**Noname manuscript No.**
(will be inserted by the editor)

# Have Your Spaghetti and Eat it Too: Evolutionary Algorithmics and Post-Evolutionary Analysis (Draft*)

**Kfir Wolfson · Shay Zakov · Moshe Sipper · Michal Ziv-Ukelson**

Received: date / Accepted: date

**Abstract** This paper focuses on two issues, first perusing the idea of algorithmic design through genetic programming (GP), and, second, introducing a novel approach for analyzing and understanding the evolved solution trees. Considering the problem of *list search*, we evolve iterative algorithms for searching for a given key in an array of integers, showing that both correct linear-time and far more efficient logarithmic-time algorithms can be repeatedly designed by Darwinian means. Next, we turn to the (evolved) dish of spaghetti (code) served by GP. Faced with the all-too-familiar conundrum of understanding convoluted—and usually bloated—GP-evolved trees, we present a novel analysis approach, based on ideas borrowed from the field of bioinformatics. Our system, dubbed *G-PEA* (GP Post-Evolutionary Analysis), consists of two parts: 1) Defining a functionality-based similarity score between expressions, G-PEA uses this score to find subtrees that carry out similar semantic *tasks*; 2) Clustering similar sub-expressions from a number of independently-evolved fit solutions, thus identifying important *semantic building blocks* ensconced within the hard-to-read GP trees. These blocks help identify the important parts of the evolved solutions and are a crucial step in understanding how they work. Other related GP aspects, such as code simplification, bloat control, and building-block preserving crossover, may be extended by applying the concepts we present.

**Keywords** Genetic programming · Search algorithms · Post-evolutionary analysis · Edit distance · Reasoning · Building blocks

## 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 [25] defines the subject

* Final version available at: http://www.springerlink.com/content/k19620u263n45lmw/

All authors are with the
Department of Computer Science, Ben-Gurion University, Beer-Sheva 84105, Israel
E-mail: {wolfsonk, zakovs, sipper, michaluz}@cs.bgu.ac.il

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. Though their application within the field of software engineering has been gaining increased attention [26], the issue of algorithmic design via evolution has been explored only to a limited extent. This dearth of research might be partly attributed to the complexity of algorithms—and the even greater complexity of their design and analysis.

Our aim in this paper is twofold:

1. To demonstrate algorithmic design through Darwinian evolution.
2. To introduce a framework for analyzing the evolved algorithms.

To find out whether our approach has merit we focus herein on a benchmark algorithmic 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. In the first part of the paper we show that our first goal can be attained, and we name this GP approach *evolutionary algorithmics*. We evolve iterative algorithms for arrays of integers (rather than lists), and refer to them as *array search algorithms*, or simply *search algorithms*. We pose two questions: 1) Can evolution be applied to finding a search algorithm? and 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 indeed be evolved for this far more difficult case.

Having attained our first stated goal—algorithm design through evolution—we now turn to our second goal, namely, the analysis of the resulting code. Genetic programming is famous (some might say infamous) for producing spaghetti code (due to phenomena such as bloat and introns [5, 57]), which is arduous to analyze and comprehend. This is not so surprising given that the biologically inspiring metaphor for genetic programming—evolution by natural selection—has also produced a tangled genomic web, which biologists and bioinformaticians are now hard at work to disentangle. Indeed, the field of bioinformatics is where we turn to in order to design a methodology for analyzing—and thereby comprehending—our genetically programmed programs.

In the second part of the paper we thus turn to post-evolutionary analysis. A similarity measure between expressions is defined, based on which we describe an algorithm for discovering a core set of *semantic building blocks*, i.e., a set of important tasks which are implemented by sub-expressions in the evolved solutions.

Our proposed similarity measure between expressions is implemented here via a new variant of the tree-edit distance metric [7]. The distance metric is carefully designed to detect semantic similarities between code segments. In contrast to previous approaches for comparing code segments [11, 32, 45, 63, 68], we look for *similarity* rather than *equivalence*. The paper provides several examples of *similar* code segments detected by our algorithm—which are *not* equivalent—where similarity provides important information for code reasoning.

The approach takes advantage of the multitude of correct solutions offered by evolution, which differ in the genotype but perform the same phenotypic task. A standard clustering method is applied to detect the occurrence of phenotypically similar expressions, which are used to infer the semantic building blocks and allow code reasoning.

We demonstrate the viability and efficacy of our approach on the array search problem, showing its applicability to the detection of semantic building blocks in different evolved search solutions.

This research lays the basis for a novel approach in algorithm design and analysis using GP, defining the terminology, framework, and methods. The paper is organized as follows: In the next section we describe our setup and results for evolving search algorithms. In Section 3 we discuss our approach to post-evolutionary analysis, and develop and demonstrate our system for detecting common semantic building blocks in evolved search algorithms. Related work on program evolution and post-evolutionary analysis is described in Section 4, with concluding remarks and future work following in Section 5.

## 2 Evolving Efficient List Search Algorithms

In this section we detail our evolutionary setup, methodology and results for the problem of evolving array search algorithms.

### 2.1 The Evolutionary Setup

We use Koza-style GP [41], in which a population of individuals, each represented by an ensemble of LISP-like S-Expressions composed of functions and terminals, evolves. 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 [51], which may ultimately help in evolving more understandable algorithms. We used the ECJ package to conduct the experiments [47].

#### *2.1.1 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` incremented after each

**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 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 |

```
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.

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.

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 2.3 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.

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.

### *2.1.2 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)`:

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

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 $t$th 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.

Of note is the absence of a parsimony factor in our fitness function. This is due to two reasons: (1) reported deleterious effects of parsimony pressure [64, 65], and (2) our interest in obtaining code of any size—and then being able to analyze it (intelligently) nonetheless.

**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 2.1.2) |
| Selection | Tournament of size 7, elitism of size 2, generational |
| Population Size | 250 |
| Initial Population | Created using ramped-half-and-half, with depth between 2 and 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 |

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, Jr. [35] 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 (in Section 3 we address the issue of automated analysis). Though some solutions were quite large, the intuition behind the *algorithmic idea* could be gleaned by focusing on the ADF code (Section 2.3).

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.

## 2.2 Results

### *2.2.1 Linear*

Evolving a linear-time search algorithm turned out to be 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

```
(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-like genotype. (b) Equivalent Java code, which, when plugged into the full-program template (Fig. 1), forms the complete algorithm. (Note: the actual Java 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.)

```
(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-like genotype. (b) Equivalent Java code, which, when plugged into the full-program template (Fig. 1), forms the complete algorithm.

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.

#### *2.2.2 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. The evolutionary runtime was quite varied, with successful runs taking from 29 minutes to 6 days (the average being about 2 days). More statistics are given in Table 5, in Section 2.3. 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*.

Note the non-intuitive `(INDEX:= M1)` instruction, discussed in detail in Section 3.6, which deals with a base case of the binary search step (an interval of size 2, where the key is found in the second index).

**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** | | | |
| +, −, × | 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 |

### 2.3 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, augmenting the evolutionary setup with the mechanism of automatically defined functions (ADFs) [42], which we apply to the sublinear-search problem. (Note on terminology: We use the term *main tree* rather than result-producing branch (RPB) [42], 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. The evolutionary runtime was also varied, with successful runs taking from 48 minutes to 14 days (the average being a few days). Some more statistics are given in Table 5.

The ADFs in the perfect solutions are equivalent to one of the following: `(M0+M1)/2`, `(M0 + M1 + 1)/2`, or `(M0/2 + (M1 + 1)/2)` (all fractions truncated); to wit, they are

**Table 4** GP parameters for ADF runs.

| | |
|---|---|
| Objective | Find a key in a given input array of sorted positive integers, in a prefixed number of iterations |
| Function and Terminal sets | As detailed above in this section |
| Fitness | Average error per `search` call on training set, with bonus reduction for hits (as detailed in Section 2.1.2) |
| Selection | Tournament of size 7, elitism of size 2, generational |
| Population Size | 250 |
| Initial Population | Created using ramped-half-and-half method, with depth between 2 and 6 for main tree and between 1 and 2 for ADF |
| Max tree depth | main tree: 10; ADF tree: 4 |
| Generations | 5000 (or until individual with perfect fitness emerges) |
| Crossover | Standard subtree exchange from same tree (main or ADF) in both parents |
| 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 0.25 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 |

```
(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-like genotype. (b) Equivalent Java code, which, when plugged into the full-program template (Fig. 1), forms the complete algorithm.

reminiscent of the original `[M0+M1]/2` terminal we dropped. 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 (automatic simplification is among the goals of G-PEA, presented in the next section).

## 3 Task-Oriented Post-Evolutionary Analysis

At the end of the run of an evolutionary algorithm, the best-of-run individual is usually taken as the output of the algorithm. Typically, when employing genetic programming, such solutions are difficult to read and comprehend. The solutions tend to be

**Table 5** Statistics of GP runs. 50 runs per setup were executed. The "Gens" column lists the number of generations until evolution stopped (either 0 fitness was achieved or the upper bound of 5000 was reached). The "Size" and "Hits %" statistics refer to the best-of-run individual, and show the number of its nodes and the percentage of correct answers ("hits") on the final generality test (i.e., with much larger arrays).

| **Setup** | | **% of Runs** | | **Gens** | **Hits %** | **Size** |
|---|---|---|---|---|---|---|
| Linear | All runs | 100% | average | 832 | 98 | 42 |
| | | | stdev | 1152 | 14 | 35 |
| | "Perfect" runs | 92% | average | 726 | 100 | 38 |
| | | | stdev | 1018 | 0 | 30 |
| Sublinear | All runs | 100% | average | 1560 | 93 | 131 |
| | | | stdev | 1560 | 20 | 65 |
| | "Perfect" runs | 70% | average | 1054 | 100 | 116 |
| | | | stdev | 966 | 0 | 49 |
| Sublinear with ADF | All runs | 100% | average | 3542 | 44 | 175 |
| | | | stdev | 1940 | 42 | 76 |
| | "Perfect" runs | 24% | average | 913 | 100 | 92 |
| | | | stdev | 1280 | 0 | 51 |

bloated [57], to contain unnecessary code (syntactic introns) and unoptimized code (semantic introns) [5], and to be non-human-friendly in general. Thus, post-evolutionary analysis of the obtained solutions is common practice in genetic programming [41, 57]. In many cases, such analysis includes some manual simplification and interpretation of the solutions, or portions of them thereof (as we did above).

In many domains, it is not always possible to apply efficient code simplification of the evolved solutions. Moreover, reducing solution size is not necessarily sufficient for understanding the manner by which the code tackles the underlying problem, and more sophisticated domain-specific analysis is required, e.g., as in [17, 27, 28].

