Automatic Search-and-Replace from Examples with Coevolutionary Genetic Programming

Alberto Bartoli, Andrea De Lorenzo, Eric Medvet, and Fabiano Tarlao

***Abstract*—We describe the design and implementation of a system for executing search-and-replace text processing tasks automatically, based only on examples of the desired behavior. The examples consist of pairs describing the original string and the desired modified string. Their construction, thus, does not require any specific technical skill. The system constructs a solution to the specified task that can be used unchanged on popular existing software for text processing. The solution consists of a search pattern coupled with a replacement expression: the former is a regular expression which describes both the strings to be replaced and their portions to be reused in the latter, which describes how to build the modified strings. Our proposed system is internally based on Genetic Programming and implements a form of cooperative coevolution in which two separate populations are evolved independently, one for search patterns and the other for replacement expressions. We assess our proposal on six tasks of realistic complexity obtaining very good results, both in terms of absolute quality of the solutions and with respect to the challenging baselines considered.**

***Index Terms*—Find-and-replace, Programming by examples, Regular expressions, Diversity promotion**

I. Introduction

A common processing task which users perform on text documents consists in finding all the items of interest and replacing them with modified versions [1]. Many text editing programs tailored to different kinds of text include a functionality usually called *search-and-replace* (or find-and-replace) which allows users to perform this processing task automatically.

The required task could be very simple, such as replacing all the occurrences of a given string with another given string (e.g., replace `ie` with `i.e.`), or more complex, as changing the date format (e.g., from month-day-year to day-month-year) or removing thousands-separator from numbers. The user is required to specify the task by means of a search pattern and a replacement expression: the former is a *regular expression* which describes both the strings to be replaced and their portions to be reused in the latter, which describes how to build the modified strings. For instance, the date format change task mentioned above can be specified by means of the search pattern `(\d+)-(\d+)-(\d+)` and the replacement expression `$2-$1-$3`.

Popular software for, e.g., editing documents (e.g., Microsoft Word), developing code (e.g., Netbeans), authoring LaTeX documents (e.g., Overleaf web application), do support in their search-and-replace functionality the specification of the task by means of a search pattern and the corresponding replacement expression. Yet, many users cannot take advantage of this possibility because they lack the skills necessary to write regular expressions. While mastering regular expressions is an highly regarded achievement among developers [2], to the point that specific contests exist for proving this ability [3], many users still rely on web communities to find the regular expression that fits their needs, also for relatively simple tasks [4]. The specification of a search-and-replace task using a search pattern and a corresponding replacement expression is even harder, because those two parts have to be built accordingly, based on the knowledge of the formalism for describing the search pattern, of the formalism for describing the replacement expression and the task-specific domain.

In this paper, we propose a system for building *automatically* the specification of a search-and-replace task starting from a set of examples. The users just need to provide some tens of examples, each consisting of the text before and after the desired modification, and the system outputs a search-and-replace expression, i.e., a pair consisting of the regular expression defining the search pattern and the corresponding replacement expression. The user is not required to provide any other hint to the system as, e.g., marking within the text the substring to be replaced or giving an initial description of the task. The system generates a search-and-replace expression which is able to generalize beyond the provided examples, i.e., it is able to solve an under-specified task. The generated expressions can be reused in any compatible text editing software for later processing of text documents different than those used to provide the example and possibly very large[1].

The proposed system is based on Genetic Programming (GP), a popular Evolutionary Algorithm (EA) which has already been shown to be able to compete with humans in both efficiency and effectiveness for the specification of regular expressions from examples of the desired behavior [4]. In fact, we are not aware of any other automated approach with these properties. The problem of generating the search pattern and the replacement expression together is even harder, because the search space is larger and the two parts of the solution are dependent on each other. In our proposed approach, we employ many research results concerning Evolutionary Computation (EC) as a diversity promotion scheme and a cooperative coevolutionary framework, in order to obtain an overall effective and efficient tool for generating search-and-replace expressions.

Authors are with the Department of Engineering and Architecture (DIA), University of Trieste, 34127 Trieste, Italy.
E-mail: bartoli.alberto@units.it, andrea.delorenzo@units.it, emedvet@units.it, ftarlao@units.it

Manuscript received …

[1]We used the PCRE (Perl Compatible Regular Expression) syntax and the Java Language regular expression engine.

Our work constitutes a significant improvement over our earlier proposal [5], which, to the best of our knowledge, was the first system able to generate automatically both the search pattern and the replacement expression based only on examples. Our earlier proposal was based on GP and consisted of three steps: (i) a first evolutionary search aimed at constructing a preliminary search pattern; (ii) construction of a replacement expression of predefined structure, tailored to the provided examples; and, (iii) a further evolutionary search aimed at constructing the search pattern that, coupled with the replacement expression obtained at the previous step, would solve the user-specified task. In this work we provide instead a conceptually much simpler framework in which *both* the solution components—search pattern and replacement expression—are found through the *cooperative coevolution* of two separate populations, one for each component. The quality of the replacement expression, thus, does not depend on whether its predefined structure fits the specific task to be solved, but on the ability of the evolutionary search to find regions of the solution space that are suitable for that task. Furthermore, we carefully designed the evolutionary searches based on recent results in the area of evolutionary synthesis of text extractors from examples, which provided significant improvement over earlier proposals in this area [6]. In particular, we introduced the enforcement of a *phenotypic diversity promotion* scheme [7], we extended the number and variety of individuals in the initial population that are generated based on the available examples (rather than at random) and, most importantly, we introduced a significantly different fitness function. In our earlier work [5], we used a multi-objective fitness function based on two indexes: the amount of character-level errors and a quantification of the structural mismatch between the two components of a candidate solution, i.e., search pattern and replacement expression (please refer to the cited paper for full details). In this work we introduce a new fitness function composed of three objectives: the amount of character-level errors, a quantification of the ability to identify all the substrings in the examples that have to be modified (a form of character-level recall), a complexity measure of the candidate solution. This fitness definition is inspired by the one in [6], an important difference being that in the cited work complexity was quantified simply with the the length of the candidate solution whereas here we assign a different weight to each solution component.

We assessed our new proposal on six tasks of realistic complexity, four of which were included also in [5], and considered two significant baselines: our earlier GP proposal [5] and the FlashFill algorithm incorporated into Microsoft Excel [8]. The results demonstrate that our multi-objective cooperative coevolutionary GP (MOCCGP) proposal delivers very good results, both from the point of view of the absolute quality of the constructed solutions and with respect to the baselines. Furthermore, MOCCGP takes much shorter execution times than GP, often by one order of magnitude, which confirms that the proposed evolutionary framework is both more effective and more efficient than the previous one.

Our contribution may be summarized as follows. (1) Specification of a search-and-replace text processing problem from examples of the desired behavior. (2) Description of an automated method for solving this problem based on Multi-Objective Cooperative Coevolutionary Genetic Programming;[2] The method extends an earlier proposal based on Genetic Programming in terms of: usage of cooperative coevolution of two different populations, one for each component of a candidate solution; enforcement of a phenotypic diversity criterion; a novel fitness function composed of three objectives; usage of a fitness objective that quantifies complexity of a candidate solution based on the terminals and functions actually used. (3) Experimental assessment of the proposed method on six tasks of realistic complexity against two significant baselines.

The remainder of the paper is organized as follows. In Section II, we survey relevant previous research works. In Section III, we formally define the problem of the automatic generation of a search-and-replace expression from a set of examples. In Section IV, we describe in detail our proposal. In Section V, we discuss the experimental evaluation we performed, including the data and the baselines we used. Finally, in Section VI, we draw the conclusions.

II. Related work

In this section we focus on proposals concerning (a) the (possibly partial) automatization of string manipulation tasks which resemble search-and-replace, but which do not output a standard search-and-replace expression, (b) the automatic generation of regular expressions from examples, and (c) evolutionary approaches employing solutions for improving search effectiveness and/or efficiency which are related to those of our approach.

A. Automatic string manipulation

In general, the problem of synthesizing a string-to-string function consistent with a set of input-output examples is NP-complete [9]. However, instances of that problem have been solved in which the desired manipulation and/or the kind of input text were related to real-world applications, often in the context of text editing or spreadsheet software.

A system called LAPIS has been proposed in [10] which is able to perform simple string manipulation tasks (including some form of simple search-and-replace) specified using a pattern language defined previously by the same authors [11]. Since some skill is still required to use the language, LAPIS offers an assisted mode in which an initial pattern is inferred from a set of positive and negative examples. Differently from our work, the assisted mode addresses only the search portion of the search-and-replace task. A similar scheme for inferring a pattern from examples is used in [12]: the goal here is to guess multiple selections for simultaneous editing.

A method for assisting the user in executing a search-and-replace task is proposed in [13]. The authors consider a scenario where, in order to mitigate the difficulty of defining a search pattern, the users can work with imprecise patterns and then manual check each suggested match. The authors propose

[2]The code of our method is publicly available: https://github.com/MaLeLabTs/SearchAndReplaceGenerator.

to cluster the suggested matches so as to reduce the number of manual checks, since the user approve or reject the whole cluster instead of inspecting each single match.

More recently, the problem of automatic string manipulation has been tackled as a form of Programming by Examples (PBE): in the usual scheme, a string manipulation domain-specific language (DSL) is designed and an engine able to infer programs in that language which are consistent with user-provided examples is proposed. The work in [8] is the most significant of those approaches w.r.t. the present paper: indeed, we included it in our experimental evaluation (see Section V. The cited work proposes a DSL that supports restricted forms of regular expressions, conditionals, and loops and may represent common string manipulation tasks; the corresponding inference algorithm is interactive by design and takes fractions of seconds to infer the program in the considered string manipulation tasks. The algorithm, named FlashFill (FF), has been integrated into the Microsoft Excel spreadsheet software.

A similar approach is proposed in [14] for automatizing repetitive string processing tasks. The proposed DSL allows to perform tasks such as sorting and counting of occurrences; the associated PBE inference engine is able to extract clues of those constructs from the input examples.

We remark that none of the cited works output a search-and-replace expression which can be reused in any compatible text editing software, differently than our proposed approach.

