Evolving Teleo-Reactive Programs for Block Stacking using Indexicals through Genetic Programming

Mykel J. Kochenderfer
Department of Computer Science
Stanford University
Stanford, CA 94305
mykel@cs.stanford.edu

Abstract

This paper demonstrates how strongly-typed genetic programming may be used to evolve valid teleo-reactive programs that solve the general block-stacking problem using indexicals.

1 INTRODUCTION

Teleo-reactive programs have been proposed by Nils Nilsson (1992, 1994) as a robust agent control structure. The agent is directed towards a goal based on continuous evaluation of perceptual inputs. Teleo-reactive programs consist of an ordered list of production rules, and they tend to be relatively simple for humans to write and understand. Teleo-reactive programs have been successfully applied to a variety of domains, including robot planning and aircraft control in a flight simulator (Benson, 1996).

The effectiveness of teleo-reactive programs has also been demonstrated in the blocks world domain.[1] A teleo-reactive system proposed by Nilsson (2001) is capable of stacking any specified tower of blocks on a table from any configuration without search. A wide variety of learning techniques for teleo-reactive programs have been proposed (Nilsson, 2000) that may be applied to solving block-stacking problems and other robot problems.

This paper demonstrates the suitability of genetic programming for the automatic creation of teleo-reactive programs for the block-stacking problem. Although work has already been done involving genetic programming and block stacking (Koza, 1992; Baum and Durdanovic, 2000), this paper is the first to apply genetic programming to learning teleo-reactive programs.

Work done by Koza (1992) focused on a simplified block-stacking problem. The goal of the reduced version of the problem is to move a predefined set of blocks from the table to a specified tower. Any blocks on the table may be randomly accessed in Koza's problem, but in the version of the problem treated in this paper, only blocks at the top of stacks are accessible. In other words, in this paper we assume that for the agent to manipulate a block in the middle of a particular stack, the agent must first remove the blocks on top of it.

The block-stacking teleo-reactive program discussed in Nilsson's paper (2001) uses predicates of the form `On(x,y)` to describe the environment sensed by the agent, where `x` may be any block and `y` may be any block or the table. The blocks are denoted by letters, `A`, `B`, `C`, ..., and the table is denoted by `Ta`. The only action available to the agent is `Move(x,y)` that moves block `x`, which does not have any blocks on top of it, to the top of `y`, which may be either another block or the table.

Work done by Baum and Durdanovic (2000) involving strongly-typed genetic programming and block stacking followed a convention similar to that of Nilsson. In addition to constants, arithmetic functions, conditional tests, loop controls, their S-expressions included the following functions: `Look(i,j)` that returns the color of the block at location `(i,j)`, `Grab(i)` that grabs the top block off of column `i`, `Drop(i)` that drops the block the agent is holding onto the top of column `i`, and `Done` that terminates the program.

In Koza's application of genetic programming to block-stacking (1992), he decides to use indexicals instead of explicit references to the blocks. Indexicals are expressions whose meaning depends upon the context in

[1] A link to an animated Java applet demonstrating the use of teleo-reactive programs and the triple tower architecture may be found online here: `http://cs.stanford.edu/~nilsson/trweb/tr.html`.

which it is employed (as discussed in Perry, 1996). The indexicals used by Koza included terminals such as `MN`, which represents the "next needed block", and functions such as `MS`, which moves the specified block to the target tower. The experiments described in this paper use indexicals similar to those used by Koza for the function and terminal sets.

This paper demonstrates how strongly-typed genetic programming can evolve valid teleo-reactive programs that solve the general block-stacking problem using indexicals.

Some of the relevant details of teleo-reactive programs and a discussion of their incorporation into genetic programming is found in the next section. In section 3, we discuss the methods used in evolving teleo-reactive programs for block stacking. A report of the results follows in section 5, and conclusions are drawn from these results in section 6. Further work is discussed in the final section.

