When to Apply the Fifth Commandment: The Effects of Parenting on Genetic and Learning Agents

Michael Berger* Jeffrey S. Rosenschein
School of Engineering and Computer Science
Hebrew University
Jerusalem, Israel
{mberger,jeff}@cs.huji.ac.il

Abstract

This paper explores hybrid agents that use a variety of techniques to improve their performance in an environment over time. We considered, specifically, genetic-learning-parenting hybrid agents, which used a combination of a genetic algorithm, a learning algorithm (in our case, reinforcement learning), and a parenting algorithm, to modify their activity. We experimentally examined what constitutes the best combination of weights over these three algorithms, as a function of the environment's rate of change.

For stationary environments, a genetic-parenting combination proved best, with genetics being given the most weight. For environments with low rates of change, genetic-learning-parenting hybrids were best, with learning having the most weight, and parenting having at least as much weight as genetics. For environments with high rates of change, pure learning agents proved best. A pure parenting algorithm operated extremely poorly in all settings.

1. Introduction

1.1. Genetic and Learning Algorithms

Among the most prominent types of heuristic algorithms used in dealing with NP-hard problems are genetic algorithms. Genetic algorithms operate by representing every possible solution to a given problem as a series of "genes". Multiple agents, each containing its own series of genes, start working to solve the problem. As in biological evolution, these genes are developed over many generations by a repeated process of genetic crossovers, mutations, evaluations, and "survival of the fittest". Over time, the developed genes contain information that is gathered over these multiple generations, and hopefully, they eventually provide an optimal or near-optimal solution to the problem.

The main disadvantage of genetic algorithms is that they are mainly suitable for dealing with stationary problems. Problems, however, may be of a dynamic nature, i.e., have a hidden parameter, or state, that occasionally changes. In such dynamic situations, when a change occurs in the problem's state, genetic algorithms tend to collapse and start over again, searching for a new optimal solution, often without much success [4].

A modification that can help genetic algorithms handle dynamic problems is to combine them with a learning algorithm. Such an approach has been used in developing both agents that handle stationary environments [8] and agents that handle dynamic environments [9].

Genetic algorithms and learning algorithms are inherently different, in that genetic algorithms gather information over many generations, while learning algorithms gather information only over a single generation. Herein lies the flexibility of learning—it is unbiased by information gathered in previous generations, and is therefore more suitable for dealing with a dynamic problem, where information from previous generations might not be relevant. On the other hand, a learning process might actually be missing *some* important information from previous generations, causing it to be more wasteful in relearning information, and ultimately less accurate [7].

1.2. Parenting Algorithms

In the discussion above of algorithms that deal with dynamic problems, two extremes have been presented: a genetic algorithm, which has high accuracy but low relevancy, and a learning algorithm, which has low accuracy but high relevancy. One might consider how algorithms that are "in-between" might fare—specifically, for a problem with the following two conditions:

Condition C1 The problem state can only change to a state adjacent to it in the state space, with adjacency defined

* Michael Berger is a student.

by some metric over that space.

Condition C2 The problem has a very low (but positive) dynamic rate of change.

For such problems, it seems that an in-between algorithm might provide more accuracy than learning, while still preserving an adequate level of relevancy, because *C1* and *C2* combined imply that a current problem state is somewhat similar to problem states that were encountered over the last few generations.

Thus, we introduce the notion of parenting. We define a *parenting algorithm* as one in which agents follow instructions from their parents, i.e., agents of the previous generation, and the parents determine their instructions according to what they themselves had learned over the *complete* course of their generation—*a posteriori*. In contrast to learning agents, parenting agents do not execute their own actions according to what they learn for themselves.

A parenting algorithm has higher accuracy than a learning algorithm, because it avoids decisions based on momentary experience, which might be more prone to error [5]. Conversely, a parenting algorithm has lower accuracy than a genetic algorithm, because its information is gathered over the span of a single generation, and not over many generations.

As for relevancy, in a given generation, a parenting algorithm has lower relevancy than a learning algorithm, because its information is gathered in the previous generation and not in the current one. Conversely, a parenting algorithm has higher relevancy than a genetic algorithm, because its information is unbiased by whatever happened in generations before the previous one.

A parenting algorithm seems to fit the definition of an "in-between" algorithm. Can it help handle a problem that fulfills conditions *C1* and *C2*? This question constitutes one focus of this paper.

