Evolution of Cooperative Problem Solving in an Artificial Economy

Eric B. Baum
Igor Durdanovic
NEC Research Institute, Princeton, NJ 08540

We address the problem of how to reinforce learning in ultracomplex environments, with huge state-spaces, where one must learn to exploit a compact structure of the problem domain. The approach we propose is to simulate the evolution of an artificial economy of computer programs. The economy is constructed based on two simple principles so as to assign credit to the individual programs for collaborating on problem solutions. We find empirically that starting from programs that are random computer code, we can develop systems that solve hard problems. In particular, our economy learned to solve almost all random Blocks World problems with goal stacks that are 200 blocks high. Competing methods solve such problems only up to goal stacks of at most 8 blocks. Our economy has also learned to unscramble about half a randomly scrambled Rubik's cube and to solve several commercially sold puzzles.

1 Introduction

Reinforcement learning (cf. Sutton & Barto, 1998) formalizes the problem of learning from interaction with an environment. The two standard approaches are value iteration and policy iteration. At the current state of the art, however, neither appears to offer much hope for addressing ultracomplex problems. Standard approaches to value iteration depend on enumerating the state-space and are thus problematic when the state-space is huge. Moreover, they depend on finding an evaluation function showing progress after a single action, and such a function may be extremely hard to learn or even to represent. We believe that what is necessary in hard domains is to learn and exploit the compact structure of the state-space, for which powerful methods are yet to be discovered. Policy iteration, on the other hand, suffers from the fact that the space of policies is also enormous, and also has a bumpy fitness landscape. Standard methods for policy iteration, among which we would include the various kinds of evolutionary and genetic programming, thus grind to a halt on most complex problems.

The approach we propose here is to evolve an artificial economy of modules, constructed so as to assign credit to the individual modules for their contribution. Evolution then can progress by finding effective modules.

This divide-and-conquer approach greatly expedites the search of program (a.k.a. policy) space and can result in a compact solution exploiting the structure of the problem.

Holland's (1986) seminal classifier systems previously pursued a similar approach, using an economic model to assign credit to modules. Unfortunately, classifier systems have never succeeded in their goal of dynamically chaining modules together to solve interesting problems (Wilson & Goldberg, 1989). We believe we have understood the problems with Holland's proposal, and other multiagent proposals, and how to correct them in terms of simple but fundamental principles. Our picture applies widely to the evolution of multiagent systems, including ecologies and economies. We discuss here evidence for our picture, evolving artificial economies in three different representations and for three problems that dynamically chain huge sequences of learned modules to solve problems too vast for competing methods to address. Control experiments show that if our system is modified to violate our simple principles, even in ways suggested as desirable in the classifier system literature, performance immediately breaks as we would predict.

We present results on three problems: Blocks World, Rubik's cube, and a commercially sold set of puzzles called "Rush Hour." Block's World has a lengthy history as a benchmark problem, which we survey in section 3. Methods like TD learning using a neural net evaluation, genetic programming, inductive logic programming, and others are able to solve problems directly comparable to the ones we experiment with only when they are very small and contain a handful of blocks—that is, before the exponential growth of the state-space bites in. By contrast, we report the solution of problems with hundreds of blocks, and state-spaces containing on the order of 10^{100} states, with a single reward state. On Rubik's cube we have been able to evolve an economy able to unscramble about half of a randomly scrambled cube, comparable to the performance of many humans. On Rush Hour, our approach has solved 15 different commercially sold puzzles, including one dubbed "advanced" by the manufacturer. Genetic programming (GP) was unable to make headway on either Rubik or Rush Hour in our experiments, and we know of no other learning approach capable of interesting results here.

Note that we are attempting to learn a program capable of solving a class of problems. For example, we are attempting to learn a program capable of unscrambling a randomly scrambled Rubik's cube. We conjecture that for truly complex problems, learning to solve a class is often the most effective way to solve an individual problem. No one can learn to unscramble a Rubik's cube, for example, without learning how to unscramble any Rubik's cube. And the planning approach to Blocks World has attempted a huge search to solve individual Blocks World problems, but what is needed is to learn to exploit the structure of Blocks World. In our comparison experiments, we attempted to use alternative methods such as GP to learn to solve a class of problems, but we failed. We train by presenting small instances, and our method learns to exploit the underlying compact structure of the problem domain. We conjecture that learning to exploit underlying compact structure is closely related to "understanding," a human ability little explained.

Section 2 describes our artificial economy and the principles that it embodies. Section 3 describes the Blocks World problem and surveys alternative approaches. Section 4 describes the syntax of our programs. Section 5 describes our results on Blocks World. Section 6 describes a more complex economic model involving metalearning. Section 7 describes our results with Rubik's cube. Section 8 describes our results with the Rush Hour problem. Section 9 sums up what we have learned from these experiments and suggests avenues for further exploration. Appendix A discusses results on Blocks World with alternative methods such as TD learning and genetic programming. Appendix B gives pseudocode. Appendix C discusses parameter settings.

2 The Artificial Economy

This section describes and motivates our artificial economy, which we call Hayek. Hayek interacts with a world that it may sense, where it may take actions, and that makes a payoff when it is put in an appropriate state. Blocks World (see Figure 1) is an example world. Hayek is a collection of modules, each consisting of a computer program with an associated numeric "wealth." We call the modules agents[1] because we allow them to sense features of the world, compute, and take actions on the world. The system acts in a series of auctions. In each auction, each agent simulates the execution of its program on the current world and returns a nonnegative number. This number can be thought of as the agent's estimate of the value of the state its execution would reach. The agent bids an amount equal to the minimum of its wealth and the returned number. The solver with the highest bid wins the auction. It pays this bid to the winner of the previous auction, executes its actions on the current world, and collects any reward paid by the world, as well as the winning bid in the subsequent auction. Evolutionary pressure pushes agents to reach highly valued states and to bid accurately, lest they be outbid.

In each auction, each agent that has wealth more than a fixed sum $10W_{init}$ creates a new agent that is a mutation of itself. Like an investor, the creator endows its child with initial wealth W_{init} and takes a share of the child's profit. Typically we run with each agent paying one-tenth of its profit plus a small constant sum to its creator, but performance is little affected if this

[1] This is in the spirit of the currently best-selling AI textbook (Russell & Norvig, 1995) which defines an agent as "just something that perceives and acts" (p. 7).

share is anywhere between 0 and .25. Each agent also pays resource tax proportional to the number of instructions it executes. This forces agent evolution to be sensitive to computational cost. Agents are removed if, and only if, their wealth falls below their initial capital, with any remaining wealth returned to their creator. Thus, the number of agents in the system varies, with agents remaining as long as they have been profitable.

This structure of payments and capital allocations is based on simple principles (Baum, 1998). The system is set up so that everything is owned by some agent, interagent transactions are voluntary, and money is conserved in interagent transactions (i.e., what one pays, another receives) (Miller & Drexler, 1988). Under those conditions, if the agents are rational in that they choose to make only profitable transactions, a new agent can earn money only by increasing total payment to the system from the world. But irrational agents are exploited and go broke. In the limit, the only agents that survive are those that collaborate with others to extract money from the world.

Hayek guarantees that everything is owned by auctioning the whole world to one agent, which then can alone act on the world, sell it, and receive reward from it. Agent interactions are voluntary in that they respect property rights. For example, the agent owning the world can refuse to sell by outbidding other agents, which costs her nothing since she pays herself. The guide for all ad hoc choices (e.g., the one-tenth profit fraction) was that the property holder might reasonably make a similar choice if given the option. In principle, all such constants could be learned.

By contrast, such property rights are not enforced in real ecologies (Miller & Drexler, 1988), simulated ecologies like Tierra (Ray, 1991) and Avida (Lenski, Ofria, Collier, & Adami, 1999) or other multiagent learning systems such as Lenat's Eurisko (Lenat, 1983; Miller & Drexler, 1988) or Holland classifier systems (Holland, 1986). When not everything is owned, or money is not conserved, or property rights are not enforced, agents can earn money while harming the system, even if other agents maximize their profits. *The overall problem, then, cannot be factored because a local optimum of the system will not be a local optimum of the individual agents.* For example, because in Holland classifiers many agents are active simultaneously, there is no clear title to reward, which is usually shared by active agents. A tragedy of the commons (Hardin, 1968) can then ensue in which any agent can profit by being active when reward is paid, even if its action harms system performance (Baum, 1996). Such systems evolve complex behavior but not accurate credit assignment (Miller & Drexler, 1988; Baum, 1998).

We have done runs with conservation of money broken in various ways. The system learns to exploit any way of "creating money" without solving the hard problems posed by the world (cf. Baum, 1996). Conversely, if money leaks out of the system too fast (the resource tax is a small leak, but must be kept very small, at less than 10^{-5} per instruction), the economy collapses to essentially random agents. We have also experimented with violations of property rights. For example, we did runs identical to Hayek in every

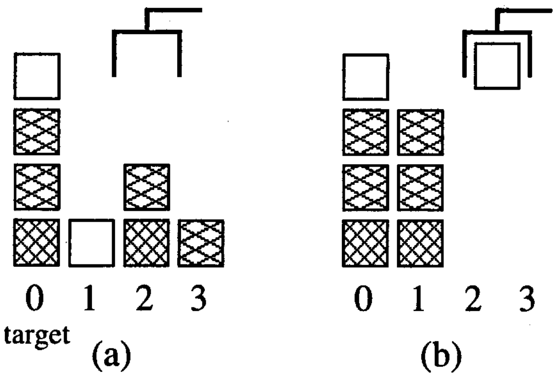


Figure 1: A Blocks World instance with four blocks and three colors. (a) Initial state. (b) The position just before solution. Hatching represents color. When the hand drops the white block on stack 1, the instance will be solved, and Hayek will see another random instance.

way, except using the procedure advocated in zeroth-level classifier systems (Wilson, 1994) of choosing the winning bidder with probability proportional to its bid. These runs never solved more than six-block problems, even if given *NumCorrect*, or made any progress on Rubik's cube.

