Evolving Efficient List Search Algorithms (Draft[⋆])

Kfir Wolfson[⋆⋆] and Moshe Sipper

Dept. of Computer Science, Ben-Gurion University, Beer-Sheva, Israel

Abstract. We peruse the idea of algorithmic design through Darwinian evolution, focusing on the problem of evolving *list search algorithms*. Specifically, we employ genetic programming (GP) to evolve iterative algorithms for searching for a given key in an array of integers. Our judicious design of an evolutionary language renders the evolution of linear-time search algorithms easy. We then turn to the far more difficult problem of logarithmic-time search, and show that our evolutionary system successfully handles this case. Subsequently, because our setup might be perceived as being geared towards the emergence of binary search, we generalize our genomic representation, allowing evolution to assemble its own useful functions via the mechanism of automatically defined functions (ADFs). We show that our approach routinely and repeatedly evolves general and correct efficient algorithms.

1 Introduction

One of the most basic tasks a computer scientist faces is that of designing an algorithm to solve a given problem. In his book *Algorithmics*, Harel [1] defines the subject matter as "the area of human study, knowledge, and expertise that concerns algorithms." Indeed, the subtitle of Harel's book—"The Spirit of Computing"—evidences the importance of algorithm design in computer science.

While simple problems readily yield to algorithmic solutions, many, if not most, problems of interest are hard, and finding an algorithm to solve them is an arduous task. Compounding this task is our desire not only to find a *correct* algorithm but also an *efficient* one, with efficiency being measured in terms of resources to be used with discretion, such as time, memory, and network traffic.

Evolutionary algorithms have been applied in recent years to numerous problems from diverse domains. However, their application within the field of software engineering in general, and algorithmic design in particular, is still quite limited. This dearth of research might be partly attributed to the complexity of algorithms—and the even greater complexity of their design.

Our aim in this paper is to introduce the notion of algorithmic design through Darwinian evolution. To find out whether this approach has any merit at all,

[⋆] Final version available at http://www.springerlink.com/content/k19620u263n45lmw/

[⋆⋆] Kfir Wolfson was partially supported by the Frankel Center for Computer Science at Ben-Gurion University.

we begin with a benchmark case, one familiar to any freshman computer-science student: searching for a given key in an array of elements. A solution to this problem is known as a *list search algorithm* (implying a one-dimensional array, or a linked-list of elements), and can be either iterative or recursive in nature. Herein, we evolve iterative algorithms for arrays of integers (rather than lists), and refer to them as *array search algorithms*, or simply *search algorithms*. We ask two questions:

1. Can evolution be applied to finding a search algorithm?
2. Can evolution be applied to finding an *efficient* search algorithm?

Employing genetic programming (GP) to find algorithmic innovations, our findings show that the answer to both questions is affirmative. Indeed, our judicious design of an evolutionary language renders the answer to the first question quite straightforward: A search algorithm that operates in linear time is easily evolved. We then turn to finding a more efficient algorithm for *sorted* arrays, concentrating on execution time as a measure of efficiency. We show that a logarithmic-time search algorithm can be evolved, and proceed to analyze its workings.

This paper is organized as follows: In the next section we describe our setup for evolving search algorithms, followed by results in Section 3. A more general representation involving automatically defined functions (ADFs) is presented in Section 4. Related work on program evolution is described in Section 5, with concluding remarks following in Section 6.

2 The Evolutionary Setup

We use Koza-style GP [2], in which a population of individuals evolves. An individual is represented by an ensemble of LISP expressions, each composed of functions and terminals. Each individual represents a computer program—or algorithm—for searching an element in an array. Since most common computer languages are typed, we opted for strongly-typed genetic programming [3], which may ultimately help in evolving more understandable algorithms. We used the ECJ package to conduct the experiments [4].

2.1 Representation

We designed a representation that has proven successful in the evolution both of linear and sublinear search algorithms. The genotypic function set is detailed in Table 1. In order to evaluate an individual, a phenotype is constructed by plugging the genotypic code into the template given in Fig. 1. The genotype thus represents the body of the `for` loop, the hardest part to develop in the algorithm, while the incorporating phenotype adds the necessary programmatic paraphernalia.