1.3. Overview of the Paper

In Section 2, we describe the overall environment of our experiments, and what constituted our goal for the agents. In Section 3, the hybrid agents themselves are presented, along with their genetic, learning, and parenting aspects. Section 4 describes the various experimental runs that were made, while Section 5 presents the results of those experiments. Our conclusions are discussed in Section 6.

2. Environment and Task

2.1. Environment

We define a *round* to be the basic time unit. The environment consists of a rectangular grid, with dimensions $w^{Env} \times h^{Env}$. The grid contains agents and food. In every round, each agent can either "eat" or "starve". This environment is generally similar to those used in [8, 9]. To save some analytical complications, the grid is defined to be cyclic.

In each round, each square on the grid may contain any number of agents, and it may or may not contain unlimited food—unlimited in the sense that when food is present in a square, all agents in that square are said to have eaten (unlike [8]). The reason for making food unlimited is to avoid dependencies between the actions and performance of different agents, thus avoiding the complexity of social interactions. At the end of a round, an agent may travel to an adjacent square or stay put.

The appearance or non-appearance of food is actually controlled by structures in the environment to which agents are oblivious. These are *food patches*. A food patch is a positive probability function P_{Food} defined over a group of adjacent squares on the grid. If s is a square, then $P_{Food}(s)$ defines the probability that in a given round, food will appear in square s. The environment contains n^{Env}_{Patch} food patches.

A fundamental attribute of the environment is α^{Env}, which is the probability that in a given round, the food patches move (i.e., their domains move). When this occurs, *all* food patches move one square, each in its own random direction.

2.2. Mapping the Environment to an Abstract Problem

If we define an abstract problem as that of finding food in an environment that has a pre-determined set of n food patches, each with a pre-determined shape, then a given position of all food patches actually constitutes one state of that problem. For food patch i, let (x_{Food_i}, y_{Food_i}) be any representational point from within it.

A metric can then be defined on the $2n$-dimensional space of these states by defining any vector v in that space, $v \equiv (x_1, y_1, \ldots, x_n, y_n)$, to denote the positions of the representational points of all food patches in that state, i.e., $\forall i \, (x_{Food_i} = x_i \wedge y_{Food_i} = y_i)$. Thus, the environment presented above in Section 2.1 fulfills condition *C1*. For very low positive values of α^{Env}, it fulfills condition *C2* as well.

2.3. Task

The goal in our experiments has been to synthesize an agent that enjoys a maximal *eating-rate*—the fraction of rounds in which it eats, out of the total number of rounds. This task definition requires more detail, in two respects: 1) how to integrate agent generations in the definition, and 2) how to measure the success of agents.

An integration of agent generations in the task definition is required since the development of genetic agents and parenting agents does not occur in a single agent's lifetime, but over many lifetimes. Thus, the concept of *generation* is defined: a generation of agents is a group of agents that start their life in the environment simultaneously, and end their life in the environment simultaneously. To avoid having some agents receive unfair advantages over others, all agents of the same generation start at the same position, a position that is randomly determined for each generation.

It is also important to note that the environment is completely oblivious to the concept of generations. This means that the position of food patches is not reset at the start of a new generation—the environment is "continuous" across generations.

When considering how to measure agent success, such measurement must not consist of only the initial generation; agents that develop over multiple generations must be allowed time to develop. Also, the measurement must include a large enough number of generations so as to increase the statistical validity of the results.

Having presented those aspects above that required more detail, a more accurate task can now be formulated. For a given agent generation, let a *BER (best eating-rate)* be defined as the maximal eating-rate of all agents in that generation. In a run of n^{Task}_{GenRun} agent generations, each consisting of n^{Task}_{Ag} agents that live for n^{Task}_{Rnd} rounds, let the measure of success λ be defined as the average over the BERs of the last $n^{Task}_{GenTest}$ generations. Our goal is then to develop agents that maximize λ.

3. Agents

The agents used in our experiments are hybrids of a genetically-developing agent, a self-learning agent, and a parenting agent. Before defining these agent types, a few definitions are in order.

3.1. Definitions

Perception An element of the set:

$$\{0,\ldots,w^{Env}-1\}\times\{0,\ldots,h^{Env}-1\}\times\{0,1\}$$

Indicates a position (of the agent) on the grid, and whether the agent found food in that position.

Reward That part of the perception which indicates whether the agent found food (reward is 1) or not (reward is 0).

Action A direction of movement (for the agent to move next). One of EAST, SOUTH, WEST, NORTH, HALT.

Memory A sequence containing the agent's last perception-action pairs. A memory of length x will contain the last perception, preceded by the previous $x-1$ perception-action pairs.

