Efficient Methods for Solving String Constraints with Large Repetition Bound

Denghang Hu[1] and Zhilin Wu[1]

State Key Laboratory of Computer Science, Institute of Software, Chinese Academy of Sciences, China
{hudh,wuzl}@ios.ac.cn

Abstract. Regular expression with large repetition bounds frequently appears in the real world. However, state-of-art string solvers can not efficiently solve string constraints with such regular expression, which slows down string solver applications like formal verification and symbolic execution. In this paper, we propose a new systematic algorithm based on cost-enriched automaton. We use heuristic ways like over-approximation and under-approximation to accelerate the search. Further, we extract amounts of regex expressions with large repetition bounds from real-world programs and generate a significant benchmark. We evaluate the algorithm's performance on our developed benchmarks and other typical benchmarks. The result shows that our solver outperforms the state-of-art string solvers.

Keywords: string constraints, cost-enriched automata, regex expression, bounded repetition

1 Introduction

Regular expressions are widely used in programming languages. About 30–40 % of Java, JavaScript, and Python software uses regex matching [Davis(2019)]. To handle the problems in software verification reasoning about string, the string theory is introduced and amounts of string solvers are developed. However, the string theory is one of the most challenging theories because it is easy to be undecidable [Ganesh and Berzish(2016)]. String solvers try very hard to improve efficiency: DPLL(T)-based string solver Z3 [?] and Cvc5 [?] apply many heuristic derivation rules. Automaton-based string solver Trau+[?] proposed flat automaton and uses the CEGAR framework to under-approximate and over-approximate the string constraints again and again. Unfortunately, the string solvers above perform poorly when the string constraints contain large repetition bounds. [add a example]. To address this issue, we extend cost-enriched finite automaton(CEFA)[Chen et al.(2020)] with linear arithmetical constraint. Use backward-propagating algorithm to search a solution. Our main contributions are as follows:

1. Extend cost-enriched finite automaton to model the regular expression with large repetition bound.

2. Devise an efficient algorithm based on CEFA to solve string constraints with large repetition bound.
3. Generate many instances with large repetition bound from real-word regular expressions.
4. Implemented our efficient algorithm on Ostrich?? and compared it with state-of-art string solvers. The result shows the superiority of our automaton model and algorithm.

2 Preliminaries

Tokens. A finite *alphabet* Σ is the set of all *letters*. A *string* is a finite sequence of letters from Σ. We use Σ^* to denote the set of strings over Σ, ϵ to denote the empty string, and $a, b, \cdots$ to denote constant letters in Σ. We use $u, v, \cdots$ to denote constant string and $x, y, \cdots$ to denote variable string. Moreover, we also consider a set of natural integer numbers $\mathbb{N} = \{0, 1, 2, 3, \cdots\}$ and a set of integer numbers $\mathbb{Z} = \{\cdots, -1, 0, 1, \cdots\}$. We use $m, n, \cdots$ to denote integer constant, and $i, j \cdots$ to denote integer variable. For vector, we use $\vec{v}$ to denote the vector of integer constant, 0_n to denote the zero vector with length n, $v(i)$ to denote the integer value of $\vec{v}$ at position i, $\vec{v_1} \cdot \vec{v_2}$ to denote the concatenation of $\vec{v_1}$ and $\vec{v_2}$, R to denote the vector of integer variable, $R_1 \cap R_2$ to denote the same variables of R_1 and R_2, and $|\vec{v}|$ or $|R|$ to denote the length of vector.
CNF, DNF, clause, cube. We assume that the reader is familiar with first-order logic. A *literal* is an atomic proposition or its negation. A *clause* is a disjunction of literals. The empty clause is *true*. A formula is in *conjunctive normal norm* (CNF) if it is a conjunction of clauses. A *cube* is a conjunction of a consistent set of literals; The empty cube is $false$. A formula is in *disjunctive normal form* (DNF) if it is a disjunction of cubes.
Basic Regular Language. A nondeterministic finite state automaton (NFA) is a tuple $\mathcal{A} = (Q, \Sigma, \delta, I, F)$ where Q is a finite set of states; Σ is a finite alphabet; $\delta \in Q \times \Sigma \times Q$ is the transition relation; $I, F \subseteq Q$ are the set of initial states and finite states respectively. We write a transition $(q, a, q') \in \delta$ as $q \xrightarrow{a} q'$ for readability. A *run* of an NFA $\mathcal{A}$ on a string $w = a_1 \cdots a_n$ is a sequence of transitions $q_0 \xrightarrow{a_1} q_1 \cdots q_{n-1} \xrightarrow{a_n} q_n$ with $q_0 \in I$. The run is *accepting* if $q_n \in F$. A word w is *accepted* by an NFA $\mathcal{A}$ if there is an accepting run on w. The *language* of $\mathcal{A}$ is the set of all words accepted by $\mathcal{A}$, denoted by $\mathcal{L}(\mathcal{A})$. From automata theory[Hopcroft and Ullman(1979)], basic regular expression only contains union, concatenation, and closure operations. We know that the language of basic regular expression are also in the class of $\mathcal{L}(\mathcal{A})$. We call it *Basic Regular Language.*

3 String Theory with Regex

In this section, the syntax and semantic of string theory is defined.

