Genetic Programming and Evolvable Machines (2022) 23:105–131
https://doi.org/10.1007/s10710-021-09411-x

Automatic generation of regular expressions for the Regex Golf challenge using a local search algorithm

André de Almeida Farzat[1] · Márcio de Oliveira Barros[1]

Received: 2 July 2021 / Revised: 17 August 2021 / Accepted: 18 August 2021 /
Published online: 1 October 2021

Abstract

Regular expression is a technology widely used in software development for extracting textual data, validating the structure of textual documents, or formatting data. Regex Golf is a challenge that consists in finding the smallest possible regular expression given a set of sentences to perform matches and another set not to match. An algorithm capable of meeting the Regex Golf requirements is a relevant contribution to the area of semi-structured document data extraction. In this paper, we propose a heuristic search algorithm based on local search, combined with a regular expression shrinker, to find valid results for Regex Golf problems. An experimental study was conducted to compare the proposed technique with an exact algorithm and a genetic programming algorithm designed for the Regex Golf challenge. The proposed local search was shown to outperform both competing algorithms in six out of fifteen problem instances, tying in another three instances. On the other hand, all algorithms still lack the ability to outperform human software developers in designing regular expressions for the challenge.

Keywords Regular expressions · Regex Golf · Local search · Heuristic search

✉ Márcio de Oliveira Barros
marcio.barros@uniriotec.br

André de Almeida Farzat
andre.farzat@uniriotec.br

[1] Federal University of the State of Rio de Janeiro, Av. Pasteur, 458 Urca, Rio de Janeiro, RJ 22290-240, Brazil

1 Introduction

Regular expressions are a technology widely used in software development for extracting data from textual documents, validating the structure of such documents, or formatting data. However, building a regular expression tailored to a specific problem is often difficult, tricky, and time-consuming. It requires a considerable amount of skill, expertise, and creativity by the programmer [5] to assure that the regular expression is correct in all scenarios expected for its usage.

The wide application and high development cost of regular expressions has attracted many researchers to develop algorithms to automatically generate these expressions. Among the techniques tested for that purpose we find exact algorithms [14], genetic programming [2, 4, 7, 8, 11], machine learning [4, 16], and a combination of them. Some of the techniques for automatic creation of regular expressions use a list of text samples to be extracted from textual documents by means of the generated regular expression to ensure its quality [7].

In 2014, Peter Norvig published a blog post featuring a challenge called *Regex Golf* [14]. It is based on Code Golf challenges, where the goal is to create the smallest algorithm possible to solve a given problem. The *Regex Golf* challenge involves creating the smallest regular expression that matches all phrases in a list (known as match list) and does **not** match any sentence in a second list (known as unmatch list) [3]. Table 1 presents an instance of the problem: a match list with the names of all books from the “Harry Potter” series and an unmatch list with the names from the books of the “The Galaxy Backpacker’s Guide” series.

A solution to this instance is the regular expression `^the\s[^r][^i]`, where ^ is the line start marker used to determine that the regular expression needs to look for the string at the beginning of the line, `the` symbolizes the literal string “the”, `\s` marks a blank space, `[^r]` indicates that the following character cannot be the letter `r` and `[^i]` indicates that the next character cannot be the letter `'`i’. This instance of the problem will be used in the examples in the next sections.

An algorithm capable of meeting the Regex Golf challenge may help extracting data from textual documents, particularly from sets of documents sharing the same structure, such as a set of filled documents from the same form. Algorithms that generate regular expressions from positive examples only (match lists) may tend to

Table 1 An instance of the Regex Golf challenge

Match list	Unmatch list
The philosopher’s stone	The hitchhiker’s guide to the galaxy
The chamber of secrets	The restaurant at the end of the universe
The prisoner of azkaban	Life, the universe and everything
The goblet of fire	So long, and thanks for all the fish
The order of the phoenix	Mostly harmless
The half-blood prince	And another thing
The deathly hallows	

generate generic expressions (e.g. `.*`) due to the lack of constraints and examples of text snippets that cannot be accepted by the expression being built. On the other hand, using text snippets from other parts of the document as counterexamples to create regular expressions requires the ability to use an unmatch list. Finally, reducing the size of the regular expression has the advantage of making it easier for a developer to process and understand.

Generating regular expressions is a complex and error-prone process [1] and Regex Golf is considered an NP-hard problem [3]. We rely on heuristic search to find good solutions for the problem. Heuristic search algorithms are methods that yield good results for combinatorial parsing problems with this level of complexity. In this paper, we describe a Local Search algorithm that, combined with a regular expression shrinker, constitutes a new heuristic for the Regex Golf challenge. We tested the proposed algorithm using 15 instances of the challenge and compared it to an exact algorithm and a Genetic Programming heuristic designed for the same problem. The heuristic proposed in this paper produced competitive results to those of the exact algorithm, winning in eight instances and tying in three, and better results than Genetic Programming, losing only in two instances and executing in a shorter time. The proposed algorithm was also compared to humans' results, although it ties in three instances and loses all others.

The major contributions of this paper include: (a) a local search based algorithm to automatically create regular expressions for the Regex Golf challenge; (b) an experimental study performed to compare the proposed algorithm quantitatively with other state-of-the-art algorithms designed to the same end, as well as compare them to human-designed solutions; and (c) a thorough qualitative analysis of the former results, highlighting the conditions that lead one or another algorithm to produce better solutions. While the first two contributions allow us to claim the badge for the best algorithm to create solutions for the Regex Golf challenge, the last contribution paves the way to dethroning our proposed algorithm.

The local search outperforms the more complex genetic programming search because (a) it uses mutation operators carefully designed for the domain of the problem at hand; (b) the shrinking algorithm simplifies solutions made complex by the search process and allows new searches from a functional-equivalent standpoint; and (c) it is fast. These properties allow integrating the local search (or the domain knowledge used to build its mutation operators) into a Genetic Programming framework. Besides that, the paper establishes a benchmark for the *Regex Golf* problem, as we show human solutions that outperform both search approaches. Human-developed regular expressions set a new target for the competitiveness of automated regex generation algorithms, showing the distance that must be gained by such algorithms to outperform human developers on this task.

This paper is organized into five sections, starting with this introduction. Section 2 presents related work, briefly describing several approaches that have been used to generate regular expressions automatically. Section 3 presents our proposed solution, which is evaluated in Sect. 4. Section 5 closes the paper, wrapping up its contributions.

2 Related work

By mapping the literature related to the generation of regular expressions, we found out that three classes of algorithms were already used to that end: exact algorithms, machine learning [4, 16], and heuristic optimization [2, 4, 7, 8, 11].

We hereby classify an algorithm as `exact` if it produces the same outputs for a given set of inputs, regardless of how many times, when, or in which computing environment it is executed. Rastogi et al. [15] created a log analysis system that generates regular expression to extract data from the logs and create reports for the user. Larson et al. [12] present an algorithm to create test cases for regular expressions. In comparison with a similar algorithm, the proposal proved to be more efficient in identifying regular expressions that accept a broader selection of texts than they should. Zhang et al. [17] created an algorithm that uses generated regular expressions to detect automatic download attacks, checking if the URL addresses accessed by a web application belong to a list of websites known to distribute *malware*. Norvig [14] presents an exact algorithm for the *Regex Golf* challenge that builds regular expressions from the elements contained in the match and unmatch lists. The author tested his algorithm with lists created with arbitrary data (such as the subtitles of *Star Wars* and *Star Trek* movies) and showed that it was able to build regular expressions that worked according to the challenge.

As for heuristic search algorithms, there is a predominance in the literature of heuristics based on genetic algorithms, although some studies use local search for the automatic generation of regular expressions. Yunyao Li [13] presents a local search algorithm for the automatic generation of regular expressions based on positive and negative examples. The algorithm was tested on 50,000 web pages to generate expressions to extract software names, course names, and phone numbers. These expressions achieved a success rate of 92%, 69% and 35%, respectively. The approach presented in this paper applies a shrinking algorithm and restarts the search systematically when it finds a local minimum. Also, it uses a broader list of neighborhood operators to improve the search than Li's work.

