusena *et al.* 1999)), we do not calculate the population size linearly with regard to the number of memory states. Rather, the population size grows with $|O| \times \log(|A| \times |\mathcal{M}|)$ In early tests, however, this population size was not large enough, so it was increased by a factor of the square of its log. The population size is therefore $\log^2(|O| \times \log(|A| \times |\mathcal{M}|)) \times |\mathcal{O}| \times \log(|\mathcal{A}| \times |\mathcal{M}|)$. Each chromosome is initialized randomly.

In genetic algorithms, a population is taken through several generations until a terminating condition is achieved. In this implementation, we terminate the genetic algorithm in either of two circumstances. First, if the best policy (which the algorithm stores after each generation) occurs more than a particular number of times (in this case, 10 is arbitrarily chosen), then improvement is not taking place and the genetic algorithm terminates. If the standard deviation of the values of the policies in the population drops below a specified threshold (here, .0001, which is sufficiently small for all examples considered in this paper), and the best policy value has been encountered in the last five generations, then improvement is not taking place, and the genetic algorithm halts.

The terminating conditions are checked during selection, in which each policy is evaluated using a user-specified finite horizon. If termination is not indicated, each policy's value is normalized by subtracting the minimum allowed value, which is two standard deviations below the mean of the values. Any policy whose value is lower than this bound has its value set to 0.0. Any policy whose value is greater than the mean plus two standard deviations has its value set to this upper bound before it is normalized, in order to help preserve diversity. Policies are then selected randomly, but weighted proportionately to the square root of their adjusted values. The square root was used in order to keep diversity high during the earlier generations.

Once a new generation has been selected, it undergoes two genetic operations: cross-over and mutation. In each generation, cross-over is performed a number of times equal to the cross-over rate times the population size. In cross-over, two phenotypes (policies) from the new mating population are randomly selected. A pivot observation and a pivot memory state are randomly selected about which cross-over will take place. All elements (action/new memory state pairs) of the policy which occur before the pivot observation and the pivot memory state are swapped between the two phenotypes. All elements occuring at or after these pivots are also swapped. We choose two pivots in the hope that good actions from two parents will be combined in some cases, and good new memory states will be combined in other cases. This form of cross-over is reminiscent of two-point cross-over (Syswerda 1989), in that two pivots are chosen. The second operation, mutation, is performed a number of times equal to the mutation rate times the population size in each generation. Mutation is a unary operation, in which a random member of the mating population (which has already been altered by cross-over) is chosen. A random element in this phenotype is chosen, and its action/new memory-state pair is changed to a randomly selected action/new memory-state pair.

The genetic algorithm determines whether it should terminate during selection. If improvement has not taken place (a new best policy has not been found) for the last ten generations, or if the standard deviation of the phenotypes' values drops below a threshold of .0001 (which is low enough for the samples used in this work) so that there is sufficiently little diversity in the population, then the algorithm halts.

Of course, the description above is incomplete. The cross-over and mutation rates have not yet been determined. We tried several rates on McCallum's Maze and aloha.10. The results are in Figures 3 - 8. (The results for 1 memory state were omitted, due to space constraints.) We measure the "quality" of a genetic algorithm by the number of times it finds the optimal policy for McCallum's Maze, or by the mean policy value it finds for aloha.10. We evaluate the time it takes for a genetic algorithm to run by the number of evaluations (in this case, backward iterations) the algorithm performs before termination. As the table shows, higher cross-over rates and higher mutation rates (up to a point) are more likely to find an optimal policy.

