# Population Management for Automatic Design of Algorithms through Evolution

Roland Olsson

Department of Computer Science, Østfold College, Os Allé 11, 1757 Halden, Norway
(Roland.Olsson@hiof.no)

## Abstract

The paper describes population based search in the system Automatic Design of Algorithms through Evolution (ADATE) that maintains chains of gradually bigger and better programs. The main challenge is to avoid missing links that lead to entrapment in too bad local optima. To avoid entrapment, the ADATE system employs iterative re-expansion of programs, population maintenance using a syntactic complexity / evaluation value ordering scheme and three different diversification methods that strive to avoid too similar programs.

When combined with the general program transformations explained in a previous paper, these techniques enable ADATE to synthesize recursive programs with automatic invention of recursive help functions. We also briefly present experimental results supporting that automatic synthesis of complex programs from "first principles" is possible indeed, but only if vast computational resources are employed effectively.

## 1 Introduction

Automatic Design of Algorithms through Evolution (ADATE) is a methodology that gradually builds more and more complex functional programs. General information and executable code are available through [7, 8, 9]. ADATE is not based on any previous machine learning or evolutionary computation technique and does not fit into the framework for Evolutionary Algorithms (EAs) given in [2], for example. Nevertheless, ADATE has been inspired by natural evolution and should be regarded as a part of Evolutionary Computation.

The primary contribution of the paper is how to choose which individuals that should be allowed to stay in the population, that varies in size dynamically. An individual is a purely functional program together with additional information such as the identities of ancestors and their evaluation values. Both the synthesized programs and ADATE itself are written in Standard ML, which we also use when presenting code below.

The goal of a run of the ADATE system is to produce a genealogical chain of programs $P_1 - P_2 - \ldots - P_m$ such that $P_1$ is the empty initial program, $P_m$ is a desirable program and each $P_i$ is produced from $P_{i-1}$ using a so-called compound program transformation [8]. Please note that such a transformation has essentially nothing in common with "mutation" in other evolutionary techniques and that a discussion of transformations is outside the scope of this paper.

Typically, most of the programs in a genealogical chain are in order of increasing syntactic complexity i.e., size. Even though many chains contain regressing subchains of the form $P_{k-1} - P_k$, where $P_k$ is smaller than $P_{k-1}$, the general tendency is that a child $P_i$ is both a little bigger and a little better than its parent $P_{i-1}$.

Section 2 describes why and how ADATE tries to keep most of the programs in an interesting chain in the population, which is the basic population management technique. Section 3 discusses how additional programs are added to the population in order to avoid getting stuck on local optima, for example due to a phenomenon similar to loss of diversity in other EAs [2]. In ADATE, most of the techniques for avoiding such premature convergence are based on measures of the difference between individuals. These are heuristic and imprecise since program equivalence in general is undecidable. Section 4 briefly presents experimental results. ADATE can synthesize general recursive programs and invent recursive help functions as they are needed, which is beyond the abilities of genetic programming [4], for example.

## 2 Basic Population Management

Several general search techniques [10], for example tabu search and simulated annealing, are variants of neigh-

```
fun genealogical_search Pop =
  let
    val L =
      min { L : There is a program in Pop such
        that L is the last cost limit used to
        expand that program. }
    val Ps =
      { P : P is a program in Pop such that L
        is the last cost limit used to expand
        P. }
    val P = A program in Ps of minimum
      syntactic complexity.
    val Children =
      The result of expanding P using a cost
      limit of max( 10000, 3.6 * L ).
    fun new_pop( [], Pop ) = Pop
      | new_pop( Child :: Children, Pop ) =
          new_pop( Children,
            modify_pop( Child, Pop ) )
  in
    genealogical_search(
      new_pop( Children, Pop ) )
  end
```

Figure 1: The overall search in ADATE.

bourhood search, also called local search. In ADATE, the neighbourhood of a given parent program $P$ is the set of all children programs that can be produced from it using a cost limit $L$ that specifies the desired neighbourhood cardinality. The cost of a transformation indicates the complexity or "size" of the transformation as explained in [7]. We will call the process of producing the children of $P$ an expansion of $P$.

It is desirable to keep a major part of a chain $P_1 - P_2 - \ldots - P_m$ in the population since ADATE may want to re-expand one or more of the programs in the chain with a greater cost limit. This iterative re-expansion strategy was derived from iterative-deepening [3, 6].