3.1 Syntax

In this paper, we propose a new logic called *Extended String Logic (ESL)*. ESL contains the following sorts: string sort `Str`, integer sort `Int`, the enumerable set of string sort $\texttt{Str}^m$, and the enumerable set of integer sort $\texttt{Int}^m$ for each $m \in \mathbb{N}$. Furthermore, ESL includes the following predicate and function constants: string concatenation $\texttt{con} : \texttt{Str} \times \texttt{Str} \to \texttt{Str}$; string length $\texttt{len} : \texttt{Str} \to \texttt{Int}$; regular membership $\in: \texttt{Str} \times \texttt{Str}^m$; the regex operators such as closure, concatenation, conjunction, disjunction, intersection, repetition and complement; and the usual constants of linear arithmetic.

The syntax of ESL is presented in Table 1. φ is a first-order logic formula that can be a regular membership $x \in \mathcal{R}$, a word equation e, a quantifier-free Presburger arithmetic formula β, a negation of a formula, or a disjunction of two formulas. A word equation e is an equality of two string terms.β is an (in)equality of two integer terms and the (in)equality operator $\odot$ contains $=, \geq$. A string term s is an empty string ϵ, letters $a \in \Sigma$, string variable x, or a concatenation of two string terms. An integer term α is an integer constant 0, integer constant 1, integer variable i, the length of the string term s, minus an integer term, or plus of two integer terms. The regular expression $\mathcal{R}$ is built on empty string ϵ, constant letter $a \in \Sigma$. The supported regex operations involve concatenation $\cdot$, disjunction $+$, intersection $\times$, closure $*$, complement C, and repetition $\{m, n\}$.

We consider *straight-line* fragment[Chen et al.(2020)][Le and He(2018)] ESL_s of the string theory logic. A formula φ is said to be straight-line if it can be converted to a DNF formula $\bigvee_i \phi_i$ and $\phi_i = (\bigwedge_{i=1}^n e_i) \wedge \Re \wedge P$ where $\bigwedge_{i=1}^n e_i$ is straight-line, $\Re$ is a *cube* of regular membership $x \in \mathcal{R}$ or its negation $x \notin \mathcal{R}$, and P is a Presburger arithmetic formula. A word equation *cube* $\bigwedge_{i=1}^n e_i$ is said to be straight-line if it can be rewritten as the form $\bigwedge_{j=1}^m x_j = s_j$ such that:

1. $x_1, \cdots, x_m$ are different variables,
2. $FV(s_j) \cap \{x_1, x_2, \cdots, x_{j-1}\} = \emptyset$ where $FV(\varphi)$ are all free variables of φ.

We call straight-line formula $(\bigwedge_{i=1}^n e_i) \wedge \Re \wedge P$ *normal form.*We use $s_1 s_2$ to stand for $\texttt{concat}(s_1, s_2)$, $|x|$ to stand for $\texttt{len}(x)$.

$$
\begin{aligned}
&\mathcal{R} ::= \epsilon \mid a \mid \mathcal{R} \cdot \mathcal{R} \mid \mathcal{R} + \mathcal{R} \mid \mathcal{R} \times \mathcal{R} \mid \mathcal{R}^* \mid \mathcal{R}^C \mid \mathcal{R}\{m, n\} \\
&\varphi ::= x \in \mathcal{R} \mid e \mid \beta \mid \neg\varphi \mid \varphi \vee \varphi \\
&e ::= s_1 = s_2 \\
&\beta ::= \alpha_1 \odot \alpha_2 \\
&s ::= \emptyset \mid \epsilon \mid a \mid x \mid \texttt{concat}(s_1, s_2) \\
&\alpha ::= 0 \mid 1 \mid i \mid \texttt{len}(s) \mid -\alpha \mid \alpha_1 + \alpha_2
\end{aligned}
$$

Table 1: Syntax of ESL

Example 1 (Straight-line Formula).

$$\varphi_s ::= x = yz \wedge y = z \wedge x \in a\{1, 100\} \wedge |x| = 1 \quad (1)$$

φ_s is straight-line because we can split it to three parts: (i) a straight-line word equation cube $x = yz \wedge y = z$, (ii) a regular membership cube $x \in a\{1, 100\}$, and (iii) a Presburger arithmetic formula $|x| = 1$.

Example 2 (Non Straight-line Formula).

$$\varphi_{ns} ::= x = yz \wedge y = xz \wedge x \in a\{1, 100\} \wedge |x| = 1 \tag{2}$$

φ_{ns} is not straight-line because the word equation cube
$x = yz \wedge y = xz$ does not satisfy condition 2: $FV(yz) \cap \{y\} \neq \emptyset$ and $FV(xz) \cap \{x\} \neq \emptyset$.