Cetinkaya [8] presents an algorithm based on Grammar Evolution that uses a list of statements to automatically create a regular expression that matches such statements. The technique was evaluated based on a web page with 266 anchor *tags* (`a` tags with links to other pages) and was successful in identifying similar *tags* on other web pages. Cody-Kenny et al. [10] present a Grammatical Evolution-based algorithm that improves the performance of regular expressions. For a given regular expression, the algorithm searches a functionally equivalent expression (one that passes a set of positive and negative test cases) that requires less runtime to evaluate under a test suite. The authors selected nine regular expressions used in libraries such as AngularJS and D3 or recognized as problematic cases regarding execution time. They built a test suite for fitness evaluation using known texts that matched the expressions and systematically changed them to create negative tests. They found regular expressions with shorter runtime for all cases subjected to optimization with observed improvement ranging from 7 to 12,000%.

Bartoli et al. [2] created a Genetic Programming algorithm to generate regular expressions for textual data extraction based on examples. Each example consists in a string `t` and the *substring* `s` of `t` that should be extracted by the desired expression. The authors compared the proposed algorithm to the local search presented by Yunyao Li et al. [13] and found similar results. They also found that the genetic programming based algorithm outperformed Cetinkaya's algorithm [8] obtaining almost 100% precision (against 76%) in the proposed tasks.

Bartoli et al. [3, 6] also proposed a genetic programming algorithm to generate regular expressions for the *Regex Golf* challenge. This algorithm used the work of Norvig [14] as a basis to improve the generation of regular expressions for the challenge. The authors tested their algorithm using 16 instances extracted from a website (https://alf.nu/RegexGolf) that provides an online version of *Regex Golf* in which any person can suggest regular expressions for various instances of the challenge. The algorithm presented better solutions than those suggested by humans, but it did not reach the maximum score for all instances. The same website was used in the experimental studies carried out in this paper. However, the solutions found by humans improved significantly and currently exceed the results found by Bartoli et al's algorithm [3].

Cochran et al. [9] present a programming crowd-sourcing approach that consists of hiring several developers to generate solutions to a difficult problem and feeding these solutions as the initial population of a Genetic Programming algorithm that will merge and evolve them into a final solution. The underlying ideas are that each developer will solve part of the problem and the optimization algorithm can merge the best parts of their solutions to build a better solution. The authors applied the proposed approach to create regular expressions for four problems, using positive and negative examples to validate these expressions. They used symbolic finite automata to represent the regular expressions and developed specific crossover and mutation operators that leverage in this representation. Experimental evaluation indicates that the crossover operator, despite of being applied more frequently, yields limited benefits to the evolutionary process, whereas the mutation operator is more relevant. Results show an average increase of 16.25% on the accuracy of the initial regular expressions by using the proposed approach.

Finally, among the `machine learning` algorithms designed to generate regular expressions, active learning seems to be the most frequently used technique. Wu et al. [16] present a semi-supervised active learning algorithm that identifies relevant snippets of input data for generating regular expressions based on examples presented by the user. Bartoli et al. [4] use a similar approach; while the regular expressions themselves are build using genetic programming, active learning is used to scan a large textual corpus and find the next relevant piece of data to be collected based on examples given by the user. Upon knowing the correctness of the selection, the machine learning algorithm feeds the heuristic approach with a new set of sample texts from which the expression is created.

3 A local search algorithm to build regular expressions

The Regex Golf challenge is considered an NP-hard [3, 14] problem and heuristic search algorithms have shown good results while addressing such problems. We already have automatic regular expression generation algorithms complying with the Regex Golf challenge, such as Norvig's exact algorithm [14] and Bartoli's Genetic Programming algorithm [3]. However, while the first does not attempt to minimize the regular expression that it generates, the second requires a lot of computational power to produce a regular expression.

We developed a local search algorithm that works with a regular expression shrinker algorithm to find solutions for the Regex Golf challenge. Local search was selected as the basis of the new algorithm because we wanted to explore the capabilities of a simple heuristic approach in comparison to the state-of-the-art genetic algorithm presented by Bartoli et al. [3]. We believed such algorithm might produce a comparable regular expression by using a fraction of the computational time required by the genetic algorithm and be useful as a platform to test domain-specific mutation operators that might be further integrated into a genetic programming approach.

The following subsections describe the proposed algorithm in detail. It receives the match and unmatch lists as input and creates two character lists to help generating the neighborhood for a solution (Sect. 3.1). The search starts from a randomly generated solution (Sect. 3.4) and is driven by a fitness function that encodes the objectives of the Regex Golf challenge (Sect. 3.2). The neighborhood of this solution is generated following the transformations presented in Sect. 3.3.

The local search heuristic uses the Best-ascent Hill Climbing strategy, that is, all solutions that make up the neighborhood of the current solution are evaluated and the algorithm moves to the best among them in each interaction. All possible combinations of operands for each operator are tested to transform the current solution into a new one. If any combination of operator and operands produce a solution that is fitter than current one, the fittest among these solutions is selected as the basis for the next interaction. If the neighborhood has no better solution than the current one, the regular expression shrinker algorithm (Sect. 3.5) is enacted to shorten the representation of the current solution. If the shrinker succeeds in finding a smaller regular expression that is equivalent to the current solution, the local search proceeds from this smaller expression. Otherwise, a new solution is randomly generated and the local search is restarted from it.

3.1 Solution representation

A solution evolved by the local search is represented by a syntax tree whose leaf nodes are characters and whose intermediate (function) nodes are operators used in the regular expression. A leaf node is represented by the character it conveys, while function nodes have characters representing the regular expression operator and a placeholder, the • character, which represents the location where operands represented in its branches are placed when the expression is converted to its

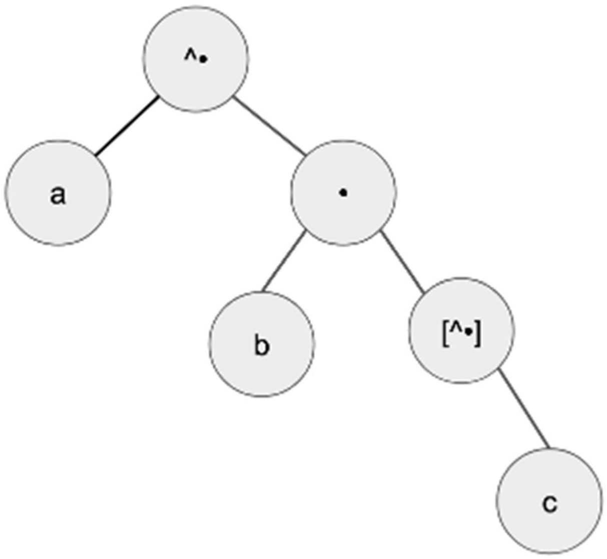

Fig. 1 Syntax tree representation of regular expression `^ab[^c]`

textual format. This model is based on the solution representation used by Bartoli et al. [3]. Figure 1 presents the regular expression `^ab[^c]` in its syntax tree format.

Some operators in the regular expression are represented by more than one function node type. For instance, the quantifier operator can be represented by three different function nodes: the `Star` quantifier, the `Plus` quantifier, and the `Repetition`. The `Character Set` operator can be represented by two function nodes—List and Range—to denote different subsets of characters. The `Concatenation` operator, which concatenates its two operands, has been added to aid in the representation and mutations performed in the syntax tree. Table 2 presents all possible function nodes and their representation.

3.2 Fitness function

In heuristic search, a fitness function is used to determine the quality of a solution. A solution is considered feasible if it represents a regular expression that matches all phrases in the match list and does not match any phrases in the unmatch list. The fitness function used in the proposed local search algorithm is based on the work of Bartoli et al. [3], which also used the fitness function of the website https://alf.nu/RegexGolf from where instances[1] of the Regex Golf challenge were selected for the experimental evaluation of their algorithm.

To calculate the fitness of a solution, we employ the same procedure used by Bartoli et al. [3]. The regular expression is executed for each phrase in the match list

[1] By *instance* we mean a description of a Regex Golf challenge, comprised by a *match* and a *unmatch* list. We have collected 15 of such descriptions from the website for experimental evaluation purposes.