We will later discuss a function `modify_pop` that determines the new population given a child and the current population. This section discusses the basic definition of `modify_pop`, which will be extended in Section 3.

Figure 1 shows how programs in the population are selected for expansion and also the iterative-deepening of the transformation cost limit. Just after starting the ADATE system, it executes the call

```
genealogical_search Initial_population,
```

where `Initial_population` only contains the initial program. This call is non-terminating, giving the user of the ADATE system the responsibility of ending execution when programs with sufficiently good evaluation values have been synthesized.

In Figure 1, `P` is the selected parent i.e., the next program to be expanded. The cost limit used to expand `P` is chosen to 3.6 times the previous limit with an initial limit of 10000. The list `Children` contains all programs resulting from the expansion of `P`. The new population is produced by the call `new_pop( Children, Pop )`, which considers one child at a time for inclusion in `Pop` using the function `modify_pop`. Then, the tail-recursive call to `genealogical_search` repeats the process for the new population.

Please see [7] for a discussion of the iterative-deepening search in ADATE. Please note that Figure 1 was written to be easy to understand. The actual ADATE code is more efficient and does not waste storage by producing the entire list of children before selecting the ones worthy of population membership.

Which requirements should the population management algorithm i.e., `modify_pop`, satisfy?

1. Populations should be small in order to decrease run times. A maximum of a few of the thousands or millions of children produced from a given parent should typically be inserted into the population, maybe causing the removal of a few others.

2. Another requirement is that the algorithm should be efficient, preferably taking amortized time $O(\log n)$ to decide if a new child should be added to a population of $n$ individuals.

3. The really challenging requirement is that the algorithm should not cause missing links in a relevant genealogical chain, which happens when a program $P_i$ in a relevant chain $P_1 - P_2 - \ldots - P_m$ is prematurely discarded by the algorithm.

Before coding the first version of the ADATE system in 1992, we constructed about a dozen genealogical chains by hand and carefully studied their properties. It should have been many more, but manual construction and analysis of such chains is quite time consuming. Together with subsequent and extensive experimental activity, this study has substantiated the following three observations:

1. There are usually very many "continuous" chains leading from the empty initial program to a given desirable program. We say that a chain $P_1 - P_2 - \ldots - P_m$ is continuous if each $P_i$ is obtainable from $P_{i-1}$ using a reasonable transformation cost limit, which currently is a limit yielding a maximum of a few million programs.

2. Almost always, there are quite a few different desirable programs.

3. Surprisingly often, there is at least one desirable program $P_m$ and at least one continuous chain $P_1 - P_2 - \ldots - P_m$ such that each $P_i$ is better than all other programs that are produced by ADATE and that have the same or smaller syntactic complexity.

We will now present a basic population management technique that always satisfies requirement three provided that observation three holds and sometimes even if it does not hold.

Assume that the programs in the population, say $Q_1, Q_2, \ldots, Q_n$, are sorted in order of increasing syntactic complexities $S_1, S_2, \ldots, S_n$ and that the evaluation value of program number $i$ is $\Psi_i$. The basic technique is to maintain the population so that each $\Psi_i$ is better than $\Psi_{i-1}$. Assume that a new child $C$, which is to be considered for insertion into the population, has syntactic complexity $S'$ and evaluation value $\Psi'$. Here are the possible outcomes when considering $C$:

- There is already a program $Q_i$ in the population that is neither bigger nor worse i.e., $S_i \leq S'$, and $\Psi_i$ is not worse than $\Psi'$. The new child $C$ is then discarded.

- If $C$ is better than all programs in the population that are not bigger, $C$ is inserted into the population and all programs that are neither smaller nor better are removed.

The ADATE code uses a kind of binary tree, a so-called splay tree [11], to carry out these operations in amortized time $O(\log n)$.

This simple and basic definition of `modify_pop` will keep an entire chain, that fits observation three, in the population and then obviously satisfy requirement three i.e., not prematurely discard programs that are links in a relevant chain.

However, requirement three will also be satisfied if only the last seen link of the chain, say $P_k$, is not thrown away before it has had a chance to become parent with a cost limit that is sufficient to yield the next link $P_{k+1}$, which naturally becomes the new last seen link.