3 Blocks World

We have trained Hayek by presenting Blocks World (BW) problems of gradually increasing size (see Figure 1). Each problem contains 4 stacks of colored blocks, with $2n$ total blocks and k colors. The leftmost stack, stack 0, serves as a template only and is of height n. The other three stacks contain, between them, the same multiset of colored blocks as stack 0. The learner can pick up the top block on any but stack 0 and place the block on top of any stack but 0. The learner takes actions until it asserts *"Done,"* or exceeds $10n \log_3(k)$ actions. If the learner copies stack 0 to stack 1 and states *Done*, it receives a reward of n. If it uses $10n \log_3(k)$ actions or states *Done* without copying stack 0, it terminates activity with no reward. Note that the goal is to discover an algorithm capable of solving random new instances. The best human-generated algorithm of which we are aware (a recursive algorithm similar to that solving the Tower of Hanoi problem that we developed in conversation with Manfred Warmuth) is capable of solving arbitrary BW problems in $4n\log_2(k)$ grabs and drops.

Blocks World (BW) has been well studied (cf. Korf, 1987; Whitehead & Ballard, 1991; Koza, 1992; Koehler, 1998; Bacchus & Kabanza, 1996; Dzeroski, Blockeel, & De Raedt, 1998). Although it is easy for humans to solve or program, its exponential-size state-space and the complexity of code that would solve it have stymied fully autonomous methods. One line of research has involved planning programs, which are told the goal and do an extensive search to attempt to solve an individual instance. Until about 1998, state-of-the-art planning programs such as Prodigy 4.0 could solve only problems involving about five blocks before succumbing to the combinatorial explosion (Bacchus & Kabanza, 1996). More recently, it has been discovered that the Blum-Furst graphplan heuristic (Blum & Furst, 1997) can be effectively used to constrain the search, postponing, although not preventing, the combinatorial explosion. The best results are by Koehler (1998), who solved a related 80-block stacking problem in 5 minutes and a 100-block problem in 11 minutes. (Note the exponential increase.) Koehler's problems are simpler than those discussed here in that all blocks start on the table, with no block on top of them, avoiding any necessity for the long, Tower-of-Hanoi-like chains of stacks and unstacks required by a three-block-wide table.

No standard method appears able to address the BW reinforcement learning problem. Previous attempts to reinforcement-learn in BW (e.g., Whitehead & Ballard, 1991; Koza, 1992; Birk & Paul, 1994; Dzeroski et al., 1998) addressed only much simpler versions not involving abstract goal discovery (they attempted to stack a fixed order of colors rather than match an arbitrary target stack); nor did they have a fixed small table size; nor did they involve goal stacks of more than a handful of blocks.

Q-learning (Watkins, 1989) is inapplicable to this problem because there are combinatorial numbers of states; there are almost 10^{100} distinct possible 200-block goal stacks and for each of these, a search space of some 10^{100} states, among which there is a single goal state. The standard approach to dealing with large state-spaces is to train a neural net evaluator in TD(λ) learning (cf. Sutton & Barto, 1998). This has been effective in some domains, such as backgammon (Tesauro, 1995). However, the success at backgammon was attributed to the fact that a linear function is a good evaluator (Tesauro, 1995). TD(λ) using a neural net evaluation function is not suitable here because the goal is so nonlinear and abstract, and the size of the problem description is variable.[2] We nonetheless attempted to train a neural net.[3] This succeeded in solving four-block problems from a raw representation or eight-block problems using a powerful hand-coded feature *NumCorrect*. *NumCorrect* returns the current number of correct blocks on stack1, the largest integer l such that the bottom l blocks of stacks 0 and 1 agree in color. A recent attempt at inductive logic programming solved only two-block problems (Dzeroski et al., 1998). We have experimented extensively with genetic programming (Baum & Durdanovic, 1999) and also

[2] For a discussion of various other difficulties, see Whitehead & Ballard (1991)

[3] We report in this paragraph results on a substantially simpler version of the problem where, rather than demanding that the learner say "done" when the stack is correct, we externally supplied the "done" statement. Run on the full problem, these standard methods did worse.

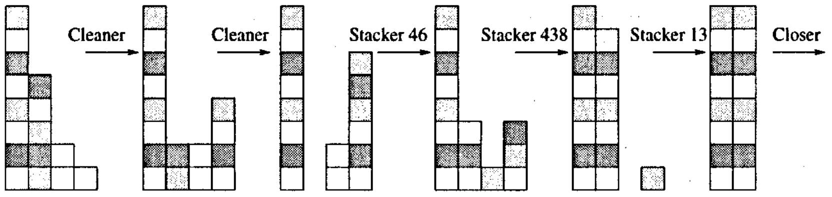


Figure 2: Data from Hayek3 runs with *NumCorrect*. The horizontal axis is in millions of instances presented. The right vertical axis is in units of agents. The left vertical axis is in units of numbers of blocks or, equivalently, reward. (A) The moving average over the last 100 instances of winning first and last bid and payoff from the world. (B) The moving average (solid line) of the score, computed as $\sqrt{2\sum_{i=1}^{200} p(i)i}$ for $p(i)$ = fraction instances of size i solved. If all instances of every size up to S were solved and no instances above size S were solved, the score would be about S. The score is computed only over instances with no new agents introduced. This approximates the average size instance being solved if agent creation is suspended. The moving average of payoff is lower than the score because new agents frequently lead to failures. A system performing at the level of score can, however, be achieved at any time by suspending creation. The dashed line shows the moving average of the number of agents in the population (measured against the right-hand axis). Effective solving coincides with a split of first and last bids, when it learns to estimate value of states.

some less standard approaches: a hill-climbing approach on computer programs, previous economic models, and an attempt to induce an evaluation function using S-expressions. These approaches all solved at most four- or five-block problems reliably.[4] (See appendix A for a fuller discussion of these results.)

4 Representation Language

This section discusses the syntax of the agent programs. The programs of the agents discussed in this article are typed S-expressions (Montana, 1994), recursively defined as a symbolic expression, as in Lisp, consisting of either a symbol or a list structure whose components are S-expressions. S-expressions are isomorphic to parse trees. A simple example is shown in Figure 6a.

All our expressions are typed, taking either integer, void, color, or boolean values. All operations respect types so that colors are never compared to

[4] A simpler economic model learned to solve arbitrary BW problems, but only if provided intermediate reward in training whenever an action was taken partially advancing solution (Baum, 1996).

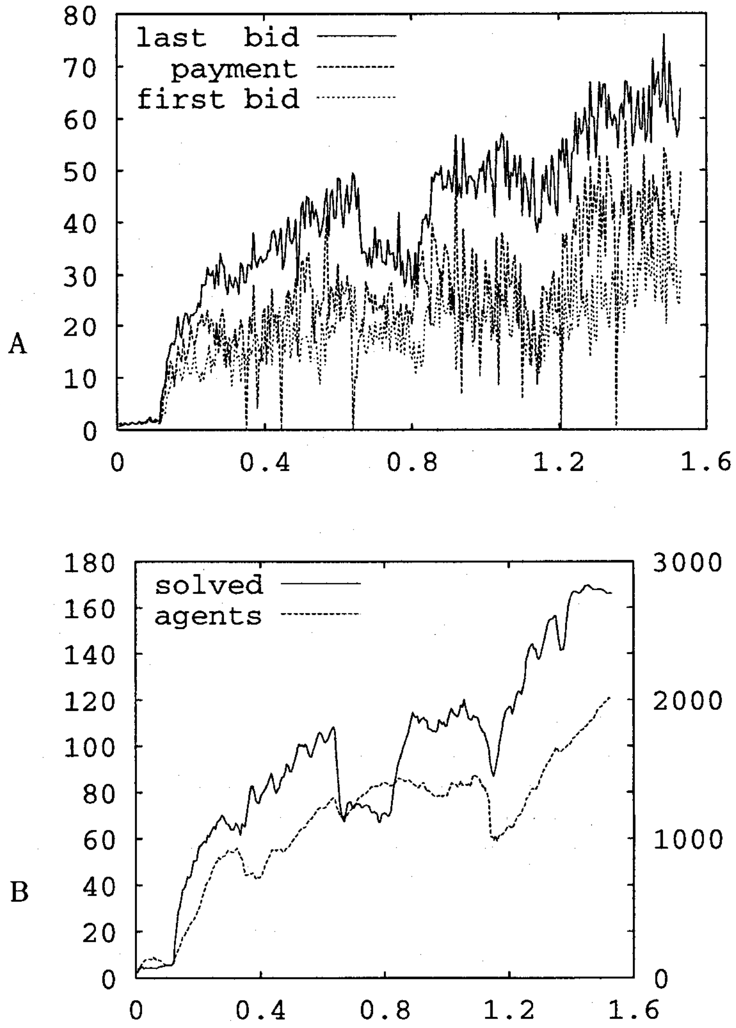
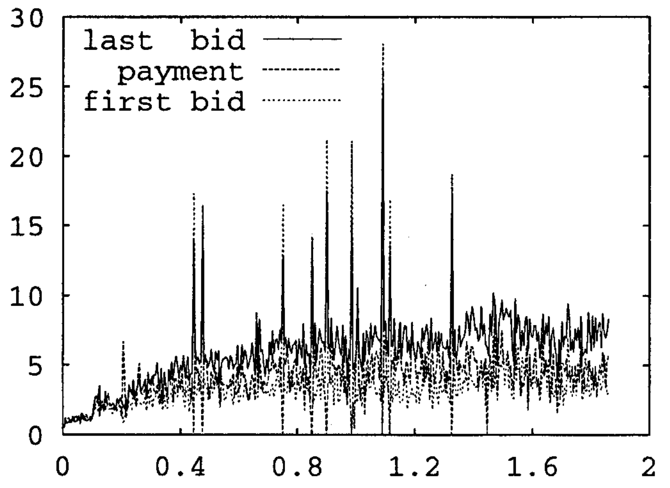


Figure 3: Example of solution of an eight-block instance.

integers, for example. The use of typing semantically constrains mutations and thus improves the likelihood of randomly generating meaningful and useful expressions.

