Efficient Enumeration of Regular Expressions for Faster Regular Expression Synthesis

Su-Hyeon Kim, Hyeonseung Im, and Sang-Ki Ko(✉)

Department of Computer Science and Engineering, Kangwon National University, 1, Gangwondaehak-gil, Chuncheon-si, Gangwon-do, South Korea
{tngus98207,hsim,sangkiko}@kangwon.ac.kr

Abstract. We study the problem of synthesizing regular expressions from a set of positive and negative strings. The previous synthesis algorithm proposed by Lee *et al.* [12] relies on the best-first enumeration of regular expressions. To improve the performance of the enumeration process, we define a new normal form of regular expressions called the *concise normal form* which allows us to significantly reduce the search space by pruning those not in the normal form while still capturing the whole class of regular languages. We conduct experiments with two benchmark datasets and demonstrate that our synthesis algorithm based on the proposed normal form outperforms the previous algorithm in terms of runtime complexity and scalability.

Keywords: Regular expression · Program synthesis · Normal form · Enumerative search

1 Introduction

Regular expressions (REs) are widely used for the pattern matching problem to effectively and efficiently describe strings of interest. Due to their compact representations and various advantages, REs are supported in many practical applications such as search engines, text processing, programming languages, and compilers. However, writing a minimal and correct RE for a given set of strings is error-prone and sometimes difficult even for experts. With recent advances in the program synthesis technology [8], to help novice users, many researchers have investigated various methods that automatically generate REs from a set of positive and negative examples [12,17], natural language descriptions [10,14], or both [3,19].

In order to synthesize a RE satisfying the provided examples, it is often inevitable to enumerate REs in some order and check if each RE satisfies the synthesis constraints. Lee *et al.* [12] proposed a best-first enumeration algorithm called AlphaRegex which synthesizes a RE from a set of positive and negative strings. They also suggested various pruning algorithms that identify semantically equivalent (language-equivalent) expressions and prune out hopeless intermediate expressions determined by given positive and negative examples to reduce the search space. Indeed, their pruning algorithms drastically improved the naïve enumerative search algorithm, yet still far from being scalable for more complex examples.

Meanwhile, there has been much interest in the descriptional complexity of REs including several heuristics for simplifying them [2,5–7]. Brüggemann-Klein introduced the *star normal form* (snf) to improve the time complexity of constructing the position automata from REs from cubic to quadratic time. While Brüggemann-Klein considered REs recursively defined with union, concatenation, and Kleene-star, Gruber and Gulan [7] extended the definition of the snf with the question operator $R_1^?$ defined as $L(R_1^?) = \{\epsilon\} \cup L(R_1)$ and called their extension the *strong star normal form* (ssnf). A RE R is in ssnf if for any subexpression of the form R_1^* or $R_1^?$, the language represented by R_1 does not include the empty string ϵ. They also showed that the ssnf is more concise than the previous snf and still computable in linear time as snf is. Lee and Shallit [11] discussed enumeration of REs and corresponding regular languages using unambiguous grammars generating REs and their commutative images. They also provided exact numbers of regular languages representable by REs of given length. Broda *et al.* [1] studied the average behavior of REs in ssnf by computing the asymptotic estimates for the number of REs in ssnf and conducted several experiments for corroborating the estimates.

In this paper, we revisit the problem of synthesizing a RE from a given set of positive and negative examples. In particular, we aim to improve the performance of previous studies by introducing a new normal form called the *concise normal form* (cnf) of REs for an efficient enumeration during the best-first search. We introduce several rules where the equivalence of REs is identifiable in polynomial time and incorporate the rules to define the cnf. We show that the cnf is considerably more concise than the ssnf by actually enumerating all expressions in each normal form up to a given length. Finally, we demonstrate that our RE synthesis algorithm based on the cnf improves the previous state-of-the-art algorithm AlphaRegex.

The rest of the paper is organized as follows. Section 2 gives some definitions and notations. We introduce our normal form definition in Sect. 3 and the synthesis algorithm in Sect. 4. Finally, the experimental results are provided in Sect. 5.

2 Preliminaries

This section briefly recalls the basic definitions used throughout the paper. For complete background knowledge in automata theory, the reader may refer to textbooks [9,18].