3.2 Semantics

Regular Language We have introduced basic regular language at section 2. In addition to basic regular operations, we syntactically support intersection, complement and repetition. As we all know, basic regular language is closed under intersection and complement. Furthermore, the operation *repetition* $\mathcal{R}\{m, n\}$ means repeating regex $\mathcal{R}$ at least m times and at most n times. It can be syntactically rewritten by concatenation and union: $\mathcal{R}\{m, n\} \equiv \mathcal{R}^m \mid \cdots \mid \mathcal{R}^n$ where $\mathcal{R}^k$ defines concatenate k times($m \leq k \leq n$) to $\mathcal{R}$. So the regular language defined in Table 1 is semantically equal to basic regular language. However, if we naively rewrite the repetition operation, the automaton size will linear to the repetition times, causing the search space to be exponential. To address the issue, we proposed a new automaton model called *Cost-Enriched Finite Automaton (CEFA)*, whose size is not relevant to the repetition times. CEFA can not handle nested repetition (e.g., $\mathcal{R}\{m_1, n_1\}\mathcal{R}\{m_2, n_2\}$ or $(\mathcal{R}\{m, n\})*$) and complementation of repetition (e.g, $(\mathcal{R}\{m, n\})^C$). So we have to rewrite these regexes syntactically (e.g., $a\{1, 100\}\{2, 2\}$ is rewritten to $a\{1, 100\}a\{1, 100\}$).
Semantics We assume that S is the set of string variables over Σ^*, and I is the set of integer variables. $\eta : S \times \Sigma \rightarrow \Sigma^*$ is the interpretation on string where $\eta(c) = c$ for every letter $c \in \Sigma$ and $\eta(s_1 s_2) = \eta(s_1)\eta(s_2)$. $\theta : I \rightarrow \mathbb{Z}$ is the interpretation of arithmetic that is the same as Presburger arithmetic. Then the semantics is given by a satisfaction relation: $\eta, \theta \models \varphi$ defined in Table 2. We say a formula φ is *satisfiable* if a solution (η, θ) exists such as $\eta, \theta \models \varphi$. A formula φ is *unsatisfiable* if no solution exists.

$\eta, \theta \models \varphi_1 \vee \varphi_2$	iff $\eta, \theta \models \varphi_1$ or $\eta, \theta \models \varphi_2$
$\eta, \theta \models \neg\varphi$	iff $\eta, \theta \not\models \varphi$
$\eta, \theta \models x \in \mathcal{R}$	iff $\exists w \in \mathcal{L}(\mathcal{R}), \eta, \theta \models x = w$
$\eta, \theta \models s_1 = s_2$	iff $\eta(s_1) = \eta(s_2)$
$\eta, \theta \models \alpha_1 \odot \alpha_2$	iff $\theta(\alpha_1) \odot \theta(\alpha_2)$ where $\odot = \{=, \geq\}$

Table 2: Semantics

3.3 Problem

In this paper, we consider the following problem:

Problem:	Is an ESL_s formula φ satisfiable?
Input:	A straight-line ESL_s formula φ in the normal form.
Output:	*sat* or *unsat*.

4 Cost-Enriched Finite Automaton and Basic Operation

In this section, the cost-enriched finite automaton and operations on it is defined. The operations contains intersection, union, complement, concatenation, and repetition.

4.1 Cost-Enriched Finite Automaton

Definition 1 (Cost-Enriched Finite Automaton). *A cost-enriched finite automaton $\mathcal{A}$ is a tuple $(Q, \Sigma, \delta, I, F, R, \theta)$ where*

- *Q, Σ, I, F is defined as NFA,*
- *$R = (r_1 \cdots r_n)$ is a vector of mutually distinct cost registers,*
- *δ is the transitions set whose elements are transition tuple $(q, a, q', \vec{v})$ where q, q' are states in Q, a is a letter in alphabet Σ and $\vec{v}$ is vector of natural number. The transition $(q, a, q', \vec{v})$ is written as $q \xrightarrow[\vec{v}]{a} q'$ for readability.*
- *θ is the linear integer arithmetic constraint of registers.*

A run *of $\mathcal{A}$ on string $a_1 \cdots a_m$ is a transition sequence $q_0 \xrightarrow[\vec{v_1}]{a_1} q_1 \cdots q_{m-1} \xrightarrow[\vec{v_m}]{a_m} q_m$ such that $q_0 \in I$ and $\mathcal{V}(r_i) = \sum_{j=1}^{m} v_j(i)$ is the* value *of register r_i after the run. Note that $\mathcal{V}(r_i)$ is initiated to 0 at state q_0. The run is* accepting *if $q_m \in F$ and $\theta[r_i/\mathcal{V}(r_i)]$ is satisfiable. Note that $\theta[r_i/\mathcal{V}(r_i)]$ means a substitution of all occurrences of r_i in θ to its current value. A string word w is accepted by $\mathcal{A}$ if there is an accepting run of $\mathcal{A}$ on w. The language of $\mathcal{A}$ is the set of string words accepted by $\mathcal{A}$, denoted by $\mathcal{L}(\mathcal{A})$.*

Example 3 (The CEFA $\mathcal{A}_{(ab)\{1,100\}}$). Table 1 is the CEFA of the regular expression $(ab)\{1, 100\}$. The transition set are $\{q_0 \xrightarrow[1]{a} q_1, q_1 \xrightarrow[0]{b} q_2, q_2 \xrightarrow[0]{\epsilon} q_0\}$. The register is r_1. And the linear arithmetic is $1 \leq r_1 \leq 100$. Intuitively, each accepted run of $\mathcal{A}_{(ab)\{1,100\}}$ repeatedly run word ab from one time to 100 times because $1 \leq r_1 \leq 100$ restrict the value of r_1 to the range $[1, 100]$. More precisely, the repeat time of the transition $q_0 \xrightarrow[1]{a} q_1$ is restricted to $[1, 100]$.