Our S-expressions are built out of constants, arithmetic functions, conditional tests, loop controls, and four interface functions: *Look(i,j)*, which returns the color of the block at location *i, j.*, *Grab(i)* and *Drop(i)*, which act on stack *i*; and *Done*, which ends the instance. Some experiments also con-

A



B

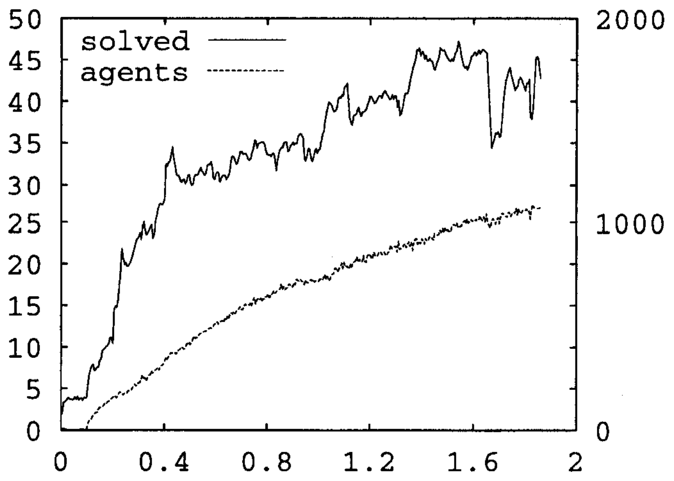


Figure 4: Data from Hayek3 runs without *NumCorrect* or *R* nodes. For further information, see the caption of Figure 2, which presents identically formatted data for a different run.

tain the function *NumCorrect*, and some contain a random node $R(i, j)$, which simply executes branch i with probability 1/2 and j with probability 1/2.

The system starts with a single, special, hand-coded agent, called *Seed*, with zero wealth. *Seed* does not bid, but simply creates children as random S-expressions in the same way that expressions are initiated in genetic programming (Koza, 1992; Montana, 1994), choosing each instruction in the

S-expression at random from the instruction set with increasing probability of choosing a terminal expression so the tree terminates. W_{init} is initially set to 0, so all agents can create freely. Eventually one of *Seed*'s descendants earns reward. Thereafter, W_{init} is set equal to the size of the largest reward earned, and agents can no longer create unless they earn money.

We report elsewhere experiments using a similar economic model, but with three other representation languages. Hayek1 (Baum, 1996) used simple productions of a certain form in some ways similar to the language used by classifier systems (Holland, 1986). In Hayek and also in classifier systems, a configuration is stable only if agents are on average followed by agents of higher bid, else agents will go broke. Conversely, a human writing programs in such a language will naturally use higher bids to prioritize agents, which may result in high-bidding agents preceding lower-bidding ones. This means that many programs that a human could write in a given language may not be dynamically stable. Although the classifier system language has been proved universal (Forrest, 1985), it is not clear whether it remains universal when restricted to stable configurations. Baum (1996) and Lettau and Uhlig (1999) independently pointed this problem out. Similarly, because of this problem, Hayek1 was able to solve large BW problems only when given an intermediate reward for partial progress.

Hayek2 used an assembler-like language inspired by Tierra (Ray, 1991). This proved both ineffective and incomprehensible to humans.

We call the system experimented with here, using typed S-expressions, Hayek3.

Hayek4, reported on in a companion paper, uses postproduction systems. This language is Turing complete, and Hayek4 succeeds in learning a program capable of solving arbitrary BW problems (of our type). By comparison, we do not believe the S-expression language used in Hayek3 is Turing complete, and so we will see that Hayek3 succeeds in building only systems capable of solving large, but finite, BW problems.

5 Blocks World Behaviors

We now report on experiments running the Hayek3 system. We trained Hayek on the following distribution of problems:

- One-block problems if there was no money in the system (e.g., until the first instance is solved).
- Else we presented problems with size uniformly distributed between 1 and $2 + m + m/5$, where m was set as follows. We maintained a table $S_p(i)$ as the fraction of the last 100 instances of size i presented that were solved. m was increased by 1 whenever $S_p(m) > 75\%$ and $S_p(m+1) > 30\%$. m was decreased by 1 whenever $S_p(m) < 25\%$ and $S_p(m-1) < 70\%$.

This distribution was chosen ad hoc, with a small amount of experimentation, to present larger instances smoothly as Hayek learned. The reason that we used different conditions for increasing and decreasing *m* was to prevent rapid oscillation between two neighboring values.

We first describe experiments with *NumCorrect* in the language. Within a few hundred thousand training instances, Hayek3 evolves a set of agents that cooperate to solve 100-block goals virtually 100% of the time and problems involving much higher goal stacks over 90% of the time. (See Figure 2.)

Our best run to date solved almost 90% of 200-block goals, the largest stacks among our tests, in a problem with $k = 3$ colors. Such problems have about 10^{100} states. Hundreds of agents act in sequence, each executing tens of actions, with a different sequence depending on the instance. The system is stable even though reward comes only at the end of such long sequences. The learning runs lasted about a week on a 450 MHz Pentium 2 processor running Linux, but the trained system solved new 200-block problems in a few seconds.

Empirically we find Hayek3 evolves a system with the following strategy. The population contains 1000 or more agents, each of which bids according to a complex S-expression that can be understood, using Maple, to be effectively equal $A \cdot \mathit{NumCorrect} + B$, where A and B are complex S-expressions that vary across agents but evaluate, approximately, to constants. The agents come in three recognizable types. A few, which we call "cleaners," unstack several blocks from stack 1, stacking them elsewhere, and have a positive constant B. The vast majority (about 1000), which we call "stackers," have similar positive A values to each other, small or negative B, and shuffle blocks around on stacks 2 and 3, and stack several blocks on stack 1. "Closers" bid similarly to stackers but with a slightly more positive B, and say *Done*.

At the beginning of each instance, blocks are stacked randomly. Thus, stack 1 contains about $n/3$ blocks, and one of its lower blocks is incorrect. All agents bid low since *NumCorrect* is small, and a cleaner whose B is positive thus wins the auction and clears some blocks. This repeats for several auctions until the incorrect blocks are cleared. Then a stacker typically wins the next auction. Since there are hundreds of stackers, each exploring a different stacking, usually at least one succeeds in adding correct blocks. Since bids are proportional to *NumCorrect*, the stacker that most increases *NumCorrect* wins the auction. This repeats until all blocks are correctly stacked on stack 1. Then a closer wins, either because of its higher B or because all other agents act to decrease the number of blocks on stack 1 and thereby reduce *NumCorrect*. The instance ends successfully when this closer says *Done*. A schematic of this procedure is shown in Figure 3.

This evolved strategy does not solve all instances. It can fail, for example, when the next block to place is buried under 5 or 10 other blocks and no agent can find a way to make progress. The next block needed is more likely to be buried deep for higher numbers of colors, so runs with $k = 10$ or $k = 20$ colors evolve more agents, up to 3000 in some experiments, and hence run slower than runs with $k = 3$. Nonetheless, they follow a similar learning curve and have learned to solve BW instances with up to 50 blocks consistently.

Classical planning programs search a large space of possible actions. By contrast, Hayek3 learns to search only a few hundred macroactions and learns to break BW into a sequence of subgoals, long an open problem in the planning community (Korf, 1987). Note that the macroactions are tuned to the problem domain. For example, the macroaction of the cleaner involves unstacking from 1 and stacking elsewhere, and the macroaction of each stacker involves unstacking some block (or blocks) and placing it (or them) on stack 1.

Standard RL approaches, as well as classifier systems, try to recognize progress after a single action. Such an evaluation requires not only understanding *NumCorrect*, but also concepts such as "number of blocks on top of the last correct block," "next block needed," and "number of blocks on top of the next block needed." It is too complicated to hope to learn in one piece. Hayek3 instead succeeds by simultaneously learning macroactions and evaluation. Its macroactions make sufficient progress that it can be evaluated.

We also ran Hayek3 without the function *NumCorrect*. *NumCorrect* can still be computed as

$$For(And(EQ(Look(0,h), Look(1,h)), Not(EQ(Look(0,h), Empty)))).$$

Hayek3 learned to approximate *NumCorrect* as $For((EQ(Look(0,h), Look(1,h))))$ and, employing a similar strategy as above, solved problems involving goal stacks of about 50 blocks (see Figure 4).

To improve $For((EQ(Look(0,h), Look(1,h))))$ to the full *NumCorrect*, Hayek3 has to make a large jump at once. We have discovered that adding a node $R(a, b)$ to the language greatly improves the evolvability. $R(a, b)$ simply returns the subexpression a with probability $1/2$ and the subexpression b with probability $1/2$. In addition, we add mutations changing $R(a, b)$ to $R(a, a)$ or $R(b, b)$. The profitability of an S-expression containing an R node interpolates between that of the two S-expressions, with the R node simply replaced by each of its two arguments. This seems to smooth the fitness landscape. If one of the two alternatives evolves to a useful expression, the mutation allows it to be selected permanently.

