Development of Block-Stacking Teleo-Reactive Programs Using Genetic Programming

Praveen Srinivasan (praveens@stanford.edu)
Department of Computer Science, Stanford University
Stanford, California - 94305

Abstract:

This paper describes the development of Teleo-Reactive (T-R) block-stacking programs using genetic programming. T-R programs are a class of programs that are also a specific form of *k*-decision lists, and are useful for programming tasks for autonomous agents. Using only the predicates of ***On***, a test to see if one block is atop another, and ***Move,*** moving one block to another's column, genetic programming was able to evolve programs capable of stacking 4 blocks in a predefined order for as many as 500 different randomly generated fitness cases. Programs for 5 and 6 block situations were also successfully developed, but attempts to create programs satisfying the desireable regression property were largely unsuccessful. Rewarding programs based on performing the least number of moves after perfectly stacking blocks in all fitness cases also had limited success, yielding an individual likely generated by random chance and not by evolution.

Introduction:

Teleo-Reactive (T-R) programs are a special form of *k*-decision lists, and are an important part of control theory. In essence, a set of conditions are continuously evaluated, and actions are performed based on these, eventually leading to the completion of a particular goal. One particular way to view T-R programs is as a highly structured LISP expression of the following general form:

(if *some condition* **take action 1** ***else***
(if *another condition* **take action 2** ***else***
....
(if T **take action n** ***else do nothing***))...)).

When the top action has been executed, the program's execution is considered complete, with the expectation that the desired goal has been reached. The benefit of this control structure is that due to the constant evaluation, a programmer need not worry about being needing to return from a subroutine in the event that conditions change; instead, since the entire expression tree is being constantly reevaluated, he or she need only specify associated conditions and actions and the hierarchy regarding them. One particularly desirable property that T-R programs may have is that of *regression property*, or after an action in the tree is performed, the condition of an ***If*** statement higher in the tree becomes true, essentially allowing execution to climb the tree constantly until the program terminates. This property has the advantage that the execution of the programs is well defined, and the strategy of the program is easily understood from examining it (Nilsson 2001).

T-R programs in general have a wide variety of applications, although use of them in practice has been limited by the general novelty of this type of program. Most automated tasks, such as those performed by agents in a changing environment, can be encoded in terms of T-R programs. Hence, this class of programs is of interest for many real-world robotics applications. However, for the purposes of research one particular problem that lends itself to the application of T-R programs is block-stacking. In a blocks world, there are a certain number of blocks

which can be placed on a infinitely large table or arbitrarily stacked on each other. The goal is then to stack these blocks in a particular order, using an ***On*** predicate of whether or not a block is on another block or the table, e.g. if ***On***(A,B) evaluates to true, then block A is on block B. In order to realize the goal, the program can execute a ***Move*** command, e.g. ***Move***(A,B) would place block A upon block B (or the top of the column B is in if there are blocks on B).

In order to develop T-R programs, either a programmer must write one or a computer can learn one via a variety of learning methods. Genetic programming (GP) is a particularly interesting and appealing approach due to its ability to evaluate a large and diverse set of candidate programs in an efficient and effective manner. The learning of a program to stack blocks via GP has been done before, such as in work by Koza, 1992. In his book *Genetic Programming*, he describes the evolution of a program to stack blocks using higher-level predicates known as indexicals, or predicates which indicate which block should be placed next in the column being formed. Also, the place to stack the blocks was predefined while in the present work the program is free to stack the blocks in whichever column it finds most appropriate, making the task slightly more difficult. While using GP to learn the task with indexicals as predicates turns out to be very effective, learning the same task using only the ***On*** predicate and T-R program structure is somewhat more difficult. Although learning with these lower order predicates is more difficult, the resulting programs operate at a level that is more similar to what an agentmay perceive in the environment in a sensory context and hence may translate better to real-world applications.

