# 10 Evolving Recusive Programs for Tree Search

**Scott Brave**

This article compares basic genetic programming, genetic programming with automatically defined functions (ADFs), and genetic programming with ADFs using a restricted form of recursion on a planning problem involving tree search. The results show that evolution of a recursive program is possible and further that, of the three techniques explored, genetic programming with recursive ADFs performs the best for the tree search problem. Additionally, genetic programming using ADFs (recursive and non-recursive) outperforms genetic programming without ADFs. The scalability of these techniques is also investigated. The computational effort required to reach a solution using ADFs with recursion is shown to remain essentially constant with world size, while genetic programming with non-recursive ADFs scales linearly at best, and basic genetic programming scales exponentially. Finally, many solutions were found using genetic programming with recursive ADFs which generalized to any world size.

## 10.1 Introduction

Iterative structures are an essential component in programming. Iteration and recursion lead to more compact programs and, in many cases, facilitate generalization. Since genetic programming is a genetic algorithm which evolves programs, iterative and recursive capability is desirable. Additionally, the ability to create simpler programs means that an evolved solution may take less computational effort to find.

The evolution of iteration and recursion using genetic programming was first explored by Koza[1992]. To allow the evolution of iterative solutions to the block stacking problem, Koza introduced the Do Until (DU) operator. The DU operator takes two arguments: the first specifies the work to be iteratively executed, and the second specifies a predicate which when satisfied will end the iteration. To prevent infinite loops, an upper limit on the number of iterations is externally imposed. Variations of this form of restricted iteration have proven useful in several problem domains including protein sequencing and optical character recognition [Koza 1994], [Andre 1994a], [chapter 8].

Koza[1992] also investigates a very limited form of recursion for sequence induction. Programs capable of producing the Fibonacci sequence are achieved by allowing the S-expression to reference previously computed values in the sequence. Although much work has been done in the synthesis of recursive functional programs using other machine learning methods (see [Banerjee 1987], [Hutchinson 1994]), little has been done to explore the evolution of recursion using genetic programming (see [Nordin 1995]).

This article investigates a method for evolving recursive programs for tree search. Many techniques used in AI systems, including those for game-playing and planning, depend on an agent's ability to search tree structures [Nilsson 1971], but are not optimal. These systems are thus prime candidates for genetic programming. Similar to restricted iteration, the proposed method includes a finite limit on the depth of recursive calls to avoid infinite (and near infinite) loops. Such a procedure is often undesirable because any limit we impose will affect the evolution and thus the solutions produced by the genetic program. In the case of tree search, however, if the program is doing anything but simply looping at a single node in the tree, it will search and reach the base case within $d$ recursive calls (where $d$ = tree depth), assuming a tree with no parent pointers. We can therefore stop any recursion after $d$ iterations without affecting the behavior of the program. In fact, we can set the limit to any finite number greater than $d$ without affect.

The proposed method also takes advantage of the base case inherent in tree search. This base case occurs when attempting to search past the leaves of the tree. The functions which would normally cause the program to look deeper into the tree can do nothing if we are currently looking at a leaf node, alleviating the need to evolve an explicit base case.

## 10.2 Problem Specifications

A simple planning problem is used to investigate the evolution of recursive tree search. The agent to evolve acts in a world represented by a full binary tree. Starting at the root of the tree, the agent must successfully navigate through the world to reach a specified goal at one of the leaf nodes. Movement, however, can only proceed in the downward directions (to deeper levels of the tree), forcing the agent to make no mistakes on the path to a goal. To accomplish this, the agent is given access to a memory which is isomorphic to the world [Andre 1994b], [Teller 1994]. The agent can read and write to any node and must use this information to make correct movement decisions. All memory locations start with a value of 1. To allow recognition of the goal however, reading the memory location corresponding to the goal node always returns a value of 0. The evolved program is evaluated multiple times with the goal at each of the leaf nodes in succession. The fitness measure is the agent's distance away from the goal averaged over these $2^{depth}$ runs (distance is measured as the minimum number of arcs we would need to follow to reach the goal). A small penalty for inefficient searching of the tree is also added and will be discussed later. Although searching the memory space is essentially free, any actual movement in the world is irreversible. The success of an individual thus reflects its ability to create and follow a mental plan for reaching an arbitrary goal (for a more in depth discussion of mental models, see [Andre 1994b], [Holland 1986], and [Teller 1994]). The experiments in this article use tree depths of 3, 4, 5, 6, and 7.