2 TELEO-REACTIVE PROGRAMS

A teleo-reactive program, as proposed by Nilsson (1992, 1994), is an ordered list of production rules, as shown below:

$$\begin{aligned} K_1 &\rightarrow a_1 \\ &\vdots \\ K_i &\rightarrow a_i \\ &\vdots \\ K_m &\rightarrow a_m \end{aligned}$$

The K_i are conditions, which are evaluated with reference to a world model, and the a_i are actions on the world. The conditions K_i may have free variables that are bound when the teleo-reactive program is called. Actions may be primitive, they may be a set of actions to be executed in parallel, or they may be another teleo-reactive program (thereby enabling recursion). Typically, K_1 is the goal condition, and a_1 is the null action.

The rules are scanned from top to bottom for the first condition that is satisfied and then the corresponding action is taken. Since the rules are scanned continuously, *durative* actions are possible in addition to *discrete* actions. In our version of block stacking, we are concerned only with *discrete* actions. For a more detailed explanation of how teleo-reactive programs work, see Nilsson (1992, 1994).

Teleo-reactive programs may be represented as S-expressions, making it possible to evolve such programs using the standard genetic programming algorithms. Figure 1 shows the general structure of such an S-expression, as proposed by Nilsson (2002).

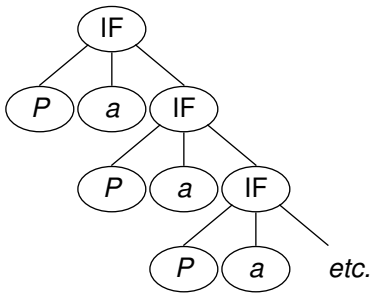

Figure 1: The general structure of a teleo-reactive program represented as an S-expression.

$\mathcal{P}$ is any allowed condition, a is any primitive action, and `IF` is the three-place "if-then-else" function. If we limit conditions to be conjunctions, the condition $\mathcal{P}$ may be expanded in BNF according to the rule

$$\mathcal{P} \rightarrow Q \mid (\texttt{and}\ Q\ \mathcal{P})$$

where Q is an atomic formula and `and` is the usual two-place conjunction. Nilsson suggests that the bottom node of the program should be a two-place `if` function whose condition is `T`.

This paper uses a slightly different structure for the teleo-reactive S-expression than the form proposed by Nilsson. The S-expression may be represented by the following grammar. The terminal p may be any primitive perception of the agent, including `T` and `nil`. The terminal a may be any primitive action, including the action `nil` that does nothing.

$$\begin{aligned} S &\rightarrow A \\ A &\rightarrow (\texttt{if}\ P\ A\ A) \mid a \\ P &\rightarrow (\texttt{and}\ P\ P) \mid p \end{aligned}$$

The adapted structure makes it so that the strongly-typed genetic programming system only has to deal with two types instead of three. The two types used in this paper are the *action* type and the *perception* type, which corresponds to the variables A and P in the grammar. The grammar allows actions to be either primitive actions or other teleo-reactive programs. The actual terminals used will be discussed in the following section.

3 METHODS

The teleo-reactive tree structures described in the previous section proved easy to implement using the strongly-typed version of LIL-GP made available by Sean Luke.[2] This section explains the setup for the genetic programming experiments. The standard genetic programming tableau appears in Table 1.

Objective:	Find a teleo-reactive program for stacking a specified tower from an arbitrary configuration of blocks.
Terminal set:	*Action type*: `anil`, `mt`, `mb`, `mu`; *Perception type*: `pnil`, `bc`, `nnc`, `nn`, `tbn`, `tbb`, `tcb`.
Function set:	*Action type*: `if`; *Perception type*: `and`, `eq`, `neq`.
Fitness cases:	A random sample of block stacking worlds and target towers.
Raw fitness:	The total number of blocks that were correctly stacked into the target towers.
Standardized fitness:	The sum of all the target tower sizes in the fitness cases minus the raw fitness.
Hits:	The number of target towers that were stacked correctly.
Wrapper:	None.
Parameters:	Population size $M = 50,000$
Success predicate:	An S-expression representing a teleo-reactive program that correctly stacks the blocks in all of the fitness cases (standardized fitness = 0).

Table 1: Genetic programming tableau for teleo-reactive block stacking problem.

