Boosting Search for Recursive Functions Using Partial Call-Trees

Brad Alexander and Brad Zacher

School of Computer Science, University of Adelaide, 5005, Australia
bradley.alexander@adelaide.edu.au
brad.zacher@alumni.adelaide.edu.au
http://www.cs.adelaide.edu.au/~brad

Abstract. Recursive functions are a compact and expressive way to solve challenging problems in terms of local processing. These properties have made recursive functions a popular target for genetic programming. Unfortunately, the evolution of substantial recursive programs has proven difficult. One cause of this problem is the difficulty in evolving both correct base and recursive cases using just information derived from running test cases. In this work we describe a framework that exploits additional information in the form of partial call-trees. Such trees - a by-product of deriving input-output cases by hand - guides the search process by allowing the separate evolution of the recursive case. We show that the speed of evolution of recursive functions is significantly enhanced by the use of partial call-trees and demonstrate application of the technique in the derivation of functions for a suite of numerical functions.

Keywords: Recursion, Genetic Programming, Call-Tree, Adaptive Grammar.

1 Introduction

Recursion is a compact and expressive way to define solutions to challenging problems in terms of local processing. The brevity and power of recursion have made the evolution of such functions a popular target for genetic programming (GP) [5], [9], [6]. Unfortunately, the evolution of non-trivial recursive functions through GP has proven difficult in practice [1]. One cause of this difficulty is the need to simultaneously evolve correct code for base and recursive cases [1], [7] before a good fitness score is achieved.

Several approaches to improve search been tried. These have included: the use of niches to preserve diversity during search [7]; the automated discovery and separate evolution of base cases [6]. Other work has narrowed the search-space using templates expressing common patterns of recurrence [10,11].

However, while these approaches are beneficial, a central problem remains that the test cases used to evaluate fitness in GP provide poor guidance in the search for the *recursive* clause in recursive functions. In this work we improve search using additional information in form of *partial call-trees*. We show how this

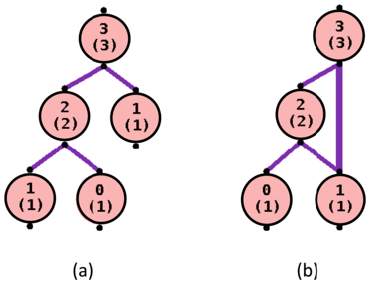

Fig. 1. An example call tree for a Fibonacci function (a) and an equivalent graph (b) with different ordering of children and shared child nodes

extra information can substantially improve GP search for recursive functions by allowing code for recursive calls to be separately evolved.

1.1 Call Trees

A call tree, an example of which is shown in Fig 1(a), is a diagram often used when informally reasoning about recursive problems. Part (b) of Fig 1 shows a graph which is equivalent to part (a) from the point of view of our framework but faster for the user to draw[1]. Each node in the call tree contains the parameter(s) of the call and each child represents the the sub-calls made by that node. The return value from a call can also be included in a call tree node. In Fig 1 these return values are shown in brackets. In this work we only require the user to provide call-trees with return values for *some* nodes. Moreover, not every child call of each node is required and the tree can even be disjoint if the user desires. Our framework is designed to require no more information from the user than they might already create in the first, informal, stages of reasoning about a recursive function. This extra information can then be harnessed to boost GP search by allowing for the separate evolution of the code for the recursive case.

1.2 Contributions

We describe a framework that extracts information from a call-tree to boost GP search. We demonstrate that our framework, which we name: Call-Tree-Guided Genetic Programming (CTGGP), significantly improves the speed of search over conventional GP search on a range of benchmarks. Moreover, we show that the structure of partial call trees can be used to guide the choice of grammars which further restricts the search space. This restricted search space permits the evolution of functions with quite complex behaviour including functions that make their calls in loops.

[1] For the sake of brevity we will refer to both as trees in this article since our framework treats both equivalently.

The rest of this article is structured as follows. In the next section we outline related work. In section 3 we describe CTGGP and how it processes its inputs to produce recursive functions. In section 4 we describe our experimental setup including the benchmarks and grammars we use. In section 5 we present our our results and, finally, in section 6 we summarise our findings and canvas future work.