**Table 10.1**
Function set

| Function | Arity | Action |
|---|---|---|
| `LOOK_LEFT` | 1 | Moves the *look pointer* to the left child of the node it is currently pointing to, executes the argument, then restores the previous l*ook pointer* location. Returns the value of its argument. |
| `LOOK_RIGHT` | 1 | Same as `LOOK_LEFT`, but moves the *look pointer* to the right child. |
| `READ` | 0 | Returns the value of the node which the l*ook pointer* is pointing to. |
| `WRITE` | 1 | Writes the value returned by its arguments to the node which the *look pointer* is pointing to and returns the value written. |
| `MIN` | 2 | Evaluates its two arguments and returns the minimum value. |
| `GO_MIN` | 0 | Moves the *location pointer* to the child of the node it is currently pointing to which has the lower value written to it and returns that value. If the two child nodes have equal value, the l*ocation pointer* moves to the right child. |
| `PROGN` | 2 | Evaluates its two arguments and returns the value of the second. |

## 10.3 Basic Function Set

Table 10.1 shows the function set for this problem. The agent has two pointers to nodes in the tree world: a *location pointer* which points to the node it is physically at, and a *look pointer* which points to the node the agent is currently looking at. Two `LOOK` functions allow the agent to "look around" in its memory. `LOOK_LEFT` and `LOOK_RIGHT` move the *look pointer* to the respective child of the node that is currently being looked at before evaluating is argument (if no child exists, the pointer does not move). After its argument is evaluated, the *look pointer* is restored to its previous value (the value before the `LOOK` function was called).

The `READ` and `WRITE` functions, along with `MIN` (which returns the minimum of its two arguments) allow the agent to manipulate its memory. The `READ` function returns the value in memory at the current *look pointer* location, and the `WRITE` function writes the value returned by its argument into memory at the current *look pointer* location. The value returned by the `WRITE` function is simply the value written. Although the goal node in memory can be written to, `READ`ing this memory node will always return a 0. This is done to allow the agent to recognize the goal state without giving it an explicit `IF_GOAL` function. This simplification does not give the agent too much information however, since the following `GO_MIN` function uses the actual value of the node as specified by the agent. Again, all nodes start with a value of 1.

For further simplification, `GO_MIN` is the only function which will cause the agent to physically move in its world. Calling `GO_MIN` will move the agent (*location pointer*) to the child node of its current location which has a lower value associated with it in memory (defaulting to the right). If this function is called when no child exists, the agent does not move. The fact that GO_MIN only allows movement down the tree (to deeper levels) is critical, as mentioned before. This forces some mental representation of a the desired path since all physical movement in the world is irreversible.

An example of an evolved solution in the very simple 2-deep world is shown in Figure 1. This individual reaches the goal in all four evaluation cases. We will look at the agent's actions in the 2-deep world, shown in Figure 2a, for demonstration purposes. The agent starts at node #1 in the tree and the goal is at node #4. In lines 2-10 (line numbers refer to Figure 1), the agent reads nodes #3 and #4, writing the minimum of the two values in node #2. Remember that reading node #4, the goal node, returns a 0. The call to GO_MIN in line 11 then uses this information to move the agent to node #2. The state of memory and location of the agent after execution of line 11 are shown in Figure 2b. The call to GO_MIN also moves the look pointer to node #2. Notice that, if neither node #3 or #4 were the goal, a 1 would have been written to node #2 and the call to GO_MIN would have moved the agent to node #5 since the function defaults to the right. The agent now looks to node #3 in line 12 and writes a 1 to it. The second call to LOOK_LEFT in line 14 does nothing. The GO_MIN which follows then moves the agent to node #4, the goal, as shown in Figure 2c. If the goal had been at node #3 instead, lines 12-16 would have written a 0 to node #3 causing correct movement, once again, to the goal with the GO_MIN.

```
1.  (PROGN
2.     (LOOK_LEFT
3.        (PROGN
4.           (WRITE
5.              (MIN
6.                 (LOOK_LEFT
7.                    (WRITE
8.                       (READ)))
9.                 (LOOK_RIGHT
10.                   (READ)))))
11.         (GO_MIN))
12.    (LOOK_LEFT
13.       (PROGN
14.          (LOOK_LEFT
15.             (WRITE
16.                (READ)))
17.          (GO_MIN)))
```

**Figure 10.1**
Evolved solution to 2-deep world. For readability, the program shown is a reduced version of the actual evolved program with ineffectual branches removed. (Remember that GO_MIN defaults to the right).

