Self-Improving Genetic Programming

Roland Olsson
Østfold College
1750 HALDEN
Norway
Roland.Olsson@hiof.no http://www-ia.hiof.no/~rolando

Abstract: The paper dicusses how ADATE, a system that is not GP but similar in spirit, can automatically improve itself by synthesizing parts of its own code. Great care must be taken to avoid overfitting to training data during such self-improvement. ADATE is based on Kimura's generally accepted theory of neutral mutations in evolutionary biology. Self-improvement is discussed both for neutral mutations, also called semantics-preserving program transformations, and for more "brutal" mutations that almost always can be quite small.

Keywords: Program synthesis, automatic programming, self-adaption, neutral random walks

1 Introduction

A self-improving genetic programming (SIG) system modifies parts of itself to become better at automatic programming. In other words, the system produces an improved system which may produce another even more improved system which may produce yet another improved system and so on.

In this paper, we exhibit a simple form of self-improvement for Automatic Design of Algorithms Through Evolution (ADATE) [6]. ADATE primarily relies on neutral "mutations", aiming at exploration through walks along neutral networks in program space. Note that neutral mutation is just another name for semantics-preserving program transformation, which is very well studied in compiler design. Some of ADATE's transformations, for example function invention (ABSTR) [7], were actually found by studying compilation of purely functional programs.

In general, there are many different ways to self-improvement of mutation operators. Here are some alternatives ordered according to the complexity of the part of a mutation operator that needs to be automatically synthesized.

1. Direct automatic synthesis of the entire code used for mutation.
2. Synthesis of classification code that ranks synthesized expressions according to their expected usefulness.
3. Reshaping the distribution of synthesized expressions so that they cover as many equivalence classes as possible with a limited number of syntheses and without unnecessary increase in expression size.
4. Automatic synthesis of semantics-preserving rewriting rules that are employed for neutral random walks.
5. Adapting numerical parameters such as overall mutation frequency and conditional or unconditional probabilities of occurrence for given functions and constants.

A major problem with alternatives one, two and five is to find so many relevant training inputs i.e., fitness cases, that the automatically synthesized mutation operator is not specialized to them. Thus, overfitting is difficult to overcome for alternatives one, two and five.

The complexity of the self-improvement required for alternative five is smaller than that required for alternative two, which in turn is less complex than alternative one. Due to Occam's razor, the problem of finding training inputs becomes easier

to solve as we move from alternative one to two and from two to five.

However, alternatives three and four do not suffer from a similar generalization problem since much relevant training data is easy to find. Consider a mutation operator for a specific set of functions and constants. With such a set, we synthesize at least one thousand expressions and label them according to the output they produce. The ones with the same output for all training inputs are regarded as equivalent and receive the same label. Note that such equivalence classes is all that is needed for fitness computation in alternatives three and four.

Below, we experimentally show self-improvement according to alternative three for the synthesis of boolean expressions using the functions `not`, `and` and `or`. However, due to the overwhelming importance of neutrality in both natural and artificial evolution, alternative four is more promising in general, but we have not yet examined it experimentally.

We will refer to alternative three as SIG-reshaping. Alternative four will be called SIG-rewriting.

The next section gives a brief overview of related work. Section three presents theory and experiments for SIG-reshaping. SIG-rewriting is discussed in section four. The final section contains preliminary conclusions.

2 Literature overview

Our inspiration for studying self-improvement comes from running n-dimensional numerical optimization experiments on self-adaption using Evolution Strategies (ES) [8, 9]. An ES individual is mutated by first randomly changing mutation step sizes and then employing these new step sizes to mutate the object variables. Thus, good step-sizes indirectly have a higher probability of surviving since they are more likely to lead to a positive mutation.

We tried an analogous experiment in GP, where each individual carried rejection rules for mutations instead of step sizes. However, this approach failed completely due to over-specialization to the training data i.e., the genealogical history trace.

