



A Benchmark Production Tool for Regular Expressions

Angelo Borsotti¹, Luca Breveglieri¹(✉), Stefano Crespi Reghizzi^{1,2},
and Angelo Morzenti¹

¹ Politecnico di Milano, 20133 Milan, Italy
angelo.borsotti@mail.polimi.it,

{luca.breviglieri, stefano.crespireghizzi, angelo.morzenti}@polimi.it
² CNR-IEIIT, 20133 Milan, Italy

Abstract. We describe a new tool, named *REgen*, that generates regular expressions (RE) to be used as test cases, and that generates also synthetic benchmarks for exercising and measuring the performance of RE-based software libraries and applications. Each *group* of REs is randomly generated and satisfies a user-specified set of constraints, such as length, nesting depth, operator arity, repetition depth, and syntax tree balancing. In addition to such parameters, other features are chosen by the tool. An RE group may include REs that are ambiguous, or that define the same regular language but differ with respect to their syntactic structure. A *benchmark* is a collection of RE groups that have a user-specified numerosity and distribution, together with a representative sample of texts for each RE in the collection. We present two generation algorithms for RE groups and for benchmarks. Experimental results are reported for a large benchmark we used to compare the performance of different RE parsing algorithms. The tool *REgen* and the RE benchmark are publicly available and fill a gap in supporting tools for the development and evaluation of RE applications.

Keywords: Regular expression generation ·
Benchmark for regular expressions · Regular expression tool

1 Introduction

Regular expressions (RE) are a widely applied language definition model. Actually the term RE refers not only to the Kleene formal model, but also to programming notations like that of the *Java.regex* library, which we generically call *technical RE* [2]. Many algorithms and software libraries using REs, referred to as *RE SW*, continue to be developed for many purposes, e.g., string matching, text editing, code inspection, intrusion detection, etc. Our focus is on those RE SW that not just recognize regular sets of strings, but also assign them a structure, i.e., do parsing. Quite often, the REs in such applications are ambiguous, therefore a string can be parsed in many different ways.

The development and evaluation of RE SW would benefit from using large collections of REs, but surprisingly we could not find any available one, which we needed for measuring the performance of different RE parsing algorithms, including our own [1]. This motivated the design and implementation of a new tool, called *REgen*, to produce customizable RE collections, as well as string samples of the corresponding regular languages. By using the tool, we synthesized a large RE benchmark, described in Sect. 4, that we used for an objective comparison of RE parsing algorithms. However, *REgen* is not biased towards a specific RE SW and is planned as a general-purpose customizable tool.

Related Work. We briefly consider two research directions, and why they could not be exploited here. First, string generation programs driven by a given context-free (CF) grammar have been used, at least since the '80s, to produce input data for testing compilers, e.g., [3], and other software systems, e.g., [7]. Such generator algorithms are typically guided by simple criteria, such as the minimal coverage criterion, ensuring that all CF rules are used to produce a minimal set of strings. Of course, most produced strings, though syntactically correct, are illegal input data for the SW system under test; they may help debugging, but they are not intended for evaluating SW execution performance. In practice, compiler evaluation relies on big hand-written language-specific benchmarks, e.g., SPEC for the C language. Our tool is also driven by a CF grammar, namely the meta-grammar of REs, but it essentially differs because *REgen* has a meta-level and a terminal level. At meta-level, each output string is a *generated RE (GRE)*, and is used at terminal level to drive the text generation component of *REgen*, which creates a set of *generated texts (GT)*.

Second, our project objectives differ from those of more theoretical research on RE enumeration, although enumeration algorithms may seem to address similar requirements as our meta-language level. The RE enumeration procedure in [5] is actually targeted at enumerating regular languages rather than REs, thus it carefully avoids generating two equivalent REs, whereas for a tool like ours, generating also equivalent GREs is acceptable and even desirable, e.g., to evaluate how the structure of equivalent REs impacts on SW performance.