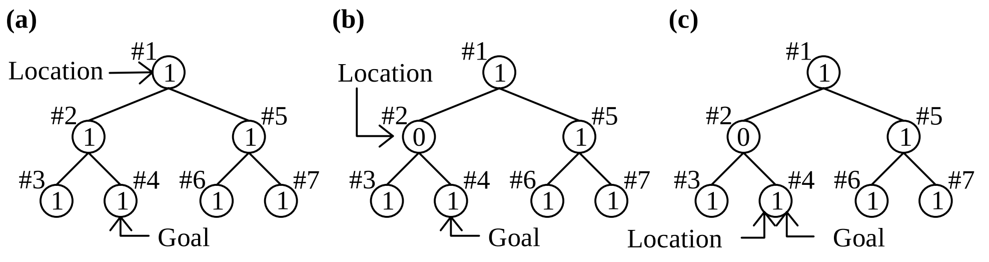

**Figure 10.2**
(a) Example of a single 2-deep world (memory values are shown within the actual nodes for convenience). The starting location of the agent is node #1. The goal is node #4. (b) Intermediate state of memory and location of agent. (c) Final state of memory and location of agent..

## 10.4 ADFs and Recursion

Three techniques for evolving a successful individual with the above function set were investigated. The first technique tested was basic genetic programming with the function tree allowed to be a maximum size of 600 nodes. Table 10.2 summarizes the key features of this technique.

**Table 10.2**
Tableau without ADFs for planning problem

| | |
|---|---|
| Objective: | Find a program to control an agent so that it reaches the goal on all fitness cases. |
| Function set: | `LOOK_LEFT(arg1), LOOK_RIGHT(arg1), READ, WRITE(arg1), MIN(arg1, arg2), GO_MIN, PROGN(arg1, arg2).` |
| Fitness cases: | Goal node at each of the $2^{depth}$ leaf nodes. |
| Raw Fitness: | Average distance away from the goal over the fitness cases plus 0.0001 times the total number of unnecessary looks. |
| Standardized fitness: | Same as the raw fitness. |
| Hits: | Not used. |
| Wrapper: | Not used. |
| Parameters: | Population size (M) 5000. |
| Success predicate: | A program which reaches the goal on all fitness cases. |

**Table 10.3**
Tableau with non-recursive ADFs for planning problem

| | |
|---|---|
| Objective: | Find a program to control an agent so that it reaches the goal on all fitness cases. |
| Function set: (main branch): | `LOOK_LEFT(arg1), LOOK_RIGHT(arg1), READ, WRITE(arg1), MIN(arg1, arg2), GO_MIN, PROGN(arg1, arg2),` *`ADF0`*`,` *`ADF1`*`.` |
| Function set (ADF0): | `LOOK_LEFT(arg1), LOOK_RIGHT(arg1), READ, WRITE(arg1), MIN(arg1, arg2), PROGN(arg1, arg2).` |
| Function set (ADF1): | `PROGN(arg1, arg2), GO_MIN.` |

Automatically defined functions, as described by Koza[1992], were then introduced. Two automatically defined functions (ADFs), of zero arity, with a maximum size of 150 nodes each were added to the function set. The main branch was decreased to 300 nodes to keep the total number of nodes constant. Following [Andre 1994a], the function sets for the ADFs were grouped by functionality; the first ADF was capable of containing all functions except `GO_MIN`, while the second could contain only `GO_MIN` and `PROGN` (Table 10.3). The first ADF could thus only be used for searching and the second, only for moving.

The final technique explored was a limited form of recursion. Both ADFs from the previous technique were simply allowed to call themselves (ADF1 now contained ADF1 in its function set, and ADF2 contained ADF2 as shown in Table 10.4). As mentioned previously, this technique exploits the base case inherent in the problem: any attempt to `LOOK` or `GO` past a leaf node does nothing. The ADFs were only allowed to proceed to a recursive depth equal to the depth of the tree, since any productive function will reach a leaf node by this point. Calling a recursive function past this depth returns the value of the node the look pointer is pointing to. This limit on the depth of the recursion was

**Table 10.4**
Tableau with recursive ADFs for planning problem

| | |
|---|---|
| Objective: | Find a program to control an agent so that it reaches the goal on all fitness cases. |
| Function set: (main branch): | `LOOK_LEFT(arg1), LOOK_RIGHT(arg1), READ, WRITE(arg1), MIN(arg1, arg2), GO_MIN, PROGN(arg1, arg2), ADF0, ADF1.` |
| Function set (ADF0): | `LOOK_LEFT(arg1), LOOK_RIGHT(arg1), READ, WRITE(arg1), MIN(arg1, arg2), PROGN(arg1, arg2),` *`ADF0`*`.` |
| Function set (ADF1): | `PROGN(arg1, arg2), GO_MIN,` *`ADF1`*`.` |

chosen to save computation time, but could have been set to any finite number greater than the depth of the tree without effecting the evolution of the agent. All that is needed is a way to prevent calling a recursive ADF to an infinite depth.

Since a recursive ADF which calls itself excessively can result in inefficient, time consuming, and cryptic programs, a small penalty was added for each `LOOK_LEFT` or `LOOK_RIGHT` which exceeded the minimum number of looks necessary. This minimum is simply the number of `LOOK`s needed to look at every node in the tree once. The penalty was added in all three of the above techniques for consistency. The fitness of an individual is equal to the average distance away from the goal over the fitness cases plus 0.0001 times the total number of unnecessary looks (the exact value of the penalty is not significant) .