Table 2 Function nodes of the syntax tree representing regular expressions in the solution space

Function node	Representation
Concatenation	•
Alternation	• | •
Negative char set	[^•]
Start anchor	^•
End anchor	•$
Star Quantifier	•*
Plus Quantifier	•+
Repetition	•#
List	[•]
Range	[•-•]
Optional	•?
Group	(•)
Dot	•
Back-reference	\#•

and the unmatch list. When a match occurs in the match list, the solution scores a point. When a match occurs in the unmatch list, the solution loses a point. The result is multiplied by a weight defined on an instance basis to represent the instance's complexity. Finally, the number of characters in the text representation of the regular expression is subtracted from the score to favor smaller solutions. Therefore, the fitness function is defined by Eq. (1).

$$fitness(R) = W_R \times (M - U) - length(R) \tag{1}$$

where R is the regular expression represented by the solution, M is the number of matches in the match list, U is the number of matches in the unmatch list, and W_R is the weight of the instance as defined in the website that stores instances for the Regex Golf challenge. If two solutions have the same score, the tie-breaking criteria are the number of characters in the solution (the smaller, the better) and the number of matches in the match list.

The maximum score a solution can achieve is the number of phrases in the match list times W_R. It is important to emphasize that the minimum and maximum fitness values are related to the problem instance and not to the problem itself. Furthermore, negative results are possible.

3.3 Neighborhood generation

The neighborhood of a solution is built performing transformations on the regular expression represented by the solution. These transformations were based on both the exact [14] and genetic programming [3] algorithms created for the Regex Golf challenge. They use four-character lists, formed from the phrases of the problem instance, presented below.

Table 3 Match char list: character list composed by extracted and sorted characters from the match list

t	\s	o	e	h	r	i	f	l	p	n	b	a
s	d	c	m	z	k	g	’	x	–	y	w	

3.3.1 Match char list

A character list consisting of characters from the match list. All valid characters are extracted from the match list, removing duplicates to make a list of unique characters. Characters are ordered based on how often they appear in sentences. Table 3 presents this list for the problem instance used as an example in the introduction.

3.3.2 Unmatch char list

A character list similar to match char list, but based on unmatch list. It contains all valid characters from the unmatch list phrases and follows the same sorting criteria based on the characters most present in these phrases. Table 4 presents this list for the problem instance used as an example.

3.3.3 Inclusive char list

A list of all characters presented in match char list that are not present in the unmatch char list. Table 5 presents this list for the problem instance used as an example.

3.3.4 Exclusive char list

A list of all characters presented in unmatch char list that are not present in match char list. Table 6 presents this list for the problem instance used as an example.

The neighborhood of a solution is comprised of all solutions generated from the transformations presented below:

- **Swap**: exchanges each tree node `A` for a leaf node with a character from the inclusive char list, if `A` is either a leaf node or a function node with a single parameter, or a concatenation node with a character from the inclusive char list on the left and `A` on the right, if `A` is a function node. Concatenation nodes from the original tree are not replaced and *start anchor* nodes are converted to concatenation when placed on the right. This transformation is performed N times for each node of the current solution, N being the number of items in the inclusive char list. The result is $N \times L$ generated solutions, L being the number of nodes in the current solution except for concatenation nodes. For instance, given the inclusive char list (t, x) and the regular expression `^ab[^c]`, the following solutions are generated: `tab[^c]`, `^tb[^c]`, `^at[^c]`, `^abt`,

Table 4 Unmatch char list: Character list composed by extracted and sorted characters from the unmatch list

t	\s	a	r	e	h	s	o	i	d	g	n
l	f	y	u	k	v	,	x	’	c	m	

Table 5 Inclusive char list: Character list composed by characters from the match char list which do not appear in the unmatch char list

p	b	z	–	w

Table 6 Exclusive char list: Character list composed by characters from unmatch char list which do not appear in the match char list

u	v	,

`^ab[^t]`, `xab[^c]`, `^xb[^c]`, `^ax[^c]`, `^abx`, and `^ab[^x]`. Figure 2 depicts the first four solutions generated by the swap transformation if applied to the regular expression `^ab[^c]`;

- **Concatenation**: exchanges a tree node for a `Concatenation` operator containing a leaf node with a character from inclusive char list and a node extracted from the branches of the solution. This transformation is done $2 \times N$ times for each node of the current solution, N being the number of items in the inclusive char list. The result is up to $2 \times N \times L$ generated solutions, L being the number of nodes in the current solution. For instance, given the inclusive char list (t, x) and the regular expression `^ab[^c]`, the following valid solutions are gener-
ated: `^tab[^c]`, `^atb[^c]`, `^abt[^c]`, `^ab[^ct]`, `^ab[^c]t`, `^-xab[^c]`, `^axb[^c]`, `^abx[^c]`, `^ab[^cx]`, and `^ab[^c]x`. Invalid solutions, such as `t^ab[^c]` and `x^ab[^c]`, are discarded, as nothing could be on the left of the start anchor node. The transformation is performed twice for

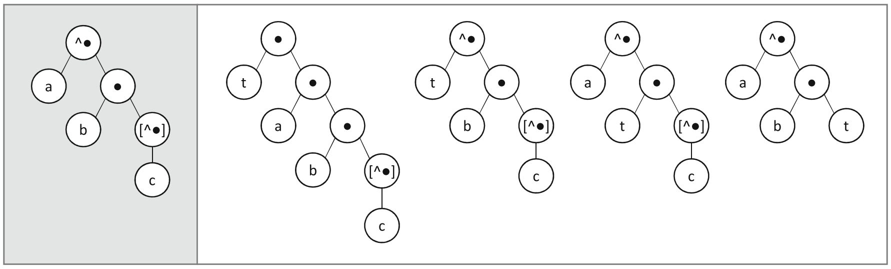

Fig. 2 The first four solutions generated by the swap operator if applied to the base regular expression (shown on the left-side)

each node of the current solution to ensure that the additional leaf node is positioned in both branches. Figure 3 depicts the first four solutions generated by the concatenation transformation if applied to `^ab[^c]`;

- **Start anchor**: adds a node with the `Start Anchor` operator as the root of the solution tree. For instance, given the regular expression `abc`, the solution `^abc` is generated. This transformation also occurs in function node branches of type `Alternation`. For instance, given the regular expression `abcd|ef|`, the solutions `^ab|cd|ef|`, `|ab|^cd|ef|`, and `ab|cd|^ef` are generated. This leads to $X + 1$ solutions, X being the number of `Alternation` nodes in the current solution;
- **End anchor**: exchanges the last node for a function node with the `Concatenation` operator, containing the replaced node and a node with the `End Anchor` on its branches. For instance, given the regular expression `abc`, the solution `abc$` is generated. As the **Start Anchor** transformation, this transformation also occurs in function nodes with the `Alternation` operator. For instance, given the regular expression `abcd|ef|`, the solutions `ab|cd|ef$`, `ab|cd$|ef`, and `ab$|cd|ef` are generated. This leads to $X + 1$ solutions, X being the number of `Alternation` nodes in the current solution;
- **Alternative**: replaces a tree node with a function node with the `Alternation` operator containing the replaced node in one of its branches. This transformation is performed once for each node of the current solution, but often the generated result is invalid and ends up being discarded. For instance, given the regular expression `^ab[^c]`, the following valid solutions are generated: `^a|b[^c]` and `^ab|[^c]`;
- **Negation**: replaces a tree node for a function node with the `Negative char set` operator and having one character from the exclusive char list as its operand. This transformation is performed N times for each node of the current solution, N being the number of items in the exclusive char list. The result is $(N + 1) \times L$ solutions, L being the number of nodes in the current solution. If the node to be exchanged is already a `Negative char set` node, the value that would be added to the tree is added to the current node, resulting in a `negative char set` node with one more character. For example, given the exclusive char list (x, y) and the regular expression `ab[^c]`, the solutions `[^x]ab[^c]`, `[^x]b[^c]`, `a[^x]b[^c]`, `a[^x][^c]`, `ab[^x][^c]`, `ab[^cx]`, `ab[^c][^x]`, `[^y]ab[^c]`, `[^y]b[^c]`,

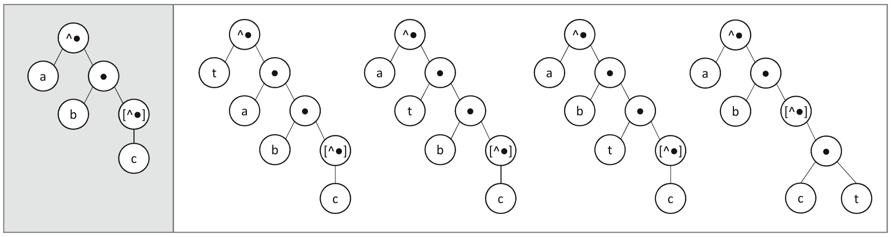

Fig. 3 The first four solutions generated by the concatenation operator if applied to the base regular expression (shown on the left-side)

a[^y]b[^c], a[^y][^c], ab[^y][^c], ab[^cy], and ab[^c][^y] are generated;