While it seems like an intuitive goal for GP solution analysis, the concept of *code reasoning* has no trivial interpretation. For human-written programs, it is usually possible to determine the "reason", or "purpose", of a relatively complex expression in the program. In contrast, in the case of automatically produced programs such as those obtained from a GP process, it may be quite difficult to decipher expression tasks (if such a task can, indeed, be defined at all). Many other terms in the GP analysis terminology are not formally defined in a coherent manner throughout the literature. Among these, there are terms such as *phenotype*, *building block*, and *context*, for which it is possible to find several definitions (e.g., [15, 36, 45, 49, 54, 63]). *Building blocks* have been studied extensively, and in many of the previous works the definition of a building block was accompanied by a theoretical discussion, some extending similar concepts in GAs [23].

In this work we provide an alternative definition of a GP building block, which is based on the *semantics* (or *phenotype*) of expressions, rather than on their *syntax* (or *genotype*). Here, we define the semantics of an expression in terms of the modifications it engenders in an environment (McPhee et al. [49] also addressed the semantics of building blocks, though in a manner different than ours—see Section 4).

Using this approach, we propose a paradigm for code reasoning entitled *task-oriented analysis*, and exemplify its utilization for the post-evolutionary analysis of the array search problem. We include a general framework for post-evolutionary analysis, which is expected to mine information of high quality from the results obtained

by evolutionary runs. We use the term *expression*, where possible, instead of tree or subtree, since we believe that our approach can be used for different kinds of genetic programming. As noted, we exemplify it using tree-based GP. For ideas on other types of GP, see Section 5.

### 3.1 Terminology

In this section we formulate some basic concepts that will be used later on. Our first step to fully understanding a large evolved solution is to break it into a number of smaller chunks, and analyze the more interesting ones separately. In the world of tree-based GP, this entails processing evolved solution trees to discover their potentially interesting subtrees.

The basic elements of computer programs are *code expressions*, where each expression may be a terminal (i.e., some basic phrase in the given programming language), or composed from other sub-expressions. In the standard GP world, expressions are represented as nodes in a tree, where terminal expressions are leaves, composite expressions are internal nodes, and the root corresponds to the complete individual genome. An *environment* of an expression is defined by the set of values assigned to the variables that are examined or modified by the expression. The execution of some expression within a program causes the program to modify the current environment. Therefore, expressions can be viewed as *implementations of functions*, for which both the domain and the range are the set of possible environments. In some special cases, such as arithmetic expressions, this modification affects a single variable (containing the return value). Note that this would be the case for GP languages with no side effects. In the general case a single expression can modify a subset of variable values in the environment. We consider two expressions to be *equivalent* if both implement the same function. Note that this definition of equivalence does not imply syntactic identity.

Even if the exact function implemented by some expression can be defined, it may still be complicated for a human to understand the significance of this function, and the manner by which it is utilized by a GP individual in order to "fit its surrounding." Moreover, given a specific position within some evolved GP algorithm, it is possible that some of the environments may never occur when running the algorithm up to this position, and thus the manner by which the expression modifies such environments is irrelevant for understanding its essence.

In our paradigm, we interpret the concept of code expression *reasoning* as supplying a compact, clear, and human-friendly abstraction of the environment modification induced by the execution of expressions. We use *conditions* to abstract key properties satisfied by environments, where a *condition* is a predicate applied over an environment, resulting in either true or false values. We formulate the term of expression *task* as a pair of conditions—a *precondition* and a *postcondition*—and say that an expression *admits* some task if for every environment satisfying the *precondition*, executing the expression will result in an environment satisfying the *postcondition*.

To demonstrate the concept of expression tasks, consider the array search problem. It is possible to formulate the required task for an algorithm that solves this problem by defining the precondition "`Array` is sorted and it contains `KEY`, and `M0` and `M1` point to the first and last indices in the array, respectively", and the postcondition "`INDEX` is the index of `KEY` within `Array`". Any expression that is executed given that the environment in the beginning satisfies the precondition, and is guaranteed to result

with an environment satisfying the postcondition, fulfills the required task and thus is a valid solution for the problem. Of course, if the precondition does not hold (for example if the input array is not sorted), there is no guarantee that the execution terminates with an environment satisfying the postcondition. In order to perform this complex task, the algorithm needs to contain sub-expressions that perform auxiliary or partial tasks of the goal task. A typical example for the array search problem is that of an expression whose task is to reduce the size of the interval in which the algorithm looks for the key, while guaranteeing that the key remains within the reduced interval. This task can be formulated by defining the precondition "`Array` is sorted and it contains `KEY`, the index of `KEY` in `Array` is at least `M0` and at most `M1`, and `M1 - M0 = R` (for some integer `R`)", and the postcondition "`Array` is sorted and it contains `KEY`, the index of `KEY` in `Array` is at least `M0` and at most `M1`, and `M1 - M0 < R`".

Note that the environment of some expression within a program might be input dependent, and thus can be deterministically defined only with respect to a specific run of the program. On the other hand, it is sometimes possible to capture in a condition common features of all possible environments at some stage of the program (e.g., "whenever the program executes expression $e$, the value of variable $x$ is at most 100"). We refer to such a condition as the *context* of the expression within the program.

Note that it might be possible to formulate several different pairs of preconditions and postconditions for a given expression, thus it may be argued that it admits several tasks. Nevertheless, in the process of reasoning about the code, we aim to detect those tasks which are crucial for the success of the algorithm, and ignore tasks that have no effect on it. We refer to such essential tasks as *semantic building blocks*.

In this paper we use the terms "functional," "semantic," and "phenotypic" interchangeably; also used interchangeably are the terms "structural," "morphological," "syntactical," and "genotypic." Aside from convenience, this usage also reflects our philosophy regarding the conflation of these terms into a unified meaning.

### 3.2 Algorithmic Framework

We present a general framework for utilizing *task-oriented analysis* in the process of post-evolutionary reasoning about code. This framework can be easily extended and improved, or tailored with domain- and problem-specific adaptations. At this stage, we only assume the availability of a procedure that assesses *semantic similarity* between a pair of expressions, where such a measure is described in the next subsection.

The framework is motivated by the assumption that if some expression performs a task that is crucial to the success of the individual—hence comprising a building block—other expressions implementing the same building block should appear in many solutions that were found by GP, possibly independently evolved. On the other hand, any task that is not important to the success of an individual is not likely to abound in the resulting independently evolved solutions. Thus, if multiple repeats of a single task are detected, it is reasonable to consider this task as a candidate to being a building block.

Note that the last assumption does not imply the existence of *identical*, or even *equivalent*, repetitive copies of a given expression, which may be quite rare among separately evolved programs. Rather, in order to apply such a "common-task" data-mining approach in an effective and practical manner, one needs to address the challenge of seeking repetitions of *similar* expressions, where similarity should correlate with the

corresponding set of common tasks shared by the expressions. This approach follows the same logic as that observed in real biological evolution, where genome elements that are conserved in a wide range of organisms seem to be more likely to carry out important biological functionality [8, 50, 67, 71].

The automatic detection of expressions admitting such essential tasks is a key step in understanding the manner by which the evolved algorithms behave. As a first step in the process of reasoning about code, we suggest here a general framework for semantic building-block detection. The identification of these building blocks can then supply insights into the solution strategies developed by evolution.

Our system, dubbed G-PEA (GP Post-Evolutionary Analysis), assists the analyzer in detecting building blocks, by employing two bioinformatics-based techniques: 1) assessment of similarity between a pair of trees [46, 56], and 2) clustering of similar trees [60] (Fig. 5). The raw material supplied to G-PEA is a host of evolved solutions from multiple GP runs.

The G-PEA system framework is defined as follows:

**Phase 1:** Assemble as input a set of the best evolved solutions, obtained from several independent runs of the evolutionary algorithm.

**Phase 2:** For every pair of (sub-)expressions appearing in the input set, compute the corresponding semantic similarity measure.

**Phase 3:** Apply a clustering procedure for detecting clusters of similar expressions, and report each of the obtained clusters. Then try to deduce a common task for the expressions within the cluster, which is a candidate semantic building block.

Note that in the case of standard GP, expressions are represented by trees, where an expression can be composed of several simpler sub-expressions that appear as its child subtrees. If the similarity between two expressions is computed, in Phase 2, with respect to the similarity between their corresponding internal sub-expressions (as in this work), iterating over the expression trees should be done in a bottom-up, post-order manner (Algorithm 1). This guarantees that when computing the similarity between two expressions, the similarities between all their corresponding sub-expressions (i.e., subtrees) have already been computed. This fact can be used to achieve a semantic similarity measure computed in O(1) time for every pair of expressions. Thus, the overall runtime of Algorithm 1 is quadratic in the number of nodes in the input set. For ideas on other types of GP, see Section 5.

**Algorithm 1** Pairwise-based Similarity Computation

```
1: // Input: 𝕊 – Set of GP trees
2: // Output: distance – Pairwise distance matrix between every two subtrees in 𝕊
3: for every subtree i ∈ 𝕊 (post-order) do
4:    for every subtree j ∈ 𝕊 (post-order) do
5:       distance[i][j] ← semantic similarity between i and j
6:       // similarity calculation is based on distances between the children of i and of j,
7:       // already computed due to bottom-up post-order
8:    end for
9: end for
```

As we will show later, Phase 3 (clustering) helps achieve two goals: First, it reduces the number of "false positive" hits, where an expression is considered important only if similar expressions appear in several other solutions (and not only in a single