## 10.5 Results

The following runs were executed using DGPC (Dave's Genetic Programming Code) on a shared network of Sparc Station 20's. All runs used a population size of 5,000 and continued until either a solution was found (an agent which reached the goal on all fitness cases) or the best-of-generation fitness remained unchanged for over 25 generations. No mutation was used and the maximum initial tree depth was 4 for the recursive technique and 8 for the other two techniques. The increase to 8 was due to the observation that, for larger world sizes, individuals in the initial population needed to be larger than 4 deep to have a chance of achieving anything but the worst possible fitness. The correct way to assign this maximum tree depth is unclear since, obviously, we could not continue to increase the number indefinitely as world size grows. We should realize however, that this inconsistency in the parameters between methods can only make basic genetic programming and genetic programming with ADFs appear better in comparison to the recursive technique than they actually are, for the tree search problem.

The graphs in Figures 10.3-10.7 compare the performance of basic GP, ADFs, and Recursive ADFs on worlds of depth 3, 4, 5, 6, and 7. The success rate at each interval represents the percentage of runs that have, at that point, produced a program which reaches the goal on all evaluations. The solutions were not required to be efficient.

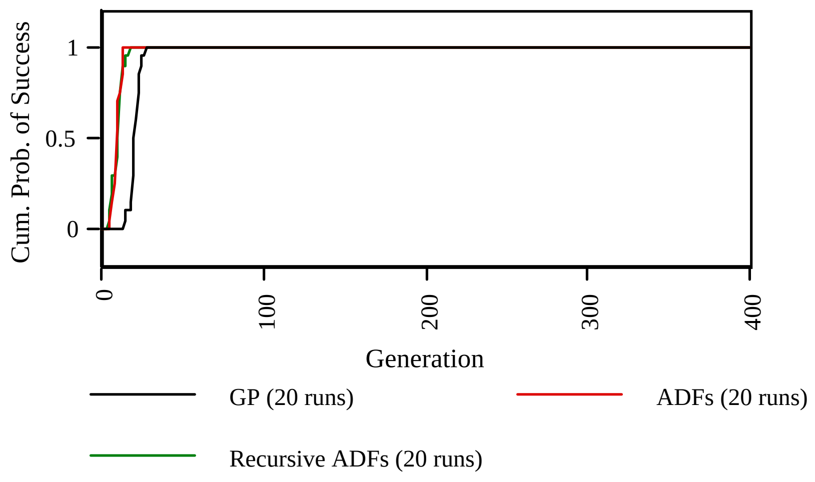

**Figure 10.3**
Cumulative probability of success for genetic programming without ADFs, GP with non-recursive ADFs, and GP with recursive ADFs on worlds of depth 3.

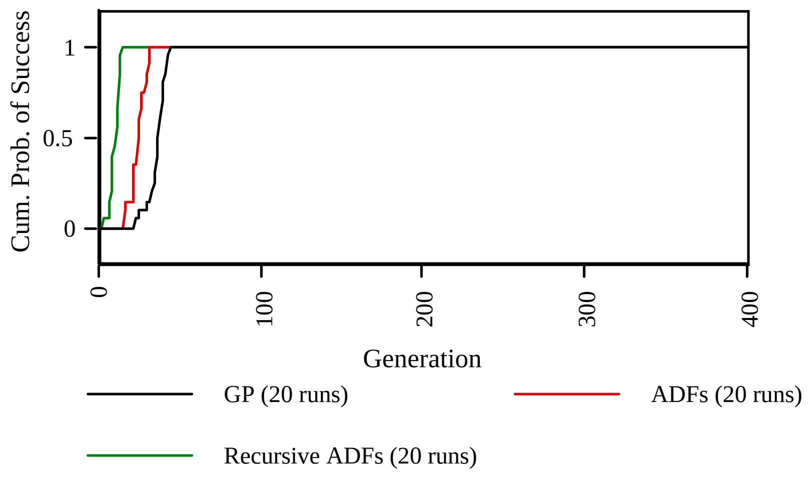

**Figure 10.4**
Cumulative probability of success for genetic programming without ADFs, GP with non-recursive ADFs, and GP with recursive ADFs on worlds of depth 4.