2 Related Work

The difficulty in evolving the base and recursive case of recursive functions has been recognised by several authors [1], [7], [6]. Nishiguchi and Fujimoto [7] improved search by allowing less fit individuals to be preserved in a niche to maintain diversity. Moraglio et al. [6] showed that separate evolution of code for the base-case significantly improved search. This work is the most similar to ours in using the idea of separating the search of the cases. However, our work is differs by separately evolving the recursive-case rather than base-case and, thus, is complementary to Moraglio's work.

Other authors have improved performance by reducing the search space by: restricting grammars to common patterns of computation [11]; or by adapting grammars to the application [10].

Finally, the use of direct inference about the relationship between recursive calls is a feature of inductive programming [3,4]. This work has been very effective in performing direct search. However, such work is typically restricted to list functions where operators can be more readily inferred from I/O cases.

3 CTGGP

The search algorithms in CTGGP take a partial-call-tree as input and produce a recursive function in C as output. In our current experiments, CTGGP is restricted to discovering functions of one or two integer parameters producing an integer result[2]. The input tree is quickly authored by the user using a Java GUI forming part of the system.

The search process in CTGGP is built on Grammatical Evolution [8] using the C++ GELib (0.26) distribution. Grammatical Evolution (GE) is a GP framework that allows the user to specify an arbitrary grammar for a target language. GE is then able to use this grammar to guide a genotype-to-phenotype mapping that maps a bit-string genome into a syntactically correct individual program. Because this mapping is guided by the grammar, all individuals produced are syntactically correct in the target language.

CTGGP's search process has two phases. These are shown in Fig 2. Phase one evolves code determining the recursive calls of the target function and also selects grammars to be used in phase two. Phase two evolves the remainder of the target function. The second phase is a conventional application of GE using the

[2] Though we forsee no barriers to generalisation to other types.

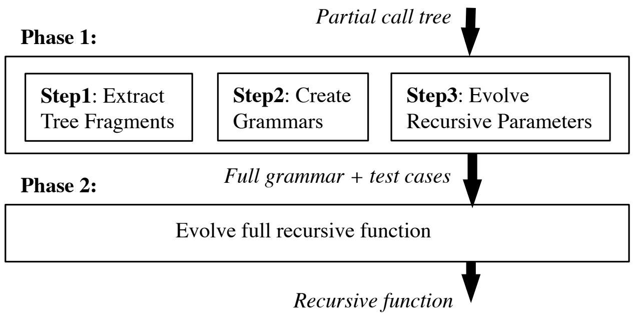
Partial call tree
Phase 1:
Step1: Extract Tree Fragments
Step2: Create Grammars
Step3: Evolve Recursive Parameters
Full grammar + test cases
Phase 2:
Evolve full recursive function
Recursive function

Fig. 2. Evolutionary process of CTGGP

input and output values (the ones in brackets in Fig 1) as tests in the evaluative function. As such we won't describe phase two in further detail. The primary contribution of this article is in phase one and we describe this next.

3.1 Phase One Search

Phase one has three steps. Step one, extracts tree fragments from the partial call-tree provided by the user. Step two, adapts and chooses grammars for phase one *and* phase two search. Step three uses GE to evolve the code that determines the parameters to the recursive calls in the recursive case. The code from step three will be embedded in the candidate programs generated in phase two. We outline each step in phase one next.

Step One: Producing Tree Fragments. Step one traverses the user-supplied call tree and, for each non-leaf node, i produces a target tree fragment of the form (tp_i, tc_i) where tp_i is the input parameter to the parent node and target child list: tc_i is an unordered list of the child node input parameters. To illustrate: the list of target fragments for both parts of Fig 1 is

$$[(3,[1,2]),(2,[0,1])] \tag{1}$$

These target fragments are compared to those produced by candidate individuals in step three of phase one.