- **Range**: first, this operator creates a list of function nodes with the Range operator having the values of all possible two-character combinations of the including char list. For instance, if the including char list is (*a*, *h*, *m*, *x*), the following nodes are created: [ah], [am], [ax], [hm], [hx], and [mx]. Than, a tree node is exchanged for a function node of this list. This transformation is performed *N* times for each node of the current solution and *N* times by adding the node as the root of the tree, *N* being the number of node in the newly created list. The result is $(N + 1) \times L$ solutions, *L* being the number of nodes in the current solution. For instance, given the including char list (*a*, *m*, *y*) and the regular expression xyz, the solutions [am]xyz, [am]yz, x[am]yz, x[am]z, xy[am]z, xy[am], xyz[am], [ay]xyz, [ay]yz, x[ay]yz, x[ay]z, xy[ay]z, xy[ay], xyz[ay], [my]xyz, [my]y, x[my]yz, x[my]z, xy[my]z, xy[my], and xyz[my] are generated;
- **Star quantifier**: exchanges a tree node for a function node with the Concatenation operation having a function node with the Star quantifier and the extracted node in its branches. This transformation is performed only on leaf nodes and function nodes of the types List, Range, Group and Negation. This leads to *N* solutions, with *N* being the number of leaf nodes plus the number of function nodes of the types supported by this transformation. For instance, given the regular expression ab[hz], the solutions a*b[hz], ab*[hz], and ab[hz]* are generated;
- **Plus quantifier**: has the same behavior as the *Star quantifier* transformation, but uses a Plus quantifier node. For instance, given the regular expression ab[hz], the solutions a+b[hz], ab+[hz] and ab[hz]+ are generated;
- **Repetition**: exchanges a tree node for a function node with a Concatenation operator having a Repetition node (with the literal value 2) and the extracted node in its branches. Like the *Star quantifier*, this transformation is performed only on leaf nodes and function nodes of the types List, Range, Group and Negation. For instance, given the regular expression ab[^z], solutions a{2}b[^z], ab{2}[^z], and ab[^z]{2} are generated. If a Repetition node is being subjected to the transformation, the number of repetitions is incremented by one instead of the change described above. For instance, transforming the regular expression ab{2}[^z] yields the solution ab{3}[^z]. This leads to $N + L$ solutions, *N* being the number of end nodes plus the number of function nodes of the types accepted by this transformation and *L* the number of nodes of type Repetition;
- **Dot**: replaces a tree node with a Concatenation node having a function node with the Dot operator and the node extracted from the solution in its branches. This transformation is performed *N* times for each node of the current solution and once by adding the Dot node as the root of the tree, *N* being the number of nodes in the current solution. Since invalid solutions can be generated, the result is up to $N + 1$ solutions. For instance, given the regular expression ab[^c],

the following valid solutions are generated: `.ab[^c]`, `.b[^c]`, `a.b[^c]`, `a.[^c]`, `ab.[^c]` and `ab.`;

- **Back reference**: wraps a tree node with a `Group` node and appends or replaces the other nodes with a back reference node (such as `\1`). The number of the back reference is calculated according to the number of back references already in the expression. For instance, given the regular expression `abc`, the following solutions are generated: `(a)\1 bc`, `(a)\1c`, `(a)b\1c`, `(a)b\1`, `(a)bc\1`, `(ab)\1c`, `(ab)\1`, `(ab)c\1`, `(abc)\1`, `a(b)\1c`, `a(b)\1`, `a(b)c\1`, `a(bc)\1`, and `ab(c)\1`;
- **Optional**: has the same behavior as the *Star quantifier* transformation, but uses an optional node. For instance, given the regular expression `ab[hz]`, the solutions `a?b[hz]`, `ab?[Hz]`, and `ab[hz]?` are generated;
- **Extraction**: removes a node from the tree. This transformation is performed once for each node of the current solution, including both terminal nodes and function nodes (along with their operands). For instance, given the regular expression `a[bh][^k]$`, the solutions `[bh][^k]$`, `a[^k]$`, `a[h][^k]$`, `a[b][^k]$`, `a[bh]$`, and `[bh][^k]` are generated.

3.4 Randomly generating a solution

To generate a solution to start or restart the search, the algorithm uses two strategies. First, it uses a set of *n-grams* based on the sentences in the match list. A *n-gram* is a fragment of a phrase (that is, a set of characters) transformed into a regular expression that must match at least one sentence on the match list and no sentence on the unmatch list. A *n-gram* can be as small as one character and as big as the whole phrase, as long as it matches some entry of the match list and does not match any sentence on the unmatch list. The algorithm generates the list of all possible *n-grams* and consumes the entries of this list one at a time when a new starting solution is required.

Once the *n-gram* list is empty, the algorithm uses a *tree depth* parameter and generates a number between one and the parameter value to serve as the tree depth for the initial solution. Then, a `Concatenation` node is created, a leaf node is sampled and added as a branch. The leaf node can be a random character from the match char list, a negation based on a random character from the unmatch char list, a `Dot` operator, a `Start anchor` operator, or a `End anchor` operator. The leaf node is randomly picked from the available choices. Next, a new `Concatenation` node is created, added as the second branch of the former `Concatenation` node, a new leaf node is sampled and added as a branch for the new function node. The process is repeated $X - 1$ times, being X the randomly sampled tree depth.

3.5 Regular expression shrinker

The regular expression shrinker algorithm is intended to reduce the number of characters in a regular expression while keeping the same amount of matches in the match list and unmatches in the unmatch list. This algorithm performs transformations on each node of the syntax tree, trying to transform it into an equivalent node with fewer characters on its textual representation.

We expected that, when the search reached a local minimum, the shrinker might lead to a different point in the search space representing an equivalent regular expression but with a neighborhood amenable for improvement. The transformations applied by the shrinker are:

- **Remove redundant operators**: a regular expression with two or more `Start anchor` or `End anchor` operators is a valid expression, but excess operators are unnecessary and can be removed to reduce the size of the expression. For instance, `^^abc` and `abc$$` can be transformed, respectively, into `^abc` and `abc$`;
- **Remove duplicate values**: Nodes of type `List` and `Negative Char Set` may contain unnecessarily repeated values. This transformation removes these duplicate values. For instance, the regular expressions `abc[eefg]` and `a[^bbc]` can be transformed into `abc[efg]` and `a[^bc]`, respectively;
- **Merge repetitions**: when `Repetition` nodes are juxtaposed in the tree, they can be merged into a single `Repetition` node without affecting the number of matches and unmatches of the solution. For instance, `a{1}a{2}` can be transformed into `a{3}`;
- **Convert to repetition**: when two or more leaf or function nodes of types `Concatenation`, `List`, `Range`, `Group`, and `Negative Char Set` have the same values and are juxtaposed in the tree, they can replace by a `Repetition` node containing a single instance of the original nodes. For instance, the regular expressions `abcccccdef`, `[ad][ad]` and `[^z][^z]` can be transformed into `abc{5}def`, `[ad]{2}`, and `[^z]{2}`, respectively;
- **Merge quantifiers**: nodes with the `Star` or `Plus quantifier` and having the same operands can be merged into a single node. For example, the regular expressions `a+a+`, `b*b*` and `c+c*` can be transformed into `a+`, `b*`, and `c+`, respectively;
- **Simplify alternation**: three or more nodes of type `Alternation` can be replaced by a `List` or *Range* node. For instance, the regular expressions `a|e|h|o` and `a|b|c|d` can be transformed into `[aeho]` and `[a-d]`, respectively;
- **Remove duplicate alternation**: two or more nodes of type `Alternation` having the same operands can be removed without changing the number of matches and unmatches. For instance, the regular expression `ab|ab` can be transformed into `ab`;
- **Simplify ranges**: a `Range` node can be reduced to a representation that uses fewer characters and performs the same matches and unmatches. If the node has a single character as its operand, it can be replaced by a leaf node with the same value. For instance, the regular expressions `a[b-ed-h]c` and `[a-a]` can be transformed into `a[b-h]c` and `a`, respectively;
- **Convert range to shorthand**: a `Range` node containing values equivalent to a shorthand character class can be replaced by a leaf node containing the

shorthand character class. For instance, the regular expressions `[a-zA-Z0-9]` and `[0-9]` can be transformed into `\w` and `\d`, respectively;

- **Adding backrefs**: when two or more nodes have the same values and such values are more than three characters in size, the first node can be converted to a `Group` node containing the value, while the other nodes may be replaced by `Back-reference` nodes referring to the `Group` just added. For instance, the regular expressions `abcdefabc` and `defabcdefabcabc` can be transformed into `(abc)def\1` and `(def)(abc)\1\2\2`, respectively;
- **Simplify negation**: a `Negation` node having no value can be replaced by a `Dot` node affecting the matches and unmatches of the solution. For instance, the regular expression `a[^]bc` can be transformed into `a.bc`.

