A Demonstration of Neural Programming Applied to Non-Markovian Problems

Gabriel Catalin Balan and Sean Luke

George Mason University, Fairfax, VA 22030
gbalan@cs.gmu.edu, sean@cs.gmu.edu

Abstract. Genetic programming may be seen as a recent incarnation of a long-held goal in evolutionary computation: to develop actual computational devices through evolutionary search. Genetic programming is particularly attractive because of the generality of its application, but it has rarely been used in environments requiring iteration, recursion, or internal state. In this paper we investigate a version of genetic programming developed originally by Astro Teller called *neural programming*. Neural programming has a cyclic graph representation which lends itself naturally to implicit internal state and recurrence, but previously has been used primarily for problems which do not need these features. In this paper we show a successful application of neural programming to various partially observable Markov decision processes, originally developed for the learning classifier system community, and which require the use of internal state and iteration.

1 Introduction

Early evolutionary computation work (famously [1]) centered not on optimizing the settings for multidimensional parameter spaces but on using simulated evolution to develop *mechanisms* which perform computational functions. This goal has resurfaced in different forms over time, including the evolution of classifier systems, neural networks and automata, and most recently, genetic programming. The impetus for GP has from the beginning been "computer programs" which ideally can "perform iteration and recursion", "define intermediate values and subprograms", etc. ([2], p. 2). In keeping with this motivation, many GP methodologies have encorporated forms of iteration, recursion, and internal state. This notwithstanding, nearly all of genetic programming concerns itself with relatively simple feed-forward functions, and given the lack of even rudimentary iteration or internal state in common GP problems it seems strange to refer to GP as genetic *programming* at all.

As discussed later, much of the work in computational mechanisms involving iteration and internal state, and many illustrative problem domains, have recently been in learning classifier systems. Extensions to classifier systems such as XCS have of late added basic facilities to learn agent policies for partially-observable Markov decision processes, whose non-Markovian features require internal state. But classifier systems are of limited generality. Our interest is in evolving programs as graph structures with at least basic iteration and internal state, in forms sufficiently general to be used for a wide variety of tasks.

Most work in recurrent GP has so far centered around methods where a single node in a graph is currently in control at any one time, and control passes forward to a single successor node. An exception to this pattern is Astro Teller's *neural programming* (NP) architecture [3] which activates all nodes in the graph simultaneously, letting data flow synchronously in parallel to successor nodes. Teller's model has appealing features which we believe could provide interesting advantages over a single state model: data parallelism, more natural use of functions rather than state transitions, and the promise of more graceful degradation under modification. However Teller did not use his model in non-Markovian environments requiring internal state and iteration, choosing instead to demonstrate its functionality with problems to which a traditional GP representation may be easily applied. In this paper we will demonstrate, as a proof of concept *only*, that the NP model may be used in non-Markovian environments.

The test problems we chose in this paper are "Woods" problems common to learning classifier systems. We chose these problems for our test cases specifically because they are *not* problems for which neural programming is likely to be well-designed, particularly because our neural programming representations can be of any size. These problems are custom-tailored for discrete, policy-oriented single-state systems such as XCS: but we will demonstrate neural programming's efficacy on these problems. In future work we hope to test neural programming on continuous environments for which we think it is likely to be a much better match, such as two-pole balancing with no provided velocity information.

The rest of the paper is organized as follows. First we discuss relevant previous work in GP and learning classifier systems, including the Woods problems. We then describe our version of neural programming and how we apply it to these problems. Last, we present the experimental work and results.

1.1 Genetic Programming and Neural Programming

Though some early work attempted to add mapping, iteration, or recursive operators to genetic programming [2], these aims have never caught on in the community, with a few important exceptions ([4, 5]). Iteration operators and internal memory are generally rare in GP. Recursion is easily achievable with *automatically defined functions* (ADFs), but ADFs are primarily used to add modularity rather than recursive function calls.

Angeline [6] proposed a variation on the ADF theme, called *multiple interacting programs* (MIPs) nets, which replaced recursion with forward data flow: trees generated values which were then fed into "parent" trees: cycles in parenthood were permitted. Like neural programming, MIPs nets allowed for parallelism.

