Evolving Recursive Programs using Non-recursive Scaffolding

Alberto Moraglio
School of Computer Science, University of Birmingham, Birmingham, UK,
Email: A.Moraglio@cs.bham.ac.uk
Fernando E. B. Otero, Colin G. Johnson, Simon Thompson and Alex A. Freitas
School of Computing, University of Kent, Canterbury, UK,
Email: {F.E.B.Otero, C.G.Johnson, S.J.Thompson, A.A.Freitas}@kent.ac.uk

***Abstract*—Genetic programming has proven capable of evolving solutions to a wide variety of problems. However, the successes have largely been with programs without iteration or recursion; evolving recursive programs has turned out to be particularly challenging. The main obstacle to evolving recursive programs seems to be that they are particularly fragile to the application of search operators: a small change in a correct recursive program generally produces a completely wrong program. In this paper, we present a simple and general method that allows us to pass back and forth from a recursive program to an equivalent non-recursive program. Finding a recursive program can then be reduced to evolving non-recursive programs followed by converting the optimum non-recursive program found to the equivalent recursive program. This avoids the fragility problem above, as evolution does not search the space of recursive programs. We present promising experimental results on a test-bed of recursive problems.**

I. Introduction

Recursion is a key technique in the design of programs. However, genetic programming (GP) and related techniques have struggled with the evolution of recursive programs. One difficulty is that GP operators, are rarely successful when applied to recursive programs. There are a number of reasons for this: two important ones are that small changes to the text of a recursive program cascade through the recursion, amplifying the difference between a correct program and a near-mutant; and, the recursive program needs to construct two separate structures—the call itself and the base-case—for fitness to be meaningful.

In this paper we introduce a novel approach to this problem. During evolution, recursive calls are replaced with calls to what we term a *scaffolding* function, which provides the correct answer for all entries in the training set. This allows evolution to work on parts of the recursive structure without needing to evolve a whole recursive structure at once. Once the program has correctly evolved, the non-recursive call is replaced by a recursive call to the evolved program, so the final program is a fully-recursive program with none of the scaffolding remaining.

The remainder of the paper is structured as follows. Section II reviews prior efforts to evolve recursive programs. Section III introduces the new ideas through an extended illustrative example, and Section IV gives details of a specific implementation of this approach and provides results for experiments on two problems. This is followed by brief conclusions and suggestions for future work.

II. Related Work

A first attempt to evolve recursive programs was presented by Koza [1, chapter 18.3], where a function to calculate the Fibonacci sequence was evolved. It uses a special function `SRF` (sequence referencing function) that takes two arguments (`K` and `D`) and returns the `K`-th Fibonacci number if it was already calculated or a default value `D`. In order to evolve a solution, the input values for the Fibonacci sequence are given in ascending order and each result is stored in a table that can be referenced by the `SFR` function. Koza successfully evolved a program to generate the Fibonacci sequence using the first twenty values as input examples, although the evolved program is not actually a recursive program—instead, the `SRF` function allows the program to reference previously computed values (with the requirement that input values are given in ascending order) without the need for recursive calls.

Brave [2] investigated a restricted form of recursion using Automatically Defined Functions (ADFs) to evolve a recursive tree search program. The proposed recursive ADFs method uses two ADFs that are allowed to call themselves, i.e. the function set of ADF1 contains the symbol ADF1 and the function set of ADF2 contains the symbol ADF2. The recursive ADFs were compared against a basic GP (without ADFs) and a GP using (non-recursive) ADFs, and the results show that the recursive ADFs variant has the best performance, both in terms of probability of success and computational effort.

Brave has highlighted two of the main problems that make recursion difficult in GP. Firstly, small variations of the structure of a recursive program can have a big impact in its functionality, and consequently, its fitness. The fitness of a candidate program on such problems does not necessarily reflect its proximity to the optimal solution. Secondly, there is a need to deal with the potential problem of non-terminating recursion. Brave argues that the tree search problem can avoid this difficulty, since small changes in the structure of a candidate program do not have a big impact on its functionality—i.e., a small variation on the sub-portion of a program that searches a specific section of the tree will not affect how the program searches other sections of the tree—and the tree depth can be used as the maximum number of recursive calls to avoid endless recursion; but other class of problems may have to deal with them.

Whigham and McKay [3] investigated the learning of recursive member functions—which take a list and an element, and return true if the element is found in the list or false otherwise—using a tree-based GP by adding the function to be evolved to the non-terminal sets (functions) available, enabling a candidate solution to recursively call itself. They found out that the GP could not successfully find a recursive definition of the member function. They observed that partial solutions—i.e., solutions that satisfy some of the test cases (e.g., solutions that can find the first-element member)—cannot lead the GP towards finding the recursive solution. Moreover, they argue that as GP does not have a mechanism to identify the usefulness of the components (internal structure) of a candidate solution—e.g., a candidate program that fails because of an infinite recursive call will not help to propagate the use of the recursive call, although it only requires the addition of a termination condition.