As can be seen in Fig. 1, the individual's genotypic code is executed `iterations` times for an input array of size n, with a global variable `ITER`

Table 1. Terminal and function sets for the evolution of search algorithms (both linear and sublinear). int refers to Integer, bool – Boolean.

Name	Arguments	Return Type	Description
TERMINALS			
`INDEX`	none	int	Current pointer into array
`Array[INDEX]`	none	int	Element at location `INDEX` in the input array. If `INDEX` is not in $[0, n-1]$, for array length n, 0 is returned
`KEY`	none	int	The element we are searching for
`ITER`	none	int	Current iteration number
`M0, M1`	none	int	Getters to global variables, at the algorithm's disposal
`[M0+M1]/2`	none	int	Average of `M0, M1` (truncated to nearest integer)
`NOP`	none	void	Does nothing
`TRUE, FALSE`	none	bool	Boolean terminals
FUNCTIONS			
`INDEX:=`	int	void	Sets the value of variable `INDEX` to value returned by argument
`M0:=, M1:=`	int	void	Setters to global variables
$>$, $<$, $=$	int, int	bool	Returns true if the first argument is greater than, less than, or equal to the second argument, respectively; else returns false
`PROGN2`	void, void	void	Sequence: execute first argument, then execute second argument
`If`	bool, void, void	void	Conditional branching: if the first argument evaluates to true, execute second argument, otherwise execute third argument

incremented after each iteration. For the linear case we set `iterations` to n, whereas for the sublinear case `iterations` is set to $\lceil \log_2 n \rceil$. This upper limit on the number of loop iterations is the only difference between the evolution of the two cases and can be considered as part of the fitness function (described below), specifically, the differentiating part.

We decided not to add an early-termination condition, which exits the loop when the index of the searched-for key is found, in order to render the problem harder for evolution: The evolving search algorithm should learn to retain the correct index, if the key is located before the loop terminates.

The terminal and function sets include read access to the variable `ITER` and the searched-for `KEY`, and read/write access to a global variable `INDEX`, initialized to 0. `INDEX` is used to access array elements through the `Array[INDEX]` terminal, and the value of `INDEX` after the final iteration is taken as the return value of the run. To discourage `INDEX` being set to values outside the array bounds ($[0, n-1]$ since we use Java), `Array[INDEX]` returns 0 if `INDEX` is outside of bounds. Note that the key 0 does not appear in any input array because all the keys are positive, as described below.

```
public static int search(int[] arr, int KEY) {
    int n     = arr.length;
    int M0    = 0;
    int M1    = n-1;
    int INDEX = 0;
    for (int ITER = 0; ITER < iterations; ITER++) {
        -> GENOTYPE INSERTED HERE <-
    }
    return INDEX;
}
```

Fig. 1. Evolution of search: The evolving genotype, composed of elements delineated in Table 1, is incorporated into the above phenotypic JAVA template. The variable `iterations` is set to n for evolving linear search algorithms, and is set to $\lceil \log_2 n \rceil$ for evolving sublinear algorithms.

The evolving search algorithm is provided with read/write access to two global variables, `M0` and `M1`, which the algorithm may use as it (or, more precisely, evolution) sees fit. The variables are initialized to 0 and $n-1$, respectively, which affords the individual potential knowledge of the array length. This information should prove useful in sublinear solutions. The `[M0+M1]/2` terminal embodies human intuition about the problem, to facilitate the solution, which, nonetheless, still requires crucial algorithmic insight—to be derived via evolution. In Section 4 we re-examine this terminal, repealing it altogether.

The remaining functions and terminals include standard comparative predicates ($<, >, =$), conditional branching (`If`), a sequence operator (`PROGN2`), the Boolean terminals `TRUE` and `FALSE`, and a simple `NOP` (no-operation) to enable, e.g., the evolution of an *if* without an *else* part.

Note that the evolving algorithms can inherently deal with keys not in the array, by wrapping the `search` method in a method that returns an illegal index value (e.g., -1) if the array does not contain the key in the returned index. Thus, the algorithms will not be trained or tested on such inputs. We also mention that using our function and terminal sets (specifically, `ITER` being read-only, and not defining a nested-loop function) and limiting the number of iterations of the `for` loop, we avoid generating non-terminating phenotypes.

