Synthesizing Regular Expressions from Examples for Introductory Automata Assignments

Mina Lee [*]
Korea University, Korea
0x01.ming@gmail.com

Sunbeom So[*]
Korea University, Korea
sunbeom_so@korea.ac.kr

Hakjoo Oh [†]
Korea University, Korea
hakjoo_oh@korea.ac.kr

Abstract

We present a method for synthesizing regular expressions for introductory automata assignments. Given a set of positive and negative examples, the method automatically synthesizes the simplest possible regular expression that accepts all the positive examples while rejecting all the negative examples. The key novelty is the search-based synthesis algorithm that leverages ideas from over- and under-approximations to effectively prune out a large search space. We have implemented our technique in a tool and evaluated it with non-trivial benchmark problems that students often struggle with. The results show that our system can synthesize desired regular expressions in 6.7 seconds on the average, so that it can be interactively used by students to enhance their understanding of regular expressions.

Categories and Subject Descriptors D.1.2 [*Programming Techniques*]: Automatic Programming; I.2.2 [*Artificial Intelligence*]: Automatic Programming—Program Synthesis; F.3.1 [*Logics and Meanings of Programs*]: Specifying and Verifying and Reasoning about Programs

Keywords Regular expression; program synthesis; programming by example

1. Introduction

The motivation for this work came from our experiences in teaching regular expressions to undergraduate students. Regular expressions are widely used, and hence, taught in many computer science courses such as automata theory. However, we found that students often struggle to solve problems regarding regular expressions for several reasons. For example, many do not know where to start and usually give up after a certain number of trials. Furthermore, even after they manage to solve the problems, students are likely to have doubts on whether their answers are correct and optimal. Our goal is to assist students who learn regular expressions for the first time by providing an automatic tool that synthesizes regular expressions from examples given by students.

A notable feature of the tool is that regular expressions are synthesized from a few examples quickly so that the tool can be interactively used by students. The system takes positive and negative examples. Each time when a set of examples is given, the system instantly generates a regular expression that accepts all the positive examples while rejecting all the negative ones. These examples can be very simple, short, and rudimentary since they are expected to be given by students. Playing with the tool interactively, students could enhance their understanding of regular expressions and improve their skill to construct regular expressions without manual guidance from human teachers.

Existing techniques for synthesizing regular expressions are not suitable for our purposes. For example, automatic synthesis of regular expressions from examples has been extensively studied in the context of text extraction [6, 8, 10]. However, these approaches have different goals from ours; while the goal of text extraction is to find an approximate solution that generalizes the extraction behavior represented by the given examples to unseen examples, we aim to find a precise solution that is consistent with the given examples. Furthermore, because existing techniques are based on genetic algorithms, which are powerful but slowly converging, they are not suitable for an interactive system. We provide a more detailed discussion in Section 6.

In this paper, we present a new algorithm for synthesizing pedagogical regular expressions from a few examples. In order to find the simplest possible solutions, our algorithm performs an exhaustive search by prioritizing simpler regular expressions; we enumerate all possible regular expressions and check whether each candidate is consistent with

[*] These two authors contributed equally.
[†] Corresponding author

all examples. However, due to the huge search space, this naive algorithm is unlikely to find a solution with reasonable time cost for an interactive system. We present novel techniques that prune out the search space effectively while guaranteeing to find the simplest solutions. The key idea is to over- and under-approximate regular expressions to predict whether the current search state can be the final solution or not. We formally present the techniques and prove that they prune out a particular portion of the search space in a sound and complete way.

We implemented our method in a tool, AlphaRegex, and evaluated its performance with 25 benchmark problems. Most problems came from popular textbooks on automata theory [17, 20, 26]. The benchmarks include tricky problems that even instructors find hard to solve. The results show that AlphaRegex can synthesize desired regular expressions within 6.7 seconds on the average.

Contributions We summarize our contributions:

- We present a new algorithm for synthesizing regular expressions from examples. We present novel techniques that effectively prune out a large search space using over- and under-approximations of regular expressions.
- We prove the effectiveness of the technique with 25 benchmark problems including tricky ones that most students find difficult. The results show that our method quickly derives regular expressions on all of the benchmarks.
- We provide a tool, AlphaRegex, which is publicly available.[1]

2. Overview

In this section, we first illustrate the utility of our tool with two motivating examples. Then, we briefly explain the key ideas behind our synthesis algorithm. Throughout the paper, we assume that the binary alphabet $\Sigma = \{0, 1\}$ is used.

2.1 Motivating Examples

Our tool is useful for automatically assisting students who learn regular expressions for the first time.

Finding optimal solutions Suppose we want to find a regular expression for the language below:

$$L = \{\text{Strings have exactly one pair of consecutive 0s}\}.$$

The description of L states that the language consists of the strings which have exactly one pair of consecutive 0s (i.e. 00). For example, 00, 1001, and 010010 are in the language since they have one pair of continuous 0s, whereas 11, 000, and 00100 are not, because either they do not have it or they have more than one pair of consecutive 0s.

When we asked students to construct regular expressions for the language, we found that most of their answers were far from being optimal, containing lots of unnecessary symbols and operators. The most common answer we observed was $1^*(01)^*1^*001^*(10)^*1^*$; there is one pair of consecutive 0s at the middle, and on both sides, there can be any expressions as long as not having consecutive 0s, represented as $1^*(01)^*1$ and $1^*(10)^*1^*$. Although the students could find a correct answer, most of them were curious to know the existence of a simpler solution. In this case, AlphaRegex responded to them by synthesizing the regular expression $(0?1)^*00(10?)^*$, which is much simpler than the regular expression constructed by students. Besides being the simplest, it is also very compact, precise, and easier to interpret, which makes it much more desirable. The examples used for this synthesis problem are listed in Table 1. Given the examples, AlphaRegex is able to synthesize the solution in one second.