Note that a population typically contains several chains that may merge and diverge many times. For example, a population $Q_1, Q_2, \ldots, Q_7$ may contain the three chains $Q_1 - Q_4 - Q_6 - Q_7$, $Q_1 - Q_3 - Q_5 - Q_6 - Q_7$ and $Q_1 - Q_2$.

The basic technique favours small programs each step of the way, which has the following pros and cons:

- A small program is likely to be more general than a bigger program with the same evaluation value. This is basically Ockham's razor principle as suggested by Wilhelm von Ockham in the 14'th century.

- A smaller program has fewer transformation "sites" and is combinatorially cheaper from this point of view.

- A smaller program is quicker to compile, particularly with optimizations.

- There is some evidence from genetics that introns i.e., code which is permanently or temporarily out of commission, may be important for effective evolution in general. For this reason and others, it is sometimes beneficial to supplement size minimization with additional heuristics that allow bigger programs as in Section 3.

There are numerous statistics and machine learning papers dealing with Ockham's razor and the relationship between model size and generality [5]. For example, recent work in genetic programming by Byoung-Tak Zhang and Heinz Mühlenbein [12] includes the complexity into the evaluation (fitness) function, setting the fitness $F(P)$ for a program $P$ to $E(P) + \alpha(P)S(P)$, where $E(P)$ is a real-valued error measure, $\alpha(P)$ is a so-called Ockham factor and $S(P)$ is the syntactic complexity. The Ockham factor $\alpha(P)$ is adapted according to the distribution of errors and complexities in past populations and in the current population.

Their goal is to obtain a balance between fitness optimization and complexity minimization, whereas we primarily want to ensure the presence of relevant genealogical chains in the population. The ADATE size-evaluation ordering as described above is superior for the latter purpose since it never throws away an individual which is the best for its size but still quite inferior to some bigger individual(s). Another major problem with using Zhangs and Mühlenbeins fitness in ADATE would be that the fitness is real-valued. In ADATE, an evaluation function $\Psi$ returns multiple discrete values that are difficult to convert to a single and representative real value. ADATE only compares a value $\Psi_1$ with another value $\Psi_2$ and never performs arithmetic operations on them.

The ADATE system uses at least four evaluation functions and maintains one sub-population for each function. Each sub-population is maintained according to the basic technique above and organized as a splay tree. An attempt is made to insert a new child into each sub-population, which means that the same child

may occur in several sub-populations if it is favoured by more than one evaluation function. Therefore, the ADATE code corresponding to Figure 1 makes sure that an individual only is expanded once with a given cost limit.

Finally, we discuss requirement one that expresses a yearning for small populations. Assume that

$$(S_1, \Psi_1), (S_2, \Psi_2), \ldots, (S_n, \Psi_n)$$

is a sequence of pairs $(S_i, \Psi_i)$ such that $S_i$ is the syntactic complexity and $\Psi_i$ is the evaluation value of a program $Q_i$ in a sub-population $Q_1, \ldots, Q_n$. Recall that the sub-population is organized so that $S_1 < S_2 < \ldots < S_n$ and each $\Psi_i$ is better than $\Psi_{i-1}$. Obviously, the biggest program $Q_n$ is also the best.

Consider the complexity difference $S_i - S_{i-1}$ between two programs $Q_i$ and $Q_{i-1}$. If the grading scale used by the output evaluation function is more fine-grained and identifies smaller program improvements, $S_i - S_{i-1}$ is likely to be smaller. The sub-population cardinality $n$ is determined by:

1. The granularity of the output evaluation function.
2. The complexity $S_n$ of the best found program.

If $\delta$ is the average complexity difference, influenced by the granularity, we have $n \approx S_n/\delta$. Currently, it is typical for $\delta$ to be between one and three bits whereas $S_n$ typically is less than 400 bits, yielding maximum sub-population cardinalities of a few hundred individuals. Please see Section 3.5.1 in [7] for a definition of syntactic complexity.

The four or more evaluation functions employed by ADATE are so similar that there is much overlap between sub-populations, which means that the total number of unique individuals in the population is considerably less than the sum of the sub-population cardinalities.

# 3 Extending the Population to Avoid Missing Links

The main challenge for local search methods is to avoid being trapped by local optima. Even though the basic population management in Section 2 is amazingly good at tracking at least one relevant genealogical chain, it sometimes fails due to missing links and is trapped in one or more local optima. The obvious way to reduce the probability of a critical missing link is to increase the population cardinality to try to prevent that the link is discarded prematurely.