Paper Contributions. The design of a new tool for generating REs and the corresponding texts raised various technical questions for which existing systems did not offer ready solutions. Since the range of RE SW is ill-defined and open, we decided that our tool should be *customizable*, so that the future users of *REgen* will be able to match the features of the GRE benchmarks, with the typical properties of the REs occurring in, say, a code inspection versus a web searching application. For instance, the REs used in text searching typically have a very small nesting level for the star (more generally, repetition) operator. Unfortunately, we could not find any established classification of REs oriented towards their application domains, and we had to examine the following question: how to select a not too large set of parameters that would allow to classify REs according to flexible empirical criteria. We started from the classical parameters such as RE length and star (better, repetition) depth, then we added and experimented others, later described in the paper, such as the maximal arity of an operator (*or*

and *dot*). Other parameters specify that certain combinations of operations are forbidden. Another interesting structural parameter is whether the RE tree is balanced or not; we found that it is better to separate the generation algorithms for the two cases. Clearly, the RE parameters we have identified and included in the current tool are just a well-thought initial model, which is open to revision and tuning to better adjust to unanticipated use cases.

The main output of our research is the specification and implementation of the *REgen* tool, which actually includes two generator levels, GRE and GT. The GRE generator is the innovative and major component, while the GT generator is indispensable but more traditional. More precisely, the tool consists of three parts. Part one outputs a single GRE compliant with user-specified or default parameter values, thus, by repeated application with fixed values, it generates a random *group* of GREs that fulfill identical constraints. Part two repeatedly applies Part one, each time with stepping parameter values, and thus outputs a random collection of GRE groups, to be called a *benchmark*, which has user-specified features. Part three works on a given GRE and produces a representative sample of GTs, with a user-specified length and numerosity.

By using *REgen*, a large RE benchmark has been created, and we report the main measurements of the GREs and GTs, and the tool execution times. Moreover, we have further analyzed the GREs for the following important properties that are not controlled by the current input parameters: the ambiguity of GREs, and how many GREs present in the benchmark define nondisjoint languages.

The Java code of *REgen* and the benchmark are available at the URL <http://github.com/FLC-project/REgen>, and they will hopefully serve software developers and formal language researchers.

Paper Organization. Section 2 contains the basic definitions of GREs and their operators and abstract syntax trees, followed by the input parameters that constrain tree shape and contents. Section 3 outlines the generation algorithms. Section 4 presents some quantitative aspects of the benchmark generated. The Conclusion hints to future developments.

2 Basic Definitions

The notions we use are standard in formal language theory. First, we define the RE family to be considered. A *generated RE (GRE)* is a string over an alphabet $\Omega = M \uplus \Sigma$, where set M contains the metasymbols and set Σ , called the text alphabet, contains the terminals that may occur in a *generated text (GT)*. The GRE structure is better represented by an *abstract syntax tree (AST)*. Table 1 and the examples in Fig. 1 should give a sufficient idea of the language of GREs, denoted L_{GRE} , and also of the ASTs. To denote a GRE we use letters such as e, f, e_1, \dots ; the empty string is ε . Notice that rule 5 in Table 1 defines the *repetition* operation, which contains rules 5.1, 5.2 and 5.3 as special cases.

Definition 1 (AST). *Let $e \in L_{GRE}$, the corresponding AST, denoted e_T , has the structure and node labels specified in Table 1.* \square

Table 1. Components and rules of a GRE and the corresponding AST tree.

#	Rules defining GRE e	Node label of the corresponding AST
1	$e = \varepsilon$	Leaf node with label ε (empty string)
2	$e = a \in \Sigma$	Leaf node with label a (terminal character)
3	$e = e_1 \mid e_2 \mid \dots \mid e_k \quad k \geq 2$	Inner node with label “ ” and k children
4	$e = e_1 \cdot e_2 \cdot \dots \cdot e_k \quad k \geq 2$	Inner node with label “.” and k children
5	$e = e_1^{i,j} \quad 0 \leq i < j \leq \infty$	Inner node with label i, j and one child e_1
5.1	$e = e_1^*$ same as $e_1^{0, \infty}$	Inner node with label “*” and one child e_1
5.2	$e = e_1^+$ same as $e_1^{1, \infty}$	Inner node with label “+” and one child e_1
5.3	$e = e_1^?$ same as $e_1^{0, 1}$	Inner node with label “?” and one child e_1

See the examples in Fig. 1. An internal node of an AST belongs to the types:

$$type = \left\{ \overbrace{\{ union \mid, concatenation \cdot, \}}^{non\text{-}unary}, \overbrace{\{ repetition(\min \dots \max) \}}^{unary} \right\}$$

As said, the *iterators* star “*” and cross “+”, and the *optionality* operator “?”, are subcases of repetition nodes, with the values already shown in Table 1.