Description	
Strings have exactly one pair of consecutive 0s	
Examples	
Positive	Negative
00	01
1001	11
010010	000
1011001110	00100
Answers	
Students	$1^*(01)^*1^*001^*(10)^*1^*$
AlphaRegex	$(0?1)^*00(10?)^*$

Table 1. Finding optimal solutions

Description	
Strings have at most one pair of consecutive 1s	
Examples	
Positive	Negative
0	111
11	110011
101010	0110110
00011000	00011001100
00100110001	011100011110
Answers	
Students	A variety of uncertain answers
AlphaRegex	$(1?0)^*1?1?(01?)^*$

Table 2. Solving difficult problems

Solving difficult problems Consider the following language:

$$L = \{\text{Strings have at most one pair of consecutive 1s}\}.$$

The language L is composed of the strings which have either zero or one pair of consecutive 1s (i.e. 11). For instance, 0, 11, and 101010 are in the language since they either do

[1] http://prl.korea.ac.kr/AlphaRegex

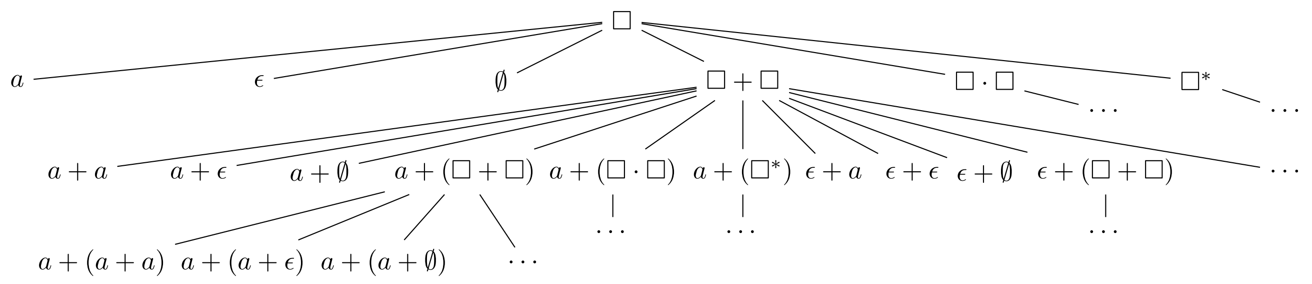

Figure 1. Basic search algorithm. a denotes an alphabet, either 0 or 1.

not have a pair of continuous 1s or have exactly one pair of consecutive 1s. On the contrary, 111, 110011, and 0110110 are not in the language because they have more than one pair of consecutive 1s. When we asked students to construct regular expressions for the language, each person came up with different, complex regular expressions, making it hard to compare and difficult to check the correctness. Also, they were uncertain about their answers and curious to know an easy-to-understand solution. In this case, ALPHAREGEX responded by synthesizing the expression $(1?0)^*1?1?(01?)^*$ which is much simpler than all of the answers suggested by students. The full list of the examples used for this problem is in Table 2. Given those examples, ALPHAREGEX is able to synthesize the regular expression in 28.8 seconds.

2.2 Informal Description

We illustrate our synthesis algorithm with a very simple regular expression problem. Suppose that we are given a set of positive examples and a set of negative examples, $\mathcal{P} = \{0, 00, 000\}$ and $\mathcal{N} = \{1, 11, 111\}$, respectively. The desired regular expression to synthesize is 0^*.

Basic search algorithm In order to find the simplest solutions, our algorithm performs an exhaustive search by prioritizing simpler regular expressions as shown in Figure 1. That is, we enumerate all regular expressions until we find a solution. Starting from the most trivial cases with single symbols in the alphabet, our system checks whether each regular expression is consistent with the given examples. For instance, it starts checking the regular expression 0 by examining whether the expression can accept all the positive examples. 0 accepts the first positive example 0, but fails to accept the next example 00. It keeps checking until it finds a regular expression that accepts all the positive examples while rejecting all the negative examples. After examining the trivial cases, it checks more complex cases such as $0+0$, $0+1$, $1+0$, $1+1$ (i.e. expressions in the form of $\square+\square$), $0\cdot 0$, $0\cdot 1$, $1\cdot 0$, $1\cdot 1$ (i.e. expressions in the form of $\square\cdot\square$), and 0^*, 1^* (i.e. expressions in the form of $\square^*$). Finally, it finds 0^* to be the simplest regular expression that is consistent with all the examples.

Here, we introduce a *hole* ($\square$) which is a placeholder for any regular expression. We call regular expressions with or without holes *states* to distinguish it from regular expressions without holes (i.e. closed regular expressions). The algorithm iteratively generates regular expressions by replacing holes with other states according to the syntax defined in Section 3 (1). For example, $\square + \square$ can be instantiated as $0+\square$, $1+\square$, $0+(\square+\square)$, and so on. It enables the system to perform best-first enumerative search with appropriate cost metric to find optimal regular expressions.

Pruning search space The naive algorithm is unlikely to find a solution quick enough due to the huge search space. Thus, we introduce novel techniques that prune out the search space effectively while guaranteeing to find the simplest solutions.

The key idea is to over- and under-approximate regular expressions to decide whether the current search state can be the final solution or not. In other words, a state cannot be a solution if any replacement of holes in the state cannot make it a solution. We call such a state a *dead* state. There are two ways to check whether a state is dead or not. First, we replace all the holes in a state with $(0+1)^*$ (i.e. over-approximation). That is, we make a state most general, accepting as many examples as possible. If the replacement fails to accept any positive example, the state is dead, meaning that it cannot be a solution by any means. Second, we replace all the holes in a state with $\emptyset$ (i.e. under-approximation). That is, we make a state least general, accepting as few examples as possible. If the replacement fails to reject any negative example, the state is dead, meaning that it cannot be a solution.

We also prune out states when we know there are simpler solutions that can be derived from other states. Unlike dead states, these states can be solutions with proper replacement of holes, but there exist simpler solutions. We call them *redundant* states, meaning that further exploration on such states is known to be redundant, causing the system to take more time to search. For example, the state $(0+\epsilon)\cdot\square$ is redundant since $0\cdot\square$ is a simpler state which yields same results with respect to the given examples. Note that the state is not dead and can be a solution with appropriate replacement (e.g. $(0+\epsilon)\cdot 0^*$), since its over-approximation (i.e. $(0+\epsilon)\cdot(0+1)^*$) accepts all the positive examples and its under-approximation (i.e. $\emptyset$) rejects all the negative examples. However, it is clear that ϵ is not necessary in the expression. Hence, we prune out the state.

3. Problem

In this section, we define regular expressions and the synthesis problem.

Regular Expressions In this paper, we consider the syntax of regular expressions that are used in introductory textbooks on automata theory (e.g. [17, 20, 26]):

$$e \to a \in \Sigma \mid \epsilon \mid \emptyset \mid e_1 + e_2 \mid e_1 \cdot e_2 \mid e^* \tag{1}$$

A symbol a from an alphabet Σ, the empty string ϵ, and the empty language $\emptyset$ constitute the primitive regular expressions. The remaining cases are inductively defined. Given regular expressions e_1 and e_2, we can construct regular expressions by taking the union $e_1 + e_2$ or the concatenation $e_1 \cdot e_2$. e^* denotes the Kleene closure of e. In the introductory courses, the alphabet is typically assumed to be binary; we assume $\Sigma = \{0, 1\}$ in the rest of this paper.