4 Experimental evaluation

To compare the local search algorithm proposed in this paper with the aforementioned exact and genetic programming algorithms, as well as solutions presented by software developers, we selected fifteen instances that were used in the previous experimental study carried out by Bartoli et al. [3]. These instances were extracted from the website https://alf.nu/RegexGolf.

The aforementioned website presents popular challenges for Regex Golf, having been used both by the algorithms and software developers that address the challenge. The comparison reported in the below uses the fitness function presented in Sect. 3, which was used in Bartoli el al.'s study [3]. Therefore, the results provided by human developers and by the selected algorithms are comparable.

4.1 Instances under evaluation

Table 7 shows the selected instances, the size of their match and unmatch lists, and the maximum score attainable by each instance. The *Long count v2* instance, mentioned in the Bartoli el al.'s study [3], was discarded because it was not available on the website.

4.2 Data collection

To obtain the results of the exact algorithm, we adapted the Python implementation of the algorithm proposed by Norvig [14] and executed it for each selected instance. The adaptations in the implementation were required because the original code did not *escape* characters that represent regular expression operators and might be present in match or unmatch lists. This problem occurred when we executed the algorithm upon the *Glob* instance. After fixing this problem, we obtained all regular expressions found by Norvig's algorithm for all instances.

The results of the genetic programming algorithm were obtained from the paper published by Bartoli et al. [3], which also presents the regular expressions developed by human beings that were compared to the genetic algorithm. However, these regular expressions could be obsolete, as years have passed since the

Table 7 Instances selected for the experimental study designed to evaluate the performance of the local search algorithm in comparison to a genetic algorithm and an exact algorithm

Instance name	Match list size	Unmatch list size	Weight	Maximum score
Plain strings	20	20	10	210
Anchors	21	21	10	210
Ranges	21	21	10	210
Backrefs	21	21	10	210
Abba	21	22	10	210
A man, a plan	19	21	10	190
Prime	20	20	15	300
Four	21	21	10	210
Order	21	21	20	210
Triples	21	21	30	630
Glob	21	21	20	420
Balance	32	32	10	320
Powers	11	11	10	110
Long count	1	20	270	270
Alphabetical	17	17	20	340

publication of the work. To find updated regular expressions developed by humans for the same instances, we performed a search on *Google* using the keywords *"regex*

Table 8 Best regular expressions developed by humans

Instance name	Regular expression	Fitness
Plain strings	`foo`	207
Anchors	`k$`	208
Ranges	`^[a-f]*$`	202
Backrefs	`(...).*\1`	201
Abba	`^(?!.*(.)\1)|ef`	195
A man, a plan	`^(.)[^p].*\1$`	177
Prime	`^(?!(..+)\1+$)`	286
Four	`(.)(.\1){3}`	199
Order	`^.{5}[^e]?$`	199
Triples	`00($|3|6|9|12|15)|4.2|.1.+4|55|.17`	596
Glob	`ai|c$|^p|[bcnrw][bnopr]`	397
Balance	`.{37}^(<(..(?!<.>$))*>)*$`	294
Powers	`^(?!(.(..)+)\1*$)`	93
Long count	`((.+)0\2+1){8}`	256
Alphabetical	`.r.{32}r|a.{10}te|n.n..`	317

golf answers". Table 8 presents the best regular expressions developed by humans we found on various websites for the selected instances.

4.3 Local search settings

The local search algorithm requires the assignment of values for two parameters: the depth of the syntax tree and the stop criteria. The *depth of the syntax tree* determines the number of nodes in the longest path from the root to a leaf that represents an initial solution to the local search algorithm. It is also used to generate a new random solution when the algorithm finds a local optimum solution and the shrinker algorithm is unable to produce a smaller, equivalent regular expression. The *stop criteria* is comprised of an evaluation limit and a time limit (a wall-clock *timeout*). Upon reaching one of such limits, the algorithm returns the best solution found so far as its results.

We carried out a set of experiments to determine the values for both parameters. One million evaluations of the fitness function was selected as the evaluation limit for the stop criteria, as well as a timeout of 10 min (twice the average execution time in our experiments). The depth of the syntax tree parameter was defined as the length of the largest sentence in the match list. Thus, the algorithm can generate random solutions for the local search up to the size of the largest sentence with which it needs to match.

The local search algorithm was executed 50 times for each instance. Table 9 presents the best regular expressions generated over all executions of the local search algorithm for each instance under analysis.

Table 9 The best regular expressions generated over 50 executions of the local search algorithm for each instance

Instance name	Regular expression	Fitness
Plain strings	`foo`	207
Anchors	`k$`	208
Ranges	`^[a-f]*$`	202
Backrefs	`ala|ea|l[op]|ro| \bm|ec|te|[m-r][^d]?.$`	172
Abba	`acr|nv|mi|te|st|z|.u|ph`	187
A man, a plan	`ten|ep|mu|oo|^[^nmpy]*$`	167
Prime	`x{33}|^xx.?$`	153
Four	`ev|de|lit|ara|o[mn]|[rs].$`	184
Order	`[cd]e|ch|fi|[op]s|lo`	190
Triples	`0{9}|819|015|012|003|900|54|009|06|2[34]|[^26].5..`	580
Glob	`ro|rr|lle|eat|up|ig|de|lo|fa|ow|co|gen`	382
Balance	`<>>>><<|<<>><>>|>>><<<<|><.>+>$`	239
Powers	`^..?$`	15
Long count	`0{4} 0001 0010 0011 0100 0101 0110 0111 1000 1001 1010 101`	212
Alphabetical	`ar te|rt r|at e[e-r]|r sn|ne t|nant |esen`	299

4.4 Comparison

Table 10 consolidates the results for the three algorithms under comparison, as well as solutions provided by software developers. The regular expressions developed by humans always outperform the algorithms. Thus, hereafter we compare only the algorithms with each other to see which comes closer to human-provided solutions. The bold values in Table 10 indicate the algorithm that produced the best result for a given instance. For the local search, we pick the best value over the 50 optimization rounds. The mean fitness and standard deviation for the solutions found by the local search over the 50 rounds are shown in the "$\mu \pm \sigma$" column.

Considering their best solutions, the three algorithms tied in the *Plain strings* and *Anchors* instances because those are solved by using the smallest solutions among the selected set. The local search algorithm achieved the best results for the *Ranges*, *Abba*, *A man, a plan*, *Order* (a draw with the results of Norgiv's exact algorithm), and *Alphabetical* instances, besides *Long count* and *Balance* instances, which are considered difficult instances. The local search was able to find a solution compatible with the best solution developed by humans for *Ranges*. The exact algorithm obtained the best results for the instances *Backrefs*, *Four*, *Order*, *Triples*, and *Glob* and the genetic programming algorithm obtained the best results for *Powers* and *Prime*, both considered difficult instances.