Further complicating the problem is the large size of the problem search space, which is also another major reason the use of GP is attractive. For a given depth n (of ***If*** statements), we can have at each level one of approximately $(b+1)^2$ (where b is the number of blocks and we add one more for the table) different ***Move*** operations (actions). For each condition for an If-statement, we can have very complex statements, certainly utilizing all approximately $(b+1)^2$ possible ***On*** predicates and many combinations of these using the Boolean functions available, ***And***, ***Or*** and ***Not***. Using just one ***And*** or ***Or*** with ***On*** predicates as children squares the number of possible combinations, since these two Boolean operators require two arguments. With the use of just one ***And*** in each condition for all n ***If*** statements, we have $n*((b+1)^2)^3$ different possible statements. For n=10 and b=4, the result is over 156,000 different possible programs. Adding to the conditions for each ***If*** and increasing the tree depth increases the size of the search space greatly, making the number of possible programs easily in the millions. Certainly, increasing the number of blocks would also greatly increase the search space as well, as the search space is proportional to the sixth power of the number of blocks just in the limited situation described. The genetic operators of crossover and point mutation do indeed provide access to the entire space, however, allowing any desired T-R program to be formed.

Methodology:

Tableau 1 represents the important features of this problem and the approach used to solve it. As discussed, the goal was to stack blocks given the functions of ***Move***, ***On, And, Or, Not*** and ***If***, with terminals of blocks, NullIf (which has no child ***If*** statement, hence all programs use this to show the end of their programs), NullAction, and the Boolean values true (T) and false (F). The fitness cases were randomly generated because random generation of block configurations is much easier and provides a smaller set of cases than enumerating all possible worlds, and it provides a reasonable situation that an agent might encounter in the real world. It is unlikely that such an agent will encounter every single configuration of blocks, but rather a random subset of them. Raw fitness was chosen as described in *Genetic Programming* for their similar block-stacking problem (Koza 1992). Specifically, the raw fitness is calculated by iterating over the columns and choosing the column with the most number of blocks in the correct position in the stack in that column. The value of the number of blocks in the correct position in that column is then added to a running total for all the fitness cases, and the resulting sum after evaluation represents the raw fitness. Standardized fitness follows from raw fitness, and normalizes raw fitness on a scale of 0 to 1, where 1 is represents an individual which stacks blocks perfectly in the fitness cases, and 0 represents an individual which stacks blocks in such a bad way that non are in the correct order as described in the

fitness function. The parameters 'Hits' is recorded as the number of fitness cases where the individual stacks the blocks such that the resulting tower is in exactly the desired order. An individual which does this for all fitness cases will have a raw fitness of 0, which also serves as a termination criteria, along with the upper bound of 2000 generations, chosen simply because it represents a great deal of time in which to evolve a solution. Experience showed that runs that did not finish after that long or even after 1000 generations were indicative of a bug or other problem in the program. Finally, the population size was chosen to be 2000 since that size provides good results and still runs through generations relatively fast, usually at most 5 seconds a generation. Overall, runs tended to take at most 10 minutes.

The tableau was implemented using a Java-based software package for GP known as ECJ, available online at <http://www.cs.umd.edu/projects/plus/ec/ecj/>. All the parameters used for the various runs were in the simple.params and koza.params files provided with ECJ, with the exception of population size (2000), and number of generations until termination (2000). The population size was chosen as the largest practical size, as larger populations resulted in very slow evaluation/breeding and had memory usage issues. Blocks were coded as ephemeral random constants, thereby allowing point mutation of the blocks themselves. The initial population was generated using the default GrowBuilder class, which randomly chooses a tree depth and then builds a random tree of that depth, randomly choosing appropriate functions and terminals in the process. Crossing over and mutation used the default classes of CrossoverPipeline and MutationPipeline, and the probabilities for these are as specified in the koza.params and simple.params files. Worlds were randomly generated by placing blocks in an two-dimensional grid, and then "compressing" blocks such that all blocks either rested at the bottom or on other blocks. One important deviation from traditional methods was that the ***If*** statements were coded slightly differently than expected. The condition of the statement would be evaluated, and if found to be true, the associated action would be executed. However, instead of simply returning, the interpreter would also evaluate the child ***If*** statement. Similarly, if the condition evaluated to false in the first place, the child ***If*** statement would be evaluated, and after it returned the associated condition would be reevaluated and the action executed if the result was true, otherwise the statement would return. Traditional evaluation of these ***If*** statements results in an enforcement of the regression property, which while highly desirable, was never achieved in the present work, and forcing execution of ***If*** statements to be in the traditional manner greatly hampered learning efforts, stunting the best individual at solving only ~10% of the provided fitness cases and not improving over even hundreds of generations. Therefore this modified, but still well-defined execution of ***If*** statements was used. It is important to note that this constant reevaluation of conditions is in many ways the "spirit" of T-R programs, and the modified execution of ***If*** statements reflects this.