With R added to the language, Hayek3 consistently succeeds in discovering the exact expression for *NumCorrect* (see Figure 5). The runs then follow a strategy identical to that with *NumCorrect* included as a primitive. Unfortunately, the runs are slower by a factor of perhaps 100 because the single built-in instruction *NumCorrect* becomes a complex *For* loop with an execution cost in the hundreds of slow instructions. Accordingly, after a week of

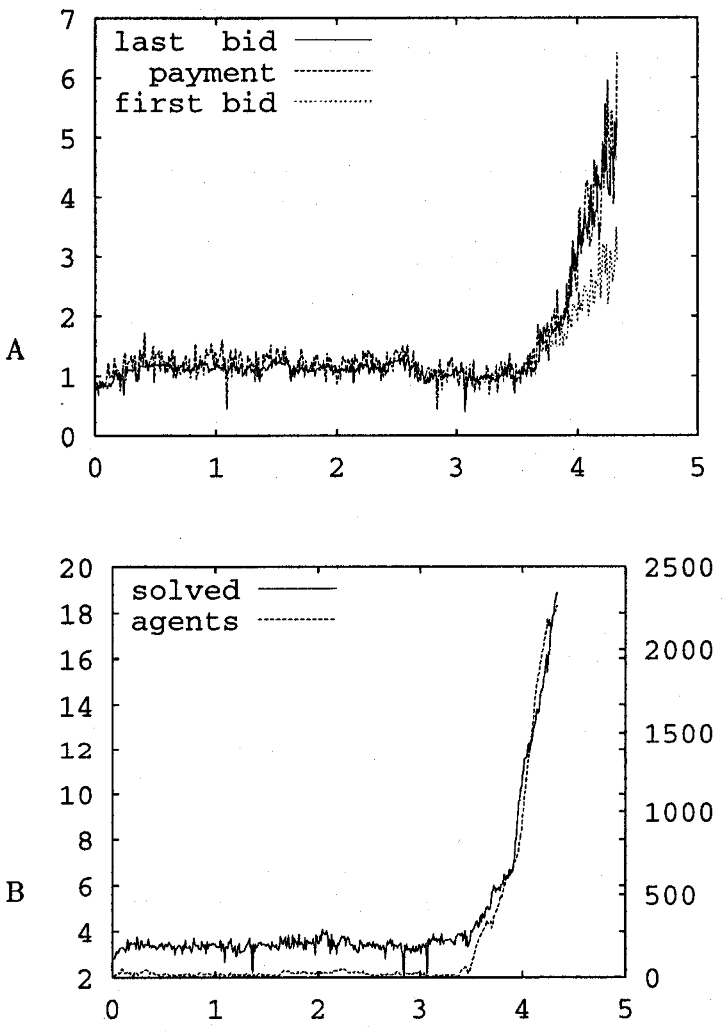


Figure 5: Data from Hayek3 runs without *NumCorrect* or *R* nodes. For further information, see the caption of Figure 2, which presents identically formatted data for a different run.

computation, the system has learned to solve only 20-block problems consistently. Nevertheless, this system is stably following a similar learning curve to that with a primitive *NumCorrect* supplied, apparently learning only a constant factor slower, which could be made up with a faster computer. By contrast, standard strategies typically incur an exponentially increasing cost for solving larger problems. The 50- and 20-block problems solved by

Hayek3 without *NumCorrect* are each several times bigger than were solved by competing methods with *NumCorrect*.

6 Metalearning

This section describes experiments with a more complex approach where new agents are created by existing agents rather than as simple mutations of existing agents. The point of this is metalearning. By assigning credit to agents that create other good agents, we hope to learn how to create good agents, and thus expedite the learning process greatly.

In this scheme there are two kinds of agents: solvers and creation agents. An agent is a creation agent if the instruction at the root of its S-expression is a modify or a create, otherwise it is a solver. Solvers behave like the agents we discussed in previous sections.

Creation agents do not bid. Instead, in each auction, all the creators that have wealth more than W_{init} (a fixed sum) are allowed to create. The creator endows its new child with an initial wealth W_{init}. Creators are compensated for creating profitable solvers, profitable creators, or agents modified into profitable agents, as follows. In each instance, all agents pay one-tenth their profit (if any) plus a small constant sum toward this compensation. For agents created de novo, this payment goes to their creator (which then recursively passes one-tenth on to its creator). For agents created by a "modify" command, the payoff is split equally between their creator and the holders of intellectual property used to create them. Intellectual property rights in an agent A are deemed shared equally by its creator and, recursively, the holder of intellectual property rights in the agent (if any) modifed to create A.

Creators that profit create more agents. Creators with no surviving sons and wealth less than W_{init} are removed. Creators are thus removed when their wealth, plus that of all surviving descendants (which they are viewed as "owning"), is less than W_{init} so that creators, like solvers, are removed whenever their net impact has been negative. Thus, there is evolutionary pressure toward creators that are effective at creating profitable progeny.

The system is initiated with a single creation agent, called Seed, with zero capital. Seed is written (by us) to create children that consist of random code. These agents can create other agents if their code so indicates. As in the simpler scheme described in section 2, W_{init} is initiated as zero but raised when an agent first earns money from the world. Creators endow their child with W_{init} capital.

In addition to the instruction set described in section 4, Creators employ one "wildcard" $*$ of each type; four binding symbols $1,$2,$3,$4 of each type, and two functions, *create* and *modify*.

Wildcards are treated as symbols of appropriate type when they appear in creators. When a wildcard (or an unbound binding symbol) appears in an S-expression of a solver, however, it is expanded into a random subtree.

This is done similarly to the general approach in strongly typed genetic programming (Montana, 1994), by growing from the top down. At each node, one randomly chooses an expression of the correct type. Unless this expression is a constant (i.e., takes zero arguments), we must iterate to the next level, choosing random expressions for the children (subexpressions). At each step, we multiplicatively decrease the probability of generating a nonterminal symbol so that the expansion terminates with a small tree.[5]

Create takes one argument. *Create* creates a new agent whose expression is simply identical to its argument.

The *modify* instruction has two arguments. *Modify* chooses a random agent in the population[6] and attempts to match the expression in its first argument with a subexpression of either the randomly chosen agent's void or integer expressions. If it fails to find a match, no agent is created. If it succeeds, it creates a new agent identical to the matched one except that *modify*'s second argument is substituted in place of the matched expression. (See Figure 6.) A match must be exact, except that the arguments of the *modify* expression may contain *binding* symbols and wildcards. Binding symbols and wildcards can match arbitrary expressions of the appropriate type. The difference between them is that such binding symbols in the first argument are bound when it matches, and if the same binding symbol also occurs in the second argument, the binding symbol is replaced by the expression it was bound to. Wildcards match independently and do not bind. Every tenth new agent, however created, is further mutated by having a randomly chosen node or nodes replaced by random trees. This helps to ensure the system will not get stuck in a configuration from which it cannot further evolve.

Note that a modify operation may use code from a solver in the population to create another creator. This creator may then use code from a second solver in creating another solver. In this way, it is possible for code fragments from different agents to be combined.

When we ran on BW with creation agents, the system learned a collection of solvers employing a similar strategy to that described in section 5. The behavior of creation agents was less clear. Some evolved to modify se-

[5] Let $P_{\text{expand}}(d)$ be the probability of choosing a nonterminal at depth d. We set $P_{\text{expand}}(d+1) = P_{\text{expand}}(d) * C$. We imposed limit conditions that $P_{\text{expand}}(\text{depth} = 0) = \text{initial} - \text{expansion}$ and $P_{\text{expand}}(\text{average} - \text{depth}) = 0.5$. We chose (ad hoc, without experimentation) an average depth of 3 and initial expansion of 0.9. The constant C is then computed as: $C = \exp((\log(0.5) - \log(\text{initial expansion}))/\text{average depth})$.

[6] Note that there is no "selection" operator per se. *Modify* chooses a random agent in the population. Selection occurs because unprofitable agents are removed from the population. Modifiers can only sense which agent they modify in that they must match a pattern. We have not yet explored allowing modifiers other sensations and options, such as choosing to modify a wealthy agent or even one recently active.

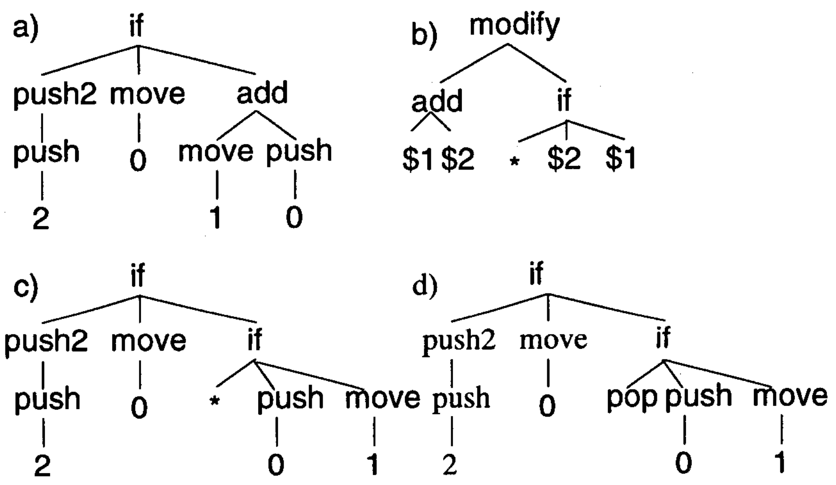


Figure 6: (a) An integer expression. This is equivalent to *if[(look(0,0)=look(1,0)) and(look(3,0)=E)] then 0 else* −3. It returns 0 if the bottom block of the 0 stack is the same color as the bottom block of the 1 stack and there are no blocks on the 3 stack; else it returns −3. (b) The void expression of a creator, with *modify* at its root. This can match with the expression of *a* by matching the *and* expressions, binding $1 to *eq(look(0,0),look(1,0))*, and binding $2 to *eq(E,look(3,0))*. Substituting the second argument of the modify (the subtree rooted by *or*) results in *c*. Finally the wildcard $*$ is expanded into a random tree, resulting in the expression of *d*.

quences of grabs and drops in ways that create useful stackers or cleaners. Others were not transparent to us.