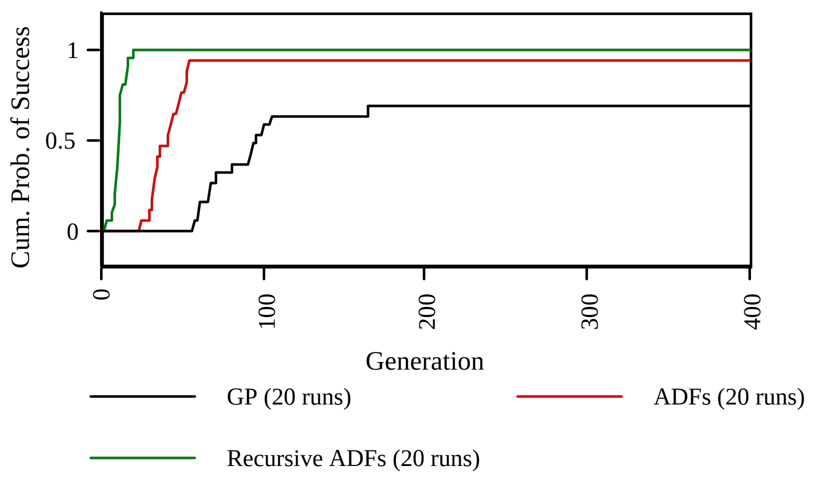

**Figure 10.5**
Cumulative probability of success for genetic programming without ADFs, GP with non-recursive ADFs, and GP with recursive ADFs on worlds of depth 5.

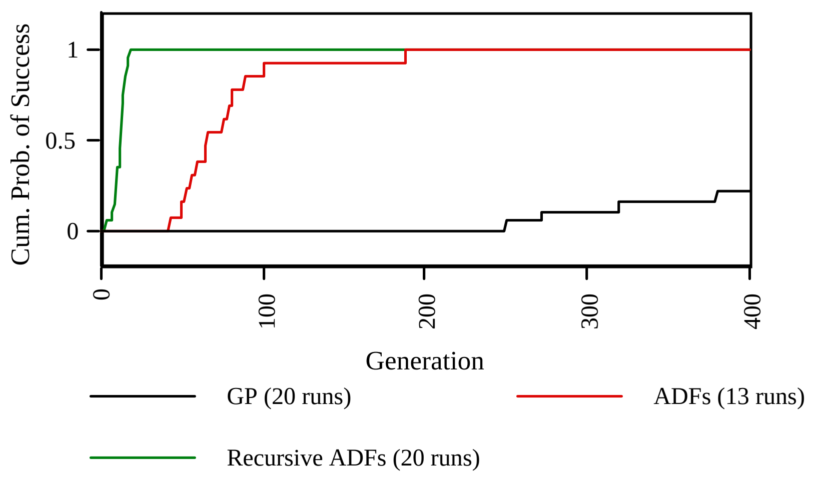

**Figure 10.6**
Cumulative probability of success for genetic programming without ADFs, GP with non-recursive ADFs, and GP with recursive ADFs on worlds of depth 6.

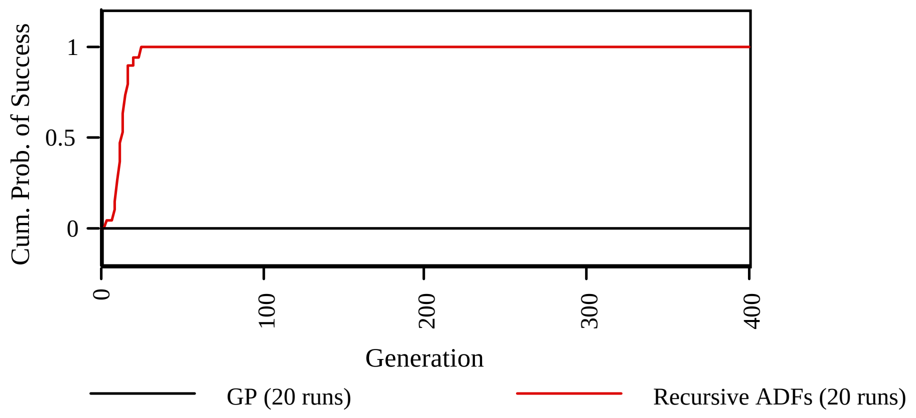

**Figure 10.7**
Cumulative probability of success for genetic programming without ADFs and GP with recursive ADFs on worlds of depth 7.

Once the world depth was increased to 7, we were unable to obtain a sufficient set of data for non-recursive ADFs due to extreme running times (some of the completed runs took over three weeks). Such large running times present a problem because the machines used to execute these runs are time-shared; machine failure is likely within a three week period. Of the 6 runs that completed, half were deemed failures and the other three solved on generations 118, 124, and 287. Although supplemental runs could have been executed until a total of 20 runs was achieved, such a procedure would bias the results toward failure. Runs which become stagnant usually do so much faster than successful runs find solutions; therefore, failed runs are, in general, less affected by machine failure than successful runs. The only valid way to obtain a complete data set would be to run the identical runs which "crashed" over again (i.e. with the same random seed). Several attempts to complete the data set in this way were unsuccessful. Figure 10.7, therefore, does not include a graph for the non-recursive ADF technique.