Results:

Utilizing this tableau and implementation with ECJ, GP was able to develop programs capable of solving even hundreds of different fitness cases. However, for the purposes of illustration, the following example of the evolution of a T-R program was done with a set of fitness cases consisting of only 2 cases, since examining the evolution of a program with hundreds of fitness situations is rather difficult and complex. The following example given here has one notable deviation from the tableau, however in that the fitness measure also attempts to reward a program for following the regression property and penalize it for deviating from it. Specifically, when an ***If*** statement executes its actions, execution continues down the tree looking for other true conditions and performs the associated actions. If after performing the action at one level, no actions are performed at lower levels, then the program's fitness is incremented by 1. If the reverse is true, the program's fitness is decremented by 1. This modification to the fitness, although unsuccessful, results in the process taking 3 generations to develop a solution to the listed test cases. Without rewards and penalties for the regression property, an individual is found in generation 1, even with as a few as 20 individuals in the population. Because the development of suitable individuals is so fast with small and easy to understand problems such as this one, adding this constraint and hence lengthening the process as a side effect also makes it a good example for understanding the evolution of these

block-stacking T-R programs.

Example Result:

In Table 1, the table of fitness cases, each row represents a separate fitness case, specifying the coordinates of each block in the blocks world. “Inherent fitness” indicates the fitness value of the initial configuration of blocks. An individual program which does nothing would have fitness 2*4-3-2=3 (2*4 represents 2 cases times a maximum fitness of 4 blocks when perfectly stacked, and then we subtract the inherent fitness values), as calculated from the fitness measure in the tableau. In the programs, the block '-1' represents the infinite table, which can have all the blocks upon it simultaneously if need be.

The evolution of the program is shown in Program Listing 1. In generation 1, the top program is unable to solve either of the two problems (the “”List of problems solved:” is empty) since it never moves block 3, a step which is essential for a solution to either problem. Interestingly, the result of the program's execution is that both worlds end up the same as before the program acted upon them. While ordinarily the resulting fitness would be 3, the program is rewarded for satisfying the regression property in problem 1 since execution of the action of the 2nd ***If*** results in the condition of the1st ***If*** becoming true. In generation 2, the program correctly moves block 3 to the column block 2 is in, which solves problem 0, but does not change the world in problem 1 resulting in a fitness of 2 (it breaks even on the regression property). Finally, in generation 3, the best individual is able to solve both problems, although somewhat inefficiently, as it moves 2 to the table, then moves to the column block 1 is in, and finally moves block 3 to the column block 0 is in. Since 1 and 0 remain in the same place, the program correctly stacks the blocks. Concomitant with the increasing block-stacking ability of the top individual, the fitness also increases, going from 2 and 0 hits to 2 and 1 hit to 0 and 2 hits. As noted earlier, while GP correctly found a solution to this problem, in more complex situations the modification to the fitness to encourage development of the regression property greatly hindered the development of programs. The most fit individual after even hundreds of generations was only capable of solving approximately 10% of the given fitness cases, and hence this modification was removed for further testing.

Fortunately, using the modified execution of ***If*** functions as described in the example, T-R programs were generated that could solve problems containing over 300 randomly generated different situations in 104 generations. Program Listing 2 is the best individual produced by a run with 350 randomly generated fitness cases, and was able to stack the blocks correctly in all of the given situations. While this program is seemingly complex, the basic approach used by this program is to move blocks to the table until the 0 block is free, then pile them back on. Snippets such as (If (On 3 -1) (Move 0 -1) ...), which is probably used in block configurations when block 3 was on top of block 0 in a stack, and (If (Not F) Move(0 -1) ...), which always moves block 0 to the table in order to stack more blocks on top of it, are indicative of this. This approach is quite logical, since the movement of the blocks to table reduces all problems to a single base case that is easily solved by piling them on into a column in the correct order. Runs using similarly sized fitness sets for 5 and 6-block worlds also succeeded in producing individuals capable of stacking blocks in all given configurations, and were of similar form.