B. Automatic generation of regular expressions

Regular expressions are a long-established way of specifying string patterns concisely. They are widely used in all sort of tasks, not only by programmers. However, authoring a regular expression is time-consuming and requires skills related to both the task at hand (e.g., knowing the format of an Italian VAT code) and the regular expression formalism: during the March 2016 alone, more than 140 000 questions about regular expressions have been posted on StackOverflow, a figure which shows that many users still struggle in writing their regular expressions [4]. It is not surprising, hence, that a wealth of approaches for the automatic generation of regular expressions have been proposed, some based on some form of evolutionary computation (EC) [15], [16], [17], [18], [19], [20], [21], [3], [22], [6], [23] and some not [24], [25], [26], [27], [28]. As a further classification, some of the approaches consider the problem of generating a regular expression able to match exactly the substring labeled in the example, others the problem of generating a regular expression able to match a non-empty substring in a subset of the examples and nothing on the remaining examples. The two kinds of problems are related to *extraction* and *flagging*, the former being, in general, harder.

Concerning non EC-based approaches, in [26] the user is required to provide a set of examples and an initial regular expression: the algorithm then applies successive transformations until it reaches a local optimum in terms of precision and recall. The system proposed in [27] works similarly but the authors focus on noisy data. The method proposed in [28] does not rely on an initial regular expression: instead, it identifies relevant patterns in the set of examples and then combines the most promising pattern into a single regular expression. The proposal is evaluated on several business-related text extraction tasks, e.g., phone numbers and invoice numbers. The system proposed in [24] is tailored to data mining within criminal justice information systems: it starts from a single example and produces a reduced form of regular expression exploiting the operator interventions during the learning process, which is hence not fully automatic. A similar scheme for regular expression generation which involves the human operator is presented in [25]: here an active learning algorithm is proposed which starts from a single example and then requires an external operator to respond to membership queries about candidate expressions.

EC has been used for generating regular expressions in the last three decades and in many variants: Genetic Algorithms (GA) [19], [20], Grammatical Evolution (GE) [18], and Genetic Programming (GP) [15], [16], [17], [21], [3], [22], [6], [23]. None of these approaches considered the problem of synthesizing both a regular expression *and* a replacement expression, though. Furthermore, the ability to solve practically relevant problems was demonstrated only by the most recent GP-based approaches (see [6] for a detailed comparison). In this respect, the seminal work is [21] on which later sophisticated proposals are based that address the flagging problem [3], the extraction problem [6], and the active learning extraction problem (i.e., the scenario where the examples are provided by the user interactively based on system-generated queries) [23]. In the present paper we built on those proposals and customized many system components to the specific case of the generation of search-and-replace expressions. In particular, it is worth noting that simply using as a search pattern of a search-and-replace expression a regular expression able to match the substrings to be replaced is not enough, because it will lack the *capturing groups* needed to perform the replacement (see Section IV).

C. Evolutionary Computation perspective

Our proposal is a form of multi-objective Cooperative Coevolutionary GP. Cooperative Coevolutionary Evolutionary Algorithms (CCEA) have been introduced in [29] and are a kind of EA where the individuals in the populations are not assessed individually, but are grouped together to form a complete solution for the problem. The effectiveness of the individuals derives from the effectiveness of the complete solution they take part in. The search-and-replace problem can be naturally partitioned in two sub-problems, generating the search pattern and generating the replace expression: for this reason, it fits a multi-population CCEA, where the complete solution is made of two sub-parts taken from two distinct populations.

The evolution of solutions for multi-objective problems through many cooperative populations have been proposed in [30] for combinatorial optimization: the cited work also performs some Pareto local search to further improve solution quality. Local search is indeed a promising method to increase efficiency of EAs, in particular when tackling problems where the fitness evaluation is computationally expensive or where strict constraints hold on the execution time as in, e.g., the dynamic job shop scheduling problem [31]. In our work, we do not perform local search steps: instead, we adopt domain-based heuristics in the building of the initial population from the examples.

A different form of splitting of the main problem in sub-problems (separate-and-conquer) is presented in [22], for the learning of regular expressions able to capture entities with different syntax patterns. In the cited work, as in our proposal, the diversity in the population is promoted. Diversity promotion schemes have been indeed widely used to avoid premature convergence [7]: common approaches consist in self-adapting the probabilities of genetic operators (e.g., in [32] for the mining of rules for subgroup discovery) or dynamically vary the size of the population (e.g., in [33] as a variant to the Differential Evolution EA). In our case, similarly to [22], we promote diversity by simply prohibiting the presence of duplicates (i.e., individual with the same phenotype) in the populations.

III. Problem Statement

A *search-and-replace expression* is a pair $\langle s, r\rangle$ of strings composed of a *search pattern* s, which is a regular expression, and a *replacement expression* r. The pair $\langle s, r\rangle$ completely describes a text replacement task: in an input text t, every substring of t which matches s is replaced by a different substring as described by the replacement expression r.

The search pattern s may contain capturing groups, a *capturing group* being a substring s that is also a regular expression and is enclosed between round parentheses. When a regular expression s containing a capturing group matches a string, the capturing group matches a substring of the matched string.

The replacement expression r describes the string that replaces a string matched by s. Available constructs for r include *back-references* to the substrings matched by the capturing groups in s. The syntax[3] for back-references is $\$n$, where n is the index of occurrence of the capturing group in s: `$0` indicates the entire string matched by s, `$1` indicates the substring matched by the first capturing group in s, and so on. For example, a search-and-replace expression which can be used to change the date format from month-date-year to day-month-year may be composed of the search pattern s = `(\d+)-(\d+)-(\d+)`, which includes three capturing groups, and the replacement expression r = `$2-$1-$3`.

We consider the problem of constructing a search-and-replace expression *automatically* from a user-provided *learning set* of *examples* E describing a text replacement task. Examples in E are pairs of strings $\langle t, t'\rangle$, where t is a string to be replaced by t'. An example in which $t' = t$ is called a *negative* example and an example in which $t' \neq t$ is called a *positive* example. An embodiment of our framework could include a GUI in which the user loads a text, selects portions that do not have to be modified (negative examples) and portions that have to be modified, indicating the desired modification for each portion (positive examples).

Intuitively, the problem consists in learning a search-and-replace expression $\langle s, r\rangle$ whose behaviour is consistent with the provided examples—$\langle s, r\rangle$ transforms t into t' for each of the examples in E. Furthermore, $\langle s, r\rangle$ should capture the underlying pattern describing the replace operation, thereby *generalizing* beyond the provided examples. In other words, the examples constitute an incomplete specification of the behaviour of an ideal and unknown search-and-replace expression $\langle s^\star, r^\star\rangle$ and the learning algorithm should infer an expression with the same behaviour as $\langle s^\star, r^\star\rangle$ not only on the learning set but also on unseen text.

IV. Our approach

A. Overview

The proposed method is a form of Multi-Objective (MO) EA consisting of a multi-population *Cooperative Coevolutionary* (CC) [34] *tree-based Genetic Programming* (GP) [35] with several optimizations. We provide an overview of our proposal in this section and full details in the next sections.

Two fixed-size populations S and R are evolved in which individuals are search patterns and replace expressions, respectively, internally represented as trees (see Section IV-C). The function and terminal sets for the trees of the two populations include predefined elements and a number of elements determined based on the examples in the user-provided learning set E (see Section IV-D).

The initial composition of the two populations is determined based on the examples in E (see Section IV-D). Then, the populations are evolved iteratively for a specified number of generations or until a termination criterion is met (see Section IV-F). The populations S and R at a given generation are constructed from the corresponding populations at the previous generation, by applying the classic tree-based genetic operators (mutation and crossover [35]) to individuals chosen with a reproduction selection criterion. Furthermore, the best individuals of a population are included unchanged in the population at the next generation (a form of *elitism* [36]).

In each population, no duplicate individuals are allowed: whenever a new individual s (r) is generated (upon initialization or application of a genetic operator), if another individual $s' \in S$ ($r' \in R$) exists such that $s = s'$ ($r = r'$)—i.e., the two strings are the same—the new individual is discarded, parent(s) selection is repeated, and a new one is generated.

The two selection criteria (reproduction and survival) operate based on the fitness of the individuals. In order to compute the fitness, individuals of the two populations are paired using a *pairing procedure* (see Section IV-E) which outputs a number of search-and-replace expressions $\langle s, r\rangle$ out of the individuals in S and R: each search pattern $s \in S$ and replacement expression $r \in R$ may be a part of zero or more search-and-replace expressions. Three numerical indexes (see Section IV-E) are computed for each search-and-replace expression: the fitness of a search pattern s is given by the three indexes of the best search-and-replace expression among the

[3]Some regular expression engines (e.g., Python, .NET, Ruby) use the $\backslash n$ notation, instead of the $\$n$ notation used, e.g., in Java, JavaScript, PHP.

ones including s; the same for r. Fitness of individuals (triplets of indexes) are compared based on Pareto-dominance and, in case of tie, lexicographic ordering of fitness components (see Section IV-E). If no search-and-replace expression exists for a given individual s or r, the corresponding fitness is set to the worst values (see Section IV-E).

The full evolutionary search procedure above is repeated a number of times by varying the initial random seed. A single search-and-replace expression is chosen as the final solution among the best search-and-replace expressions in the final populations of the repetitions (see Section IV-F).

Summarizing, the proposed method is a form of MOCCGP with optimizations concerning (a) the construction of terminal sets from the examples, (b) the construction of the initial populations from the examples, (c) the phenotypic diversity promotion scheme, (d) and the repetition of the evolutionary search.

B. Definitions and notation

We here introduce the terms and notations related to string manipulation and define constructs that are necessary in the following.

A *substring* of a string t is a string of consecutive characters in t identified by the indexes of the first (inclusive) and last (exclusive) characters. We denote by $t_{a,b}$ the substring of t where a is the index in t of the first character of $t_{a,b}$ and $b-1$ is the index in t of the last character of $t_{a,b}$. For example, given t = I␣love␣the␣vegetables, $t_{2,6}$ = love.