We ran control experiments to discover whether we were in fact able to learn effective ways to create. We first compared a run with the creation agents to a control in which creation agents attempted to create, but we discarded the agent they created in favor of a mutated agent. Hayek3 with creation agents performed 30% to 50% better after a week of execution than this control. This indicates that the creation agents were learning useful techniques. However, the pattern matching in the creation agents was expensive in time, and the system with creation agents performed no better than the less complicated system reported in previous sections.

The conclusion of our experience with metalearning is thus mixed. With our current techniques, metalearning is probably not a practical alternative. It remains plausible that with a more powerful creation language and for yet more complex problem domains, metalearning may reemerge as a useful technique.

7 Rubik's Cube

We tried to learn a Hayek capable of unscrambling a randomly scrambled Rubik's cube. This section will assume some familiarity with Rubik's cube. For the uninitiated, a "cubie" is one of the 26 physical little cubes out of which a Rubik's cube is constructed. We call a "square" one face of a cubie.

Rubik's cube is a complex problem. We experimented with a variety of instruction sets, including several versions of *numcorrect* functions, several presentation schemes of increasingly harder instances, and several reward schemes. All of the versions we experimented with used creation and modify operators, as discussed in the previous section.

We used instruction sets with several types. For example, one set of runs used six types: boolean, integer, coordinate, face, square, and void (i.e., actions). To describe a particular square, you could write $s(i, x, y)$ where i is a coordinate of type "face" specifying a face of the cube (e.g., the face with red center), and x and y are coordinates taking values 0, 1, or 2, specifying the location on the face. Having specified two squares $s1$ and $s2$, it would be possible to compare their color with the boolean-valued function $EQCs(s1, s2)$, which returns true if and only if they have the same color. We supplied an instruction *Sum* taking a boolean argument, which summed any constructed boolean function over all cube faces. Hayek evolved evaluation functions, which used sums to estimate the number of correct cubies. We realized, however, that it was impossible for Hayek to express the physical concept of a cubie in this language and so also experimented with a language based around cubies, in which we supplied a function *NumCorrect* describing the number of correct cubies.

We also worked with two different presentation schemes. In *Presentation Scheme 1,* we initially presented cubes scrambled with one rotation, and as Hayek learned to master cubes at a given level of scrambling, presented cubes scrambled with increasing numbers of rotations. We then gave reward only for completely unscrambling the cube. In *Presentation Scheme 2,* we presented a completely scrambled cube—a cube scrambled with 100 random rotations—and presented Hayek with reward proportional to its partial progress at the time it said "done." For this purpose we measured partial progress according to three different metrics (in different runs): (1) the number of cubies it got right, (2) the number of cubies it got right according to a fixed sequence (i.e., we numbered the cubies starting with the top face and gave Hayek reward equal to one less than the first cubie in the sequence that was wrong), and (3) the number of cube faces it got right (maximum 54).

We also worked with two different cube models. *Cube model A* allowed three actions: F– a one-quarter move on the front face, and X and Y rotations of the whole cube. *Cube model B* held the x-, y- and z-axes of the cube fixed, but allowed a one-quarter move on each of six faces (L,R,F,B,T,D).

Some representative results are as follows. In a run using Presentation Scheme 1, with Hayek given an instruction *NumCorrect* that we set equal to the total number of cubie faces correct relative to the center cubie face, Hayek learned after a few days to descramble cubes scrambled with 20 quarter-moves 80% of the time and cubes scrambled with 30 quarter-moves 40% of the time.

In another experiment, using Presentation Scheme 2, with reward equal to the total number of cubie faces it left correct, Hayek learned to improve scrambled positions by about 20 cubie faces (the average number of correct cubie faces at the end minus the number of correct cubie faces at the beginning was about 20).

In runs with Presentation Scheme 2, with Hayek given a reward for partial success defined as the number of correct cubies in a particular (x, y, z) order, stopping at the first incorrect cubie (and not counting the center cubies, which are always correct by definition), Hayek learned to solve about 10 cubies or the whole first slice and 2 more cubies of the middle slice. This is further than many humans get.

For each of the schemes we tried, Hayek made significant progress, but after a few days of crunching, it would plateau and cease further improvement. Actually, this describes as well the progress of many humans, because as the cube becomes more solved, it becomes increasingly difficult to find operators that will make progress without destroying the progress already achieved. Such operators become increasingly long chunks of code, and there are no evident incremental properties of the chunks, so it becomes exponentially hard to find them unless some deep understanding can be achieved.

Hayek's strategy in these runs was fundamentally similar to its strategy in BW: it produced a collection of 300 to 700 agents that took various clever sequences of actions and bid an estimate of the value of the state they reached. Typical instances involved a dozen or so auctions, the winner in each making partial progress. Hayek made progress in all of the different schemes, and it was not obvious that any scheme was particularly preferable to the others.

It is unclear whether Hayek, at least using the S-expression representation here, can exploit the compact structure of the problem domain in more powerful ways. It is worth noting, for example, that within the S-expression languages used here, it does not seem that high-level concepts such as "cubies" can be represented (never mind learned) to the extent that we do not simply insert them by hand. However, it also seems likely there is fundamentally less structure to be exploited in Rubik's cube than in BW. The smallest program a human could write that will solve Rubik's cube is much much larger than the smallest human-crafted program to solve BW, while at the same time the state-space in BW is huge (in fact infinite) and the state-space in Rubik is merely large. Indeed Rubik can be solved by brute search (Korf, 1997).

8 Rush Hour

This section describes experiments applying Hayek to the commercially available game Rush Hour. The game comes with 40 different puzzle instances. Figure 7 shows instance 35. Each instance consists of several cars placed on a 6×6 grid. Cars are of width 1 and come in two different lengths, 2 or 3. A car can only be moved forward or backward, not turned. The goal of the game is to move the cross-marked car out the exit. The game was honored for excellence by Mensa, and humans find the problems challenging. Moreover the game is provably hard. Flake and Baum (1999) show that the generalization to an $n \times n$ grid is P-space complete, equivalent to a reversable Turing machine, and that solution may require a number of moves exponential in n.

We trained Hayek, and also a genetic program, by presenting problems of gradually increasing difficulty, including easy training problems generated from the original 40 by randomly removing cars. Rush Hour presents new challenges, particularly in the selection of a representation language for the agents, since agents need flexibility in specifying actions and must be able naturally to express relations such as "the car that is blocking this car." The version of the program discussed here used integer-type S-expressions and creation operators. We call this Hayek3.1

Agents were limited to expressions of up to 20,000 nodes. The language consists of constants (0,1,2), arithmetic functions (add, subtract, multiply), if-then-else, and some additional functions, as follows. There is a pointer that is initiated pointing to the marked car. The function *PUSH* moves the car pointer to either the car blocking forward motion of the pointed car or the car blocking its backward motion, depending on whether its argument is zero or not. (If there is no such blocking car—motion in

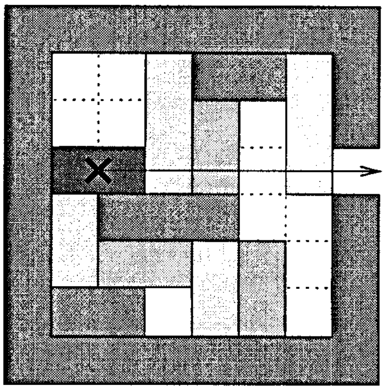


Figure 7: Rush hour problem 35.

Table 1: Ordering of the Tasks by Random Moves.

Order	**1**	**2**	3	**4**	5	**6**	**7**	8
Task	**9**	**13**	28	**14**	23	**10**	**7**	22
Moves	**1394**	**3181**	5909	**6063**	6069	**6854**	**7487**	8931
Order	**9**	**10**	**11**	**12**	**13**	14	15	**16**
Task	**2**	**12**	**4**	**6**	**20**	27	16	**1**
Moves	**9919**	**10,204**	**11,718**	**13,134**	**13,532**	14,281	14,451	**20,916**
Order	**17**	18	19	20	21	**22**	**23**	24
Task	**21**	25	30	17	32	**3**	**5**	38
Moves	**21,968**	22,348	23,255	24,336	25,813	**26,077**	**27,572**	27,828
Order	25	26	27	28	29	**30**	31	32
Task	37	11	15	19	29	**8**	18	26
Moves	28,127	36,892	41,421	57,901	58,439	**70,804**	79,533	87,473
Order	33	34	35	36	37	38	39	40
Task	24	40	39	34	31	33	36	35
Moves	100,016	111,243	123,884	157,495	160,507	171,687	184,341	209,923

Notes: *Task* is the number supplied by the manufacturer. *Moves* is the number of moves taken to solve the problem using random search. Problems printed in boldface were solved by Hayek3.1.

that direction is blocked only by a wall or, alternatively, it is possible to drive off the board—then *PUSH* fails and takes no action.) As a side effect, the previously pointed car is pushed onto a car stack. Cars can be popped back, giving our language a weak backtracking ability. Function *PUSH2* behaves like *PUSH*, except that it moves the car pointer to the second blocking car (the car that would block next if the first blocking car were removed). Function *MOVE* moves the pointed car forward or backward. Each of these functions returns 1 if they succeed and 0 if they fail.

Creators employ additional statements: *MODIFY*, a "wildcard" $*$, and four binding symbols \$1,\$2,\$3,\$4.