More specifically, the clear evolution of the program is evident in Chart 1, which shows the progressive decrease (and hence improvement) in raw fitness, both of the top individual of each generation and the overall mean raw fitness, leading to an individual with a raw fitness of 0, or capable of stacking blocks correctly in all the situations. Interestingly, by G=23 the top individual was capable of solving the vast majority of problems, but the GP process required another 81 generations to produce an individual with perfect fitness. This falloff in improvement in fitness was characteristic of almost all runs, and can likely be attributed to individuals finding solutions to easy problems early on, then having to modify their programs to be able to solve a few more difficult problems (or problems which simply were not solved early on), while retaining the ability to solve the previous problems. Therefore, for the sake of efficiency and time programs can be quickly evolved that can stack blocks correctly in the vast majority of cases, without needing to spend an extended amount of time developing a perfect solution. This result may make possible the use of GP for online development of T-R programs in response to a changing environment, particularly if the number of fitness cases is somewhat smaller. Finally, one last result was

that an attempt to optimize the execution of the individuals by rewarding programs for making fewer moves was marginally successful. In one run, the number of movements was reduced from approximately 530 to 450. However, the change occurred over just one generation, suggesting that merely random chance, as opposed to actual evolution, produced the individual.

Conclusions:

GP was able to evolve T-R programs in the form of high structured LISP expressions suitable for solving a large number of block-stacking situations. As show in the example and visible in all runs, GP consistently improved both the raw fitness of top individual in each generation and the mean raw fitness until the best individual was able to stack blocks correctly in all of the test situations. Even given nearly 300 different randomly generated situations at a time, GP was able to develop a suitable program in approximately 100 generations, and was able to develop these programs for 4, 5 and 6 block configurations, and likely more with the only limit being computational power. Individuals evolved quickly (approximately G=20) to solve the vast majority of all the fitness cases, and spent a significantly longer period (as much as 80 generations longer) evolving into a individual capable of solving all the problems. Modifying the fitness function to improve performance had a limited effect, as the number of movements performed by individuals shifted by a large amount from individual to individual, and hence it is difficult to produce a fitness function which leads to a smooth progression in fitness values based on efficiency. Also, attempts to develop a T-R program which statisfied the desirable regression property were less sucessful, and the best results were achieved when using the modified ***If*** function that allowed for the inspection of conditions further down in the tree.

Further Work:

One clear area where further work is needed is in creating a program which obeys the regression property. While the programs generated here are still useful as their execution is well-defined (even though the ***If*** functions are executed somewhat unconventionally), programs which obey the regression property are still highly desireable and their structure can be exploited in interesting ways. For example, a method known as SQUISH has been developed in order to merge T-R programs based on the steps to the goal they share in common (John 1994). This merging of T-R programs can lead to space reduction and improved evaluation performance, but the process requires that the programs to be merged satisfy the regression property. Also, programs which satisfy the regression party are easier to read and make it much easier to understand exactly how the program operates. Another useful refinement to the current programs would be the institution of a fitness measure which also rewards programs which not only stack blocks correctly but also do so with the least number of ***Move*** operations possible.

Extensions to the current work might include the use of higher-level predicates, such as perhaps a more powerful ***Move*** operation which would automatically unpile blocks on top of the source and destination blocks, or perhaps the introduction of a NextBlock indexical function which would suggest the next block to use, based on a predetermined heuristic. As in Koza's block-stacking program, an iterative do until procedure combined with these indexicals may also yield good results. Other interesting extensions may include more complex structures than a simple tower, such as two towers, or even a goal structure which varied depending on the positions of the initial blocks, such as if block 1 was on 2 to start with, a tower should be built as normal except that in the tower, 1 should be on 2 instead of the normal reverse configuration. Multiple agent approaches to stacking blocks are also appealing, and GP co-evolution techniques are likely to be very effective in this regard. More radical changes may include doing away with block-stacking altogether and using a problem which is not so discrete in its operation, such as some sort of movement of items in a 3-dimensional space, or a problem where another agent intervenes in a opposing manner. Certainly, there are a large number of different problems to pursue in developing T-R programs with GP, and the results of the present work shows that this approach has great promise in this regard.

Acknowledgements:

I would like to thank John Koza for his excellent class on genetic programming at Stanford University and Nils Nilsson for his help in understanding T-R programs.

