An Evolutionary Post Production System

Eric B. Baum, Igor Durdanovic
NEC Research Institute, 4 Independence Way, Princeton, NJ 08540,USA.
eric@research.nj.nec.com,igord@research.nj.nec.com
609-951-2712

Abstract

We study the problem of how a computer program can learn, by interacting with an environment, to return an algorithm for solving a class of problems. We describe experiments with a learning system we call Hayek4. Presented with a sequence of randomly chosen Block stacking problems, Hayek4 learns and returns a program for a Post Production system. The program solves arbitrary block stacking problems. The program essentially consists of about 5 learned rules and some learned control information. Solution of an instance with n blocks in its goal stack requires the automatic chaining of the rules in correct sequence about $2n$ deep.

1 Introduction

We study the problem of how a computer program can learn, by interacting with an environment, to return an algorithm for solving a class of problems. This is a problem that humans are often good at. An example can be seen in Blocks World, c.f. figure 1. Humans easily describe a procedure that can solve arbitrary size instances. Rubik's cube is harder, well known example. Humans, after playing with the cube and thinking for a week, often learn so as to be able to solve a randomly-scrambled cube quickly.

This problem is formalized as reinforcement learning(RL)[1]. In RL the learner interacts with an environment which it can sense and take actions on, and which makes "money" payments when a series of correct actions puts it in the right state. The learner's goal is to discover a strategy that earns money efficiently. The literature discusses two approaches to RL. The first, called "value iteration", attempts to learn an evaluation function mapping each state to an estimate of its value, and then returns the algorithm: take the action leading to the state of highest value. This approach has had one striking success in Backgammon, but this is apparently due to the fact that a linear evaluation function is effective in this domain[2]. Value iteration appears essentially hopeless in domains with huge state spaces unless they have an extremely simple and learnable evaluation function[3]. The Blocks World state space grows exponentially with the number of blocks. Without hand coded features, but with algorithmic improvements designed to grapple with the problem, TD learning, the main value iteration approach, could only find a specific block if under no more than 2 other blocks[3]. Given a useful hand coded feature, TD learning succeeds in solving about 8 block problems[4], but not larger ones.

The second approach (known in the RL literature as "policy iteration") attempts to learn a program directly. Evolutionary programming methods can be applied here. However, the space of programs is huge, and its fitness landscape is typically rough, so such methods are of limited applicability. Koza [5] applied GP to the far simpler problem of solving a single instance of Blocks World, rather than producing an algorithm to solve arbitrary BW instances . Given several hand coded features including "next block needed", GP

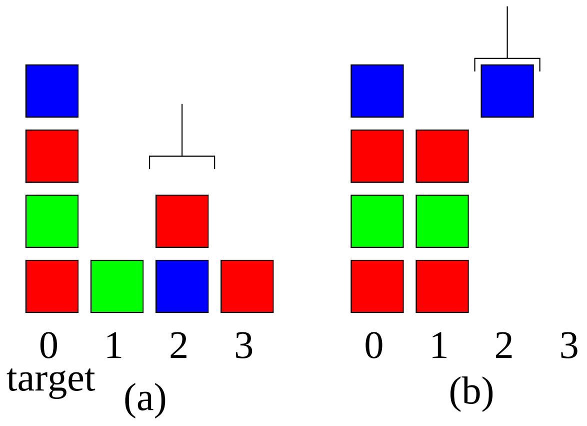

Figure 1: We present a series of randomly chosen Blocks World instances, gradually increasing the size as the system learns. Each instance contains 4 stacks of colored blocks, with $2n$ total blocks and k colors. The leftmost stack, stack 0, serves as a template only and is of height n. The other three stacks contain, between them, the same multi-set 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. Figure (a) shows the initial state for an example with $n = 4$ and $k = 3$. Fig. (b) shows the position just before solution. The instance ends with the blue block on stack 1. Note the goal is to discover an *algorithm* capable of solving random new instances. The best human-generated algorithm of which we are aware uses $4n\log_2(k)$ actions.

succeeded in solving a single 9 block problem. Asked to produce an algorithm for solving random problems, and given a powerful hand coded feature ("number correct"), GP succeeded in producing a program capable of solving only instances with about 5 blocks[4]. Similarly, Inductive Logic Programming solved only about 2 block instances[6].