The language defined by a GRE e is denoted $L(e) \subseteq \Sigma^*$, and to prevent confusion, its sentences are called *generated texts* (*GT*). The GREs e and f (Fig. 1) are *weakly equivalent* since $L(e) = L(f)$. Yet, since GRE f assigns to the text $abab$ a syntax tree (ST) different from the two trees assigned to the same string by GRE e (in Fig. 1, middle right), the two GREs e and f are not interchangeable when they are used for string matching or searching.

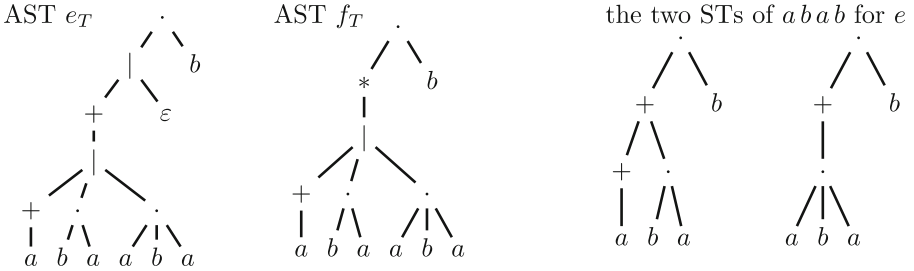
In Table 2 we list and describe the parameters and choices the user may enter, to customize the collection of GREs produced by *REgen*; some parameters are illustrated in Fig. 1 (bottom). The current selection of parameters tries to balance the complication of having too many generation parameters and the flexibility needed for tailoring GREs to specific applications.

3 The RE Generator

The goal of the *RE generator* is to produce a set of possibly very many GREs that match the parameter values specified by the user or by default. This is achieved through iteratively executing a procedure that generates random REs. Such a procedure is designed to produce, with a very high probability, a distinct RE at each invocation, so as to minimize the likelihood of discarding duplicates.

More precisely, the RE generator produces a set of ASTs, called a *group*, such that for all ASTs in the same group, the *primary* parameters of Table 2 take the same values, e.g., all ASTs are balanced or all are unbalanced. However, as it would be detrimental that all ASTs have the same frontier length, we decided that the number φ of leaves is not fixed, but that it ranges in the same interval.

$$e = ((a^+ | ba | aba)^+ | \varepsilon)b \qquad f = (a^+ | ba | aba)^* b$$



<i>parameter types for AST</i>	<i>parameter values for the AST e_T</i>
text alphabet size	$\alpha = 2$ (alphabet $\Sigma = \{a, b\}$)
frontier length	$\varphi = 8$ (frontier $abababab\varepsilon b$)
tree depth	$\tau(e_T) = \max(5, 5, 5, 5, 5, 5, 2, 1) = 5$
balanced or unbalanced	$\beta = \text{unbalanced}$
associative-operator degrees (in left-to-right top-down order)	$\delta_\cdot = 2 \quad \delta_+ = 2 \quad \delta_+ = 3 \quad \delta_\cdot = 2 \quad \delta_\cdot = 3$ $\delta = \max(\delta_\cdot, \delta_+) = 3$
repetition depth (paths listed from left to right)	$\rho = 2 \quad \rho = 1 \quad \dots \quad \rho = 1 \quad \rho = 0 \quad \rho = 0$
forbidden (parent, child) pair	$(\cdot,)$, $(\cdot, +)$ (\cdot, \cdot) , (\cdot, \cdot) $(\text{unary}, \text{unary})$

Fig. 1. Top: GREs e and f (a and b are arbitrary terminals). Middle left: the ASTs e_T and f_T of e and f . Middle right: the syntax trees in $T(e)$ of the text $abab$, where in the left ST the higher node “+” has two children, since there are two iterations. Bottom: values of the main parameters of e_T ; in the last row, the (parent, child) pairs that do not occur in e_T are listed as forbidden.

Thus, a *group* is qualified by the following parameter values. The text alphabet Σ (thus its cardinality $\alpha = |\Sigma|$) and the balance parameter β have fixed values. The tree depth τ ranges over an interval (min and max may coincide), and δ_\cdot has a fixed value, which specifies the interval $[2 \dots \delta_\cdot]$; the case of δ_+ is identical. In the GREs, we rule out the occurrence of immediately nested unary operators, since they are idempotent (in an obvious sense) and it is unlikely that any RE SW needs to discriminate between the sub-REs, say, $(e)^*$ and $((e)^*)^?$.