Figure 10.8 plots the computational effort, E, necessary to yield a solution to the problem with 99% probability for the five world sizes. E is defined as the minimum number of individuals which need to be processed to achieve this probability (Koza 1992). E = minimum(M * *i* * R(z)) where M is the population size, *i* is the generation number, and R(z) is the number of runs required to achieve a probability of z.

$$R(z) = \left\lceil \frac{\log(1 - z)}{\log(1 - P(M, i))} \right\rceil \tag{1.1}$$

P(M, *i*) is simply the probability of success shown in Figures 10.3-10.7.

As mentioned previously, we do not have a sufficient data set for non-recursive ADFs at depth 7; however, we can infer from the data a lower limit on the performance to get an idea of the scaling. The indicated point in Figure 10.8 represents the computational effort assuming that all "crashed" runs would have solved on the next generation if allowed to continue (i.e. if the run was terminated at generation 156 we assume it would solve at generation 157 regardless of the observed fitness). The actual point is thus necessarily above the plotted point and, likely, much higher. Although not seen in Figure 10.8, the computational effort for genetic programming without ADFs is at 31800 (x1000) for a world depth of 6, and is effectively at infinity for a world depth of 7 since no solutions were found at this depth.

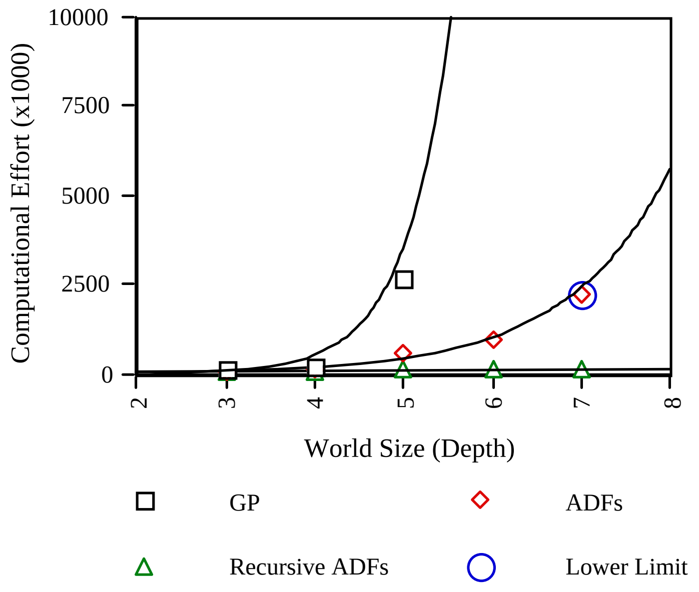

**Figure 10.8**
Comparison of computational effort required using GP without ADFs, GP with non-recursive ADFs, and GP with recursive ADFs. The point for GP, depth 6 is at 38100 and the point for GP, depth 7 is effectively at infinity.

Because this measure of computation effort is often a reflection more of the generation at which the last run solved than of the average performance of all runs, we offer Fig 10.9 which shows the average generation on which solutions were found for the three methods. We must realize however, that this graph paints an extremely optimistic picture of the performance of basic genetic programming and GP with non-recursive ADFs. While genetic program with recursive ADFs achieved 100% success rate on all tested world sizes, some runs failed with the other two techniques on the larger world sizes; Figure 10.9 takes into account only those runs which solved and completely ignores failed runs. Once again, the indicated point represents a lower limit.