Let Σ be a finite alphabet and Σ^* be the set of all strings over the alphabet Σ. A *regular expression* (RE) over Σ is $a \in \Sigma$, or is obtained by applying the following rules finitely many times. For REs R_1 and R_2, the union $R_1 + R_2$, the concatenation $R_1 \cdot R_2$, the star R_1^*, and the question $R_1^?$ are also REs. Note that $L(R_1^?)$ is defined as $L(R_1) \cup \{\epsilon\}$. Two REs R_1 and R_2 are *equivalent* if $L(R_1) = L(R_2)$. When R_1 and R_2 are equivalent, we write $R_1 \equiv R_2$ instead of $L(R_1) = L(R_2)$ for notational convenience.

The *reverse Polish notation* (RPN) length of R is denoted by rpn(R) and defined as $\text{rpn}(R) = |R|_\Sigma + |R|_+ + |R|_\cdot + |R|_* + |R|_?$. For instance, rpn($ab^?$) is 4 since we also count the question operator and the (hidden) concatenation operator between a and $b^?$. In other words, rpn(R) is the number of nodes in the corresponding syntax tree of R. As we deal with REs in the form of parse trees internally, rpn(R) can be considered as more accurate measure for representing the complexity of the REs.

Let S be a set of REs and $c_k \in \mathbb{N}$ for $1 \le k \le 5$ be a natural number implying the cost of a regular operator or a symbol. We define the *cost* of REs using the cost function $C : S \to \mathbb{N}$ which associates a cost with each expression as follows:

$$\begin{aligned}
C(a) &= c_1 \\
C(R_1 + R_2) &= C(R_1) + C(R_2) + c_2 \\
C(R_1 \cdot R_2) &= C(R_1) + C(R_2) + c_3 \\
C(R^*) &= C(R) + c_4 \\
C(R^?) &= C(R) + c_5
\end{aligned}$$

Let $\preceq$ be a relation on S, and $\preceq^*$ a transitive closure of $\preceq$. A rewriting system $(S, \preceq)$ is said to be *terminating* if there is no infinite descending chain $R_0 \preceq R_1 \preceq R_2 \preceq \cdots$, where $R_k \in S$ for $k \in \mathbb{N}$. In a terminating rewriting system $(S, \preceq)$, every element in S has at least one normal form.

Here we introduce the concept of 'similar' REs which is a weaker notion of the equivalence between two regular languages represented by REs. Owens *et al.* [13] formally define the concept of being 'similar' to approximate the least equivalence relation on REs as follows:

Definition 1. *Let $\approx$ denote the equivalence relation on REs including the following equations:*

$$\begin{aligned}
R + R &\approx R & (R^*)^* &\approx R^* \\
R_1 + R_2 &\approx R_2 + R_1 & (R^?)^? &\approx R^? \\
(R_1 + R_2) + R_3 &\approx R_1 + (R_2 + R_3) & (R^*)^? &\approx R^* \\
(R_1 \cdot R_2) \cdot R_3 &\approx R_1 \cdot (R_2 \cdot R_3) & (R^?)^* &\approx R^*
\end{aligned}$$

Two REs R_1 and R_2 are similar if $R_1 \approx R_2$ and dissimilar otherwise.

It is trivial that the following statement holds from simple algebraic consequences of the inductive definition of REs.

Corollary 1. *If $R_1 \approx R_2$, then $R_1 \equiv R_2$.*

Given a set of positive and negative strings, we consider the problem of synthesizing a concise RE that is consistent with the given strings. The examples are given by a pair (P, N) of two sets of strings, where $P \subseteq \Sigma^*$ is a set of *positive strings* and $N \subseteq \Sigma^*$ is a set of *negative strings*.

Then, our goal is to find a RE R that accepts all positive strings in P while rejecting all negative strings in N. Formally, R satisfies the following condition:

$$P \subseteq L(R) \text{ and } L(R) \cap N = \emptyset.$$

Since there are infinitely many REs satisfying the condition, we aim at finding the most concise RE among all such expressions. We utilize the cost function C to quantify the conciseness of REs.