The *length* of a string t, denoted $\ell(t)$, is the number of characters composing t; for a substring $t_{a,b}$, $\ell(t_{a,b}) = b - a$.

Two substrings $t_{a,b}, t_{a',b'}$ of a string t *overlap*, denoted $t_{a,b} \top t_{a',b'}$, if and only if $(b > a') \wedge (b' > a)$.

The *intersection* of two substrings $t_{a,b}, t_{a',b'}$ of a string t which overlap, denoted $t_{a,b} \sqcap t_{a',b'}$, is the substring starting from the largest of the two starting indexes a, a' and ending at the lowest of the ending indexes b, b': i.e., $t_{a,b} \sqcap t_{a',b'} = t_{\max(a,a'),\min(b,b')}$. The intersection of two substrings which do not overlap is the empty string $\varnothing$.

Two substrings $t_{a,b}, t'_{c,d}$ are *equivalent*, denoted $t_{a,b} \equiv t'_{c,d}$, if the text in the substrings is the same, regardless of the indexes. For example, given t = this␣is␣false and t' = take␣this, $t_{0,4} \equiv t'_{5,9}$.

The set $\mathcal{C}_{t,t'}$ of *maximal common substrings* of two strings t, t' is a set composed of all the non-empty substrings of t such that (i) they have an equivalent substring in t', and (ii) they do not overlap other longer substrings of t having an equivalent substring in t', and (iii) they do not overlap other substrings of t of equal length and lower starting index that have an equivalent substring in t'. For example, given t = the␣car and t' = they␣are, $\mathcal{C}_{t,t'} = \{t_{0,3}, t_{4,5}, t_{5,7}\} = \{$the, ␣, ar$\}$. Formally, $\mathcal{C}_{t,t'} = \{t_{a,b} : (\exists t'_{c,d}, t'_{c,d} \equiv t_{a,b}) \wedge (\forall t_{e,f} : \exists t'_{g,h}, t'_{g,h} \equiv t_{e,f}, t_{e,f} \not\top t_{a,b} \vee \ell(t_{e,f}) < \ell(t_{a,b}) \vee e > a)\}$.

Given two strings t and t', we denote by $\alpha_{t,t'}$ the longest substring of t starting at index 0 for which an equivalent substring of t' exists which starts at index 0, i.e., $\alpha_{t,t'} = \arg\max_{t_{0,h} \in T} \ell(t_{0,h})$, with $T = \{t_{0,h} : \exists t'_{0,h}, t_{0,h} \equiv t'_{0,h}\}$. Similarly, we denote by $\omega_{t,t'}$ the longest substring of t ending at index $\ell(t)$ for which an equivalent substring of t' exists which ends at index $\ell(t')$, i.e., $\omega_{t,t'} = \arg\max_{t_{\ell(t)-h,\ell(t)} \in T} \ell(t_{\ell(t)-h,\ell(t)})$, with $T = \{t_{\ell(t)-h,\ell(t)} : \exists t'_{\ell(t')-h,\ell(t')}, t_{\ell(t)-h,\ell(t)} \equiv t'_{\ell(t')-h,\ell(t')}\}$. The *string difference* between two strings t, t', denoted $\delta_{t,t'}$, is the substring of t such that the concatenation $\alpha_{t,t'} \oplus \delta_{t,t'} \oplus \omega_{t,t'}$ is equivalent to t. For example, given t = the␣long␣tail and t' = the␣short␣tail, $\alpha_{t,t'} = t_{0,4}$ = the␣, $\omega_{t,t'} = t_{8,13}$ = ␣tail, $\delta_{t,t'} = t_{4,8}$ = long, and $\delta_{t',t} = t'_{4,9}$ = short. Note that, for any two strings t, t', $\alpha_{t,t'} \equiv \alpha_{t',t}$ and $\omega_{t,t'} \equiv \omega_{t',t}$, whereas, in general, $\delta_{t,t'} \not\equiv \delta_{t',t}$.

C. Individual representation

Individuals of the two populations S and R are search patterns and replacement expressions, respectively, internally represented as trees. Elements of the respective terminal sets $\mathcal{T}_S, \mathcal{T}_R$ and function sets $\mathcal{F}_S, \mathcal{F}_R$ are strings, as follows (we assume the reader has some familiarity with regular expressions [2]).

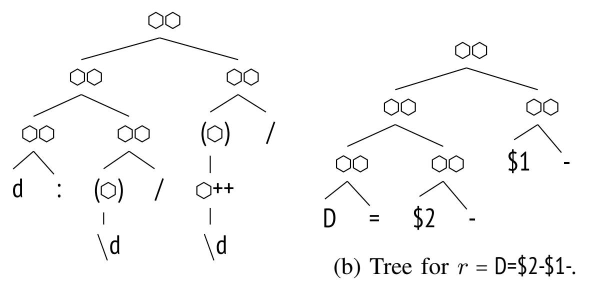
The terminal set $\mathcal{T}_S$ for S consists of elements based on the examples in the user-provided learning set E (see Section IV-D) and of the following strings: ranges a-z and A-Z, character classes \w or \d, and constants—i.e., digits 0, …, 9 and the (possibly escaped) characters \., :, ,, ;, _, =, ", ', \, /, \?, \!, \{, \}, \(, \), \[, \], <, >, @, #, ␣. The function set $\mathcal{F}_S$ for S consists of the following strings: concatenator ⬡⬡, character class operators [⬡] and [^⬡], capturing group (⬡) and *non-capturing group*[4] (?:⬡) operators, lookarounds (?<=⬡), (?<!⬡), (?=⬡), (?!⬡), and possessive quantifiers ⬡*+, ⬡++, ⬡?+, ⬡{⬡,⬡}+.

The terminal set $\mathcal{T}_R$ for R consists of elements based on the examples in the user-provided learning set E (see Section IV-D) and of the 10 strings representing the back-references (see Section III): $0, …, $9. The function set $\mathcal{F}_R$ for R consists only of the concatenator string ⬡⬡.

An individual s (r) is obtained from the corresponding tree by means of the following parsing procedure. Let a *string transformation* of a tree be the string obtained from the root node of the tree after replacing every ith occurrence of the character ⬡ with the string transformation of the ith child of the root node. Then s and r are the string transformations of the corresponding trees. Figure 1 shows two possible internal representations (trees) for a search pattern s = d:(\d)/(\d++)/ and a replacement expression r = D=$2-$1-.

Whenever, upon the application of a genetic operator, a tree is generated whose string transformation is not a syntactically valid regular expression (for S) or replacement expression (for R), the tree is discarded and a new one is generated, i.e., parent(s) selection is repeated and the genetic operator is applied again until obtaining an individual whose phenotype (i.e., string transformation) is different from the phenotype of all the individuals that are in the population already. In this respect, our EA adopts a phenotypic *diversity promotion* scheme [7]. Moreover, whenever a tree is generated whose

[4]A non-capturing group is a group which cannot be referenced in the replacement expression.



(a) Tree for s = d:(\d)/(\d++)/.

(b) Tree for r = D=$2-$1-.

Figure 1: The internal representations (trees) of a search pattern (left) and of a replacement expression (right).

depth exceeds a predefined maximum depth n_{depth}, the tree is discarded and a new one is generated.

D. Terminal set and population initialization

The terminal set and the initial composition of the two populations contain elements tailored to the user-provided learning set E of examples. The terminal set is built so as to include elements that may be useful building blocks for working on the provided examples. The initial population is built in order to provide a sort of good starting point and useful genetic material for the search. Both design choices have proven to increase efficiency and effectiveness for the evolutionary generation of regular expressions from examples [6], [23] and they work as follows.

Let $\mathcal{T}_S^E$ denote the subset of $\mathcal{T}_S$ containing the elements tailored to E. We construct $\mathcal{T}_S^E$ as follows. Let E_T be a subset of the examples in E (see Section IV-F). Initially, $\mathcal{T}_S^E = \varnothing$. Then, for each example $\langle t, t' \rangle \in E_T$, (i) the substring $\delta_{t,t'}^{\text{ext}}$ of t is built by extending by (up to) $\frac{1}{2}\ell(\delta_{t,t'})$ characters on both sides the string difference $\delta_{t,t'}$ between t and t' and (ii) each character in $\delta_{t,t'}^{\text{ext}}$ is inserted in $\mathcal{T}_S^E$.

Let $\mathcal{T}_R^E$ denote the subset of $\mathcal{T}_R$ containing the elements tailored to E. $\mathcal{T}_R^E$ is constructed with the same procedure as the one for constructing $\mathcal{T}_S$, except that $\delta_{t',t}$ is used instead of $\delta_{t,t'}^{\text{ext}}$.

Let S_0 denote the initial population of search patterns S and let n_{pop}^S be the size of S. We include up to $0.9n_{\text{pop}}^S$ individuals in S_0 from the examples in E_T as follows. For each example $\langle t, t' \rangle \in E_T$, the two string differences $\delta_{t,t'}$ and $\delta_{t',t}$ are determined and the set $\mathcal{C}_{\delta_{t,t'},\delta_{t',t}}$ of their maximal common substrings is built. Then, the following 16 regular expressions are built:

- $s_1 = \delta_{t,t'}$;
- $s_2 = s_1$ where each letter in s_1 is replaced by \w and each digit is replaced by \d;
- $s_3 = s_1$ where each occurrence in s_1 of a string c of $\mathcal{C}_{\delta_{t,t'},\delta_{t',t}}$ is replaced by (c) (i.e., it is enclosed in a capturing group);
- $s_4 = s_3$ where each letter in s_3 is replaced by \w and each digit is replaced by \d;
- $s_5, \dots, s_8$ built from $s_1, \dots, s_4$ by replacing each substring composed of two or more repetitions of the same character or class c by c++ (i.e., it is grouped using the possessive quantifier ++);
- $s_9, \dots, s_{16}$ built in the same way of $s_1, \dots, s_8$ but starting from $s_9 = \delta_{t,t'}^{\text{ext}}$ instead of $s_1 = \delta_{t,t'}$.

$$t = \overbrace{\texttt{we'll␣meet␣on␣}}^{\alpha_{t,t'}}\overbrace{\texttt{11/3/}}^{\delta_{t,t'}}\overbrace{\texttt{2014␣at␣8:00am}}^{\omega_{t,t'}}$$

$$t' = \underbrace{\texttt{we'll␣meet␣on␣}}_{\alpha_{t',t}}\underbrace{\texttt{3-11-}}_{\delta_{t',t}}\underbrace{\texttt{2014␣at␣8:00am}}_{\omega_{t',t}}$$

$$\mathcal{C}_{\delta_{t,t'},\delta_{t',t}} = \{11, 3\}$$

s_1 = 11/3/ $\qquad$ s_5 = 1++/3/

s_2 = \d\d/\d/ $\qquad$ s_6 = \d++/\d/

s_3 = (11)/(3)/ $\qquad$ s_7 = (1++)/(3)/

s_4 = (\d\d)/(\d)/) $\qquad$ s_8 = (\d++)/(\d)/

r = $2-$1-

Figure 2: Examples of the search patterns and replacement expression built for a single example $\langle t, t' \rangle$ in the population initialization—search patterns $s_9, \dots, s_{16}$ are not shown for the sake of brevity.