MAM Memory-Action Mapper. Receives a memory as input, and returns an action as output.

ASF Action-Selection Filter. Receives several actions as input (i.e., action suggestions), and returns one of them as output.

3.2. Genetic Agents

A genetic agent is one that selects its actions according to its predetermined gene sequence, and can mate with other genetic agents to produce new offspring that contain their own gene sequences [1].

A genetic agent holds a memory of length m^{Gen}. In addition, the genetic agent employs genes, with each gene composed of a *key* and a *value*. The key is a possible memory, and the value is a possible action.

The MAM of a genetic agent is a gene sequence, containing one gene for each possible memory. The position of a gene in the sequence is defined uniquely by its key (i.e., the gene sequences of all genetic agents in the same environment and with the same memory length contain genes with an identical order of keys).

The MAM of a genetic agent is created when the agent is created, and it is never updated afterwards. When the MAM receives a memory as input, it maps it to the returned action by finding the key that matches the memory, and then returning its accompanying value.

Genetic algorithms employ generations of genetic agents, letting each agent run in the environment and then producing a new generation of agents by mating agents from the current generation. Conventionally, the mating stage consists of two parts: *generational selection* and *offspring creation*. Here, only offspring creation is implemented within the genetic agent. As for generational selection, it is implemented elsewhere, and is described in Section 3.5.1.

Whenever two genetic agents mate, they create two offspring, the MAMs of which are created as follows. First, a copy is created for each of the gene sequences of the parents. We will denote the sequence copies as S^1 and S^2. A simultaneous scan is then started over S^1 and S^2, gene by gene. Let g^i_j denote the gene of sequence copy S^i at position j. As previously explained, g^1_j and g^2_j have the same key. For any j, when the scan reaches position j, the following events may occur:

Crossover This event occurs with probability P^{Gen}_{Cros}. When it occurs, all genes at positions $k \geq j$ are swapped between S^1 and S^2 (i.e., the complete sub-sequences of genes starting at g^1_j and g^2_j).

Mutation This is an event that occurs with probability P_{Mut}^{Gen} for each of the sequences. When it occurs for sequence S^i, the value of gene g_j^i changes to a randomly selected value (action).

After the scan is complete, the offspring i receives sequence i as the gene sequence in its MAM.

3.3. Learning Agents

A learning agent is an agent that selects its actions according to a learning algorithm. The most appropriate type of learning algorithm for this environment is reinforcement learning [6], in which agents receive after every action a signal that informs them how good their choice of action was (here, the presence or absence of food in their position). The reinforcement learning algorithm that was selected for the learning agent was Q-learning with Boltzmann exploration [10].

The learning agent holds a memory of length m^{Lrn}. When the agent's MAM receives memory as input, it maps it to the returned action according to the Q-learning with Boltzmann exploration algorithm.

In the general Q-learning algorithm, an agent tries to maximize rewards that it receives from the environment by improving its action selection. The Q-learning algorithm works by estimating these rewards for all possible actions, in light of the current memory. Let s denote a memory, and let a denote an action. Then the value $Q(s,a)$ (termed a *Q-value*) is defined as the expected discounted sum of future rewards obtained by taking action a when the memory contains s, and following an optimal policy thereafter. In our setting, all Q-values are initialized to 0.5, to prevent bias. At learning step n, the discounted sum of future rewards is defined as $\sum_{i=0}^{\infty} \gamma^i r_{n+i}$, where $0 \leq \gamma < 1$ is a discount factor and r_j is the reward given at learning step j.

If a is selected as the next action, $Q(s,a)$ is updated after receiving the subsequent reward r, as follows:

$$\Delta Q(s,a) = \alpha^{Lrn} \left[r + \gamma^{Lrn} \max_b Q(s',b) - Q(s,a) \right]$$

where α^{Lrn} is the algorithm's learning rate, and γ^{Lrn} is the discount factor.

For a current memory s, what action should be selected next? One trivial answer is to select the action with the highest Q-value for s, thus "exploiting" the Q-value. However, according to [10] (and, in another context, [3]), the algorithm operates better if an element of exploration is added to action-selection, thus improving the comparison that a Q-learning agent makes among different actions.