Modifications to Initial Algorithm

Better Initial Policies

The first modification we made to the original genetic algorithm was to initialize the population with policies which, while not optimal, were better than random. We

Figure 3: Performance of Initial Genetic Algorithm on McCallum's Maze with 2 memory states

Times out of 100 optimal policy was found

	Mutation					
Xover	.045	.055	.065	.075	.085	.095
0.4	29	23	25	25	28	39
0.5	34	30	31	28	33	38
0.6	34	48	44	47	37	48
0.7	39	45	37	47	52	45
0.8	35	43	41	47	50	42
0.9	39	44	53	45	56	49
1.0	42	43	49	55	44	55

Average number of backward iterations

	Mutation					
Xover	.045	.055	.065	.075	.085	.095
0.4	1786	1905	2032	2064	2134	2178
0.5	2263	2329	2328	2465	2644	2579
0.6	2614	2804	2895	2990	3021	2937
0.7	2830	2985	2963	3211	3244	3292
0.8	3149	3133	3227	3443	3421	3432
0.9	3332	3484	3446	3500	3502	3533
1.0	3402	3367	3616	3485	3652	3714

Figure 4: Performance of Initial Genetic Algorithm on McCallum's Maze with 3 memory states

Times out of 100 optimal policy was found

	Mutation					
Xover	.045	.055	.065	.075	.085	.095
0.4	25	27	19	23	18	27
0.5	27	27	36	25	30	31
0.6	35	28	29	36	28	34
0.7	28	29	31	41	35	35
0.8	32	41	39	39	31	45
0.9	47	46	49	44	55	42
1.0	51	43	47	49	44	36

Average number of backward iterations

	Mutation					
Xover	.045	.055	.065	.075	.085	.095
0.4	3355	3484	3641	3734	3567	3768
0.5	3942	4053	3918	4170	4179	4294
0.6	4534	4620	4368	4614	5006	4949
0.7	5185	4849	5109	4963	5326	5211
0.8	5126	5256	5394	5175	5250	5659
0.9	5356	5586	5628	5657	6027	5631
1.0	5332	5859	5642	5700	5489	5565

Figure 5: Performance of Initial Genetic Algorithm on McCallum's Maze with 4 memory states

Times out of 100 optimal policy was found

	Mutation					
Xover	.045	.055	.065	.075	.085	.095
0.4	15	16	15	22	25	22
0.5	23	21	25	28	26	25
0.6	26	21	30	35	35	34
0.7	32	30	30	41	34	37
0.8	32	30	32	39	40	34
0.9	30	38	42	40	36	32
1.0	42	39	48	39	41	43

Average number of backward iterations

	Mutation					
Xover	.045	.055	.065	.075	.085	.095
0.4	4259	4646	4791	4807	5045	5024
0.5	5336	5231	5699	5793	5321	5580
0.6	5838	6090	6129	6038	6119	6216
0.7	6502	6487	6646	6588	6322	6763
0.8	7020	6805	7149	7251	7132	7360
0.9	7168	7206	7499	7339	7368	7341
1.0	7165	7534	7138	7294	7762	7318

Figure 6: Performance of Initial Genetic Algorithm on aloha.10 with 2 memory states

Average value of final policy

	Mutation					
Xover	.045	.055	.065	.075	.085	.095
0.4	95.85	96.84	96.82	96.82	97.05	97.42
0.5	95.76	97.20	96.70	96.55	96.68	97.48
0.6	95.84	96.99	97.19	96.91	97.23	97.57
0.7	96.27	96.94	96.94	96.63	97.59	97.49
0.8	96.09	96.50	96.80	97.02	97.20	97.21
0.9	95.92	96.53	97.19	96.98	97.06	96.93
1.0	95.98	96.85	97.13	96.81	97.37	97.34

Average number of backward iterations

	Mutation					
Xover	.045	.055	.065	.075	.085	.095
0.4	346	506	497	507	622	611
0.5	371	559	522	494	665	654
0.6	428	594	620	584	720	763
0.7	484	643	651	620	802	808
0.8	532	705	718	727	841	812
0.9	555	723	730	740	863	846
1.0	566	766	778	728	886	880

Figure 7: Performance of Initial Genetic Algorithm on aloha.10 with 3 memory states