The alternative method, using a single active state at a time, is presented in [7]. Here, nodes in a cyclic call graph contain both actions (side effects) and conditional branches. The graph structure was later replaced with a sequence of linear genetic programming instructions [8]. Similarly [9] allowed GP programs with single active states to be augmented with forward and backward edges,

Mutate-ERC (20%) changes all ephemeral random constants.

Mutate-Nodes (20%) replaces a random number (from 1 to 3) of nodes with ones of identical arity, so that the graph's topology is not affected.

Mutate-Edges (25%) changes the source nodes of a random number (from 1 to 3) of edges.

Mutate-Result-Node (20%) randomly designates a new result node.

Crossover (10%) swaps fragments between two graphs. First, each graph randomly chooses a number N, where N is the number of nodes to be kept and the remaining M are to be crossed over. N may differ for the two graphs. The values of N are chosen with the constraints that both graphs may not crossover all, nor zero, of their nodes, and that no crossover may result in a graph of less than 2 or more than 40 nodes. A BFS traversal is then performed on each graph, starting with the result node, until N nodes have been traversed. The subgraph consisting of the M nodes not in the traversal is swapped with its counterpart in the other individual. Nodes with broken inputs have new input edges attached from randomly-chosen sources.

Randomize (5%) replaces the individual with a brand-new individual. First, between 2 and 40 nodes are generated and placed in an array: terminals (zero-arity nodes) are selected 1/3 of the time, and nonterminals 2/3 of the time. Nodes needing inputs are connected to neighboring sources chosen at random from up to 20% of the array length in either direction.

Table 1. Genetic operators, with probabilities of occurrence and descriptions.

forming cycles. [10] incorporated time delays to prevent infinite execution loops in the graph. This work also showed the effectiveness of the method using non-Markovian multiagent problems.

In Teller's neural programming (NP) [3], the representation is a graph of nodes similar to tree-based GP nodes. Each node takes input from other nodes, applies a function to its inputs, and outputs the result. Cycles are permitted and are common. In NP, all the nodes execute their functions simultaneously and synchronously based on the data each received in the previous time period. Since there is no single state, there is also no "start node", and so nodes must all have initial default values they output until they begin receiving data from other nodes. Just as in tree-based GP, NP nodes may also have zero inputs, and ephemeral random constants are permitted.

Our implementation of NP differs from the original in a few important ways. First, NP had an ADF mechanism, but in our experiments we will not use it. Second, NP's nodes were normally not restricted in arity: as many or as few nodes as necessary could feed them data. In our implementation this is not the case — we will enforce a fixed arity in a fashion similar to tree-based GP nodes. Third, NP originally featured a credit assignment mechanism used to determine which edges and nodes should be more susceptible to genetic modification. This mechanism was relatively easy to use for Teller's original signal-classification applications, but in the Woods problem it is somewhat problematic and we have discarded it. Last, neural programming graphs were augmented with one or more fixed "output nodes" whose values indicated the output of the program. We do not use a fixed output node, but instead allow the system to designate any node in the graph to also be the "result node" whose output is the output of the program. Each time step, the current value of the result node may change as data continues to flow through the graph.

Graph-structured representations inevitably require exotic genetic operators. Table 1 contains the description of the operators we used. We iteratively picked

	Ambiguous		Perfect solution	
Environment	classes	states	max length	average length
Woods101	1	2	4	2.9
Maze7	1	2	7	$4.\overline{3}$
MazeF4	1	2	7	4.5
E1	9	20	4	$2.\overline{81}$
E2	5	36	5	$2.97919\overline{6}$
Maze10	3	7	8	$5.0\overline{5}$

Table 2. The characteristics of the Woods mazes. Shown are the number of ambiguous states and the number of equivalence classes they fall into; and the maximum and average path lengths of perfect solutions.

one operator by its probability, then applied it once to selected individuals, resulting in one or (for Crossover) two children. Individuals were selected using tournament selection of size 5. Population initialization used the Randomize operator described in Table 1.