2.2 Fitness Evaluation and Run Parameters

Fitness is defined similarly both for the evolution of linear and sublinear algorithms. The basic idea is to present the evolving individual with many random input arrays, have it run and search keys in them, and reward the individual for the *closeness* of the outputs to perfect answers. It is important to note that fitness is based not on an all-or-nothing quality (key found or not), but on gradations of "finding" quality—as defined below.

Specifically, to compute fitness, each individual is run over a set of training cases, each case being an array to be searched. The set of training cases is fixed for all individuals per generation, and is randomly generated anew every generation, as we found this encouraged more general solutions. Let *minN* and *maxN* be the predefined minimal and maximal training-case array lengths, and let $N = maxN - minN + 1$. We generate N arrays of all N possible sizes in the range $[minN, maxN]$, both to induce variety during evolution and also to render the solution general, able to function correctly on as many different array lengths as possible.

In the linear case, an array of length $n \in [minN, maxN]$ holds a *random* permutation of integers in the range $[1000, 1000 + n - 1]$. In the sublinear case, an array of length $n \in [minN, maxN]$ holds a *sorted* list of random integers in the range $[n, 100n]$. Note that the key range is completely disjoint from the index range, to discourage “cheating”(e.g., in a sorted array, a program might evolve to simply return the key value, which happens to equal the index value).

All n keys are searched for by an (individual) phenotypic program in the population, using the `search(arr,KEY)` method given in Fig. 1.

In order to define fitness, we first provide a number of definitions. The error per single key search is defined as the absolute distance between the correct index of `KEY` in the array and the index returned by `search(arr,KEY)`:

$$error(arr, key, correct) = \left|correct - \texttt{search}(arr, key)\right| .$$

An error of zero means that the search was successful. All generated arrays contain unique elements, to avoid ambiguity in error definition. Note that the index returned by the `search` function may be out of array bounds, and as such suffers from a larger error value—another discouragement of illegal index values. Let *calls* be the total number of `search` calls, over all N training cases, i.e., the total number of keys searched for:

$$calls = \sum_{n=minN}^{maxN} n = \frac{maxN(maxN + 1)}{2} - \frac{minN(minN - 1)}{2} .$$

The average error per `search` call is calculated as follows:

$$avgerr = \frac{1}{calls} \sum_{t=1}^{N} \sum_{i=0}^{n_t - 1} error(arr_t, arr_t[i], i),$$

where arr_t is the tth array of the N randomly generated arrays, and n_t is its length ($n_t = minN + t - 1$). (Note: Java array indexes begin at 0.)

Note that the order of calls to `search(arr,KEY)` does not affect their outputs, so it is safe to execute the individual program for consecutive indexes in the array without bias.

We define a *hit* as the finding of the precise location of `KEY`, i.e., $error(arr, key, correct) = 0$. The total number of hits is thus given by:

$$hits = \sum_{t=1}^{N} \sum_{i=0}^{n_t - 1} \max\left(0, 1 - error(arr_t, arr_t[i], i)\right) .$$

Finally, the fitness value of an individual is defined as the average error per `search` call, with a 0.5% bonus reduction for every 1% of correct hits:

$$fitness = avgerr \times \left(1 - 0.5 \times \frac{hits}{calls}\right) .$$

For example, if an individual scored 300 hits in 1000 `search` calls, its fitness will be the average error per call, reduced by 15%. An evolving program attains a perfect raw fitness value of zero if every test is passed, i.e., for every searched `KEY` the correct index is returned.

The bonus *hits* component was added to encourage perfect answers since we felt that an individual with a higher overall error could be considered better than one with a lower overall error, if the former's hit count is higher. We also noted that the *hits* component increased fitness variation in the population.

The best solution of each run was subjected to a stringent generality test, by running it on random arrays of all lengths in the range $[2, 5000]$ (for linear search the range was smaller, $[2, 500]$, given the considerably longer runtime of such a search—and of the generality test thereof).

Kinnear [5] noted that "For any algorithm... that operates on an infinite domain of data, no amount of testing can ever establish generality. Testing can only increase confidence." To increase our confidence in the solutions evolved we added analysis by hand to the generality test. Though some solutions were quite large, the intuition behind the *algorithmic idea* could be gleaned by focusing on the ADF code (Section 4).