Formally, a regular expression e denotes a language (i.e. a set of strings). We write $[\![e]\!] \subseteq \Sigma^*$ for the language that e denotes, which is inductively defined as follows:

$$\begin{array}{rcl}
[\![a]\!] & = & \{a\}\ (a \in \Sigma) \\
[\![\epsilon]\!] & = & \{\epsilon\} \\
[\![\emptyset]\!] & = & \emptyset \\
[\![e_1 + e_2]\!] & = & [\![e_1]\!] \cup [\![e_2]\!] \\
[\![e_1 \cdot e_2]\!] & = & [\![e_1]\!][\![e_2]\!] \\
[\![e^*]\!] & = & [\![e]\!]^*
\end{array}$$

where $[\![e_1]\!][\![e_2]\!] = \{xy \mid x \in [\![e_1]\!], y \in [\![e_2]\!]\}$ and $[\![e]\!]^* = \bigcup_{i \geq 0} [\![e]\!]^i$.

The Synthesis Problem Our goal is to synthesize a minimal regular expression that is consistent with given examples. The examples are given by a pair $(\mathcal{P}, \mathcal{N})$ of example strings, where $\mathcal{P} \subseteq \Sigma^*$ is a set of *positive examples* and $\mathcal{N} \subseteq \Sigma^*$ is a set of *negative examples*. We would like to find a regular expression that accepts all the positive examples while rejecting all the negative examples. There are an infinite number of such regular expressions. Among them, we aim to find a minimal one with respect to a given cost model $\mathcal{C} : e \to \mathbb{N}$. Formally, the goal is to synthesize a regular expression e such that

$$\forall p \in \mathcal{P}. p \in [\![e]\!] \land \forall n \in \mathcal{N}. n \notin [\![e]\!]$$

holds and the cost $\mathcal{C}(e)$ is minimized. The following is the example cost model used in our tool, in which c_1 through c_4 are constants representing costs associated with each variable and operator:

$$\begin{array}{rcl}
\mathcal{C}(a) = \mathcal{C}(\epsilon) = \mathcal{C}(\emptyset) & = & c_1 \\
\mathcal{C}(e_1 + e_2) & = & \mathcal{C}(e_1) + \mathcal{C}(e_2) + c_2 \\
\mathcal{C}(e_1 \cdot e_2) & = & \mathcal{C}(e_1) + \mathcal{C}(e_2) + c_3 \\
\mathcal{C}(e^*) & = & \mathcal{C}(e) + c_4
\end{array} \tag{2}$$

$$\frac{s_1 \to s_1'}{s_1 + s_2 \to s_1' + s_2} \qquad \frac{s_2 \to s_2'}{s_1 + s_2 \to s_1 + s_2'}$$

$$\frac{s_1 \to s_1'}{s_1 \cdot s_2 \to s_1' \cdot s_2} \qquad \frac{s_2 \to s_2'}{s_1 \cdot s_2 \to s_1 \cdot s_2'}$$

$$\frac{s \to s'}{s^* \to s'^*}$$

$$\frac{}{\square \to a}\ a \in \Sigma \qquad \frac{}{\square \to \epsilon} \qquad \frac{}{\square \to \emptyset}$$

$$\frac{}{\square \to \square + \square} \qquad \frac{}{\square \to \square \cdot \square} \qquad \frac{}{\square \to \square^*}$$

Figure 2. Transition Relation between States

Intuitively, we prefer simpler expressions (e.g. expressions with fewer variables) assuming that the simpler regular expressions are, the more general they are according to the principle of Occam's razor. Our implementation also prefers regular expressions in the concatenation form rather than the union (i.e. $c_2 > c_3$), because it is less likely to overfit the examples.

4. Algorithm

In this section, we present our algorithm for synthesizing regular expressions from examples. Our algorithm basically performs the enumerative search that considers all regular expressions in the order defined by the cost model. Section 4.1 describes the basic search algorithm. However, this algorithm is too slow to be interactively used. Section 4.2 presents the techniques we used to effectively prune out the large search space in a sound way.

4.1 Basic Search Algorithm

Suppose a pair $(\mathcal{P}, \mathcal{N})$ of positive and negative examples is given. We formulate the problem of finding a regular expression as a search problem. Consider a transition system $(S, \to, I, F)$, where S is the set of states, $(\to) \subseteq S \times S$ is a transition relation, $I \in S$ is an initial state, and $F \subseteq S$ is a set of final, solution states.

- **States**: A state $s \in S$ is a partial regular expression that possibly has holes ($\square$). A hole is a placeholder that can be replaced by another regular expression. Note that we extend our cost model to include $\mathcal{C}(\square) = c_5$. Since we prefer closed expressions to expressions with holes, we make c_5 greater than any other constants in the model. The set S of states is inductively defined as follows:

  $$s \to a \in \Sigma \mid \epsilon \mid \emptyset \mid s_1 + s_2 \mid s_1 \cdot s_2 \mid s^* \mid \square \tag{3}$$

  Note that a state may have multiple holes. For example, $(a + (\square \cdot \square))^*$ is a state which has two holes in it.
- **Initial State**: The initial state is a single hole, i.e., $I = \square$.
- **Transition Relation**: The transition relation $(\to) \subseteq S \times S$ determines the next states of a given state. It is inductively defined as a set of inference rules in Figure 2. For

example, we have $(a+\square)^* \to (a+(\square\cdot\square))^*$ because we can find a derivation tree according to the inference rules as follows:

$$\cfrac{\cfrac{\overline{\square \to \square\cdot\square}}{(a+\square) \to (a+(\square\cdot\square))}}{(a+\square)^* \to (a+(\square\cdot\square))^*}$$

We write $\mathsf{next}(s)$ for the set of all states that follow s:

$$\mathsf{next}(s) = \{s' \mid s \to s'\}.$$

For example, when $\Sigma = \{0,1\}$, $\mathsf{next}(0+\square) = \{(0+0),(0+1),(0+\epsilon),(0+\emptyset),(0+(\square+\square)),(0+(\square\cdot\square)),(0+(\square^*))\}$. We write $s \not\to$ to indicate that s has no next states; that is, s is a closed expression with no holes.

- **Solution States**: A state s is a solution state iff s is a closed expression (i.e. $s \not\to$) and s is consistent with the given positive and negative examples:

$$\mathsf{solution}(s) \iff s \not\to \;\wedge\; \forall p \in \mathcal{P}.p \in [\![s]\!] \wedge \forall n \in \mathcal{N}.n \notin [\![s]\!].$$

Algorithm 1 presents the workset algorithm that solves the search problem. Initially, the workset consists of the initial state (line 1). We choose and remove a state s from the workset (line 3). We pick a state that has a minimal cost with respect to the cost model $\mathcal{C}$ defined in Section 3 (2). If a solution is found, it is returned. Otherwise, we search for the next states of s by adding them into the workset.