Average value of final policy

	Mutation					
Xover	.045	.055	.065	.075	.085	.095
0.4	97.16	97.52	97.72	98.02	98.22	97.94
0.5	97.02	97.20	97.65	97.90	98.19	98.07
0.6	97.26	97.55	97.53	97.64	98.34	98.31
0.7	97.09	97.51	97.40	97.77	98.36	98.48
0.8	97.08	97.79	97.97	97.80	98.22	97.88
0.9	97.41	97.28	97.77	97.85	98.50	98.34
1.0	97.84	97.50	97.79	97.80	98.28	98.44

Average number of backward iterations

	Mutation					
Xover	.045	.055	.065	.075	.085	.095
0.4	739	746	966	948	1156	1107
0.5	765	782	1052	1050	1185	1238
0.6	953	960	1168	1177	1424	1437
0.7	1020	1122	1213	1319	1521	1644
0.8	1136	1176	1395	1349	1723	1623
0.9	1194	1169	1431	1495	1726	1737
1.0	1264	1254	1519	1479	1799	1853

Figure 8: Performance of Initial Genetic Algorithm on aloha.10 with 4 memory states

Average value of final policy

	Mutation					
Xover	.045	.055	.065	.075	.085	.095
0.4	97.55	97.71	97.76	98.51	98.05	98.21
0.5	97.36	97.68	97.69	98.04	98.50	98.53
0.6	97.47	98.09	97.86	97.78	98.32	98.37
0.7	97.26	97.89	97.67	98.28	98.57	98.24
0.8	97.42	98.02	97.81	98.20	98.51	98.20
0.9	97.72	97.96	98.01	98.24	98.18	98.56
1.0	97.50	98.12	97.99	98.35	98.60	98.62

Average number of backward iterations

	Mutation					
Xover	.045	.055	.065	.075	.085	.095
0.4	1097	1311	1249	1543	1767	1773
0.5	1151	1385	1371	1650	2039	2043
0.6	1339	1737	1612	1972	2128	2124
0.7	1500	1765	1814	2207	2502	2421
0.8	1688	2064	2002	2324	2775	2553
0.9	1785	2127	2135	2423	2729	2844
1.0	1814	2297	2279	2505	2819	2952

Figure 9: Performance of first GA with initialized population on McCallum's Maze with 3 memory states

Times out of 100 optimal policy was found

	Mutation			
Xover	.065	.075	.085	.095
0.7	1	1	6	6
0.8	3	5	8	10
0.9	4	5	9	8
1.0	1	3	10	10

Average number of backward iterations

	Mutation			
Xover	.065	.075	.085	.095
0.7	1498	1512	2092	2062
0.8	1513	1523	2067	2083
0.9	1669	1646	2247	2235
1.0	1683	1672	2244	2252

generated each of the initial polices with a genetic algorithm which was initialized randomly. Our first setting of the final genetic algorithm's population size was the square root of its population size in the original GA. Each initializing genetic algorithm also had this population size, so that if one viewed the complete two-phase algorithm, in Phase 1 there was a complete population that was evenly divided into several parts, and in Phase 2 the results of Phase 1 were refined. The results of this two-phase GA are in Figure 9. Unfortunately, while the total population remained effectively the same for this implementation, the relative smallness of the individual GAs' population sizes led to early convergence. Hence, this implementation ran much faster, but did not find the optimal policy as often, and did not generate as good an average policy as the original GA.

We then ran the full genetic algorithm on the original size population, so that it would have the same chance for good convergence as the original GA. We also reduced the sizes of the Phase 1 GA's from the square root of the original population size to the fourth root of the population size, so that they did only a little improvement on the randomly-generated policies. Figure 10 shows that, while the time spent evaluating increased dramatically, so did the number of times the optimal policy was found.

This second implementation is promising. If a more efficient method of quick improvement was used in Phase 1 (local search for instance), perhaps the improvement of the quality of the final GA would remain, while the drastic increase in evaluation time would not occur.