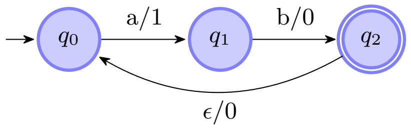

Fig. 1: $\mathcal{A}_{(ab)\{1,100\}}$ with one register r_1 and $\theta = 1 \leq r_1 \leq 100$

The example 3 gives a general picture of constructing a CEFA for a regular expression: firstly, we build an NFA from the regex $\mathcal{R}$ (e.g, To construct CEFA in example 3, an NFA is construted with transitions $\{q_0 \xrightarrow{a} q_1 \xrightarrow{b} q_2 \xrightarrow{\epsilon} q_0\}$ initial state q_0, and accepting state q_2). Secondly, we extend NFA to CEFA by adding registers to R and vector to each transition (e.g., the register r_1 is added to store repetition times and add vector $\vec{1}$ is set to transition $q_0 \xrightarrow{a} q_1$ to update the repetition time). Finally, we set the linear arithmetic θ carefully to restrict the accepted word to be included in $\mathcal{L}(\mathcal{R})$ (e.g., $\theta = 1 \leq r_1 \geq 100$ is set to restrict the repetition times to $[1, 100]$). Besides the repetition operation, other operations are non-trivial on CEFA. We individually discuss them in the following subsections.

4.2 Non-nested Repetition

Given a CEFA $\mathcal{A} = (Q, \Sigma, \delta, I, F, \emptyset, true)$, the repetition operation $\{m, n\}$ on it is defined as a CEFA $\mathcal{A}_{rep} = (Q, \Sigma, \delta', I, F, r_1, \theta')$ where:

- r_1 is the new register,
- δ' is composed by transitions $q \xrightarrow[\vec{v}\cdot 0]{a} q'$ for all transitions $q \xrightarrow[\vec{v}]{a} q' \in \delta$, and transitions $q_m \xrightarrow[0_n\cdot 1]{\epsilon} q_0$ for all $q_m \in F$ and $q_0 \in I$,
- θ' is the linear arithmetic $m \leq r_1 \leq n$ if all initial states are not accepting. Otherwise, θ' is the linear arithmetic $r_1 \leq n$.

In the construction of $\mathcal{A}_{rep}$, a new register r_1 is added to store the repetition times. The transition $q_m \xrightarrow[0_n\cdot 1]{\epsilon} q_0$ is added to update the repetition times. The linear arithmetic θ' is set to restrict the repetition times to $[m, n]$. Because arbitrary repetition times of empty string is still empty string, the value of r_i need not be greater than m when some initial states are accepting.
Note that the given CEFA $\mathcal{A}$ does not contain register. So the repetition operation on $\mathcal{A}$ is non-nested.

4.3 Concatenation

Given two CEFA $\mathcal{A}_1 = (Q_1, \Sigma, \delta_1, I_1, F_1, R_1, \theta_1)$ and $\mathcal{A}_2 = (Q_2, \Sigma, \delta_2, I_2, F_2, R_2, \theta_2)$ with $R_1 \cap R_2 = \emptyset$ and $|R_1| = m, |R_2| = n$, the concatenation is defined as $\mathcal{A}_{con} = (Q_1 \cup Q_2, \Sigma, \delta', I_1, F_2, R_1 \cdot R_2, \theta_1 \wedge \theta_2)$ where δ' is composed by

- $q_1 \xrightarrow[\vec{v_1} \cdot 0_n]{a} q_1'$ for each transition $q_1 \xrightarrow[\vec{v_1}]{a} q_1' \in \delta_1$,
- $q_2 \xrightarrow[0_m \cdot \vec{v_2}]{a} q_2'$ for each transition $q_2 \xrightarrow[\vec{v_2}]{a} q_2' \in \delta_2$,
- and $q_1 \xrightarrow[0_{m+n}]{\epsilon} q_2$ for each $q_1 \in F_1$ and $q_2 \in I_2$.

The concatenation of two CEFAs is similar to the concatenation of two NFAs, except that the registers and its updates are also concatenated.

4.4 Intersection

Given two CEFA $\mathcal{A}_1 = (Q_1, \Sigma, \delta_1, I_1, F_1, R_1, \theta_1)$ and $\mathcal{A}_2 = (Q_2, \Sigma, \delta_2, I_2, F_2, R_2, \theta_2)$ with $R_1 \cap R_2 = \emptyset$, the intersection is defined as $\mathcal{A}_{inter} = (Q_1 \times Q_2, \Sigma, \delta', I_1 \times I_2, F_1 \times F_2, R_1 \cdot R_2, \theta_1 \wedge \theta_2)$ where δ' is composed by

- transitions $(q_1, q_2) \xrightarrow[\vec{v_1} \cdot \vec{v_2}]{a} (q_1', q_2')$ for transition $q_1 \xrightarrow[\vec{v_1}]{a} q_1'$ and transition $q_2 \xrightarrow[\vec{v_2}]{a} q_2'$ exist individually in δ_1 and δ_2.

The intersection of two CEFAs is similar to the intersection of NFA, except that the registers updates and linear arithmetic are also intersected.