Though two ASTs in the same group have the same parameters, they may differ for the choice of internal node types and labels. To produce a wider variety of ASTs, the generator is repeatedly invoked with different parameter settings, thus resulting in a set of groups, which we call a *collection*.

First, we describe at a high-level the RE generation algorithm, which is actually split into two parts for balanced and unbalanced trees.

Table 2. Main GRE parameters to drive the generator.

primary parameters

text alphabet size $\alpha = |\Sigma|$. The leaf labels of an AST are in set $\Sigma \cup \{\varepsilon\}$.

frontier length $\varphi \geq 1$. It is the number of AST leaves, including the ε ones.

tree depth $\tau \geq 0$. It is the maximal path length from AST root to frontier.

balanced or unbalanced β (boolean). It specifies whether in an AST all the paths from root to frontier have the same length.

secondary parameters

associative-operator degree $\delta, \delta_1 \geq 2$. They denote the max bounds on the numbers of child nodes of a concatenation and a union node, respectively. We set $\delta = \max(\delta, \delta_1)$. In Fig. 1 their values are listed for each node.

repetition depth $\rho \geq 0$. For a path from node to leaf, it specifies the number of repetition nodes that occur on that path.

forbidden parent-child (relation) This binary relation in $type \times type$ specifies the pairs of parent-child node types that must not occur in an AST.

Balanced Tree Generation Algorithm. First, we state and prove a property, used in the algorithm, about the number of leaves in a balanced tree. In the coming discussion, we set $\delta = \max(\delta, \delta_1)$.

Proposition 1 (relation between leaf number and operator degree).

Consider the root-to-leaf paths in a balanced tree. Define the (non-empty) set $\{\langle i, n_i \rangle \mid 1 \leq i \leq \delta\}$, where $n_i \geq 0$ is the number of degree- i nodes on a given such path; therefore, the tree depth is $\tau = \sum_{1 \leq i \leq \delta} n_i$. If such a set is the same on all root-to-leaf paths, then the following relation holds for the tree leaf number:

$$\varphi = 1^{n_1} \cdot 2^{n_2} \cdot \dots \cdot \delta^{n_\delta} = \prod_{1 \leq i \leq \delta} i^{n_i}$$

□

Proof. The proof is by induction on the tree depth τ .

Base Step. Consider the (elementary) tree with a single node, thus $\tau = 0$ and $\varphi = 1$. Then, $n_i = 0$ for $1 \leq i \leq \delta$, and the above relation trivially holds.

Inductive Step. For any k with $1 \leq k \leq \delta$, consider $k \geq 1$ trees t_1, \dots, t_k that have equal depth $\tau \geq 1$ and the same set $\{\langle i, n_i \rangle \mid 1 \leq i \leq \delta\}$ of numbers n_i of degree- i nodes on any root-to-node path. By the inductive hypothesis, all such trees have the same leaf number $\varphi = \prod_{1 \leq i \leq \delta} i^{n_i}$. Build a (balanced) tree \hat{t} with a new root of degree $k \geq 1$ and append the trees t_1, \dots, t_k to this root. For tree \hat{t} , the set of the numbers \hat{n}_i of degree- i nodes on any root-to-node path, i.e., $\{\langle i, \hat{n}_i \rangle \mid 1 \leq i \leq \delta\}$, is identical on all such paths, with $\hat{n}_k = n_k + 1$

and $\hat{n}_i = n_i$ for every $i \neq k$. By construction, the leaf number of \hat{t} is $\hat{\varphi} = k\varphi = k \cdot 1^{n_1} \dots \delta^{n_\delta} = 1^{n_1} \dots k \cdot k^{n_k} \dots \delta^{n_\delta} = 1^{n_1} \dots k^{n_k+1} \dots \delta^{n_\delta} = \prod_{1 \leq i \leq \delta} i^{\hat{n}_i}$. Thus, tree \hat{t} satisfies the above relation. \square

We outline the recursive procedure that builds an AST, in the top-down order:

1. Randomly generate a factorization of parameter φ according to Proposition 1, i.e., generate a set of pairs $\langle \text{node-number, degree-value} \rangle$:

$$\left\{ \langle i, n_i \rangle \mid (1 \leq i \leq \delta) \wedge (n_i \geq 0) \wedge \left(\tau = \sum_{1 \leq i \leq \delta} n_i \right) \wedge \left(\varphi = \prod_{1 \leq i \leq \delta} i^{n_i} \right) \right\} \quad (1)$$

2. In any order, build a child node, then recursively build the child subtrees.

As the algorithm recurs down on a root-to-leaf path, it carries over as a procedure parameter the list of the degrees chosen for the nodes created on that path.