Before discussing the functions and terminals, it is important to understand the state that the indexical terminals describe. Each blocks-world problem specifies a target tower and an initial configuration of blocks. The target tower is simply the target that the agent wishes to build, which is a strict ordering of a subset of the blocks. The initial configuration is the state of the world which is initially presented to the agent. The state contains n blocks arranged as a set of columns. An example of an initial state is shown in Figure 2.

3.1 FUNCTIONS AND TERMINALS

As mentioned earlier, the teleo-reactive structures use two types in the strongly-typed genetic programming system. The first type is the *perception* type. The perception type includes the following terminals that correspond to sensors dynamically tracked by the agent: `pnil`, `bc`, `nnc`, `nn`, `tbn`, `tbb`, `tcb`.

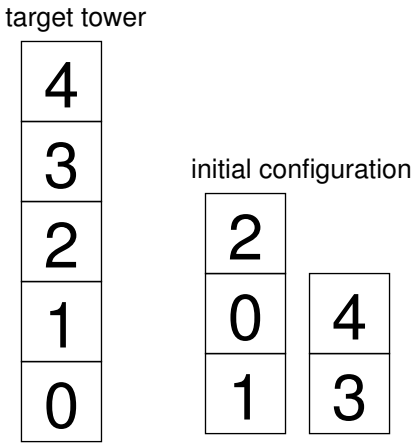

Figure 2: An example of an initial state.

- The sensor `pnil` is always `nil`.
- The sensor `bc` ("Best Column") dynamically specifies the column whose bottom block is the same as the bottom block of the target tower. If the bottom block of the target tower does not appear as the bottom block of any of the columns, the sensor returns `nil`.
- The sensor `nnc` ("Next Needed Column") dynamically specifies the column that contains the next needed block, `nn`. If `nn` is `nil`, then `nnc` is also `nil`.
- The sensor `nn` ("Next Needed Block") dynamically specifies the next needed block to build the target tower. If the target tower has been built successfully, `nn` is `nil`.
- The sensor `tbn` ("Top Block of Next Needed Column") dynamically specifies the top block of the column referenced by `nnc`. If `nnc` is `nil` then `tbn` is also `nil`.
- The sensor `tbb` ("Top Block of the Best Column") dynamically specifies the top block of the best column, `bc`. If `bc` is `nil` then `tbb` is also `nil`.
- The sensor `tcb` ("Top Correct Block") dynamically specifies the topmost block in the best column where that block and every block underneath it is in the ordering specified in the target tower. If `bc` is `nil` then `tcb` is also `nil`.

[2] Sean Luke's strongly-typed kernel is based on the LIL-GP Genetic Programming System from Michigan State University. The source is freely available here: `http://www.cs.umd.edu/~seanl/gp/patched-gp`.

Internally, perception types evaluate to integers, corresponding to the integers identifying the blocks and columns in the state. The terminal `pnil` is given the internal representation of -1. Of course, the actual values of the indexicals are not externally visible and do not appear in the generated trees.

The perception functions are the following: `and`, `eq`, and `neq`. These functions take two arguments that are perceptions and return a perception.

- The function `and` returns `T` if both of its arguments are `T`. Note that the expression `(and (eq nn pnil) tcb)` will always evaluate to `nil` since `tcb` cannot, of course, be `T`.
- The function `eq` returns `T` if its arguments are equal. Otherwise, it returns `nil`.
- The function `neq` returns `T` if its arguments are not equal. Otherwise, it returns `nil`.

The action type has four terminals corresponding to actions to be taken by the agent.

- The action `anil` is the "empty action" that has no effect on the environment.
- The action `mt` ("Move to Table") moves the top block of the next needed column, i.e. `tbn`, to the table. This creates a new column consisting of this single block. If `tbn` is `nil`, then no action is taken.
- The action `mb` ("Move to Best") moves the top block of the next needed column, i.e. `tbn`, to the top of the best column, i.e. `bc`. If `tbn` or `bc` is `nil`, then there is no effect on the environment.
- The action `mu` ("Move Unstack") moves the top block of the best column, i.e. `tbb`, to the table. This creates a new column consisting of this single block. If `tbb` is `nil`, then there is no effect on the environment.