3 Concise Normal Form for REs

Now we define the relation $\preceq$ of REs to define a terminating RE rewriting system $(S, \preceq)$ that produces a more concise RE in terms of RPN (or at least a RE with the same RPN). Let R and R_k be REs for any natural number k. First, we consider the case when a RE has a subexpression that is formed by the concatenation of similar REs.

Lemma 1 (Redundant Concatenation (RC) Rule 1). *For a RE R, the following equivalences hold:*

(i) $R^? R \preceq R R^?$
(ii) $R^* R \preceq R R^*$
(iii) $R^* R^? \preceq R^? R^* \preceq R^*$

Using the lemma above, we consider all REs with subexpressions in the form of $R^? R$, $R^* R$, $R^* R^?$, or $R^? R^*$ as redundant, as we can always rewrite those subexpressions as $R R^?$, $R R^*$, or R^* without changing the language represented by the resulting RE. We can further consider the following type of redundant concatenation even when two concatenated subexpressions do not share exactly the same expression.

Lemma 2 (RC Rule 2). *If* $\epsilon \in L(R_1)$ *and* $L(R_1) \subseteq L(R_2^*)$, *then* $R_1 R_2^* \preceq R_2^*$ *and* $R_2^* R_1 \preceq R_2^*$.

Lemma 3 (Kleene-Concatenation-Kleene (KCK) Rule).
If $L(R_1) \cup L(R_3) \subseteq L(R_2^*)$, *then* $(R_1 R_2^* R_3)^* \preceq (R_1 R_2^* R_3)^?$.

Lemma 4 (Kleene-Concatenation-Question (KCQ) Rule).
If $L(R_1) \cup L(R_3) \subseteq L(R_2^*)$ *and* $\epsilon \in L(R_1) \cap L(R_3)$, *then* $(R_1 R_2 R_3)^* \preceq R_2^*$.

When the question operator is used for the concatenation of two REs, we find the following rule.

Lemma 5 (Question-Concatenation (QC) Rule).
$(R R^*)^? \preceq R^*$ *and* $(R R^?)^? \preceq R^? R^?$ *hold.*

When the union operator is used for multiple subexpressions, we find the following four equivalence cases.

Lemma 6 (Union-Question (UQ) rule). *$R_1 + R_2^? \preceq (R_1 + R_2)^?$ holds.*

Lemma 7 (Inclusive Union (IU) Rule). *If $L(R_1) \subseteq L(R_2)$, then $R_1 + R_2 \preceq R_2$.*

We also use a rule named the *factoring rule*, which trivially holds by a simple algebraic law (distributive law), to factor the common prefix or suffix of REs within a union operator until there is no such subexpression.

Corollary 2 (Factoring Rule).
$R_1R_2 + R_1R_3 \preceq R_1(R_2 + R_3)$ and $R_2R_1 + R_3R_1 \preceq (R_2 + R_3)R_1$ hold.

Finally, we use the following observation when a Kleene-star operator is used for an expression that represents each symbol in the alphabet, as the resulting expression is equivalent to Σ^* (Sigma-star), which represents all possible strings over the alphabet Σ.

Corollary 3 (Sigma-star Rule). *If $\Sigma \subseteq L(R)$, then $R^* \preceq \Sigma^*$.*

Corollary 4. *If $R_1 \preceq R_2$, then $\mathrm{rpn}(R_1) \geq \mathrm{rpn}(R_2)$ and $R_1 \equiv R_2$ hold.*

Now we are ready to introduce our new normal form for REs called the *concise normal form* (cnf). Simply speaking, a RE is in cnf if its every subexpression does not fall into a case introduced thus far. We formally define the cnf as follows:

Definition 2. *We define a RE R to be in* cnf *if R does not contain a subexpression in any of the following forms:*