We view Holland's Classifier systems[7] as a seminal approach to deal with searching the huge program space. By using an economic model to assign credit to modules, it might be possible to factor the search. By finding modules instead of whole programs, the combinatorial explosion might be mitigated. However, Holland Classifiers have not been successful at solving complex problems either[8].

In a series of previous papers[9, 10, 4], we have reported results on why Classifiers fail and how their problems can be corrected. In our view, there are two basic problems. First, their economic model is flawed, which leads to misallocation of credit. We have corrected this by imposing an economic model based on two general principles, conservation of money and strong property rights, which prevent these misallocations. Second, the representation language used by classifiers seems insufficiently powerful. Many useful classifier programs are unstable, since useful rules will go broke and be removed unless high bidding classifiers tend to follow low bidding classifiers[9, 11], the exact opposite of what would be needed for Holland's intuition[7] of "default hierarchies". Although classifiers are in principle Turing complete[12], it is unclear whether classifier systems remain computationally universal when one restricts consideration to configurations which are dynamically stable. We address this representation problem by using a more powerful agent language.

Our first economic model, Hayek1, used simple agents, and because of the dynamic stability problem could only solve large BW problems when given intermediate reward for partial progress[9]. Our last economic model, Hayek3, used agents that compute S-expressions[4]. This model of computation was not Turing-complete, and so the system could not produce a program capable of solving arbitrary instances.

It did, however, produce systems capable of solving random instances with hundreds of blocks. The stark contrast between standard classifier systems, which have trouble forming chains more than a few classifiers deep[8], and Hayek1 and Hayek3's ability to learn systems with stable chains several hundred agents deep showed the critical importance of imposing an economic framework respecting property rights and conservation of money.

Here we report experiments with Hayek4. Hayek4 uses agents that are written in a Post Production System. This language is Turing complete. Although Post proved computational completeness of his Production systems almost as long ago as Turing and Church[13, 14], we are unaware of any previous papers studying the automatic or evolutionary programming of Post systems. The system we describe here, Hayek4, evolves collections of agents that solve arbitrary Blocks World Problems.

§2 will briefly review Post Production Systems. §3 will briefly review our economic construction. §4 will describe our experimental results. §5 is a conclusion.

2 Post Systems

This section briefly reviews Post systems, which are a Turing-complete model of computation. A Post System consists of an axiom and a sequence of productions (also called rules). The axiom consists of a string of symbols. The productions are of form $L \rightarrow R$ where L is the antecedent and R is the consequent. R and L are each strings of symbols and variables, such that any variable appearing in the consequent also appears in the antecedent. Computation proceeds by looking through the productions in order, until a production is found whose antecedent matches the axiom, that is there is some instantiation of the variables as strings of symbols making the antecedent identical to the axiom. This instantiation of the variables is then substituted in the consequent, which replaces the axiom. One iterates this procedure, looking through the productions in order to find a legal substitution, making the substitution, and replacing the axiom, until no production matches, at which time computation halts.

Post proved that any formal system (e.g. any Turing machine) can be reduced to a Post system and indeed even a Post system in canonical form, i.e. having a single axiom and productions only of the form $g\$ \rightarrow \$h$, where the \$ is a variable[14].

In this paper we discuss programs composed of a number of agents, organized into an artificial economy as described in the following section. The agents are each composed of a number of productions. The world is presented to the agent as the axiom, and the agent then computes as a Post system.

For example, in the Blocks World problem, the world is encoded as a string: $a(b)(c)(d)(e)$ where a is either not there or a single symbol chosen from the set $C = \{c_1 ... c_k\}$ of colors, and b, c, d, and e are each strings of symbols chosen from the set C. Here a represents the block in the hand (or is missing if there is no block in hand), b, c, d, e represent respectively the stacks 0,1,2,3.