4.5 Union

Given two CEFA $\mathcal{A}_1 = (Q_1, \Sigma, \delta_1, I_1, F_1, R_1, \theta_1)$ and $\mathcal{A}_2 = (Q_2, \Sigma, \delta_2, I_2, F_2, R_2, \theta_2)$ with $R_1 \cap R_2 = \emptyset$ and $|R_1| = m, |R_2| = n$, the union is defined as $\mathcal{A}_{union} = (Q_1 \cup Q_2 \cup \{q_0\}, \Sigma, \delta', \{q_0\}, F_1 \cup F_2, R_1 \cdot R_2 \cdot (r_1, r_2), \theta')$ where δ' is composed by

- transitions $q_0 \xrightarrow[0_{m+n} \cdot (1,0)]{\epsilon} q_1$ for all $q_1 \in I_1$,
- transitions $q_0 \xrightarrow[0_{m+n} \cdot (0,1)]{\epsilon} q_2$ for all $q_2 \in I_2$,
- transitions $q_1 \xrightarrow[\vec{v_1} 0_{n+2}]{a} q_1'$ for all $q_1 \xrightarrow[\vec{v_1}]{a} q_1' \in \delta_1$,
- transitions $q_2 \xrightarrow[0_m \vec{v_2} 0_2]{a} q_2'$ for all $q_2 \xrightarrow[\vec{v_2}]{a} q_2' \in \delta_2$.

r_1 and r_2 are new registers. q_0 is a new state where $q_0 \notin Q_1$ and $q_0 \notin Q_2$. Assume that $\theta = (r_1 > 0 \wedge \theta_1) \vee (r_2 > 0 \wedge \theta_2)$, θ' is constructed differently in four cases:

- both I_1 and I_2 do not contain accepting states: $\theta' = \theta$;
- I_1 contains accepting states while I_2 does not: $\theta' = \theta \vee (r_1 == 0 \wedge r_2 == 0 \wedge \theta_1)$;
- I_2 contains accepting states while I_1 does not: $\theta' = \theta \vee (r_1 == 0 \wedge r_2 == 0 \wedge \theta_2)$;
- both I_1 and I_2 contain accepting states: $\theta' = \theta \vee (r_1 == 0 \wedge r_2 == 0 \wedge (\theta_1 \vee \theta_2))$;

To simulate the semantics of the union, we add transitions from q_0 to initial states of $\mathcal{A}_1$ and $\mathcal{A}_2$ to randomly choose one automaton. $r_1 > 0$ indicates we choose $\mathcal{A}_1$ to run while $r_2 > 0$ indicates we choose $\mathcal{A}_2$ to run. We should make sure the accepting condition is consistent. Therefore, the accepting condition $(r_1 > 0 \wedge \theta_1) \vee (r_2 > 0 \wedge \theta_2)$ is generated. Furthermore, the accepting run may stop at the new initial state q_0 with $\mathcal{V}(r_1) = 0, \mathcal{V}(r_2) = 0$. It can only happen when some initial states of two given CEFAs are accepting. If some initial states of $\mathcal{A}_1$ are accepting, the accepting condition $r_1 = 0 \wedge r_2 = 0 \wedge \theta_1$ is generated. Similarly, if some initial states of $\mathcal{A}_2$ are accepting, the accepting condition $r_1 = 0 \wedge r_2 = 0 \wedge \theta_2$ is generated. Finally, the disjunction of all generated accepting conditions θ' is obtained to express the non-nondeterministic.

4.6 Complement

Given a CEFA $\mathcal{A} = (Q, \Sigma, I, F, \delta, \emptyset, true)$, the complement $\mathcal{A}^c$ is defined as the CEFA $\mathcal{A}^c = (Q, \Sigma, I, F^c, \delta, R, \theta^c)$ where $F^c = Q \setminus F$ and $\theta^c = \neg\theta$.

4.7 Closure

4.8 Complement of Repetition and Nested Repetition

As we mentioned, CEFA

4.9 From Regex to CEFA

Basis: The basis has three parts with empty registers and valid linear arithmetic, shown in Fig. 2. In part 2a, we see how to handle the expression ϵ. The language of the automaton is easily seen to be $\{\epsilon\}$ since the only path from the start state to an accepting state is labeled ϵ. Part 2b gives the CEFA for a regular expression **a**. The language of this CEFA consists of the single string a, which is also $\mathcal{L}(\mathbf{a})$. Finally, part 2c shows the construction for $\emptyset$. There are no paths from the initial state to the accepting state.

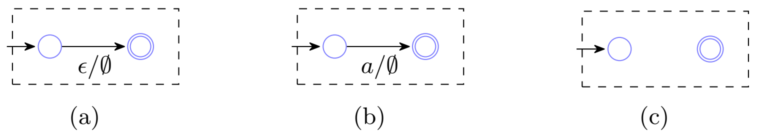

Fig. 2: The basis of the construction of a CEFA from extended regex

Induction: Given the CEFA of the subexpression, we construct the CEFA in five cases:

1. The expression is $\mathcal{R}\{m, n\}$ for some smaller expression $\mathcal{R}$. We can repeat the CEFA of $\mathcal{R}$ with lower bound m and upper bound n as shown in section 4.2.
2. The expression is $\mathcal{R}_1 \cdot \mathcal{R}_2$ for some smaller expression $\mathcal{R}_1$ and $\mathcal{R}_2$. We can concatenate the CEFA of $\mathcal{R}_1$ and the CEFA of $\mathcal{R}_2$ as shown in section 4.3.