Step Two: Grammar Production. Step two inspects the call tree provided by the user and, with confirmation from the user, builds the grammars to guide phase one and phase two search. CTGGP guesses the grammars to be used based on the number of children per node in the tree. If there are two children per-node as there are in Fig 1 then it will guess that there are two recursive calls. In contrast, if there are a variable number of children per node it will guess that the recursive calls take place in a loop. CTGGP will also guess the number of

```
<expr_root> ::= <var> <op> <digit> |
  <digit> <op> <var>
<op> ::= - | * | + | /
<digit> ::= 0 | 1 | 2 | <big_digit>
<big_digit> ::= 3 | 4 | 5 | <bigger_digit>
<bigger_digit> ::= 6 | 7 | <huge_digit>
<huge_digit> ::= 8 | 9
<var> ::= x
```

(a)

```
<expr_root>::= <guard> -- <param>
<guard> ::= (<var> < i) |
  (<var> % i) == 0 |
  (TRUE)
<param> ::= i <op> <var> | <var> <op> i |
  <var> - i
<op> ::= * | + | /
<var>::= x
```

(b)

Fig. 3. Phase one grammars (a) for fixed numbers of calls and (b) for calls in a loop

base cases based on the number of calls. Since the tree provided by the user is not required to be complete, these initial guesses could be wrong and so the user is asked to confirm and correct the number of call and base cases.

Two sets of grammars are produced. The basic grammar choices for phase one are shown in Fig 3. Part (a) shows the grammar for the parameters when the number of calls is fixed. Note there is only one variable allowed: the input variable `x` to the recursive function itself. Part (b) shows the grammar used when the calls take place in a loop. This grammar has two parts: a *guard*, which contains a condition determining when a recursive call can be made, and *param* forming the call's parameter.

Figure 4 shows three examples of recursive grammars used in phase two search. The bolded **param** and **guard** keywords indicate where the code from phase one search is inserted. The examples given are the body: of a Fibonacci function (part (a)); of a simple linear-recursive function (part (b)) and of a function making calls in a loop (part (c)). Note, the grammar in part (c) assumes that the loop is bounded by a control variable, `c`, whose initial value is passed in as a parameter to a helper function: `aux`. The presence of a control variable and guard allows interesting enumerations to be expressed – beyond those found so far in the GP literature. Also note that no explicit production for `op` is given in part (c). In this case the syntax for `op` is a pairing of an operator (e.g. +) and its left-identity (e.g. 0). The variable `result` is initialised with this left-identity in the section of code abbreviated as `preamble` in part (c).

Step Three: Evolving the Recursive Parameters. Step three performs GE search to evolve parameter expressions of the recursive calls using the phase one grammar. During this search individuals are evaluated by executing them against inputs tp_i to produce a list of one or more child parameter expressions: cc_i these can then be compared to their corresponding target child list tc_i extracted in step one. A penalty is assessed in proportion to the mismatch between each tc_i and cc_i all of these penalties are summed to derive the total penalty for the individual. Thus, in our Fibonacci example, a candidate individual consisting

```
<expr_root> ::=
 if(<var> < <digit>){
   return <lit>;
 }else{
   return <expr1> <op> <expr2>;
 }
<expr1> ::= <rec1> |
 (<rec1> <op> <lit>) |
 (<lit> <op> <rec1>)
<expr2> ::= <rec2> |
 (<rec2> <op> <lit>) |
 (<lit> <op> <rec2>)
<rec1> ::= recurse(param1)
<rec2> ::= recurse(param2)
<op> ::= - | * | +
<lit> ::= <digit> | <var>
... as per phase 1 grammar...
```

(a)

```
<expr_root> ::=
 if (<var> < <digit>) {
 return <lit>;
} else {
 return <expr1>;
}
<expr1> ::= <rec1> |
  <rec1> <op> <lit> |
  <lit> <op> <rec1>
<rec1> ::= recurse(param)
<op> ::= - | * | +
<lit> ::= <digit> | <var>
... as per phase 1 grammar..
```