In case the above procedure constructs more than $0.9n_{\text{pop}}^S$ individuals, we select $0.9n_{\text{pop}}^S$ individuals at random with uniform distribution across all the individuals and include in S_0 only the selected individuals. We recall that duplicate individuals are not allowed in the population: in particular, if a search pattern s is built from an example and another $s' = s$ exists in the initial population S_0, s is not inserted in S_0 (the comparison $s' = s$ is made on the string transformations of the two search patterns, as described in Section IV-C). The actual tree for a search pattern s is built by parsing s in order to obtain a balanced tree (best effort, we omit the details for the sake of brevity; our evolutionary search framework limits the maximum depth of trees representing individuals, thus using balanced trees allows representing a richer variety of individuals). The remaining individuals required for completing S_0 ($0.1n_{\text{pop}}^S$ or more) are built randomly using the Ramped Half-and-Half method with a maximum depth of n_{depth}^0 [35]

Let R_0 denote the initial population of replacement expressions R and let n_{pop}^R be the size of R. We include up to $0.9n_{\text{pop}}^R$ individuals in R_0 from the examples in E_T as follows. For each example $\langle t, t' \rangle \in E_T$, the two string differences $\delta_{t,t'}$ and $\delta_{t',t}$ are determined, the set $\mathcal{C}_{\delta_{t,t'},\delta_{t',t}}$ of their maximal common substrings is built, and $\mathcal{C}_{\delta_{t,t'},\delta_{t',t}}$ is sorted according to the decreasing length of its elements. Then, a single replace expression r (i) r is initially set to $r = \delta_{t',t}$; then, (ii) for each string c in $\mathcal{C}_{\delta_{t,t'},\delta_{t',t}}$, the occurrence of c in r is replaced by $\$i$, where i is the (1-based) index of c in $\mathcal{C}_{\delta_{t,t'},\delta_{t',t}}$. The remaining individuals required for completing R_0 ($0.1n_{\text{pop}}^R$ or more) are built randomly using the Ramped Half-and-Half method with a maximum depth of n_{depth}^0 [35].

Figure 2 shows an example of the individuals constructed from a single example $\langle t, t' \rangle$.

E. Fitness

The fitness of an individual (i.e., a search pattern s or a replacement expression r) is given by three numerical indexes computed on the best search-and-replace expression $\langle s, r\rangle$ among the ones built by the *pairing procedure*, described below, that the individual belongs to. In case an individual s or r does not belong to any search-and-replace expression, the worst possible fitness $(+\infty, +\infty, +\infty)$ is assigned to it.

The pairing procedure takes in input the two populations S and R of search patterns and replacement expressions, respectively, and outputs a set P of search-and-replace expressions. Starting from $P = \emptyset$, for each search pattern $s \in S$, all pairs $\langle s, r\rangle$ such that the number of backreferences in r is not greater than the number of groups in s are inserted into P.

The rationale for this pairing procedure is to associate each search pattern s with a replacement expression r which can, potentially, exploit all the capturing groups of s. Note that $|P| \leq |S|$: in particular, $|P| < |S|$ if one or more search pattern exist such that no replacement expressions with the proper $\text{groups}_R(r)$ exist in R. Moreover, in general not all the replacement expressions are members of pairs of P and some replacement expression may be included in more than one pair in P.

The three fitness indexes measure (i) the quality of the replacements resulting by the application of $\langle s, r\rangle$ to the user-provided examples, (ii) the recall, at the character level, of the search pattern on the examples, and (iii) the complexity of the search pattern s and of the replacement expression r.

Let E_T be a subset of the examples in E (see Section IV-F). In detail, the fitness $\vec{f}(\langle s, r\rangle)$ is a triplet $(f_{\text{dist}}(\langle s, r\rangle; E_T), f_{\neg\text{rec}}(s; E_T), f_{\text{comp}}(\langle s, r\rangle)) \in \mathbb{R}^3$:

$$f_{\text{dist}}(\langle s, r\rangle; E_T) = \sum_{\langle t, t'\rangle \in E_T} \frac{d(t', \text{rep}(t, \langle s, r\rangle))}{\ell(t')} \tag{1}$$

$$f_{\neg\text{rec}}(s; E_T) = \sum_{\langle t, t'\rangle \in E_T} 1 - \frac{\ell(\delta_{t,t'} \sqcap \text{sub}(t, s))}{\ell(\delta_{t,t'})} \tag{2}$$

$$f_{\text{comp}}(\langle s, r\rangle) = \text{comp}_S(s) + \text{comp}_R(r) \tag{3}$$

where $d(t_1, t_2)$ is the Levenshtein distance between the strings t_1 and t_2, $\text{rep}(t, \langle s, r\rangle))$ is the string resulting from the application of the search-and-replace expression $\langle s, r\rangle$ to t, $\text{sub}(t, s)$ is the first substring of t matched by s, and $\text{comp}_S(s)$, $\text{comp}_R(r)$ are two complexity measures which operate at the level of the internal representations (i.e., trees) of s and r, respectively, as follows.

The complexity $\text{comp}_S(s)$ of a tree representing a search pattern s is given by the sum of the complexity of its nodes, which is 0.8 for character classes `\d` and `\w`, 0.6 for quantifiers `○*+`, `○++`, `○?+`, `○{○,○}+`, 0 for concatenator `○○`, and 1 for any other node. The complexity $\text{comp}_R(r)$ of a tree representing a replacement expression r is given by the sum of the complexity of its nodes, which is 0.6 for back-references `$0`, ..., `$9`, and 1 for any other node.

For all the three components of the fitness, the lower, the better. In particular, $f_{\text{dist}}(\langle s, r\rangle; E_T)$ is, for each example $\langle t, t'\rangle \in E_T$, equals to 0 if the result $\text{rep}(t, \langle s, r\rangle))$ of the application of the search-and-replace expression $\langle s, r\rangle$ to t is exactly t'; $f_{\neg\text{rec}}(s; E_T)$ is 0 if the first match $\text{sub}(t, s)$ of s in t includes $\delta_{t,t'}$; finally, $f_{\text{comp}}(\langle s, r\rangle)$ is closer to 0 if r and s are short and make use of constructs which allow to generalize.

The first component of the fitness is a straightforward measure of the quality of the search-and-replace task which can be performed by using $\langle s, r\rangle$. The rationale for the second component is to help the search at the beginning of the evolution, i.e., when the first component cannot discriminate between bad and "almost good" search-and-replace expressions; in later stages of the evolution, the second component becomes less relevant: it can be seen that $f_{\text{dist}}(\langle s, r\rangle; E_T) = 0$ implies $f_{\neg\text{rec}}(s; E_T) = 0$, whereas the opposite is, in general, not true. Note that the precision is not taken into account explicitly: a search pattern with a non-perfect precision would match also characters which are not necessary, yet not even harmful, for the replacement task. The rationale for the third component is twofold: on one hand, penalizing large trees is a way for addressing *bloat*, i.e., the growth of solutions without any improvements of the fitness [37]; on the other hand, assigning a lighter weight to nodes like classes, quantifiers, and back-references is a way for favoring the generalization of search-and-replace expressions (e.g., `\d\d-\d\d\d\d` is more able to generalize than `\d\d-20\d\d`). The latter is an improvement over most recent proposals for GP-based regular expression generation from examples which only take into account the length of regular expression [23], [6].

Pairs of fitness values are compared based on Pareto-dominance: a fitness $\vec{f}$ dominates a fitness $\vec{f}'$ if and only if (a) each ith component f_i of $\vec{f}$ is lower than or equal to the corresponding ith component f'_i of $\vec{f}'$, and (b) at least one component f_j is strictly lower than the corresponding f'_j. Sets of fitness values are sorted based on Pareto frontiers with a lexicographic tie-breaking criterion: given a set F of fitness values, the subset F_1 (i.e., the first *Pareto frontier*) is built by including all the non-dominated fitness values of F; then, the subsets $F_2, F_3, \ldots$ are iteratively built by including, in each F_i, all the non-dominated fitness values in $F \setminus \bigcup_{j<i} F_j$. Fitness values are sorted based on their Pareto frontier (i.e., the index i of the subset F_i they belong to). In case of tie (i.e., fitness values in the same F_i), the fitness with the lower first component comes first; in case of further tie, the second or third component is used.

F. Evolutionary search

An evolutionary search works as follows (see also Fig. 3):

1) The user-provided learning set E of examples is split in a *training set* E_T and a *validation set* E_V; E_T and E_V are the same size (if the cardinality of E is odd then E_T has one more element than E_V) and are built from E with uniform sampling without replacement by ensuring that (a) the difference in the number of positive examples in E_T and E_V is at most 1, (b) the difference in the number of negative examples in E_T and E_V is at most 1, and (c) $E_T \cap E_V = \emptyset$. In other words, the positive/negative ratio of the learning set is maintained in the training and validation sets.