Huge Search Space Although regular expressions are constructed from a simple grammar, the search space induced by the grammar is enormously huge, making a naive search algorithm intractable. Note that the maximum number of holes in a state at depth d of the search tree is 2^d, and the number of the next states of a state with n holes is c^n, where c is the number of inductive rules of the regular expression grammar (e.g., when $\Sigma = \{0,1\}$, $c = 7$). Therefore, the maximum number $N(d)$ of states at depth d is defined as follows:

$$\begin{aligned} N(0) &= 1 \\ N(d+1) &= N(d)\cdot c^{2^d} \end{aligned}$$

Solving the recurrent relation, we obtain

$$N(d) = c^{\sum_{k=0}^{d-1} 2^k} = c^{2^d-1}.$$

4.2 Pruning the Search Space

We present techniques for pruning the large search space. In order to search effectively, we identify and prune out semantically equivalent states, dead states, and redundant states. All of the pruning techniques are sound and guaranteed to find a solution state as long as it exists.

Pruning semantically equivalent states can be easily done by defining semantics-preserving transformation rules such

Algorithm 1 Search Algorithm

Input: Positive and negative examples $(\mathcal{P},\mathcal{N})$
Output: A regular expression E consistent with $(\mathcal{P},\mathcal{N})$

```
1: $W := \{\square\}$
2: repeat
3:     Pick a minimal cost state $s$ from $W$
4:     if $\mathsf{solution}(s)$ then return $s$
5:     else
6:         $W := W \cup \mathsf{next}(s)$
7:     end if
8: until $W \neq \emptyset$
```

as $(e^*)^* \to e^*$ and $e+e \to e$. When a state is added into the workset, we apply the transformation rules until it reaches a fixed point and do not add the state if the resulting fixed point has been processed before. We used 36 transformations in the implementation.

4.2.1 Dead States

Let us first define the set of dead states.

Definition 1 (Dead States)**.** *Let $(\mathcal{P},\mathcal{N})$ be a regular expression problem. We say a state $s \in S$ is dead, denoted* $\mathsf{dead}(s)$*, iff every closed state s' reachable from s is not a solution:*

$$\mathsf{dead}(s) \iff \big((s \to^* s') \;\wedge\; s' \not\to \implies \neg\mathsf{solution}(s')\big).$$

Intuitively, a state s is dead if exploring further the reachable states of s is guaranteed to fail to find a solution. Our search algorithm aims to identify as many dead states as possible and does not attempt to explore the search space beyond them. Specifically, we identify two types of dead states: pdead and ndead.

Definition 2. *A state s is dead for positive examples, denoted* $\mathsf{pdead}(s)$*, iff every closed state s' reachable from s fails to accept a positive example:*

$$\mathsf{pdead}(s) \iff \big(s \to^* s' \;\wedge\; s' \not\to \implies \exists p \in \mathcal{P}.\; p \notin [\![s']\!]\big).$$

Example 1. *Suppose $1 \in \mathcal{P}$. Any closed state s' reachable from state $s = 0\cdot\square$ is doomed to reject the positive example; no matter how the hole gets instantiated, the string 1 cannot be accepted.*

Definition 3. *A state s is dead for negative examples, denoted* $\mathsf{ndead}(s)$*, iff every closed state s' reachable from s fails to reject a negative example:*

$$\mathsf{ndead}(s) \iff \big(s \to^* s' \wedge s' \not\to \implies \exists n \in \mathcal{N}.\; n \in [\![s']\!]\big).$$

Example 2. *Suppose $0 \in \mathcal{N}$. Any closed state s' reachable from state $s = 0\cdot(\square)^*$ is doomed to accept the negative example; no matter how the hole gets instantiated, the language of any reachable state includes the string 0.*

It is clear that a state is guaranteed to be dead if one of $\mathsf{pdead}(s)$ and $\mathsf{ndead}(s)$ holds:

Lemma 1. *Let s be any state. Then,*

$$\big(\mathsf{pdead}(s) \;\vee\; \mathsf{ndead}(s)\big) \implies \mathsf{dead}(s).$$

Note that, however, the converse of Lemma 1 is not true. Suppose s is a dead state. This means that every reachable state s' either rejects some positive examples or accepts some negative examples. However, $\mathsf{pdead}(s)$ requires that the reachable state s' *always* rejects some positive example. Similarly, $\mathsf{ndead}(s)$ requires a strong condition that every reachable state s' accepts some negative example.

Example 3. *When $\mathcal{P} = \mathcal{N}$, no solution exists and the initial state $\square$ is dead. However, neither $\mathsf{pdead}(s)$ nor $\mathsf{ndead}(s)$ holds, because we can always find a regular expression (e.g. $(0+1)^*$) that accepts all positive examples and we can also always find a regular expression (e.g. $\emptyset$) that rejects all negative examples.*

We identify the pdead states and the ndead states by computing over- and under-approximations of states.

Definition 4. *The over- and under-approximations, resp., $\widehat{e}$ and $\widetilde{e}$, of a state are defined as follows:*

$$\begin{array}{rcl@{\qquad}rcl}
\widehat{a} &=& a & \widetilde{a} &=& a \\
\widehat{\epsilon} &=& \epsilon & \widetilde{\epsilon} &=& \epsilon \\
\widehat{\emptyset} &=& \emptyset & \widetilde{\emptyset} &=& \emptyset \\
\widehat{e_1 + e_2} &=& \widehat{e_1} + \widehat{e_2} & \widetilde{e_1 + e_2} &=& \widetilde{e_1} + \widetilde{e_2} \\
\widehat{e_1 \cdot e_2} &=& \widehat{e_1} \cdot \widehat{e_2} & \widetilde{e_1 \cdot e_2} &=& \widetilde{e_1} \cdot \widetilde{e_2} \\
\widehat{e^*} &=& (\widehat{e})^* & \widetilde{e^*} &=& (\widetilde{e})^* \\
\widehat{\square} &=& (0+1)^* & \widetilde{\square} &=& \emptyset
\end{array}$$

Intuitively, for a state s, the over-approximation $\widehat{s}$ is obtained by replacing all holes in s by $(0+1)^*$, and the under-approximation $\widetilde{s}$ is obtained by replacing the holes by $\emptyset$.

Example 4. *Consider a state $s = 0 + (\square + \square)$. Then, $\widehat{s} = 0 + ((0+1)^* + (0+1)^*)$ and $\widetilde{s} = 0 + (\emptyset + \emptyset)$.*

We have two simple facts regarding the approximations:

Lemma 2. *For any state s, we have $s \to^* \widehat{s}$ and $\widehat{s} \not\to$.*

Lemma 3. *For any state s, we have $s \to^* \widetilde{s}$ and $\widetilde{s} \not\to$.*

The state $\widehat{s}$ is over-approximated in a sense that the language of $\widehat{s}$ contains all the languages of states reachable from s, which is formalized by the following lemma:

Lemma 4. *For any state s, we have*

$$[\![\widehat{s}]\!] \supseteq \bigcup_{s \to^* s' \wedge s' \not\to} [\![s']\!].$$

Dually, $\widetilde{s}$ is under-approximated because every state s' reachable from s subsumes the language of $\widetilde{s}$, which is formalized by the following lemma:

Lemma 5. *For any state s, we have*

$$[\![\widetilde{s}]\!] \subseteq \bigcap_{s \to^* s' \wedge s' \not\to} [\![s']\!].$$

Given a state s, we conclude that s is dead with positive example (i.e. $\mathsf{pdead}(s)$) if $\widehat{s}$ rejects some positive example:

$$\exists p \in \mathcal{P}.\ p \notin [\![\widehat{s}]\!] \tag{4}$$

and we conclude that s is dead with negative example (i.e. $\mathsf{ndead}(s)$) if $\widetilde{s}$ accepts some negative example:

$$\exists n \in \mathcal{N}.\ n \in [\![\widetilde{s}]\!]. \tag{5}$$

Theorems 1 and 2 below show that our algorithm for identifying pdead and ndead states is *both* sound and complete.