Alternate Selection Routines

In the original genetic algorithm, the square root of the policies' values was chosen to help maintain diversity. We then tested the argument by modifying the original GA so that we used the policies' values instead of their square roots during selection, while still eliminat-

Figure 10: Performance of second GA with initialized population on McCallum's Maze with 3 memory states

Times out of 100 optimal policy was found

	Mutation			
Xover	.065	.075	.085	.095
0.7	59	56	64	65
0.8	62	57	69	60
0.9	68	51	68	72
1.0	64	66	60	62

Average number of backward iterations

	Mutation			
Xover	.065	.075	.085	.095
0.7	7699	7537	7695	7707
0.8	7825	7530	8192	7984
0.9	8132	8106	7804	8071
1.0	8073	8173	8104	8247

ing outlier effects. The results appear in Figure 11. No appreciable difference either in the quality of the genetic algorithm or its running time are evident. Therefore, using the square root of the policies' values is of no advantage, and slightly increases the program's execution time.

Another method of selection widely used is that of ranked selection. It has the advantage that no single individual can disrupt the diversity of the population due to a very high fitness value. While our original algorithm disallowed this occurence, we tested whether ranked selection would improved upon our own selection routine. In each generation, after the population was evaluated, each individual was given a whole number as its fitness value. This number was the reverse of the individual's fitness rank in the population. Thus, the worst individual (the policy with the lowest value) was assigned a fitness rank of one. The second worst was given a rank of two, and so on. The best individual got a fitness rank equal to the size of the population. In this way, the best individuals were allowed to have many times the fitness of the worst individuals, but had to have very nearly the same fitnesses as one another.

While this new method did slightly worse than the original for McCallum's Maze with one memory state, it did significantly better–both in finding the optimal policy and in the number of policy evaluations used–for McCallum's Maze with two, three, or four memory states. These results are summarized in Figures 12 and 13.

Alternate Termination Criteria

The original algorithm's termination criteria, while sufficient for an initial genetic algorithm, was not terribly robust. We tested whether adjusting the number of times a best policy occurred according to the population size would affect the outcome of the genetic algorithm. So, instead of waiting until the same best policy occurred ten times, the new algorithm waited until it

Figure 11: Performance of GA with no square root in selection on McCallum's Maze with 3 memory states

Times out of 100 optimal policy was found

	Mutation		
Xover	.035	.045	.055
0.3	13	19	7
0.4	25	21	21
0.5	35	17	27

Average number of backward iterations

	Mutation		
Xover	.035	.045	.055
0.3	1985	2405	2452
0.4	2917	3032	3120
0.5	3656	3739	3750

Figure 12: Performance of GA with ranked selection on McCallum's Maze with 1 memory state

Times out of 100 optimal policy was found

	Mutation		
Xover	.035	.045	.055
0.3	43	57	57
0.4	60	66	64
0.5	53	67	73

Average number of backward iterations

	Mutation		
Xover	.035	.045	.055
0.3	154	167	173
0.4	184	206	208
0.5	202	231	229

Figure 13: Performance of GA with ranked selection on McCallum's Maze with 4 memory states

Times out of 100 optimal policy was found

	Mutation		
Xover	.035	.045	.055
0.3	14	24	36
0.4	16	24	37
0.5	25	33	39

Average number of backward iterations

	Mutation		
Xover	.035	.045	.055
0.3	1552	1794	2022
0.4	2213	2586	2875
0.5	2925	3234	3481

occured one fifth of the population size times. The results appear in Figure 14. While the new GA took significantly more evaluations for memory sizes of two, three, and four, it also found the optimal policy far more often.

A more common form of termination than the two described so far is that of detecting gene convergence. A gene is said to converge if ninety-five percent of the individuals in the population have the same value for the gene. The population is said to have converged when all genes have converged. We were unable to complete the test runs for this implementation because of severe epistasis.