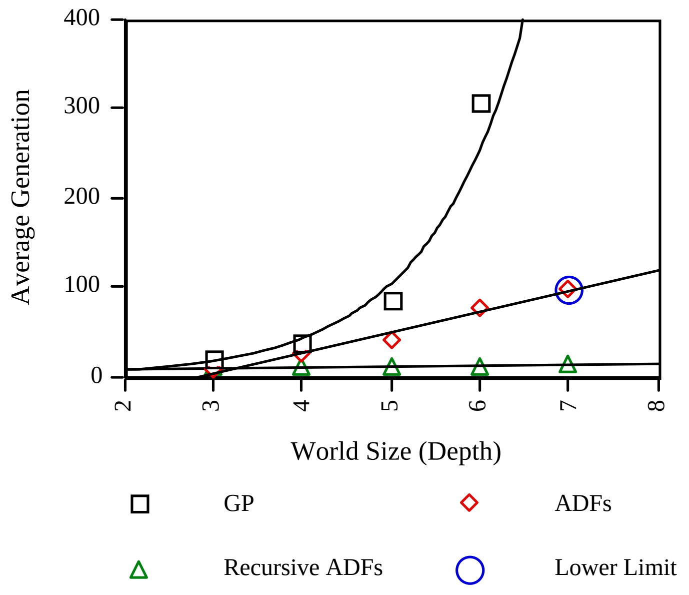

**Figure 10.9**
Comparison of average generation for solution using GP without ADFs, GP with non-recursive ADFs, and GP with recursive ADFs.

```
Main Branch:   (PROGN (LOOK_LEFT (WRITE (ADF0)))(LOOK_RIGHT (PROGN
                      (PROGN (ADF0)(READ))(PROGN (ADF1)(ADF1)))))
ADF0:          (MIN (MIN (LOOK_LEFT (READ))(LOOK_RIGHT
                             (ADF0)))(LOOK_LEFT (WRITE (ADF0))))
ADF1:          (PROGN (PROGN (PROGN (ADF1) (GO_MIN))(PROGN (GO_MIN)
                      (ADF1))) (PROGN (PROGN (ADF1) (GO_MIN))(PROGN
                      (GO_MIN)(ADF1))))
```

**Figure 10.10**
Evolved generalizable solution to 4 deep world using genetic programming with recursive ADFs.

## 10.6 Discussion

As can be seen from Figures 10.3-10.7, ADFs perform consistently better than basic genetic programming, still solving the 7 deep case with some success when basic GP is unable to reach a solution . Recursive ADFs, however, perform better than both of these techniques on all tested world sizes. While the success of ADFs decreases with increasing world size, the performance of the Recursive ADFs remains strong. Looking at the graph of computation effort, we see that the difficulty of the problem for Recursive ADFs remains essentially constant over the tested world sizes, while both basic GP and ADFs scale exponentially. Even Fig 10.9, which takes into account only successful runs, indicates exponential scaling for GP and linear scaling, at best, for GP with non-recursive ADFs. Remember that the two points for ADFs at 7 deep, on Fig 10.8 and Fig 10.9, are lower limits; the actual points are likely much higher. The comparative success of recursive ADFs becomes somewhat clearer with the realization that the results obtained using the recursive technique are often generalizable. The solution shown in Figure 10.10, for example, appeared at generation 7 in the 4 deep world, but is fully generalizable to any size world given the depth constraint on the recursion.

To ensure that the constant performance of the recursive solution was not merely a result of random combinations, rather than the fitness based recombination which defines GP, two random tests were performed. The first used a world depth of 4 with all of the parameters identical to the other runs except one: the fitness of an individual was defined to be 0 if it fully solved the problem and 1 otherwise. This forced all survival and recombination to be by chance. Of the 25 runs executed, none found solutions in 61 generations. Thus 7,625,000 individuals were processed without producing a solution. The second test was the strict generation of random programs with a maximum tree depth of 10. Groups of 15,000 individuals were tested on a 4 deep world and only 3 of 60 (5%) of these runs found solutions.

### 10.6.1 Understanding Performance Differences

Although we make no attempt to fully explain the results on a theoretical level, an intuitive understanding of the differing performance of the three techniques can be gained by considering two limiting issues. First, is the structural complexity of the minimum solution for each technique. The minimum solution for genetic programming using recursive ADFs remains constant with world size as evidenced by solutions such as the one shown in Fig 10.10. For any non-trivial world size, the number of nodes needed to solve the planning problem with recursive ADFs is the same. Looking at basic genetic programming, with realization that the agent must essentially perform two operations per world location, we see that the size of programs must increase exponentially to solve the problem. The scaling of minimum program size for genetic programming with non-recursive ADFs is somewhat more complicated. For each world size there is an arrangement of ADFs which will result in the minimum number of nodes in the program. Although we will not go into a detailed analysis here, the structural complexity scales essentially exponentially, but behaves very similar to the graph of average generation in Fig 10.9 for world sizes on the order of those in this article.

A second, related issue, is that program size was limited to 600 nodes in these experiments. We must of course always use such a limit in practice since computer memory is finite; however, this limitation does reduce the number of reachable solutions in program space. If we were to continue to increase world depth, we would clearly reach a point at which a solution with any method other than recursion is impossible in 600 nodes. The extent to which this issue influenced the observed computational effort for the world sizes used in this article was not explored.

### 10.6.2 Recursion and Tree Search

The question now arises, why does evolving recursion work here when it fails for most other applications? Normally, there are two main problems that make recursion difficult to handle. First, small variations in the structure of recursive program can lead to large variations in functionality and thus fitness. An error in a transition rule, for example, can be compounded with each recursive call; or, a small error in the base case could cause a program with a correct transition rule to perform poorly. Recursive programs are extremely deceptive in this way: the fitness of an individual does not necessarily reflect its proximity to a solution in the space of programs. This means that two individuals with good fitness may be no more likely to produce a fit child than two randomly chosen parents, undermining the process of fitness based reproduction.