Theorem 1. *Let s be any state. Then,*

$$\mathsf{pdead}(s) \iff \exists p \in \mathcal{P}.\ p \notin [\![\widehat{s}]\!].$$

Proof. Consider each direction.

- ($\Longrightarrow$) Suppose $\mathsf{pdead}(s)$ holds:

$$s \to^* s' \;\wedge\; s' \not\to \implies \exists p \in \mathcal{P}.\ p \notin [\![s']\!]. \tag{6}$$

  From (6) and Lemma 2, we obtain $\exists p \in \mathcal{P}.\ p \notin [\![\widehat{s}]\!]$.
- ($\Longleftarrow$) Suppose $p \notin [\![\widehat{s}]\!]$. By Lemma 4, we have

$$p \notin \bigcup_{s \to^* s' \wedge s' \not\to} [\![s']\!].$$

  which implies that $p \notin [\![s']\!]$ for all closed s' reachable from s.

□

Theorem 2. *Let s be any state. Then,*

$$\mathsf{ndead}(s) \iff \exists n \in \mathcal{N}.\ n \in [\![\widetilde{s}]\!].$$

Proof. Consider each direction.

- ($\Longrightarrow$) Suppose $\mathsf{ndead}(s)$ holds:

$$s \to^* s' \;\wedge\; s' \not\to \implies \exists n \in \mathcal{N}.\ n \in [\![s']\!]. \tag{7}$$

  From (7) and Lemma 3, we obtain $\exists n \in \mathcal{N}.\ n \in [\![\widetilde{s}]\!]$.
- ($\Longleftarrow$) Suppose $n \in [\![\widetilde{s}]\!]$. By Lemma 5, we have

$$n \in \bigcap_{s \to^* s' \wedge s' \not\to} [\![s']\!]$$

  which implies that $n \in [\![s']\!]$ for all closed s' reachable from s.

□

4.2.2 Redundant States

Let us first illustrate the idea by a series of examples.

Example 5. *Suppose we are given the positive examples* $\mathcal{P} = \{001, 101, 0011, 1011, 10111\}$ *and suppose further that we encounter the state* $s_1 = (0+1)\cdot 0 \cdot 0^* \cdot \square$ *during the search. Note that this state is not dead with respect to the positive examples (i.e. we can obtain a regular expression* $(0+1)\cdot 0\cdot 0^*\cdot 1^*$ *by replacing* $\square$ *by* 1^**, which accepts all the positive examples). However, the part* 0^* *of* s_1 *is redundant in that it can accept the same set of examples without* 0^**. A simpler, hence more desirable solution in this case is* $(0+1)\cdot 0\cdot 1^*$.

Example 6. *Suppose that we have a set of positive examples that does not have the string* 0 *(i.e.* $0 \notin \mathcal{P}$*). Then, the state* $0 + \square$ *is redundant because the part* 0 *is needless to accept the positive examples.*

Example 7. *Let* $\mathcal{P} = \{0, 01, 011, 0111\}$*. Then, the states* $1^*\cdot\square$ *and* $0^*\cdot\square$ *are both redundant. Although they can develop into solution states, we can find a simpler solution that does not have* 1^* *or* 0^* *as a prefix.*

We aim to eliminate those redundant states and guide the search to more effectively find the desired solution. Our algorithm is based on a heuristic that is specialized to identifying states that have the unions ($e_1 + e_2$) or closures (e^*) as the redundant parts. We informally describe the algorithm and explain why it is sound.

We detect redundant closures via unrolling. Consider the positive examples in Example 7 and suppose the current state is $1^*\cdot\square$. To check if the state has redundant closures, we unroll every outermost closure for two times: the state $1^*\cdot\square$ is transformed into $1\cdot 1\cdot 1^*\cdot\square$. Next, we check if any positive example can be accepted by the unrolled state. In Example 7, no strings in $\mathcal{P}$ can be accepted by $1\cdot1\cdot1^*\cdot\square$. Therefore, we determine the closure expression 1^* in the original state to be redundant and prune out the state $1^*\cdot\square$. Similarly, when the current state is $0^*\cdot\square$, we unroll the closure for two times, i.e., $0\cdot 0\cdot 0^*\cdot\square$, and check if the unrolled state can accept any positive example. Since it is not the case, we eliminate the original state. This pruning strategy does not impair the soundness; the algorithm is still guaranteed to find a solution. To see why, suppose that the state $1^*\cdot\square$ can accept all positive examples but the unrolled one $1\cdot1\cdot1^*\cdot\square$ does not accept any of the examples. This means that the regular expression denoting the difference of the two languages (i.e. $[\![1^*\cdot(0+1)^*]\!] - [\![1\cdot1\cdot1^*\cdot(0+1)^*]\!]$) must accept all positive examples. Since regular languages are closed under the set difference, we can always find a regular expression for the difference, i.e., $(1+\epsilon)\cdot(0+1)^*$.

We detect redundant unions by decomposing union expressions into separate cases. For example, consider Example 6 and suppose the current state is $0+\square$. We first decompose the state into the set of states $\{0, \square\}$. Then, we check if there is a redundant state in the set, which rejects all the positive examples. If so, we prune out the original state.

These ideas can be stated in a unified algorithm. We use the following functions unroll and split for unrolling and decomposition, respectively:

$$\begin{array}{rcl}
\mathsf{unroll}(c) &=& c\ (c \in \Sigma \cup \{\epsilon, \emptyset\}) \\
\mathsf{unroll}(e_1 + e_2) &=& \mathsf{unroll}(e_1) + \mathsf{unroll}(e_2) \\
\mathsf{unroll}(e_1 \cdot e_2) &=& \mathsf{unroll}(e_1) \cdot \mathsf{unroll}(e_2) \\
\mathsf{unroll}(e^*) &=& e \cdot e \cdot e^* \\
\mathsf{unroll}(\square) &=& \square \\
\\
\mathsf{split}(c) &=& \{c\}\ (c \in \Sigma \cup \{\epsilon, \emptyset\}) \\
\mathsf{split}(e_1 + e_2) &=& \mathsf{split}(e_1) \cup \mathsf{split}(e_2) \\
\mathsf{split}(e_1 \cdot e_2) &=& \{e_1' \cdot e_2, e_1 \cdot e_2' \mid e_i' \in \mathsf{split}(e_i)\} \\
\mathsf{split}(e^*) &=& \{e^*\} \\
\mathsf{split}(\square) &=& \{\square\}
\end{array}$$

As for redundant states, we unroll the closure expression twice because doing so was most effective in practice. Note that the number of unrolling affects the performance, but not the soundness of the search algorithm.