Our productions are strings over $\{g1, g2, g3, d1, d2, d3, *, (,), y0, xi$ for $i = 1, ..., 8\}$. Here $g1...g3$ (resp. $d1, ..., d3$) mean grab (drop) from stack 1,2,3; $*$ denotes "done" ending the instance, variables xi match a string and variable $y0$ matches only a single character. We use a greedy variable match where the largest string that allows a match is chosen.

Examples of rules that have evolved include $(x0)(x0)(x7)(x1) \rightarrow *g1$ and $(x0y0x1)(x0)(x7)(x3y0x5) \rightarrow g3g2d1$. For more examples, and explanations of how they work, see section 4.

3 Economic Model

Our system, which we call Hayek4, consists of a collection of rules, and a collection of agents. Each agent is composed of a sequence of rules from the collection, a wealth, and a numerical bid.

Computation proceeds in a series of auctions. In each auction, each agent computes its next action by executing a Post-system with the world as initial axiom. The actions of the highest bidder are applied on the world transforming it to the new state.

The winning agent in each auction pays its bid to the winner of the previous auction. If it solves the instance and says done, it collects reward from the world.

After each instance, all agents are assessed a tax proportional to the amount of computation they have done, in order to promote evolution of efficient agents. Also, any agent with less money than it was initiated with is removed and its money returned to its creator. Any rule not used in some living agent, and any rule that has not been applicable in the last 1000 instances is removed.

A number W is initiated as 0, and then raised as larger instances are solved, to be slightly larger than the reward for solving the largest instances being presented. Each auction, any agent with wealth at least $10W$ creates a child, giving it an initial endowment of W. The child is a modified version of its creator, as described below. The system is initiated with a single special agent called "Root". Root does not bid but simply creates random agents. At the end of each instance, each agent passes .25 fraction of its profit in that instance, plus an additional increment of 10^{-4} to its parent.

This structure of payments and capital allocations is based on simple principles. The system is set up so that everything is owned by some agent, property rights are respected, and money is conserved. Under those circumstances, if agents are rational in that they choose to make only profitable transactions, a new agent can earn money only by increasing payment to the system from the world. The agents are not initially rational, indeed they are random, but less rational agents are exploited and go broke.

We ensure that everything is owned by auctioning the world to a single agent. The guideline for all *ad hoc* choices, e.g. the one quarter fraction of profit passed to one's creator, is that the property holder might reasonably make a similar choice if given the option. Creators are viewed as investors-in, (or alternatively owners-of) their children. For example, endowing one's child with W is reasonable since it will need about this amount of money to bid rationally. We did not experiment with these various choices. Our experience with past experimentation in related models is that within reasonable ranges performance is not very sensitive to parameter values, and since runs are stochastic and take a day or more it is impossible to optimize.

By contrast, such property rights are not enforced in most multi-agent systems. For example, Holland's classifiers have multiple agents active at once, so there is no clear title to payments from the world, which are then typically divided among active agents. This is a recipe for "Tragedy of the Commons", since all agents want to be active when payment is expected, whether or not their actions harm the system. ZCS systems[15] have only one action active, but decide which action wins the auction probabilistically, with probability proportional to bid. This violates property rights by forcing agents to accept low bids for their property. When we modify our system to choose the winning bidder probabilistically in this fashion, it immediately breaks and can no longer form long chains of agents or learn to solve Blocks World instances larger than a handful of blocks.

When property rights and conservation of money are not enforced, agents can profit at the expense of the system. Evolution maximizes the interests of the agents. But a local optimum of the system will **not** then be a local optimum for the agents. Thus the system can **not** converge to a local optimum. No wonder you can't form long chains of agents. The problems with Holland Classifiers and related models and the necessity for imposing property rights and conservation of money are discussed in more detail in [9, 10, 4].