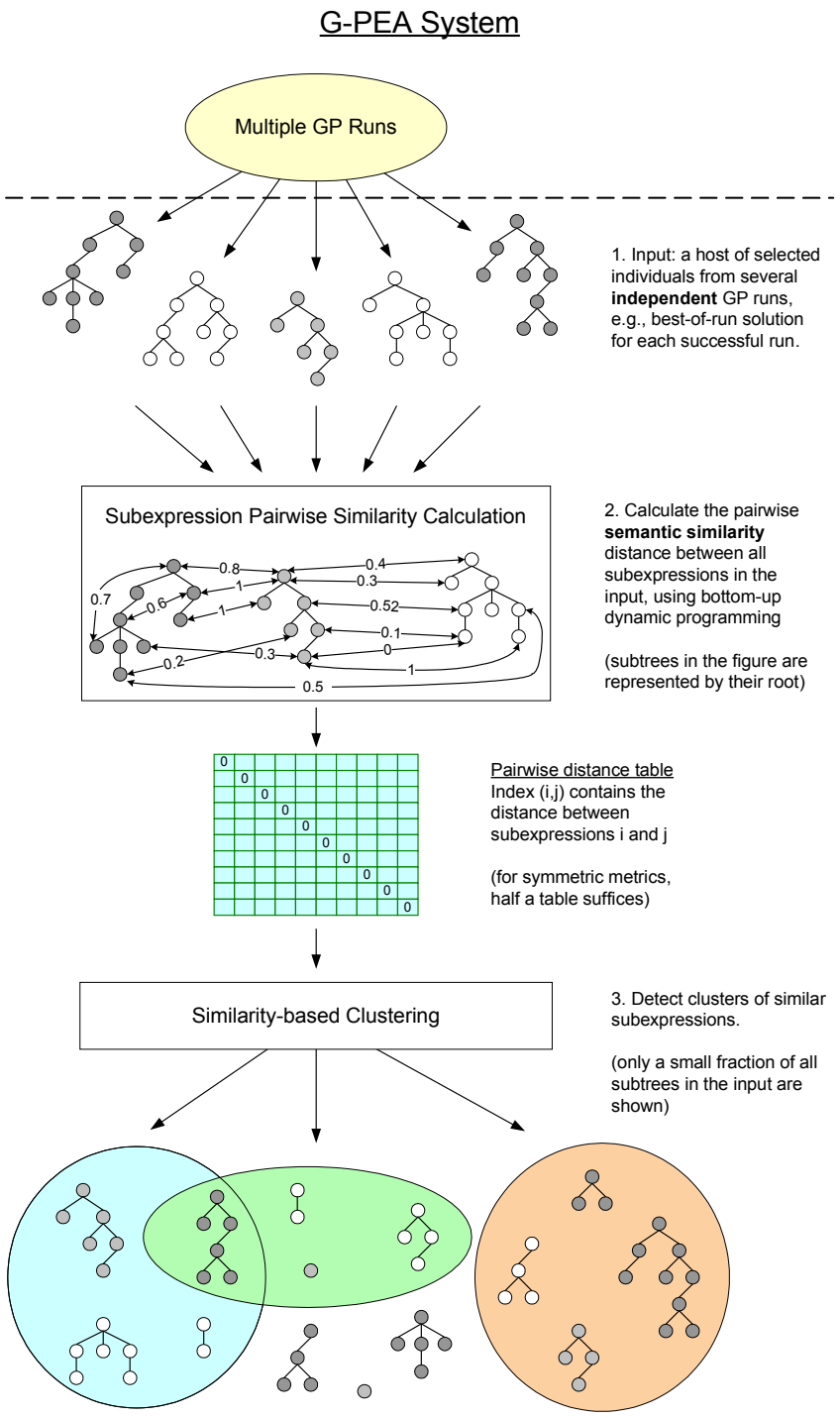



**Fig. 5** Overview of the G-PEA system. The input to G-PEA is a multitude of solutions evolved by GP (e.g., best-of-runs). G-PEA first computes the pairwise distance between all sub-expressions in the input, and then clusters similar ones. By detecting these clusters the system enables the GPer to infer common tasks exhibited by the cluster members, i.e., to reason about—and possibly simplify—their evolved, likely bloated, spaghetti code.

additional solution). Second, by examining together a set of functionally related expressions, it is easier for the human analyzer to infer a common consensus task for all of the expressions, than to infer such a task only for a pair of expressions.

### 3.3 Measuring Expression Similarity

As defined above, two expressions are considered *phenotypically equivalent* if they perform exactly the same function. Thus, for example, the expressions `(x:= (+ y z))` and `(x:= (+ z y))` are phenotypically equivalent, since it is easy to see that applying each one of the expressions, given some environment, will result in exactly the same (modified) environment.

Relaxing the demand of equivalence, we may think of *phenotypic similarity* between two expressions, where such similarity may be correlated with the probability that the two expressions modify a given environment in the same manner. Or, alternatively, the phenotypic similarity between two expressions could be measured as a function of the similarity between the two modified environments obtained from the execution of the two expressions. For example, we may argue that the expressions `(If B X Y)` and `(X)` are similar since in some cases (when `B` is true) they perform *exactly the same environmental modification*, and that the expressions `(x:= (+ x 100))` and `(x:= (+ x 101))` are similar since they result in similar modified environments for every given environment.

In general, the problem of determining whether two expressions are phenotypically equivalent is known to be undecidable [62]. Moreover, the actual phenotypic behavior is usually engendered by the *context* of the expression within the containing solution (i.e., the set of possible environments that may occur when the expression is executed). Full-blown static context analysis (compiler-like) is beyond the scope of this work, and is detailed in the future work section. It is still possible to calculate exact phenotypic behavior without such analysis, yet it would require applying various environments to cover all possible representative scenarios (e.g., [49]), a highly challenging task in its own right, which is also computationally intensive. Since phenotypic equivalence is a special case of phenotypic similarity, it is clear that it would be impractical to directly compute the phenotypic similarity measure as suggested above.

Nevertheless, some evidence of phenotypic similarity may be obtained from the *genotypic similarity*, which is based on the *structural resemblance* of expressions. If two expressions are identical, it is clear that they have exactly the same phenotype. For nonidentical expressions, we may try to assess their phenotypic similarity based on their genotypic similarity. Thus, little genotypic dissimilarity between two compared expressions may have a low impact (or none at all) on their phenotypic similarity measure (e.g., the above example of reordering the arguments of a `+` operation). On the other hand, in those cases where no evidence for phenotypic similarity can be found between two dissimilar genotypes, no phenotypic similarity will be assumed. Though it is not always possible to correlate genotypic and phenotypic similarities, we demonstrate in this work that genotypic-based similarity can still expose important functional patterns, despite its limitations. Our approach differs from previous approaches that focused on genotypic structural motifs, such as schemata, tree fragments, and exact subtree repetition [45, 54, 63]. We use genotypic information as a *means* to assess environmental modifications, rather than as the goal of the analysis.

It is important to observe that expression similarity is completely independent of the specific problem at hand (i.e., the fitness function that assesses individual qualities), rather, it depends only on the language that is used to describe individuals. Therefore, defining and implementing a phenotypic similarity measure can be done once per language, then to be shared when analyzing results of evolutionary processes of several different problems.

Since standard GP expressions are tree-like structures, we base our similarity measure on a standard tree similarity metric: *tree-edit distance* [7, 53]. In the edit-distance metric, the similarity between two trees corresponds to the amount of "editing" one must apply to one tree in order to obtain the other. In Section 4 we survey previous applications of tree-edit distance in the field of GP.

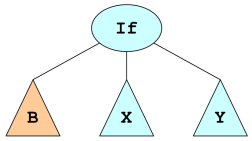
Consider a set of tree-edit operations, such as subtree pruning and attaching, children reordering, replacement of a tree with one of its subtrees, etc., and assume that a cost (or a penalty) is associated with each such edit operation. Given two trees, $T_1$ (source tree) and $T_2$ (target tree), an *edit script* between $T_1$ and $T_2$ is a sequence of edit operations that, when applied to $T_1$, transforms it into $T_2$. Let the cost of an edit script be the sum of costs of its operations. The *edit distance* between $T_1$ and $T_2$ is then defined to be the minimum cost of an edit script that transforms the former tree to the latter.

In this work, we normalize distances to within the range $[0, 1]$, and define that a distance value of 0 reflects *phenotypic equivalence* between two trees, or subtrees. A distance value of 1 corresponds to the case where no indication of phenotypic similarity could be detected. Any other distance value in the range [0,1] inversely reflects the level of our confidence in the equivalence between the trees: the closer the score is to 0, the more semantic similarity was detected between the trees. We make a small exception to this definition with respect to the special case of Boolean expressions, as will be described later on.

The distances are calculated in a recursive manner, such that the distance between two subtrees is derived from the (previously computed) distances between their respective children, as will be explicated below. Note that Algorithm 1 generalizes this approach, and replaces recursion with dynamic programming, for efficiency. In this work we chose to define a symmetric distance measure, though this is not mandatory.

### 3.4 Edit Operations and Cost Examples for the Array Search Language

We show several examples of increasing complexity with respect to the array search language, displaying the intuition behind the selection of the allowed edit operations and their costs. The operations were chosen by following a few simple guidelines: The end goal is to ascribe a low distance value to semantically similar expressions; the system should be robust enough so as not to depend on a specific parameter or cost, and, when in doubt, prefer to err on the side of caution: the tool should be encouraged to highlight potential building blocks, and we prefer a few extra false positive-marked expressions over missing potentially important blocks. Parameter tweaking is an integral part of every complex system, but we found that we hardly had to experiment with the edit operations—most of our initial costs were sufficient for detecting the interesting building blocks presented below, without inundating us with too many results to sift through.




(a) `(If B X Y)`


(b) `Z`

**Fig. 6** Distance calculation: Example 1. Two code expressions: (a) an `If` expression with a predicate branch `B` (returning a Boolean value), a consequent branch X, and an alternative branch Y (both returning void); (b) a general (void-returning) expression Z.

Note that these examples, as suggested by the title of this subsection, relate to the GP *language* presented in this paper, and are independent of the array search *problem* itself. Most functions in this language are standard, general-purpose nodes, like `If` and `PROGN2`, and are used in many GP problems. For clarity, we present only a selection of operations and costs, omitting the full technical details. The complete list can be found in the supplemental online material. We assume that nodes of different types (void, numeric, or Boolean) are incomparable, with a similarity distance of 1.

*Example 1.* As a first example, consider the two expressions in Fig. 6. To convert expression (a) into expression (b) we can apply the following edit operation sequence:

1. convert $X$ into $Z$,
2. convert $Y$ into $Z$, and
3. replace the `If` node with $Z$.

Once both children of the `If` node equal $Z$, the complete expression is equivalent to $Z$, and thus the cost of the last replacement operation is 0. The cost of the two conversion operations is obtained by summing the (recursively computed) distances $distance(X, Z)$ and $distance(Y, Z)$, normalized by a factor of 0.5 each, thus maintaining the requirement for a score in the range [0,1]. The intuition is that in some cases expression (a) executes sub-expression X and in others sub-expression Y, whilst the target expression (b) always executes Z. Therefore, the cost of the genotypic change from (a) to (b) is based on both conversion costs. We chose 0.5 as the normalizing factor, because we assume we have no information about the value distribution of the Boolean child B. If we do know that B is true with probability $p$, we can refine the above formula to

$$cost_{if-void}((a),(b)) = p \times distance(X, Z) + (1 - p) \times distance(Y, Z).$$

The value of $p$ can be estimated empirically or by other semantic genotype analysis techniques. We use the $cost_{if-void}$ notation to denote that this value is the total penalty for the edit script composed of the three edit operations mentioned above, converting an `If` expression to a void-returning expression (Z). There may be other edit scripts for converting (a) to (b), as demonstrated below, and by definition, the value of $distance((a),(b))$ will be the value of the minimum-cost script.

If context information is taken into account, we may notice that X is executed only when B evaluates to true, thus the set of possible environments for the execution of X is restricted to those satisfying the condition that B is true. This can influence

the computation of the distance between X and Z, for example, in the case where the information that B is true implies some knowledge on the phenotypes of X and Z.

*Example 2.* As another example, we show how to compute the distance between expressions that return a Boolean value. In this special case, we define a distance score of 0 to reflect an equivalence between the two compared expressions, and a score of 1 to reflect an equivalence of one expression to the *Boolean negation* of the other. Any value in the range [0,1] represents our confidence in either of these cases (a score of 0.5 indicates we have no information about the correlation). Note that the two cases are complementary, so a score composed of such a single component suffices in this case.