Imagine, for example, a recursive factorial program that is correct except that the base case returns a 2 when n = 1. Although the individual is close to the solution in the space of programs, the number returned by the program will be off by a factor of two.

Divergent errors can also appear due to small variations in program structure, leading to even more misleading fitness. For example, if the factorial program returns 1.1n times the previous value as the transition rule instead of n times the previous value, the final value returned will be off by a factor of $1.1^n$. In addition to stable and divergent errors, recursive functions such as the logistic function are extremely sensitive to their base case (initial condition) so that nearly equal initial conditions yield vastly different return values after just a few iterations. In the extreme case, such as when chaos emerges, crossover produces individuals whose fitness is uncorrelated with the parents' fitness and genetic algorithms are reduced to random search.

The other main problem for genetic programming is unending recursion. To return to the factorial example, suppose both the transition rule and the return value of the base case are correct, but the condition for the base case is never satisfied. Even when the function is prevented from executing an infinite number of times, the fitness to assign to such a program is unclear.

The class of tree search problems, however, can avoid these difficulties allowing for the successful evolution of recursive programs. On an intuitive level, evolving recursive programs for tree search works because small changes in program structure rarely lead to the extreme changes in fitness described above. Although errors may be iterated, they are not compounded to the same extent as with arithmetic functions. A small deviation in the sub-portion of a program which searches a certain section of the tree, for example, will not usually affect how the program searches other sections of the tree. The manifestation of this fact is effective crossover; more often than not, two fit parents have a better chance of producing a fit child than two randomly chosen parents. Finally, we can take advantage of the base case inherent in tree search and use a non-intrusive depth limit as we have discussed.

## 10.7 Conclusion

This article has described an approach for investigating the evolution of recursive programs. Basic genetic programming, ADFs, and Recursive ADFs were explored and compared as techniques in the evolution of tree search on a simple planning problem. Of these techniques, ADFs have proven to perform better than basic genetic programming, and Recursive ADFs have been shown to perform the best. Additionally the advantage of using Recursive ADFs increases as larger world sizes are used, with the computational effort required for Recursive ADFs remaining essentially constant in contrast to the apparent exponential growth of basic ADFs. In fact, this method produced many solutions which generalized to worlds of any depth. Given these results, it is clear that the evolution of recursive programs is not only possible, but extremely useful for tree search and essential for larger world sizes. Recursive ADFs have proven to be a tool worthy of further investigation.

## 10.8 Future Work

Future work will proceed as a more in depth investigation of recursive techniques in GP. The feasibility of evolving recursive programs where an explicit base case is needed will be investigated and the range of problems to which the described method can be used will also be explored. Though Recursive ADFs have shown to be successful in this problem, the extent of their utility in genetic programming is not yet clear.

## Acknowledgments

Discussions with Nils Nilsson, John Koza, and David Andre have been extremely helpful. Many thanks to David Andre for the use of DGPC.

## Bibliography

Andre, D. (1994a) "Evolution of Mapmaking: Learning, Planning, and Memory Using Genetic Programming," in *Proceedings of the First IEEE Conference on Evolutionary Computation.* New York, NY: IEEE Press. Volume I.

Andre, D. (1994b) "Learning and Upgrading Rules for an OCR system using Genetic Programming," in *Proceedings of the First IEEE Conference on Evolutionary Computation.* New York, NY: IEEE Press. Volume I.

Andre, D. (1996) and J. Koza, "Locating Transmembrane Domains in Proteins Using Genetic Programming," in *Advances in Genetic Programming 2,* 8, P. Angeline and K. Kinnear, Eds. Cambridge, MA: MIT Press.

Banerjee, A. (1987) "A Methodology for Synthesis of Recursive Functional Programs," in *ACM TOPLAS*.

Hutchinson, A. (1994) *Algorithm Learning*. Oxford: Clarendon Press.

Holland, J. (1986), K. Holyoak, R. Nisbett, and P. Thagard. *Induction: Process of Inference, Learning, and Discovery*. Cambridge, MA: MIT Press.

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

Koza, J. (1994) *Genetic Programming 2: Automatic Creation of Reusable Subroutines*. Cambridge, MA: MIT Press.

Nilsson, N. (1971) *Problem-Solving Methods in Artificial Intelligence.* New York, NY: McGraw Hill.

Nordin, J. (1995) and W. Banzhaf, "Evolving Turing Complete Programs for a Register Machine with Self-Modifying Code," in *Proceedings of the Sixth International Conference of Genetic Algorithms.* San Francisco, CA: Morgan Kaufmann Publishers Inc.

Teller, A. (1994) "The Evolution of Mental Models," in *Advances in Genetic Programming*, P. Angeline and K. Kinnear, Eds. Cambridge, MA: MIT Press.