Algorithms for Checking Intersection Non-emptiness of Regular Expressions

Weihao Su[1,2], Rongchen Li[1,2], Chengyao Peng[1,2], and Haiming Chen[1]

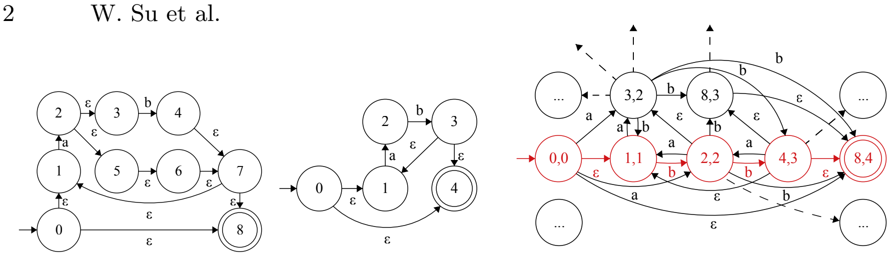
[1] State Key Laboratory of Computer Science, Institute of Software, Chinese Academy of Sciences, 100190, Beijing, China
[2] University of Chinese Academy of Sciences, 101400, Beijing, China
{suwh,lirc,pengcy,chm}@ios.ac.cn

Abstract. The intersection non-emptiness problem of regular languages is one of the most classical and fundamental decision problems in formal language theory, which plays an important role in many areas. Because of its wide applications, the efficiency of the algorithms becomes particularly crucial. In practice, it is quite common that automata have large numbers of states, therefore the explicit construction of automata may incur significant costs in terms of both time and space, significantly impacting the performance of the related programs. To overcome this challenge, in this paper, we present four efficient algorithms for checking the intersection of regular expressions without the need for automata construction. Our algorithms employ lazy evaluation strategies to simulate intersection non-emptiness checking on automata to avoid constructing automata. They also use automata with fewer states to reduce the state complexity. We conducted experiments and compared our results with six state-of-the-art SMT solvers. The results show significant advantages of our algorithms over existing methods, achieving significant speedups over the current SMT solvers.

Keywords: Regular Expressions · Intersection Non-emptiness Problem · Online Algorithms.

1 Introduction

The intersection non-emptiness problem of regular languages is one of the most classical and fundamental decision problems in formal language theory. The problem has a wide range of applications and has been extensively studied by researchers. Given a set of m regular languages $L(E_1)$, ..., $L(E_m)$ defined over the same alphabet Σ, the problem is to determine whether $\bigcap_{i=1}^{m} L(E_i) \neq \varnothing$, where $\varnothing$ denotes the empty set. In other words, it decides whether there exists a common word in all languages. The algorithms for solving this problem play an important role in various fields such as SMT (Satisfiability Modulo Theories) solvers, model checking tools, artificial intelligence [46] , data privacy [36], and ReDoS (Regular expression Denial of Service) vulnerability detection [42]. For instance, SMT solvers like Z3 [26], CVC4 [43], and OSTRICH [25] use these



(a) Thompson automaton of E_1 (b) Thompson automaton of E_2 (c) A part of the intersection automaton for illustrating an accepting path from the starting state (0,0) to the final state (8,4) in the intersection automaton, which indicates $\bigcap_{i=1}^{2} L(E_i) \neq \varnothing$

* Thompson's construction [45] is one of the most frequently used methods for constructing automata from regular expressions, for example, [25, 26, 48].

Fig. 1. The classical process for determining $\bigcap_{i=1}^{2} L(E_i) \neq \varnothing$ for $E_1 = (a(b+\epsilon))^*$ and $E_2 = (ab)^*$.

algorithms to solve regular membership queries. Similarly, model checking tools like Mona [37] use these algorithms to verify hardware and software systems.