Example 8. *Consider a state* $s = (0^*\cdot 1)^*\cdot\square$*. Then,* $\mathsf{unroll}(s) = (0^*\cdot1)\cdot(0^*\cdot1)\cdot(0^*\cdot1)^*\cdot\square$.

Example 9. *Consider a state* $s = (0+1)\cdot(1^*+0\cdot1)\cdot\square$*. Then,* $\mathsf{split}(s) = \{0\cdot(1^*+0\cdot1)\cdot\square, 1\cdot(1^*+0\cdot1)\cdot\square, (0+1)\cdot1^*\cdot\square, (0+1)\cdot(0\cdot1)\cdot\square\}$.

With unroll and split, we identify redundant states as follows. Let s be a state. To check the redundancy, we unroll and decompose the state. Let A be the result, i.e., $A = \mathsf{split}(\mathsf{unroll}(s))$. We determine the state s to be redundant if A has some state that rejects all positive examples:

$$\exists s' \in A.\ (s' \to s'' \ \land\ s'' \not\to \implies \forall p \in \mathcal{P}.\ p \notin [\![s'']\!])$$

We can check the condition by using the over-approximation:

$$\exists s' \in A.\ \forall p \in \mathcal{P}.\ p \notin [\![\widehat{s'}]\!]).$$

4.2.3 Final Algorithm

To incorporate the pruning techniques into the search algorithm, we modify the next function as follows:

$$\mathsf{next}(s) = \begin{cases} \emptyset & \exists p \in \mathcal{P}.\ p \notin [\![\widehat{s}]\!] \\ \emptyset & \exists n \in \mathcal{N}.\ n \in [\![\check{s}]\!] \\ \emptyset & \exists e \in A. \forall p \in \mathcal{P}. p \notin [\![\widehat{e}]\!] \\ \{s' \mid s \to s'\} & \text{otherwise} \end{cases}$$

where $A = \mathsf{split}(\mathsf{unroll}(s))$.

5. Evaluation

Implementation We implemented our synthesis algorithm in a tool AlphaRegex. In current implementation of AlphaRegex, input characters for an example are 0, 1, and x.

Level	No	Description	Examples		Full	No Apr	No Rd	Output
			Pos	Neg	sec	sec	sec	
Easy	1	Start with 0.	3	3	0.0	0.0	0.0	0(0+1)*
	2	End with 01.	3	6	0.0	0.1	0.0	(0+1)*01
	3	Contain the substring 0101.	3	7	1.4	10.6	1.9	(0+1)*0101(0+1)*
	4	Begin with 1 and end with 0.	3	5	0.0	0.0	0.0	1(0+1)*0
	5	Length is at least 3 and the 3rd symbol is 0.	3	4	0.0	0.0	0.0	(0+1)(0+1)0(0+1)*
	6	Length is a multiple of 3.	2	3	0.1	0.1	0.1	((0+1)(0+1)(0+1))*
Normal	7	The number of 0s is divisible by 3.	8	7	9.5	238.4	36.9	(1+01*01*0)*
	8	Even number of 0s.	7	7	0.1	1.0	0.2	(1+01*0)*
	9	The 5th symbol from the right end is 1.	3	3	9.0	56.3	14.3	(0+1)*1(0+1)(0+1)(0+1)(0+1)
	10	0 and 1 alternate.	9	8	1.7	19.7	3.5	1?(01)*0?
	11	Each 0 is followed by at least one 1.	7	6	0.0	0.0	0.0	(0?1)*
	12	0^n1^m where $n \geq 3$ and m is even.	6	5	0.2	1.0	0.7	000*0(11)*
	13	Have at most two 0s.	8	7	1.4	9.5	2.3	1*0?1*0?1*
	14	Start with 0 and have odd length or start with 1 and have even length.	5	5	16.5	771.9	14.7	(0+1(0+1))((0+1)(0+1))*
	15	Any strings except 0 and 1.	3	2	0.0	0.0	0.0	(0+1)(0+1)(0+1)*
	16	Do not end with 01.	8	3	17.8	302.1	15.2	0+1+(0+1)*(0+1(0+1))
Hard	17	Contain at least one 0 and at most one 1.	6	9	2.7	33.5	7.7	0*(01?+100*)
	18	At least two occurrences of 1 between any two occurrences of 0.	7	7	0.2	0.4	0.2	0?(1(10)?)*
	19	Do not contain 100 as a substring.	8	4	0.1	0.1	0.1	0*(1(0+1)?)*
	20	Every odd position is 1.	6	9	3.6	10.5	5.0	(1(0+1))*1*
	21	Have exactly one pair of consecutive 0s.	4	4	1.0	16.1	2.9	(0?1)*00(10?)*
	22	Do not end with 10 and have a length of at least two	9	4	57.2	4248.0	75.3	(0+1)(0+1)*1+(0+1)*0(0+1)
	23	Even number of 0s and each 0 is followed by at least one 1.	6	9	1.8	30.2	4.3	(1+01*101)*
	24	Every pair of adjacent 0s appears before any pair of adjacent 1s.	9	9	14.4	506.1	73.7	0?((01)?10?)*
	25	At most one pair of consecutive 1s.	5	5	28.8	863.6	211.2	(1?0)*1?1?(01?)*
Average					6.7	284.8	18.8	

Table 3. Performance of AlphaRegex: the output and elapsed time in seconds for each given problem (Pos: positive examples, Neg: negative examples, Full: all pruning techniques applied, No Apr: all pruning techniques applied except for over- and under-approximations, No Rd: all pruning techniques applied except for identifying redundant states, Output: the output synthesized by AlphaRegex with all pruning techniques). We used the wild card x in the examples for the problems 1–6, 9, 14–16, 19, 20, and 22.

0 and 1 are symbols from the binary alphabet Σ. For convenience, users can use a wild card x to represent a symbol which can be both 0 and 1. When x is used in any example, we use the expression $0+1$ as an idiom; it is used as a primitive building block for synthesizing regular expressions. The implementation supports $e? = (e + \epsilon)$ as a syntactic sugar.

Benchmarks To evaluate our synthesis algorithm, we collected 25 benchmark problems from textbooks on automata theory. We classified the problems based on their level of difficulty: easy, normal, and hard. According to our experience, most students could solve the easy problems. About half of the students could correctly solve the normal level of problems. Most students found it difficult to solve the hard problems; they either failed to solve them or ended up with solutions far from being optimal.