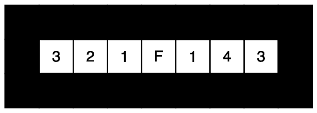
2 The Woods Environments

In a simple version of the Woods problem [11, 12] an agent navigates a grid-based maze trying to find food. The grid cells are either obstacles ("woods"), food (the goal), or are empty. An agent can only sense the contents of cells in its immediate 8-neighborhood, and may only transition to one of the eight neighbors in that timestep. If an agent attempts to transition to a woods cell, it fails to do so and remains at its current location. The goal is to guide the agent to the food in the minimal number of steps *regardless of initial position.*

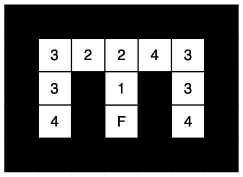
Many woods problems are Markovian: a set of behaviors may be learned for the agent which guide it to the food each time based solely on the current sensor values. In other problems, different cells yield identical sensor values but require different actions, and so the agent must rely to some degree on internal memory. This situation is referred to as the *perception aliasing problem*, and the associated environments are said to be non-Markovian.

Non-Markovian environments have long been studied by the learning classifier system literature. Early work [14] allowed rules to write to a global internal state. Wilson [15] more formally suggested adding internal memory condition and action segments to rules, allowing the storage of information in bit-registers. Bit registers have since been used to extend other models ([16, 17]). Relatively few such papers allow more than a fixed, and usually very small, number of registers. An alternative approach is to allow the action part of a rule be a sequence of actions rather than a single action [18].

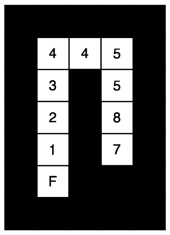
Work in these problems has also been done using Pitt-approach rule systems [13], extending the mechanism to finite-state automata. Some work has also been done in performing pareto optimization to minimize the amount of internal state necessary [19].



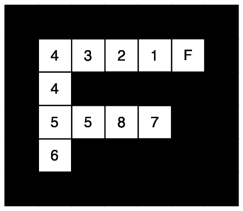
(a)



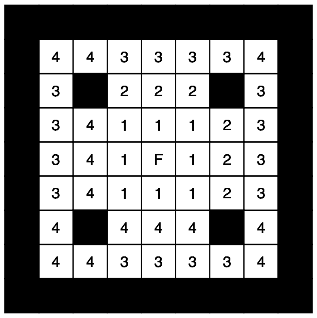
(b)



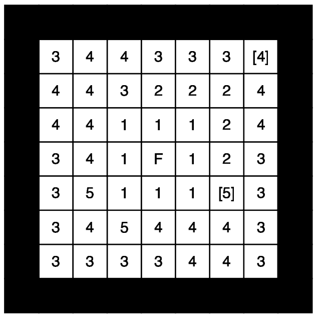
(c)



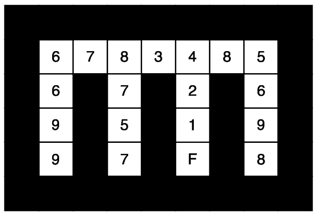
(d)



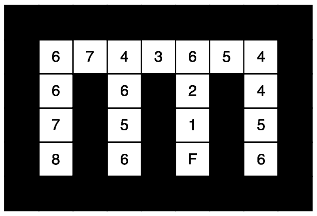
(e)



(f)



(g)



(h)

Fig. 1. The non-Markovian Woods environments used in this paper. The notation is consistent with [13]: black cells are obstacles and F is the goal state. Each figure also shows one (possibly suboptimal) solution to the problem: a cell is numbered with the steps necessary to reach the goal from that cell using this solution. (a) Woods100 shows the solution represented by the neural program in Figure 2. (b) Woods101, (c) Maze7, (d) MazeF4, (e) E1, (f) E2, and (g) Maze10 show solutions discovered by the NP system in the paper; cells marked [ ] are one step longer than an optimal solution. (g) shows an optimal solution for Maze10 suggested by Samuel Landau.