In [10], a Boltzmann exploration method was selected. According to this method, every possible action a_i has the following probability to be selected:

$$p(a_i) = \frac{e^{\frac{Q(s,a_i)}{t}}}{\sum_a e^{\frac{Q(s,a)}{t}}}$$

where t is a computational parameter that controls the amount of exploration, and in learning step n, $t = f_{Temp}^{Lrn}(n)$. t is an annealing temperature that decreases over time, thus increasing exploitation and decreasing exploration (the action with the highest Q-value receives probabilities that approach 1). Also, a freezing temperature t_{Freeze}^{Lrn} has been defined, so that when $t < t_{Freeze}^{Lrn}$, exploration (which is better for early steps) ceases completely, and full exploitation kicks in.

3.4. Parenting Agents

A parenting agent is an agent that differentiates between *learning* experiences from the environment, and *performing actions* based on that experience. In fact, when a parenting agent decides on an action to perform, its experience has absolutely no weight in that decision. Instead, the agent turns to another parenting agent, its "parent", and seeks advice about what action to perform. The parent, in turn, uses its own experience to return an answer. Note that the parent is an agent that interacted with the environment in the previous generation, but no longer does so.

A group of methods that seems appropriate for such parenting agents is Monte Carlo methods, or MC-methods for short [11]. MC-methods are used for evaluating and improving policies, i.e., memory-action mappings. In contrast to Q-learning, with MC-methods the evaluation and improvement are performed after complete *episodes* of rounds, and not after every single round. In the current context, an episode is simply a generation.

In the evaluation stage, the mapping of each possible memory to each possible action is evaluated. The evaluation produces a value, termed an *MC-value*. Let s be a memory and a an action. The MC-value of (s,a) is the average of rewards in all rounds following the agent's encounter of memory s and selection of a as the next action. In our setting, all MC-values are initialized to 0.5, to prevent bias.

In case the agent encountered memory s and responded with action a more than once, the following approach was used: for every encounter of the agent with (s,a), the average of rewards in all rounds following that encounter was calculated. Then, the MC-value was set to the average of these averages. This method is called an *every-visit* MC-method.

In the improvement stage, the parenting agent stores in its MAM a mapping of memories to actions, to be used only by the agent's offspring. Any possible memory s is mapped to an action a^* by simply selecting $a^* = \arg\max_a MC(s,a)$. In addition to the MAM, the

parenting agent holds a memory of length m^{Par}. Also, the parenting agent has an ASF component.

So far, the functions of parents and offspring have been discussed, without mentioning the number of parents that each offspring has. In this experiment, we used one of nature's models—every offspring has two parents. If they both give the offspring advice, which advice does it take? In principle, this is controlled by the ASF component. In our experiments, each parent has an equal chance for its advice to be accepted by offspring.

One final note concerns the exploration issue. As noted in [11], MC-methods usually require exploration for them to be effective. Otherwise, deterministic policies lead to the exploration of only a single action for every memory. However, the intention here was purposely to develop a "pure" parenting function, where the parent always gives advice that it deems the best—i.e., the most exploitive one. In contrast to learning, here exploration is not essential, because a parenting agent does not start its life with zero-experience—it has its parents' experience.

3.5. Complex Agents and Processes

As mentioned above, the agents that are situated in our environment are actually hybrids of genetic, learning, and parenting agents. The agent types mentioned in Sections 3.2, 3.3 and 3.4 are not situated directly in the environment. Rather, the agent population is made up of agents that are termed *complex agents*. Each complex agent contains within it an instance of each of the mentioned types (genetic, learning, and parenting), in a subsumption architecture [2]. Figure 1 helps visualize this structure. The complex agent mediates between the genetic, learning, and parenting agents, on the one hand, and the environment, on the other. The complete processes of mating, receiving a perception, and executing an action are detailed in the following subsections.

3.5.1. Mating At the end of a generation's run, the performance of each complex agent is evaluated by its eating-rate. The evaluations' average m and standard deviation d are calculated, and the agents are classified in strata of width d around m. Each agent is then awarded mating rights, determined by its startum [1], while preserving the principle of "survival of the fittest" and some genetic variance. Mating rights specify the number of matings to which the agent is entitled. This is the generational selection part of the genetic algorithm, which was mentioned in Section 3.2. When two complex agents mate, two offspring are created.

The course of an individual mating process is as follows. Two complex agents mate and create two offspring by mating their respective inner genetic agents, as well as their respective inner parenting agents, and then encapsulating the inner offspring in new complex agent offspring. There are three points that should be noted:

1. The inner genetic offspring receives genetic information from its parents during mating, while the inner parenting offspring does not.
2. In any complex offspring, the parents of its inner parenting agents are those associated with the parents of the inner genetic agents—i.e., the inner parenting agents take advice from their real parents, and not foster parents.
3. The inner learning agents of the complex offspring have absolutely no association with the inner learning agents of the complex parents.