Performance The main purpose of the experimental evaluation was to assess how fast AlphaRegex can synthesize regular expressions for the given benchmark suites. The overall result shows that AlphaRegex can synthesize our non-trivial benchmark problems efficiently. The column labeled as "Full" in Table 3 shows the running time of AlphaRegex equipped with all pruning techniques we explained in Section 4. In that case, the average runtime is 6.7 seconds and the median runtime is 1.4 seconds. AlphaRegex is able to synthesize all of the benchmarks within 1 minute. All experiments were conducted on an Ubuntu machine with Intel Xeon CPU E5-2630 (2.40GHz).

Effectiveness of the pruning techniques We evaluated the effectiveness of our pruning techniques. To examine the impact of over- and under-approximations, we modified our algorithm not to identify any dead states. The column labeled "No Apr" in Table 3 shows the running time of ALPHAREGEX that does not perform over- and under-approximations. As shown in the column, the impact of them was huge: without identification of pdead and ndead states, the average runtime was 248.8 seconds, whereas the algorithm fully equipped with over- and under-approximations was 42.5 times faster than the modified algorithm without over- and under-approximations.

To evaluate the impact of identifying redundant states, we also modified our algorithm not to identify any redundant states. The running time of the modified algorithm is shown on the column "No Rd" in Table 3. The result shows that identifying redundant states is effective, but not as mush as identifying dead states using over- and under-approximations. Nevertheless, the impact is still notable; on the average, the fully equipped algorithm including identifying redundant states was 2.8 times faster than the modified algorithm without detecting them. In 14th and 16th problems, the original algorithm was slightly slower than the modified algorithm because of computational overhead for identifying such states.

Discussion During the experiment, we found that it is important to choose proper input examples. Especially, we consider the choices of examples when different sets of examples result in the same regular expression. In that case, time cost for the synthesis depends on the choices of examples. Our main observations are as follows:

- Provide specific examples for positive examples.
- Provide general examples for negative examples.
- Provide examples as few as possible.

Intuitively, it is better to be specific when providing positive examples. It is because when we prune out the search space using over-approximation, our method prunes out the states that fail to accept all the positive examples. Hence, the more specific the examples are, the more restrictions they put on. Likewise, it is better to be general for negative examples since under-approximation prunes out the states that fail to reject any negative example. That is, general negative examples are likely to be accepted by more states and we can prune out the states effectively. We also found that providing fewer examples results in better performance most of the time. It is mainly due to the fact that the time cost for checking consistency with examples greatly increases as synthesizing more complex regular expressions. Thus, users are encouraged to provide examples as few as possible.

6. Related Work

In this section, we describe several related works to our technique from the areas of program synthesis to learning and inferring regular expressions.

Program synthesis Our work has been inspired from recent advances of the program synthesis technology [1, 3, 15, 16, 25]. Recently, program synthesis techniques have been successfully used for synthesizing programs from examples that manipulate recursive functions [1], data structures [15], numbers [25], and strings [16], etc. On the other hand, we use the program synthesis technology for synthesizing regular expressions from examples. In particular, we use a search-based synthesis that is similar to the approach by Feser et al. [15]. However, we use the algorithm for a different purpose (i.e. synthesizing regular expressions) with different pruning techniques specialized for regular expressions. The synthesis language (called Flare) in [3] is a regular expression extended with geometric constructs. However, our work focuses on regular expressions for automata assignments while Flare is a domain-specific language for extracting relational data.

Text classification Learning regular languages from examples for text classification has been the subject of active research [11–13, 24, 27, 28]. It aims to learn regular expressions as a classifier for languages, which can distinguish positive and negative examples. For instance, it learns regular expressions from examples which are links to other web pages and detects HTML lines containing such links [12].

Unlike many of these efforts which aim to generalize beyond the provided examples, there is another approach to classify the examples with overfitting, namely regex golf [7]. Its goal is to find minimum regular expressions that are consistent with given sets of positive and negative examples, which is very similar to ours. However, we found that regex golf (`http://regex.inginf.units.it/golf/`) fails to solve most of our benchmark problems except for some of the easy problems. It is because it uses Genetic Programming (GP) [19] to find regular expressions with minimal cost with respect to the number of matches and the length of regular expressions. Since our method only takes a few examples, GP is unlikely to perform well because it is hard to rank candidate regular expressions for evolving next generations.

Text extraction Regular expressions are widely used for text extraction. Thus, there have been extensive works for learning regular expressions from examples in the context of text extraction [4–6, 8, 10]. The primary goal of this approach is to find regular expressions that generalize the extraction behavior represented by examples. For example, RegexGenerator++ [6] learns regular expressions from a set of input strings and substrings in each string to be extracted. This problem is different from ours in that it requires users to annotate the parts of examples to be extracted, where examples are typically a text file composed of tens or hundreds of lines.

Learning DFA from examples It is possible to learn DFAs from examples and convert them to regular expressions (instead of learning of regular expressions from examples directly). The problem of learning DFAs from examples is long-established [2, 9, 23] and we can easily convert DFAs to regular expressions with standard algorithms. However, there are crucial disadvantages to apply this approach to our system. First of all, learning DFA does not concern with minimizing regular expressions. Since regular expression minimization is computationally hard, namely PSPACE-complete [21], it introduces another difficult problem to solve. In addition, many methods for learning DFA require training data to meet certain properties [22, 23]. For example, the work in [22] requires a structurally complete training set which covers every state transition in the target DFA. However, it is simply unreasonable to assume that users know the internal structure of the target DFA, which is necessary to construct training data.

Logical inference There are many algorithms proposed that can infer regular languages from examples [14, 18]. However, most of inference algorithms are based on grammars and automata instead of regular expressions. The work by Fernau [14] proposes algorithms to directly infer regular expressions from positive examples. Yet, the algorithms can only generate simple regular expressions and have limitations regarding the form of expressions. Another work by Kinber [18] requires representative examples and membership queries to be answered by users to infer regular expressions without union operations. In contrast, our method performs synthesis without restricting the extent of regular expressions synthesized or imposing any requirement on input examples.

7. Conclusion

We presented an algorithm for synthesizing regular expressions from examples for introductory automata assignments. The algorithm is based on an enumerative search with a set of techniques for pruning a large search space in sound ways. We implemented the algorithm in a tool, ALPHAREGEX, and showed that the tool is able to quickly synthesize desired solutions for non-trivial regular expression problems.

Acknowledgements This work was supported by the Institute for Information & communications Technology Promotion (IITP) grant funded by the Korea government (MSIP) (No.R0190-15-2011, Development of Vulnerability Discovery Technologies for IoT Software Security) and the Basic Science Research Program through the National Research Foundation of Korea (NRF) funded by the Ministry of Science, ICT & Future Planning (NRF-2016R1C1B2014062).