The action type has one function, which is the three-argument `if` function. The first argument is a perception type, and the last two arguments are action types. Before evaluating the last two arguments, the first argument is evaluated. If the first argument evaluates to `T`, then the second argument is evaluated. Otherwise, the third argument is evaluated.

Notice that if the agent ever takes an action that has no effect on the environment, all future actions will be the same and, hence, will have no effect on the environment. The same action will be taken in the teleo-reactive tree because only the agent may change the environment. Therefore, an experiment may terminate as soon as the agent takes an action that has no effect on the environment.

3.2 SAMPLE TREES

The following is a (human-produced) teleo-reactive program that solves the block-stacking problem.

```
                    (eq nn pnil)  →  anil
(and (eq nn tbn) (eq tbb tcb))    →  mb
                    (eq tbb tcb)  →  mt
                     (eq nn tbn)  →  mu
```

In words, if the next needed block is `nil`, meaning that the target tower has been stacked correctly, then the agent stops. If the next needed block is the top block of its column and the top block of the best column is the top correct block, then the agent moves the next needed block to the best column. Otherwise, if the top block of the best column is the top correct block, then the agent moves the top block of the next needed column to the table. Otherwise, the agent moves the top block of the best column to the table. This teleo-reactive program may be converted to the following S-expression:

```
(if (eq nn pnil) anil
    (if (and (eq nn tbn) (eq tbb tcb)) mb
        (if (eq tbb tcb) mt
            (if (eq nn tbn) mu anil))))
```

3.3 EVALUATION OF FITNESS

The most natural way to measure fitness is simply to count the number of blocks that were stacked correctly by the agent by the time the agent decides to stop or within a certain number of operations. Evaluating a teleo-reactive stacking program on a single test case is unlikely to produce a solution to the general block-stacking problem. Therefore, it is best to evaluate the fitness of a particular teleo-reactive program on a collection of test cases, with a variety of target towers and initial configurations with varying numbers of blocks.

The number of initial configurations for a given number of blocks may be extremely large. The number of possible configurations is given by the "sets of lists" sequence, which counts the number of partitions of $\{1, \ldots, n\}$ into any number of lists (Motzkin, 1971). Starting with $n = 2$, the sequence proceeds: 3, 13, 73, 501, 4051, 37633, 394353, 4596553, ..., according to

the recursive formula:

$$a(n) = (2n-1)a(n-1) - (n-1)(n-2)a(n-2)$$

The number of configurations grows extremely quickly. For 18 blocks, there are 588,633,468,315,403,843 possible arrangements.

We rely on random samples of this space for our test cases. To generate a test case with n blocks, we start with a list $(0, \ldots, n-1)$ and then randomize the positions of the elements. We then randomly partition the list into some number of smaller lists. Each list defines the contents of a column in the initial configuration. We then select a random number r from 1 to n and then create the target tower $(1, \ldots, r)$.

Once a collection of test cases is generated, we begin by evaluating the teleo-reactive program on each test-case. The agent is allocated a certain number of steps to solve each problem or to give up. The raw fitness is the sum of the number of blocks stacked correctly in all the test cases. The standardized fitness is simply the sum of all the target tower sizes minus the raw fitness. We also define the number of hits to be the number of problems that were solved by the agent.

3.4 PARAMETERS

Our experiments involve population sizes of 5,000 and 50,000, and the maximum number of generations is set to 50. The "grow" method of generating the initial random population was used with a depth ramp ranging from 5 to 9.

There are four breeding phases. The first breeding phase is function-point crossover (70%), any-point crossover (20%), reproduction (9%), and mutation (1%). These parameters approximately follow the sample parameters that come with Sean Luke's version of LIL-GP, with the addition of mutation. The maximum depth for evolved trees was set to 17.

Many different collections of fitness cases were used to evaluate fitness. Trials were done with the number of blocks ranging from 3 to 100 blocks with up to 100 samples for each block size. The number of steps allocated for solving each problem varied but was kept sufficiently high so that it would not limit the agent in finding a solution. For problems with $n < 8$, the maximum number of steps per problem was set to 250.