Array-length parameters were set to $minN = 2$ and $maxN = 10$ for the linear case, to decrease evaluation time, and $minN = 2$ and $maxN = 100$ for the sublinear case, as a trade-off between generality and performance. (When we used lower boundary values, evolved solutions did not prove general. Higher boundary values yielded general solutions, at the expense of increasing evaluation time by a quadratic factor.)

The GP run operators and parameters are summarized in Table 2.

Table 2. GP parameters.

Objective	Find a key in a given input array of unsorted (linear-time case) or sorted (sublinear-time case) positive integers in a prefixed number of iterations
Function and Terminal sets	As detailed in Table 1
Fitness	Average error per `search` call on training set, with bonus reduction for hits (as detailed in Section 2.2)
Selection	Tournament of size 7, elitism of size 2, generational
Population Size	250
Initial Population	Created using ramped-half-and-half, with a maximum depth of 6
Max tree depth	10
Generations	5000 (or until individual with perfect fitness emerges)
Crossover	Standard subtree exchange
Mutation	Standard grow (generate new subtree at chosen node)
Node Selection	Nodes chosen for crossover or mutation are function nodes with probability 0.9 and terminal nodes with probability 0.1
Genetic Operator Probabilities	On the selected parent individual: with probability 0.1 copy to next generation (reproduction); with probability 0.05 mutate individual; with probability 0.85, select a second parent and cross over trees

3 Results

3.1 Linear

It turned out that evolving a linear-time search algorithm was quite easy with the function and terminal sets we designed. We performed 50 runs, 46 of which (92%) produced solutions with a perfect fitness of 0, also passing with flying colors the generality test, exhibiting no errors up to length 500. In fact, our representation rendered the problem easy enough for a perfect individual to appear in the randomly generated generation 0 in three of the runs.

An example of an evolved solution is shown in Fig. 2, along with the equivalent Java code. When plugged into the template of Fig. 1, we observe a linear-time search algorithm that proceeds as follows: As long as KEY is not in location INDEX, INDEX is incremented by one along with ITER. From the index wherein the key is found (i.e., Array[INDEX] = KEY), INDEX is no longer modified, preserving the correct value until the end of the algorithm's execution. An irrelevant setting of M1 to [M0+M1]/2 takes place, but does not have any effect on the returned index.

```
(If (= Array[INDEX] KEY)
    (M1:= [M0+M1]/2)
    (INDEX:= ITER))
```

(a)

```
if (arr[INDEX] == KEY)
    M1 = (M0+M1)/2;
else
    INDEX = ITER;
```

(b)

Fig. 2. An evolved linear-time search algorithm. (a) LISP genotype. (b) Equivalent JAVA code, which, when plugged into the full-program template (Fig. 1), forms the complete algorithm. (Note: the actual code contains an additional check when executing the arr[INDEX] instruction; if the value of INDEX is within $[0, \texttt{arr.length} - 1]$ return arr[INDEX], otherwise, return 0.)

3.2 Sublinear

The sublinear search problem proved (unsurprisingly) a greater challenge for evolution. We performed 50 runs, 35 of which (70%) produced perfect solutions, exhibiting no errors up to length 5000. The solutions emerged in generations 22 to 3632, and their sizes varied between 42 and 244 nodes. Seven runs (14%) produced near-perfect solutions, which failed on a single key in the input arrays, usually either the first or last key (scoring 99.96% hits on the generality test).

A simplified version of one of the evolved solutions is given in Fig. 3, along with the equivalent Java code. The solution was simplified by hand from a tree of 50 nodes down to 14, and it turns out to be an implementation of the well-known *binary search.*

```
(PROGN2
  (INDEX:= [M0+M1]/2)
  (if (> KEY  Array[INDEX])
      (PROGN2
        (M0:= [M0+M1]/2)
        (INDEX:= M1))
      (M1:= [M0+M1]/2))))
```

(a)

```
INDEX = (M0+M1)/2;
if (KEY > arr[INDEX]) {
    M0 = (M0+M1)/2;
    INDEX = M1;
}
else
    M1 = (M0+M1)/2;
```

(b)

Fig. 3. An evolved sublinear-time search algorithm (simplified). Evolved solution reveals itself as a form of *binary search.* (a) LISP genotype. (b) Equivalent JAVA code, which, when plugged into the full-program template (Fig. 1), forms the complete algorithm.

