Self-Improvement for the ADATE Automatic Programming System

Roland Olsson
Computer Science Dept.
Østfold College
1750 HALDEN
Norway
http://www-ia.hiof.no/∼rolando

Brock Wilcox
NAU Box 8087
Flagstaff, AZ 86011
USA
rbw3@cet.nau.edu
928 - 523 - 4903

Abstract

Automatic Design of Algorithms Through Evolution (ADATE) is a system for automatic programming based on the neutral theory of evolution. This work examines methods of self-improvement for the ADATE system. Experiments are performed testing two of the suggested methods, named SIG-rewriting and SIG-reshaping. SIG-rewriting synthesizes transformations to explore neutral networks, while SIG-reshaping creates a predicate for accepting or rejecting non-neutral mutations. It is found that SIG-rewriting has created no significant improvement for the ADATE system, while SIG-reshaping does have improved results.

1 Introduction

Automatic Design of Algorithms Through Evolution (ADATE) [8] is a system for automatic programming based on the neutral theory of evolution [4, 5]. This theory states that the majority of molecular changes in evolution are due to neutral or almost neutral mutations. A consequence is that most of the variability and polymorphism within a species comes from mutation-driven drift of alleles that are selectively neutral or nearly neutral. Since Darwinism and neo-Darwinism state that survival of the fittest is the most important source of genetic variation, the paper by King and Jukes [5] has the remarkable title "Non-Darwinian evolution". The neutral theory is now generally accepted and is supported by studying the rates of synonymous and non-synonymous substitution in various organisms [6].

Section 2 gives a brief overview of ADATE and explains why neutral walks provide performance critical combinatorial gains. The next section discusses self-improving genetic programming (SIG) and outlines five ways to self-improvement of mutation operators. Sections 4 and 5 describe experiments with two of these five SIG methods. The final section contains conclusions and future work considerations.

2 Overview of ADATE and its application of the neutral theory

In ADATE, as well as in natural evolution, neutral walks in genotype space are essential for avoiding combinatorial explosions due to complex mutations. 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. Neutral mutations enable evolution to explore fitness plateaus and find suitable points where it is easy to reach a higher plateau of fitness. Typically, long jumps are combinatorially expensive and should be avoided.

Given a parent program, consider the set of all possible mutations of some maximum complexity. The set is divided into three separate partitions by considering whether the mutation had a negative, neutral, or positive effect on the fitness of the program. The negative partition usually has higher cardinality than the neutral one, which in turn is bigger than the positive partition. Often, cardinalities decrease by many orders of magnitude when moving from the negative partition to the neutral one or from the neutral to the positive.

Thus, it is much more difficult to directly find a positive mutation than to find a neutral one. However, by applying a sequence of neutral mutations to the parent program, one can almost always find a fitness-equivalent program for which a quite small positive mutation exists.

Shipman [12] et. al. use the concept of neutral networks instead of fitness plateaus. In both natural and artificial evolution, genotype space consists of neutral networks such that all genotypes in a network have practically equal fitness and are on the same fitness plateau.

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

ADATE primarily relies on neutral mutations, exploring program space through walks along neutral networks. The transition from one neutral network (i.e., species) to another consists of the following.

1. A neutral walk along the current network attempting to visit 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 will now briefly discuss the neutral program transformations (i.e., mutations) in ADATE and classify them as fitness neutral or semantically neutral. Of course, a transformation can be fitness neutral without preserving semantics whereas semantic neutrality implies fitness neutrality. ADATE uses these transformations to perform its walk along a neutral network.

ADATE's semantically neutral transformations are abstraction (ABSTR), duplication (DUPL) and `case` or `let` distribution or lifting (CASE-DIST).

ABSTR invents new functions and 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.

The only fitness neutral, but not necessarily semantics neutral, transformation in ADATE is "replacement preserving equality" (REQ), which mutates a subexpression and computes fitness to ensure that it does not decrease. Note that neutral REQs are much more frequent than positive ones but the latter are not discarded even if they happen to increase fitness. ADATE has a recombination system for exploring various combinations of established REQs, but the details are beyond the scope of this paper.

The population management in ADATE maintains a chain of bigger-and-better "base" individuals. Each base is the smallest member found so far in the neutral network i.e., species, that it represents. Other members of a species are retained using various criteria such as syntactic complexity, time complexity, output diversity and genealogical diversity. The last criterion is intended to encourage wide exploration of the neutral network.

3 How to automatically improve mutation operators

Our inspiration for studying self-improvement comes from running n-dimensional numerical optimization experiments on self-adaption using Evolution Strategies (ES) [10, 11]. 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 overfitting to the training data i.e., the genealogical history trace.