To get a rough estimate of the difficulty of the problems, we applied random search to them. Table 1 shows the problems ordered by the number of moves used to solve them by a random search algorithm. We call the ranking according to Table 1 "order no." and the number supplied by the manufacturer "task no."

For each task we allowed a total number of car moves bounded by $100 +$ *#random-moves*/100 where *#random-moves* was the average number of moves taken to solve that particular instance random search. In each instance we allowed $10 +$ *#random-moves*/1000 auctions. If the move limit or the auction limit is exceeded, the instance ends with no reward.

Hayek3.1 was trained using random problems drawn from a distribution crafted to present easier problems first, as follows. We first select a problem

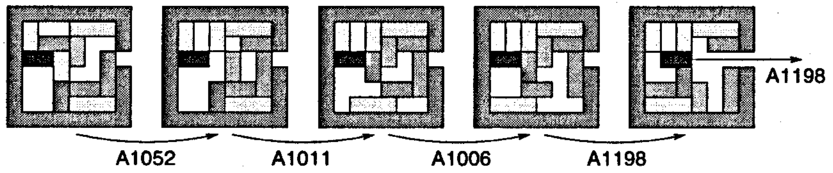


Figure 8: Sequence of positions solving problem 20.

randomly and uniformly from the 40 supplied with the commercial game. The instance is then subsampled by removing cars as follows. For each problem number, we maintain a count of the fraction of instances of that problem number solved out of the last 100 times this instance was presented.[7] We remove a number of cars so that the fraction of cars remaining, called p, is equal to the fraction of the last 100 problems of that number solved (up to round-off, since we must present an integer number of cars). This allows Hayek3.1 to learn from simpler examples, and as it learns on each example, we gradually ramp up p until we are presenting the full example. The reward R given for solving an instance was $R = (\text{orderno.})e^{(4.06p-4.06)}$. Thus more reward was given for harder instances, and reward is exponentially increased for solving instances with no subsampling. The coefficients in the exponent were determined by fitting $e^{(ap+b)}$ to be equal to 0.01 for $p = 0$ and 1.0 for $p = 1$.

After training, Hayek contains a collection of hundreds of agents that collaborate to solve some of the problems. Figure 8 shows a sequence of agents acting to solve task no. 20 (a.k.a. order no. 13). Table 2 shows the sequence of bids and number of actions taken. Notice that the agents recognize as they close on a solution and bid higher. Note that the final bid is quite accurate (although in this case a slight overbid). (This agent was profiting on other instances, but lost money on this one.) In instances where the problem is ultimately not solved, bids stay low, as agents realize they are not close to solution. The ability of agents to recognize distance from the solution is critical to Hayek's performance. In each of a series of auctions, it chooses the agent that in its own estimation makes the most progress, eventually converging on a solution. This breaks down a lengthy search into a sequence of operations, effectively achieving subgoals.

Figures 9 and 10 show the time evolution of statistics for this typical Hayek3.1 run. The data points are a sample every 10,000 instances of the average over 100 instances. Figure 9 shows the winning bid in the first

[7] Actually, the bins record data from runs only where a new agent did not win the bidding and immediately die. Such failures are deemed irrelevant for the purpose of determining the distribution, as this dead agent is no longer part of Hayek3.1 and thus does not affect future performance.

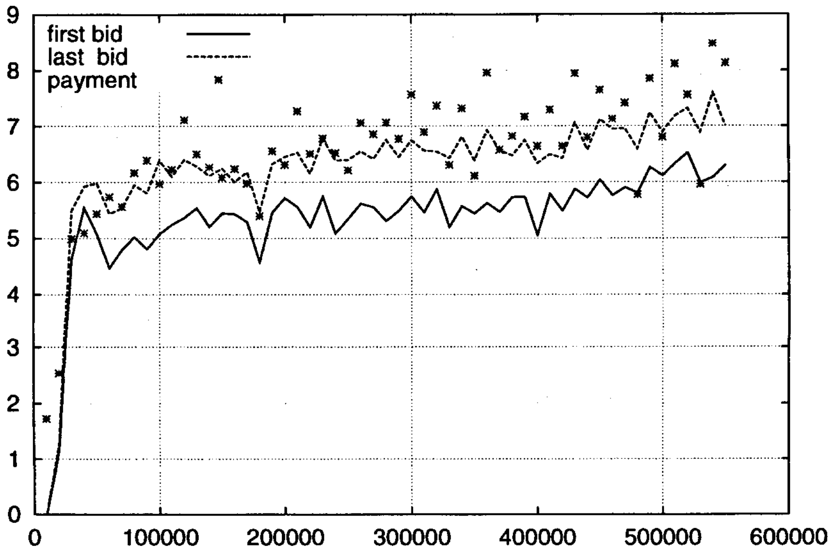


Figure 9: Bid as discriminator between positions.

auction, the winning bid in the last auction, and the final payoff and the the last bid accurately estimates payoff and the first bid is lower.

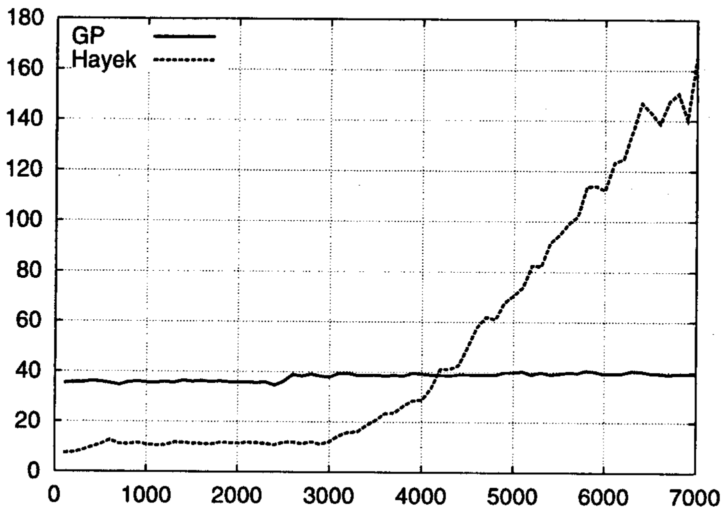
Figure 10a shows the reward earned and Figure 10b the linearly weighted sum of fully solved problems

In each run, Hayek has learned to solve a number of unsubsampled problems. A typical run creates a collection of agents that is solving about five unsubsampled problems at a time. The best run solved nine unsubsampled problems simultaneously. In one run or another, Hayek has solved problems 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 12, 13, 14, 20, and 21 according to task number (i.e., the manufacturer's ordering). When Hayek solves an unsubsampled problem, it generally involves a collaboration of two to five agents. Subsampled problems are much easier, of course, and are often solved by a single agent.

Table 2: Auctions for a Solution of Problem 20.

Auction #	Winning Agent	Winning Bid	Reward	Actions
0	a1052	6.749	0	14
1	a1011	7.369	0	15
2	a1006	9.208	0	20
3	a1198	13.741	0	22
4	a1198	13.853	13	35

A



B

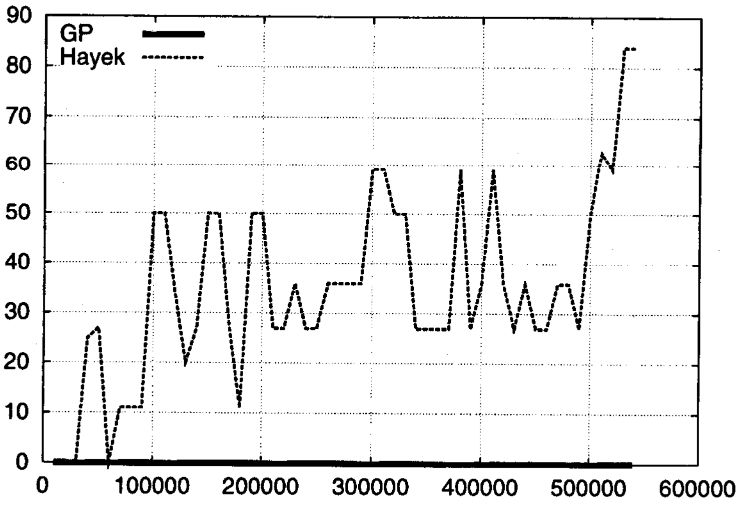


Figure 10: (A) Reward per instance. Horizontal axis times 100 is instances presented to Hayek. Horizontal axis times 10 is GP generations. The first point shown is after 100 GP generations. GP at generation 1 scores about 2 on this scale and learns rapidly at first. Each GP generation involves presentation of 100 instances of each type (4000 instances total) to each program in the population. The scales were chosen to agree roughly in wall clock time. GP shows reward per instance of the best program in the population. (B) Score as linearly weighted sum of fully solved instances.

We have tried several approaches to getting a genetic program to solve these problems, varying the instance presentation scheme and other parameters. In each case, our GP used the same instruction set as Hayek3.1. The runs shown used a population size of 100 and an instance presentation scheme as similar to that Hayek saw as possible.[8] As the graphs show, although GP learns to solve subsampled problems, and thus to earn reward, it has never learned to solve any of the original problem set. It reaches a plateau where it is solving only subsampled problems after fewer than 100 generations. Even when trained solely on Problem 1, we have not been able to learn a program solving the full problem. Possibly some better scheme for ramping up problem difficulty would improve performance.

9 Discussion

We have created an artificial economy that assigns credit to useful agents. When we enforce property rights to everything and conservation of money, and when agents are rational in the sense that they accept transactions only on which they profit, new agents can enter if and only if they improve system performance. Agents are initiated far from rational, but we hoped exploitation of irrational agents would lead to the evolution of rational agents and a smoothly functioning system that divides and conquers hard problems. Conversely, we argued that when property rights or conservation of money are not enforced, agents will be able to profit at the expense of the system. Under those conditions, a local optimum of the system will not be evolutionarily stable, and there is no reason to expect evolution to produce a useful system.