To exemplify this, consider the expressions given in Fig.7. If we know in advance that sub-expressions $A$ and $C$ are equivalent, and that sub-expressions $B$ and $D$ are also equivalent, we can conclude that (a) evaluates to true if and only if (b) evaluates to true. Thus, (a) and (b) are equivalent, i.e., $distance((a),(b)) = 0$. On the other hand, if we know in advance that $A$ is equivalent to $D$ and $B$ is equivalent to $C$ then (a) evaluates to true implies that (b) evaluates to false, and vice versa (note that this relation does not hold in the special case where $A$, $B$, $C$, and $D$ evaluate to the same value, when both Boolean expressions evaluate to `false`; we address this possibility below). In this case (a) is almost equivalent to the negation of (b), and $distance((a),(b))$ should be close to 1. Relaxing these requirements in order to replace equivalence with similarity, we can compute a distance score between 0 and 1, according to the following formulas:

$$directCost((a),(b)) = 1 - (1 - distance(A,C)) \times (1 - distance(B,D)),$$

$$negatedCost((a),(b)) = 1 - (1 - distance(A,D)) \times (1 - distance(B,C)).$$

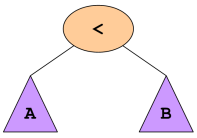
$directCost$ is a value in the range $[0,1]$, which is closer to 0 the smaller the distance is between A and C and between B and D. Similarly, $negatedCost$ represents the combined similarity in the reverse order.

Note that if we have no similarity information (distances are all 1), both costs equal 1. Combining $directCost$ and $negatedCost$ to obtain a distance score for the whole expression, we get

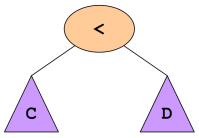
$$cost_{<-<}((a),(b)) = \frac{directCost((a),(b))}{directCost((a),(b)) + negatedCost((a),(b))},$$

which fulfills our requirements: the value is between 0 and 1, where 0 means equivalence and 1 means total dissimilarity, i.e., equivalence to the negated case. For the case where both $directCost$ and $negatedCost$ are equal, the distance is 0.5. For the special case where A, B, C, and D are all equivalent, we define the distance to be 0, since in this case the two expressions (a) and (b) always evaluate to false and thus are equivalent.

A more in-depth look at the last example shows that the distance calculation should be further refined, to compensate for the fact that $<$ is not the exact opposite of $>$, but rather is the opposite of $\geq$. For example, consider Fig. 8. Here, the previous formula would yield the score of 1, indicating a scenario of: (a) true if and only if (b) false. But note that if A and B evaluate to the same numerical value during a certain execution of the programs—even though they are not equivalent—both the (a) and (b) expressions will evaluate to false. Our formulation can be extended to include a penalty for $negatedScore$, to compensate for such cases. We have implemented this in our model, but exclude the specific details here for brevity.

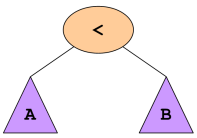



(a) `(< A B)`

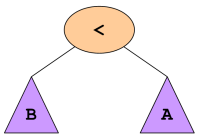



(b) `(< C D)`

**Fig. 7** Distance calculation: Example 2. Two less-than expressions. (Expressions `A`, `B`, `C` and `D` all return an integer value.)




(a) `(< A B)`




(b) `(< B A)`

**Fig. 8** Two less-than expressions, with identical children in reverse order. (a) is not equivalent to the negation of (b) since both return false in the case of `A` and `B` returning the same value.

Following the reasoning of the examples given above, we have designed similarity rules and costs for each pair of Boolean expression types (functions and terminals in the function and terminal sets). For instance, the penalty for similarities between the pairs $(<, >)$ and $(>, >)$ were defined in a similar manner to the definition of the $(<, <)$ expression pair discussed above.

Note that the above example could have been simplified, and the comparison strengthened to be more precise, if the underlying language had allowed us the minus (`-`) function and `0` terminal. We could have then transformed the expressions in Fig. 7 to the equivalent canonized form: `(< (- A C) 0)` and `(< (- B D) 0)`, and thus would only have had to compare two arithmetical expressions. This would have allowed us to also cover cases in which there is no evidence for similarity, e.g., between `A` and `C` and between `B` and `D`, yet we do know something about the similarity of the two minus expressions.

*Example 3.* Next, we look at the case of two `If` expressions, depicted in Fig. 9. As before, to assess the similarity between these expressions, we try to answer the question: "Given some environment, how similar would the two modified environments be, after applying each one of the expressions?" When executing expression (a) either sub-expression X or Y will be executed, depending on the value of B1 in the given environment; similarly for (b). Thus, the similarity between (a) and (b) should incorporate the similarity between the sub-expressions as well as our confidence in which sub-expressions will be executed in each tree, given a common environment.

This confidence is represented by the distance between the Boolean sub-expressions. The closer $distance(B1, B2)$ is to 0, the higher is our confidence that in a given environment the Boolean sub-expressions will return the same value (true or false). Therefore, either X will be executed in (a) and Z in (b), or Y in (a) and W in (b). Thus, when


(a) `(If B1 X Y)` (b) `(If B2 Z W)`

**Fig. 9** Distance calculation: Example 3. Two `If` expressions, wherein distance calculation has to take several edit scripts into account.

$distance(B1, B2) = 0$ we would like the distance score between (a) and (b) to depend solely on $distance(X, Z)$ and $distance(Y, W)$. For a $distance(B1, B2)$ close to 1, the score should mainly rely on the distances in reverse order, i.e., $distance(X, W)$ and $distance(Y, Z)$. The following formula incorporates the above intuition to compute similarity between two `If` nodes:

$$cost_{if-if}((a),(b)) = (1 - distance(B1, B2)) \times \frac{distance(X, Z) + distance(Y, W)}{2}$$

$$+(distance(B1, B2)) \times \frac{distance(X, W) + distance(Y, Z)}{2}$$

Again, instead of choosing the factor 0.5, the two elements of the above formula might be factored with $p$ and $(1 - p)$ respectively, for some $0 \leq p \leq 1$, if we know something about the probability of the Boolean sub-expressions to evaluate to `true`.

The above formula computes the cost for converting the children of the `If` node in (a) to the children of the `If` node in (b), and is marked $cost_{if-if}((a),(b))$. Since the *distance* between (a) and (b) is the cost of the minimum-cost script, we have to consider more edit scripts to calculate it.

Another script that comes to mind is converting both X and Y to the complete (b) subtree, and then removing one of them along with the `If` parent, leaving us with a single copy of the (b) subtree. This is the exact script discussed in Example 1, with the void-returning (b) subtree acting as the Z expression in the example. Thus, we will use the formula from Example 1 to compute $cost_{if-void}((a),(b))$.

Symmetrically, we have the script that first duplicates the (a) subtree and adds an `If` parent node and a B2 left sibling, and then converts one copy of (a) into Z and the other into W. This script is marked $cost_{void-if}((a),(b))$ and its cost is calculated by applying Example 1 to converting (b) to (a), i.e., it is equal to $cost_{if-void}((b),(a))$.

Assuming there are no other edit scripts between (a) and (b), the final distance between these expressions can then be formulated as:

$$distance((a),(b)) = \min\{cost_{if-if}((a),(b)), cost_{if-void}((a),(b)), cost_{void-if}((a),(b))\}.$$

*Example 4.* As another example of distance calculation, consider the expressions given in Fig. 10. In expression (a) both sub-expressions X and Y will be executed, and in expression (b) both Z and W will be executed. There are several plausible edit scripts for converting (a) to (b) and, as mentioned, the final distance between (a) and (b) will be set to the cost of the lowest-cost edit script.


(a) `(PROGN2 X Y)` (b) `(PROGN2 Z W)`

**Fig. 10** Distance calculation: Example 4. Two `PROGN2` expressions. Various penalties can be incurred, such as a penalty for child-order reversal.

One edit script converts X to Z and Y to W. The smaller the distance between X and Z and between Y and W, the smaller the cost of this script. Another script converts X to W and Y to Z, and then reverses the order of the `PROGN2` children. This edit script will have to incorporate some penalty for reversing execution order, because in some cases this impacts upon the performed task. For instance, an `(x:= (+ y 1))` followed by `(y:= 5)`, will perform a different task than if the expression order were reversed. This penalty can be assessed on a per-case basis (depending on the given expressions and the common environment variables they access), but a simpler option is to ascribe a small penalty to child-order reversal. This latter method worked well for us.

Another edit script that comes to mind is the conversion of X to the whole (b) expression, and the deletion (pruning) of the Y sub-expression (or, in other words, replacement of (a) root with the converted X). The edit scripts converting Y to (b), Z to (a), or W to (a), are symmetric cases, and should also be considered. Note that the distance between the sub-expressions and the whole expression, e.g., between X and (a), can be recursively computed, so the distance calculation between (a) and (b) just has to select the minimum of the aforementioned script costs.

### 3.5 Results: Pairwise Similarity

We demonstrate the first part of our G-PEA system (Fig. 5), namely, computation of pairwise similarity, on the 35 correct binary search solutions evolved in Section 2.2. We chose the language without ADFs because the terminal `ADF0` might have different phenotypic behaviors in different solutions. The model can, of course, be adapted to calculating the distance between ADF genotypes, but special care would have to be taken for some of the functions. Some distance rules for arithmetic functions could resemble the ones we defined, e.g., the rule to calculate the distance between expressions (+ A B) and (+ C D) might be similar to the rule for (= A B) and (= C D), since both do not take the order of the children into account. But other functions, like division and subtraction, would have to adhere to different rules, with a different penalty scheme.

Before analyzing the evolved programs we "cleaned" them up by applying some basic simplification rules, as in [18, 74], in order to remove some of the semantic introns [5] found in our solutions. These were applied in a recursive manner and included rules such as `(INDEX:= INDEX)`→`NOP`, `(PROGN2 NOP X)`→`X`, `(= A A)`→`TRUE`, `(If (TRUE) X Y)`→`X`, etc. To achieve tree canonization, we also applied the rule: `(PROGN2 X (PROGN2 Y Z))`→`(PROGN2 (PROGN2 X Y) Z)`, which "hanged" all subsequent `PROGN2` nodes as left children. All rules were designed not to change the phe-

```
(PROGN2
  (INDEX:= [M0+M1]/2)
  (If (< KEY  Array[INDEX])
      (M1:= INDEX)
       NOP))
```

(a) Tree 30, Subtree 14

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

(b) Tree 32, Subtree 9

**Fig. 11** Similar expressions from two evolved array search solutions, detected by G-PEA, which returned $distance((a),(b)) = 0.024$. The distance calculation for this example is detailed in the online material.

notypic behavior of the individual, while simplifying the genotypes in order to increase the potential of detecting similarity between expressions. If the model is extended to arithmetic expressions, we believe this simplification phase will be highly important. There are several examples in [18, 74] that support this assumption.

After the cleanup phase, the size of the solutions varied between 17 and 175 nodes. The smaller individuals are quite easy to understand, but the larger ones are bloated and difficult to analyze or even to follow. In what follows, we present some examples of similar expressions found by our pairwise distance evaluator. (A note on notation: the trees are numbered 0 to 34, and the subtrees of each solution are numbered starting from 0, in a post-order manner.)