2) Set E_T is used for constructing the example-based subsets $\mathcal{T}_S^E, \mathcal{T}_R^E$ of the terminal sets for the two populations, as detailed in Section IV-D.
3) The initial populations S, R, composed of respectively n_{pop}^S and n_{pop}^R individuals, are built from E_T as detailed in Section IV-D. A set $\hat{P} = \varnothing$ of the best search-and-replace expressions, initially empty, is built.
4) A set P of search-and-replace expressions is built from S, R with the pairing procedure detailed in Section IV-E and the fitness of the elements in P are computed. The set $\hat{P}$ is populated by including the best (according to the sorting of the fitness of its elements, as detailed in Section IV-E) $n_{\hat{P}}$ search-and-replace expressions of $P \cup \hat{P}$. Finally, two populations $S' = \{s, \langle s, r\rangle \in P \cup \hat{P}\}$ and $R' = \{r, \langle s, r\rangle \in P \cup \hat{P}\}$ of search patterns and replacement expressions are built by considering the respective elements of the pairs in $P \cup \hat{P}$.
5) The new population S is constructed that consists of the best $p_{\text{elitism}} n_{\text{pop}}^S$ individuals of S', $p_{\text{cross}} n_{\text{pop}}^S$ new individuals generated by applying the crossover operator on pairs of individuals in S', $p_{\text{mut}} n_{\text{pop}}^S$ new individuals generated by applying the mutation operator on individuals in S', $p_{\text{rnd}} n_{\text{pop}}^S$ individuals randomly chosen in S', and $p_{\text{new}} n_{\text{pop}}^S$ new individuals built randomly using the Ramped Half-and-Half method with a maximum depth of n_{detpth}^0; a tournament selection criterion of size n_{tour} is used for choosing individuals for applying the genetic operators—comparison and sorting criteria are the ones detailed in Section IV-E applied to the fitness values of the corresponding best pairs in $P \cup \hat{P}$ (or the worst fitness value if no corresponding pair exists in $P \cup \hat{P}$). The new population R is populated in the same way from R' using n_{pop}^R instead of n_{pop}^S.

Steps 4 and 5 constitute a generation of the evolution. The search is completed when at least one of the following criteria is met: (a) a predefined number n_{gen} of generations has been executed, or (b) the best search-and-replacement expression $\hat{p} \in \hat{P}$ is such that $f_{\text{dist}}(\hat{p}; E_T) = 0$ and (all the three components of) the fitness of the best expression in $\hat{P}$ in the last n_{imp} generations have never changed. The outcome of the evolutionary search is the best individual $\hat{p} = \langle \hat{s}, \hat{r}\rangle$ in $\hat{P}$ at the last generation. The rationale for the termination criterion b and, in particular, for the choice of continuing the evolution for a while even after a search-and-replace expression with perfect $f_{\text{dist}}(\hat{p}; E_T)$ has been found, is to allow for further improvements in the third component of the fitness (recall that $f_{\text{dist}}(\hat{p}; E_T) = 0$ implies $f_{\neg\text{rec}}(\hat{s}; E_T) = 0$), i.e., it is aimed at obtaining simpler search-and-replace expressions which are also abler to generalize (see Section IV-E).

The search procedure described above is repeated n_{searches} times starting with different random seeds, in order to improve generalization and mitigate possible unfortunate stochastic conditions.

The final search-and-replace expression $\langle s, r\rangle$ resulting from the entire process is the one, among the n_{searches} pairs $\langle \hat{s}, \hat{r}\rangle$ resulting from the repetitions, which minimizes $f_{\text{dist}}(\langle \hat{s}, \hat{r}\rangle; E)$, i.e., the one whose application to all the examples in E is the best in terms of f_{dist}.

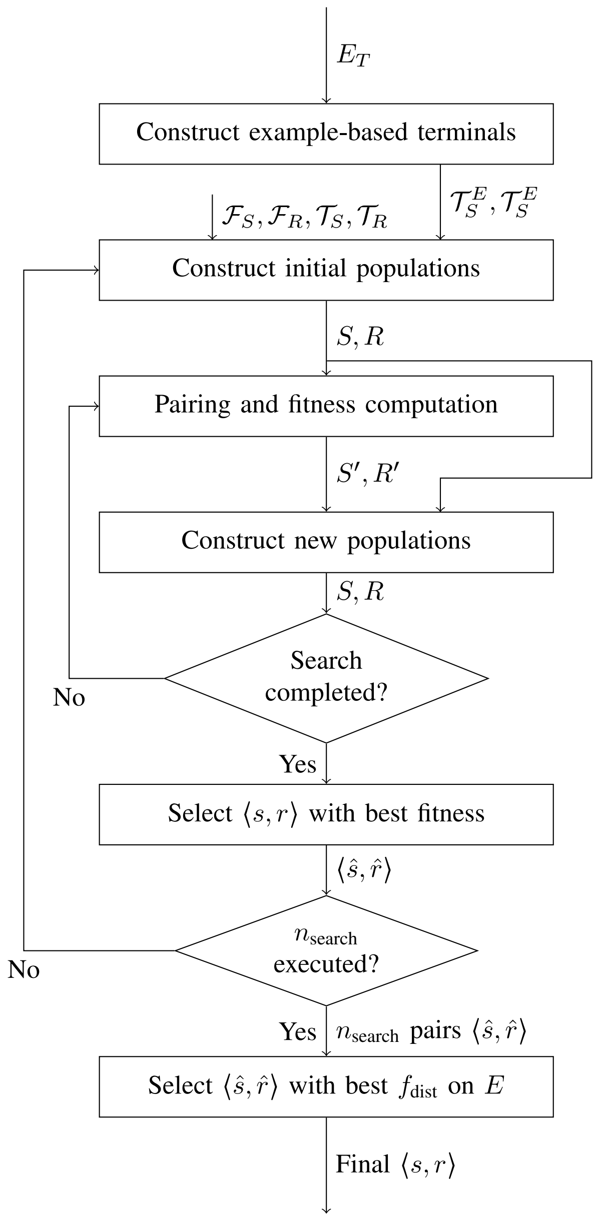

E_T
Construct example-based terminals
$\mathcal{F}_S, \mathcal{F}_R, \mathcal{T}_S, \mathcal{T}_R$
$\mathcal{T}_S^E, \mathcal{T}_S^E$
Construct initial populations
S, R
Pairing and fitness computation
S', R'
Construct new populations
S, R
Search completed?
No
Yes
Select $\langle s, r\rangle$ with best fitness
$\langle \hat{s}, \hat{r}\rangle$
n_{search} executed?
No
Yes
n_{search} pairs $\langle \hat{s}, \hat{r}\rangle$
Select $\langle \hat{s}, \hat{r}\rangle$ with best f_{dist} on E
Final $\langle s, r\rangle$

Figure 3: Outline of our proposed approach. The two first blocks implement the procedure in Section IV-D, the "Pairing and fitness computation" block and the "Construct new population" block correspond, respectively, to steps 4 and 5 in Section IV-F.

Summarizing, the parameters of the proposed approach and their actual values that we used in our experiments are the maximum tree depth $n_{\text{depth}}^0 = 7$ for initialization and the maximum tree depth $n_{\text{depth}} = 15$ during the evolution, the sizes $n_{\text{pop}}^S = 500, n_{\text{pop}}^R = 250$ of the two populations S and R, the size $n_{\hat{P}} = 200$ of the best search-and-replace expressions set $\hat{P}$, the size $n_{\text{tour}} = 5$ of the tournament selection criterion, the rates $p_{\text{elitism}} = 0.4$, $p_{\text{cross}} = 0.4$, $p_{\text{mut}} = 0.05$, $p_{\text{rnd}} = 0.05$, and $p_{\text{new}} = 0.1$ for evolving the populations S and R, the maximum number $n_{\text{gen}} = 1000$ of generations, the minimum number

$n_{\text{imp}} = 50$ of generations without fitness improvement, and the number $n_{\text{searches}} = 4$ of repetitions of the evolutionary search. Concerning genetic operators, we select subtrees to replace (mutation) or exchange (crossover) as follows: we first select one category between non-leaf or leaf nodes with probability 0.9 and 0.1, respectively; then, we select a node accordingly with uniform probability.

G. Parameter tuning

We did not execute a full optimization of all the numerous parameters. We started by using parameter values identical to those of our prior works on automatic construction of regular expressions [6] and then executed a few explorations on a subset of the tasks, as follows. We compared $n_{\text{tour}} = 7$ (used in [6]) to $n_{\text{tour}} = 5$, as we felt that a smaller tournament size could compensate the higher evolutionary pressure intrinsic into the coevolutionary approaches used in this work; indeed, we verified that $n_{\text{tour}} = 5$ led to better results. We compared $n_{\text{searches}} = 4$ (as in the public webapp implementing the framework in [6]) to $n_{\text{searches}} = 6$; we found negligible improvements with $n_{\text{searches}} = 6$ the latter thus we opted for the much less demanding $n_{\text{searches}} = 4$. We experimented with $n^R_{\text{pop}} = 50, 250, 500$ (there was no population of replacement expressions in [6]); we noticed that $n^R_{\text{pop}} = 250$ and $n^R_{\text{pop}} = 500$ led to similar results, both significantly better than $n^R_{\text{pop}} = 50$, thus we chose the less demanding value $n^R_{\text{pop}} = 250$.

As for any application of EC, finding the proper trade-off between exploration and exploitation is a central point which can eventually make the application successful or not. We designed our proposed approach for search-and-replace generation from examples as to properly balance the two aspects. We aimed at promoting exploration by including a diversity promotion scheme, by repeating the evolutionary search multiple times and by using a tournament size smaller than in similar scenarios (i.e., 5 rather than 7 as used in [5], [6]). On the other hand, for supporting exploitation, we included a strong elitism, exercised through the $p_{\text{elitism}} = 0.4$ while building the populations S and R and, indirectly, by using the set $\hat{P}$ of the best search-and-replace expressions so far.