We discuss some special cases occurring in the preceding algorithm. Since the range of values of the secondary parameters δ , and δ_1 is typically quite restricted, for certain values of the primary parameter φ there may not be any factorization (1); for instance, when $\varphi = 69 = 3 \times 23$ and $\delta < 23$. In such cases, the following heuristics is applied: parameter φ is decreased by a few units, then a balanced tree is built and is adjusted to the original value of φ by attaching the few missing nodes to the tree bottom branches. Therefore, all the generated ASTs are balanced and every root-to-leaf path comprises the same number of nodes for any degree, except possibly for the two last tree levels. We note that such adjustments are possible because an exact compliance with the parameter values is not required for a benchmark generator. On the other hand, a formal enumeration algorithm, e.g., [5], is subjected to much stricter requirements.

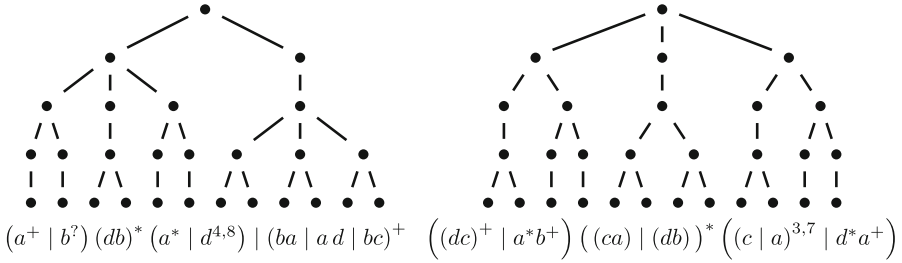
Unbalanced Tree Generation Algorithm. Unbalanced trees are also generated top-down, by means of a recursive procedure that combines the primary parameters for the frontier length φ and tree depth τ , thus achieving a large variety of tree shapes. The procedure repeatedly executes the following two phases:

1. For an internal node, starting from the root, randomly elect one of the child nodes as *distinguished*, which will be the root of the deepest subtree that has the maximum number of leaves. The total leaf number N (initially $N = \varphi$ for the root of the whole tree) is partitioned, and a fixed fraction of leaves, equal to $\lfloor \frac{N}{F} \rfloor$ for a certain F , is assigned to such a child node, while the remaining $N - \lfloor \frac{N}{F} \rfloor$ leaves are randomly distributed to the other siblings.
2. Iterate phase 1 at the next tree level. Notice that the number of leaves assigned to the current distinguished child node decreases with the distance of the node from the root, according to a geometric progression.

To satisfy the tree depth parameter τ , the equality $F = \sqrt[\tau]{\varphi}$ must hold. The maximum node degree δ must agree with F as well, hence also with the frontier length φ and the tree depth τ from which F derives, to ensure that the distinguished child node is assigned a leaf number greater than all those of the other siblings. Since the number of siblings is at most $\delta - 1$ and in total they

have $N - \lfloor \frac{N}{F} \rfloor$ leaves, for the above condition to be satisfiable, this (properly rounded) inequality must hold: $\frac{N}{F} > (N - \frac{N}{F}) / (\delta - 1)$. It follows $\delta > F$. This constraint is easily satisfied in all the practical cases. For instance, with $\varphi = 100$ and $\tau = 5$, it holds $F = 2.512$, and the maximum node degree is constrained to $\delta > 2$. Therefore, all the generated ASTs have one or more root-to-leaf paths of the desired length, and all the other paths are shorter. Moreover, the degree of all the nodes is within the limit δ .

$$\alpha = 4 \quad \beta = \textit{balanced} \quad \varphi = 12 \quad \tau = 4 \quad \delta = 3 \quad (\text{in (1)} \quad n_1 = 1 \quad n_2 = 2 \quad n_3 = 1)$$



$$\alpha = 5 \quad \beta = \textit{unbalanced} \quad \varphi = 13 \quad \tau = 4 \quad \delta = 3 \quad (F = 1, 8988)$$