Consider the two expressions given in Fig. 11. This example shows how "almost"-equivalent expressions are found to be very similar by the algorithm. Note that for all possible environments, except for those in which `KEY` evaluates to exactly the same value as `Array[INDEX]`, the execution of both expressions would result in exactly the same modified environment.

Another example of expression similarity detected by our algorithm is given in Fig. 12. In this example, the "spaghetti-code" of expression (a) is, in fact, equivalent to expression (b). First, note that the predicate genotypes in both roots are different—but phenotypically equivalent.

The consequent branch (left child) in both expressions is identical and therefore equivalent. As for the alternative branches (right children, `If` node `#2` in (a), and the `NOP` node in (b)), a closer look reveals that these two expressions, in the given context, are also equivalent. This is so because of `If #2` residing in the alternative branch of `If #1`, and having the same Boolean predicate, thus always evaluating to false, and executing the `NOP` instruction.

Though we did not aim to detect such context-based similarity, our algorithm has flagged these two sub-expressions as similar. If we ignore the context, and assume that each of the three Boolean predicates inside the `If #2` subtree has a 50% chance of being true, 7 out of 8 possible predicate value combinations result in the execution of a `NOP` instruction. Thus, it is *similar* to a single `NOP` node. This example shows the added value of the similarity-based method, since it would have detected similarity even if the context of `If #2` in expression (a), or its nested predicates, had been different.

### 3.6 Results: Similarity-Based Clustering

One of the drawbacks of using the pairwise-similarity criterion alone to detect candidate building blocks is the large number of potential false-positive hits. Using the flexible

```
(If (> KEY  Array[INDEX])        //#1
    (M0:= [M0+M1]/2)
    (If (> KEY  Array[INDEX])    //#2
        (If (> KEY  Array[INDEX])
            NOP
            (If (> KEY  Array[INDEX])
                (INDEX:= [M0+M1]/2)
                NOP))
        NOP))
```

(a) Tree 32, Subtree 60 (size: 23)

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

(b) Tree 19, Subtree 38 (size: 7)

**Fig. 12** Similar expressions from two evolved array search solutions, detected by G-PEA, which returned $distance((a),(b)) = 0.0625$.

similarity measure we have defined, it is likely that many expression pairs will turn out to be similar, regardless of the importance of the task they encode. In order to reduce the false-positive rate, i.e., to report only expressions that are more likely to be significant, we search for those common to a number of separate solutions. Since these solutions evolved in different evolutionary runs, we assume that it is unlikely for semantically similar expressions of sufficiently large size to appear in many solutions, unless they perform an important task for the problem at hand. These tasks are the building blocks we are searching for.

To this end, in the last phase of G-PEA (Fig. 5), we apply a basic clustering approach, by setting a distance threshold $\tau$, and searching for cliques in the set of expressions in all 35 evolved array search solutions (from Section 2.2), such that the similarity distance score between any pair of expressions in the clique is at most $\tau$. Setting $\tau$ to 0 means that all elements of detected clusters must be phenotypically equivalent. Increasing $\tau$ decreases the specificity of the tool, potentially increasing cluster sizes, and the amount of interesting clusters, at the cost of increased genotypic variance within clusters. The cluster examples below were found using $\tau = 0.3$, an intermediate value empirically found to produce interesting results for the scoring scheme we have applied.

Since we are searching for cross-solution common subtrees, we limited the clusters to no more than one subtree per solution. This canceled out the effect of repeating subtrees within the same solution, a situation that might arise due to crossover, possibly resulting in the subtrees' being tagged as common.

In what follows, we discuss some of the clusters detected by G-PEA. Interestingly, we note that many of the detected clusters appear to perform a task related to binary search.

As a first example, consider cluster A, presented in Fig. 13. As can be observed, though (genotypically) different in syntax, all expressions perform a common (phenotypic) task, that of setting `M1` to the value of `INDEX` if `KEY` is smaller than `Array[INDEX]`—a typical subroutine in binary search algorithms. This evidence supports our contention that expressions common to many solutions are likely to play an important role in each evolved solution. Notice that not all expressions in the cluster are equivalent, e.g., expressions (e) and (f), which operate differently for the case where `KEY` equals `Array[INDEX]`.

At a cursory glance expression (d) appears to do an additional task, that of updating `M0` to `INDEX` for some preconditions. Nevertheless, our algorithm has detected that, for a significant number of preconditions, the task performed by this expression is the same

```
(If (< KEY  Array[INDEX])
    (M1:= INDEX)
    NOP)
```

(a) Tree 7, Subtree 106

```
(If (> Array[INDEX]  KEY)
    (M1:= INDEX)
    NOP)
```

(b) Tree 15, Subtree 88

```
(If (< Array[INDEX]  KEY)
    NOP
    (If (= KEY  Array[INDEX])
        NOP
        (M1:= INDEX)))
```

(c) Tree 6, Subtree 97

```
(If (< KEY  Array[INDEX])
    (If (< KEY  Array[INDEX])
        (M1:= INDEX)
        (M0:= INDEX))
    NOP)
```

(d) Tree 9, Subtree 64

```
(If (< KEY  Array[INDEX])
    (M1:= INDEX)
    NOP)
```

(e) Tree 30, Subtree 13

```
(If (> KEY  Array[INDEX])
    NOP
    (M1:= INDEX))
```

(f) Tree 32, Subtree 36

**Fig. 13** Cluster A, expressing a typical binary search building block. Note that not all expressions are phenotypically equivalent, e.g., (e) and (f). Note also that (d) appears to perform additional tasks, but the `(M0:= INDEX)` instruction is unreachable, rendering the expression equivalent to others, e.g., (a).

as the other expressions in the cluster. Thus, the computed similarity score was good enough (lower than the threshold) to be included in the cluster.

Such a scenario, demonstrating an expression which is similar to a significant number of expressions in other solutions, but containing some extra code segments, raises the hypothesis that the additional code segments are possibly insignificant, e.g., semantic introns performing no important task, or simply unreachable (syntactic introns). Taking a closer look at expression (d) we observe that the `(M0:= INDEX)` instruction is in fact unreachable, due to the nested `If` instruction. This implies that expression (d) is actually equivalent to other expressions in the cluster (for instance, (a)), demonstrating the ability of G-PEA to control the sensitivity of the results by relaxing the genotypic equivalence criteria using the clustering threshold $\tau$. Note that incorporating context analysis in the similarity-measuring procedure could have detected the unreachable code segment in (d), giving a more accurate similarity score. Similar arguments explain how expression (c) has joined cluster A, and again, context analysis would have improved the similarity score between it and the other expressions in the cluster.

Another example of a detected cluster is cluster B, shown in Fig. 14. In general, the common behavior of the presented expressions is as follows: If the key is found (i.e., `Array[INDEX]`= `KEY`) then do nothing, otherwise move `INDEX` to the mid-point between `M0` and `M1` (and possibly do some additional computations). At first glance, this task does not seem crucial for the success of a binary search algorithm, and it is unclear why a code performing it proliferates in many solutions evolved in separate runs. However, further inspection reveals that, in fact, the code reflects a technique for dealing with a critical base case of the search step.

Typically, `M0` and `M1` are used by the evolved algorithms to bound the sub-array in which the key may be found. The algorithms use the `(INDEX:= [M0+M1]/2)` instruction to examine the mid-point of this interval, and then halve the interval size by moving either `M0` or `M1` to this mid-point.

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

(a) Tree 6, Subtree 78

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

(b) Tree 9, Subtree 78

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

(c) Tree 11, Subtree 19

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

(d) Tree 22, Subtree 46

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

(e) Tree 20, Subtree 170

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

(f) Tree 26, Subtree 50

**Fig. 14** Cluster B, expressing a non-intuitive binary search building block. The common task in the expressions deals with the base case of an interval of size 2, where the key is found in the second index.

This standard approach for binary search works well for all cases, except for the base case of an interval of size 2 (i.e., `M1`= `M0+1`), in which `KEY` is found in `M1`. In this case, the standard setting of `(M0:= [M0+M1]/2)` does not decrease the interval size, since the value of `[M0+M1]/2` is `M0` (due to integer truncation). The standard setting of `INDEX` to `[M0+M1]/2` does not suffice either, as the correct index to return is `M1`. Without any special treatment of this case, `INDEX` would retain the `M0` value until the last iteration, and the search function would return this incorrect answer. In order to deal with this difficulty, many evolved algorithms developed the following approach: after executing a step that reduces the relevant interval size, `INDEX` is assigned the value of `M1`. Then the common task presented in the above cluster is executed, i.e., if the solution is found—the algorithm maintains it by doing nothing, otherwise—`INDEX` returns to being set to the average of `M0` and `M1`. (Note that this sequence of operations is performed in every iteration, not just for intervals of size 2, since the algorithm cannot discern the size of the current interval.)

Examining the context in which these expressions appear in the full genotypic trees (not displayed here for brevity—see online material), we find that in the four solutions wherein the clustered subtrees (a), (b), (d), and (g) were reachable, the precondition of `INDEX`= `M1` indeed holds in every run. The two expressions (c) and (f) were actually found to be parts of unreachable code segments. A possible explanation for "interesting" code segments being unreachable is that they originate in a crossover operation with another individual, in which they may have been used.

Note that some expressions in the cluster perform additional tasks, after the common task detailed above. As in the example of cluster A, this genotypic variance did not increase the distance score as to exceed the cluster threshold.

The final example shown is cluster C, given in Fig. 15. This is a clear example of the ability of the G-PEA system to detect similarity between code segments of significantly different sizes. The first four expressions, (a) through (d), clearly perform

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

(a) Tree 20, Subtree 169 (size: 10 nodes)

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

(b) Tree 15, Subtree 89, (size: 10 nodes)

```
(PROGN2
  (INDEX:= [M0+M1]/2)
  (If (< KEY  Array[INDEX])
      (M1:= INDEX)
      NOP))
```

(c) Tree 30, Subtree 14 (size: 10 nodes)

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

(d) Tree 32, Subtree 9 (size: 10 nodes)

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

(e) Tree 29, Subtree 16 (size: 17 nodes)

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

(f) Tree 34, Subtree 19 (size: 20 nodes)

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

(g) Tree 31, Subtree 40 (size: 41 nodes)

**Fig. 15** Cluster C. Expressions (a)-(d) encode a typical binary search task. Expressions (e) and (f) include this task and extend it to the cases where `Array[INDEX]` $<$ `KEY`. Expression (g) becomes comprehensible once shown alongside expression (f), a much shorter, equivalent expression.

the same task. The three remaining expressions, (e) through (g), perform the same task and extend it by additional operations for some preconditions. Observe that it would have been very difficult to notice this similarity manually for expression (g), due its large size. G-PEA helps detect such cases automatically.