3.5.2. Receiving a Perception The process of receiving a perception is as follows. When the complex agent receives a perception, it passes it on to the inner genetic, learning, and parenting agents. The genetic agent updates its memory accordingly. The learning agent updates its memory and its MAM. The parenting agent updates its memory and its MAM (this is equivalent to updating the MAM only at the generation's end, because the MAM isn't used during the current generation).

Note that the parents from the previous generation are completely unaware of the perception—they do not learn any new information.

3.5.3. Executing an Action The flow of the action execution process is depicted in Figure 1. When the complex agent executes an action, it asks its inner agents to suggest it.

The genetic agent feeds its memory to its MAM, and suggests the returned action as the action to execute.

The learning agent feeds its memory to its MAM, and suggests the returned action as the action to execute.

The parenting agent feeds its memory to the MAMs of its parents. The parenting agent feeds the returned actions to its ASF, and then suggests the action returned by the ASF as the action to execute.

The actions suggested by the inner agents are fed to the complex agent's ASF, which selects the genetic agent's suggestion with probability P_{Gen}^{Comp}, the learning agent's suggestion with probability P_{Lrn}^{Comp}, and the parenting agent's suggestion with probability P_{Par}^{Comp}. The complex agent executes the action selected by its ASF, and also perceives it by causing the inner agents to perceive it in the usual manner (see above, Section 3.5.2).

4. Runs

This section specifies which parameters were set to constant values, which were treated as variables, and what runs were performed in our experiments.

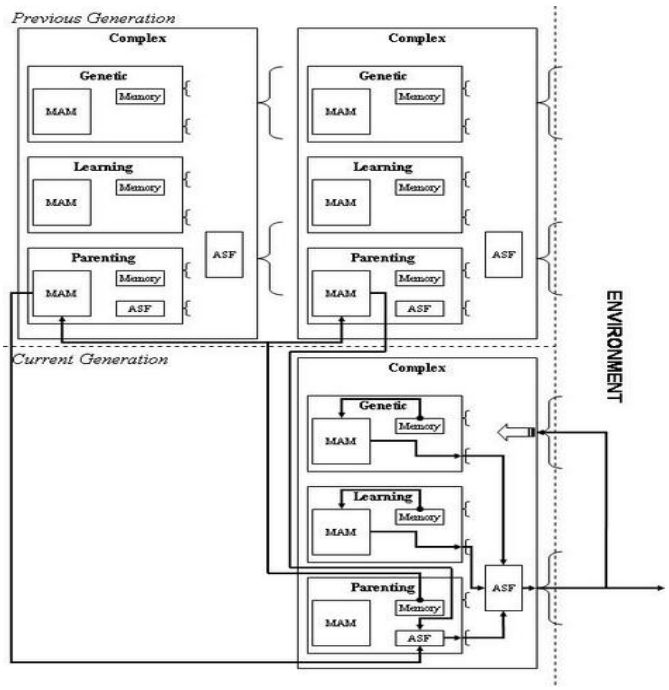

Figure 1. Action Flow

4.1. Constants

$n_{Ag}^{Task} = 20$, $n_{GenRun}^{Task} = 9500$, $n_{GenTest}^{Task} = 1000$, $n_{Rnd}^{Task} = 30000$, $h^{Env} = 20$, $w^{Env} = 20$, $n_{Patch}^{Env} = 1$, $m^{Gen} = 1$, $P_{Cros}^{Gen} = 0.02$, $P_{Mut}^{Gen} = 0.005$, $m^{Lrn} = 1$, $\alpha^{Lrn} = 0.2$, $\gamma^{Lrn} = 0.95$, $f_{Temp}^{Lrn}(n) = 5 \cdot 0.999^n$, $t_{Freeze}^{Lrn} = 0.2$, $m^{Par} = 1$.

The single food patch is 5×5 in size. Its center has a value of 0.8. All inner frame squares have a value of 0.4. All outer frame squares have a value of 0.2.

4.2. Variables

The parameters that constituted independent variables were as follows: α^{Env}, P_{Gen}^{Comp}, P_{Lrn}^{Comp}, P_{Par}^{Comp}. δ (used below) will be defined as $(P_{Gen}^{Comp}, P_{Lrn}^{Comp}, P_{Par}^{Comp})$. The only dependent variable was λ.