3. The expression is $\mathcal{R}_1 \times \mathcal{R}_2$ for some smaller expression $\mathcal{R}_1$ and $\mathcal{R}_2$. We can intersect the CEFA of $\mathcal{R}_1$ and the CEFA of $\mathcal{R}_2$ as shown in section 4.4.
4. The expression is $\mathcal{R}_1 + \mathcal{R}_2$ for some smaller expression $\mathcal{R}_1$ and $\mathcal{R}_2$. We can union the CEFA of $\mathcal{R}_1$ and the CEFA of $\mathcal{R}_2$ as shown in section 4.5.
5. The expression is $\mathcal{R}*$ for some smaller expression $\mathcal{R}$. We can construct the closure Star of the CEFA of $\mathcal{R}$ as shown in section **??**.
6. The expression is $\mathcal{R}^C$ for some smaller expression $\mathcal{R}$. We can construct the complement of the CEFA of $\mathcal{R}$ as shown in section 4.8

Special Case: When the expression $\mathcal{R}$ contains nested repetition, we firstly write $\mathcal{R}$ to a new regex $\mathcal{R}'$ without nested repetition. Then we build the CEFA of the new expression $\mathcal{R}'$ by basis and induction steps.

5 Solve String Constraints with Large Repetition Bound

Following the decision procedure defined in the paper [Chen et al.(2020)], we propose an efficient algorithm to solve string constraints with help of CEFA. Firstly, we replace all length operations $i = \texttt{len}(x)$ to pre-images which are represented by CEFAs, and intersect these CEFAs to $lenAut_x$. Then for each term x, we generate CEFAs of all regular expressions $\mathcal{R}$ in all terms $x \in \mathcal{R}$, and intersect these CEFAs with $lenAut_x$ to get the final CEFA. Finally, we check whether each final CEFA is empty under the linear integer arithmetic P. If one of the final CEFAs is empty, the string constraints are unsatisfiable. Otherwise, the string constraints are satisfiable. The high-level algorithm is shown in the Algorithm 1.
The emptiness checking problem of CEFAs under linear integer arithmetic P is theoretically pspace-complete[Chen et al.(2020)]. To solve it efficiently for practical example, we add some heuristics. The details of the heuristics are shown in Algorithm 2.

5.1 High-level algorithm

The pseudocode presented in Algorithm 1 outlines the framework of our solving process. The implemented details are eliminated for clarity. From line 1 to line 10, we construct the CEFAs of all length operations occurring in the string constraints. The CEFA $lenAut_x$ is the automaton used to store length information on x. $lenAut_x$ is initiated to accept all strings at line 2. For each length operation $i = \texttt{len}(x)$, we construct the pre-image of it at line 3, and intersect the pre-image to $lenAut_x$ at line 4 to store length information. Note that the final $lenAut_x$ is the intersection of all pre-images so that it can store all length information on x.
In the following steps, we call a CEFA to be *atomic* if we will not operate it anymore. The set *atomAuts* is the set of all atomic CEFAs for all terms x occurring in the string constraints, and is initially empty at line 6. From line 7 to line 14, each atomic automaton in the set is constructed by intersecting the length

Algorithm 1 High-level algorithm

Input: Conjunction φ of the form $x \in \mathcal{R}$, and conjunction of linear integer arithmetic P over string lengths
Output: *sat* or *unsat*

```
 1: for all length operation $i = \texttt{len}(x)$ do
 2:     $lenAut_x \leftarrow \mathcal{A}_{allstring}$
 3:     Let $\mathcal{A}_i$ be the pre-image of length operation with length $i$
 4:     $lenAut_x \leftarrow lenAut_x \times \mathcal{A}_i$
 5: end for
 6: $atomAuts \leftarrow \emptyset$
 7: for all string variables $x$ occurring in $\varphi$ do
 8:     Let $S$ be the set of all regexes $\mathcal{R}$ in all terms $x \in \mathcal{R}$
 9:     CEFA $\mathcal{A}_x \leftarrow$ intersection of $lenAut_x$ and all CEFAs corresponding to regexes in $S$
10:     if $nfa(\mathcal{A}_x)$ is empty then
11:         return unsat
12:     end if
13:     $atomAuts \leftarrow atomAuts \cup \{\mathcal{A}_x\}$
14: end for
15: if $\texttt{isEmpty}(atomAuts, P)$ then
16:     return unsat
17: else
18:     return sat
19: end if
```

automaton $lenAut_x$ to all CEFAs corresponding to the regex expressions with regard to string term x. If the NFA form of the atomic automaton is empty (i.e., the NFA accepts no word), we return *unsat* directly because the NFA form is an over-approximation of CEFA.
After obtaining the atomic CEFAs of all string terms, we check whether the atomic CEFAs are empty under the linear integer arithmetic P at line 15. If the atomic CEFAs are empty under P, we return *unsat*. Otherwise, we return *sat*. The emptiness checking at line 15 is complex, and we discuss it detailly in the following subsection.