Untangling the "spaghetti code" of expression (g) presents quite a challenge if one were to do this by hand. G-PEA both flagged the subexpression as important,

and displayed similar expressions that might help to highlight the essence of the task performed by this code. An examination of expressions (f) and (g) reveals that they are, in fact, phenotypically equivalent. Both perform the full task of the binary search iteration, which halves the relevant interval and deals with the base case discussed in the above example of cluster B.

Several expressions in cluster C contain sub-expressions that appeared in cluster A (e.g., expression (e) of cluster A is a sub-expression of expression (c) of cluster C). Note also that the right branch of expression (a) in cluster C, subtree 168 of tree 20, was not detected as part of cluster A, though it is equivalent to several expressions in the cluster. This is due to the fact that the distance between the instructions `(M1:= INDEX)` and `(M1:= [M0+M1]/2)` is calculated to be 1 in our setup, since we did not integrate the estimation of the probability of `INDEX` equaling `[M0+M1]/2`. Adding context analysis to our approach would have improved the distance calculations, as the precondition `INDEX`= `[M0+M1]/2` appears in all cluster-A solutions.

The relatively large expressions in cluster C show that since the pairwise algorithm works on pairs of all subgraphs, it is computationally inexpensive, compared to related tools that enumerate all subtrees (in exponential time). See Section 4.2.

This example suggests a number of important applications for the proposed distance metric. The first application that comes to mind is the automatic simplification of GP code, by replacing large code segments, such as expression (g) above, with a similar, yet simpler, expression. This simplification can be done as a part of post-evolutionary processing, where the simplified solution is re-evaluated and retained if no drop in fitness is observed. The process can also be used *during* evolution for the sake of bloat control. In this case, the fitness criterion might be relaxed. We detail ideas for such future work in Section 5.

Note that our claim is not that *all* or even *most* solutions should contain the *same* building block. Rather, it is suggested that building blocks will repeat in several solutions among the many separately evolved solutions that developed similar strategies to solve the problem. Each cluster in the given examples contains similar expressions from 6-7 solutions out of 35, which—given the sizes of the detected building blocks and the fact that each of the 35 solutions was evolved entirely independently—is highly improbable to have happened by chance alone. Again, we take advantage of the assumption that even though different strategies may evolve, by repeating the experiment a sufficient number of times, some strategies should evolve more than once, and could be detected by our approach.

## 4 Related Work

In this section we review related work both on program evolution and GP program analysis.

### 4.1 Related Work on Evolutionary Algorithm Design

We performed an extensive literature search, finding no previous work on evolving list search algorithms, for either arrays or lists of elements, except our own preliminary results reported in [73]. 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 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 in computer science [39]. 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 Jr. [34, 35], who was able to evolve solutions equivalent to the $O(n^2)$ bubble-sort algorithm. Kinnear Jr. 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. [72]. 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 and Lucas [2, 3]. The evolutionary setup was based on their object-oriented genetic programming system. In [2] 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 [3] 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 and Lucas 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 2.3). Other interesting works on evolving sorting algorithms include [1, 55, 61, 66].

Another related line of research is that of evolving iterative programs. Koza et al. [43] defined automatically defined iterations (ADIs), and Kirshenbaum [38] defined an iteration schema for GP. These constructs iterate over an array or a list of elements, executing their body for each element, and 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) [43], 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).

## 4.2 Related Work on Post-Evolutionary and Building-Block Analysis

Related research includes a number of works that search for repeated patterns in GP populations. Mostly, patterns are *exact* subtree repetitions, an approach with a number of disadvantages that are detailed below.

Tackett [68] mined genetic programs for important (salient) sub-expressions, by keeping track of all distinct subtrees in the population *during* the evolutionary run. Subtrees were enumerated using a hash table. On the subject of *understanding* the sub-expressions, the author suggests using cross-comparison of important sub-expressions developed in separate runs. We address this open problem in our work. Recently, Joó and Neirotti [31] extended Tackett's work by allowing for wildcards in the searched-for patterns, and suggesting an automatic method for identifying salient patterns.

Ciesielski and Li [11] data-mined log files from different GP problems, looking for recurring patterns in the population trees. They used a number of pattern-matching measures: exact matching (identical trees), same tree shapes (ignoring node types), and a measure intended for the commutative properties of nodes such as the '+' function. The following heuristic was defined for this last measure: two trees are commutative-equivalent if they have the same shape and obtain the same fitness. The first measure is quite restrictive, and we believe their last two measures are not general enough to encompass many GP problems, such as languages with relatively large function or terminal sets or languages with side effects like the one described in this paper.

Smart et al. [63] defined a compact representation for tree fragments and showed statistics of fragment frequency during a run, in moderately sized populations. This is an enhancement over simple enumeration-based methods, but still has strong limitations on tree size because the number of fragments in a tree grows double-exponentially with its depth.

Langdon and Banzhaf [45] analyzed repeated patterns in solution trees, using a number of measures, including statistics based on searching for exact subtree and fragment repetitions *within* a GP-evolved solution.

Kameya et al. [32] mined exact subtrees and fragments from the best solutions of each generation, and protected the frequent subtrees during crossover, reasoning that these were potentially good "building blocks."

Our methodology has the following advantages vis-a-vis the approaches described above. Firstly, these approaches report exact occurrences of an enumerated subtree, while our system is far more flexible, able to discover *approximate* occurrences of subtree motifs. Second, the enumeration of all possible subtrees up to a certain size is computationally expensive (increasing exponentially with subtree size), and thus is restricted in practice to the mining of subtrees whose size is bounded by a small constant. Our approach, on the other hand, is based on an alignment algorithm, whose complexity is quadratic in the size of the sought subtrees, enabling us to readily compare subtrees of all sizes. Moreover, any type of intron, inherent to all GP languages, would have a highly negative impact on such exact replica-detecting measures, resulting in many equivalent patterns being detected as distinct ones. Many of the above reported works could be extended with our approach, to possibly yield better results.

Somewhat related to our research is the work of McPhee et al. [49], who defined the semantics of a Boolean expression as the truth table of the expression. Two subtrees were considered semantically equivalent if they shared the same truth table. Their definition can be considered as a special case of our own definition, restricted to environments containing only Boolean variables and expression-induced modifications that

change a single (output) variable value. Nevertheless, there are several fundamental differences between the two approaches. First, McPhee et al. detected semantically *equivalent* expressions, by enumerating all possible environments in a Boolean domain. In our work, expressions with *similar* semantics are detected, using light-weight syntactic analysis of a more-expressive language. In addition, aiming to analyze properties of the GP crossover operator, McPhee et al. focused on data collected during the evolutionary run of a single population. Our work is targeted at post-evolutionary analysis for code reasoning by building-block detection within data obtained from several runs. Moreover, though the term "building block" appears in the title of [49], it is actually never explicitly defined in the paper.

Distances between trees and between populations were used in a number of genetic programming works, mostly for measuring population diversity. Both genotypic and phenotypic methods were defined. We mention here some of those based on tree-edit distance—for a good review of other methods see [10, 20].

Edit distance was first utilized in GP research by Keller and Banzhaf [33], who defined a set of edit operations based on the GP node type (label) in order to adapt GA population diversity measures to the GP world. O'Reilly [53] used edit distance to calculate distances between GP individuals, for analyzing the effects of several crossover operators and the diversity within a population's best individuals. The allowed edit operations were single-node insertion, deletion, and substitution. De Jong et al. [16] used a similarly defined edit distance, normalized by the smaller tree size, for measuring diversity in a multi-objective scenario. Ekárt and Nemeth [19] defined an edit distance metric specifically for GP, taking into account each node's depth and type, and used it to apply fitness sharing [23] to genetic programming.

Wedge and Kell [70] used a genotypic edit distance that ignores subtree size in the substitution operation and compensates by normalizing with respect to the larger tree size. They defined a "fitness distance" measure, calculated the genotype-fitness correlation, and showed that it can be used for predicting preferred population sizes for regression problems, under specific conditions. Tree-edit distance was also used in a number of GP works (e.g., [12, 69]) to calculate Fitness Distance Correlation, a measure of problem difficulty originating in GAs [30]. Forrest et al. [21] used GP for doing software repair: Given an input program with a bug, a population of variants was evolved until a program that passed all fitness tests was found. Edit distance was then used in post-evolutionary processing to identify the set of changes between the original program and the evolved solution.

In this study we use edit distance for a different purpose than the works discussed above. We apply edit distance to the detection of code segments of similar functionality as a step in post-evolutionary analysis. Consequently, instead of reflecting the *morphological modifications*, the penalties were chosen so as to reflect the *functional modifications* of the corresponding expressions (see Section 3.4).

Cummins and O'Riordan [14] defined *phenotypic distance measures* between GP solutions for a problem of information retrieval, and used it to cluster solutions that are 'close' to each other in the solution space. Phenotypic measures were also used for improving crossover [52] and for program simplification [36, 37]. Phenotypic distances suffer from being difficult to define, and very costly to evaluate for many real-world problems. Moreover, their computation might not be able to cover all possible inputs, e.g., in algorithmics or evolution of controllers, thus degrading their accuracy and repeatability.

## 5 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 [39] 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." Our evolutionary algorithm 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).

In order to analyze the evolved solutions, we have defined a novel approach to post-evolutionary GP solution analysis. This approach is based on several important intuitions:

- The standard search for identical structural or syntactical motifs in GP trees is too strict for code created by evolution, which typically includes syntactic introns, for example. It seems that *similarity*-based measures are more adequate for a meaningful analysis of these kinds of entities.
- The analysis relies on the *function* carried out by GP expressions. Like previous works, we examine syntactic features of GP trees, yet our similarity criteria aim to correlate with expression functionality rather than with its syntax or structure. This seems to be a more intuitive goal when analyzing GP programs.
- The repetition of similar fragments in a number of evolved individuals suggests the importance of these fragments, while fragments with no observed similar instances are less likely to play a significant role.

We have refined the definition of several terms, such as building blocks and expression tasks, and given an algorithm for detecting and clustering similar expressions in GP solutions, using a customized tree edit-distance measure. This algorithm was applied to the array search problem, successfully highlighting important non-trivial building blocks in the evolved solutions. The results also showed expressions differing in structure, but with equivalent functionality, demonstrating the strength of our approach.

It may be argued that we have applied G-PEA to a simple problem. While binary search has well-known, efficient solutions, *evolving* such efficient solutions is far from trivial. Moreover, the GP system supplies a wealth of bloated, convoluted solutions, upon which we were able to demonstrate G-PEA. Of course, a major line of future work is applying G-PEA to additional (hard) problems.

Our work opens up a number of possible avenues for future research. In the algorithm evolution field, we would like to explore the coevolution of individual main trees and ADFs, as in the work of Ahluwalia and Bull [4]. Also, our phenotypes are not Turing complete [75], e.g., because they always halt. It would be interesting to use a Turing-complete GP system to evolve search algorithms.

As noted, many of the evolved solutions are bloated. It would be interesting to see how adding parsimony pressure affects evolution and our post-evolutionary analysis. We also plan to delve into related areas, such as other search problems and sorting algorithms, and show evolutionary innovation in action. Ultimately, we wish to find an algorithmic innovation not yet invented by humans.