A given program almost always has very many programs that are semantically equivalent and therefore belong to the same equivalence class. Consider the programs that belong to the same class and that have been produced by ADATE. As long as these programs are no more than say twice as big as the smallest program in the class, it often does not matter which one that is used as a link in a chain. In such situations, it is wasteful to keep more than one program from the equivalence class in the population.

When extending the basic population described in Section 2, we therefore try to avoid adding equivalent or similar programs, which certainly is not easy. It is rather futile to try to directly compare two programs to detect similarity or equivalence since they may have completely different syntactic structures and still be semantically equivalent. We also rejected the possibility of using a deductive rewriting system to find equivalence since such systems are too slow and unable to handle a sufficiently large fraction of the generally recursive programs synthesized through ADATE.

The heuristics presented below has been implemented and tested since early 1996, but is still tentative.

We say that an individual in the basic population from Section 2 is a base individual. For each base individual, ADATE maintains three classes of additional individuals. Each of these three classes has a fixed, predetermined maximum cardinality and uses its own diversification technique that is supposed to avoid similarities between the programs in the class.

If a program does not qualify as a base, it becomes a candidate for insertion into the classes of an existing base individual in the population. Assuming that the candidate has evaluation value $\Psi'$, insertion is attempted into the classes of the biggest base individual with an evaluation value that is not better than $\Psi'$. Note that most of the individuals in the classes of a given base individual usually have the same evaluation value as the base individual.

We will now discuss each of the three classes piggybacked onto each base individual, assuming that each class has the same maximum cardinality $N$, which is chosen by the specifier.

1. **The output class**. Diversity is maintained by requiring that the base program and all programs admitted into the output class produce different outputs. Note that they still usually have the same evaluation value. Minimization of syntactic complexity is used as a secondary heuristic.

2. **The syntactic complexity management (SCM) class**. This class uses the weakest diver-

sification technique, preferring programs with syntactic complexities that are as close as possible to predetermined values, which actually is a kind of randomization. Given that the base has syntactic complexity $S_i$, the predetermined target values are $\alpha^1 S_i$, $\alpha^2 S_i$, ..., $\alpha^N S_i$, where $\alpha$ is a constant currently chosen to $1.35^{1/N}$.

3. **The longest common subsequence (LCS) class**. Informally, the length of the LCS [1] of two sequences says how well they match. For example, LCS is used in the UNIX command `diff` and when analyzing DNA sequences. A useful heuristic for maximization of the difference between two sequences is minimization of the length of their LCS.

   The principle employed in the LCS class is that two programs are likely to be different if their development histories are different. In each individual, ADATE stores the sequence of evaluation values of the ancestors. The heuristic in the LCS class is to use the LCS length of two such sequences as a measure of the difference between the two corresponding individuals, but an explanation of the details would be too long-winded for this paper.

   When the LCS heuristic produces equal values, insertion or rejection is determined using an SCM heuristic with $\alpha$ chosen to $2^{1/N}$. Thus, an LCS class has the LCS as the primary heuristic and the SCM as the secondary.

These three classes are pipelined, which means that an individual knocked out of one class moves on to the next and is discarded only if it has gone through all three classes.

# 4 Experimental Results

The experiments should determine the usefulness of the basic population management as well as the importance of the output class, the SCM class and the LCS class. Ideally, we should run the ADATE system on hundreds of reasonably complex problems, performing a number of runs with varying class cardinalities for each problem. However, the run times are quite long, which necessitated the following experimental restrictions:

- For each problem, we ran the system using the same maximum cardinality $N$ for each class and only tried $N = 0$, $N = 1$ and $N = 2$.
- No problem required ADATE to invent more than a few recursive help functions.

Table 1: Experimental results.

| *Problem* | *Best N* | *Run time in seconds* |
|---|---|---|
| Reversing a list | 0 | $5.7 \cdot 10^2$ |
| List delete min | 0 | $3.9 \cdot 10^3$ |
| Cartesian product | 0 | $9.8 \cdot 10^2$ |
| Intersecting two lists | 0 | $4.6 \cdot 10^4$ |
| String comparison | 0 | $5.0 \cdot 10^4$ |
| Sorting a list | 0 | $1.5 \cdot 10^3$ |
| Locating a substring | 0 | $5.0 \cdot 10^5$ |
| BST insertion | 0 | $1.2 \cdot 10^4$ |
| Binary multiplication | 0 | $4.5 \cdot 10^4$ |
| Simplifying a polynomial | 0 | $7.5 \cdot 10^4$ |
| Transposing a matrix | 0 | $4.2 \cdot 10^4$ |
| Permutation generation | 1 | $4.0 \cdot 10^5$ |
| BST deletion | 0 | $3.4 \cdot 10^5$ |
| Path finding | 2 | $8.2 \cdot 10^5$ |
| Binary addition | 0 | $7.4 \cdot 10^5$ |
| Rectangle intersection | 0 | $2.2 \cdot 10^5$ |