(b)

```
<expr_root> ::=
 aux(x,<small_digit>) --
 int aux(int x, int c){
   .. preamble ..
   if(<var> <rel> <small_digit>){
     return <lit>
   }else{
     .. preamble ...
     for(i=<rl>; i< <ru>; i++){
       if(guard)
         result =
            result <op> aux(param,i);
     }
     return result
<rl> ::= c | <lit>
<ru> ::=  (<var> + <digit>) |
  (<var> - <digit>) |
  (<digit> - <var>) | <var>
<rel> ::=  < | >
```

(c)

Fig. 4. Phase two grammar for the parameter to the recursive call for Fibonacci. The param function where the phase one grammar will be substituted is marked in bold. Note the `digit` and `var` productions are the same as for the phase one grammar.

of just one parameter expression `x-1` generates for the tp_i in (1) above the candidate child lists: $[[2], [1]]$. with [2] being generated from the input 3 and [1] being generated from 2. After candidates cc_i are generated our phase 1 evaluative function gauges the match between the cc_i and their corresponding targets tc_i. In our example this means we try to match $[1, 2]$ with [2] and $[0, 1]$ with [1]. Every match attempt generates a penalty by the **assess_match** procedure shown in Fig 5. This procedure works by repeatedly: finding the numerically closest pair of values between *targlist* and *candlist* (a process labeled *bestMatch* in Fig 5); calculating a distance penalty; and removing matched items from both lists as it goes. The main loop terminates when one of the lists is depleted. After the loop, the presence of surplus target expressions means that the candidate list didn't cover the target list (i.e. there weren't enough calls) and a large penalty is applied. Conversely, surplus candidate expressions could, benignly, indicate that user didn't supply all of the child nodes when drawing the partial tree. A small penalty is applied in proportion to the number of extra candidates. In our example, when matching candidate [2] with target $[1, 2]$ **assess_match** will match the 2's (penalty: 0) and then have [1] remaining in the target list with a total penalty of *BIG_PENALTY*. The same penalty is assessed for the match between [1] and $[0, 1]$ and penalties are summed resulting in a total penalty of $2 \times$ *BIG_PENALTY*. The values of *BIG_PENALTY* and *SMALL_PENALTY* are set so that *BIG_PENALTY* is larger than any expected difference in result values and *SMALL_PENALTY* is much smaller than 1.0. The penalties are summed to form an evaluative score for an individual. Evolution proceeds until either an

```
**assess_match**(targlist, candlist)
    $penalty = 0$;
    **while** $(|targlist| > 0 \wedge |candlist| > 0)$
        $(targlist_i, candlist_j) = bestMatch(targlist, candlist)$
        $penalty = penalty + |targlist_i - candlist_j|$
        $targlist = targlist \setminus targlist_i$
        $candlist = candlist \setminus candlist_j$
    **end while**
    $penalty = penalty + |targlist| * BIG\_PENALTY$
    $penalty = penalty + |candlist| * SMALL\_PENALTY$
    **return** $penalty$
**end**
```

Fig. 5. Procedure to assess match between target and candidate lists

individual with zero penalty is found or a set number of generations has elapsed. Note, that for grammars with more than one recursive call, we run step-three search once for each recursive call, excluding previously found solutions as we go. Also note that for grammars with loops we apply the guard to restrict the calls made but we have to assume large loop bounds. Loop-bounds will not be evolved precisely until phase two. This can lead to surplus candidate calls and the small fitness penalty that this entails.

4 Experimental Setup

In our experiments, we compare the search performance of CTGGP to standard GE. The benchmarks for our experiments are, for an integer parameter (n): *factorial* returns the factorial of n; *odd-evens* returns 0 if $n \bmod 2 = 0$ and 1 otherwise; *log2* finds $\lfloor \log_2 n \rfloor$; *fib* and *fib3* calculates the Fibonacci and Fibonnaci-3 number for n; *lucas* calculates the nth Lucas number; *factorings* returns the number of unique factorings of n. *sums* returns the number of unique sum-decompositons of n. These latter two benchmarks are not trivially coded by humans and are not found elsewhere in the literature on recursive GP.