We have now performed experiments on BW using three different representation languages for the agents (one in Baum, 1996; one here; and a very different one reported in Baum & Durdanovic, 2000, which adopts a radically different solution strategy), and also with and without creation agents. We have performed experiments on Rubik's cube using two different representation languages (here and in Baum & Durdanovic, 2000). We have reported experiments here on Rush Hour. In each of these cases, we have stably evolved systems with agents collaborating to solve hard problems. In the BW examples in particular, we are evolving systems with stable sequences of agents hundreds long, receiving reward only at the end. This contrasts with classifier systems, which are rarely able to achieve stable chains more than a few agents long (Wilson & Goldberg, 1989). Our dynamics have also been largely robust to parameter variations.

[8] For a GP, there is a question as to how to compute the fraction of problems of type i solved in recent generations. The runs shown used the average over the last 10 generations of the fraction of problems of that type solved by the upper half scoring members of the population, which worked as well as any of the other approaches we tried.

All of this dynamic stability vanishes when conservation of money is broken. If the tax rate is raised too high, so that money leaks out, the system dies. If, on the other hand, money is nonconserved in a positive fashion, for example by introducing new agents with externally supplied money (Baum, 1996), or in other ways as has happened more than once due to bugs in the code, the system immediately learns to exploit the ability to create money, and cooperation in solving problems from the world disappears. If property rights are broken, for example by choosing the winner of the auction probabilistically, as sometimes suggested in the classifier system literature (Wilson, 1994), again cooperation immediately breaks and the system can no longer function.

We believe that all of this provides powerful evidence for our economic picture. As we have discussed elsewhere (Baum, 1998), we believe that many of the dynamic phenomena in natural complex systems such as ecologies, as well as many problems with other learning programs, can be profitably viewed as manifestations of violations of property rights or conservation of money. We also propose that Hayek is of interest as an artificial life. In particular, our experiments (particularly with metalearning) answer the open question (Thearling & Ray, 1997) of how to get a collection of self-reproducing programs to cooperate like a multicellular being in solving problems of interacting with an external world.

We further conjecture that progress in reinforcement learning in ultra-complex environments requires learning to exploit the compact structure of the problem. This means that learning in complex environments will essentially involve automatic programming. We have learned to exploit the structure from small problems and generalized the knowledge to produce programs solving large problems.

The problem with automatic programming methods, including ours, is that the space of programs is enormous and has a very bumpy fitness landscape, making it inherently difficult to search. Our proposal to divide and conquer automatically using an economy helps, but the problem remains. This raises the key question of what language will be evolvable.

We have made progress using S-expressions, which have previously been used in genetic programming (Koza, 1992) precisely because, intuitively, their structure makes mutations more likely to create semantically meaningful expressions. Typing (Montana, 1994) has also been critical here in cutting down the search space.

We found that using a certain generic random node smoothed the fitness landscape in BW and increased the evolvability of complex expressions. We found that Hayek was able to solve problems so long as the longest chunk of code it needed as a component was not too long to find. Hayek has been able to create very complex programs made of relatively small chunks of code.

Nonetheless, our ability to evolve complex code has been limited. For example, we were not able to progress on Rubik's cube because after a point, the next chunk of code needed was too long to evolve. Metalearning was proposed as a possible solution to this problem, and our economy was able to support some degree of metalearning, but to date we have not been able to extend the overall abilities of Hayek using metalearning.

A related language question is the power of the language. We believe the S-expression language used here is not computationally universal, and accordingly Hayek has not been able to evolve a program solving, for example, arbitrary BW problems. By contrast, we report elsewhere (Baum & Durdanovic, 2000) a Hayek employing postproduction system language, which is computationally universal and succeeds in evolving a simple universal solver for BW. However, this system also founders in the search problem for Rubik's cube.

More generally, for truly complex problems, one might imagine somehow evolving an information economy, with agents buying and selling information from one another. We have not solved the problem of how to make this work. Also, it would be interesting to have a single, universal language able to address many problem domains and see if it is possible to transfer knowledge from one domain (e.g., BW) to another (e.g., Rubik's cube.)

Appendix A: Other Methods Applied to Blocks World

For comparison, we tested TD(λ) using a neural net evaluator. (For a description of the method, see Sutton & Barto, 1998, or Tesauro, 1995.) We tested several values of λ, several discount rates,[9] and several neural net topologies. We report results on only three color problems. As will be evident, 10 color problems would have further stressed the encoding requiring much larger nets if a unary coding were desired. We also report results only on an easier version of our BW where, rather than requiring the learner to recognize when it is done, we externally supply that information. We considered a single move to be a grab-and-drop combination, which improved performance.

Neural nets have a fixed input dimension, so it is unclear how to encode a problem like BW with varying size. An evaluator that does not see the whole space, however, will cause aliasing, turning a Markov problem into a non-Markovian one causing great difficulties. (For a discussion of such problems, see Whitehead & Ballard, 1991.) After some experimentation, we used neural nets with inputs for each of the top five blocks in each stack. We tried both unary and analog encoding for the block color. In unary encoding, each block was represented by three inputs to the net, with one having value

[9] A discount rate of 1 (no discounting), as used by Tesauro in his backgammon program (Tesauro, 1995), is the most logical in an episodic presentation scheme like ours. However, empirically discounted rates of about 0.9 stabilized learning in some runs.

1 and the others value 0 to indicate which color the block had and all value 0 indicating no such block (because the stack was not five blocks high). In analog encoding, the situation was indicated by a single integer input value from 0 to 3. We ran experiments both with and without *NumCorrect* supplied as an input.

We presented increasingly larger instances as the system learned, as in our experiments with Hayek3. However, to keep learning stable, we found we had to slow the increase in instance size greatly. We presented instances chosen uniformly over the sizes from 1 up to $l + 1$, where l was the largest size being solved 80% of the time.

In experiments without *NumCorrect*, using a 61-20-1 feedforward net, with 60 inputs giving unary encoding of the colors of the top five blocks in each of four stacks and one "on" input to realize a threshold, we were able to learn stably and solve problems with $n = 4$ after several days of learning. No other system without *NumCorrect* supplied did better.

In experiments with *NumCorrect* supplied as an input, the system learned fairly rapidly to solve problems up to about $n = 8$. Once it got up to this size, its learning would destabilize and it would crash to much worse performance. Then it would rapidly learn again. The system was never able to learn beyond $n = 8$.

These results are perhaps disappointing but not surprising. While neural nets are good at associative memory and approximating some functions, there is no evidence to indicate that they are suited to solving symbolic search problems or inducing programs as necessary for solving BW. Tesauro's success with backgammon, for example, is attributable to the fact that a linear function provides a reasonable evaluator for backgammon (Tesauro, 1995).

We also extensively tested genetic programming. Results are reported in Baum and Durdanovic (1999). Since this publication, we have done further extensive tests using population sizes up to 1000, various selection methods, and various combinations of instruction sets, including *NumCorrect*. As discussed in Baum and Durdanovic (1999), our results were of some independent interest as bearing on the hotly debated question within the genetic programming community, whether genetic programming benefits from crossover, or would be just as effective using the "headless chicken macromutation" where, rather than swapping random subtrees between trees in the population, random subtrees are simply replaced with new random subtrees. A recent textbook (Banzhaf, Nordin, Keller, & Francone, 1998) reports several studies showing these methods essentially equivalent and none showing a big advantage for crossover. BW is thus in some sense an unusually suitable problem for genetic programming, as our BW results represent the only published comparison of which we are aware showing crossover much superior to headless chicken. Nonetheless, genetic programming had only limited success on this problem. Run with a learner having to say *done*, our best genetic programming run solved 70%

of size 3 and 30% of size 4. Run with *done* externally supplied, GP produced at best S-expressions capable of solving 80% of size 4 problems and 30% of size 5. These results are arguably directly comparable to our Hayek results as both GP and Hayek used the same language for their S-expressions.

We also extensively tested hill-climbing approaches using an assembler-like language. Results, reported in Baum and Durdanovic (1999), were that only about four block instances could be solved.

We also tested a system that searched for an evaluator as the maximum of a set of S-expressions in Hayek's language. This system differed from Hayek3 in that we did not learn agents capable of sequences of actions. Rather, as in TD(λ), we considered all pairs of a grab followed by a drop, evaluated the resulting state, and chose the move leading to the highest evaluated state. We modified our evaluator better to estimate the value of the next state reached. There is no standard method for training complex functional expressions as evaluators. We tried to approximate the evaluation from below, by using the maximum of a number of S-expressions, and removing an S-expression from our population when it overestimated the evaluation, replacing it with another S-expression, generated either randomly or as a mutation of an existing S-expression. This method was also ineffective, solving only single-digit BW problems. We believe that it is critical to learn macroactions, as it will be very difficult to learn any evaluator capable of showing progress after only a single grab-drop pair.

Baum (1999) discusses several other ineffective approaches we experimented with.