1. *R^* or $R^?$ where $\epsilon \in L(R)$ (*ssnf*)*
2. *$R^?R$, R^*R, $R^*R^?$ or $R^?R^*$ (RC Rule 1)*
3. *$R_1R_2^*$ or $R_2^*R_1$ where $\epsilon \in L(R_1)$ and $L(R_1) \subseteq L(R_2^*)$ (RC Rule 2)*
4. *$(R_1R_2^*R_3)^*$ where $L(R_1) \cup L(R_3) \subseteq L(R_2^*)$ (KCK Rule)*
5. *$(R_1R_2R_3)^*$ where $L(R_1) \cup L(R_3) \subseteq L(R_2^*)$, $\epsilon \in L(R_1) \cap L(R_3)$ (KCQ Rule)*
6. *$(RR^*)^?$ or $(RR^?)^?$ (QC Rule)*
7. *$R_1 + R_2^?$ (UQ Rule)*
8. *$R_1 + R_2$ where $L(R_1) \subseteq L(R_2)$ (IU Rule)*
9. *$R_1R_2 + R_1R_3$ or $R_2R_1 + R_3R_1$ (Factoring Rule)*
10. *R^* where $R \neq a_1 + a_2 + \cdots + a_n$ and $\Sigma \subseteq L(R)$ (Sigma-star Rule)*

In order to prove that there always exists a RE in cnf for any given RE, we prove the following result:

Lemma 8. *The rewriting system $(S, \preceq)$ is terminating.*

Proof. For the sake of contradiction, suppose that $(S, \preceq)$ is not terminating and there is an infinite chain $R_0 \preceq R_1 \preceq R_2 \preceq \cdots$. Since Corollary 4 guarantees that the RPN length of REs does not increase by $(S, \preceq)$, it is easy to verify that there exists a RE R which is repeated infinitely many times in the chain.

Therefore, it suffices to consider the following cases where the rewriting system results in the same RPN length:

(i) $R^?R \preceq RR^?$ (By Lemma 1)
(ii) $(R_1R_2^*R_3)^* \preceq (R_1R_2^*R_3)^?$ (By Lemma 3)
(iii) $(RR^?)^? \preceq R^?R^?$ (By Lemma 5)
(iv) $R_1 + R_2^? \preceq (R_1 + R_2)^?$ (By Lemma 6)

In the following, we demonstrate that no rule can initiate an infinite chain of REs with proof by cases.

Case (i): Assume that the infinite chain is formed by the first rewriting relation $R^?R \preceq RR^?$. This implies that there exists a derivation in the form of $R_1RR^?R_2 \preceq^* R_1R^?RR_2$ for any $R_1, R_2 \in S$. Since there is no relation that rewrites the concatenation of two expressions other than $R^?R \preceq RR^?$, we should consider derivations of the following form:

$$R_1RR^?R_2 \preceq R_1'R_1''RR^?R_2'R_2'' \preceq^* R_1RR^?R_2,$$

where $R_1 = R_1'R_1''$ and $R_2 = R_2'R_2''$.

In this case, $R_1''R$ should be converted into $R_1'''R^?$ where $(R_1''')^? = R_1''$ by the case (i) since there is no other possibility to convert $R^?$ into R. Hence, we have the intermediate expression $R_1'R_1'''R^?R^?R_2'R_2''$. Now, we can see that $\mathrm{rpn}(R_1'R_1''') < \mathrm{rpn}(R_1)$ and therefore there is no possibility to reach $R_1RR^?R_2$ by the rewriting system.

Case (ii): Let us consider the second case $(R_1R_2^*R_3)^* \preceq (R_1R_2^*R_3)^?$. It is easy to see that the rule cannot be used to form the infinite chain of REs as the rule replaces a Kleene-star operator with a question operator. Since there is no relation that places the removed question operator back, it is simply impossible to use the rule in the infinite chain.

Case (iii): The third case $(RR^?)^? \preceq R^?R^?$ can be applied when concatenation is used inside the question operator. In order to move back to the form before the rule is applied, we need a relation that places a question operator enclosing an expression which is a concatenation of two expressions. However, there is no such rule in the rewriting system.

Case (iv): The fourth case $R_1 + R_2^? \preceq (R_1 + R_2)^?$ can be applied when union is used inside the question operator. In order to move back to the form before the rule is applied, we need a relation that places a question operator enclosing an expression which is a union of two expressions. However, there is no such rule in the rewriting system.