5.2 Emptyness Checking

As mentioned, the emptiness checking problem of CEFA under linear integer arithmetic P is theoretically pspace-complete [Chen et al.(2020)]. In our previous research, we rewrote CEFA to an infinite system and used a model-checking tool *NuSMV* to solve it. However, the previous method was not effictive enough. So we put forward a new method to solve it.
As shown in Algorithm 2, We simplify the CEFAs at line 1 because the input CEFAs may have many transitions and registers. The purpose of the simplification is removing duplicated transitions and registers in the CEFA. After simplifying the input CEFAs, we try to find a solution by under-approximation at line 3. If we find a solution, we return $false$ at line 5 because the CEFAs under linear integer arithmetic P are not empty. Otherwise, we try to find an unsat core by over-approximation at line 8. If we discover an unsat core, we return *true* at line 10 because we know that the CEFAs under P are empty. Suppose that both under- and over-approximation do not make sense, we compute the Parikh images of the simplified CEFAs by algorithm in the paper [Verma et al.(2005)], and check if the Parikh images are satisfiable in conjunction with P at line 12. The satisfiability implies non-emptiness directly.
Note that the under-approximation program is executed many times until $MaxBound$ is reached. The details of simplification is shown in Algorithm 5. The details of under- and over-approximation are shown in Algorithm 3 and Algorithm 4 respectively. The details of Parikh image computation are omitted for clarity.

5.3 Under-Approximation

To solve the satisfiable instances of the emptiness checking problem, under-approximation procedure is come up. The main idea is exploring the CEFA deep-firstly. The string length bound the deep, and linear integer arithmetic is generated from low to high string length. This section only discusses the situation in where the string length bound is fixed. The details are shown in the Algorithm 3. Firstly, the linear integer arithmetic φ_{under} is assigned to be false at line 1. Then for each CEFA, we enumerate all the accepted paths with lengths less than

Algorithm 2 `isEmpty`$(auts, P)$

Input: CEFAs *auts* and linear integer arithmetic P
Output: *true* or *false*

```
1: $simpliAuts \leftarrow$ simplify$(auts)$
2: for $bound \leftarrow 1, MaxBound$ do
3:     $\varphi_{under} \leftarrow$ underApprox$(simpliAuts, bound)$
4:     if $\varphi_{under} \wedge P$ is sat then
5:         return $false$
6:     end if
7: end for
8: $\varphi_{over} \leftarrow$ overApprox$(simpliAuts)$
9: if $\varphi_{over} \wedge P$ is unsat then
10:     return $true$
11: end if
12: $\varphi \leftarrow$ parikhImage$(simpliAuts)$
13: if $\varphi \wedge P$ is unsat then
14:     return $true$
15: else
16:     return $false$
17: end if
```

the bound and translate the paths to linear integer arithmetic. The linear integer arithmetic is in conjunction with φ_{under} at line 4.

5.4 Over-Approximation

Sometimes when the bound reaches the maximum, the linear integer arithmetic φ_{under} is still unsatisfiable. We must use over-approximation to solve the emptiness checking problem in this case. The main idea of over-approximation is to split each CEFA into sub-CEFAs, translate each sub-CEFA respectively, and finally in conjunction with the LIA of all sub-CEFAs. The details are shown in the Algorithm 4. Firstly, each CEFA is split into n sub-CEFAs, where n is the number of registers in the CEFA. The sub-CEFA has only one register. After simplifying each sub-CEFA by Algorithm 5, the vector of each transition must be $\vec{1}$. So the register value of each sub-CEFA equals the length of the accepted path. We apply the efficient construction of semilinear representations[Sawa(2010)] to get the linear integer arithmetic φ_{over}.

5.5 Simplification of CEFA

The main purpose of simplification is to remove duplicated transitions and registers. In the emptiness checking Algorithm 2, the vectors on the transitions are meaningful while the letters on the transitions are not. So we see the alphabet of the CEFA as unary (i.e., the alphabet is $\{a\}$). Then the vector in the CEFA is seen as the letter in the NFA, and the vector $\vec{0}_n$ is seen as ϵ. Based

Algorithm 3 `underApprox`$(auts, bound)$

Input: The CEFAs *auts* and string length *bound*
Output: The linear integer arithmetic φ_{under} representing under-approximation of the register values of *auts*

```
1: $\varphi_{under} \leftarrow false$
2: for $aut \in auts$ do
3:     Suppose $aut = (Q, \Sigma, \delta, q_0, F, (r_1, \cdots, r_m), \theta)$
4:     $S_0 \leftarrow \{(q_0, \vec{0}_m)\}$
5:     for all path $q_0 q_1 \cdots q_n$ whose length is $bound$ and $\vec{R}$ is the sum of the vectors
   on the path do
6:         if $q_n \in F$ then
7:             $\varphi_{under} \leftarrow \varphi_{under} \bigwedge_{i \in [1,m]} r_i = \vec{R}(i)$
8:         end if
9:     end for
10: end for
11: return $\varphi_{under}$
```

Algorithm 4 `overApprox`$(auts)$

Input: CEFAs *auts*
Output: Linear integer arithmetic φ_{over}

```
1: $\varphi_{over} \leftarrow true$
2: for $aut \in auts$ do
3:     Let $subAut$ be the sub-CEFA of $aut$ with only one register
4:     Let $subAut \leftarrow$ simplify($subAut$)
5:     Let $\varphi_{subAut}$ be the linear integer arithmetic generated by efficient construction
   of semilinear representations
6:     $\varphi_{over} \leftarrow \varphi_{over} \wedge \varphi_{subAut}$
7: end for
8: return $\varphi_{over}$
```

on the homomorphism between NFA and CEFA, we determinate and minimize [Hopcroft and Ullman(1979)] the CEFA by vector firstly. Then we merge the registers having the same updating at all transitions. The main simplification framework is shown in Algorithm 5.