4 Less Knowledge—More Automation

Re-examining the representation used until now (Table 1), we note that most terminals and functions are either general-purpose ones (e.g., conditional and predicates), or ones that represent a very basic intuition about the problem to be solved (e.g., the straightforward need to access `INDEX` and `KEY`). However, one terminal—`[M0+M1]/2`—stands out, and might be regarded as our "intervening" too much with the course of evolution by providing insight born of our familiarity with the solution. In this section we remove this terminal and augment the evolutionary setup with the mechanism of automatically defined functions (ADFs) [6]. (Note on terminology: We use the term *main tree* rather than result-producing branch (RPB) [6], since the tree does not actually produce a result: The behavior of the program is mainly determined by the side effects of functions in the tree, e.g., `INDEX:=` changes the value of `INDEX`.)

Specifically, the terminal `[M0+M1]/2` was removed from the terminal set, with the rest of the representation remaining unchanged from Table 1. We added an ADF—`ADF0`—affording evolution the means to define a simple mathematical function, able to use the variables `M0` and `M1`. The evolved function receives no arguments, and has at its disposal arithmetic operations, integer constants, and the values of the global variables, as detailed in Table 3.

The evolutionary setup was modified to incorporate the addition of ADFs. The main program tree and the ADF tree could not be mixed because the function sets are different, so crossover was performed per tree type (main or ADF). We noticed that mutation performed better than crossover, especially in the ADF tree. We increased the array-length parameters to $minN = 200$ and $maxN = 300$, upon observing a tendency for non-general solutions to emerge with arrays shorter than 200 in the training set. The rest of the GP parameters are summarized in Table 4.

The sublinear search problem with an ADF naturally proved more difficult than with the `[M0+M1]/2` terminal. We performed 50 runs with ADFs, 12 of which (24%) produced perfect solutions. The solutions emerged in generations 54 to 4557, and their sizes varied between 53 and 244 nodes, counting the sum total of nodes in both trees.

Table 3. Terminal and function sets for the automatically defined function `ADF0`.

Name	Arguments	Return Type	Description
TERMINALS			
`M0, M1`	none	int	Getters to global variables
`0, 1, 2`	none	int	Integer constants
FUNCTIONS			
$+$, $-$, $\times$	int, int	int	Standard arithmetic functions, returning the addition, subtraction, and multiplication of two integers
/	int, int	int	Protected integer division. Returns the first argument divided by the second, truncated to integer. If the second argument is 0, returns 1

Table 4. GP parameters for ADF runs. (Parameters not shown are identical to those of Table 2).

Function and Terminal sets	As detailed above in this section
Initial Population	Created using ramped-half-and-half, with a maximum depth of 6 for main tree and 2 for ADF
Max tree depth	main tree: 10; ADF tree: 4
Crossover	Standard subtree exchange from same tree (main or ADF) in both parents
Genetic Operator Probabilities	On the selected parent individual: with probability 0.1 copy to next generation (reproduction); with probability 0.25 mutate individual's main tree; with probability 0.4 mutate individual's ADF tree; with probability 0.2, select a second parent and cross over main trees; with probability 0.05, select a second parent and cross over ADF trees

Analysis revealed all perfect solutions to be variations of *binary search.* The algorithmic idea can be deduced by inspecting the ADFs, all eleven of which turned out to be equivalent to one of the following: $(\texttt{M0}+\texttt{M1})/2$, $(\texttt{M0}+\texttt{M1}+1)/2$, or $(\texttt{M0}/2+(\texttt{M1}+1)/2)$ (all fractions truncated); to wit, they are reminiscent of the original `[M0+M1]/2` terminal we dropped. We then simplified the main tree of some individuals and analyzed them. A simplified version of one of the evolved solutions is given in Fig. 4, along with the equivalent Java code. The solution was simplified by hand from 58 nodes down to 26.