Our creation process for new agents/rules is as follows.

```
Root:
  creates agent with randomly between 1..4 rules
  where each rule with p=0.5 is random (new rule)
  or with p=0.5 is a randomly picked existing rule.

Wealthy Agent:
 creates agent that is a modification of itself. To modify
 repeat with exiting p=0.25 the following:
  with p=0.3  it inserts a new rule
  with p=0.3  it deletes an old rule
  with p=0.15 it reshuffles rules
  with p=0.15 it replaces an old rule with new rule
  with p=0.10 it mutates an old rule.

insertion of a new rule:
  with p = 0.25 a new random rule is created
  with p = 0.75 an existing rule is picked

replacement of a rule:
  with p = 0.5 a new random rule is created
  with p = 0.5 an existing rule is picked

mutation of a rule
 left side:
  repeats with exiting p=0.25:
   with p = 1/3 delete a symbol
   with p = 1/3 insert a symbol
   with p = 1/3 replace a symbol
 right side:
  repeats with exitting p=0.25
    with p=0.25 delete a symbol
    with p=0.25 insert a symbol
    with p=0.25 replace a symbol
    with p=0.25 reshuffle rules.
  Brackets, i.e ‘‘(’’ and ‘‘)’’, are being used for structuring purposes
  only and are neither inserted nor deleted.

Any new rule so created is inserted in the rule population.
```

This rule creation process was also not experimented with, simply picked *ad hoc*.

New agents are assigned a numeric bid using the “bid-epsilon” procedure[9]: the first time a new agent has a production that matches, the agent is assigned a bid that is ϵ higher than all competing bids. The new agent thus wins that auction, and its bid is then fixed. ϵ was .01 in these experiments.

4 Experimental Results on Blocks World

Hayek4 was trained by presenting random BW instances, with size chosen according to a distribution that presented increasingly larger instances as Hayek4 learned to solve smaller ones. The distribution was as follows. We present instances of size 1 until one instance is solved. Then we initiate $c = 1$, and present instances of size i with probability chosen from a Gaussian distribution around instances of size c. To be precise, we let $\sigma = c/10 + 2$ and choose i with probability $p(i)$ proportional to $exp(-(c-i)^2/2\sigma^2)$. We maintain a running estimate $solved(c)$ of the fraction of the last 100 instances of size c that have been solved. When $solved(c) > 0.75$ we increase c by 1, and when $solved(c) < .25$ we decrease it by 1. This presents larger instances as we learn.

After a day of computation on a 300 MHz Pentium II processor, Hayek4 learns a program capable of solving arbtirary BW instances. The learned program solves random, new, 100-block instances in several seconds. Fig 2 shows evolution of such a run. In this case, among the over 1000 agents present, agents 1134, 1147, 1154, and 1161 are currently winning all bids, and serve together as a program solving arbitrary instances. The program is simple and intuitive. It first clears blocks off stack 1, putting them on stack 3, until only correctly colored blocks remain on stack 1. Then, if the next block it needs is on stack 3, it digs down in 3 to find it, putting all the blocks it removes on 2. Alternatively, if the next block it needs is on 2, it digs down in 2 to find it, putting all blocks removed on 3.