References:

1. Nilsson, Nils J. 2001. Teleo-Reactive Programs and the Tripe-Tower Architecture. Electronic Transactions on Artificial Intelligence. 5(B): 99-110.

2. Koza, John R. 1992. Genetic Programming. Cambridge, Massachusetts: The MIT Press.
459-470

3. John, G. 1994. SQUISH: A Preprocessing Method for Supervised Learning of T-R Trees From Solution Paths. Draft Memo, Stanford Computer Science Department.

Objective:	Develop a T-R program which given a set of blocks in a variety of configurations, is able to stack them in a fixed, predetermined order.
Terminal Set:	Block (a particular block, e.g. block 1), NullAction (don't do anything), T, F, NullIf (ends a T-R program)
Function set:	On(Block a, Block b), Move(Block a, Block b), If (condition1, action, child If), And, Or, Not
Fitness cases:	A set of configurations of blocks, usually randomly generated and numbering in the hundreds.
Raw fitness:	The number of blocks stacked in their correct final positions throughout all the fitness cases.
Standardized fitness:	(raw fitness)/((# of configs.)*(#of blocks))
Hits:	Number of configurations where the blocks were perfectly stacked in.
Population size:	2000 (Increasing the population beyond this increases memory usage too much)
Termination:	Maximum number of generations G=2000. Also terminates if a program is able to stack blocks correctly in all configurations.

Tableau 1: Tableau for block-stacking problem.

Problem #	***Block 0***	***Block 1***	***Block 2***	***Block 3***	***Inherent Fitness***
0	[0,0]	[0,1]	[0,2]	[1,0]	3
1	[0,0]	[0,1]	[1,1]	[1,0]	2

Table 1: This table shows the two fitness cases used in a particular run of GP. For each problem, the coordinates representing the initial locations are shown for each block, and the inherent fitness is the fitness value of the initial configuration using the fitness measure described in the tableau.