The classical process of determining the intersection non-emptiness of two regular expressions is illustrated in Figure 1, which can be solved in polynomial time [39]. The key steps include (1) compiling the expressions into finite automata, (2) constructing the intersection automaton from the compiled automata by the cross product algorithm [50], and (3) performing non-emptiness testing on the intersection automaton. For this example, the path colored in red in Figure 1(c) shows an accepting path in the intersection automaton, indicates that the intersection exists between E_1 and E_2, so the process returns true. The above process can be naturally extended to determine the intersection non-emptiness of multiple regular expressions, still in polynomial time [49, 54]. When taking the number k of the regular expressions and the maximum number of states n in the automata from those expressions as parameters, this problem is fixed-parameter tractable [27, 53]. For an unbounded number of inputs, this problem is PSPACE-complete [41].

Computing the intersection non-emptiness of regular expressions is a common task in various applications, making efficient algorithms particularly important. The aforementioned classical process is based on automata. However, the above algorithm may incur significant costs in terms of both time and space during automata construction[3], since in practice it is quite common for automata to have large numbers of states. In particular, the states of the intersection automaton grow rapidly: for two automata with Q_1 and Q_2 states respectively, the intersection automaton will have $Q_1 * Q_2$ states. Indeed in our experiments (see section 5), the automata-based algorithms Z3-Trau [1] and OSTRICH [25] can very rarely solve the problem within 20 seconds. Our key observation is that explicit construction of automata for determining the intersection non-emptiness

[3] Consisting of both compiling regular expressions into finite automata and constructing the intersection automaton.

of regular expressions can significantly impact the performance of related programs, even acting as a bottleneck. For instance, we conducted an experiment with ReDoSHunter [42], which uses the dk.brics.automaton library [48] for automata construction, on regular expressions from the corpus library [18], and the result reveals that intersection checking based on automaton construction occupies 59.26% of the runtime and causes the maximum memory usage of the entire program. In addition, the program often crashes due to the timeout of the intersection checking. This shows the necessity of not constructing the whole automaton explicitly when optimizing program performance.

To address this issue, we use lazy evaluation strategies to simulate intersection non-emptiness checking on automata to avoid constructing automata. It is well known that looking for a reason to fail or finding a reachable path is easy to spot. Also, the number of states in an automaton has a direct impact on the size of the solution space of the intersection non-emptiness problem [38, 56]. Thus using a finite automaton with fewer states can help reduce the state complexity of the intersection automata. So our algorithms simulate finite automata with fewer states [8] to achieve this goal.

Specifically, in this paper, we present four intersection non-emptiness detection algorithms. The first type utilizes the positions of characters in the expressions. In detail, the first algorithm is based on the position sets [31], which can be used to compute transition tables of the position automata [57], resulting in the `Pos_intersect` algorithm. Building on `Pos_intersect`, we further utilize the equivalence relation $\equiv_f$ (detailed in Chapter 3) to obtain the `Follow_intersect` algorithm, which simulates intersection non-emptiness search on the follow automata [40], effectively reducing the size of the solution space. The second type of algorithms employ derivatives: Based on c-continuation, we propose the `CCon_intersect` algorithm for simulating intersection non-emptiness search on the $M_{ccon}(E)/_{\equiv_c}$ automaton. The time and space complexity of c-continuation is lower than partial derivative [16], but the solution space of the algorithm is larger than that of the equation automaton [2]. To reduce the solution space, we develop a second algorithm, `Equa_intersect`, based on the equivalence relation $\equiv_c$ (see Chapter 4), which effectively simulates the intersection non-emptiness search on the equation automata.

To validate the efficiency and effectiveness of our algorithms, we compared them to six state-of-the-art SMT solvers. Our experiments demonstrate that our four algorithms have significant advantages over existing methods in solving the intersection non-emptiness problem of regular expressions. In particular, our approach outperforms the competition in terms of speed and accuracy, highlighting the effectiveness of our methodology. In addition, we have observed the potential for the extensibility of these algorithms (as detailed in Chapter 7): such as the output of the entire intersection automaton, the addition of extended features in real-world regular expressions, and the use of local search or real-time heuristic search strategies to optimize these algorithms.

2 Preliminaries

In this section, we briefly recall the necessary definitions in regular languages and automata theory, for further details, we suggest referring to [55].