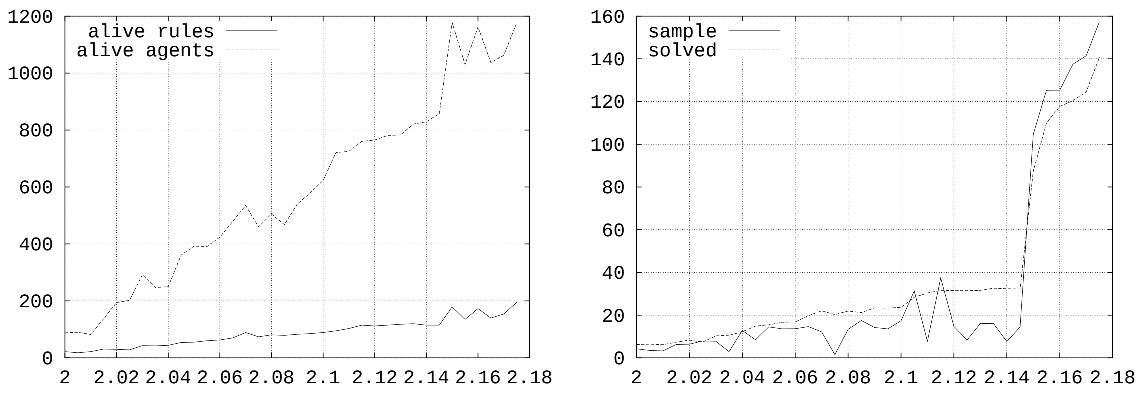

Figure 2: Fig (a) shows the number of alive rules and alive agents. Fig (b) shows the moving average, over the last 100 instances, of the score, computed as $\sqrt{2\sum_i p(i)i}$ for $p(i)$ = fraction instances of size i solved. "Solved" gives score as the system is running, "sampled" gives score on periodic samples where we turn off new agent creation. In sampled mode, Hayek4 is solving *all* instances presented (which are up to size about 158) and is using a program that would solve arbitrary instances. The horizontal axis is in millions of instances. We are showing the period between 2 and 2.18 million instances where it discovered how to solve arbitray instances.

This program is embodied in the following rules. The respective agents contain other rules that don't fire. We have cleaned some semantically unimportant symbols from the rules for pedagogical clarity.
(1) $(x5)(x6)(x1)(x7) \rightarrow g1d3$ in agent 1147, which bids 7.78.
(2) $(x6y0x2)(x6)(x5y0)(x3) \rightarrow g2d1$ in agent 1154, which bids 8.07.
(3) $(x2y0x5)(x2)(x3)(x7y0x0) \rightarrow g3d2$ also in agent 1154.
(4) $(x3y0x5)(x3y0)(x0)(x1) \rightarrow g2d3$ in agent 1134, which bids 8.05.
(5) $(x4y0)(x4)(x7y0)(x1) \rightarrow g2d1*$ in agent 1161, which bids 35.8.

These agents work together as follows. Rule 1 always matches. All rules contained by higher bidding agents in the population match only when stack 1 contains no incorrect blocks, i.e. when every block in stack 1 is the same color as the corresponding block in stack 0. Thus, whenever there are incorrect blocks on stack 1, agent 1147 wins and clears a block from stack 1. This will occur as many auctions in a row as necessary to clear all incorrect blocks from stack 1. Once stack 1 contains no incorrect blocks, the next block needed to extend it must be on stack 2 or stack 3. If the next-needed block is on top of stack 2, rule 2 matches (matching y0 to the color of the next block needed) and moves this block to stack 1. Otherwise, if the next-needed block is on stack 3, rule 3 matches, and moves the top block from from stack 3 to stack 2. As long as the next-needed block is on stack 3, 1154 wins successive auctions and digs down 3 to find the correct block. When the next-needed block is not on 3 or on top of 2, 1134 wins the auction, and uses rule (4) to move blocks from stack 2 to stack 3, until it uncovers the next-needed block on stack 2. Finally, when stack 1 and stack 0 are identical except for the last-needed block, which is on stack 2, agent 1161 wins with a bid of 35.8 and applies rule (5), which moves the last block to 1 and says "done".

Note: (a) this program will solve arbitrary instances. (b) All the agents are profitable: 1147 comes earliest and bids least, 1134 is always followed by itself or 1154. 1154 loses tiny amounts of money when it is followed by 1134, but more than makes up for it by being eventually followed by 1161. 1161 is profitable for any instances with final reward over 35, and so is wealthy in the distribution the system was seeing at the time this set of agents was winning bids. (d) This particular set of agents is winning auctions at the moment, but new agents are continually created and the set of agents winning auctions is thus changing as time goes on. It continues, however, to stably solve instances, sometimes briefly disrupting the universal solver, but soon reassembling it from available rules. (e) Solution of instances depends on all of the more than 1000 agents in the population having bids in appropriate ranges (so that they don't interfere). (f) Solution involves chaining roughly $2n$ agents to solve an instance with n blocks on the goal stack. (g) The solution is intuitive and reasonably efficient. For the randomly colored distribution of instances presented, no strategy would be substantially more efficient. To use substantially fewer actions in worst case (where this algorithm could be quadratically slow) requires a sophisticated strategy that temporarily stacks incorrect blocks on stack 1.