Beyond the exploration-exploitation trade-off which manifests during the evolution, we designed a population initialization procedure heavily based on our domain knowledge, with the aim of providing several good starting points for the problem instance at hand. Moreover, we attempted to reduce the huge search space (and hence increase the search effectiveness) in two ways: (a) by tailoring of terminal sets to the examples and (b) by splitting the search for the search-and-replace expression in two sub-problems (i.e., two populations) whose respective candidate solutions are then merged based on the pairing procedure which we designed based on our domain knowledge. The latter corresponds, in other words, in searching just a small subset of the Cartesian product of the space of all the possible search patterns and the space of all the possible replacement expressions, instead of the entire large space. The idea of splitting a problem in two or more subproblems and setting the fitness of each solution component according to the ability of the corresponding full solution to solve the problem has been proven to be effective also with other EAs [34].

V. Experiments

A. Data

We experimentally evaluated our proposal on six search-and-replace tasks inspired by practical needs and based on real-world text corpora:

- *Twitter*. A task of anonymization in Twitter posts consisting in replacing each username by the string `@xxxxxx`—e.g., `@MaleLabTs` becomes `@xxxxxx`. The tweet corpus has been taken from [38].
- *IP*. A task of partial anonymization of IP addresses consisting in replacing each one of the last two digit groups of each IP address (expressed in dot-decimal notation) found in a web server log by `xxx`—e.g., `127.0.0.1` becomes `127.0.xxx.xxx`.
- *Date*. A task of date format change consisting in changing the format of each date found in the web server log of the IP task from the Gregorian little-endian slash-separated format to the Gregorian big-endian dash-separated format—e.g., `31/Dec/2012` becomes `2012-Dec-31`.
- *Phone*. A task of phone number format change consisting in changing the format of each phone number found in an email collection by removing the parenthesis around the area code and adding a dash—i.e., `(555)␣555-5555` becomes `555-555-5555`. The email corpus has been taken from [39] and was used in [26], [28], [21], [6], [23] for the generation of a regular expression able to match phone numbers from examples where it turned out to be a hard task.
- *Salary*. A task of number format change consisting in removing the thousand-separating commas from salaries of NBA players in a plain text file—e.g., `$301,450,000` becomes `$301450000`.
- *Ebook*. A task of fixing paragraph terminations in ebook files consisting in replacing wrong paragraph terminations (i.e., occurrences of `<p><\p>` not preceded by the ., !, ?, : characters) by a space character in ePub (an ebook XHTML-based format) files. The corpus consisted of some portions of three publicly available ebooks: I Promessi Sposi (Alessandro Manzoni), Pride and Prejudice (Jane Austen), Don Quijote de la Mancha (Francisco de Robles).

The first four tasks have already been used in [5]. We made the data corresponding to the last four tasks publicly available[5]; data for Twitter and IP tasks cannot be published due to license and privacy limitations.

For each task, we created a dataset E_0 of examples $\langle t, t' \rangle$ by (i) concatenating the original corpora in a single string t_0, (ii) applying a hand-written search-and-replace expression able to perform the task to t_0 obtaining a string t'_0, and, finally,

[5]http://machinelearning.inginf.units.it/data-and-tools/automatic-search-and-replace

(iii) splitting t_0, t'_0 in a number of examples $E_0 = \{\langle t, t' \rangle\}$ such that each example contained zero or one occurrences of a substring to be replaced. The size of the six resulting datasets ranged from $|E_0| = 553$ (for Salary) to $|E_0| = 1026$ (for Ebook).

B. Procedure, indexes, and baselines

For each task, we executed the following experimental procedure: (i) we split the task dataset E_0 in a learning set E of predefined size $|E|$ and a test set $E^\star$ such that $E, E^\star$ have the same positive/negative ratio of E_0 and $E \cap E^\star = \varnothing$ (the splitting is done with uniform random sampling without replacement), (ii) we generated a search-and-replace expression $\langle \hat{s}, \hat{r} \rangle$ automatically using our approach on E, and (iii) we applied $\langle \hat{s}, \hat{r} \rangle$ to the examples in $E^\star$ and measured the following error indexes:

$$\epsilon_{\text{dist}} = f_{\text{dist}}(\langle \hat{s}, \hat{r} \rangle; E^\star) = \sum_{\langle t, t' \rangle \in E^\star} \frac{d(t', \text{rep}(t, \langle \hat{s}, \hat{r} \rangle))}{\ell(t')} \quad (4)$$

$$\epsilon_{\text{count}} = \frac{1}{|E^\star|} |\{\langle t, t' \rangle \in E^\star : \text{rep}(t, \langle \hat{s}, \hat{r} \rangle) \neq t'\}| \quad (5)$$

The former, ϵ_{dist}, measures the average relative Levenshtein distance on the examples in $E^\star$ between the string t' desired after the search-and-replace and the actual string $\text{rep}(t, \langle \hat{s}, \hat{r} \rangle)$ obtained by applying the generated expression $\langle \hat{s}, \hat{r} \rangle$. The latter, ϵ_{count}, measures the ratio of $E^\star$ examples for which the $\langle \hat{s}, \hat{r} \rangle$ does not perform as requested. For both indexes, the lower, the better.

The error indexes defined above allow quantifying the quality of the solution. In particular, ϵ_{count} quantifies the ratio of examples processed correctly while ϵ_{dist} quantifies the mistakes in the examples that are not processed correctly. We remark that our evolutionary approach is based on fitness indexes that are correlated to these error indexes but are different. Most importantly, the fitness is defined so as to reward improvements in terms of ϵ_{dist} (character level) even when such improvements are still not enough to be reflected in terms of ϵ_{count} (full example level). We based this design choice on the results of our prior work on automatic construction of regular expressions from examples. We found that this design choice was fundamental for the effectiveness of the search procedure [6]. We emphasize that we assessed the generated search-and-replace expressions on a set of examples different (and much larger, see below) than those used to generate them. In other words, the examples of $E^\star$ are representative of the ideal search-and-replace expression $\langle s^\star, r^\star \rangle$ able to solve the task at hand on unseen test. Hence, zero error on $E^\star$ (i.e., $\epsilon_{\text{dist}} = 0$ and $\epsilon_{\text{count}} = 0$) means that an ideal expression has been generated.

We repeated the above procedure 15 times for each task by varying the size $|E|$ of the learning set in $\{20, 50, 100\}$ and, for each $|E|$, by repeating the procedure 5 times with different splitting of dataset E_0.

In order to place the results of our approach in perspective, we considered two baselines: our previous GP-based algorithm [5] (denoted by GP in the following) and the FlashFill algorithm [8] (denoted by FF in the following) incorporated into Microsoft Excel—for the former, we used the parameters suggested in the corresponding paper, whereas the latter is parameter-free (in the Excel implementation). We measured the error indexes ϵ_{dist} and ϵ_{count} for the two baselines with the very same procedure of our approach: in particular, we computed them based on the same 15 learning/test sets for each of the 6 tasks.

Beyond the error indexes, for our approach (denoted by MOCCGP in the following) and GP we also measured the learning time τ, i.e., the elapsed machine (wall-clock) time to output the generated search-and-replace expression. We executed all the experiments on a virtual machine with four virtual cores on an Intel Xeon X3323 (2.53 GHz) with 16 GB of RAM. The server was fully devoted to executing one virtual machine at a time. We implemented a prototype for both MOCCGP and GP in Java. We did not measure the learning time for FF: as an interactive tool part of the Microsoft Excel software, FF learning time is much shorter than the learning time of GP and MOCCGP. We remark, though, that FF does not output a search-and-replace expression which can be reused in any compatible text editing software.

C. Results and discussion

Table I summarizes the salient results for the proposed approach and the two baselines GP and FF. Each row corresponds to one combination of task and learning set size $|E|$: the row reports the average of the error indexes ϵ_{dist} and ϵ_{count} and the learning time τ (in minutes and only for MOCCGP and GP) computed across the 5 repetitions performed on the corresponding task and learning set size $|E|$. The rightmost column shows the ratio $\frac{\tau_{\text{MOCCGP}}}{\tau_{\text{GP}}}$ between the learning time of MOCCGP and the one of GP: the lower, the faster our approach compared to the one in [5].

The foremost finding of the experimental evaluation is that our MOCCGP exhibits very good performance and greatly improves over the baselines. In detail, MOCCGP exhibits zero error (i.e., $\epsilon_{\text{dist}} = \epsilon_{\text{count}} = 0$) in the Twitter task with every considered learning set size, while neither GP nor FF manages to obtain such perfect values. Furthermore, MOCCGP performs also the replacement IP and Date tasks without any mistake, which is a remarkable improvement over GP that needs at least 50 examples in order to provide a perfect search-and-replace expression. MOCCGP consistently improves over both the baselines also on tasks Phone, Salary and Ebook. It is also important to remark that in four tasks MOCCGP is able to process all the test examples correctly, when there are at least 50 examples available for learning, while in the most challenging tasks Phone and Ebook, MOCCGP processes 99% and 96.6%, respectively, of the test examples correctly (i.e., $1 - \epsilon_{\text{count}}$).

Another interesting result concerns the learning time τ. The figures shown in Table I shows that MOCCGP is clearly much faster than GP: the former takes less than 10% of the time taken by our earlier proposal on four of six tasks and never takes longer. The absolute values of τ are in the order of the minutes or, for the most challenging tasks, of the tens of minutes: these times are likely too long to make MOCCGP an

Table I: Experiment results. Error indexes ϵ_{dist} and ϵ_{count} are expressed in percentage; times are expressed in seconds. The best error indexes for each row, i.e., combination of task and $|E|$.