Since we have shown that an infinite chain of REs by the rewriting system $(S, \preceq)$ cannot exist, the proof is completed. □

As a corollary of Lemma 8, we observe the following result:

Corollary 5. *Given a RE R, there always exists a RE R' in* cnf *such that $R \equiv R'$.*

Unfortunately, it is well-known that the problem of testing inclusion between two REs is PSPACE-complete [16]. Hence, we can easily deduce that the problem of testing whether a given RE is in cnf is also PSPACE-complete as follows:

Lemma 9. *Given a RE R, the problem of determining whether or not R is in* cnf *is PSPACE-complete.*

Proof. Without loss of generality, we assume that two REs R_1 and R_2 do not share the common prefix or suffix as we can easily factor out them. We also assume that R_1 and R_2 do not contain a question operator.

Note that testing $L(R_1) \subseteq L(R_2)$ is PSPACE-complete. Now a RE $R_1 + R_2$ can be converted into a cnf expression R_2 if and only if $L(R_1) \subseteq L(R_2)$. Therefore, it is easily seen that the problem of determining whether a given RE is in cnf is also PSPACE-complete. □

Since the cnf testing is PSPACE-complete, we instead introduce a relaxed concept of the cnf called the *soft concise normal form* (scnf) by relaxing the language inclusion restrictions in the cnf such as $L(R_1) \subseteq L(R_2)$.

We first introduce a weaker notion of the language inclusion relation as follows which can be determined in linear time:

Definition 3. *Given two REs R_1 and R_2 over $\Sigma = \{a_1, a_2, \ldots, a_n\}$, we define $R_1 \sqsubseteq R_2$ if R_1 and R_2 satisfy one of the following conditions:*

(i) $R_1 \approx R_2$
(ii) $R_2 = R^*$ *for any* $R \in S$ *such that* $\Sigma \subseteq L(R)$
(iii) $R_2 = R_1^*$
(iv) $R_2 = R_1^?$
(v) $R_2 = R^*$ *and* $R_1 = R^?$ *for any* $R \in S$
(vi) $R_2 = (R_1 + R)^*$ *for any* $R \in S$

Note that the following relation trivially holds:

Corollary 6. *If $R_1 \sqsubseteq R_2$, then $L(R_1) \subseteq L(R_2)$.*

Now we formally define the scnf as follows:

Definition 4. *We define a RE r to be in* scnf *if r does not contain a subexpression in Definition 2 where every restriction in the form of $L(R_1) \subseteq L(R_2)$ is replaced by $R_1 \sqsubseteq R_2$.*

Actually, it turns out that it is possible to determine whether or not a given RE is in scnf in polynomial time.

Lemma 10. *Given a RE R, we can determine whether or not R is in* scnf *in polynomial time.*

4 RE Synthesis Algorithm

We synthesize REs by relying on the best-first search while only considering REs in scnf as REs not in scnf have more concise expressions representing the same regular languages. Hence, we can prune out numerous REs by simply checking if the expressions are in the scnf regardless of the given examples.

4.1 Best-First Search Algorithm

As in the AlphaRegex [12], we utilize the best-first search to find the most concise RE consistent with the given examples. Starting from the simplest form of REs, we examine more complicated expressions until finding the solution.

We introduce a *hole* ($\square$) that is to be replaced with some concrete RE. We call REs with holes the *templates*. In order to perform the best-first search, we rely on a priority queue to determine the next candidate. After pushing the initial template $\square$ into the priority queue, we retrieve each template with the minimal cost determined by the cost function C from the priority queue. For each retrieved template, we generate more complicated templates or concrete expressions by replacing holes with each symbol in Σ, ϵ, $\emptyset$, $\square + \square$, $\square \cdot \square$, $\square^*$, and $\square^?$ and push them into the priority queue to continue the best-first search. The search algorithm terminates when we find a solution which is consistent with the given examples and not redundant.

We also use the additional pruning rule considered in AlphaRegex. Given a template R, we define $\widehat{R}$ ($\widetilde{R}$, resp.) to be a concrete RE obtained by replacing every hole in R with Σ^* ($\emptyset$, resp.). Informally, $\widehat{R}$ is an over-approximation of R as Σ^* is the most general RE and $\widetilde{R}$ is an under-approximation of R as $\emptyset$ is an expression for the smallest set of strings among all REs. During the search, we prune a template R if either $P \not\subseteq L(\widehat{R})$ or $L(\widetilde{R}) \cap N \neq \emptyset$ holds as it is already impossible for R to reach any concrete expression consistent with (P, N).