Lee Spector [13] 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 overfitting quagmire as we did.

In general, there are many different paths to self-improvement of mutation operators. Here are some possibilities ordered according to the complexity of the automatically synthesized portion of the mutation operator.

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. Such rules increase the number of connections in a neutral network and thereby also the number of genotypes reachable through a neutral walk without fitness computation.
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.

We will refer to alternatives three and four as SIG-reshaping and SIG-rewriting respectively.

4 SIG-rewriting

Since ADATE is optimized to exploit neutrality, it is more difficult to improve ADATE using SIG-rewriting. An advantage of automatically synthesized rewrite rules, however, is that they can be optimized to the specific problem in a way that could not be anticipated when we designed ADATE.

We implemented SIG-rewriting in a way similar to a sequential covering algorithm [7] for learning sets of propositional or first-order rules. ADATE was 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.

We stopped after generating nine rewrite rules, but could have continued generating more or less trivial rules practically ad infinitum. It would be advantageous to include some estimate of rule usefulness in the fitness function, but such estimation seems computationally too demanding using our current hardware. When running ADATE with and without these nine rules on a few boolean problems, we did not detect any major performance difference.

5 SIG-reshaping

The primary goal of 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 minimum size requirement but then found that the synthesized predicates sometimes only accepted the biggest member of a class, causing an undesirable increase in complexity.

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 with and without the predicate. In the former 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.

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's population contains about $4N$ representatives of the neutral network for a base in addition to the base itself since approximately N individuals are kept for each of four different neutral selection strategies.

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 ADATE SIG version with $N = 2$ found a correct program without function invention for the six bit xor problem 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.

6 Conclusions and future work

This paper has taken a look at several methods of self-improvement for the ADATE system. Two of these methods were tested experimentally, SIG-rewriting and SIG-reshaping.

SIG-rewriting demonstrated no significant improvement on the performance of the ADATE system. The rules generated for SIG-rewriting are semantics preserving transformations in addition to the ones that ADATE already employs. The disappointing results may be due to this overlap in functionality.

SIG-reshaping, however, has demonstrated some self-improvement. After generating an acceptance predicate for boolean expressions, limiting the duplication of equivalent expressions, the predicate was tested on four, five, and six bit xor problems. Comparing the results with and without the predicate indicate that self-improvement has occurred.

Preliminary examination might suggest that SIG-rewriting and SIG-reshaping have opposite goals. SIG-rewriting seeks to increase the number of transforms available for the walk along a neutral network, creating more paths available for walking. SIG-reshaping accepts or rejects the results of non-neutral mutations, preferring the less complex of an equivalent set of expressions. Though decreasing the number of accepted non-neutral mutations seems to contradict the above increase in paths, this makes it so that the entry point to a higher-fitness neutral network is less complex than it would be otherwise, Occam's razor at work. Thus the two approaches do not counter-act one another, rather they appear to complement one another.

Increasing the number of neutral transforms available to ADATE alone hasn't wielded any beneficial results, as stated above. Including a measure of rule usefulness to decide on the value of a synthesized neutral transform may make SIG-rewriting much more useful, but is unfortunately very computationally expensive. The results, however, may be SIG-rewriting rules that are more likely to generate expressions closer to points where mutation to a higher fitness plateau is possible. One future experiment to be performed is testing the above hypothesis by actually spending the computation time needed to create refined SIG-rewrite rules based on their usefulness.

Experimentation with function sets more complicated than boolean expressions will also be performed to test the scalability of these self-improvement techniques. The main problem with complicated function sets will be generating a reasonable set of sample inputs to demonstrate the validity of generated self-improvement routines using either of the above methods.

References

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

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

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

[4] M. Kimura (1968). Evolutionary rate at the molecular level, *Nature* **217** : 624–626.

[5] J.L. King and T. H. Jukes (1969). Non-Darwinian evolution. *Science* **164** : 788–798.

[6] W. H. Li (1997). *Molecular Evolution.* Sinauer Associates.

[7] T. M. Mitchell (1997). *Machine Learning.* WCB / McGraw-Hill. Page 276.

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

[9] J. R. Olsson (1999). How to Invent Functions, *European workshop on genetic programming.* Springer Verlag.

[10] I. Rechenberg (1973). *Evolutionsstrategie, Optimierung technischer Systeme nach Prinzipien der biologischen Evolution.* Frommann-Holzboog.

[11] H. P. Schwefel (1981). *Numerical Optimization of Computer Models.* Wiley.

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

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