We plan to enhance our G-PEA system with a number of features and extensions, including:

1. Writing it as a generic, open-source, publicly available package, so it can be used by the GP community, thus taking computer-assisted, post-evolutionary analysis a step closer to being a common practice in GP research.
2. Adding support for a number of common GP languages and functions, such as arithmetic operations.
3. Testing several advanced clustering algorithms that may improve G-PEA's performance.

The ability to assess semantic expression similarity gives rise to many other applications in GP. We list some suggestions for future research, extending the main concepts presented here:

1. *Code simplification.* As the results of this work suggest, expressions performing similar tasks may differ significantly in size. A possible size-reduction simplification can be obtained by replacing sub-expressions in a solution, with similar, smaller sub-expressions. If the replaced and replacing expressions are equivalent, it is guaranteed that the modified solution is valid. Otherwise, a standard validation process may be applied before accepting the modified solution. This simplification, when applied after the evolutionary process, enables one to avoid intense parsimony bloat control during evolution, which may restrict the maintained genetic information [64, 65], and yet to process solutions so that the output of the evolutionary process will be easier to handle by humans.
2. *During-evolution similarity analysis.* Detection of similar expressions during evolution, rather than after it has finished, may help with faster propagation towards the desired solution. Extending an approach presented previously in [74] with identity-based building block detection, it is possible to incorporate online simplification, as well as the following operators within the evolutionary process:
   (a) *Building-block preserving crossover.* An interesting enhancement to evolutionary algorithms is aimed at reducing the probability of crossovers occurring within building blocks (e.g., in [32]), following the hypothesis that such an operation is more likely to reduce individual fitness. Semantic similarity-based building block detection can naturally replace syntactic identity-based building block detection in algorithms applying this feature. We expect it to perform better as it is less sensitive to syntactic introns, and due to the more general definition of building blocks.
   (b) *Block-level mutations.* Once an expression corresponding to a semantic building block has been identified, it would be possible to introduce a *block insertion* mutation operator, which modifies an individual by randomly inserting a complete building-block expression into its genome, or a *block replacement* mutation operator, which replaces an expression with another that expresses a similar, yet different, variant of the same semantics. The rational of these two evolutionary operators is clear: In the case where a building block indeed performs a crucial task for the success of an individual, its ability to spread quickly in the population should increase significantly when using the block insertion operator, rather than simple crossovers. Replacing a block with a semantically similar block is more likely to introduce a non-lethal modification to the individual while still exploring the domain's solution space. These ideas are related to subroutine extraction in GP [42, 58, 59].

3. *Improving similarity computation using standard tools.* Standard code-equivalence detection techniques may be used in a pre-processing stage for similarity computation. Much research has been carried out, e.g., in axiomatic semantics [6, 22, 44], producing off-the-shelf tools that can be applied to assist in finding the special cases of phenotypically equivalent expressions.
4. *Context-based similarity.* As noted in Section 3.3, incorporating context information may considerably increase the quality of the similarity metric. A simple example of this idea is in the case of two nested `If` expressions that examine the same Boolean argument. In this case, it is possible to know in advance which of the sub-expressions of the internal `If` expression will be executed, and such information allows us to refine the similarity measuring procedure so that it will examine only the relevant "child" when compared with other expressions. In general, *context information* of some expression in a program may be thought of as information restricting the possible environments that may occur when an expression is executed, and is based in the case of GP on the values of the nodes appearing before the expression's node in a pre-order scan of the GP tree.
   Context information can be detailed, specifying exactly the set of all possible environments, or it can be more compact, and may imply some environments that cannot occur in the context or miss some possible environments. If, for example, it is known that in some context the value of the variable `x` is either 3, 10, 11, 13, 14, or 15, it is possible to encode this complete knowledge in the context information, or to compactly encode some approximate knowledge such as `10` $\leq$ `x` $\leq$ `15` (missing environments in which `x = 3` and including impossible environments in which `x = 12`). Then, a Boolean argument such as `(x < 30)` in an expression within this context is *equivalent* to `true`, and arguments such as `(x < 11)` may be assumed to be *most likely* (i.e., *similar to*) `false`. Such knowledge may affect the expression similarity measure, e.g., by weighting edit operations differently.
   Note that it would be interesting to conduct research on context information representation on its own, where maintaining detailed information can improve the similarity measure quality while exploiting computational resources (via intense storage usage and increased similarity computation complexity). In addition, note that under an *edit distance-like* approach, context information implies an asymmetric similarity measure: for two expressions $a$ and $b$ within two programs $A$ and $B$, respectively, it is possible that the cost of replacing $a$ with $b$ in $A$ would be different than the cost of replacing $b$ with $a$ in $B$, due to different context information for $a$ and $b$ in their corresponding programs.
5. *Other types of GP.* We believe our methodology can be used with other types of GP, e.g., linear GP [9]. The genotype of a linear GP individual is a sequence of instructions, and a natural approach would define a sub-expression to be a sub-sequence of continuous instructions. Edit distance was initially defined for (linear) strings and is used in modern bioinformatics research for comparing DNA and RNA sequences [24]. It is thus a natural choice for comparing linear GP sub-genomes. As in our work, the allowed edit operations and their costs would have to be functionality oriented and depend on the genome encoding language. For example, the deletion of an instruction irrelevant to program output might have a zero cost, while removing a conditional jump might incur, say, a 50% penalty due to the chance of jumping to another code section. Special care would have to be taken to ensure efficient algorithms [13, 48]. An additional possibility would be to invoke parsing tools over the linear genomes, obtain parse trees, and apply a tree edit distance-based anal-

ysis (as the one presented here) over these trees. Such approaches were previously successful when applied within related scientific domains, such as natural language processing [40] and RNA structure analysis [29].

## Online Material

Supplemental online material, containing a full list of the edit operations we used and their costs and penalties, and the evolved individuals from the sublinear runs, including those used for running G-PEA, can be found at `www.cs.bgu.ac.il/~sipper/gpea`.

**Acknowledgements** Kfir Wolfson and Shay Zakov were partially supported by the Frankel Center for Computer Science at Ben-Gurion University.

## References

1. Abbott, R., Guo, J., Parviz, B.: Guided genetic programming. In: The 2003 International Conference on Machine Learning; Models, Technologies and Applications (MLMTA'03). CSREA Press, las Vegas (2003)
2. Agapitos, A., Lucas, S.M.: Evolving efficient recursive sorting algorithms. In: Proceedings of the 2006 IEEE Congress on Evolutionary Computation, pp. 9227–9234. IEEE Press, Vancouver (2006)
3. Agapitos, A., Lucas, S.M.: Evolving modular recursive sorting algorithms. In: EuroGP, pp. 301–310 (2007)
4. Ahluwalia, M., Bull, L.: Coevolving functions in genetic programming. Journal of Systems Architecture **47**(7), 573–585 (2001)
5. Angeline, P.J.: A historical perspective on the evolution of executable structures. Fundamenta Informaticae **35**(1–4), 179–195 (1998)
6. Bellon, S., Koschke, R., Antoniol, G., Krinke, J., Merlo, E.: Comparison and evaluation of clone detection tools. IEEE Transactions on Software Engineering **33**(9), 577–591 (2007)
7. Bille, P.: A survey on tree edit distance and related problems. Theoretical Computer Science **337**(1-3), 217–239 (2005)
8. Boffelli, D., Nobrega, M., Rubin, E.: Comparative genomics at the vertebrate extremes. Nature Reviews Genetics **5**(6), 456–465 (2004)
9. Brameier, M., Banzhaf, W.: Linear genetic programming. Springer-Verlag, New York (2007)
10. Burke, E.K., Gustafson, S., Kendall, G.: Diversity in genetic programming: An analysis of measures and correlation with fitness. IEEE Transactions on Evolutionary Computation **8**(1), 47–62 (2004)
11. Ciesielski, V., Li, X.: Analysis of genetic programming runs. In: Mckay, R.I., Cho, S.B. (eds.) Proceedings of The Second Asian-Pacific Workshop on Genetic Programming. Cairns, Australia (2004)
12. Clergue, M., Collard, P., Tomassini, M., Vanneschi, L.: Fitness distance correlation and problem difficulty for genetic programming. In: Langdon, W.B., et al. (eds.) GECCO 2002: Proceedings of the Genetic and Evolutionary Computation Conference, pp. 724–732. Morgan Kaufmann Publishers, New York (2002)

13. Crochemore, M., Landau, G., Ziv-Ukelson, M.: A Subquadratic Sequence Alignment Algorithm for Unrestricted Scoring Matrices. SIAM Journal on Computing **32**, 1654 (2003)
14. Cummins, R., O'Riordan, C.: An analysis of the solution space for genetically programmed term-weighting schemes in information retrieval. In: Bell, D.A. (ed.) 17th Irish Artificial Intelligence and Cognitive Science Conference (AICS 2006). Queen's University, Belfast (2006)
15. Daida, J., Bertram, R., Polito, J., Stanhope, S.: Analysis of single-node (building) blocks in genetic programming. Advances in genetic programming **3**, 217–241 (1999)
16. De Jong, E.D., Watson, R.A., Pollack, J.B.: Reducing bloat and promoting diversity using multi-objective methods. In: Spector, L., et al. (eds.) Proceedings of the Genetic and Evolutionary Computation Conference (GECCO-2001), pp. 11–18. Morgan Kaufmann, San Francisco, California, USA (2001)
17. Doherty, D., O'Riordan, C.: A phenotypic analysis of GP-evolved team behaviours. In: Thierens, D., et al. (eds.) GECCO '07: Proceedings of the 9th annual conference on Genetic and evolutionary computation, vol. 2, pp. 1951–1958. ACM Press, London (2007)
18. Ekárt, A.: Shorter fitness preserving genetic programs. In: Fonlupt, C., et al. (eds.) Artificial Evolution. 4th European Conference, AE'99, Selected Papers, *LNCS*, vol. 1829, pp. 73–83. Dunkerque, France (2000)
19. Ekárt, A., Nemeth, S.Z.: A metric for genetic programs and fitness sharing. In: Poli, R., et al. (eds.) EuroGP'2000: Proceedings of Third European Conference on Genetic Programming, *LNCS*, vol. 1802, pp. 259–270. Springer-Verlag, Edinburgh (2000)
20. Ekárt, A., Németh, S.Z.: Maintaining the diversity of genetic programs. In: EuroGP '02: Proceedings of the 5th European Conference on Genetic Programming, pp. 162–171. Springer-Verlag, London, UK (2002)
21. Forrest, S., Nguyen, T., Weimer, W., Le Goues, C.: A genetic programming approach to automated software repair. In: GECCO '09: Proceedings of the 11th Annual conference on Genetic and evolutionary computation, pp. 947–954. ACM, New York, NY, USA (2009)
22. Gabel, M., Jiang, L., Su, Z.: Scalable detection of semantic clones. In: Proceedings of the 30th international conference on Software engineering, pp. 321–330. ACM New York, NY, USA (2008)
23. Goldberg, D.E.: Genetic Algorithms in Search, Optimization and Machine Learning. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA (1989)
24. Gusfield, D.: Algorithms on Strings, Trees, and Sequences: Computer Science and Computational Biology. Cambridge Univ Press (1997)
25. Harel, D.: Algorithmics: The Spirit of Computing. Second edn. Addison-Wesley Publishing Company, Readings, MA (1992)
26. Harman, M.: The current state and future of search based software engineering. In: FOSE '07: 2007 Future of Software Engineering, pp. 342–357. IEEE Computer Society, Washington, DC, USA (2007). DOI http://dx.doi.org/10.1109/FOSE.2007.29
27. Hauptman, A., Sipper, M.: Analyzing the intelligence of a genetically programmed chess player. In: Rothlauf, F. (ed.) Late breaking papers at Genetic and Evolutionary Computation Conference (GECCO'2005). Washington, D.C., USA (2005)
28. Hauptman, A., Sipper, M.: Emergence of complex strategies in the evolution of chess endgame players. Advances in Complex Systems **10**, 35–59 (2007)