4 RESULTS AND DISCUSSION

The experiments were run on a single-processor Pentium III 800MHz workstation, and runtime varied between minutes and hours depending on the collection of fitness cases used. In most cases, completely fit individuals were evolved.

Our first experiment was done with a collection of 10 random block-stacking problems with exactly 3 blocks. An initial population size of 5,000 was used, and an individual that could solve all 10 problems was evolved by generation 21. The individual scored 10 hits and had a raw fitness of 20. The individual is shown below.

```
(if (and (neq pnil tcb)
         (eq nn tbn)) mb mt)
```

This individual is certainly not a solution to the general block-stacking problem; in fact, it does not even correctly stack all possible 3-block problems. For example, consider the initial configuration shown on the left side of Figure 3 and suppose that the target tower is all 3 blocks stacked numerically. The individual will choose the action `mb` since `tcb` $\neq$ `nil` and `nn` = `tbn`. So, the agent will move the next needed block, i.e. block `1`, to the top of of the best column, as shown in Figure 3. The next action taken is again `mb`, which has no effect on the state of the blocks which effectively terminates the teleo-reactive program with an incorrect tower with a raw fitness of 1.

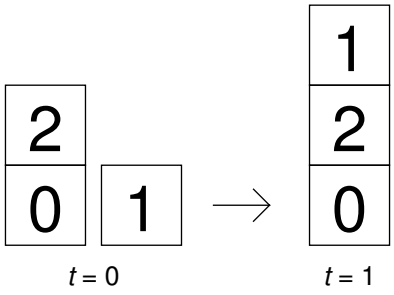

Figure 3: An example of a completely fit individual that cannot solve the general block stacking problem.

Since the fitness cases that were randomly selected for the evaluation of the previous individual were not successful in guiding the evolution of a completely general solution, experiments were done with a wider selection of fitness cases over the space of possible 3-block problems. Instead of 10 samples, 20 samples were selected randomly from the space of $6 \times 13 = 78$ problems. A completely fit individual was not found within 50 generations with an initial population size of 5,000. The raw fitness of the best individual during this run seemed to be stuck at 39 correctly stacked blocks out of a possible 40. However, with a population size of 50,000, a completely fit individual was found in only 3

generations. The individual is shown below.

```
(if (neq tcb tbb)
     (if nn mu mu)
     (if (and (eq tcb pnil)
              (neq tbb nnc)) mt mb))
```

This individual is quite a bit more complex than the completely fit individual in the previous experiment with only ten fitness cases. However, this individual is not a general solution for stacking n blocks. Notice that the individual makes the rather odd comparison `(neq tbb nnc)` between a block and a column. Since the indexicals `tbb` and `nnc` are represented as integers, the comparison is allowed even though the actual number assigned to each column was intended to be abstracted away.

The results of some initial experiments involving between 3 and 5 blocks is summarized in Table 2. It is important to note that each row in the table represents only a single run, and that the maximum number of generations was set to 50.

What we see in Table 2 is that genetic programming with an initial population of 50,000 individuals easily evolved completely fit individuals in relatively few generations. It was considerably more difficult for the population size of 5,000 to evolve completely fit individuals within 50 generations.

Genetic programming was able to find a solution to the general block stacking problem. The run that produced this solution had an initial population size of 50,000 and was evaluated on 575 fitness cases where the number of blocks ranged from 3 to 25. The solution had a raw fitness of 4,193, and the individual is shown below.

```
(if (eq tcb tbb)
     (if (neq (neq tbb tbb) tcb) mb mt) mu)
```

Since `(neq tbb tbb)` always evaluates to `nil`, we may simplify the expression to the following:

```
(if (eq tcb tbb)
     (if (neq pnil tcb) mb mt) mu)
```

Observe that the program above contains the "subprogram" `(if (neq pnil tcb) mb mt)`, which moves the top block of the next needed column to the top of the best column until `tcb` becomes `nil` or when the condition that triggered the program becomes `nil`. The S-expression above is equivalent to the following teleo-reactive program:

$$\begin{aligned} (\texttt{neq tcb tbb}) &\rightarrow \texttt{mu} \\ (\texttt{eq tcb pnil}) &\rightarrow \texttt{mt} \\ (\texttt{neq tcb pnil}) &\rightarrow \texttt{mb} \end{aligned}$$

This completely general solution begins by unstacking the tower containing the first block in the target tower. It then moves the top block from the top of the column containing the next needed block to the top of the best column. If this block is not the next needed block, then it is then moved to the table.