	N. of examples		MOCCGP			GP			FF		
Task	$\lvert E\rvert$	$\lvert E^\star\rvert$	ϵ_{dist} [%]	ϵ_{count} [%]	τ [s]	ϵ_{dist} [%]	ϵ_{count} [%]	τ [s]	ϵ_{dist} [%]	ϵ_{count} [%]	$\frac{\tau_{\text{MOCCGP}}}{\tau_{\text{GP}}}$
Twitter	20	980	**0.0**	**0.0**	15	4.0	5.5	24	35.2	47.5	0.63
	50	950	**0.0**	**0.0**	29	13.6	3.1	38	23.6	35.7	0.76
	100	900	**0.0**	**0.0**	58	4.7	2.0	121	22.4	33.8	0.48
IP	20	980	**0.0**	**0.0**	57	**0.0**	0.5	712	**0.0**	**0.0**	0.08
	50	950	**0.0**	**0.0**	186	**0.0**	**0.0**	2067	**0.0**	**0.0**	0.09
	100	900	**0.0**	**0.0**	301	**0.0**	**0.0**	2508	**0.0**	**0.0**	0.12
Date	20	980	**0.0**	**0.0**	93	29.9	60.0	1033	**0.0**	**0.0**	0.09
	50	950	**0.0**	**0.0**	90	**0.0**	**0.0**	2250	**0.0**	**0.0**	0.04
	100	980	**0.0**	**0.0**	145	**0.0**	**0.0**	4833	**0.0**	**0.0**	0.03
Phone	20	980	**0.0**	**1.4**	324	6.4	52.4	2160	10.0	43.5	0.15
	50	950	**0.0**	**6.1**	1455	3.2	8.2	4850	12.8	48.4	0.30
	100	900	**0.0**	**1.0**	2659	3.0	6.6	7820	12.0	45.6	0.34
Salary	20	533	**0.1**	**6.1**	77	0.2	10.2	771	3.9	96.4	0.10
	50	528	**0.0**	**0.0**	127	**0.0**	**0.0**	1411	4.3	91.0	0.09
	100	453	**0.0**	**0.0**	241	0.3	13.9	2678	4.0	81.9	0.09
Ebook	20	1006	**0.8**	**10.3**	231	1.4	29.2	2887	17.6	36.2	0.08
	50	976	**0.5**	**11.2**	697	**0.5**	16.4	8713	22.2	42.3	0.08
	100	926	**0.3**	**3.4**	1494	0.6	14.5	18 674	20.9	40.0	0.08

interactive tool, but support the feasibility of a fully automatic method for generating search-and-replace expressions from few tens of examples. By the way, it has been showed that even the skilled users take several minutes for authoring non-trivial regular expressions [4], which are indeed just a part of a search-and-replace expression. It is fair to emphasize, though, that FF is sufficiently fast to support interactive usage.

The previous discussion is based on the average values for the error indexes exhibited by the generated search-and-replace expressions. In order to gain more insight into the behavior of the three methods considered in our experimental evaluation, we show in Figure 4 the box plots of ϵ_{dist} and ϵ_{count} for MOCCGP, GP, and FF on the 6 tasks for all the three values of $|E|$. It can be seen that MOCCGP is at least as good as the other two methods on all the tasks w.r.t. both error indexes and for every statistics (median, 25th, and 75th percentile): Figure 4 hence suggests that our approach is consistently able to generate good search-and-replace expressions.

To corroborate the previous statements, we executed a Wilcoxon signed-rank test in order to check if there is a significant difference in the distributions of ϵ_{dist} among the three methods. We verified with significance-level $\alpha = 0.01$ that the distributions of ϵ_{dist} for MOCCGP vs. GP, MOCCGP vs. FF, and GP vs. FF are different with statistical significance—the obtained p-values being, respectively, 3.51×10^{-7}, 2.2×10^{-16}, 7.70×10^{-8}. Similar figures hold also for ϵ_{count}. The number of samples used for each of the three approaches in the Wilcoxon test was $3 \times 5 \times 6 = 90$ (all combinations of learning set size $|E|$, learning set E, and task).

In order to verify the generalization ability of our approach, we analyzed the relation between the error index ϵ_{dist} computed on the validation set $E \setminus E_T$ and on the test set $E^\star$ for each one of the $4 \times 3 \times 5 \times 6 = 360$ search-and-replace expressions generated by an evolutionary search of MOCCGP. Figure 5 summarizes the results of this analysis by plotting ϵ_{dist} on $E^\star$

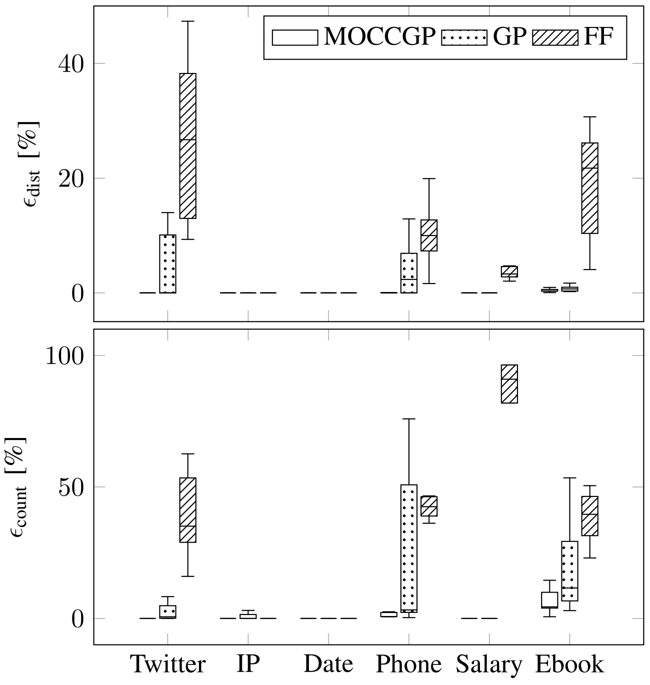


Figure 4: Box plots of ϵ_{dist} and ϵ_{count} for MOCCGP, GP, and FF on the 6 tasks for the 3 values of $|E|$.

vs. ϵ_{dist} on $E \setminus E_T$, one mark for each expression. It can be seen that, in general, the two figures are well correlated. In other words, the error index ϵ_{dist} on the validation set $E \setminus E_T$ is a good predictor of the same error index on the test set $E^\star$ which is, we remark, much larger: this finding supports the fact that MOCCGP is able to generate search-and-replace expressions which work well not only on the user-provided examples, but also on “unseen” text (here represented by $E^\star$).

Finally, Table II shows one search-and-replace expression

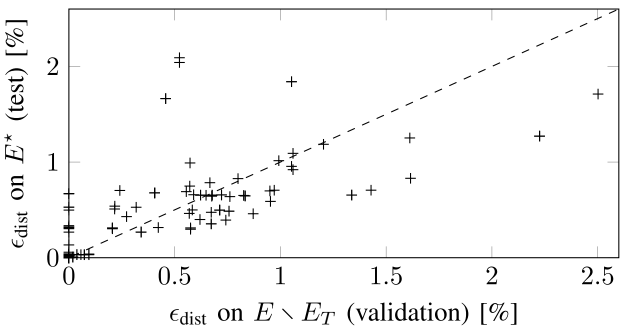


Figure 5: ϵ_{dist} computed on the test set $E^{\star}$ vs. ϵ_{dist} computed on the validation set $E \smallsetminus E_T$ for each search-and-replace expressions generated by an evolutionary search of MOCCGP.

Table II: Examples of the search-and-replace expressions generated by MOCCGP.

Task	$\hat{s}$	$\hat{r}$
Twitter	`(@)\w++`	`$1xxxxxx`
IP	`\d++(\.)\d++(\s)`	`xxx$1xxx$2`
Date	`(\d++)/(\w++)/(\d++)`	`$3-$2-$1`
Phone	`\((\d++)[^\w]*+`	`$1-`
Salary	`,(\w++),?+`	`$1`
Ebook	`(\w)</p[^\w]*+\w>`	`$1`

$\langle \hat{s}, \hat{r} \rangle$ for each of the 6 tasks—we chose one randomly among the best performing ones for each task. It can be seen that MOCCGP succeeds in generating very compact and human-readable solutions for all the tasks. As a comparison, the best search-and-replace expression generated by GP for the Twitter task is (the search pattern $\hat{s}$ is split across 3 lines):

$\hat{s}$ =
```
@\w\w[^\d]\w(?:r\w(?:[.])\wy*+\we)?+\w\w[^\d]
(?:\w(?:\w\w(?:@\w)?+)?+(?:\w(?:\w\w)?+\w?+)?+)?+
(?:\w(?:\w\w\w\w\w[^A]\w\w\w)?+)?+
```
$\hat{r}$ = `@xxxxxx`

We recall that FF does not output a search-and-replace expression usable in any compatible text editing software.

VI. Concluding remarks

We have presented an approach based on multi-objective Genetic Programming for solving a challenging problem, i.e., the automatic construction of search-and-replace expressions based solely on examples of the text manipulation task to be performed. The user provides only a set of examples, each consisting of the input text and the desired modified text, and the tool constructs a search-and-replace expression that can be used unchanged on a broad variety of text processing tools. The expression consists of a search pattern that takes the form of a regular expression and of a replacement expression that describes the changes to be applied to the substring matching the search pattern.

We assessed the effectiveness and efficiency of our proposal on six real-world search-and-replace tasks and compared its performance against the only methods available in the literature that we are aware of: the GP-based algorithm proposed in [5] and the FlashFill algorithm [8] as incorporated into the commercially available Microsoft Excel software—the latter does not output reusable search-and-replace expressions, but artifacts in a specific language usable only by FlashFill itself. We showed that our proposal is effective in constructing compact and human-readable search-and-replace expressions able to generalize beyond the provided examples. Indeed, MOCCGP clearly outperforms the two baselines in effectiveness and greatly improves over GP in efficiency (time required for learning a search-and-replace expression from the examples). Our proposal may effectively permit to people with no technical expertise to construct search-and-replace expressions in a few minutes and constitutes a further demonstration of the potential of evolutionary computation techniques in applications of practical interest.

The problem considered in this work is particularly challenging because the effectiveness of a solution does not depend only on the structure of its two components—search pattern and replacement expression—but also on their mutual interaction. Problems of this kind are generally difficult to solve with evolutionary techniques, because of the potential explosion of the solution space. The key strength of our approach is that the two components are optimized jointly. To this end, we have defined a framework based on two populations, one for each solution component, in which there is a cooperative coevolution of the two populations. While we have demonstrated the practicality of our approach only on of the automatic construction of search-and-replace expressions, our cooperative coevolutionary framework could be useful even beyond this specific problem.