Appendix B: Pseudocode for Hayek3

```
hayek() {
 for ( ;; ) {
  task.new() // create an instance
  solve()   // try to solve it
  payment() // perform money transactions
  tax()     // collect taxes
 }
}

solve() {
  record.clean()
  for ( Agent A = Solvers.begin(); A < Solvers.end(); A++ ) {
   A.old'wealth = A.wealth
  }
  while ( task.state() == RUN ) {
   create() // creates new agents
   auction() // performs an auction
  }
```

```
}

create() {
 for ( Agent A = Creators.begin(); A < Creators.end(); A++ )
  if ( A.wealth > InitialWealth() ) A.create()
}

auction() {
 best'bid  = -1
 best'state = task
 best'agent = nil
 for ( Agent A = Solvers.begin(); A < Solvers.end(); A++ ) {
  if ( A.wealth > best'bid ) {
   A.move( task, state );
   bid = min( A.eval( state ), A.wealth );
   if ( bid > best'bid ) {
     best'bid  = bid;
     best'state = state;
     best'agent = A;
   }
  }
 }
 if ( best'agent != nil ) { // the winner takes the world into
                            // a new state
  task = best'state;
  update'agent'wealth( best'agent, best'bid )
  record.add( best'agent, best'bid, task.reward() )
 }
}

update'agent'wealth( agent, bid ) {
 best'agent.wealth -= best'bid
 if ( record.last = nil ) {
   pay'to'world += best'bid
 else {
   record.last.wealth += best'bid
 }
}

payment() {
 for ( AuctionRecord R = record.begin() ; R < record.end(); R++ ) {
   delta = R.agent.wealth - R.agent.old'welth
   if ( delta > 0 ) {
    R.agent.wealth += 0.9 * delta
    copyright'payment( A, 0.1 * delta )
   } else {
    R.agent.wealth += delta
    }
  }
}

copyright'payment( Agent A, Money M ) {
  if ( A.father == nil ) {
    A.wealth += M
  } else {
    A.father += 0.5 * M
    copyright'payment( A.father, 0.5 * M )
  }
}

tax() {
  for ( Agent A = Agents.begin(); A < Agents.end(); A++ ) {
    A.wealth -= A.executed'instructions * 1E-6
    A.executed'instructions = 0
    if ( A.wealth < A.initial'wealth AND A.no'living'sons == 0 ) {
      Agents.remove( A )
    }
  }
}
```

Appendix C: Parameters

There are a number of constants in this system chosen in an ad hoc way, such as the one-tenth profit sharing, the resource-tax, W_{init}, and, in versions with creation agents, the equal split between creators and IP holders. These parameters were not subject to evolution within the system. We attempted only a small amount of empirical tuning. Within reasonable bands in each parameter, the behavior of the system was not found empirically to be qualitatively sensitive to these choices, and we did not establish quantitative differences. Runs of the system require a day or more to learn, and there is considerable random variation from run to run, so empirical tuning is quite difficult unless a qualitatively different behavior can be observed. We discuss our choices of these values in turn.

Since creators are intuitively considered to own their children, in principle the creator might be allowed to set the profit-sharing fraction as it pleases, say by writing appropriate code in the child, and indeed this was done in an earlier version (Baum & Durdanovic, 1999). For the work reported here, just for simplicity, we fixed the profit-sharing fraction at one-tenth. We also experimented with one-half. Our subjective impression was that passing one-half of the profit passed too much money to the creation agents, causing an increase in creation, which slowed the system. However, the increase, if any, was not huge, and we have no statistically significant claim that one-tenth is better than one-half.

W_{init} was initially set equal to zero. Once there was money in the system because instances were solved, W_{init} was simply set equal to the maximum payoff possible in the run. Thus, if we were going to run up to a maximum instance size of 200, we set W_{init} to 200. This ensured that the created agent would have enough money to make any possible rational bid. If the newly created agent's money ever went below its initial capital, it was removed, with its remaining money transferred back to its creator. Thus a creator that invested 200 in a child would not lose the full 200 if the child proved to be unprofitable. We did not experiment with any other choices.

We experimented with several values of the resource tax. Resource tax above 10^{-4} per instruction sucked too much money out of the system, which then could not get started. We did not establish a difference between resource taxes of 10^{-5}, 10^{-6}, and 10^{-7}, although subjectively runs of 10^{-7} ran slower. Our best runs used simply 10^{-6}.

The runs here with creation agents used an equal split of the passed profit between creators and IP holders, but we also experimented with not using IP at all.[10] The equal split of passed profit with IP holders subjectively worked marginally better, but again we do not claim to have established a statistically significant difference.

A more principled approach to all of these choices is a subject for future work. Fixing these parameters by fiat has the feel of fixing prices, which typically can cause inefficiencies in economies.

References

Bacchus, F., & Kabanza, F. (1996). Using temporal logic to control search in a forward chaining planner. In M. M. Ghallab & A. Milani (Eds.), *New directions in planning* (pp. 141–153). Hillsdale NJ: IOS Press.

Banzhaf, W., Nordin, P., Keller, R. E., & Francone, F. D. (1998). *Genetic programming: An introduction*. San Mateo, CA: Morgan Kaufmann.

Baum, E. B. (1996). Toward a model of mind as a laissez-faire economy of idiots, extended abstract. In L. Saitta (Ed.), *Proc. 13th ICML '96*. San Mateo, CA: Morgan Kaufmann.

Baum, E. B. (1998). Manifesto for an evolutionary economics of intelligence. In C. M. Bishop (Ed.), *Neural networks and machine learning*. Berlin: Springer-Verlag.

Baum, E. B. (1999). Toward a model of intelligence as an economy of agents. *Machine Learning*, *35*(2), 155–185.

[10] Although popular opinion seems to hold that enforcing intellectual property rights is important to progress in real economies, the scientific literature on this subject is wide open (cf. Breyer, 1976; Malchlup, 1958).

Baum, E. B., & Durdanovic, I. (1999). Toward code evolution by artificial economies. In L. F. Landweber & E. Winfree (Eds.), *Evolution as computation*. Berlin: Springer-Verlag.

Baum, E., & Durdanovic, I. (2000). *An evolutionary post production system*. Submitted. Available online at: http://www.neci.nj.nec.com:80/homepages/eric/eric.html.

Birk, A., & Paul, W. J. (1994). Schemas and genetic programming. In *Conference on Integration of Elementary Functions into Complex Behavior*. Bielefeld.

Blum, A., & Furst, M. (1997). Fast planning through planning graph analysis. *Artificial Intelligence, 90*, 281.

Breyer, S. (1976). The uneasy case for copyright: A study of copyright in books, photocopies, and computer programs. *Harvard Law Review, 284*, 281.

Dzeroski, S., Blockeel, H., & De Raedt, L. (1998). Relational reinforcement learning. In J. Shavlik (Ed.), *Proc. 12th ICML*. San Mateo, CA: Morgan Kaufmann.

Flake, G., & Baum, E. B. (1999). *Rush hour is pspace-complete*. Submitted. Available online at: http://www.neci.nj.nec.com/homepages/flake/.

Forrest, S. (1985). Implementing semantic network structures using the classifier-system. In *Proc. First International Conference on Genetic Algorithms* (pp. 188–196). Hillsdale, NJ: Erlbaum.

Hardin, G. (1968). The tragedy of the commons. *Science, 162*, 1243–1248.

Holland, J. H. (1986). Escaping brittleness: The possibilities of general purpose learning algorithms applied to parallel rule-based systems. In R. S. Michalski, J. G. Carbonell, & T. M. Mitchell (Eds.), *Machine learning* (Vol. 2). San Mateo, CA: Morgan Kaufmann.

Koehler, J. (1998). Solving complex planning tasks through extraction of subproblems. In J. Allen (Ed.), *Proceedings of the 4th International Conference on Artificial Intelligence Planning Systems* (pp. 62–69). Menlo Park: AAAI Press.

Korf, R. E. (1987). Planning as search: A quantitative approach. *AIJ, 33*, 65.

Korf, R. E. (1997). Finding optimal solutions to Rubik's cube using pattern databases. In *Proceedings of the National Conference on Artificial Intelligence* (pp. 700–705).

Koza, J. (1992). *Genetic programming*. Cambridge, MA: MIT Press.

Lenat, D. B. (1983). EURISKO: A program that learns new heuristics and domain concepts, the nature of heuristics III: Program design and results. *Artificial Intelligence, 21*(1,2), 61–98.

Lenski, R., Ofria, C., Collier, T., & Adami, C. (1999). Genome complexity, robustness and genetic interactions in digital organisms. *Nature, 400*, 661–664.

Lettau, M., & Uhlig, H. (1999). Rules of thumb versus dynamic programming. *American Economic Review, 89*, 148–174.

Machlup, F. (1958). *An economic review of the patent system*. Study No. 15, Subcommittee on Patents, Trademarks and Copyrights, Senate Committee on the Juciciary, 85th Cong., 2d Sess.

Miller, M. S., & Drexler, K. E. (1988). Markets and computation: Agoric open systems. In B. A. Huberman (Ed.), *The ecology of computation* (p. 133). Amsterdam: North-Holland.

Montana, D. J. (1994). Strongly typed genetic programming. *Evolutionary Computation, 3*(2), 199–230.

Ray, T. S. (1991). An approach to the synthesis of life. In *Artificial life II* (Vol. 11). Redwood City, CA: Addison-Wesley.

Russell, S. & Norvig, P. (1995). *Artificial intelligence: A modern approach.* Englewood Cliffs, NJ: Prentice Hall.

Sutton, R. S. & Barto, A. G. (1998). *Reinforcement learning, an introduction.* Cambridge, MA: MIT Press.

Tesauro, G. (1995). Temporal difference learning and td-gammon. *CACM, 38*(3), 58.

Thearling, K., & Ray, T. (1997). Evolving parallel computation. *Complex Systems, 10*(3), 229–237.

Watkins, C. (1989). *Learning from delayed rewards.* Unpublished doctoral dissertation, Cambridge University.

Whitehead, S., & Ballard, D. (1991). Learning to perceive and act. *Machine Learning, 7*(1), 45.

Wilson, S. (1994). ZCS: A zeroth level classifier system. *Evolutionary Computation,* 2(1), 1–18.

Wilson, S., & Goldberg, D. (1989). A critical review of classifier systems. In *Proc. 3rd ICGA, p 244.* San Mateo, CA: Morgan Kaufmann.

Received February 23, 2000; accepted March 15, 2000.