Table 1 lists the problems in order of increasing synthesized program complexity and shows the value of $N$ yielding the shortest run time for each problem. The run times are for a single 200 MHz PentiumPro CPU using Standard ML of New Jersey version 109.11 in conjunction with the ML compiler built into the ADATE system. Complete specifications and log files are available from [9].

Currently, we believe that no other automatic programming method can solve even one single of these problems without resorting to specifications that contain extra information, for example predefined, problem-specific and special-designed auxiliary functions.

The results in Table 1 indicate that the basic population management is amazingly effective, but the problems are not sufficiently complex to say much about the importance of classes. Note that ADATE may eventually find a desirable program with $N = 0$ even when Table 1 says that $N = 1$ or $N = 2$ is quicker.

# 5 Summary and Conclusions

One major goal of combinatorial optimization strategies is to escape from local optima that are too bad in comparison with a global optimum. In ADATE, the most important ways not to get stuck on such local optima are as follows.

1. ADATE tries hard to maintain diversity in the population using several different evaluation functions and measures of the difference between individuals.

2. When being temporarily stuck, ADATE iteratively re-expands individuals in order to find new and better children.

The basic strategy is to try to track relevant chains of bigger-and-bigger, better-and-better programs. The resulting basic population may optionally be supplemented with additional individuals using three different diversification strategies. However, the basic strategy is so successful that it is difficult to gauge the effectiveness of these three strategies without quite complex problems and corresponding very long run times.

The program transformations from [8] and the population management in this paper form the core of ADATE, which has several essential and unique capabilities, for example automatic synthesis of "Turing complete" recursive programs with automatic invention of recursive help functions as they are needed.

**Acknowledgement**

I would like to thank the anonymous referees for stimulating questions and suggestions.

# References

[1] A. V. Aho, J. E. Hopcroft and J. D. Ullman, *The design and analysis of computer algorithms*, Addison-Wesley, 1983.

[2] T. Blickle and L. Thiele, A comparison of selection schemes used in evolutionary algorithms, *Evolutionary Computation*, volume 4, number 4, winter 1996, pp. 361–394.

[3] R. E. Korf, Depth-first iterative-deepening: an optimal admissible tree search, *Artificial Intelligence*, volume 27, 1985, pp. 97–109.

[4] J. R. Koza, *Genetic programming: on the programming of computers by means of natural selection*, MIT Press, 1992.

[5] M. Li and P. M. B. Vitányi, *An Introduction to Kolmogorov Complexity and Its Applications*, Springer-Verlag, 1993.

[6] J. R. Olsson, Execution of logic programs by iterative-deepening A* SLD-tree search, *BIT*, volume 33, 1993, pp. 214–231.

[7] J. R. Olsson, Inductive functional programming using incremental program transformation and Execution of logic programs by iterative-deepening A* SLD-tree search, Research report 189, Dr scient thesis, ISBN 82-7368-099-1, University of Oslo, October 1994.

[8] J. R. Olsson, Inductive functional programming using incremental program transformation, *Artificial Intelligence*, volume 74, number 1, March 1995, pp. 55–83.

[9] J. R. Olsson, Web page for Automatic Design of Algorithms through Evolution, http://www-ia.hiof.no/~rolando/adate_intro.html (current Jun. 26, 1997).

[10] C. R. Reeves (editor), *Modern heuristic techniques for combinatorial problems*, ISBN 0-470-22079-1, Blackwell Scientific Publications, 1993.

[11] R. E. Tarjan, *Data structures and network algorithms*, ISBN 0-89871-187-8, SIAM, 1983.

[12] B. Zhang and H. Mühlenbein, Balancing accuracy and parsimony in genetic programming, *Evolutionary Computation*, volume 3, number 1, 1995, pp. 17–38.