A similar argument is presented by Wong [4], [5]. A recursive program is described as consisting of one or more base (control) statements and a number of recursive statements. The difficulty of evolving recursive programs arises from the fact that appropriate base and recursive statements, and their correct ordering have to be evolved simultaneously by the GP. Moreover, the fitness function does not reward incorrect programs that contain correct components of the optimal solution—e.g., a candidate program with the correct base statement, but missing or incorrect the recursive statement; or, *vice versa*.

In order to overcome this, Wong proposes an adaptive grammar-based GP (GBGP) which dynamically adjusts the production rule weights to increase/decrease the probability of producing good/bad candidate programs, and to reduce the chance of producing non-terminating programs. Recursion is allowed by placing a production rule that calls the program being evolved into the grammar and a maximim execution time limit was used to penalise non-terminating candidate programs. This was found to increase the probability of success when compared to a non-adaptive GBGP and also reduced (but did not avoided) the number of non-terminating candidate programs generated.

Koza et al. [6] introduced a mechanism similar to ADFs to evolve recursion, called Automatically Defined Recursion (ADR). ADRs involve a recursion condition branch (`RCB`), a recursion body branch (`RBB`), a recursion update branch (`RUB`) and a recursion ground branch (`RGB`). In order to evaluate an ADR, its `RCB` branch is executed first. The `RCB` controls the recursion and while it returns certain values (in general a positive numeric value), the recursion is continued and the `RBB` is executed. The `RBB` may contain references to the ADR of which it is a part, triggering a recursive call. After the execution of the `RBB` branch, the `RUB` is executed. When the recursion is terminated by the `RCB` branch—i.e., the `RCB` does not return a value that indicates to continue the recursion—the `RGB` branch is executed. Therefore, the value returned by an ADR is the value returned by the `RBB` branch, if the `RCB` returns a value that indicates to continue the recursion; otherwise, it is the value returned by the `RGB` branch. Since the bodies of the four branches are subject to modification during the run of the GP, the use of an ADR separates the evolution of the control statement and the recursive statement in a program, and also enforces their correct ordering. On the other hand, it imposes a structure on the evolved programs and also requires special operators to create/modify/delete ADRs. A similar approach was proposed in [7], where Automatically Defined Nodes (ADN) were used to generate recursive programs using graph structured program evolution (GRAPE).

Yu and Clack [8] proposed the use of implicit recursion through a higher-order function `foldr`. The `foldr` function takes a binary operator (a function that takes two arguments) as the first argument and a list of values as the second argument; then, it places the operator between each item of the list and evaluates the resulting expression from right to left. The use of implicit recursion avoids the problem of non-terminating programs, although the evolved program is not actually a recursive program. Such methods can only work on a limited number of problems.

Spector et al. [9] evolved recursive programs using the PushGP system. The Push language provides an execution stack that can be manipulated by the GP in order to achieve recursion and loops, but there is a need to impose an execution-step and program size limits to deal with unbounded recursion or iteration.

Agapitos and Lucas [10], [11] explored the idea of evolving Object Oriented recursive programs, which represent method implementations conforming to a specified interface. Recursion is achieved by allowing calls to the evolved method within the evolved method's body. In this case, the evolved method is placed together with the built-in methods (function set) available for creating new solutions. To mitigate the potential problem of non-terminating recursive programs, a limit on the number of permitted recursive calls is used in their experiments. They draw attention to the fact that using a higher probability of mutation was better than using a higher probability of crossover on most of the problems. They observed that the candidate programs using the recursive call are usually non-terminating ones and consequently have a lower fitness, which may cause the premature elimination of recursive structures when a lower mutation probability is used.

III. Proposed Approach

In this paper we introduce a novel approach to this problem. During evolution, it is assumed that recursive calls from programs in the evolving population return the correct answer—this is clearly a circular assumption, as it entails the knowledge of the sought function. However, at the end of fitness calculation, the circularity can be resolved: instead of carrying out the recursion as such, the correct answer is provided using the information about the sought function stored in the fitness cases.

We now present the proposed approach using a worked example. We will touch on, but not describe systematically, program representation, genetic operators and fitness function. The details will be presented in the next section.

Let us suppose that our task is to evolve a recursive function `reverse` that given a list of items (of any size) returns the reversed list. An example of an optimal candidate solution is:

```
function reverse(list) {
    if (empty(list)) {
        return list;
    }
    else {
        return snoc(
            reverse(tail(list)),
                head(list));
    }
}
```

The above definition returns the empty list when the input list is empty, otherwise it returns the list obtained by appending (`snoc`) the first element of the input list (`head`) to the reversed tail of the input list. This is clearly a recursive definition as it comprises a call to itself.