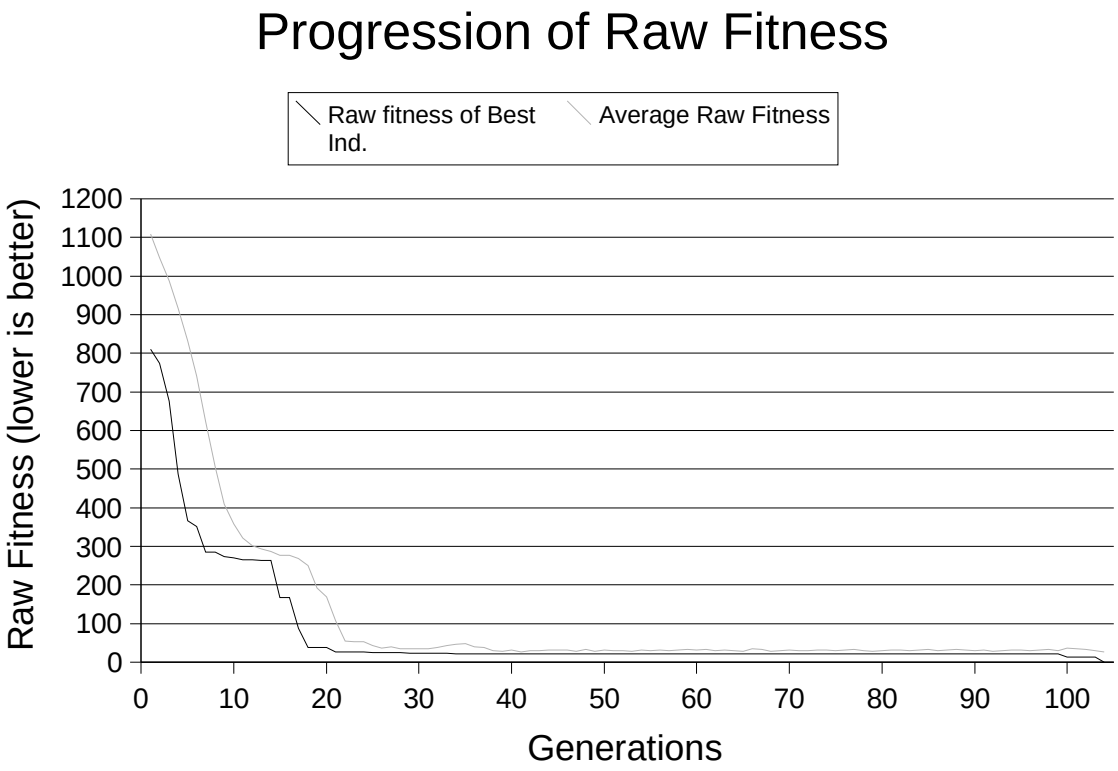

Chart 1: This chart demonstrates the improvement in raw fitness for individuals with respect to the 350 fitness cases from a 4-block world. At generation 104, the fitness reaches 0, indicating that an individual capable of stacking blocks correctly in all fitness cases has been created.

```
Generation 1
================
Subpopulation 0
---------------
Mean fitness raw: 3.4475 adjusted: 0.24028526 hits: 0.0305
Best Individual of Generation:
Evaluated: true
Fitness: Raw=2.0 Adjusted=0.33333334 Hits=0
Tree 0:
 (If (On 2 1) (Move 2 3) (If (Not (On 2 -1))
    (Move 2 0) (If (Or F F) (Move 2 -1) (If T
    noAction null))))
List of problems solved:

Generation 2
================
Subpopulation 0
---------------
Mean fitness raw: 3.0945 adjusted: 0.25361463 hits: 0.0545
Best Individual of Generation:
Evaluated: true
Fitness: Raw=2.0 Adjusted=0.33333334 Hits=1
Tree 0:
 (If (On 3 0) (Move 3 2) (If (Or (On 2 1)
    (And F (Or (And (Not F) (And F F)) (Not (Or
       T F))))) (Move 3 1) (If (Not T) (Move 1 1)
    (If F noAction null))))
List of problems solved:
0

Generation 3
================
Subpopulation 0
---------------
Mean fitness raw: 2.8745 adjusted: 0.27779275 hits: 0.2345
Best Individual of Generation:
Evaluated: true
Fitness: Raw=0.0 Adjusted=1.0 Hits=2
Tree 0:
 (If (Not (On 3 2)) (Move 2 -1) (If (Or (Or
    (Or T F) (And T F)) (On -1 2)) (Move 2 0)
    (If (On 2 1) (Move 3 0) (If T noAction null))))
List of problems solved:
0 1
```

Program Listing 1: This listing shows the evolution of a simple program, evaluated on only two fitness cases. Each generation, the program improves in fitness and hits until a perfect individual is reached that is capable of stacking the blocks correctly in both fitness cases.

```
(If (Or (And T T) (And (On -1 1) (Not (And
    T F)))) (Move 3 -1) (If (Not F) (Move 2 -1)
    (If (And (Or T T) (Not F)) (Move 0 -1) (If
       (Not F) (Move 1 -1) (If (Or (And T T) (And
       (On -1 1) (Not (Not F)))) (Move 3 -1) (If
       (On 3 -1) (Move 0 -1) (If (And (Or T T) (Not
       F)) (Move 2 1) (If (Not F) (Move 3 -1) (If
       (And (And (On -1 1) (Not (And T (And (Or
          (And T T) (Or (Not F) F)) (And (And T F)
          (And T T)))))) (Not F)) (Move 3 3) (If (Not
       F) (Move 0 -1) (If (Not (On -1 1)) (Move
       2 -1) (If (On 0 -1) (Move 1 0) (If (On -1
       0) (Move 0 3) (If (Not (Or (Not F) (Not F)))
       (Move 3 2) (If (On 3 -1) (Move 0 -1) (If
       (And (Or T T) (Not F)) (Move 3 -1) (If (Not
       F) (Move 1 0) (If (Or (And T T) (And (On
       -1 1) (Not (And T F)))) (Move 2 1) (If (And
       (On -1 1) (And (And T F) (And T T))) (Move
       0 -1) (If (Or (Not F) (Not F)) (Move 3 2)
       (If (Not F) (Move 3 2) (If F noAction null))))))))))))))))))))))
```

Program Listing 2: This is the individual produced by a run with 350 fitness cases for the 4-block problem. This individual was produced in 104 generations and can correctly stack the blocks in all 350 cases. See Chart 1 for a chart demonstrating the improvement in fitness throughout the run.