Lee Spector [11] is trying an even more radical form of self-adaption, Pushpop, where each individual contains the code necessary for its own reproduction and diversification. However, his experimental results are preliminary and it remains to be seen if Pushpop will run into the same over-specialization quagmire as we did.

Parts of our work was inspired by Kimura [4] who is the most famous proponent of the neutral theory in evolutionary biology for the past 30 years. He argues that a substantial fraction of all natural mutations are neutral and that very few of the other ones are positive i.e., increase fitness. Neutral mutations enable evolution to explore fitness plateaus and find suitable jump points where it is easy to move to a higher plateau.

Shipman [10] et. al. discuss redundant genotype-phenotype mappings and how genotypes form neutral networks that make it possible to explore genotype space using the simplest of mutations.

3 SIG-reshaping

The primary goal with SIG-reshaping is to avoid trying too many equivalent expressions, for example synthesizing and using both `E` and `not(not(E))` or both `or(`E_1`,`E_2`)` and `or(`E_2`,`E_1`)` for arbitrary boolean expressions `E`, E_1 and E_2.

The space of all expressions with a size not exceeding a given limit is unsuitable for uniform random sampling if some equivalence classes have huge cardinalities, whereas other classes are quite small and very rarely sampled even though they contain desirable expressions.

SIG-reshaping aims at alleviating this oversampling of huge equivalence classes by automatically synthesizing an acceptance predicate that determines if a given synthesized expression is used. For example, such a predicate can choose to reject `not(not(E))` and also `or(`E_1`,`E_2`)` if E_2 is less than E_1 according to some total ordering of expressions.

We wrote an ADATE specification for synthesis of an acceptance predicate as follows.

First, we used ADATE's expression synthesis subroutine to generate all 1055 boolean expressions of size five or less consisting of `not`, `and`, `or`, `false`, `true` and three variables `X1`, `X2`, `X3`. Then, we evaluated each expression for all eight possible values of the input (`X1`, `X2`, `X3`) and labeled each expression with the equivalence class that it belongs to. There are 36 equivalence classes with highly varying cardinalities for this expression space.

For example, the equivalence class that was given number 36 contains only the following four expressions, here shown with the label.

```
( 36, and( X3, or( X1, X2 ) ) )
( 36, and( or( X2, X1 ), X3 ) )
( 36, and( or( X1, X2 ), X3 ) )
( 36, and( X3, or( X2, X1 ) ) )
```

Our web site, given at the top of the first page, contains an ADATE specification file for SIG-reshaping as well as the source code of ADATE itself, which should make it easy to reproduce the results below.

The fitness function requires that a synthesized acceptance predicate accepts at least one minimum size expression in each equivalence class and rejects as many other expressions as possible. We first tried without the minumum size requirement but then found that the synthesized predicates sometimes only accepted the biggest member of a class, which is undesirable in general due to Occam's razor.

Given this fitness function and a total ordering on expressions, ADATE generated an acceptance predicate that rejects 999 out of the 1055 expressions in the space while still accepting at least one minimum size member of each class. This predicate was fairly easy for ADATE to produce, requiring only 15 hours of CPU time on our 16-node Beowulf cluster with 800 MHz Pentium III processors. The synthesized predicate `f` contains one automatically invented help function which is used together with `f` in a mutually recursive fashion. It is not yet clear how the code for `f` works. Explanations from readers of this paper are welcome.

To test if this acceptance predicate leads to self-improvement, we ran four, five and six bit xor problems, also called even parity, both without and with the predicate. In the latter case, the predicate was used to reject boolean expressions synthesized by ADATE's only non-neutral "mutation" operator, the so-called R-transformation, which is quite different from a standard GP mutation operator.

For example, ADATE does not use randomization in any way, which means that only one run was performed for each example in table 1. ADATE systematically generates expressions in order of increasing size, which is not as combinatorially unreasonable as it may seem since neutral walks in program space typically lead to a program that only needs an extremely small change to be improved. Amazingly, ADATE can generate code for the 6-bit even parity problem synthesizing only expressions of size three or less!