This evolved program is simpler than the human-programmed solution discussed earlier. However, this teleo-reactive program does not necessarily stack a target tower in the minimum number of moves. Instead of moving a block that is not the next needed block from the next needed column immediately to the table, the individual moves the block to the top of the best column and then to the table. This minor loss in efficiency is illustrated in Figure 4.

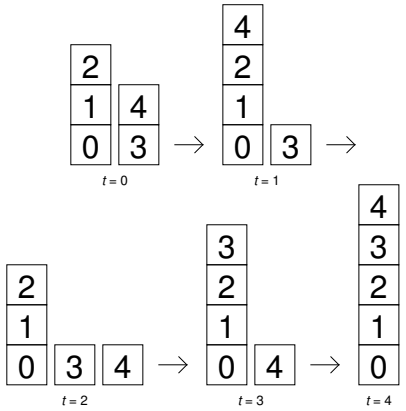

Figure 4: An example of inefficiency in an evolved general solution.

Since there was no selective pressure to evolve an optimal solution, it is not surprising that the first evolved solution is suboptimal. To encourage optimality, the number of steps allocated to finding a solution was reduced to twice the number of blocks, since in the worst case the optimal program would have to unstack all the blocks to the table and then move them individually to the best column. A suboptimal, but correct program would simply run out of time before stacking all of the blocks. Of course, this method to assign fitness values could assign a completely correct, but suboptimal individual the same fitness value as an individual that cannot solve arbitrary stacking problems.

The time constrained evaluation function described

3 blocks	Population size: 5,000				Population size: 50,000			
Num cases	Generation	Hits	Fitness	Solved	Generation	Hits	Fitness	Solved
10	21	10	20	Yes	1	10	20	Yes
20	36	19	39	No	3	20	40	Yes
30	7	25	52	No	5	30	60	Yes
40	7	34	77	No	1	40	86	Yes
50	2	50	105	Yes	2	50	105	Yes
4 blocks	Population size: 5,000				Population size: 50,000			
Num cases	Generation	Hits	Fitness	Solved	Generation	Hits	Fitness	Solved
10	13	14	22	No	4	10	25	Yes
20	24	17	45	No	2	20	50	Yes
30	5	30	81	Yes	1	30	81	Yes
40	2	40	105	Yes	5	40	105	Yes
50	4	11	121	Yes	2	50	121	Yes
5 blocks	Population size: 5,000				Population size: 50,000			
Num cases	Generation	Hits	Fitness	Solved	Generation	Hits	Fitness	Solved
10	2	10	28	Yes	1	10	28	Yes
20	49	18	59	No	4	20	63	Yes
30	21	27	84	No	2	30	91	Yes
40	18	35	112	No	3	40	123	Yes
50	7	50	156	Yes	1	50	156	Yes

Table 2: Results from experiments with 3, 4, and 5 blocks. The maximum number of generations is 50.

above was used on a population of 10,000 randomly generated individuals. To encourage simpler trees, the initial population was grown with a depth ranging from 1 to 5, and the maximum depth was set to 5. The first run produced a correct and optimal teleo-reactive program by generation 8. The individual is shown below.

```
(if (neq tbb tcb) mu
    (if (and (neq pnil bc)
             (eq tbn nn))
        (if nnc mu mb) mt))
```

Since `nnc` always evaluates to `nil`, we may simplify the expression to the following.

```
(if (neq tbb tcb) mu
    (if (and (neq bc pnil) (eq tbn nn))
        mb mt))
```

The expression above is equivalent to the following teleo-reactive program:

$$\begin{aligned} \texttt{(neq tbb tcb)} &\rightarrow \texttt{mu} \\ \texttt{(and (neq bc pnil) (eq tbn nn))} &\rightarrow \texttt{mb} \\ \texttt{T} &\rightarrow \texttt{mt} \end{aligned}$$

Even without the simplification, the evolved solution consists of only 18 points, which is remarkable since the human-programmed solution consisted of 25 points.