Epistasis in a genetic algorithm means that certain genes are interdependent during evaluation. An example of a problem that is epistasis-free would be the counting-bits genetic algorithm. In this GA, chromosomes are 30-bit strings. The value of an individual chromosome is the number of bits in it which are 1s. Each bit is initialized randomly, and cross-over and mutation are performed at each generation. So each gene (bit) is independent of all other genes, since a gene g will be counted as a 1 if and only if gene g is a 1-bit.

Now consider a slight variation in this simple example. Evaluation still counts the bits with values of 1, but only those bits which are 1 and have a 1 immediately to their left or are the left-most bit are counted as 1s. So 11010 would have value two, and 01111 would have value three. This example has epistasis, because in evaluation a gene is partially dependent on the gene to its left in the chromosome. A more precise description of epistasis can be found in (Davidor 1991).

Another discussion of epistasis is done by (Naudts, Suys, & Verschoren 1997), in which they describe bit decidability as a measure of epistasis, or the "hardness" of a genetic algorithm to be solved. This work seems to apply directly to our termination-by-convergence algorithm. Even with one memory state, McCallum's Maze (Figure 1) can become undecidable to this genetic algorithm if all deterministic observations have already converged to their optimal values, but approximately one-half the population goes left in the two squares marked "O", and the other half goes right in these squares. Whether an individual goes left or right does not matter to its value in this case, since either way results in an optimal policy value. Since the two decisions look the same to the evaluation routine, the GA has no way of eliminating one group or the other (except through blind chance). When we introduce extra memory states, the problem only becomes worse. Since for many observations the current memory state does not matter, it is impossible for the GA to distinguish some memory states from one another.

We terminated our test run for this implementation after McCallum's Maze with two memory states–at a cross-over rate of 0.5 and a mutation rate of 0.085–took on average 648,785.8 backward iterations to terminate.

Figure 14: Performance of termination related to population size on McCallum's Maze with 3 memory states

Times out of 100 optimal policy was found

	Mutation			
Xover	.065	.075	.085	.095
0.7	75	71	67	73
0.8	69	76	82	80
0.9	76	78	82	79
1.0	76	75	86	80

Average number of backward iterations

	Mutation			
Xover	.065	.075	.085	.095
0.7	10579	10459	11067	10638
0.8	11305	11231	12010	11031
0.9	11023	10803	11900	11582
1.0	11503	11453	11619	11879

Inversion and Reordering

Because some genes in our phenotypes (policies) are dependent on other genes whose loci are far away on the chromosome, we thought reordering the loci during each generation of a genetic algorithm would be beneficial. This method, called inversion, was first proposed by (Holland 1975). It is intended to move closely-related genes closer together on a chromosome, so that during cross-over, they are more likely to stay together than to be swapped between the two parents.

Before applying inversion to our initial genetic algorithm, we to modified the chromosomes so that they were linear and the each cells in the chromosomes so that they stored not only the gene's value, but its location (observation by old memory-state) as well. The chromosomes were initialized so that cells for particular loci were placed randomly along the chromosome. Then we defined inversion to be an operator performed before cross-over in each generation. It was applied inversion-rate $\times$ the population size times at each generation. In our inversion operator, a random cell was chosen, and the two cells on either side of it were swapped. Cross-over had to be modified since the chromosomes were now linear. We used a one-point cross-over where the elements to the left of a randomly selected pivot were swapped between two parents to produce the two children.

All runs in this section were performed with a mutation rate of 0.055. The results for our runs on McCallum's Maze are in Figure 15. With an inversion rate of zero, the algorithm is the same as the original algorithm except that one-point cross-over is performed instead of our modified two-point cross-over. High inversion rates appear to lead to better results, but they take far more policy evaluations to terminate. This may be explained by the fact that when we attempt to reorder the genes, we expand the search space of the GA; not only is it searching for a good policy, it is searching for a good ordering of the loci of genes on a chromosome.