Specification	$N = 0$	$N = 1$	$N = 2$
xor4	$3.2 \cdot 10^5$	$3.1 \cdot 10^4$	$2.5 \cdot 10^4$
xor4sig	$8.2 \cdot 10^4$	$4.0 \cdot 10^4$	$8.5 \cdot 10^4$
xor5	$3.5 \cdot 10^6$	$9.6 \cdot 10^6$	$2.4 \cdot 10^6$
xor5sig	$4.0 \cdot 10^6$	$1.9 \cdot 10^6$	$2.5 \cdot 10^6$
xor6	$4.8 \cdot 10^7$	$1.5 \cdot 10^7$	$1.3 \cdot 10^7$
xor6sig	$1.9 \cdot 10^7$	$6.9 \cdot 10^6$	$1.7 \cdot 10^6$

Table 1: Total number of evaluations required for finding a 100% correct program.

The parameter N in table 1 indicates the degree of population redundancy. ADATE maintains a chain of bigger-and-better so-called base individuals. Each base is the smallest member found so far in the neutral network that it represents. ADATE's population contains about $4N$ representatives of the neutral network for a base in addition to the base itself.

As can be expected from a complex evolutionary process, the variance in table 1 is high, but it appears that self-improvement has taken place. A remarkable result from the table is that the SIG version of the six bit xor problem found a correct program without function invention using only 1.7 million evaluations. This may be one of the best results reported in GP literature for this problem without invented functions. For example, Koza [3] and Chellapilla [2] report results without ADFs only for the three, four and five bit even parity problems but not for the six bit problem. Note that ADFs are different from the ABSTR transformation in ADATE.

The performance advantage of the SIG versions is smaller than anticipated since the removal of redundant boolean expressions decreases the connectivity of neutral networks which may lead to missing links in the most continuous genealogical chains.

4 SIG-rewriting

In ADATE, as well as in natural evolution, neutral walks in genotype space are essential for avoiding combinatorial explosions due to complex muta-

tions. For a given mutation complexity limit, the set of negative mutations usually has much higher cardinality than the set of neutral mutations which in turn is bigger than the set of positive mutations.

Combinatorially, it is typically much cheaper to move through a sequence of neutral mutations finished by a minute positive mutation than to directly search for a single more complex positive mutation.

ADATE's so-called compound transformations [6] were designed according to this principle and consist of a large neutral part followed by a small and raw combinatorial search for a positive i.e., improving, transformation. The neutral transformations ABSTR, CASE-DIST and DUPL in ADATE are formulated as semantics-preserving rewrite rules.

ABSTR is basically a non-deterministic inverse of the inlining (β-expansion) transformation used by optimizing compilers. CASE-DIST changes the scope of `case`- and `let`-expressions. DUPL inserts a case-test with all alternatives equal to a chosen expression in the program and does not change semantics as long as the analyzed expression does not raise exceptions.

These three semantics preserving transformations are general and probably difficult to improve automatically. We will now outline how ADATE can invent rewrite rules for problem-specific transformations that were not anticipated when we designed ADATE. This is SIG-rewriting and can be implemented in a way similar to a so-called sequential covering algorithm [5] for learning sets of propositional or first-order rules.

ADATE is used instead of the LEARN-ONE-RULE algorithm called as a subroutine by the main sequential covering algorithm. Each call to LEARN-ONE-RULE corresponds to a complete ADATE run that produces a program capable of transforming some expressions so that their semantics is preserved.

For example, if boolean expressions are considered, such an ADATE run may yield a program like

```
fun f E = not( not E )
```

This rewrite rule obviously increases the size of a program and can lead to bloating. Another run may then give the program

```
fun f E =
  case E of
    and( E1, E2 ) => and( E2, E1 )
  | _ => ?
```

This program only knows commutativity for `and`. It outputs ADATE's built-in `?` constant, meaning "don't know" and implemented as exception raising, for all expression trees not having `and` in the root.