2.1 Regular Expressions

Let Σ be an alphabet and Σ^* the set of all possible words over Σ; $|\Sigma|$ denotes the size of Σ, ε denotes the empty word. A *language* over Σ is a subset of Σ^*. A *regular expression* over Σ is either $\varnothing$, ε or $a \in \Sigma$, or is the union $E_1 + E_2$, the concatenation E_1E_2, or the Kleene star E_1^* for regular expressions E_1 and E_2. $\varnothing$ denotes the empty set. The regular language defined by E is denoted by $L(E)$. The size of a regular expression E is denoted by $|E|$ and represents the number of symbols and operators (excluding parentheses) in E when written in postfix. The alphabetic width of E, denoted by $\|E\|$, is the number of symbols occurring in E. Σ_E denotes the symbols occurring in E, i.e., the minimal alphabet of E.

We define $nullable(E) = true$ if $\varepsilon \in L(E)$, and $false$ otherwise. We mark symbols in E with numerical subscripts to obtain a linearized regular expression $E^\#$ over $\Sigma^\#$ such that all marked symbols in $E^\#$ occur no more than once. For example, let $E = (a + b)(a^* + ba^* + b^*)$, a linearized regular expression is $E^\# = (a_1 + b_2)(a_3^* + b_4a_5^* + b_6^*)$. The reverse of marking, i.e., dropping off the subscripts, is denoted by $E^\natural$, then we have $(E^\#)^\natural = E$. We extend the notations for sets of symbols, words and automata in an obvious way.

2.2 Deterministic Regular Expressions

Deterministic (one-unambiguous) regular expressions were first proposed and formalized by Brüggemann-Klein and Wood [12].

Definition 1. *A regular expression E is deterministic if and only if for all words $uxv, uyw \in L(E^\#)$ s.t. $|x| = |y| = 1$, if $x \neq y$ then $x^\natural \neq y^\natural$. A regular language is deterministic if it is denoted by some deterministic expression.*

For example, $b^*a(b^*a)^*$ is deterministic, while its semantically equivalent regular expression $(a + b)^*a$ is not deterministic.

2.3 Position Automaton and Star Normal Form

A deterministic finite automaton (DFA) is a quintuple $M = (Q, \Sigma, \delta, s, F)$, where Q is the finite set of states, Σ is the alphabet, $\delta \subseteq Q \times \Sigma \to Q$ is the state transition function, $s \in Q$ is the starting (or initial) state, and $F \subseteq Q$ is the set of final states. A non-deterministic automaton (NFA) is a quintuple $M = (Q, \Sigma, \delta, s, F)$ where Q, Σ, s, and F are defined in exactly the same way as a DFA, except that $\delta \subseteq Q \times \Sigma \to 2^Q$ is the state transition function where 2^Q denotes the power set of Q. $L(M)$ denotes the language accepted by the automaton M.

Let $\equiv \subseteq Q \times Q$ be an equivalence relation. For $q \in Q$, $[q]_\equiv$ denotes the equivalence class of q w.r.t. $\equiv$ and, for $S \subseteq Q$, $S/_\equiv$ denotes the quotient set $S/_\equiv = \{[q]_\equiv \mid q \in S\}$. We say that $\equiv$ is right invariant w.r.t. M if and only if:

1. $\equiv \subseteq (Q - F)^2 \cup F^2$,
2. $\forall p, q \in Q$, $a \in A$, if $p \equiv q$, then $\delta(p,a)/_\equiv = \delta(q,a)/_\equiv$.

If $\equiv$ is right invariant, then the quotient automaton $M/_\equiv$ is constructed as $M/_\equiv = (Q/_\equiv, \Sigma, \delta_\equiv, [s_0]_\equiv, F/_\equiv)$, where $\delta_\equiv = \{([p]_\equiv, a, [q]_\equiv) \mid (p,a,q) \in \delta\}$; Notice that $L(M/_\equiv) = L(M)$.

Independently introduced by Glushkov [31] and McNaughton and Yamada [45], the position automaton is considered as the standard automaton of a regular expression [51]. Until now, the deterministic position automaton still serves as the major matching model of `awk` [32], `sed` [34] and `grep` [33] with multiple extensions. Here we follow [11] and define "the Glushkov NFA" as the position automaton.