Acknowledgments

We are grateful to the anonymous reviewers for their numerous and detailed suggestions leading to an improved presentation.

References

[1] A. Bartoli, A. D. Lorenzo, E. Medvet, and F. Tarlao, "Data quality challenge: Toward a tool for string processing by examples," *Journal of Data and Information Quality (JDIQ)*, vol. 6, no. 4, p. 13, 2015.

[2] J. Friedl, *Mastering Regular Expressions*. O'Reilly Media, Inc., 2006.

[3] A. Bartoli, A. De Lorenzo, E. Medvet, and F. Tarlao, "Playing regex golf with genetic programming," in *Proceedings of the 2014 Annual Conference on Genetic and Evolutionary Computation*. ACM, 2014, pp. 1063–1070.

[4] ——, "Can a machine replace humans in building regular expressions? a case study," *IEEE Intelligent Systems*, vol. 31, no. 6, pp. 15–21, 2016.

[5] A. De Lorenzo, E. Medvet, and A. Bartoli, "Automatic string replace by examples," in *Proceedings of the 15th annual conference on Genetic and evolutionary computation*. ACM, 2013, pp. 1253–1260.

[6] A. Bartoli, A. De Lorenzo, E. Medvet, and F. Tarlao, "Inference of regular expressions for text extraction from examples," *IEEE Transactions on Knowledge and Data Engineering*, vol. 28, no. 5, pp. 1217–1230, 2016.

[7] G. Squillero and A. Tonda, "Divergence of character and premature convergence: A survey of methodologies for promoting diversity in evolutionary optimization," *Information Sciences*, vol. 329, pp. 782–799, 2016.

[8] S. Gulwani, "Automating string processing in spreadsheets using input-output examples," in *ACM SIGPLAN Notices*, vol. 46, no. 1. ACM, 2011, pp. 317–330.

[9] J. Hamza and V. Kunčak, "Minimal synthesis of string to string functions from examples," *arXiv preprint arXiv:1710.09208*, 2017.

[10] R. Miller and B. Myers, "Lapis: Smart editing with text structure," in *CHI'02 extended abstracts on Human factors in computing systems*. ACM, 2002, pp. 496–497.

[11] R. Miller, B. Myers *et al.*, "Lightweight structured text processing," in *Proceedings of 1999 USENIX Annual Technical Conference*, 1999, pp. 131–144.

[12] R. Miller and B. Myers, "Multiple selections in smart text editing," in *Proceedings of the 7th international conference on Intelligent user interfaces*. ACM, 2002, pp. 103–110.

[13] R. Miller and A. Marshall, "Cluster-based find and replace," in *Proceedings of the SIGCHI conference on Human factors in computing systems*. ACM, 2004, pp. 57–64.

[14] A. Menon, O. Tamuz, S. Gulwani, B. Lampson, and A. Kalai, "A machine learning framework for programming by example," in *Proceedings of the 30th International Conference on Machine Learning (ICML-13)*, 2013, pp. 187–195.

[15] M. Tomita, "Dynamic construction of finite automata from example using hill-climbing," in *Proc. of the Fourth Annual Cognitive Science Conference*, 1982, pp. 105–108.

[16] B. Dunay, F. Petry, and B. Buckles, "Regular language induction with genetic programming," in *Evolutionary Computation, 1994. IEEE World Congress on Computational Intelligence., Proceedings of the First IEEE Conference on*, vol. 1. IEEE, 1994, pp. 396–400.

[17] B. Svingen, "Learning Regular Languages Using Genetic Programming," in *Genetic Programming 1998 Proceedings of the Third Annual Conference*, J. R. Koza, W. Banzhaf, K. Chellapilla, K. Deb, M. Dorigo, D. B. Fogel, M. H. Garzon, D. E. Goldberg, H. Iba, and R. Riolo, Eds. Morgan Kaufmann, 1998, pp. 374–376.

[18] A. Cetinkaya, "Regular expression generation through grammatical evolution," in *Proceedings of the 9th annual conference companion on Genetic and evolutionary computation*. ACM, 2007, pp. 2643–2646.

[19] D. F. Barrero, D. Camacho, and M. D. R-moreno, "Automatic web data extraction based on genetic algorithms and regular expressions," *Data Mining and Multi-agent Integration*, pp. 143–154, 2009.

[20] A. González-Pardo, D. F. Barrero, D. Camacho, and M. D. R-Moreno, "A case study on grammatical-based representation for regular expression evolution," in *Trends in Practical Applications of Agents and Multiagent Systems*. Springer, 2010, pp. 379–386.

[21] A. Bartoli, G. Davanzo, A. De Lorenzo, M. Mauri, E. Medvet, and E. Sorio, "Automatic generation of regular expressions from examples with genetic programming," in *Proceedings of the 14th GECCO conference companion*. ACM, 2012, pp. 1477–1478.

[22] A. Bartoli, A. De Lorenzo, E. Medvet, and F. Tarlao, "Learning text patterns using separate-and-conquer genetic programming," in *European Conference on Genetic Programming*. Springer, 2015, pp. 16–27.

[23] ——, "Active learning of regular expressions for entity extraction," *IEEE Transactions on Cybernetics*, 2017.

[24] T. Wu and W. Pottenger, "A semi-supervised active learning algorithm for information extraction from textual data," *Journal of the American Society for Information Science and Technology*, vol. 56, no. 3, pp. 258–271, 2005.

[25] E. Kinber, "Learning regular expressions from representative examples and membership queries," *Grammatical Inference: Theoretical Results and Applications*, pp. 94–108, 2010.

[26] Y. Li, R. Krishnamurthy, S. Raghavan, S. Vaithyanathan, and A. Arbor, "Regular Expression Learning for Information Extraction," *Computational Linguistics*, no. October, pp. 21–30, 2008.

[27] R. Babbar and N. Singh, "Clustering based approach to learning regular expressions over large alphabet for noisy unstructured text," in *Proceedings of the fourth workshop on Analytics for noisy unstructured text data*, ser. AND '10. New York, NY, USA: ACM, 2010, pp. 43–50.

[28] F. Brauer, R. Rieger, A. Mocan, and W. Barczynski, "Enabling information extraction by inference of regular expressions from sample entities," in *Proceedings of the 20th ACM international conference on Information and knowledge management*. ACM, 2011, pp. 1285–1294.

[29] M. A. Potter and K. A. De Jong, "A cooperative coevolutionary approach to function optimization," in *International Conference on Parallel Problem Solving from Nature*. Springer, 1994, pp. 249–257.

[30] L. Ke, Q. Zhang, and R. Battiti, "A simple yet efficient multiobjective combinatorial optimization method using decompostion and pareto local search," *IEEE Trans on Cybernetics, accepted*, 2014.

[31] S. Nguyen, M. Zhang, M. Johnston, and K. C. Tan, "Automatic programming via iterated local search for dynamic job shop scheduling," *Cybernetics, IEEE Transactions on*, vol. 45, no. 1, pp. 1–14, 2015.

[32] J. M. Luna, J. R. Romero, C. Romero, and S. Ventura, "On the use of genetic programming for mining comprehensible rules in subgroup discovery," *Cybernetics, IEEE Transactions on*, vol. 44, no. 12, pp. 2329–2341, 2014.

[33] M. Yang, C. Li, Z. Cai, and J. Guan, "Differential evolution with auto-enhanced population diversity," *Cybernetics, IEEE Transactions on*, vol. 45, no. 2, pp. 302–315, 2015.

[34] K. A. De Jong and M. A. Potter, "Evolving complex structures via cooperative coevolution." in *Evolutionary Programming*, 1995, pp. 307–317.

[35] J. R. Koza, "Genetic Programming: On the Programming of Computers by Means of Natural Selection (Complex Adaptive Systems)," vol. 1, 1992.

[36] K. A. De Jong, *Evolutionary computation: a unified approach*. MIT press, 2006.

[37] E. D. De Jong and J. B. Pollack, "Multi-objective methods for tree size control," *Genetic Programming and Evolvable Machines*, vol. 4, no. 3, pp. 211–233, 2003.

[38] E. Medvet and A. Bartoli, "Brand-related events detection, classification and summarization on twitter," in *Proceedings of the 2011 IEEE/WIC/ACM International Conferences on Web Intelligence and Intelligent Agent Technology - Volume 01*, ser. WI-IAT '12. IEEE Computer Society, 2012, to appear.

[39] E. Minkov, R. C. Wang, and W. W. Cohen, "Extracting personal names from email: applying named entity recognition to informal text," in *Proceedings of the conference on Human Language Technology and Empirical Methods in Natural Language Processing*, ser. HLT '05. Stroudsburg, PA, USA: Association for Computational Linguistics, 2005, pp. 443–450.

Alberto Bartoli received the degree in Electrical Engineering in 1989 cum laude and the PhD degree in Computer Engineering in 1993, both from the University of Pisa, Italy. Since 1998 he is an Associate Professor at the Department of Engineering and Architecture of University of Trieste, Italy, where he is the Director of the Machine Learning Lab. His research interests include machine learning applications, evolutionary computing, and security.

Andrea De Lorenzo received the diploma degree in Computer Engineering cum laude from the University of Trieste, Italy in 2006 and the MS degree in Computer Engineering in 2010. He received the PhD degree in Computer Engineering in 2014, endorsed by the University of Trieste. His research interests include evolutionary computing, computer vision, and machine learning applications.

Eric Medvet received the degree in Electronic Engineering cum laude in 2004 and the PhD degree in Computer Engineering in 2008, both from the University of Trieste, Italy, where he is currently an Assistant Professor in Computer Engineering and the Director of the Evolutionary Robotics and Artificial Life Lab. His research interests include Genetic Programming and machine learning applications, in particular concerning Android malware detection and information retrieval.

Fabiano Tarlao received the degree in Electronic Engineering in 2010 and the PhD degree in Computer Engineering in 2017, both from the University of Trieste, Italy. He is currently a PostDoc research fellow at the Department of Engineering and Architecture at University of Trieste, Italy. His research interests are in the areas of web security, Genetic Programming, and machine learning applications.