5 Related Work

We performed an extensive literature search, finding no previous work on evolving list search algorithms, for either arrays or lists of elements. The "closest" works found were ones dealing with the evolution of sorting algorithms, a problem that can be perceived as being loosely related to array search. Note that

```
(PROGN2
   (PROGN2
      (if (< Array[INDEX] KEY)
          (INDEX:= ADF0)
          NOP)
      (if (< Array[INDEX] KEY)
          (M0:= INDEX)
          (M1:= INDEX)))
   (INDEX:= ADF0)))

ADF0:
(/ (+ (+ 1 M0) M1) 2)
```

(a)

```
if (arr[INDEX] < KEY)
     INDEX = ((1+M0)+M1)/2;
if (arr[INDEX] < KEY)
     M0 = INDEX;
else
     M1 = INDEX;
INDEX = ((1+M0)+M1)/2;
```

(b)

Fig. 4. An evolved sublinear-time search algorithm with ADF (simplified). Evolved solution is another variation of *binary search*. (a) LISP genotype. (b) Equivalent JAVA code, which, when plugged into the full-program template (Fig. 1), forms the complete algorithm.

both problems share the property that a solution has to be 100% correct to be useful.

Like search algorithms, the problem of rearranging elements in ascending order has been a subject of intensive study [7]. Most works to date were able to evolve $O(n^2)$ sorting algorithms, and only one was able to reach into the more efficient $O(n \log n)$ class, albeit with a highly specific setup.

The problem of evolving a sorting algorithm was first tackled by Kinnear [5, 8], who was able to evolve solutions equivalent to the $O(n^2)$ bubble-sort algorithm. Kinnear compared between different function sets, and showed that the difficulty in evolving a solution increases as the functions become less problem-specific. He also noted that adding a parsimony factor to the fitness function not only decreased solution size, but also increased the likelihood of evolving a general algorithm.

The most recent work on evolving sorting algorithms is that of Withall et al. [9]. They developed a new GP representation, comprising fixed-length blocks of genes, representing single program statements. A number of list algorithms, including sorting, were evolved using problem-specific functions for each algorithm. A `for` loop function was defined, along with a `double` function, which incorporated a highly specific double-for nested loop. With these specialized structures Withall et al. evolved an $O(n^2)$ bubble-sort algorithm.

An $O(n \log n)$ solution was evolved by Agapitos et al. [10, 11]. The evolutionary setup was based on their object-oriented genetic programming system. In [10] the authors defined two configurations, one with a hand-tailored `filter` method, the second with a static ADF. The former was used to evolve an $O(n \log n)$ solution, and the latter produced an $O(n^2)$ algorithm. Runtime was evaluated empirically as the number of method invocations. In [11] an *Evolvable Class* was defined, which included between one and four *Evolvable Methods* that could call each other. This setup increased the search space and produced $O(n^2)$ modular recursive solutions to the sorting problem. Agapitos et al. noted that mutation performed better than crossover in their problem domain, a conclusion we also reached regarding our own domain of evolving search algorithms with ADFs (Section 4). Other interesting works on evolving sorting algorithms include [12–15], not detailed herein due to space limitations.

Another related line of research is that of evolving iterative programs. Koza [16] defined automatically defined iterations (ADIs) and Kirshenbaum [17] defined an iteration schema for GP. These constructs iterate over an array or a list of elements, executing their body for each element, an thus cannot be used for sublinear search, as their inherent runtime is $\Omega(n)$.

Many loop constructs were suggested, e.g., Koza's automatically defined loops (ADLs) [16], and the loops used to evolve sorting algorithms mentioned above. But, as opposed to the research on sorting algorithms, herein we assume that an external `for` loop exists, for the purpose of running our evolving solutions. In the sorting problem, the $O(n^2)$ solutions requires nested loops, which the language must support. The $O(n \log n)$ solution was developed in a language supporting recursion. In linear and sublinear search algorithms, there will *always* be a single loop (in non-recursive solutions), and the heart of the algorithm is the *body* of the loop (which we have evolved in this paper).

In summary, our literature survey has revealed several related interesting works on the evolution of sorting algorithms and on various forms of evolving array iteration. There seems to be no work on the evolution of array search algorithms.

6 Concluding Remarks and Future Work

We showed that algorithmic design of efficient list search algorithms is possible. With a high-level fitness function, encouraging correct answers to the search calls within a given number of iterations, the evolutionary process evolved correct linear and sublinear search algorithms.

Knuth [7] observed that "Although the basic idea of binary search is comparatively straightforward, the details can be somewhat tricky, and many good programmers have done it wrong the first few times they tried." Evolution produced many variations of correct binary search, and some nearly-correct solutions erring on a mere handful of extreme cases (which one might expect, according to Knuth). Our results suggest that, in general, algorithms can be evolved where needed, to solve hard problems.