References

[1] Aws Albarghouthi, Sumit Gulwani, and Zachary Kincaid. Recursive program synthesis. In *Proceedings of the 25th International Conference on Computer Aided Verification - Volume 8044*, CAV 2013, pages 934–950, New York, NY, USA, 2013. Springer-Verlag New York, Inc.

[2] Dana Angluin. Learning regular sets from queries and counterexamples. *Inf. Comput.*, 75(2):87–106, November 1987.

[3] Daniel W. Barowy, Sumit Gulwani, Ted Hart, and Benjamin Zorn. Flashrelate: Extracting relational data from semi-structured spreadsheets using examples. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '15, pages 218–228, New York, NY, USA, 2015. ACM.

[4] David F. Barrero, María D. R-Moreno, and David Camacho. Adapting searchy to extract data using evolved wrappers. *Expert Syst. Appl.*, 39(3):3061–3070, February 2012.

[5] Alberto Bartoli, Giorgio Davanzo, Andrea De Lorenzo, Marco Mauri, Eric Medvet, and Enrico Sorio. Automatic generation of regular expressions from examples with genetic programming. In *Proceedings of the 14th Annual Conference Companion on Genetic and Evolutionary Computation*, GECCO '12, pages 1477–1478, New York, NY, USA, 2012. ACM.

[6] Alberto Bartoli, Giorgio Davanzo, Andrea De Lorenzo, Eric Medvet, and Enrico Sorio. Automatic synthesis of regular expressions from examples. *Computer*, 47(12):72–80, December 2014.

[7] Alberto Bartoli, Andrea De Lorenzo, Eric Medvet, and Fabiano Tarlao. Playing regex golf with genetic programming. In *Proceedings of the 2014 Annual Conference on Genetic and Evolutionary Computation*, GECCO '14, pages 1063–1070, New York, NY, USA, 2014. ACM.

[8] Alberto Bartoli, Andrea De Lorenzo, Eric Medvet, and Fabiano Tarlao. Inference of regular expressions for text extraction from examples. *IEEE Trans. on Knowl. and Data Eng.*, 28(5):1217–1230, May 2016.

[9] Josh Bongard and Hod Lipson. Active coevolutionary learning of deterministic finite automata. *J. Mach. Learn. Res.*, 6:1651–1678, December 2005.

[10] Falk Brauer, Robert Rieger, Adrian Mocan, and Wojciech 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*, CIKM '11, pages 1285–1294, New York, NY, USA, 2011. ACM.

[11] Duy Duc An Bui and Qing Zeng-Treitler. Learning regular expressions for clinical text classification. *Journal of the American Medical Informatics Association*, 21(5):850–857, 2014.

[12] Ahmet Cetinkaya. Regular expression generation through grammatical evolution. In *Proceedings of the 9th Annual Conference Companion on Genetic and Evolutionary Computation*, GECCO '07, pages 2643–2646, New York, NY, USA, 2007. ACM.

[13] B. D. Dunay, F. E. Petry, and B. P. Buckles. Regular language induction with genetic programming. In *Evolutionary Computation, 1994. IEEE World Congress on Computational Intelligence., Proceedings of the First IEEE Conference on*, pages 396–400 vol.1, Jun 1994.

[14] Henning Fernau. Algorithms for learning regular expressions from positive data. *Inf. Comput.*, 207(4):521–541, April 2009.

[15] John K. Feser, Swarat Chaudhuri, and Isil Dillig. Synthesizing data structure transformations from input-output examples. In *Proceedings of the 36th ACM SIGPLAN Conference on*

Programming Language Design and Implementation, PLDI '15, pages 229–239, New York, NY, USA, 2015. ACM.

[16] Sumit Gulwani. Automating string processing in spreadsheets using input-output examples. In *Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '11, pages 317–330, New York, NY, USA, 2011. ACM.

[17] John E. Hopcroft. *Introduction to Automata Theory, Languages, and Computation*. Pearson Addison Wesley, 3rd edition, 2007.

[18] Efim Kinber. Learning regular expressions from representative examples and membership queries. In *Proceedings of the 10th International Colloquium Conference on Grammatical Inference: Theoretical Results and Applications*, ICGI'10, pages 94–108, Berlin, Heidelberg, 2010. Springer-Verlag.

[19] John R. Koza. *Genetic Programming: On the Programming of Computers by Means of Natural Selection*. MIT Press, Cambridge, MA, USA, 1992.

[20] Peter Linz. *An Introduction to Formal Language and Automata*. Jones and Bartlett Publishers, Inc., USA, 2006.

[21] A. R. Meyer and L. J. Stockmeyer. The equivalence problem for regular expressions with squaring requires exponential space. In *Proceedings of the 13th Annual Symposium on Switching and Automata Theory (Swat 1972)*, SWAT '72, pages 125–129, Washington, DC, USA, 1972. IEEE Computer Society.

[22] Rajesh Parekh and Vasant Honavar. An incremental interactive algorithm for regular grammar inference. In *Proceedings of the Thirteenth National Conference on Artificial Intelligence - Volume 2*, AAAI'96, pages 1397–1397. AAAI Press, 1996.

[23] Rajesh Parekh and Vasant G. Honavar. Learning dfa from simple examples. *Mach. Learn.*, 44(1-2):9–35, July 2001.

[24] Paul Prasse, Christoph Sawade, Niels Landwehr, and Tobias Scheffer. Learning to identify concise regular expressions that describe email campaigns. *J. Mach. Learn. Res.*, 16(1):3687–3720, January 2015.

[25] Rishabh Singh and Sumit Gulwani. Synthesizing number transformations from input-output examples. In *Proceedings of the 24th International Conference on Computer Aided Verification*, CAV'12, pages 634–651, Berlin, Heidelberg, 2012. Springer-Verlag.

[26] Michael Sipser. *Introduction to the Theory of Computation*. Cengage Learning, 3rd edition, 2012.

[27] Borge Svingen. Learning regular languages using genetic programming. In John R. Koza, Wolfgang Banzhaf, Kumar Chellapilla, Kalyanmoy Deb, Marco Dorigo, David B. Fogel, Max H. Garzon, David E. Goldberg, Hitoshi Iba, and Rick Riolo, editors, *Genetic Programming 1998: Proceedings of the Third Annual Conference*, pages 374–376, University of Wisconsin, Madison, Wisconsin, USA, 22-25 July 1998. Morgan Kaufmann.

[28] Yinglian Xie, Fang Yu, Kannan Achan, Rina Panigrahy, Geoff Hulten, and Ivan Osipkov. Spamming botnets: Signatures and characteristics. In *Proceedings of the ACM SIGCOMM 2008 Conference on Data Communication*, SIGCOMM '08, pages 171–182, New York, NY, USA, 2008. ACM.