Let us now consider the following non-optimal candidate solution obtained by mutating the program above and removing the test of the base case of the recursion (i.e., checking whether or not the input list is empty):

```
function reverse(list) {
    return snoc(
        reverse(tail(list)),
            head(list));
}
```

This fails as the commands `head(list)` and `tail(list)` produce an error when applied to the empty list. Notice that this happens for any input list as the recursion consumes the input list and reduces it to the empty list which then produces the error. This program has the worst fitness possible, as it returns an error for any fitness case. This exemplifies the fragility of recursive programs under mutation: a single mutation is sufficient to transform a program with the best fitness into a program with the worst fitness. Furthermore, this is not a particularly unfortunate case, it is a typical case.

Let us now *assume* that instead of the call to itself we have a function `correct-reverse` that, given a list, returns the reversed list. At first, this seems to make no sense as we are assuming we know the very function we are trying to find. We will show later how to resolve this apparent circularity.

An example of an optimal candidate solution with the new functional set is:

```
function reverse(list) {
    if (empty(list)) {
        return list;
    }
    else {
        return snoc(
            correct-reverse(tail(list)),
                head(list));
    }
}
```

There are three important observations about this solution. Firstly, it is not a recursive definition as `reverse` does not call itself in the definition but instead calls `correct-reverse`. Secondly, there is a natural one-to-one correspondence between recursive functions (which call `reverse`) and functions whose definition uses the call to `correct-reverse` obtained by replacing `correct-reverse` with `reverse`, and *vice versa*. Thirdly, exchanging `correct-reverse` with `reverse` or *vice versa* in an optimal solution, we obtain an optimal solution.

Let us now consider the corresponding non-recursive function to the sub-optimal recursive function presented earlier.

```
function reverse(list) {
    return snoc(
        correct-reverse(tail(list)),
            head(list));
}
```

It is interesting to compute the fitness of this solution.[1] This program fails when the input list is an empty list, as the commands `head` and `tail` fail when applied to an empty list. However, when the input list is not an empty list the program produces always the correct solution as the function `correct-reverse` can handle appropriately the base case of the recursion on the tail of the input list (as `correct-reverse` is correct on any list by definition). Therefore, the fitness of this program is almost optimal as it fails only on the fitness case corresponding to the empty input list, but it returns the exact output on any other input list.

There are two important observations that can be made on this example. Firstly, the non-recursive optimal program is more robust to mutation than the corresponding recursive program, as in the recursive case the fitness of the mutated program drops dramatically, whereas in the non-recursive case the fitness drops only one fitness case. In general, the reason why recursive programs are more fragile than the corresponding non-recursive programs seems to be that when there are one or more errors in the program (i.e., when the program is one or more mutations away from the optimum) the recursive calls re-use the erroneous definition of the program multiple times and propagate and amplify the effect of the errors. One the other hand, in non-recursive programs errors do not percolate down the recursive call, so the fitness reflects more closely the actual number of errors. Secondly, the fitness landscape of non-recursive programs has a gradient that can make it easier to search than the corresponding fitness landscape of recursive programs, while both fitness landscapes have exactly the same global optima.

There is a constraint that needs to be imposed. We do not allow the program to call the reverse function on a list of the same size of the input list or larger. When this happens the program fails (for all inputs for which the program fails to meet this condition). This constraint is important for two reasons. In recursive programs, it helps to prevent infinite recursion. In the corresponding non-recursive programs, it

[1]Here we assume that better solutions have higher fitness, so an optimal solution has the highest fitness.

allows us to resolve the circularity deriving from allowing `correct-reverse` calls within the definition of candidate solutions.

It is clear from the discussion above that to find the optimal recursive program it would be desirable to proceed as follows: (i) evolve non-recursive programs which are associated with an easier fitness landscape and find an optimum non-recursive program; (ii) convert the optimum non-recursive program found into the corresponding optimum recursive program by replacing in it all occurrences of `correct-reverse` with `reverse`.

However, this seems problematic: in order to evolve non-recursive programs one needs to know the function `correct-reverse`, which is what is being searched for! But, consider how programs are evolved using GP. During evolution, the program is tested on a number of *fitness cases* for which the desired output is known. For example, for `reverse` we would have a set of fitness cases such as:

```
Size   Function
0      {}-->{}
1      {X1}-->{X1}
2      {X1,X2}-->{X2,X1}
3      {X1,X2,X3}-->{X3,X2,X1}
```

There are two observations that need to be made. Firstly, the fitness cases are an explicit enumeration of a set of cases for which we know the function `correct-reverse`. In the specific example the fitness cases tell us the following:

```
correct-reverse({})={}
correct-reverse({X1})={X1}
correct-reverse({X1,X2})={X2,X1}
correct-reverse({X1,X2,X3})={X3,X2,X1}
```

Secondly, the set of fitness cases chosen covers all cases of the target function for lists from size 0 to size 3. Therefore these fitness cases define completely the function `correct-reverse` for lists up to size 3. In general, to be able to apply our approach, analogously to the example above, we require the fitness cases to be a complete sequence of input-output pairs defining the target function starting with the first case (i.e., base case of the recursion) up to a fixed later term in the recursion.