Delay (v_1)	Outputs its input.
To8 (v_1)	Rounds its input to point to the nearest of the eight canonical directions.
Wall Filter / Empty Filter (v_1)	Returns its input if the input points to a wall / empty cell. Otherwise, returns the vector {0,0}.
Wall Radar / Empty Radar ()	Cycles through vectors pointing towards the neighboring walls / empty cells in trigonometric order, outputting each in turn. After every complete cycle, outputs one {0,0} vector.
Wall Sum / Empty Sum ()	Returns the sum of those vectors from the eight directions that point to walls / empty spaces.
Add (v_1, v_2)	Adds two vectors together.
Sub (v_1, v_2)	Subtracts two vectors.
Accumulator (v_1)	Accumulates the vectors inputted over time. Equivalent to an Add node with one input connected to its own output.
Time Weighted Accumulator (v_1)	Accumulates the vectors inputted over time, each multiplied by the time step. The output at time t is $\sum_{i=1}^{t-1} i \times input_i / \sum_{i=1}^{t-1} i$.
Vector ERC ()	An ephemeral random 2D constant.
Rotation ERC (v_1)	Rotates its input vector clockwise by an angle equal to $k\pi/8$, where k is an integer ephemeral random constant in the range $1 \ldots 15$.
If-Equal (v_1, v_2, v_3, v_4)	Returns v_3 if $v_1 = v_2$, else v_4
If-Greater (v_1, v_2, v_3, v_4)	Returns v_3 if $\lvert v_1 \rvert > \lvert v_2 \rvert$, else v_4
If-Dot (v_1, v_2, v_3, v_4)	Returns v_3 if $v_1 \cdot v_2 > 0$, i.e. $\sphericalangle(v_1, v_2) \in [-\pi/2, \pi/2]$, else v_4

Table 3. The Function Set for the Woods Problem. Each function is followed by the inputs to the function, and a function description. All functions operate on 2D real-valued vectors. If a node has not yet received input, its output is the vector {0,0}.

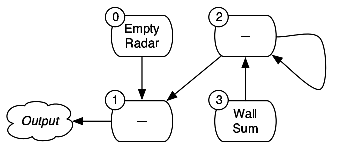
The Woods problems studied here are shown in Figure 1. Characteristics of these environments (number of ambiguous states, number of classes those states fall into, and maximum and average path lengths of perfect solutions) are shown in Table 2.

2.1 Application of Neural Programming to the Woods Problem

To move the agent around a Woods maze, we chose to have the neural program output a vector rather than move the agent via function side effects. All functions in the function set (shown in Table 3) operated on 2D continuous vectors and returned vectors.

Each time we needed to move the agent one square, we first determined if the agent was right next to food. If so, we hard-coded its movement directly to the food. If not, we evaluated the neural program graph to determine where the agent should go. First, the graph was stepped m times to "prime" it. Then the graph continued to be stepped until the result node outputted a vector in a valid direction (that is, one not pointing into woods and not {0,0}), and moved the agent to the eight-neighbored square in the direction of that vector. If after $M \geq m$ steps the result node still did not yield a valid direction, the agent was not moved. The particular sizes of M and m chosen are problem-specific, and are indicated as (m, M) in the discussion for each experiment.

Fitness assessment may be done in a variety of ways. We chose to run the neural program with the agent placed in each empty square, then averaged the lengths of the paths it took to find the food. An agent might never reach the food, so path length is limited to a problem dependent cutoff threshold *max-walk*.



0	Empty Radar
1*	Sub (2, 0)
2	Sub (2, 3)
3	Wall Sum

Fig. 2. Two views of a neural program solving the Woods100 maze. The table at right lists the four program nodes (0–3), and indicates the nodes provided as input. The result node is marked with a ∗. The figure shows the equivalent graph structure.