The input is a small tree drawn by CTGGP's GUI. Trees for three benchmarks are shown in Fig 6. Note how in part (a) and part (b) we have shared some child nodes between sub-trees and in (a) we have included some disjoint single node trees. Trees for our other benchmarks are of very similar size to these.

In both phases of evolution we used GE running on an underlying steady-state GA with tournament selection. The replacement probability used was 0.25 and probabilities for crossover and mutation were, respectively, 0.9 and 0.01. In both phases individuals were evaluated by using scripts to insert evolved code into test harnesses and running Tiny-C-Compiler [2] (TCC) to quickly generate binaries.

For phase one evolution we used small population of 100 individuals running for ten generations. For phase two we ran with populations of 200 for the

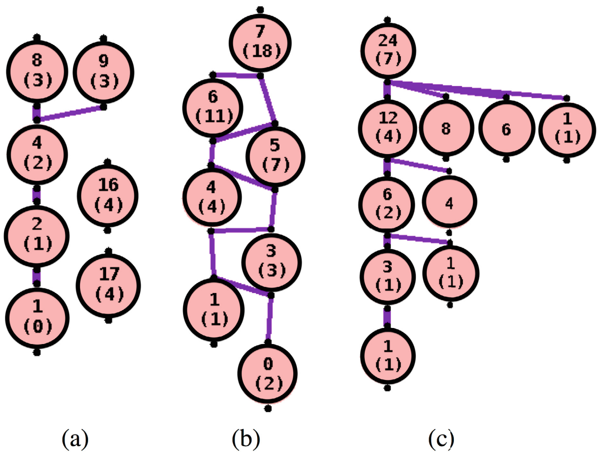

Fig. 6. A selection of input trees: log2a, part (a), lucas part (b), factorings part (c)

smaller *factorial*, *log2* and *oddeven* benchmarks, and with 1000 for the remaining benchmarks. All phase two experiments were run for 100 generations.

As a control, we ran all benchmarks in a single conventional GE phase with with a grammar including the recursive case clauses used in phase one. Note, to better expose the impact of CTGGP these larger grammars were specialised to each benchmark so that the only difference between the conventional GE and the CTGGP runs is the former is required to evolve the recursive parameter code along with the rest of the code[3]. When evaluating CTGGP we sum the total number of evaluations required for *both* phases. All benchmarks were run on a 2.4GHz Intel core i7 with 4GB of RAM.

5 Results and Discussion

We ran both phases of CTGGP on all benchmarks 100 times. We did likewise for our single phase comparison benchmarks for conventional GE. Phase one of CTGGP succeeded in finding the correct code for parameters in all benchmarks in all runs.

The results comparing the combined cost of phase 1 and phase 2 of CTGGP with conventional GE are shown in Table 1 The columns show, respectively, the mean number of evaluations, the sample standard deviation of the evaluation count, and the percentage of runs yielding a correct result for conventional GE and CTGGC. Note, that because we limit experiments to 100 generations the value of $\overline{\mathbf{x}}$ is a lower bound when not all runs are correct. Statistical significance was assessed using a log-rank test to take account of this truncation.

[3] This is perhaps generous to conventional GE which would not be able to select the correct grammar in the absence of tree information.

Table 1. Mean number of evaluations and number of correct answers for raw GE and CTGGP. Significantly better ($p << 0.01$) means are marked in bold.

problem	**Conventional GE**			**CTGGP**		
	$\overline{x}$	σ	**correct**	$\overline{x}$	σ	**correct**
factorial	1984	1461	99	**436**	79	100
oddevens	507	343	100	484	160	100
log2	7756	2138	24	**3618**	3520	81
fib2	32933	10700	53	**2156**	256	100
fib3	31375	3437	3	**7863**	4852	100
lucas	32012	10577	40	**11713**	7736	99
factorings	28266	15695	60	**1937**	1071	100
summands	38912	7611	26	**24926**	11642	91

The data from our runs show that, in most benchmarks, CTGGC, significantly out-performs conventional GE both in terms of the reduced number of evaluations required and the number of times a benchmark was correctly evolved. The only benchmark that did not show a significant improvement was *oddeven* which presented an easier target than other benchmarks, partly because it admits a reasonable diversity of solutions. In contrast, the *log2* benchmark exhibited a tendency to prematurely converge toward locally strong solutions. Likewise, *summands* exhibited similar tendencies as well being sensitive to the upper bound of its loop.