The most used regular expression operator in the results produced by the three algorithms is **alternation**, as it provides a "divide to conquer" strategy. Since it is

Table 10 The results of the exact algorithm (Norvig-exact), the genetic programming algorithm (Bartoli-GP), local search algorithm (best result, mean, and standard deviation over the 50 optimization rounds), and solutions built by software developers (Humans)

Instance name	Norvig-exact	Bartoli-GP	Local search		Humans
			Best result	$\mu \pm \sigma$	
Plain strings	**207**	**207**	**207**	207 ± 0.0	207
Anchors	**208**	**208**	**208**	199 ± 21.2	208
Ranges	191	195	**202**	197 ± 1.9	202
Backrefs	**175**	138	172	168 ± 0.6	201
Abba	186	184	**187**	187 ± 0.0	195
A man, a plan	157	136	**167**	144 ± 12.8	177
Prime	− 398	**188**	153	147 ± 6.1	286
Four	**192**	183	184	182 ± 0.3	199
Order	**190**	186	**190**	190 ± 0.0	199
Triples	**589**	430	580	529 ± 48.5	596
Glob	**392**	340	382	381 ± 9.9	397
Balance	− 1457	130	**239**	237 ± 0.3	294
Powers	− 1969	**51**	15	5.6 ± 32.2	93
Long count	189	191	**212**	212 ± 0.0	256
Alphabetical	294	132	**299**	299 ± 0.0	317

difficult to get a lean regular expression that performs the necessary matches and unmatches in the lists, it becomes easier to create regular expressions with few characters to perform some matches in the match list and no matches in the unmatch list, using the **alternation** operator to concatenate them into a single regular expression. Norvig [14] calls these small regular expressions *n-grams* and uses this strategy as the basis of his exact algorithm. Although the heuristic algorithms do not follow this strategy systematically, many of their results are similar to those produced by the exact algorithm.

The local search algorithm outperforms the other algorithms when there are a variety of different characters in both the match and unmatch lists. Instances with few unique characters, such as *Prime*, *Powers*, and *Long count*, led to poor performance because they fail to generate good neighborhoods and depend on character lists derived from the match and unmatch lists. The local search was only able to produce a good result for *Long count* because the regular expression shrinker algorithm was executed on the single sentence of the match list, which was used when the algorithm restarted. In the *Balance* and *Powers* instances, there were many *timeouts* due to the large sentences in the match list, some having more than 100 characters. This caused the neighborhood to be unnecessarily long, as many neighbors would not be better than the current solution and would only consume the fitness evaluation budget up to the timeout. The regular expression shrinker algorithm was relevant for instances with large sentences in their match list, such as *Prime*, *Triples*, *Powers*, and *Long count*. Executing only the local search algorithm would not be enough to produce the regular expressions listed in Table 9, as the search cannot generate some of the operation nodes that improve the solution, such as the function node `09` used in the *Triples* instance.

The distributions of fitness for the solutions found by the local search over the 50 optimization rounds are shown in the box-plots presented in Fig. 4. The *Anchors*, *Glob*, and *Powers* instances show a few outlier solutions with smaller fitness than the remaining solutions in the distribution. The *Triples* instance shows a genuine range of different solutions found by the search algorithm, its standard deviation ranging close to 9% of the mean fitness of the solutions. All other instances have

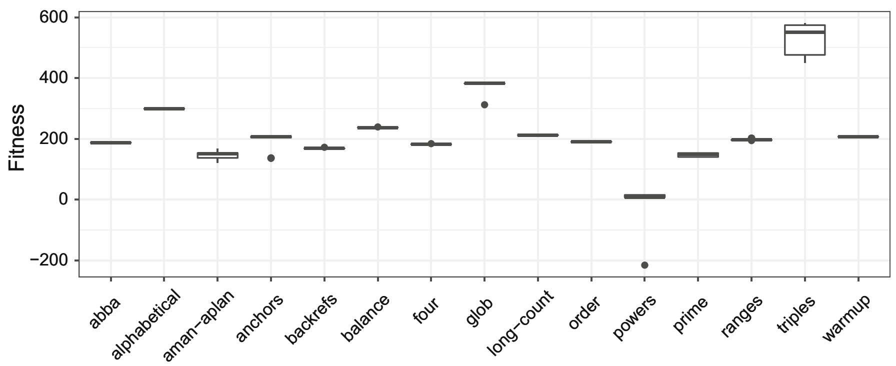

Fig. 4 Distributions of fitness for the solutions found by the local search on an instance basis

small standard deviation, which shows that the local search converges for most instances. We performed a Wilcox–Mann–Whitney test comparing the distributions of fitness found by the local search to the fitness found by Norgiv's algorithm and the published fitness of the genetic programming approach. All results were significantly different with $\alpha = 0.05$, except for *Plain Strings* (for which all algorithms yielded the same fitness) and *Order* (in the comparison between the exact algorithm and the local search).

4.4.1 Comparison to the genetic programming algorithm

The local search algorithm outperforms the genetic programming algorithm in eleven instances, loses in two, and ties in two instances. In the *Ranges* and *Long count* instances, the local search obtained better results due to using the **quantifier** operator added by the local search or the **convert to repetition** operator introduced by the regular expression shrinker algorithm. That is, the resulting regular expressions in both search algorithms were very close for these instances, but the shrinker managed to remove some characters, leading the local search algorithm to victory by a few points of difference.

In the instances *Four*, *Backrefs*, *Triples*, and *Glob* both algorithms presented solutions using the *alternation* operator several times. However, the regular expressions produced by genetic programming have redundant or unnecessarily repeated characters. For instance, the genetic programming algorithm generated solutions that present the same *n-gram* twice or a **group** operator that does not improve the regular expression match capability. These redundancies do not appear in the local search because the shrinker removes unused operators and improves the solution.

Another advantage of the local search algorithm over the genetic programming approach is execution time. Bartoli et al. report that the execution time of the genetic programming approach exceeds 60 minutes for almost all instances [3], while the local search runs in less than five minutes in average, with the exception of the *Powers* instance that exceeded the ten minutes timeout for all executions.

We have performed our experiments in a notebook powered by an Intel Core i7 9750H processor with 16 GB of RAM, while the genetic programming approach was executed in an Intel Xeon E5-2440 processor with 4 GB of RAM. According to a benchmark[2] our processor is 8.3% faster than the former. Therefore, the gain in processing time cannot be attributed only to a faster CPU, but in large part to a simpler algorithm using domain-specific evolutionary operators.

Bartoli et al. [3] also report results collected after running the genetic programming approach for thrice the initial number of generations. This search ties with the local search and Norvig's exact algorithm for the *Order* instance, maintains its superior results for the *Prime* and *Powers* instances, but remains outperformed for the other ones. On the other hand, this search required 5.2 times the processing time of the baseline genetic programming search.

[2] https://versus.com/en/intel-core-i7-9750h-vs-intel-xeon-e5-2440.

Finally, the main advantage of the genetic programming algorithm is its ability to traverse different points of the search space by using quite different regular expressions. This algorithm uses the *backref*, *lookahead*, and *lookbehind* operators efficiently, leading to a clear advantage for instances with few unique characters, such as *Prime* and *Powers*.

4.4.2 Comparison to the exact algorithm

The exact algorithm starts by generating *n-grams* of up to four characters. This limits its search capacity to the point of having some results represented by all sentences in the match list concatenated with the **alternation** operator. For these instances, the exact algorithm's strategy produces large regular expressions with low, often negative, fitness value.

As explained in Sect. 2, the exact algorithm is always able to generate a regular expression that represents a feasible solution for the challenge. However, the algorithm does not take into account the length of the resulting regular expression; it accounts only for the number of matches and unmatches performed in the related lists. This also leads to large regular expressions which are penalized by the fitness function. Since the exact algorithm does not generate regular expressions that have **quantifier** operators, the fitness in the *Prime*, *Balance*, and *Powers* instances, whose solutions rely on such operator, is impaired due to excessive usage of **alternation** operators.

These poor fitness values caused the local search algorithm to outperform the exact algorithm in eight instances, tie in three, and lose in four instances. In the *Backrefs* and *Four* instances, the local search lost by only three and eight points, respectively. This indicates that it might be possible to improve the neighborhood operators to allow the search moving to a smaller solution. On the other hand, the exact algorithm is very fast and finds solutions in a fraction of the number of fitness evaluations required by the local search.

4.4.3 Comparison to expressions developed by humans

The regular expressions reported in Table 8 were crafted and evolved for years by humans.[3,4] While it is not possible to determine the level of knowledge of the people who participated in the development of these regular expressions, we can safely suggest that considerable skill is required to develop them. The fitness value for all instances is quite close to the maximum possible score. Therefore, it might be difficult to find software developers who honed their regular expression creation skills to that point.

The local search algorithm finds the same regular expressions developed by humans for the *Plain string*, *Anchors*, and *Ranges* instances. However, these are the only instances on which humans do not win the local search. The difference in other instances is not very large, but the number of characters in regular expressions

[3] https://gist.github.com/jpsim/8057500.