Figure 15: Performance of GA with inversion on McCallum's Maze with 3 memory states

Times out of 100 optimal policy was found

	Inversion					
Xover	0.0	0.2	0.4	0.6	0.8	1.0
0.7	6	17	29	29	37	31
0.8	8	11	19	37	36	38
0.9	4	16	24	29	38	26
1.0	3	18	26	39	30	27

Average number of backward iterations

	Inversion					
Xover	0.0	0.2	0.4	0.6	0.8	1.0
0.7	4104	5479	6371	6724	7220	7667
0.8	4592	5716	5947	7154	7272	7503
0.9	4126	5723	6603	6970	7499	6997
1.0	4370	5712	6431	7227	7375	6984

Figure 16: Performance of uniform cross-over on McCallum's Maze with 3 memory states

Times out of 100 optimal policy was found

	Mutation			
Xover	.065	.075	.085	.095
0.7	45	48	32	47
0.8	51	52	45	50
0.9	54	56	55	54
1.0	44	47	51	54

Average number of backward iterations

	Mutation			
Xover	.065	.075	.085	.095
0.7	5190	5060	5184	5409
0.8	5348	5452	5400	5447
0.9	5703	5560	5595	5570
1.0	5571	5837	5603	5797

Uniform Cross-over

(Syswerda 1989) presents a cross-over operator which makes inversion and reordering unnecessary. His cross-over takes advantage of schemata which include genes spread far away along the chromosome. Uniform cross-over, as he presents it, is implemented by choosing a random bit-map for any pair of parents being crossed over. Each locus in a chromosome has a corresponding bit in this bit-mask, where a 0 indicates that the gene will be taken from the first parent, while a 1 indicates that the gene will be taken from the second parent. A second child is produced by taking the genes not used in constructing the first child.

Syswerda suggests, and empirically demonstrates, that uniform cross-over is a superior method of cross-over for a population where the loci of genes are poorly related to the interactions of the genes in a solution. The explanation for this result is based on Syswerda's analysis of the preservation and combination of schemata of differing sizes and lengths. He lets the defining length of a schema be the number of loci between the leftmost and rightmost loci, inclusive, and describes the number of defining loci in a chromosome as the number of loci in a schema which are not "#". (It should be noted here that Syswerda dealt solely with populations of chromosomes composed of bits. It is not difficult, however, to see how his results can be applied to populations with more than two symbols possible for any gene).

While the survival rates for schemata of any defining length were found to be significantly lower using uniform cross-over than more conventional cross-overs, the rate at which schemata were combined was found to be higher in uniform cross-over for schemata of long defining lengths. A combinatorial analysis of the number of schemata of a given number of defining bits revealed that as the defining length of the schema increased, so did the number of different schemata of the given number of defining bits. Since we cannot be sure how genes in near or far loci will interact without more information about the POMDP, valuable schemata are more likely to appear with long defining lengths. According to Syswerda, these schemata are more likely to be combined with other schemata when using uniform cross-over.

The results for our runs on McCallum's Maze are in Figure 16. Uniform cross-over did better at finding the optimal policy than the original algorithm, and took approximately the same amount of time.

Dynamic Operator Rates

In our analysis of the original genetic algorithm, we tried to find optimal cross-over rates and mutation rates. (Davis 1989) describes a method in which the operator rates are dynamic. The better an operator does, the more likely it is to be used. Of course, his work only makes sense in a steady-state genetic algorithm. We implemented such an algorithm, as proposed by (Holland 1975).

A steady-state genetic algorithm differs from the original genetic algorithm of this paper in that instead of modifying the entire population in one generation, only one or two individuals are modified. In our implementation, the population is kept sorted so that the best policies are at the front of an array, and the worst are at the back. At each generation, an operator is selected: cross-over, mutation, or copy. The cross-over we use is uniform cross-over. The mutation operator remains the same. Copy is a new operator to fill a possible gap: if in a generation, neither cross-over nor mutation are selected, copy is selected. If copy is selected, an individual is selected and is copied back into the population.