4.2 Finding Redundancy Using Positive Examples

Meanwhile, we can further prune out the search space by relying on the set of positive strings that the resulting RE should accept. In AlphaRegex [12], the authors define a RE to be *redundant* if the RE contains an operator that can be omitted while still accepting the positive strings.

We first explain the functions introduced in the AlphaRegex here to be self-contained as follows:

$$\begin{aligned}
\mathsf{un}(a) &= a \ (a \in \Sigma) & \mathsf{sp}(a) &= \{a\} \ (a \in \Sigma)\\
\mathsf{un}(R_1 + R_2) &= \mathsf{un}(R_1) + \mathsf{un}(R_2) & \mathsf{sp}(R_1 + R_2) &= \mathsf{sp}(R_1) \cup \mathsf{sp}(R_2)\\
\mathsf{un}(R_1 \cdot R_2) &= \mathsf{un}(R_1) \cdot \mathsf{un}(R_2) & \mathsf{sp}(R_1 \cdot R_2) &= \{R_1' \cdot R_2, R_1 \cdot R_2' \mid R_i' \in \mathsf{sp}(R_i)\}\\
\mathsf{un}(R^*) &= R \cdot R \cdot R^* & \mathsf{sp}(R^*) &= \{R^*\}\\
\mathsf{un}(\square) &= \square & \mathsf{sp}(\square) &= \{\square\}
\end{aligned}$$

Lee *et al.* introduced the un and sp functions to check the redundancy of star and union operators used in REs, respectively. Given a RE R (possibly with holes) and a set P of positive examples, they define R to be *redundant* if there exists a regular expression $R' \in \mathsf{sp}(\mathsf{un}(R))$ such that $L(\widehat{R'}) \cap P = \emptyset$. For instance, consider a set $P = \{0, 01, 011, 0111\}$ and two templates: $1^* \cdot \square$ and $0^* \cdot \square$. Then, $1^* \cdot \square$ is redundant since $\mathsf{sp}(\mathsf{un}(1^* \cdot \square)) = \{111^* \cdot \square\}$ and apparently $L(111^* \cdot \Sigma^*)$ does not contain any string in P. Analogously, $0^* \cdot \square$ is also redundant.

Algorithm 1: Our Synthesis Algorithm

```
Input  : Positive and negative strings $(P, N)$
Output: A RE $R$ consistent with $(P, N)$
Initialize a priority queue $Q$;
Push the initial template $\square$ into $Q$;
repeat
    Pop a minimal cost template $R$ from $Q$;
    if $R$ is a complete RE then
        if $R$ is consistent with $(P, N)$ then
            return $R$
    else
        foreach $R' \in \mathsf{next}(R)$ do
            if $P \subseteq L(\widehat{R'})$ or $L(\widetilde{R'}) \cap N = \emptyset$ then
                if $R'$ is in scnf then
                    if $R'$ not redundant for $P$ then
                        Push $R'$ into $Q$;
until $Q \neq \emptyset$;
```

4.3 Our Synthesis Algorithm

Algorithm 1 shows the final synthesis algorithm. We first initialize a priority queue Q that internally sorts templates according to their costs calculated by the cost function C in increasing order. We first push the simplest template $\square$ into Q and repeat the following procedure.

1. We retrieve a minimal cost template R from Q and check whether or not R is a complete RE and consistent with the given examples (P, N). If so, we return R as a synthesized RE. Otherwise, we proceed to the next step.
2. If R is a template with holes, then we generate templates by replacing a hole with one of Σ, ϵ, $\emptyset$, $\square + \square$, $\square \cdot \square$, $\square^*$, or $\square^?$ (defined as the set $\mathsf{next}(R)$). For each generated template R', we test whether or not R' has a possibility of evolving into a RE satisfying the given examples. If so, we also test whether R' is in scnf and not redundant for positive examples P. If R' qualifies the tests, then we push R' into Q.