Now putting together the requirement that: (i) the `correct-reverse` function can be called by a program only on lists shorter that the input list; and (ii) the fitness cases define completely the `correct-reverse` function up to input lists of size 3; it then becomes possible to compute the fitness of any program making use of the function `correct-reverse` in its definition. This is because to compute the fitness of a program we need to compute its output on all the input lists of the fitness cases. As these input lists range in size from 0 to 3, in all cases the program will need to call the function `correct-reverse` on lists of size 2 or less. The output values of the `correct-reverse` function on these lists is always known (from the fitness cases), hence the fitness of the program can be computed. This resolves the apparent circularity of evolving programs that make use of the `correct-reverse` function in their definition.

It is important to notice here that the programs making use of the call to the `correct-reverse` function are well-defined, in our example, only on input lists up to size 4 (as they can then make calls to `correct-reverse` input lists up to size 3, whose cases are covered by the fitness cases). In general, the output of the programs for lists of size 5 or larger cannot be computed, as the `correct-reverse` for input lists of size 4 is not in the set of fitness cases. So, in this respect, an optimal program that uses `correct-reverse` in its definition does not differ much from the `correct-reverse` defined by enumeration on the fitness cases, as the optimal program can be only used to generalise to the case of lists of size 4. The apparently merely formal act of replacing the function `correct-reverse` with the function `reverse` in the definition of the program to obtain the associated recursive program is the true source of the generalisation. This is because the recursive program is well-defined and can be queried on input lists of arbitrary length.

The approach presented in this section is very general and can be applied to find recursive programs defined on virtually any domain. Also, it is not bound to a specific type of GP, and can be used with different flavours of GP using different representations and different search operators.

IV. Experiments

This section shows how the above approach was implemented in the context of a grammar-based GP system. Experiments have been carried out on two problems concerned with the manipulation of lists of integer numbers, *viz.* reversing a list and inserting an element into a sorted list. The same basic algorithmic approach was used in both a traditional recursive approach where the name of the function being used is available within the grammar, and the new approach, referred to below as *scaffolding-based* approach, where the correct function is available within the grammar to substitute the recursive call during evolution. We have studied a variety of genetic operator settings: crossover alone, mutation alone, and a combined crossover and mutation.

A. General Approach

In order to evaluate the effectiveness of the proposed approach in evolving recursive functions, we have used a context free grammar-based genetic programming (CFG-GP) [12]. The basic grammar productions used to define the structure of the programs (individuals) were:

```
S ::= T
T ::= if(B) { T } else { T } | return L
L ::= append(L, L) | snoc(L, N) | cons(N, L) |
      tail(L)
B ::= empty(L) | N = N | N <= N | N >= N
N ::= head(L)
```

where the start symbol is *S*; nonterminals symbols are *T* (a statement that returns a list), *L* (a statement that evaluates to a list), *B* (a statement that evaluates to a boolean value) and *N* (a statement that evaluates to a numeric value); and terminals symbols are:

- list operations: *append* (join two lists), *snoc* (appends an element to the end of the list), *cons* (appends an element to the start of the list), *tail* (elements of the list excluding the first one), *empty* (tests if the list has no elements), *head* (first element of the list);
- binary operators: = (equality), $<=$ (less than or equal to) and $>=$ (greater than or equal to);
- conditional statement: *if-else* (if the test *B* evaluates to true, executes the first block; otherwise executes the second block).

For each of the problems, the grammar productions were extended to include specific parameters and the recursive call—details of these are given in Sections IV-B and IV-C below. Using the grammar presented above, individuals are represented by derivation trees, where the internal nodes of the tree correspond to non-terminal symbols, leaf nodes correspond to terminal symbols and the root node correspond to the start symbol. A derivation tree is created in a top-down fashion by randomly selecting and applying grammar productions from a non-terminal symbol, starting from the start symbol *S* (root node of the tree), until a leaf node (terminal symbol) is reached for each tree path.

The fitness of a program consists of the sum of the edit distances between the list returned by a program and the correct list solution, over all test cases. The edit distance measures the minimum number of insert, delete and substitute operations needed to transform one list into the other. Therefore, the aim of a program is to minimise the edit distance—i.e., the more similar the lists, the better the program. When a program uses a recursive call or the correct function (in the *scaffolding-based* approach) on a list of the same size of the input list or larger, or perform an invalid operation (e.g., invoke *tail* or *head* on an empty list) for a specific test case, a penalty is imposed by setting the edit distance equal to the length of the longest test case list $+ 1$. The motivation of using the length of the longest list as a penalty is that we guarantee that this value will be greater than the distance obtained by a working program on any of the test cases, even if the list returned by the working program does not contain any element in the correct position. Also, it does not impose an arbitary high penalty.