		$\langle 1,1\rangle$						$\langle 2,1\rangle$						$\langle 3,1\rangle$						position
	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	time step
0 Empty Radar	0	1 →	0	1 →	0	1 →	0	1 →	1 ←	0	1 →	1 ←	0	1 ←	0	1 ←	0	1 ←	0	
1 Sub(2,0)	0	1 ←	1 →	1 →	1 →	3 →	3 →	5 →	5 →	7 →	6 →	5 →	7 →	6 →	7 →	6 →	7 →	6 →	7 →	
2 Sub(2,3)	0	0	1 →	2 →	3 →	4 →	5 →	6 →	6 →	6 →	6 →	6 →	6 →	6 →	6 →	6 →	6 →	6 →	6 →	
3 Wall Sum	0	1 ←	1 ←	1 ←	1 ←	1 ←	1 ←	0	0	0	0	0	0	0	0	0	0	0	0	

Table 4. Value traces for all nodes of the graph in Figure 2 operating in the Woods100 maze starting at position $\langle 1,1\rangle$. The agent is primed six times prior to making an action. The action is read from the result node shown in light gray. Cells contain the output vectors for each node in polar coordinates: size and heading.

An Example Consider the neural program shown in Figure 2. The table to the right shows the four nodes of the program, each labelled with an index 0–4. Node 1 is labelled with a ∗, indicating that it is the result node. The figure to the left shows the recurrent call-graph of the neural program.

This neural program will be run with an (m, M) value of $(6, 12)$. Table 4 shows its trace moving from position $\langle 1,1\rangle$ to $\langle 3,1\rangle$ in the Woods100 maze. In this example, before each action the neural program is primed six times, after which it happens to be outputting a valid vector and needs no further stepping. Figure 1(a) shows the path lengths of the solution followed by this program were it to be started in any particular empty location.

3 Experimental results

We have tested the described system on six non-Markovian environments: Woods101, Maze7, MazeF4, E1, E2 and Maze10 (the Woods100 maze is trivial). These environments vary widely in difficulty.

We implemented NP using ECJ ([20]). Unless specified otherwise, all experiments evolved populations of 1000 individuals for 100 generations each. Each

Problem	Priming (m, M)	Evals	Random Search Mean	Random Search Stdev	Random Search #Ideals	Evolutionary Search Mean	Evolutionary Search Stdev	Evolutionary Search #Ideals
Woods101	(6,6)	500K	3.47999	0.18735	0	3.42399	0.35945	2
Woods101	(6,12)	100K	4.08200	0.44524	0	3.40599	0.36500	2
Maze7	(17,34)	100K	7.67333	0.83944	0	5.97777	1.04424	3
MazeF4	(11,22)	100K	8.21	1.17649	0	6.97600	1.45301	0
E1	(11,22)	100K	3.22914	0.47751	0	2.92454	0.11119	12
E2	(11,22)	500K	3.17666	0.03885	0	3.08416	0.03668	0
Maze10	(12,24)	100K	19.8344	2.42619	0	13.0622	5.05325	0

Table 5. NP vs. Random search.

individual was one neural program containing between 2 and 40 nodes. For all environments but the last one, *max-walk* was set to 10.

This is the first time a generalized graph-based GP method has been tried on these problems. Even if it were not, few examples from the GP literature discussed earlier provide useful statistics to compare against. The method described here, as we imagined, is generally far outperformed by the learning classifier literature, though it fares well against the Pitt-approach rule system results. This is to be expected though: these problems are custom-tailored for LCS systems. Our method is hobbled by searching a wide range of internal states and a more general-purpose architecture. This reduces us to demonstrating proof of concept. We do so by comparing the search method against random search for the same representation.

Thus we compare the best-of-run fitness values of 50 runs with the best values found in 50 runs of random search with an equal budget of evaluations, using an ANOVA at 95% confidence. Note that by random search we mean generating random individuals every generation, and not just performing uniform selection. Figure 5 shows the results. In every case we outperform random search, except for the easy 500K Woods101 problem, for which the difference is statistically insignificant. We also find ideal solutions in several cases, whereas random search finds none.