4.3. Summary of Runs

For a given α^{Env}, all values of δ that fulfill the following conditions were tested:

1. $P_{Gen}^{Comp} \geq 0, P_{Lrn}^{Comp} \geq 0, P_{Par}^{Comp} \geq 0$.
2. $P_{Gen}^{Comp} + P_{Lrn}^{Comp} + P_{Par}^{Comp} = 1$.
3. $P_{Lrn}^{Comp} = 0.1 \cdot n$,
   $P_{Gen}^{Comp} = 0.1 \cdot k \cdot \left(1 - P_{Lrn}^{Comp}\right); \quad n, k \in N$.

Under the above conditions, there are 11 different values that P_{Lrn}^{Comp} can receive (i.e., 0, 0.1, ..., 1). For each such value, there are 11 values that P_{Gen}^{Comp} can receive, except for $P_{Lrn}^{Comp} = 1$, where P_{Gen}^{Comp} can receive only one value (0). Therefore, for a given α^{Env}, there are 111 values of δ that fulfill the above conditions.

There were 7 different values of α^{Env} that were tested: $0, 10^{-6}, 10^{-5}, 10^{-4}, 10^{-3}, 10^{-2}, 10^{-1}$. Since each combination of α^{Env} and δ was tested (in one run), this amounted to a total of 777 runs.

5. Results

This section details the performance of agents by analyzing the value of λ. It is worth noting the scale by which a value of λ should be evaluated. The food patch had a peak of 0.8. Thus, an agent that can direct itself to that peak would be expected to have an eating-rate of about 0.8. Therefore, an agent could be considered "perfect" if it has an eating-rate of 0.8.[1]

Table 1 summarizes the main results—the λ-value of the pure agent types and the best performer for each value of α^{Env}. Two points are derived from this table. First, not surprisingly, λ becomes degraded as α^{Env} rises. Second, the performance of agents can be classified according to three ranges of α^{Env}: 0, $(0, K]$ and $[K, 10^{-1}]$, where $10^{-3} \leq K \leq 10^{-2}$. More details are presented in the subsections below.

	δ				
α^{Env}	(1,0,0)	(0,1,0)	(0,0,1)	Best δ	Best λ
0	0.2425	0.7233	0.0746	$(0.7, 0, 0.3)$	0.7988
10^{-6}	0.2139	0.7188	0.0730	$(0.15, 0.7, 0.15)$	0.7528
10^{-5}	0.1806	0.6912	0.0670	$(0, 0.9, 0.1)$	0.7011
10^{-4}	0.1550	0.5454	0.0520	$(0.03, 0.9, 0.07)$	0.6021
10^{-3}	0.1462	0.3252	0.0291	$(0.02, 0.8, 0.18)$	0.3647
10^{-2}	0.0805	0.1834	0.0167	$(0, 1, 0)$	0.1834
10^{-1}	0.0366	0.0698	0.0177	$(0, 1, 0)$	0.0698

Table 1. Result Summary of Runs

5.1. Pure Parenting ($P_{Par}^{Comp} = 1$)

Pure parenting is ineffective. This is evident from Table 1, and though it is true for all tested values of α^{Env}, it is most striking in Figure 2 (parenting probability is implicit in the figure since $P_{Gen}^{Comp} + P_{Lrn}^{Comp} + P_{Par}^{Comp} = 1$).

1 Even though a higher value is possible, because the food patch is a probabilistic function, for the purpose of defining a scale this fact is negligible.

A probable explanation for this degraded performance is the decision not to include exploration in the parenting agent's algorithm. Indeed, a closer look at the eating-rates of pure parenting agents lends support to this explanation. The eating-rates go through a two-generation cycle. As an example, for $\alpha^{Env} = 0$, the eating-rates of all agents in one generation usually have values between 0.005 and 0.025, while in the next generation almost all eating-rates are 0 (nicknamed a "zero-generation"). This cycle repeats. The zero-generations occasionally have up to four eating-rates higher than 0.05, and there are rare zero-generations with one or two medium or high eating-rates.

Lack of exploration could be the culprit. What seems to be happening is that in the non-zero-generation, agents start a search for food patches, which is beneficial to a certain extent. However, the search is not good enough for the MC-method to consider it a success (since the initial MC-values are 0.5). Therefore, the offspring in the next generation, which is a zero-generation, receive advice from the parents not to go where the parents went, i.e., abandon the parents' search. The obedient offspring do just that, and search in other directions, which are usually barren. Now, the offspring in the next non-zero-generation will receive truly valuable advice, not to go in the barren directions. However, for every memory, the non-zero-generation agents still face three or four other directions in which they can go, so they will be able to start a beneficial search, but only to a certain extent. Hence, the cycle.