[4] https://gist.github.com/Davidebyzero/9221685.

developed by software developers may be less than half of those in the best solution found by local search, such as in the cases of the *Backrefs*, *Long count*, and *A man, a plan* instances. The solutions proposed by humans for these instances are so complex that it is not possible to generate them using the present version of the neighborhood operators, which lack the ability to generate regular expression operators nested to other operators in a concise manner. This is the main limitation of the present implementation of the local search algorithm, if compared to human designed regular expressions.

4.5 Contributions of the components of the heuristic search approach

In this subsection, we analyze the contribution of the regular expression shrinker and neighborhood operators for the local search. This evaluation aims to find the operators that contributed most to the quality of the solutions found by the proposed algorithm.

4.5.1 The expression shrinker and its operators

The regular expression shrinker had a limited impact on the overall performance of the search: only 14.77% of its applications resulted in an improvement of the solutions under analysis. However, the contribution of the shrinker varies strongly from instance to instance. It ranges from less than 6% improvement in 10 out of 15 instances to 49.98% and 74.4% of its applications resulting in reductions for solutions of the *Powers* and *Prime* instances, respectively. The latter instances use a single character repeated multiple times. The shrinker replaced parts of the regular expressions by the character followed by a number of repetitions or suppressed redundant parts of the sequence, thus reducing the expressions and improving the quality of the solutions found by the search.

Table 11 shows the number of times the shrinker was activated for each instance (average over the 50 optimization rounds, in column **Count**), the number of times it reduced the subjected regular expression (average over 50 rounds, in column **Effective**), the percentage of times the shrinker improved a solution (the ratio of **Effective** by **Count**, in column **% Effec.**), and the number of times each shrinking operator was used (accumulated over the 50 rounds, in the remaining columns).

The **convert to repetition** (CR) operator was used 1,239,862 times, with a large concentration of uses for the *Prime* (10.7%), *Balance* (12.4%), *Powers* (51.9%), and *Long count* (22.1%) instances. The **remove duplicate values** (RD) operator was used 163,706 times, with a large concentration of uses for the *Powers* instance (87%). The **simplify negation** (SN) and **simplify ranges** operators complement the set of effective operators used by the shrinker with far limited contribution than the former ones.

Therefore, we observe that while the shrinker applies eleven different operators, only four of them had any impact on the optimization of the selected instances. Two of these operators are related to sequences of characters repeated over the regular expression, while the latter were simplifications of complex operations that might be built during the search process. We also see a concentration of successful

Table 11 Number of applications and effectiveness of the regular expression shrinker (average over the 50 optimization rounds) and the number of times its operators were successfully applied (accumulated over the 50 rounds)

Instance name	Count	Effective	% Effec.	CR	RD	SN	SR
Plain strings	721.0	23.7	3.29	18,420	150	1027	–
Anchors	76.6	2.8	3.63	122	37	94	–
Ranges	1142.0	47.5	4.16	9134	409	1609	3
Backrefs	32.7	0.9	2.75	58	18	23	–
Abba	79.4	2.8	3.53	121	46	83	–
A man, a plan	47.6	1.8	3.74	1271	10	68	–
Prime	2343.0	1743.0	74.40	132,187	11,481	1677	–
Four	55.1	1.7	3.01	112	36	41	1
Order	102.0	3.1	3.04	455	18	134	–
Triples	149.0	8.1	5.41	3919	107	204	–
Glob	32.4	1.6	5.00	681	41	5	–
Balance	444.0	132.0	29.70	153,410	3390	6	–
Powers	11,551.0	5774.0	49.98	642,902	143,039	1391	–
Long count	606.0	105.0	17.30	274,187	4253	16	–
Alphabetical	145.0	18.3	12.60	2883	671	6	–

CR stands for the “Convert to repetition” operator, **RD** is “Remove duplicate values”, **SN** stands for “Simplify negation”, and **SR** is “Simplify ranges”

applications on instances having a limited character set in their match and unmatch lists, which fosters repetitions and their subsequent simplifications.

4.5.2 The neighborhood operators

Table 12 shows the number of times each neighborhood operator applied by the local search improved a solution over the 50 optimization rounds on an instance basis. Only the eight most frequently successful operators are shown due to space restrictions. The acronyms for the neighborhood operators are **extraction** (EXT), **swap** (SW), **range** (RNG), **dot** (DOT), **star quantifier** (STAR), **concatenation** (CC), **negation** (NEG), and **back reference** (BR).

All optimization rounds considered, four operators complement Table 12: the **plus quantifier**, with 408 occurrences (0.02%) concentrated in the *Prime* and *Ranges* instances; the **start anchor** operator, with 186 occurrences (0.01%), most in the *Plain strings* and *Ranges* instances; the **optional** operator, with 163 occurrences (0.01%) focused in the *Prime* instance; and the **end anchor** operator, with 129 occurrences (0.01%) concentrated in the *Plain strings* and *Ranges* instances.

We observe a concentration on the operators that add, remove, and swap nodes using characters from the match and unmatch lists (**extraction**, **swap**, **range**, and **concatenation**). They correspond to more than 97% of the successful applications of neighborhood operators. We also observe that these operators are useful for

Table 12 Number of effective applications of each neighborhood operator during the local search accumulated over the 50 optimization rounds on an instance basis

Instance	EXT	SW	RNG	DOT	STAR	CC	NEG	BR
Plain strings	29,508	13,663	3218	25	1136	215	43	4
Anchors	3823	13,282	2512	52	83	23	10	–
Ranges	71,648	57,845	23,626	1080	5220	1704	1441	–
Backrefs	2021	6104	7649	15	33	16	27	–
Abba	5092	11,886	4553	547	80	19	5	–
A man, a plan	2127	8778	1715	130	73	37	124	–
Prime	87,045	30,490	–	9503	6598	632	–	560
Four	3136	7240	8226	6	57	3	–	1
Order	5222	8827	20,327	12	205	61	3	1
Triples	7257	20,035	14,986	113	387	166	–	–
Glob	2433	6162	4913	–	2	–	–	–
Balance	34,604	66,024	–	7307	2	–	–	–
Powers	1,286,066	–	–	2212	154	1	–	2
Long count	28,544	10,168	–	–	–	–	–	–
Alphabetical	7321	24,227	7225	–	2416	1	–	–
Total	1,575,847	284,731	98,950	21,002	16,446	2878	1653	568
Total (%)	78.68	14.21	4.94	1.05	0.82	0.14	0.08	0.03

EXT stands for “Extraction”, **SW** is the “Swap” operator, **RNG** stands for “Range”, **DOT** is the “Dot” operator, **STAR** is the star quantifier, **CC** is “Concatenation”, **NEG** is the “Negation” operator, and **BR** stands for the “back reference” operator

Only the top eight most frequently successful operators are shown due to space restrictions

almost all instances. The same applies to the **dot** and **star quantifier** operators, whose contribution is smaller but applied to a broad range of instances.

The remaining operators have effect on specific instances, such as the **negation** operator for the *Ranges* instance and the **back reference** operator for *Prime*. The *Ranges* instance is prone to the **negation** operator because it is formed by repetitions of character sequences whose boundaries can be marked by exclusion (that is, by listing the characters that cannot occupy those positions). The start and end anchors also play a role in delimiting the repetitions in the *Ranges* instance and inhibit matches with the *unmatch* list. On the other hand, *Prime* contains a single character repeated multiple times and back referencing parts of the sequence, as well as using the star and plus quantifiers, allows for more concise regular expressions with better fitness.