Another observation to be made was the high variance in the number of evaluations required. In CTGGP this is caused by the significant number of runs which found solutions in the first one or two generations.

In terms of experimental run times we observed TCC to be fast and we set the timeouts for phase two runs to be small so the average evaluation time for an individual in both phases is 2 millseconds. Runtimes for phase one evolution varied from less than five seconds to just over a minute. Runtimes of phase two evolution varied from less than 20 seconds (for factorial) to several minutes (for lucas). A final observation to make is that all of the trees used in these experiments were very easy to draw. This ease of use and the short runtimes are positive indicators for GTGGP's future implementation as a practical tool.

6 Conclusions and Future Work

In this article we have shown that incorporating call-tree information into the GP search process can significantly improve performance at only a small cost in terms of human effort. The work here is most applicable where code is required to implement a completely unknown recurrence and the drawing of a partial call-tree is a natural part of the exploratory process. The usefulness of CTGGP is in automating the non-trivial step of deriving code from this partial call-tree.

This work can be enhanced in several ways. We could exploit the relationships implicit in return values to separately derive code combining results of recursive calls. We could exploit existing libraries of sequences to express more complex recurrences in loops. We could combine Moraglio's technique for discovering base-cases with ours. Finally, we could integrate hand-drawn-graph-recognition software into CTGGP to allow direct derivation of recursive code from paper sketches, completing the chain from pictures to programs.

References

1. Agapitos, A., Lucas, S.: Learning recursive functions with object oriented genetic programming. In: Collet, P., Tomassini, M., Ebner, M., Gustafson, S., Ekárt, A. (eds.) EuroGP 2006. LNCS, vol. 3905, pp. 166–177. Springer, Heidelberg (2006)
2. Bellard, F.: Tcc: Tiny c compiler (2003), `http://fabrice.bellard.free.fr/tcc`
3. Hofmann, M., Kitzelmann, E., Schmid, U.: A unifying framework for analysis and evaluation of inductive programming systems. In: Proceedings of the Second Conference on Artificial General Intelligence, pp. 55–60 (2009)
4. Kitzelmann, E., Schmid, U.: Inductive synthesis of functional programs: An explanation based generalization approach. The Journal of Machine Learning Research 7, 429–454 (2006)
5. Koza, J.R., Andre, D., Bennett III, F.H., Keane, M.: Genetic Programming 3: Darwinian Invention and Problem Solving. Morgan Kaufman (April 1999)
6. Moraglio, A., Otero, F.E.B., Johnson, C.G., Thompson, S., Freitas, A.A.: Evolving recursive programs using non-recursive scaffolding. In: IEEE Congress on Evolutionary Computation, pp. 1–8 (2012)
7. Nishiguchi, M., Fujimoto, Y.: Evolution of recursive programs with multi-niche genetic programming (mngp). In: Proceedings of the 1998 IEEE International Conference on Evolutionary Computation, pp. 247–252 (1998)
8. O'Neill, M., Ryan, C.: Grammatical evolution. IEEE Trans. Evolutionary Computation 5(4), 349–358 (2001)
9. Spector, L., Robinson, A.: Genetic programming and autoconstructive evolution with the push programming language. In: Genetic Programming and Evolvable Machines, pp. 7–40 (2002)
10. Wong, M.L., Leung, K.S.: Evolving recursive functions for the even-parity problem using genetic programming. In: Advances in Genetic Programming, pp. 221–240. MIT Press (1996)
11. Yu, T., Clark, C.: Recursion, lambda-abstractions and genetic programming. Cognitive Science Research Papers-University of Birmingham CSRP, 26–30 (1998)