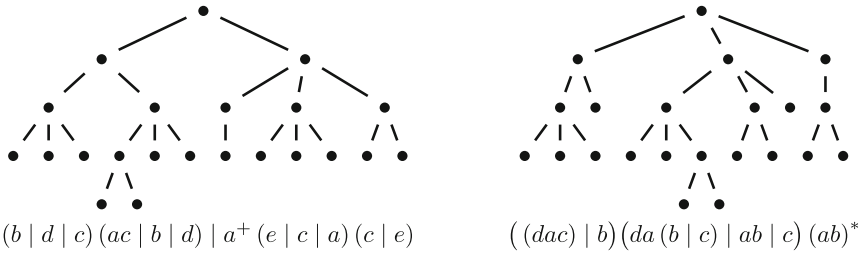


Fig. 2. Sample REs in two GRE *groups* identified by their primary parameters. The ASTs are simplified to show only the levels and highlight the balanced (top) vs. unbalanced (bottom) structure.

Group Production. As said, the GREs of a *group* are obtained by repeatedly invoking the random generation procedure with fixed primary parameter values. Due to randomness, it may happen that two identical GREs are produced, and a check is performed to avoid duplicates in the same group. We have experimentally found that duplications occur in a negligible number of cases and do not affect the generation speed. Figure 2 shows a few GREs with their ASTs, generated for two *groups* identified by the listed parameter values.

In the current version of *REgen*, the RE generator does not have any way to enforce that the generated RE is ambiguous or not. Such a control, if required, is currently performed off-line by another part of the tool, see later. We have found that a significant portion of GREs is ambiguous, typically about 60%.

Printing GREs as Formulas. The ASTs must be eventually converted into a textual form suitable for the intended RE SW. We do not describe the simple conversion, and just comment on the introduction of parentheses into a GRE. Parentheses serve two purposes: to enforce an operator evaluation priority and, for *technical* REs such as Posix, also to identify the sub-REs that have to be matched with the source text (this use is known as *capturing*). When converting an AST into a formula, *REgen* always prints the priority parentheses. In addition, if a sub-RE has to be “captured”, an embracing parenthesis pair is printed, even if it is unnecessary for priority. To know when, the new boolean parameter *capturing* is affixed to all the internal AST nodes.

3.1 Benchmark Production

In our setting, a (synthetic) *benchmark* is a large set of GREs, each one accompanied by a set of texts. More precisely, a benchmark is structured as one or more *collections* of groups, each *group* being a randomly generated set of REs that have the same user-specified primary parameters.

Since benchmarks can be generated for different and unforeseen RE SW, it would be unwise to fix any general properties for benchmarks. The case of a benchmark of broad interest, used to evaluate RE parsing SW, is described in detail in Sect. 4. The benchmark comprises many groups of GREs of increasing length, and each GRE is accompanied by a set of texts of increasing length. The wide range of GRE lengths allowed us to measure and plot the time an RE parsing algorithm takes to “compile” an RE, as a function of its length. Moreover, the text parsing time depends on both the text length and the RE length; to compare such times, our benchmark contains a large and diversified corpus of GREs and texts.

Notice that generating a few short REs and texts is simple, but doing so for long and significantly different REs and texts may run into time and memory limits. The naive approach based on enumerating REs by increasing length, and then selecting the few of them that fit the given criteria, is too slow to be feasible. In other words, an ideal requirement for an efficient generator is that only the REs that fit the benchmark objectives be generated, thus avoiding the production of REs that would be later discarded. Our tool approaches such an ideal and produces tolerably few unfit REs.