Table 13 shows the average contribution of each neighborhood operator in terms of fitness point on an instance basis. While the most frequently used operators (**extraction**, **swap**, **range**, and **concatenation**) are the most important to 6 out of 15 instances, the less common operators are paramount for the high-quality solutions found for other instances. For the sake of an example, the largest increase in fitness for the *Four* and *Order* instances is due to the **back reference** operator, even though

Table 13 Average improvement yielded by each neighborhood operator during the local search over the 50 optimization rounds on an instance basis

Instance	EXT	SW	RNG	DOT	STAR	CC	NEG	BR	Fitness
Plain strings	28.8	195.0	73.0	67.9	36.7	80.4	36.3	168.0	207
Anchors	8.1	16.2	20.9	51.6	35.1	104.0	41.5	–	208
Ranges	39.8	37.8	46.8	11.3	46.1	26.4	21.7	–	202
Backrefs	8.9	12.4	8.8	17.8	17.4	11.2	11.5	–	172
Abba	21.1	19.0	10.8	48.8	18.2	23.5	44.2	–	187
A man, a plan	5.8	11.4	11.5	18.4	17.3	28.7	8.1	–	167
Prime	11.1	115.0	–	31.7	21.3	21.1	–	64.2	153
Four	18.7	20.9	14.7	3.0	23.0	58.7	–	136.0	184
Order	13.5	32.4	17.9	15.5	27.7	26.7	53.7	86.0	190
Triples	21.1	46.7	38.2	34.0	48.8	43.0	–	–	580
Glob	34.2	29.1	21.4	–	20.0	–	–	–	382
Balance	40.8	42.1	–	30.0	8.0	–	–	–	239
Powers	8.2	–	–	14.8	14.8	9.0	–	6.0	15
Long count	7.1	236.0	–	–	–	–	–	–	212
Alphabetical	8.4	43.7	16.7	–	20.0	19.0	–	–	299

Only the top eight most frequently successful operators are shown due to space restrictions. **EXT** stands for "Extraction", **SW** is the "Swap" operator, **RNG** stands for "Range", **DOT** is the "Dot" operator, **STAR** is the star quantifier, **CC** is "Concatenation", **NEG** is the "Negation" operator, and **BR** stands for the "back reference" operator. The **Fitness** column shows the fitness of the best solution found by the local search

this operator was successful only once for each of these instances. For six other instances, the most important operators in terms of fitness gains are not reported in Table 13: the **optional** operator provides the best gain for the *Anchors* and *Ranges* instances (despite being successful in only 3 and 14 uses, respectively); the **end anchor** operator provides the best gain for *Backrefs*, despite being successful only twice, and for the *A man, a plan* instance (four successful applications); the **start anchor** operator yielded the best gain for *Triples* (over nine successful applications); and, finally, the **plus quantifier** produced the best gain for *Abba* on a single successful application.

Therefore, we observe that while the operators that grow or shorten the regular expressions by adding or removing characters are relevant and required for the evolutionary process to work, domain-specific operators are responsible for the largest improvements in a number of instances. These operators allowed the local search to outperform the genetic programming approach used as a baseline for our comparisons and should be added and applied frequently to a regular expression generation framework, be it driven by a local search or a more complex algorithm.

5 Conclusion

In this paper a new heuristic search technique was presented to generate regular expressions for problems proposed in the Regex Golf challenge. This new heuristic is based on a local search algorithm combined with a regular expression shrinker. The Regex Golf challenge was chosen for its emphasis on finding the smallest regular expression possible for a given problem, as well as presenting examples and counterexamples of the expected matches.

An experimental study was designed to evaluate the proposed local search in comparison to the state-of-the-art algorithms designed for the same problem. Fifteen instances of the Regex Golf challenge were selected and the local search algorithm was executed 50 times for each instance. This experiment compared the results generated by the local search algorithm with the other two algorithms developed for the challenge: an exact algorithm and a Genetic Programming algorithm. The local search algorithm found better solutions in most of the selected instances in less time than the Genetic Programming algorithm. The code used in the experimental studies is hosted on GitHub and can be accessed at the URL https://github.com/andrefarzat/regex-golf.

While the Regex Golf challenge rewards shorter regular expressions, this is hardly the single aspect that might be relevant for all software developers. Shorter regular expression may be harder to read by less experienced developers, as can be seem in some examples in Table 8. Such expressions may become a problem for maintenance and evolution. Software projects may also require efficient regular expressions and such property is not clearly related to their size: parsers may check simple, though larger expressions against a text snippet faster than small expressions using complex operators, such as quantifiers or back-reference. Other projects may need regular expressions that are compatible with more than one programming language or parsing library, thus limiting the operators to be used while building these expressions.

Furthermore, a balance of these aspects may be required by certain projects, leading to multi-objective optimization. While the local search algorithm may be a good choice to find regular expressions for a single fitness function, such as the one designed for the Regex Golf challenge, genetic algorithms are much more frequently used to find solutions in multi-objective settings. Therefore, adapting the general genetic programming framework for the regular expression generation problem with the domain-specific mutation operators and the shrinking algorithm proposed in this paper may be an interesting future work.

Acknowledgements The authors would like to thank CNPq (Conselho Nacional de Pesquisa) and CAPES (Coordenação de Aperfeiçoamento de Pessoal de Nível Superior) for their support for this research project. The authors also acknowledge the contributions given by the peer reviewers to improve this paper.

References

1. L. Araujo, Genetic programming for natural language processing. Genet. Program Evol. Mach. **21**, 1573–7632 (2019)
2. A. Bartoli, G. Davanzo, A. De Lorenzo, M. Mauri, E. Medvet, E. Sorio, Automatic generation of regular expressions from examples with genetic programming, in Proceedings of the 14th Annual Conference Companion on Genetic and Evolutionary Computation. ACM, pp. 1477–1478 (2012)
3. A. Bartoli, A. De Lorenzo, E. Medvet, F. Tarlao, Playing regex golf with genetic programming, in Proceedings of the 2014 Annual Conference on Genetic and Evolutionary Computation. ACM, pp. 1063–1070 (2014)
4. A. Bartoli, A. De Lorenzo, E. Medvet, F. Tarlao, Active learning approaches for learning regular expressions with genetic programming, in Proceedings of the 31st Annual ACM Symposium on Applied Computing. ACM, pp. 97–102 (2016)
5. A. Bartoli, A. De Lorenzo, E. Medvet, F. Tarlao, Can a machine replace humans in building regular expressions? A case study. IEEE Intell. Syst. **31**(6), 15–21 (2016)
6. A. Bartoli, A. De Lorenzo, E. Medvet, F. Tarlao, Inference of regular expressions for text extraction from examples. IEEE Trans. Knowl. Data Eng. **28**(5), 1217–1230 (2016)
7. F. Brauer, R. Rieger, A. Mocan, W.M. 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, pp. 1285–1294 (2011)
8. A. Cetinkaya, Regular expression generation through grammatical evolution, in Proceedings of the 9th Annual Conference Companion on Genetic and Evolutionary Computation. ACM, pp. 2643–2646 (2007)
9. R.A. Cochran, L. D'Antoni, B. Livshits, D. Molnar, M. Veanes, Program boosting: program synthesis via crowd-sourcing. SIGPLAN Not. **50**(1), 677–688 (2015). https://doi.org/10.1145/2775051.2676973
10. B. Cody-Kenny, M. Fenton, A. Ronayne, E. Considine, T. McGuire, M. O'Neill, A search for improved performance in regular expressions. CoRR arXiv:1704.04119 (2017)
11. A. Gonzalez-Pardo, D. Camacho, Analysis of grammatical evolutionary approaches to regular expression induction, in 2011 IEEE Congress of Evolutionary Computation (CEC). IEEE, pp. 639–646 (2011)
12. E. Larson, A. Kirk, Generating evil test strings for regular expressions, in 2016 IEEE International Conference on Software Testing, Verification and Validation (ICST). IEEE, pp. 309–319 (2016)
13. Y. Li, R. Krishnamurthy, S. Raghavan, S. Vaithyanathan, H. Jagadish, Regular expression learning for information extraction, in Proceedings of the Conference on Empirical Methods in Natural Language Processing. Association for Computational Linguistics, pp. 21–30 (2008)
14. Norvig, P.: Regex golf with peter norvig. https://www.oreilly.com/learning/regex-golf-with-peter-norvig (2014). Visitado em: 26-05-2018
15. R. Rastogi, S. Akash, G. Shobha, G. Poonam, D. Pratiba, A. Singh, Design and development of generic web based framework for log analysis, in 2016 IEEE Region 10 Conference (TENCON). IEEE, pp. 232–236 (2016)
16. T. Wu, W.M. Pottenger, A semi-supervised active learning algorithm for information extraction from textual data. J. Am. Soc. Inf. Sci. **56**(3), 258–271 (2005)
17. J. Zhang, C. Seifert, J.W. Stokes, W. Lee, Arrow: Generating signatures to detect drive-by downloads, in Proceedings of the 20th international conference on world wide web. ACM, pp. 187–196 (2011)