29. Hofacker, I., Fontana, W., Stadler, P., Bonhoeffer, L., Tacker, M., Schuster, P.: Fast folding and comparison of RNA secondary structures. Monatshefte für Chemie/Chemical Monthly **125**(2), 167–188 (1994)
30. Jones, T., Forrest, S.: Fitness distance correlation as a measure of problem difficulty for genetic algorithms. In: Proceedings of the Sixth International Conference on Genetic Algorithms, pp. 184–192. Morgan Kaufmann (1995)
31. Joó, A., Neirotti, J.P.: Towards identifying salient patterns in genetic programming individuals. In: Raidl, G., et al. (eds.) GECCO '09: Proceedings of the 11th Annual conference on Genetic and evolutionary computation, pp. 1885–1886. ACM, Montreal (2009)
32. Kameya, Y., Kumagai, J., Kurata, Y.: Accelerating genetic programming by frequent subtree mining. In: Keijzer, M., et al. (eds.) GECCO '08: Proceedings of the 10th annual conference on Genetic and evolutionary computation, pp. 1203–1210. ACM, Atlanta, GA, USA (2008)
33. Keller, R.E., Banzhaf, W.: Explicit maintenance of genetic diversity on genospaces (1994). Unpublished Manuscript
34. Kinnear Jr., K.E.: Generality and difficulty in genetic programming: Evolving a sort. In: Proceedings of the 5th International Conference on Genetic Algorithms, pp. 287–294. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA (1993)
35. Kinnear, Jr., K.E.: Evolving a sort: Lessons in genetic programming. In: Proceedings of the 1993 International Conference on Neural Networks, vol. 2, pp. 881–888. IEEE Press, San Francisco, USA (1993)
36. Kinzett, D., Johnston, M., Zhang, M.: How online simplification affects building blocks in genetic programming. In: Raidl, G., et al. (eds.) GECCO '09: Proceedings of the 11th Annual conference on Genetic and evolutionary computation, pp. 979–986. ACM, Montreal (2009)
37. Kinzett, D., Zhang, M., Johnston, M.: Using numerical simplification to control bloat in genetic programming. In: Li, X., et al. (eds.) Proceedings of the 7th International Conference on Simulated Evolution And Learning (SEAL '08), *Lecture Notes in Computer Science*, vol. 5361, pp. 493–502. Springer, Melbourne, Australia (2008)
38. Kirshenbaum, E.: Iteration over vectors in genetic programming. Technical Report HPL-2001-327, HP Laboratories (2001)
39. Knuth, D.E.: Sorting and Searching, *The Art of Computer Programming*, vol. 3. Addison-Wesley, Reading, Massachusetts (1975)
40. Kouylekov, M., Magnini, B.: Tree edit distance for textual entailment. Recent Advances in Natural Language Processing IV: Selected Papers from RANLP (2005)
41. Koza, J.R.: Genetic Programming: On the Programming of Computers by Means of Natural Selection. MIT Press, Cambridge, MA, USA (1992)
42. Koza, J.R.: Genetic Programming II: Automatic Discovery of Reusable Programs. MIT Press, Cambridge, MA (1994)
43. Koza, J.R., Andre, D., Bennett III, F.H., Keane, M.: Genetic Programming III: Darwinian Invention and Problem Solving. Morgan Kaufman (1999)
44. Kuhn, A., Ducasse, S., Gírba, T.: Semantic clustering: Identifying topics in source code. Information and Software Technology **49**(3), 230–243 (2007)
45. Langdon, W.B., Banzhaf, W.: Repeated patterns in genetic programming. Natural Computing **7**(4), 589–613 (2008)
46. Lozano, A., Pinter, R.Y., Rokhlenko, O., Valiente, G., Ziv-Ukelson, M.: Seeded tree matching and planar tanglegram layout. In: Proc. 7th International Workshop on

Algorithms in Bioinformatics, LNCS 4645, pp. 98–110 (2007)
47. Luke, S., Panait, L.: A Java-based evolutionary computation research system. Online (2004). http://cs.gmu.edu/~eclab/projects/ecj
48. Masek, W., Paterson, M.: A faster algorithm computing string edit distances. Journal of Computer and System sciences **20**(1), 18–31 (1980)
49. McPhee, N., Ohs, B., Hutchison, T.: Semantic building blocks in genetic programming. Lecture Notes in Computer Science **4971**, 134 (2008)
50. Miklos, G., Rubin, G.: The Role of the Genome Project in Determining Gene Function: Insights from Model Organisms. Cell **86**, 521–529 (1996)
51. Montana, D.J.: Strongly typed genetic programming. Evolutionary Computation **3**(2), 199–230 (1995)
52. Nguyen, Q.U., O'Neill, M., Nguyen, X.H., McKay, B., Lopez, E.G.: Semantic similarity based crossover in GP: The case for real-valued function regression. In: Collet, P. (ed.) Evolution Artificielle, 9th International Conference, Lecture Notes in Computer Science, pp. 13–24 (2009)
53. O'Reilly, U.M.: Using a distance metric on genetic programs to understand genetic operators. In: IEEE International Conference on Systems, Man, and Cybernetics, Computational Cybernetics and Simulation, vol. 5, pp. 4092–4097. Orlando, Florida, USA (1997)
54. O'Reilly, U.M., Oppacher, F.: The troubling aspects of a building block hypothesis for genetic programming. In: Whitley, L.D., et al. (eds.) Foundations of Genetic Algorithms 3, pp. 73–88. Morgan Kaufmann, Estes Park, Colorado, USA (1994). Published 1995
55. 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, chap. 2, pp. 23–44. MIT Press, Cambridge, MA, USA (1996)
56. Pinter, R., Rokhlenko, O., Yeger-Lotem, E., Ziv-Ukelson, M.: Alignment of metabolic pathways. Bioinformatics **21**(16), 3401–3408 (2005)
57. Poli, R., Langdon, W.B., McPhee, N.F.: A field guide to genetic programming. Published via http://lulu.com and freely available at http://www.gp-field-guide.org.uk (2008)
58. Roberts, S.C., Howard, D., Koza, J.R.: Evolving modules in genetic programming by subtree encapsulation. In: Miller, J.F., et al. (eds.) Genetic Programming, Proceedings of EuroGP'2001, *LNCS*, vol. 2038, pp. 160–175. Springer-Verlag, Lake Como, Italy (2001)
59. Rosca, J.P., Ballard, D.H.: Discovery of subroutines in genetic programming. In: Angeline, P.J., et al. (eds.) Advances in Genetic Programming 2, chap. 9, pp. 177–202. MIT Press, Cambridge, MA, USA (1996)
60. Shapiro, B.A., Zhang, K.: Comparing multiple RNA secondary structures using tree comparisons. Computer Applications in Biosciences **6**(4), 309–318 (1990)
61. Shirakawa, S., Nagao, T.: Evolution of sorting algorithm using graph structured program evolution. In: SMC, pp. 1256–1261. IEEE (2007)
62. Sipser, M.: Introduction to the Theory of Computation. 2nd edn. Course Technology (2005)
63. Smart, W., Andreae, P., Zhang, M.: Empirical analysis of GP tree-fragments. In: Ebner, M., et al. (eds.) Proceedings of the 10th European Conference on Genetic Programming, *Lecture Notes in Computer Science*, vol. 4445, pp. 55–67. Springer, Valencia, Spain (2007)

64. Smith, M., Bull, L.: Improving the human readability of features constructed by genetic programming. In: Thierens, D., et al. (eds.) GECCO '07: Proceedings of the 9th annual conference on Genetic and evolutionary computation, vol. 2, pp. 1694–1701. ACM Press, London (2007)
65. Soule, T., Foster, J.A.: Effects of code growth and parsimony pressure on populations in genetic programming. Evolutionary Computation **6**(4), 293–309 (1998)
66. 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, pp. 1689–1696. ACM, New York, NY, USA (2005)
67. Stuart, J., Segal, E., Koller, D., Kim, S.: A gene-coexpression network for global discovery of conserved genetic modules. Science **302**(5643), 249 (2003)
68. Tackett, W.A.: Mining the genetic program. IEEE Expert **10**(3), 28–38 (1995)
69. Vanneschi, L., Tomassini, M.: Pros and cons of fitness distance correlation in genetic programming. In: Barry, A.M. (ed.) GECCO 2003: Proceedings of the Bird of a Feather Workshops, Genetic and Evolutionary Computation Conference, pp. 284–287. AAAI, Chicago (2003)
70. Wedge, D.C., Kell, D.B.: Rapid prediction of optimum population size in genetic programming using a novel genotype - fitness correlation. In: Keijzer, M., et al. (eds.) GECCO '08: Proceedings of the 10th annual conference on Genetic and evolutionary computation, pp. 1315–1322. ACM, Atlanta, GA, USA (2008)
71. Will, S., Reiche, K., Hofacker, I., Stadler, P., Backofen, R.: Inferring noncoding RNA families and classes by means of genome-scale structure-based clustering. PLoS Comput Biol **3**(4), e65 (2007)
72. Withall, M.S., Hinde, C.J., Stone, R.G.: An improved representation for evolving programs. Genetic Programming and Evolvable Machines **10**(1), 37–70 (2009)
73. Wolfson, K., Sipper, M.: Evolving efficient list search algorithms. In: Collet, P. (ed.) Evolution Artificielle, 9th International Conference, Lecture Notes in Computer Science (2009)
74. Wong, P., Zhang, M.: Algebraic simplification of GP programs during evolution. In: Keijzer, M., et al. (eds.) GECCO 2006: Proceedings of the 8th annual conference on Genetic and evolutionary computation, vol. 1, pp. 927–934. ACM Press, Seattle, Washington, USA (2006)
75. Woodward, J.: Evolving Turing complete representations. In: Sarker, R., et al. (eds.) Proceedings of the 2003 Congress on Evolutionary Computation CEC2003, pp. 830–837. IEEE Press, Canberra (2003)