Text Generation Algorithm. Given an RE e , our tool randomly computes one or more generated texts $GT\ x \in L(e)$ by two simple procedures. The first procedure has two parameters, the RE e and the length ℓ of the text to be generated, and it produces just one GT x with length $|x| \approx \ell$. The second procedure is used to generate many GTs organized into a finite collection $\mathcal{C} = \{G_1, G_2, \dots, G_g\}$ of *text groups*, i.e., $G_i \subset L(e)$, each group containing about the same number of texts. The length of the text increases from one group to the next, as specified by one of the input parameters, which are the following: the RE e , the number g of groups, the group size $|G_i|$, and the *step* $s > 0$. Then, each GT group comprises GTs with approximately the following lengths:

$$\forall x \in G_1 \quad |x| \in [0 \dots s] \quad \forall i \geq 2 \quad \forall x \in G_i \quad |x| \in [(i-1) \cdot s + 1 \dots i \cdot s]$$

In this way, a user can customize the text lengths of a GT collection and so produce groups that include a statistically significant sample of texts within specified length intervals. Concerning the GT generation algorithm, it suffices to say that it operates in two phases:

1. Given a GRE e , encoded by the AST, the generator tabulates, for each subtree corresponding to a sub-RE e_j , the min and max lengths of a text in $L(e_j)$.
2. For each length ℓ , the GTs are top-down recursively computed by visiting their inner nodes and, for each node, by randomly partitioning the text length and distributing the resulting parts to its child nodes; details are omitted.

Last, notice that the GT collection is neither intended nor guaranteed to cover all the possible choices of the given RE, since *REgen* is not a test case generator for verifying SW systems. However, since the GTs are randomly generated, the benchmark can be dimensioned to be statistically significant.

4 Experimental Results

By using *REgen*, we have generated a large benchmark, which we needed for a related project, where we had developed a new deterministic parsing algorithm for ambiguous REs [1]. We wanted to objectively compare its speed against other existing algorithms, such as the RE2 library. The few collections of REs we could find, though too small for such a purpose, provided an initial indication of the kind of REs to be considered for a benchmark. Since the RE parsers are not biased towards specific uses of REs, we needed to exert them on an RE mix covering large intervals of essentially all the parameters listed in Table 2.

Some relevant features of such a benchmark are shown in Table 3 and are commented below. There are two collections of ten groups, one unbalanced and the other balanced, and each group contains one hundred GREs. The group parameters, which control the GRE generation, have the following values. The frontier length φ ranges from 1 to 100, increasing by 10 in each group; the depth τ ranges from 2 to 5; the repetition depth (in particular the star and cross depths) is limited only by τ ; and the operator arity δ is unlimited. The following operator pairs are forbidden for parent-child nodes: ($'|'$, $'|'$), ($'.'$, $'.'$) and (*unary*, *unary*). The text letters are taken from the alphanumeric alphabet (62 characters). The benchmark size in terms of GREs, which is of 2,000 REs totaling about 280,000 characters, is much larger than any existing collection.

Next, we look at other properties. Through an external ambiguity detection algorithm [1], we analyzed the GREs. In Table 3, the *ambiguous* row counts the number of ambiguous GREs, with the rather surprising result that 59% unbalanced and 67% balanced GREs are ambiguous. Of course, having a large percentage of ambiguous REs was very desirable for evaluating the parsers.

In each *collection*, we also checked the presence of GREs defining the same language, and we found almost none of them. A plausible explanation is that the

Table 3. Benchmark for the performance evaluation of RE parsing algorithms.

Applies to	Benchmark feature	Collection 1	Collection 2
Generated regular expression (GRE)	Balancing	Unbalanced	Balanced
	Total number of GREs	1,000	1,000
	N. of groups of GREs	10	10
	N. of ambiguous GREs	596	677
	N. of weakly equiv. GREs	6	2
	N. of overlapping GREs	544	666
	N. of discarded GREs	0	486
	Total GRE length	125,165 char	154,436 char
Gen. text (GT)	Number of texts	99,730	99,820
	Total GT length	45,081,981 char	45,155,166 char
GRE+GT	CPU generation time	33 s	37 s

generated corpus, though large, is very small compared to the huge generation space, so that the generation algorithms effectively produce random GREs. On the other hand, the number of GREs that define non-disjoint languages is much higher: 54% unbalanced and 66% balanced GREs define a language that overlaps the language of another GRE in the same collection.

In the row *discarded*, the unbalanced generator always ends successfully, whereas the balanced one creates 49% GREs that fail to have the required frontier length φ . This is plausibly due to the length constraint imposed by Proposition 1 for a given depth. This notwithstanding, collection generation is fast (see below). Then, we comment the results for GTs. We wanted to plot the execution times of different parsing algorithms on texts of length in $1 \dots 100$. Moreover, to improve measurement accuracy, the number of GTs in each length class must be uniform and sufficiently large. Initially, this was problematic, since the *density function* [8] of regular languages for most GREs is a high-order polynomial or exponential. Therefore, for a given GRE, the population of shorter GTs was often scarce. To increase the number of short texts, we gathered the GTs of identical length coming from different GREs present in the benchmark, thus obtaining an adequate and uniform number of GTs for all the lengths.