5.2. Runs with $\alpha^{Env} = 0$

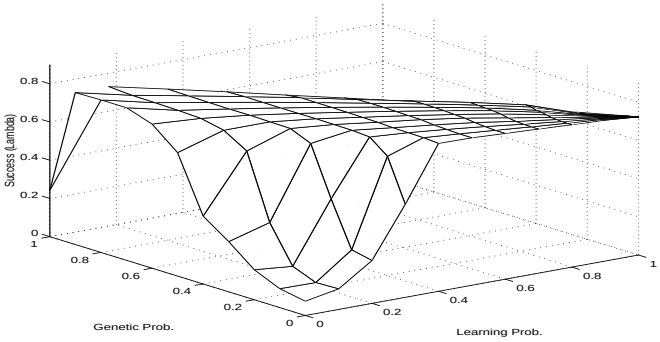

Figure 2. λ **for** $\alpha^{Env} = 0$

Figure 2 shows the results for runs with $\alpha^{Env} = 0$. The highest λ-value, 0.7988, is achieved for $\delta = (0.7, 0, 0.3)$. Generally, for a given P_{Par}^{Comp}, λ rises as P_{Lrn}^{Comp} decreases.

Surprisingly, pure genetic agents perform poorly, while the hybrids of genetic agents with learning and parenting agents achieve a very high λ. Still, in view of what is known of genetic algorithms, and since the environment is stationary, the pure genetic agent is expected to have eventually achieved a near-perfect λ had it been given enough generations to develop. Indeed, according to [6], evolution and "learning" complement each other. Learning may assist evolution by making near-perfect individuals receive high fitness scores, while evolution accelerates learning by making individuals nearly perfect to begin with. Here, learning is manifested both by learning agents and by parenting agents. Note that the acceleration of learning by evolution explains why parenting has such a positive contribution to the genetic-parenting hybrid, compared to the poor performance of pure parenting agents.

5.3. Runs with a Low Positive α^{Env}

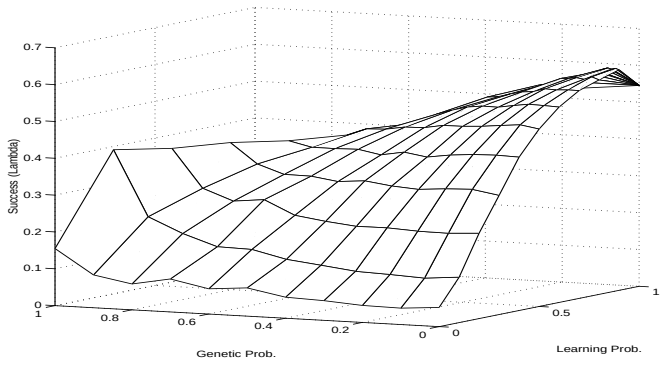

Figure 3. λ **for** $\alpha^{Env} = 10^{-4}$

The results for the runs with $\alpha^{Env} \in \{10^{-6}, 10^{-5}, 10^{-4}, 10^{-3}\}$ can be classified together. The results for one representative α^{Env}, 10^{-4}, are represented in Figure 3. In that figure, notice that the peak is higher than pure learning's λ-value. The points that can be derived from the results for these values of α^{Env} are as follows.

First, as expected, pure genetic agents perform much worse than pure learning agents. Second, pure learning agents are not the best performers. They are outperformed by hybrids where the following conditions hold:

1. $P_{Lrn}^{Comp} > P_{Gen}^{Comp} + P_{Par}^{Comp}$
2. $P_{Par}^{Comp} \geq P_{Gen}^{Comp}$

In other words, the best performers are those hybrids where learning predominates (but is not absolute), and parenting has at least as much weight as genetics. For $\alpha^{Env} > 10^{-6}$, the inequality of the second condition is a strong one—the best performer contains a mix where parenting has more

than twice the weight of genetics. Presumably, for $\alpha^{Env} = 10^{-6}$ environment changes occur at such a low rate that genetics can handle them (one change every $33\frac{1}{3}$ generations on average).

Third, The best performers of α^{Env}-values 10^{-3} and 10^{-4} perform significantly better than the pure learning agent (12.1% and 10.4% higher, respectively). For α^{Env}-values 10^{-5} and 10^{-6}, the difference is less significant (1.4% and 4.7% higher, respectively).