5 Experimental Results

We conduct several experiments to verify that the proposed normal form of REs significantly reduces the number of REs when enumerating all possible regular languages. By doing so, we first show that the new normal form is more efficient to enumerate distinct regular languages that are given in the form of REs by pruning out numerous REs not in the new normal form. Second, we demonstrate that the proposed normal form is useful when synthesizing a RE from a set of positive and negative strings by enumerating all possible candidates by pruning a vast amount of the search space during the enumeration process.

Table 1. The number of REs in a given RPN length.

rpn(R)	Exact Enum. [11]	Base	ssnf [7]	scnf	Pruning Ratio
1	2	2	2	2	0.00
2	4	4	4	4	0.00
3	7	7	7	5	28.57
4	13	38	38	24	36.84
5	32	106	90	42	60.38
6	90	364	312	146	59.89
7	189	1,444	1,236	481	66.69
8	580	5,170	3,650	1,278	75.28
9	1,347	19,741	14,849	4,636	76.52
10	3,978	77,838	52,388	14,675	81.15
11	-	302,908	188,820	46,978	84.49
12	-	1,206,042	741,108	165,818	86.25
13	-	4,853,655	2,690,537	537,446	88.93

5.1 Exact Enumeration of REs in Normal Form

First, we count the number of REs in a given RPN length and compare it with the number of REs in scnf in Table 1. Recall that Lee *et al.* [11] attempted to obtain the asymptotic estimates on the number of regular languages specified by REs of given size n by the aid of the Chomsky-Schützenberger theorem [4] and singularity analysis of the algebraic formal power series. Note that the upper bound and lower bound obtained in [11] are $O(3.9870^n)$ and $\Omega(2.2140^n)$.

In order to estimate the expected growth rate of the numbers given in Table 1, we fit exponential curves to enumeration results using SciPy's scipy.optimize.curve_fit function which implements a non-linear least-square fit. As a result, we obtain the following estimates for the number of all valid REs and the number of all REs in the proposed normal form of a given RPN length as follows:

$$0.067 \times 4.022^n + 882.444 \text{ and } 0.108 \times 3.275^n - 303.477.$$

As the numbers are growing exponentially, our synthesis algorithm is expected to run exponentially faster than simple enumeration-based algorithm and scale much better for more complicated examples.

5.2 Performance of RE Synthesis

For experiments of RE synthesis, we utilize two benchmark datasets: the AlphaRegex dataset and random dataset. The AlphaRegex dataset consists of 25 REs from famous textbooks [9,15] on automata and formal language theory. The authors of AlphaRegex created a set of positive and negative examples for

Table 2. Comparisons of performance of AlphaRegex and our synthesis algorithm.

Benchmark	Method	Avg. Count	Avg. Time	Success Ratio
Random	AlphaRegex	7,445	9.36 s	83.3%
	AlphaRegex + Redundancy Check	4,432	6.84 s	87.4%
	Ours	3,478	4.39 s	87.9%
	Ours + Redundancy Check	**1,813**	**2.68 s**	**91.9%**
AlphaRegex	AlphaRegex	7,038	8.71 s	76.0%
	AlphaRegex + Redundancy Check	5,190	7.38 s	88.0%
	Ours	3,814	4.66 s	88.0%
	Ours + Redundancy Check	**2,202**	**3.11 s**	**96.0%**

each RE in the dataset[1]. Note that both datasets only consist of REs over binary alphabet $\{0, 1\}$.

The random dataset contains 1,000 distinct randomly generated REs. We first start from an initial template '$\square$' and randomly replace a hole in the template by one of $a \in \Sigma$, ϵ, $\emptyset$, $\square + \square$, $\square \cdot \square$, $\square^*$, or $\square^?$. We repeat the process 10 times and complete the template by randomly replacing every hole with one of the symbols in Σ. If it is impossible to generate 10 positive examples from the random RE as it can only describe a finite number of strings or its length is shorter than 7, we re-generate a RE. We generate a set of 10 positive examples and 10 negative examples for each random RE. In order to generate positive examples, we utilize a Python library called the Xeger[2]. For generating negative examples, we first randomly choose a number n between 1 and 15 and generate a random string of length n. We repeat the process until we have 10 distinct strings that cannot be described by the RE.