Algorithm 5 `simplify`$(auts)$

Input: CEFAs *auts*
Output: Simplified CEFAs *simpliAuts*

```
1: $simpliAuts \leftarrow \emptyset$
2: for $aut \in auts$ do
3:     $aut \leftarrow$ determinizeByVec$(aut)$
4:     $aut \leftarrow$ minimizeByVec$(aut)$
5:     $aut \leftarrow$ mergeRegisters$(aut)$
6:     $simpliAuts \leftarrow simpliAuts \cup \{aut\}$
7: end for
8: return $simpliAuts$
```

6 Implementation and Experiment

TODO

6.1 Overall Evaluation of Experiment

tools: cvc5, z3str3, z3str3re, z3-trau, ostrich benchmark: total benchmark

6.2 Experimental Details

tools: cvc5, z3str3, z3str3re, z3-trau, ostrich benchmmark:

- Small Count: the sum of upper bounds on the repetition operator is less than 50
  - z3str3re-bench
  - regexlib
  - stackoverflow
  - ReDos-bench
- Large Count: the sum of upper bounds on the repetition operator is greater than 50
  - z3str3re-bench
  - regexlib
  - stackoverflow
  - ReDos-bench

6.3 Analysis of All Algorithms

tools:

- ostrich-baseline: the origin ostrich syntactically unwind all repetition operators.
- ostrich-new: the most efficient ostrich which using CEFA and all heuristics
- ostrich-without-under: without under-approxiamtion
- ostrich-without-over: without over-approximation
- ostrich-without-simplification: without simplification of CEFA
- ostrich-without-rapidly-finding-string: without context-aware finding string

7 Conclusion and Future Work

In this paper, we aim to solve the string constraints with large repetition time efficiently. A new automaton model CEFA is proposed to reduce the search space. Many basic operations on CEFA are non-trivial and we should formal them carefully. Moreover, we extend the algorithm in the paper [Chen et al.(2020)] with heuristics such as under- and over-approximation. The implementation of the algorithm is done on string solver ostrich. The extensive empirical comparison against z3 over a large and diverse benchmark shows the power of our model and algorithm. In the future, we plan to explore the way to solve nested repetition and use CEFA to solve more string operations.

References

[Chen et al.(2020)] Taolue Chen, Matthew Hague, Jinlong He, Denghang Hu, Anthony Widjaja Lin, Philipp Rümmer, and Zhilin Wu. 2020. A Decision Procedure for Path Feasibility of String Manipulating Programs with Integer Data Type. In *Automated Technology for Verification and Analysis*, Dang Van Hung and Oleg Sokolsky (Eds.). Springer International Publishing, Cham, 325–342.

[Davis(2019)] James C. Davis. 2019. Rethinking Regex Engines to Address ReDoS *(ESEC/FSE 2019)*. Association for Computing Machinery, New York, NY, USA, 1256–1258. `https://doi.org/10.1145/3338906.3342509`

[Ganesh and Berzish(2016)] Vijay Ganesh and Murphy Berzish. 2016. Undecidability of a Theory of Strings, Linear Arithmetic over Length, and String-Number Conversion. (05 2016).

[Hopcroft and Ullman(1979)] John E. Hopcroft and Jeff D. Ullman. 1979. *Introduction to Automata Theory, Languages, and Computation.* Addison-Wesley Publishing Company.

[Le and He(2018)] Quang Loc Le and Mengda He. 2018. A Decision Procedure for String Logic with Quadratic Equations, Regular Expressions and Length Constraints. In *Programming Languages and Systems*, Sukyoung Ryu (Ed.). Springer International Publishing, Cham, 350–372.

[Sawa(2010)] Zdeněk Sawa. 2010. Efficient Construction of Semilinear Representations of Languages Accepted by Unary NFA. In *Reachability Problems*, Antonín Kučera and Igor Potapov (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 176–182.

[Stanford et al.(2021)] Caleb Stanford, Margus Veanes, and Nikolaj Bjørner. 2021. Symbolic Boolean Derivatives for Efficiently Solving Extended Regular Expression Constraints. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation* (Virtual, Canada) *(PLDI 2021)*. Association for Computing Machinery, New York, NY, USA, 620–635. https://doi.org/10.1145/3453483.3454066

[Verma et al.(2005)] Kumar Neeraj Verma, Helmut Seidl, and Thomas Schwentick. 2005. On the Complexity of Equational Horn Clauses. In *Proceedings of the 20th International Conference on Automated Deduction* (Tallinn, Estonia) *(CADE' 20)*. Springer-Verlag, Berlin, Heidelberg, 337–352. https://doi.org/10.1007/11532231_25