In the runs that produced the results reported in this section we have used population size of 500, tournament size 5, number of generations 200, the initial population was created using a ramped-half-and-half method (from tree depth 4 to 8) and the maximum tree depth for crossover and mutation operators was 16. These parameters values have been determined based on preliminary experiments and they were used in all the experiments presented in this section, for both the recursive and *scaffolding-based* approaches.

The most critical parameters were found to be the crossover and mutation rates, which are therefore varied in the experiments. We have used standard CFG-GP crossover and mutation operators [12]. The crossover is performed selecting two individuals p_1 and p_2 from the population; then a non-terminal node from p_1 is randomly selected and a matching non-terminal is selected from p_2; if no matching node is found, the crossover restarts by selecting other individuals; finally, the subtrees below these non-terminals are swapped. The mutation is performed on a single individual by randomly selecting a non-terminal node and generating a new subtree from the grammar using this non-terminal as the starting symbol.

B. Reverse List Problem

Given a list of elements as input, the reverse list problem consists of returning a list with the elements in the reverse order—e.g., for the input list $\{1, 2, 3, 4\}$, the correct output is the list $\{4, 3, 2, 1\}$. We have used training cases involving lists of variable lengths from 0 to 5 elements (6 fitness cases in total) and the aim for the GP is to evolve a recursive program that can generalise to reverse any input list of any length. An example of the correct solution is:

```
function reverse(list) {
    if (empty(list)) {
        return list;
    }
    else {
        return snoc(
            reverse(tail(list)),
                head(list));
    }
}
```

We can identify two main structures ('building blocks') that compose an optimal solution: (1) the operation that adds the head of the input list to the list returned by the recursive call used to reverse the tail of the input list (i.e., the operation that reverses one element of the list); (2) the conditional test that stops the recursion when the input list is empty. If either of these structures is missing, the optimal solution cannot be created.

To evolve a recursive program for the reverse problem, the grammar symbol `L` was extended to include two productions: `list` (the input list) and `reverse(L)` (the recursive call to the evolved function). To evolve a non-recursive program (using the proposed *scaffolding-based* approach), the grammar symbol `L` was extended to include two productions: `list` (the input list) and `correct-reverse(L)` (the call to the function that returns the correct reversed list). Three sets of experiments were carried out: crossover-only, mutation-only, and mutation-and-crossover. The results of these are presented in the remainder of this section.

In the first set of experiments, we have set the mutation rate to 0% and crossover rate to 100% (crossover-only setting). The recursive approach did not find any solution over 100 runs, while the *scaffolding-based* approach found a solution in 5 out of 100 runs, reaching the minimum computational effort *I(M,i,z)*—i.e. the total number of individuals that must be processed in order to yield a solution to the problem with 99% probability, where the lower the value the better the computational effort—value of 228,456 at generation 1. These results illustrate the difficulty of evolving recursive programs: the operation involving the recursive call, the recursive structure (conditional test) and their correct ordering should be evolved simultaneously. It is unlikely that individuals created

TABLE I
SUMMARY OF THE RESULTS FOR THE REVERSE PROBLEM. THE BEST *I(M,i,z)* (LOWER COMPUTATION EFFORT VALUE) IS SHOWN IN BOLD.

(i) recursive approach	*P(M,i)*	*min I(M,i,z)*
crossover-only	0%	–
mutation-only	89%	201,000
crossover-and-mutation	71%	362,500
(i) scaffolding-based approach	*P(M,i)*	*min I(M,i,z)*
crossover-only	5%	228,456
mutation-only	65%	169,500
crossover-and-mutation	**100%**	**49,000**

in earlier generations will have the required structures and/or correct ordering, therefore their fitness will be generally low. This may cause the premature elimination of 'building blocks' required to create an optimal solution. Since the crossover does not introduce new structures to the population, it is limited to recombine individuals in the current population and it will not have the chance to combine individuals with the required structures to create an optimal solution.

In the second set of experiments, we have set the mutation rate to 100% and crossover rate to 0% (mutation-only setting). The recursive approach found a solution in 89 out of 100 runs, reaching the minimum computational effort value of 201,000 at generation 133. The *scaffolding-based* approach found a solution in 65 out of 100 runs, and although the number of successful runs was smaller than the recursive approach, it has reached the minimum computational effort value of 169,500 at generation 2. This is substantially better than the crossover-only setting. Since the mutation operator can introduce new structures in the population, it is not limited by diversity of the individuals in the population. The performance curves for the recursive and *scaffolding-based* approach are illustrated in Fig. 1(a) and Fig. 1(b), respectively—the *P(M, i)* curves shown in Fig. 1 correspond to the cumulative probability of success that a run with a population size $M = 500$ yields a solution by generation i.