Our work opens up a number of possible avenues for future research. We would like to explore the coevolution of individual main trees and ADFs, as in the work of Ahluwalia [18]. Our phenotypes are not Turing complete (TC) [19], e.g., because they always halt. It would be interesting to use a Turing-complete GP system to evolve search algorithms. Some of the evolved solutions are bloated. It would be interesting to see how adding parsimony pressure affects evolution.

We also plan to delve into related areas, such as sorting algorithms, and show evolutionary innovation in action. Ultimately, we wish to find an algorithmic innovation not yet invented by humans.

Acknowledgment

We are grateful for the many helpful remarks of the anonymous referees. We also thank Amit Benbassat for pointing us in an interesting direction.

References

1. Harel, D.: Algorithmics: The Spirit of Computing. Second edn. Addison-Wesley Publishing Company, Readings, MA (1992)
2. Koza, J.R.: Genetic Programming: On the Programming of Computers by Means of Natural Selection. MIT Press, Cambridge, MA, USA (1992)
3. Montana, D.J.: Strongly typed genetic programming. Evolutionary Computation **3**(2) (1995) 199–230
4. Luke, S., Panait, L.: A Java-based evolutionary computation research system. Online (March 2004) `http://cs.gmu.edu/~eclab/projects/ecj`.
5. Kinnear, Jr., K.E.: Evolving a sort: Lessons in genetic programming. In: Proceedings of the 1993 International Conference on Neural Networks. Volume 2., San Francisco, USA, IEEE Press (28 March-1 April 1993) 881–888
6. Koza, J.R.: Genetic Programming II: Automatic Discovery of Reusable Programms. MIT Press, Cambridge, MA (1994)
7. Knuth, D.E.: Sorting and Searching. Volume 3 of The Art of Computer Programming. Addison-Wesley, Reading, Massachusetts (1975)
8. Kinnear, Jr., K.E.: Generality and difficulty in genetic programming: Evolving a sort. In: Proceedings of the 5th International Conference on Genetic Algorithms, San Francisco, CA, USA, Morgan Kaufmann Publishers Inc. (1993) 287–294
9. Withall, M.S., Hinde, C.J., Stone, R.G.: An improved representation for evolving programs. Genetic Programming and Evolvable Machines **10**(1) (2009) 37–70
10. Agapitos, A., Lucas, S.M.: Evolving efficient recursive sorting algorithms. In: Proceedings of the 2006 IEEE Congress on Evolutionary Computation, Vancouver, IEEE Press (6-21 July 2006) 9227–9234
11. Agapitos, A., Lucas, S.M.: Evolving modular recursive sorting algorithms. In: EuroGP. (2007) 301–310
12. O'Reilly, U.M., Oppacher, F.: A comparative analysis of GP. In Angeline, P.J., Kinnear, Jr., K.E., eds.: Advances in Genetic Programming 2. MIT Press, Cambridge, MA, USA (1996) 23–44
13. Abbott, R., Guo, J., Parviz, B.: Guided genetic programming. In: The 2003 International Conference on Machine Learning; Models, Technologies and Applications (MLMTA'03), las Vegas, CSREA Press (23-26 June 2003)
14. Spector, L., Klein, J., Keijzer, M.: The push3 execution stack and the evolution of control. In: GECCO '05: Proceedings of the 2005 Conference on Genetic and Evolutionary Computation, New York, NY, USA, ACM (2005) 1689–1696
15. Shirakawa, S., Nagao, T.: Evolution of sorting algorithm using graph structured program evolution. In: SMC, IEEE (2007) 1256–1261
16. Koza, J.R., Andre, D., Bennett III, F.H., Keane, M.: Genetic Programming 3: Darwinian Invention and Problem Solving. Morgan Kaufman (April 1999)
17. Kirshenbaum, E.: Iteration over vectors in genetic programming. Technical Report HPL-2001-327, HP Laboratories (December 17 2001)
18. Ahluwalia, M., Bull, L.: Coevolving functions in genetic programming. Journal of Systems Architecture **47**(7) (July 2001) 573–585
19. Woodward, J.: Evolving Turing complete representations. In Sarker, R., et al., eds.: Proceedings of the 2003 Congress on Evolutionary Computation CEC2003, Canberra, IEEE Press (8-12 December 2003) 830–837