3.1 Woods101

The Woods101 environment, studied in [16, 21], is a very simple maze with only one pair of perception aliased states — see Figure 1(b). This means that learning classifiers and finite-state automata only require two internal states to solve the problem. We found optimal programs in two runs set at (5,10). The most compact was:

```
0*   If-Equal (1, 0, 2, 3)     3   Sub (4, 5)
1    Wall Filter (0)           4   If-Dot (4, 3, 0, 1)
2    Empty Radar               5   Wall Sum
```

3.2 Maze7

The Maze7 environment (Figure 1(c)) was discussed in [22, 21] and again contains only one pair of aliased positions. However, one of the paths must pass through both ambiguous states. While the longest solution path for the Woods101 environment is 4, the longest solution path in Maze7 is 7, and the average is $4.\overline{3}$. Thus Maze7 is considered more challenging than Woods101. One discovered perfect solution, set at (17,34), was:

0*	Space Filter (1)	4	Delay (6)
1	If-Dot (2, 3, 2, 1)	5	Empty Radar
2	Empty Radar	6	Time Weighted Accumulator (7)
3	Rotation ERC [$3\pi/8$] (4)	7	If-Greater (6, 5, 7, 5)

3.3 MazeF4

The MazeF4 environment (Figure 1(d)) is a rotation of Maze7 plus an extra non-ambiguous empty space. The optimal solution has an average path length of 4.5 and a maximum of 7 steps. In the fifty runs of our experiment we did not discover an optimal solution; but in other experiments we discovered several, including the following (8,16) program:

0*	If-Dot (1, 2, 3, 4)	9	Wall Radar
1	Space Filter (5)	10	Vector ERC [-0.9425225, 0.3341425]
2	Space Filter (6)	11	Sub (12, 13)
3	Empty Radar	12	If-Equal (14, 3, 15, 0)
4	If-Dot (7, 8, 8, 4)	13	Vector ERC [-0.6436193, 0.7653458]
5	Add (9, 9)	14	If-Dot (12, 13, 16, 15)
6	Rotation ERC [$7\pi/8$] (10)	15	Rotation ERC [$-\pi/8$] (3)
7	Delay (11)	16	Time Weighted Accumulator (16)
8	Empty Radar		

3.4 E1 and E2

The E1 and E2 environments were introduced in [18]. While the other mazes studied in this work consist of narrow corridors only, the E1 and E2 environments feature wide-open areas. E2 is an empty environment with a food cell in the center. There are 36 perceptually aliased positions belonging to five different perception classes. The optimal solution in E2 is $2.9791\overline{6}$ steps on average and 5 steps maximum. We discovered no optimal solutions for E2. The closest, an (11,22) program, is only suboptimal by two steps. E2 is shown in Figure 1(f) marked with the suboptimal steps. The program is below:

0*	Time Weighted Accumulator (1)	9	If-Greater (3, 45, 13, 14)
1	If-Greater (2, 3, 2, 4)	10	Delay (15)
2	To8 (5)	11	Wall Radar
3	Time Weighted Accumulator (6)	13	Space Filter (1)
4	Empty Radar	14	Accumulator (18)
5	Empty Radar	15	Empty Sum
6	Sub (7, 8)	18	Sub (19, 15)
7	Add (9, 10)	19	Vector ERC [0.0, -1.0]
8	Rotation ERC [$\pi/2$] (11)	45	Vector ERC [1.0, -1.0]

E1 (Figure 1(e)) is a significantly easier version of E2, with only 20 aliased positions belonging to nine classes. For a more intuitive depiction of the perception classes, see [13]. The average optimal path length is $2.\overline{81}$, and the maximum is 4. We found twelve optimal solutions for E1, including the following (11,22) program:

0*	Accumulator (1)	4	Empty Radar
1	If-Greater (2, 3, 4, 1)	5	Add (5, 4)
2	Add (5, 0)	6	Wall Filter (5)
3	If-Greater (2, 0, 6, 3)		

3.5 Maze10

This environment was introduced in [17]. There are seven perceptually aliased positions split into three classes, not counting ambiguous cells that require the same action. Although only one bit of memory is theoretically enough to solve the problem, it has proven prohibitively difficult [23]. The average optimal path length is 5, and the maximum is 8. Because the optimal solution contains an eight-step path, our customary *max-walk* threshold of 10 steps used so far is too small, and so we increased it to 25.