5.4. Runs with a Higher Positive α^{Env}

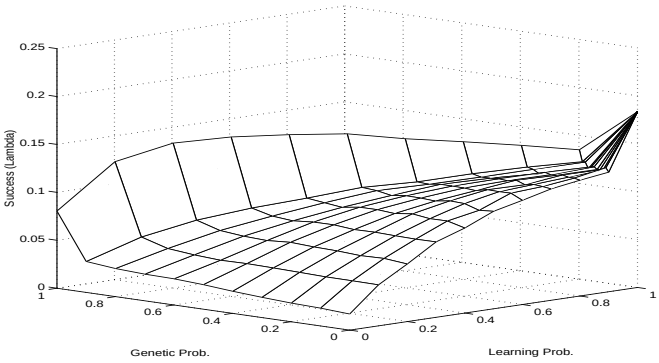

Figure 4. λ **for** $\alpha^{Env} = 10^{-2}$

The results for the runs with $\alpha^{Env} \in \{10^{-2}, 10^{-1}\}$ can be classified together. The results for one representative α^{Env}, 10^{-2}, are represented in Figure 4. As expected, for these values of α^{Env}, pure genetic agents perform much worse than pure learning agents. Also, pure learning agents are the best performers.

6. Conclusions

Let us define an agent's algorithm A to be an *action-augmentor* of an agent's algorithm B if the following conditions hold: (1) Both algorithms are always used for receiving perceptions; (2) B is applied for executing an action in most steps; (3) A is applied for executing an action in at least 50% of the other steps.

The main results of our experiments can be rephrased as follows:

- When the environment is stationary, parenting can provide a positive contribution when used as an action-augmentor for genetics. Note that in such an environment, conditions *C1* and *C2* (from Section 1.2) basically hold, although without the condition of *C2* that the rate of dynamic change be *positive* (but zero is indeed very low).
- When the environment has a very low rate of dynamic change, parenting can provide a positive contribution when used as an action-augmentor for learning. Note that in such an environment, conditions *C1* and *C2* both hold.
- When the environment has a higher rate of dynamic change, parenting loses its effectiveness, and pure learning is the best performer. Note that in such an environment, condition $C2$ does not hold.

In all cases, pure parenting performs extremely poorly.

In nature, parenting has two functions: providing for the young, and educating them. However, there are settings where the young have access to sufficient resources, eliminating the need to provide for them. An interesting point for biologists to explore might be whether for animals in such settings, a parenting role evolves, as a function of the environment's rate of change. According to this paper's results, it would be expected to evolve if and only if the environment does not change too quickly.

References

[1] R. Axelrod. *The Complexity of Cooperation: Agent-Based Models of Competition and Collaboration*. Princeton University Press, 1997.

[2] R. A. Brooks. A robust layered control system for a mobile robot. *IEEE Journal of Robotics and Automation*, 2(1):14–23, March 1986.

[3] D. Carmel and S. Markovitch. Exploration strategies for model-based learning in multiagent systems. *Autonomous Agents and Multi-agent Systems*, 2(2):141–172, 1999.

[4] H. G. Cobb and J. J. Grefenstette. Genetic algorithms for tracking changing environments. In *Proceedings of the Fifth International Conference on Genetic Algorithms*, pages 523–530, San Mateo, 1993.

[5] T. D. Johnston. Selective costs and benefits in the evolution of learning. In *Adaptive Individuals in Evolving Populations: Models and Algorithms*, pages 315–358. Addison-Wesley, 1996.

[6] M. Littman. Simulations combining evolution and learning. In *Adaptive Individuals in Evolving Populations: Models and Algorithms*, pages 465–477. Addison-Wesley, 1996.

[7] G. Mayley. Landscapes, learning costs, and genetic assimilation. *Evolutionary Computation*, 4(3):213–234, 1996.

[8] S. Nolfi, J. L. Elman, and D. Parisi. Learning and evolution in neural networks. *Adaptive Behavior*, 3(1):5–28, 1994.

[9] S. Nolfi and D. Parisi. Learning to adapt to changing environments in evolving neural networks. *Adaptive Behavior*, 5(1):75–98, 1997.

[10] T. W. Sandholm and R. H. Crites. Multiagent reinforcement learning in the iterated prisoner's dilemma. *Biosystems*, 37:147–166, 1996.

[11] R. S. Sutton and A. G. Barto. *Reinforcement Learning: An Introduction*. The MIT Press, 1998.