The selection routine used in this algorithm is an exponential-ranking algorithm, where the best policy is given a value of .95, the next best is given a value of $.95^2$, and so on (this is the same kind of selection routine used in (Davis 1989)). When a new individual is created, it is evaluated and inserted into the population. If it is worse than all members of the population, it is not actually added. If it is better than some individuals, it is inserted above them, and the worst individual is removed from the population. Termination, as we have already seen, cannot be based on gene convergence. Because generations affect only a small number of individuals, and not the entire population, our old termination criteria seems inadequate. Therefore, our termination routine is as follows: if the same best policy occurs in more than five times the population size number of generations, improvement is not taking place and the algorithm halts.

We used this steady-state algorithm as a control against a modified version of Davis' algorithm.

(Davis 1989) suggests that, as a genetic algorithm advances, operators become more (or less) useful towards evolving a good individual. He also notes that sometimes, an operator will not yield a good individual directly, but will set the stage for a good individual to be generated down the road. Each time a new best individual is found, he rewards the operators responsible.

Our algorithm is a modified form of Davis' work. In it, cross-over and mutation are given initial rates. Each individual in the population also has a weight for cross-over, mutation, and copy. Initially, the cross-over and mutation weights are both 0.5. As the GA progresses, whenever an individual is produced, it stores the operator that produced it in this way: half of the weight applied to the other two operators is removed from them and given to the current operator. So, for instance, if an individual had a cross-over weight of .3, a mutation weight of .3, and a copy weight of .4, and it was mutated, it would then have a cross-over weight of .15, a mutation weight of .65, and a copy weight of .2. During cross-over, the weights of the parents are distributed equally between the children. Thus, if parent one had a cross-over weight of .6 and a mutation weight of .4, and parent two had a cross-over weight of .8 and a copy weight of .2, each child would start off with a cross-over weight of .7, a mutation weight of .2, and a copy weight of .1. Then the weights of mutation and copy would be halved, and the children's cross-over weights would be increased, so that after cross-over was complete, each child would have a cross-over weight of .85, a mutation weight of .1, and a copy weight of .05.

Whenever a new best policy is found, each operator rate has its difference with the new best policy's corresponding operator weight added to it. So if a new best policy is generated with cross-over weight of .8, mutation weight of .1, and copy weight of .1, and the current cross-over rate is .6, the current mutation rate is .1, and the current copy rate is .3, then the new cross-over rate would be

cross-over rate $= .8 + (.6 - .8) * .5 = .7$
mutation rate $= .1 + (.1 - .1) * .5 = .1$
copy rate $= .1 + (.3 - .1) * .5 = .2$.

We ran tests for several different initial cross-over and mutation rates. The results are summarized in Figures 17 and 18. While there was noticeable improvement in the finding of an optimal policy, the new algorithm also took more policy evaluations to converge. This may be explained by the algorithm's search for good operator values as well as good policies.

Figure 17: Performance of steady-state GA on McCallum's Maze with 3 memory states

Times out of 100 optimal policy was found

	Mutation			
Xover	.065	.075	.085	.095
0.7	7	14	17	20
0.8	16	18	19	25
0.9	15	15	29	20
1.0	8	5	4	4

Average number of backward iterations

	Mutation			
Xover	.065	.075	.085	.095
0.7	1216	1254	1343	1456
0.8	1364	1466	1457	1643
0.9	1657	1627	1665	1760
1.0	1442	1410	1415	1466

Figure 18: Performance of steady-state GA with dynamic operator rates on McCallum's Maze with 3 memory states

Times out of 100 optimal policy was found

	Mutation				
Xover	0.1	0.2	0.3	0.4	0.5
0.5	17	36	39	47	57
0.6	21	30	36	42	–
0.7	27	32	39	–	–
0.8	21	30	–	–	–

Average number of backward iterations

	Mutation				
Xover	0.1	0.2	0.3	0.4	0.5
0.5	1349	1560	1689	1772	1918
0.6	1479	1620	1754	1862	—-
0.7	1535	1702	1782	—-	—-
0.8	1565	1612	—-	—-	—-