Although the results of the *scaffolding-based* approach are better than the recursive approach, they do not highlight the main advantage of the proposed approach—i.e., in the *scaffolding-based* approach, both the operation involving the recursive call (represented by the call to the correct reverse function) and conditional test can be evolved separately, since the evolved programs are not actual recursive programs. For example, consider the program with the correct conditional test:

```
function reverse(list) {
    if (empty(list)) {
        return list;
    }
    else {
        return snoc(tail(list), head(list));
    }
}
```

which can reverse lists of 0 to 2 elements—i.e., 3 out of 6 fitness cases—and the program with the correct operation involving the recursive call:

```
function reverse(list) {
    return snoc(correct-reverse(tail(list)),
        head(list));
}
```

which can reverse 5 out of 6 fitness cases (it fails in the empty input list case). Both programs represent fit individuals and have a greater chance of being selected for future generations and the use of crossover has the chance of combining them to construct the optimal solution. On the other hand, in the recursive approach, the equivalent program with the correct operation involving the recursive call:

```
function reverse(list) {
    return snoc(reverse(tail(list)),
        head(list));
}
```

fails in all fitness cases, since there is no structure to stop the recursion when the input list is empty. Therefore, the individual will have a very low fitness, and consequently, smaller chance of being selected for future generations.

To make this advantage more apparent, we run a third set of experiments setting the mutation rate to 80% and crossover rate to 20% (crossover-and-mutation setting)[2]. The performance of the recursive approach decreased slightly, finding a solution in 71 out of 100 runs and reaching the minimum computational effort value of 362,500 at generation 24. On the other hand, the performance of the *scaffolding-based* approach increased, finding a solution in 100 out of 100 runs and reaching the minimum computational effort value of 49,000 at generation 97. The performance curves for the recursive and *scaffolding-based* approach are illustrated in Fig. 1(c) and Fig. 1(d), respectively. The positive effect of the crossover operator in the *scaffolding-based* approach is a result of the fact that the population contains working individuals with the required 'building blocks', i.e., there are fit individuals with the correct conditional test and fit individuals with the correct operation involving the recursive call. Table I presents a summary of the results for the reverse problem.

C. Insert Problem

Given an ordered list of elements and a new element as inputs, the insert problem consists of inserting the new element into the list maintaining the natural order of the elements and returning that list—e.g., for the input list $\{1, 3\}$ and the element 2, the correct output is the list $\{1, 2, 3\}$. We have used training cases involving lists of variable lengths from 0 to 3 elements (10 fitness cases in total)—representing cases where an element should be inserted at the beginning, middle or end of the list—and the aim for the GP is to evolve a recursive program that can generalise to insert an element to any ordered

[2]The 80% mutation and 20% crossover rates have been empirically determined by a systematic (but coarse) parameter tuning. They are inline with the findings of Agapitos and Lucas [10], [11], which suggest the use of a higher mutation rate in relation to crossover rate.

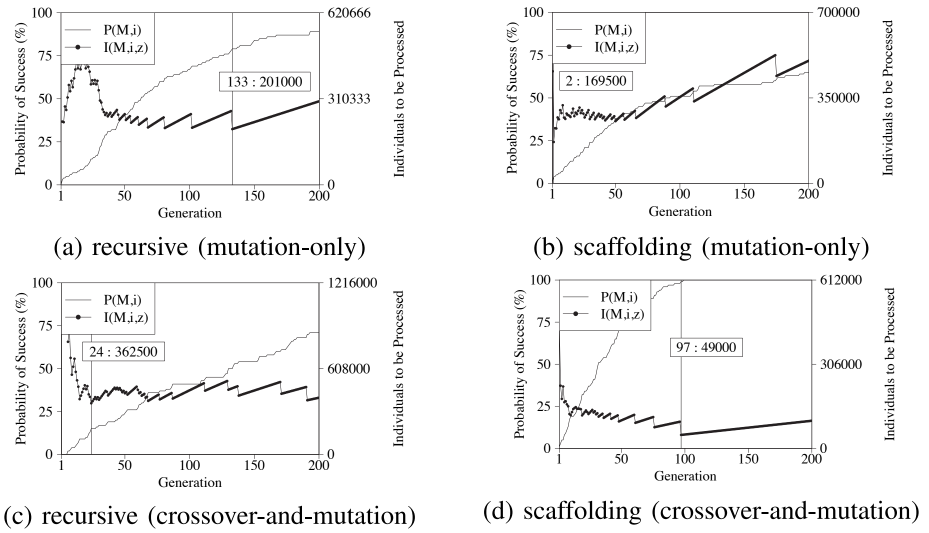


Fig. 1. Performance curves—cumulative probability of success *P(M,i)* and computational effort *I(M,i,z)*—for the recursive and *scaffolding-based* approaches in the reverse problem for the mutation-only and crossover-and-mutation settings. The value in the box corresponds to the generation followed by the lowest *I(M,i,z)* achieved; the best *I(M,i,z)* value is achieved by the *scaffolding-based* approach using the crossover-and-mutation setting.