We have done a few comparison experiments. First, we used the exact same scheme except that we chose the winning bidder in each auction with probability proportional to bid, as advocated in Zeroth level Classifier Systems[15] (a widely studied CS variant). This breaks property rights, and immediately broke performance. Such systems solved only problems with about 4 blocks.

Second, we attempted to learn a Post system by a stochastic hill climbing search. We initiated a CBS (current best solution) as an agent containing the rule $(x1)(x2)(x3)(x4) \rightarrow g3g2d1*$ which solves 1 block instances. We then iteratively modified the CBS (exactly as described in §3), tested the modified solution, and replaced the CBS with the modified solution whenever the new solution performed better. We used an instance distribution calculated to work well with this hill climber, presenting instances of a fixed size and increasing the size by one when the CBS succeeded in solving 80% of the current size. This approach built a single Post Production agent, without use of the economic framework. The best this approach could do, after testing several hundred million Post systems, was to produce a Post system that solved about 40% of 10 block problems.

5 Discussion

The success of Hayek4 on Blocks World, coupled with the success of previous Hayek versions using radically different representations, shows that our economic model is consistently able to assign credit and achieve deep chaining of agents and solution of hard Blocks World Problems. Control experiments where property rights are broken, or where no economic structure is used at all, indicate the importance of the property rights in promoting the evolution of cooperation among modules, dividing and conquering problems that are far too complex to solve by alternative means. This is the first Hayek version for which the agents are potentially Turing universal, and accordingly the first to evolve universal solvers from end reward only.

The present work is the first of which we are aware where a Post system has been trained. The pattern matching ability of the Post system appears powerful and potentially of wide application. Post rules appear to have a very different quality from the S-expression representation of Hayek3. S-expression and many other representations perform numerical computations. The Post system is evolving to match structural properties of the system that may be difficult to express numerically.

We are currently experimenting to see whether this system can learn to solve Rubik's cube, and intend to go on to other applications.

1. Sutton, R. S. & Barto, A. G. *Reinforcement Learning, an introduction*. MIT Press, Cambridge, (1998).

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

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

4. Baum, E. & Durdanovic, I. Evolution of cooperative problem-soving in an artificial economy. submitted, available from http://www.neci.nj.nec.com:80/homepages/eric/eric.html, (1999).

5. Koza, J. *Genetic Programming*. MIT Press, Cambridge, (1992).

6. Dzeroski, S., Blockeel, H. & De Raedt, L. Relational reinforcement learning. in *Proc. 12th ICML,* (Shavlik, J., ed) (Morgan Kaufman, San Mateo, CA, 1998).

7. Holland, J. H. Escaping brittleness: the possibilities of general purpose learning algorithms applied to parallel rule-based systems. in *Machine Learning, vol.2 p.593,* (Michalski, R. S., Carbonell, J. G. & Mitchell, T. M., eds). Morgan Kauffman, Los Altos, CA (1986).

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

9. Baum, E. B. Toward a model of mind as a laissez-faire economy of idiots, extended abstract. in *Proc. 13th ICML '96, p28,* (Saitta, L., ed) (Morgan Kaufman, San Francisco, CA, 1996). and in Machine Learning (1999)v35n2.

10. Baum, E. B. Manifesto for an evolutionary economics of intelligence. in *Neural Networks and Machine Learning,p.285,* (Bishop, C. M., ed). Springer-Verlag (1998).

11. Lettau, M. & Uhlig, H. Rule of thumb and dynamic programming. *American Economic Review* , (1999). in press.

12. Forrest, S. Implementing semantic network structures using the classifiersystem. in *Proc. First International Conference on Genetic Algorithms*, 188–196 (Lawrence Erlbaum Associates, Hillsdale, NJ, 1985).

13. Minsky, M. *Computation: Finite and Infinite Machines*. Prentice-Hall Inc, Englewood Cliffs, NJ, (1967).

14. Post, E. L. Formal reductions of the general combinatorial decision problem. *American Journal of Math* **52** (1943).

15. Wilson, S. Zcs: a zeroth level classifier system. *Evolutionary Computation* **2**(1), 1–18 (1994).