5 CONCLUSIONS

This paper has demonstrated how genetic programming can be used to evolve teleo-reactive programs. Teleo-reactive programs may be represented as S-expressions that are then reproduced according to fitness and recombined through the standard strongly-typed genetic programming procedures. We have seen specifically how genetic programming is capable of evolving block-stacking teleo-reactive programs with indexical terminals.

The standard genetic programming techniques evolved a completely general and optimal solution to the block-stacking problem for an arbitrary number of blocks with any initial configuration. The evolved program is simpler (in number of points) than the one produced by a human programmer. This paper shows that it is important to have a wide selection of fitness cases with varying numbers of blocks along a sufficiently large population. If optimality is a concern, then evolutionary pressure may be applied by imposing a time constraint.

It is remarkable how a few hundred fitness cases selected randomly from the extremely vast state space can guide genetic programming to evolve an optimal and general plan for stacking blocks. Of course, much of its success depended upon the availability of the appropriate indexicals. Baum and Durdanovic (2000) tried solving a related block-stacking problem without the use of genetic programming or indexicals and found that only about 4 block instances could be

solved. The preliminary results presented in this paper indicate that genetic programming is well suited for learning teleo-reactive programs.

6 FURTHER WORK

Further work will be done using genetic programming to evolve teleo-reactive programs. Certainly, it would be interesting to see if genetic programming can evolve teleo-reactive programs for use in other, more complex domains. However, there is still much to be done with evolving teleo-reactive programs for block stacking.

Indexicals are partially responsible for the success in evolving general solutions to the block-stacking problem. It would be interesting to investigate how to evolve general solutions without the use of indexicals. The teleo-reactive program that is included in Nilsson's paper (2001) does not use indexicals, but uses recursion. Further research is necessary to examine how to allow genetic programming to handle this sort of recursion gracefully. It would be fascinating to see whether genetic programming can evolve higher-order perceptions and actions automatically through the use of ADFs.

Acknowledgments

I would like to thank Nils Nilsson and John Koza for their suggestions and encouragement.

References

E. Baum and I. Durdanovic (2000). Evolution of Cooperative Problem-Solving in an Artificial Economy. Submitted, available from `http://www.neci.nec.com/homepages/eric`.

S. Benson (1996). Learning Action Models for Reactive Autonomous Agents. Ph.D. Thesis, Department of Computer Science, Stanford University.

J. R. Koza (1992). *Genetic Programming: On the Programming of Computers by Means of Natural Selection.* Cambridge, Massachusetts: MIT Press.

T. S. Motzkin (1971). Sorting numbers for cylinders and other classification numbers. In *Combinatorics, Proceedings of Symposia in Pure Mathematics*, vol. 19, 167–176. The American Mathematical Society.

N. Nilsson (1992). Toward Agent Programs with Circuit Semantics. Technical Report STAN-CS-92-1412, Department of Computer Science, Stanford University.

N. Nilsson (1994). Teleo-Reactive Programs for Agent Control. *Journal of Artificial Intelligence Research* **1**:139–158.

N. Nilsson (2000). Learning Strategies for Mid-Level Robot Control: Some Preliminary Considerations and Results. Unpublished memo, Department of Computer Science, Stanford University, May 11, 2000.

N. Nilsson (2001). Teleo-Reactive Programs and the Triple-Tower Architecture. *Electronic Transactions on Artificial Intelligence* **5**:99–110.

N. Nilsson (2002). Genetic Programming and Teleo-Reactive Programs: Rough Notes. Unpublished memo, Department of Computer Science, Stanford University, April 22, 2002.

J. Perry (1996). Indexicals. In *The Encyclopedia of Philosophy, Supplement.* New York: Simon and Schuster Macmillan, 257–258.