input list of any length. An example of the correct recursive solution for the insert problem is as follows:

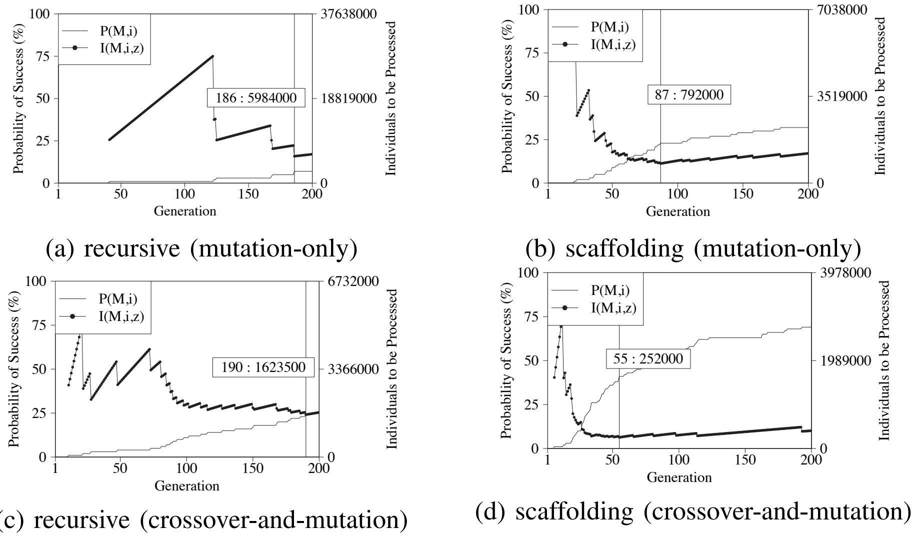
```
function insert(list, x) {
    if (empty(list)) {
        return cons(x, list);
    }
    else {
        if (x <= head(list)) {
            return cons(x, list);
        }
        else {
            return cons(head(list),
                insert(tail(list), x));
        }
    }
}
```

The insert problem represents a more complex problem than the reverse, and we can identify three main structures: (1) the operation that adds the head of the input list to the list returned by the recursive call used to insert the input value in the correct position; (2) the conditional test that stops the recursion when the input list is empty; and (3) the conditional test that stops the recursion when the correct position of the input value is found (i.e., the input value is smaller or equal to the head of the input list). If any of these structures is missing, the optimal solution cannot be created.

The experiments for the insert problem followed the same experimental setup and the same parameter settings as for reverse problem. To evolve a recursive program for the insert problem, the grammar symbol `L` was extended to include two productions: `list` (the input list) and `insert(L, x)` (the recursive call to the evolved function); and the grammar symbol `N` was extended to include a new production `x` (the input value for the evolved function). To evolve a non-recursive program (using the proposed *scaffolding-based* approach) for the insert problem, the grammar symbol `L` was extended to include two productions: `list` (the input list) and `correct-insert(L, x)` (the call to the function that returns the correct list containing the new element `x`); and the grammar symbol `N` was extended to include a new production `x` (the input value for the evolved function).

In the first set of experiments, we have set the mutation rate to 0% and crossover rate to 100% (crossover-only setting). The recursive approach did not find any solution over 100 runs, while the *scaffolding-based* approach found a solution in 1 out of 100 runs, reaching the minimum computational effort value of 2,299,590 at generation 9. In the second set of experiments, we have set the mutation rate to 100% and crossover rate to 0% (mutation-only setting). The recursive approach found a solution in 7 out of 100 runs, reaching the minimum computational effort value of 5,984,000 at generation 186. The *scaffolding-based* approach found a solution in 32 out of 100 runs, reaching the minimum computational effort value of 792,000 at generation 87. The performance curves for the recursive and *scaffolding-based* approach are illustrated in Fig. 2(a) and Fig. 2(b), respectively. The mutation-only setting is substantially better than the crossover-only setting, achieving a lower computational effort value.