Conclusions

In summary, we have experimented with different GA parameters for finding good limited finite memory policies for POMDPs. We discovered that higher cross-over rates and higher mutation rates produced better policies, but took longer. Better policies were also produced when the population was initialized with policies generated from small genetic algorithms. A ranked selection routine worked better than our original selection routine on larger problems. Our original termination method still seems to be the only viable one, though it can be modified. Allowing the best policy to repeat more often gives the genetic algorithm more chances to evolve an even better policy. Due to epistasis, however, the common termination criterion of gene termination is not feasible for our work.

Cross-over performed better when inversion was used, allowing groups of closely-related genes to clump together on a chrosome. Uniform cross-over performed even better and negated the need for searching the added search space of loci configurations.

Finally, we attempted a dynamic-operator algorithm based on the work of (Davis 1989). It did significantly better than our control steady-state algorithm.

Future work should concentrate on a better termination criterion and on additional genetic operators handcrafted to take advantage of the structure of POMDPs.

Acknowledgments

This work was partially supported by NSF grant CCR-9610348, and done while the first author was at the University of Kentucky.

References

Cassandra, A. 1997–99. Tony's POMDP page. `http://www.cs.brown.edu/research/ai/pomdp/index.html`.

Davidor, Y. 1991. Epistasis variance: A viewpoint on ga-hardness. In Rawlins, G., ed., *Foundations of Genetic Algorithms*. Morgan Kaufmann Publishers. 23–25.

Davis, L. 1989. Adapting operator probabilities in genetic algorithms. In Schaffer, J., ed., *Proceedings of the Third International Conference on Genetic Algorithms*, 61–69. Morgan Kaufmann Publishers.

Goldberg, D. 1990. The theory of virtual alphabets. In Schwefel, H., and Manner, R., eds., *Parallel Problem Solving from Nature*. Springer-Verlag. 13–22.

Hansen, E. 1998. *Finite-Memory Control of Partially Observable Systems*. Ph.D. Dissertation, Dept. of Computer Science, University of Massachussets at Amherst.

Holland, J. 1975. *Adaptation in Natural and Artificial Systems*. Univ. Michigan Press.

Lusena, C.; Li, T.; Sittinger, S.; Wells, C.; and Goldsmith, J. 1999. My brain is full: When more memory helps. In *Proceedings of the Fifteenth Conference on Uncertainty in Artificial Intelligence*, 374–381.

McCallum, R. 1993. Overcoming incomplete perception with utile distinction memory. In *Proceedings of the Tenth International Conference on Machine Learning*, 190–196. Morgan Kaufmann Publishers.

Meuleau, N.; Kim, K.-E.; Kaelbling, L. P.; and Cassandra, A. R. 1999. Solving POMDPs by searching the space of finite policies. In *Proceedings of the Fifteenth Conference on Uncertainty in Artificial Intelligence*, 417–426.

Mundhenk, M.; Goldsmith, J.; Lusena, C.; and Allender, E. 1999. Complexity results for finite-horizon Markov decision process. to appear, Journal of the ACM.

Naudts, B.; Suys, D.; and Verschoren, A. 1997. Epistasis as a basic concept in formal landscape analysis. In Back, T., ed., *Proceedings of the Seventh International Conference on Genetic Algorithms*, 65–72. Morgan Kaufmann Publishers.

Papadimitriou, C., and Tsitsiklis, J. 1987. The complexity of Markov decision processes. *Mathematics of Operations Research* 12(3):441–450.

Sondik, E. 1971. *The Optimal Control of Partially Observable Markov Processes*. Ph.D. Dissertation, Stanford University.

Syswerda, G. 1989. Uniform crossover in genetic algorithms. In Schaffer, J., ed., *Proceedings of the Third International Conference on Genetic Algorithms*, 61–69. Morgan Kaufmann Publishers.