The ADATE specification for producing boolean rewriting rules is similar to the one used for boolean SIG-reshaping with the same 36 equivalence classes. But these classes are now expanded to include all expressions up to size seven, However, only expressions of size up to five are employed as training inputs. This enables the synthesis of size-increasing rewriting rules.

The fitness function requires that each output is in the same equivalence class as the input and measures the number of input-outputs that are not covered by any previously generated rule.

Before generating children from a parent program using ADATE's R transformation, the parent is modified through a neutral random walk using the automatically synthesized rewrite rules. Each step in this walk consists of choosing a random subexpression and changing it according to a randomly chosen rule. The walk is terminated as soon as the current program is more than 50% bigger than the parent or when one thousand steps have been taken. The program to be subjected to compound ADATE transformations is randomly chosen among all intermediate programs visited during the walk.

Provided that all rewrite rules preserve semantics, the only drawback with the walk is the increase in size, i.e., bloating.

5 Conclusions

In both natural and artificial evolution, genotype space consists of neutral networks such that all genotypes in a network have practically equal fitness. The most important question in evolutionary theory is how to make the transition from one neutral network to another that has higher fitness. Artificial evolution should proceed through a sequence of such transitions between neutral networks. Note that a neutral network roughly corre-

sponds to a species in nature and that crossover should occur only between members of a neutral network instead of the usual haphazard GP crossover.

The transition to a new neutral network i.e., a new species, consists of the following.

1. A neutral walk along the current network aiming at visiting genotypes located close to a new network.

2. A brutal but extremely small mutation that is not neutral and bridges the small gap to the new network from a suitable jump point in the current network.

The compound transformations in ADATE were designed according to this model of transition from one species to the next. In ADATE, the first part of a compound program transformation is the neutral walk and the second, typically small, part is the brutal mutation.

We have discussed self-improvement for both parts. The self-improvement method for part two is SIG-reshaping that changes the mutation distribution whereas SIG-rewriting is designed to increase the number of connections in a neutral network and thereby increase the number of genotypes reachable with a neutral walk.

The experimental data presented above is inadequate for firm conclusions about the usefulness of self-improvement. However, the results are hopefully strong enough to motivate the international evolutionary computation community to pursue similar lines of research.

The ADATE source code is freely available and can serve as a test bench for numerous other experiments on neutral evolution and self-improvement than the small ones reported in section three.

References

[1] P. Angeline, Adaptive and Self-Adaptive Evolutionary Computations, Computational Intelligence: A Dynamic Systems Perspective, IEEE Press, 1995.

[2] K. Chellapilla, A Preliminary Investigation into Evolving Modular Programs without Subtreee Crossover, Proceedings of The Third Annual Genetic Programming Conference, Morgan Kaufmann, 1998.

[3] J. R. Koza, Genetic Programming II: Automatic Discovery of Reusable Subprograms, MIT Press, 1994.

[4] M. Kimura, Population Genetics, Molecular Evolution, and the Neutral Theory: Selected papers, University of Chicago Press, 1994.

[5] T. M. Mitchell, Machine Learning, WCB/McGraw-Hill, 1997, page 276.

[6] J. R. Olsson, Inductive functional programming using incremental program transformation, Artificial Intelligence, Vol. 74, No. 1, 1995, pages 55–83.

[7] J. R. Olsson, How to Invent Functions, European workshop on genetic programming, Springer Verlag, 1999.

[8] I. Rechenberg, Evolutionsstrategie, Optimierung technischer Systeme nach Prinzipien der biologischen Evolution, Frommann-Holzboog, 1973.

[9] H. P. Schwefel, Numerical Optimization of Computer Models, Wiley, 1981.

[10] R. Shipman, M. Shackleton, M. Ebner and R. Watson, Neutral Search Spaces for Artificial Evolution: A Lesson from Life, Proceedings of the Seventh International Conference on Artificial Life, MIT Press, 2000.

[11] L. Spector, Autoconstructive Evolution: Push, PushGP, and Pushpop, Proceedings of the Genetic and Evolutionary Computation Conference (GECCO), Morgan Kaufmann, 2001.