The experimental results are shown in Table 2. We compare our algorithm implemented in Python 3 with our implementation of AlphaRegex on the two benchmark datasets with or without the redundancy checking algorithm introduced in AlphaRegex. Note that we use our implementation of AlphaRegex instead of the original OCaml implementation of AlphaRegex for a fair comparison. We set the limit on the number of visited templates to be 100,000 and consider the examples synthesized before reaching the successful limit. The average numbers (e.g., count and time) are calculated only for the successful examples. The experimental results show that our synthesis algorithm is faster than AlphaRegex in terms of both the average number of visited templates (including complete REs) to find the solution and the actual runtime of our Python implementation.

[1] The OCaml implementation of AlphaRegex and dataset are publicly available at https://github.com/kupl/AlphaRegexPublic.

[2] https://pypi.org/project/xeger/.

Acknowledgements. This work was supported by the National Research Foundation of Korea (NRF) grant funded by the Korean government (MIST) (No. 2020R1A4A3079947).

References

1. Broda, S., Machiavelo, A., Moreira, N., Reis, R.: On average behaviour of regular expressions in strong star normal form. Int. J. Found. Comput. Sci. **30**(6–7), 899–920 (2019)
2. Brüggemann-Klein, A.: Regular expressions into finite automata. Theoret. Comput. Sci. **120**(2), 197–213 (1993)
3. Chen, Q., Wang, X., Ye, X., Durrett, G., Dillig, I.: Multi-modal synthesis of regular expressions. In: PLDI 2020, pp. 487–502 (2020)
4. Chomsky, N., Schützenberger, M.: The algebraic theory of context-free languages. In: Computer Programming and Formal Systems. Studies in Logic and the Foundations of Mathematics, vol. 35, pp. 118–161. Elsevier (1963)
5. Ellul, K., Krawetz, B., Shallit, J.O., Wang, M.: Regular expressions: new results and open problems. J. Autom. Lang. Comb. **10**(4), 407–437 (2005)
6. Frishert, M., Watson, B.W.: Combining regular expressions with (near-)optimal Brzozowski automata. In: Domaratzki, M., Okhotin, A., Salomaa, K., Yu, S. (eds.) CIAA 2004. LNCS, vol. 3317, pp. 319–320. Springer, Heidelberg (2005). https://doi.org/10.1007/978-3-540-30500-2_34
7. Gruber, H., Gulan, S.: Simplifying regular expressions. In: Dediu, A.-H., Fernau, H., Martín-Vide, C. (eds.) LATA 2010. LNCS, vol. 6031, pp. 285–296. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-13089-2_24
8. Gulwani, S.: Dimensions in program synthesis. In: PPDP 2010, pp. 13–24 (2010)
9. Hopcroft, J., Ullman, J.: Introduction to Automata Theory, Languages, and Computation, 2nd edn. Addison-Wesley, Reading (1979)
10. Kushman, N., Barzilay, R.: Using semantic unification to generate regular expressions from natural language. In: NAACL-HLT 2013, pp. 826–836 (2013)
11. 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
12. Lee, M., So, S., Oh, H.: Synthesizing regular expressions from examples for introductory automata assignments. In: GPCE 2016, pp. 70–80 (2016)
13. Owens, S., Reppy, J.H., Turon, A.: Regular-expression derivatives re-examined. J. Funct. Program. **19**(2), 173–190 (2009)
14. Park, J., Ko, S., Cognetta, M., Han, Y.: Softregex: Generating regex from natural language descriptions using softened regex equivalence. In: EMNLP-IJCNLP 2019, pp. 6424–6430 (2019)
15. Sipser, M.: Introduction to the Theory of Computation. Cengage Learning (2012)
16. Stockmeyer, L.J., Meyer, A.R.: Word problems requiring exponential time: preliminary report. In: STOC 1973, pp. 1–9 (1973)
17. Wang, X., Gulwani, S., Singh, R.: FIDEX: filtering spreadsheet data using examples. In: OOPSLA 2016, pp. 195–213 (2016)
18. Wood, D.: Theory of Computation. Harper & Row (1987)
19. Ye, X., Chen, Q., Wang, X., Dillig, I., Durrett, G.: Sketch-driven regular expression generation from natural language and examples. Trans. Assoc. Comput. Linguist. **8**, 679–694 (2020)