REgen is fast enough for a practical use. Table 3 reports the CPU time¹ to generate the benchmark, including both GREs and GTs, but excluding the time for testing GRE ambiguity and language disjointness, performed off-line.

Based on such an experience, we anticipate that it will be easier and faster to generate customized benchmarks for RE applications more specialized than RE parsing. For instance, REs for text or pattern searching are typically simpler and have a low nesting operator degree, in particular for repetition operators.

¹ On an AMD Athlon dual-core processor with 2.00 GB RAM and 2.20 GHz clock.

5 Conclusion

Since no previous work on RE generation systems was available, in order to make a well-thought design of *REgen* we initially considered a wide range of RE parameters. Then, we experimented with tool prototypes for different input parameters, and we compared the generation times and the sensitivity of the generated GRE corpus to various parameters. Eventually, we selected the parameters listed in this paper. With such a selection, *REgen* is capable of producing a satisfactory variety of GREs, and is fast. Yet the current choice is by no means final, and further experience will be important. We hope that this tool and the benchmark will serve the scientific and technical communities.

Future Developments. The percentage of ambiguous GREs in the benchmark is likely to be significant for some RE SW, such as those for searching, but currently it is not an input parameter to *REgen*. To add it to the parameters, we need to study how to efficiently incorporate an ambiguity test, such as [6], into the RE generator. A different possibility would be to incorporate an NFA generation algorithm (see [4] and its references) into *REgen*, and then to compute one or more REs for the language recognized. At last, a promising parameter for a future investigation is the RE *density function* [8]. By using the formal results in that paper, one might engineer a generator to produce GREs with specified density functions, say, polynomial. Yet the interplay between GRE density and ambiguity remains to be clarified: in our setting, it seems more appropriate to define for an RE a density function representing, for any length value, the number of syntax trees, instead of the number of texts as done in [8].

References

1. Borsotti, A., Breveglieri, L., Crespi Reghizzi, S., Morzenti, A.: From ambiguous regular expressions to deterministic parsing automata. In: Drewes, F. (ed.) CIAA 2015. LNCS, vol. 9223, pp. 35–48. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-22360-5_4
2. Câmpeanu, C., Salomaa, K., Yu, S.: Regex and extended regex. In: Champarnaud, J.-M., Maurel, D. (eds.) CIAA 2002. LNCS, vol. 2608, pp. 77–84. Springer, Heidelberg (2003). https://doi.org/10.1007/3-540-44977-9_7
3. Celentano, A., Crespi Reghizzi, S., Della Vigna, P., Ghezzi, C., Granata, G., Savoretti, F.: Compiler testing using a sentence generator. *Softw. Pract. Exp.* **10**, 897–918 (1980). <https://doi.org/10.1002/spe.4380101104>
4. Héam, P.-C., Joly, J.-L.: On the uniform random generation of non deterministic automata up to isomorphism. In: Drewes, F. (ed.) CIAA 2015. LNCS, vol. 9223, pp. 140–152. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-22360-5_12
5. Lee, J., Shallit, J.: Enumerating regular expressions and their languages. In: Domaratzki, M., Okhotin, A., Salomaa, K., Yu, S. (eds.) CIAA 2004. LNCS, vol. 3317, pp. 2–22. Springer, Heidelberg (2005). https://doi.org/10.1007/978-3-540-30500-2_2
6. Sulzmann, M., Lu, K.Z.M.: Derivative-based diagnosis of regular expression ambiguity. *Int. J. Found. Comput. Sci.* **28**(5), 543–562 (2017)

7. Sutton, M., Greene, A., Amini, P.: Fuzzing: Brute Force Vulnerability Discovery. Addison-Wesley, Boston (2007)
8. Szilard, A., Yu, S., Zhang, K., Shallit, J.: Characterizing regular languages with polynomial densities. In: Havel, I.M., Koubek, V. (eds.) MFCS 1992. LNCS, vol. 629, pp. 494–503. Springer, Heidelberg (1992). https://doi.org/10.1007/3-540-55808-X_48