We did not find an optimal program for this problem. Our best solution, with an average optimal path length of $6.1\overline{6}$ steps, was the following (13,26) program:

0*	If-Equal (1, 2, 3, 1)	6	Sub (3, 12)
1	To8 (3)	7	If-Greater (3, 8, 12, 8)
2	To8 (4)	8	Empty Radar
3	If-Dot (5, 6, 7, 8)	9	Wall Sum
4	If-Dot (9, 9, 10, 1)	10	Empty Radar
5	Rotation ERC [$5\pi/8$] (12)	12	Vector ERC [0.99563, 0.09339]

Figure 1(g) shows the path lengths for this solution, while Figure 1(h) shows the path lengths for an optimal solution.

4 Conclusions and Future Work

One of neural programming's most interesting and attractive features is the cyclic nature of its representation. Surprisingly, previous work in NP [3] does not rely on this feature at all; indeed recurrence in the graph is more or less relegated to providing modularity and iterative improvement of an existing solution. In this paper our goal was to demonstrate that NP could also make use of its recurrence to solve problems requiring internal state.

In order to demonstrate the generality of the mechanism, we chose a problem which we knew beforehand that NP was not suited to. We tested NP on various non-Markovian problems more commonly used for policy-learning methods such as learning classifier systems. These problems ranged widely in difficulty. Accordingly, as expected, our success rates are lower than those reported in the

LCS community and the number of evaluations required are higher. Still, in most of the problems we were able to find optimal solutions; in the remainder we found near-optimal solutions.

In future work, our next goal is to demonstrate the efficacy of NP in a very different environment for which LCS or FSAs are *not* suited but NP is: one requiring recurrent mathematical calculation and internal state: for example, a problem solvable only by partial differential equations. One such problem domain that we are planning to tackle is a two-pole balancing problem with only positional information provided.

Another area of future work we hope to pursue is how to add recursion and expandable internal state to the neural programming paradigm in a way which is tractable. While we may argue for the generality of the neural programming method in terms of application (similar to the variety of problems to which GP is applied), and while NP can perform iteration and use finite amounts of internal state, nonetheless we have not investigated how to get NP to solve problems requiring a stack. Last, we hope to investigate the application of more traditional GP methods (ADFs with indexed memory, etc.) against such problem domains as the one described. Genetic programming has always held an implied promise of discovery of arbitrary computer programs to solve problems and we hope this work might further this effort.

Acknowledgments

We thank Keith Sullivan for his help in the paper preparation, and the reviewers for their helpful suggestions.

References

1. Fogel, L.: Intelligence Through Simulated Evolution: Fourty Years of Evolutionary Programming. Wiley Series on Intelligent Systems (1999)
2. Koza, J.R.: Genetic programming: on the programming of computers by means of natural selection. MIT Press, Cambridge, MA (1992)
3. Teller, A.: Algorithm Evolution with Internal Reinforcement for Signal Understanding. PhD thesis, Carnegie Mellon University (1998)
4. Koza, J.R., Bennett III, F.H., Andre, D., Keane, M.A.: Genetic Programming III - Darwinian Invention and Problem Solving. Morgan Kaufmann (1999)
5. O'Neill, M., Ryan, C.: Under the hood of grammatical evolution. In Banzhaf, W., Daida, J., Eiben, A.E., Garzon, M.H., Honavar, V., Jakiela, M., Smith, R.E., eds.: Proceedings of the Genetic and Evolutionary Computation Conference. Volume 2., Orlando, Florida, USA, Morgan Kaufmann (1999) 1143–1148
6. Angeline, P.J.: Multiple interacting programs: A representation for evolving complex behaviors. Cybernetics and Systems **29** (1998) 779–806
7. Kantschik, W., Dittrich, P., Brameier, M., Banzhaf, W.: Meta-evolution in graph GP. In Poli, R., Nordin, P., Langdon, W.B., Fogarty, T.C., eds.: Genetic Programming, Proceedings of EuroGP'99. Volume 1598., Springer-Verlag (1999) 15–28