The best results are obtained setting setting the mutation rate to 80% and crossover rate to 20% (crossover-and-mutation setting), used in our third set of experiments. Differently than the reverse problem, the performance of the recursive approach using mutation combined with crossover increased, finding a solution in 25 out of 100 runs and reaching the minimum computational effort value of 1,623,500 at generation 190. The performance of the *scaffolding-based* approach also increased, finding a solution in 69 out of 100 runs and reaching the minimum computational effort value of 252,000 at generation 55. The performance curves for the recursive and *scaffolding-based* approach are illustrated in Fig. 2(c) and Fig. 2(d), respectively. These results show that the insert problem is more



(a) recursive (mutation-only)

(b) scaffolding (mutation-only)

(c) recursive (crossover-and-mutation)

(d) scaffolding (crossover-and-mutation)

Fig. 2. Performance curves—cumulative probability of success *P(M,i)* and computational effort *I(M,i,z)*—for the insert and *scaffolding-based* approaches in the insert problem for the mutation-only and crossover-and-mutation settings. The value in the box corresponds to the generation followed by the lowest *I(M,i,z)* achieved; the best *I(M,i,z)* value is achieved by the *scaffolding-based* approach using the crossover-and-mutation setting.

TABLE II
SUMMARY OF THE RESULTS FOR THE INSERT PROBLEM. THE BEST *I(M,i,z)* (LOWER COMPUTATION EFFORT VALUE) IS SHOWN IN BOLD.

(i) recursive approach	*P(M,i)*	*min I(M,i,z)*
crossover-only	0%	–
mutation-only	7%	5,984,000
crossover-and-mutation	25%	1,623,500
(i) scaffolding-based approach	*P(M,i)*	*min I(M,i,z)*
crossover-only	1%	2,299,590
mutation-only	32%	792,000
crossover-and-mutation	**69%**	**252,000**

complex than the reverse problem, which is expected since it requires more structures to create the optimal solution. They also show that the proposed approach successfully reduces the complexity of the search by avoiding the need to evolve these structures simultaneously. Table II presents a summary of the results for the insert problem.

V. CONCLUSIONS

We have introduced an indirect approach to evolving programs with recursion, based on the idea of using a scaffolding function that replaces the recursive call and that provides the correct answer for all entries in the training set, and which can be removed once the correct program has been evolved. The programs using the scaffolding functions are not recursive, hence much less fragile to the application of search operators than the corresponding recursive programs. This has been shown to have a considerably better performance than a traditional approach. Future work will extend this approach to a broader class of problems, in particular investigating problems where multiple calls to the recursive function are needed; and a more detailed investigation of the structure of fitness landscapes in recursive problems and how these approaches transform those landscapes.

REFERENCES

[1] J. R. Koza, *Genetic Programming: On the Programming of Computers by Means of Natural Selection*. MIT Press, 1992.

[2] S. Brave, "Evolving Recursive Programs for Tree Search," in *Advances in Genetic Programming 2*. MIT Press, 1996, pp. 203–220.

[3] P. Whigham and R. McKay, "Genetic Approaches to Learning Recursive Relations," in *Progress in Evolutionary Computation*, ser. Lecture Notes in Artificial Intelligence, 1995, pp. 17–27.

[4] M. Wong, "Applying adaptive grammar based genetic programming in evolving recursive programs," in *Proceedings of The First Asian-Pacific Workshop on Genetic Programming*, 2003, pp. 1–8.

[5] ——, "Evolving Recursive Programs by Using Adaptive Grammar Based Genetic Programming," *Genetic Programming and Evolvable Machines*, vol. 6, no. 4, pp. 421–455, 2005.

[6] J. R. Koza, F. H. Bennett III, D. Andre, and M. Keane, *Genetic Programming III: Darwinian Invention and Problem Solving*. Morgan Kaufmann, 1999.

[7] S. Shirakawa and T. Nagao, "Graph Structured Program Evolution with Automatically Defined Nodes," in *Proceedings of the Conference on Genetic and Evolutionary Computation (GECCO-2009)*, 2009, pp. 1107–1114.

[8] T. Yu and C. Clack, "Recursion, Lambda Abstractions and Genetic Programming," in *Proceedings of the Third Annual Genetic Programming Conference*, 1998, pp. 422–431.

[9] L. Spector, J. Klein, and M. Keijzer, "The push3 execution stack and the evolution of control," in *Proceedings of the Conference on Genetic and Evolutionary Computation (GECCO-2005)*, 2005, pp. 1689–1696.

[10] A. Agapitos and S. Lucas, "Evolving efficient recursive sorting algorithms," in *Proceedings of the IEEE Congress on Evolutionary Computation (CEC-2006)*, 2006, pp. 2677–2684.

[11] ——, "Learning recursive functions with object oriented genetic programming," in *Proceedings of the 14th European Conference on Genetic Programming (EuroGP-2006)*, 2006, pp. 166–177.

[12] P. Whigham, "Grammatically-based genetic programming," in *Proceedings of the Workshop on Genetic Programming: From Theory to Real-World Applications*, 1995, pp. 33–41.