8. Kantschik, W., Banzhaf, W.: Linear-graph GP – a new GP structure. In Foster, J.A., Lutton, E., Miller, J., Ryan, C., Tettamanzi, A.G.B., eds.: Proceedings of the Fifth European Conference on Genetic Programming (EuroGP-2002). Volume 2278 of LNCS., Kinsale, Ireland, Springer Verlag (2002) 83–92
9. Green, F.B.: Performance of diploid dominance with genetically synthesized signal processing networks. In Bäck, T., ed.: Proceedings of the Seventh International Conference on Genetic Algorithms (ICGA97), San Francisco, CA, Morgan Kaufmann (1997) 615–622
10. Katagiri, H., Hirasawa, K., Hu, J., Murata, J.: Network structure oriented evolutionary model-genetic network programming-and its comparison with genetic programming. In Goodman, E.D., ed.: 2001 Genetic and Evolutionary Computation Conference Late Breaking Papers, San Francisco, California, USA (2001) 219–226
11. Wilson, S.W.: Knowledge growth in an artificial animal. In: Proceedings of the International Conference on Genetic Algorithms and Their Applications, Pittsburgh, PA (1985) 16–23
12. Wilson, S.W.: The animat path to AI. In Meyer, J.A., Wilson, S.W., eds.: Proceedings of the First International Conference on Simulation of Adaptive Behavior (From animals to animats). (1991) 15–21
13. Landau, S., Sigaud, O., Picault, S., Gérard, P.: An experimental comparison between ATNoSFERES and ACS. In Stolzmann, W., Lanzi, P.L., Wilson, S.W., eds.: IWLCS-03. Proceedings of the Sixth International Workshop on Learning Classifier Systems. LNAI, Chicago, Springer (2003)
14. Holland, J.H.: Escaping brittleness: The possibilities of general-purpose learning algorithms applied to parallel rule-based systems. In Michalski, R.S., Carbonell, J.G., Mitchell, T.M., eds.: Machine Learning: An Artificial Intelligence Approach: Volume II. Kaufmann, Los Altos, CA (1986) 593–623
15. Wilson, S.W.: ZCS: A zeroth level classifier system. Evolutionary Computation **2** (1994) 1–18
16. Cliff, D., Ross, S.: Adding Temporary Memory to ZCS. Adaptive Behavior **3** (1995) 101–150
17. Lanzi, P.L.: An analysis of the memory mechanism of XCSM. In Koza, J.R., Banzhaf, W., Chellapilla, K., Deb, K., Dorigo, M., Fogel, D.B., Garzon, M.H., Goldberg, D.E., Iba, H., Riolo, R., eds.: Genetic Programming 1998: Proceedings of the Third Annual Conference, University of Wisconsin, Madison, Wisconsin, USA, Morgan Kaufmann (1998) 643–651
18. Metivier, M., Lattaud, C.: Further comparison between ATNoSFERES and XCSM. In Stolzmann, W., Lanzi, P.L., Wilson, S.W., eds.: Proceedings of the Fifth International Workshop on Learning Classifier Systems, Springer (2002) 143–163
19. Kim, D., Hallam, J.C.T.: An evolutionary approach to quantify internal states needed for the woods problem. In Hallam, B., Floreano, D., Hallam, J.C.T., Hayes, G., Meyer, J.A., eds.: From Animals to Animats, MIT Press (2002) 312–322
20. Luke, S. ECJ 11: A Java EC research system. http://cs.gmu.edu/∼eclab/projects/ecj/ (2004)
21. Lanzi, P.L., Wilson, S.W.: Toward optimal classifier system performance in nonmarkov environments. In: Evolutionary Computation. Volume 8. (2000) 393–418
22. Lanzi, P.L.: Adding Memory to XCS. In: Proceedings of the IEEE Conference on Evolutionary Computation (ICEC98), IEEE Press (1998)
23. Landau, S., Picault, S., Sigaud, O., Gérard, P.: Further comparison between ATNoSFERES and XCSM. In Stolzmann, W., Lanzi, P.L., Wilson, S.W., eds.: IWLCS-02. Proceedings of the Fifth International Workshop on Learning Classifier Systems. LNAI, Granada, Springer (2002)