For an expression E over Σ and a symbol $a \in \Sigma$, we define the following sets:

$$first(E) = \{b \mid bw \in L(E), b \in \Sigma, w \in \Sigma^*\} \tag{1}$$

$$last(E) = \{b \mid wb \in L(E), b \in \Sigma, w \in \Sigma^*\} \tag{2}$$

$$follow(E,a) = \{b \mid uabv \in L(E), u, v \in \Sigma^*, b \in \Sigma\} \tag{3}$$

Definition 2. *The position automaton $M_{pos}(E)$ of a regular expression E is defined by a 5-tuple $(Q_{pos}, \Sigma, \delta_{pos}, s_{pos}, F_{pos})$ where*
$Q_{pos} = \Sigma^{\#} \cup \{s_{pos}\}$,
$s_{pos} = 0$,
$\delta_{pos}(s_{pos}, a) = \{x \mid x \in first(E^{\#}), x^\natural = a\}$, *for* $a \in \Sigma$,
$\delta_{pos}(x, a) = \{y \mid y \in follow(E^{\#}, x), y^\natural = a\}$, *for* $x \in \Sigma^{\#}$ *and* $a \in \Sigma$,

$$F_{pos} = \begin{cases} last(E^{\#}) \cup \{s_{pos}\}, & if\ nullable(E) = true, \\ last(E^{\#}). & otherwise \end{cases}$$

We assume that 0 is a symbol that is not in $\Sigma_{E^{\#}}$. From the definition we have $follow(E^{\#}, 0) = first(E^{\#})$. We also define $last_0(E^{\#})$, which is $last(E^{\#})$ if $nullable(E) = false$, and $last(E^{\#}) \cup \{0\}$ otherwise. We denote $pos_0(E) = \Sigma^{\#} \cup \{0\}$. The construction of the position automaton defined above is improved to quadratic time in the size of the expression [11, 17, 57]. Among these works, Brüggemann-Klein [11] gave a linear-time algorithm to transform an arbitrary regular expression E into star normal form.

Definition 3. *A regular expression E is in star normal form $E^\bullet$, if for each starred subexpression H^* of E, the following SNF-conditions hold:*
$follow(H^{\#}, a) \cap first(H^{\#}) = \varnothing$, *for* $a \in last(H^{\#})$,
$nullable(H) = false$.

For example, given $E = (a^*b^*)^*$, E can be transformed into the semantically equivalent star normal form $E^\bullet = (a+b)^*$.

Furthermore, from [11] we have:

Proposition 1. *For a deterministic regular expression E, the deterministic position automaton $M_{pos}(E)$ can be computed from E in linear time.*

2.4 Derivatives of Regular Expressions

J. Brzozowski [10] introduced the notion of derivatives of regular expressions.

Definition 4. *Given a regular expression E and a symbol a, the total derivative w.r.t. a, denoted by $a^{-1}(E)$, is defined inductively as follows:*

$$
\begin{aligned}
a^{-1}(\varnothing) &= a^{-1}(\varepsilon) = \varnothing && (1)\\
a^{-1}(b) &= \begin{cases} \varepsilon, & if\ b = a \\ \varnothing, & otherwise \end{cases} && (2)\\
a^{-1}(F+G) &= a^{-1}(F) + a^{-1}(G) && (3)\\
a^{-1}(FG) &= \begin{cases} a^{-1}(F)G, & if\ nullable(F) = false \\ a^{-1}(F)G + a^{-1}(G), & otherwise \end{cases} && (4)\\
a^{-1}(F^*) &= a^{-1}(F)F^* && (5)
\end{aligned}
$$

The number of Brzozowski's (total) derivatives may not be finite. When considering similarity of associativity, commutativity and idempotence of $+$, the number of Brzozowski's derivatives can still be exponential in the worst case upon arbitrary regular expressions, e.g. on $(a+b)^*a(a+b)^k$, where k is a positive integer. But for deterministic regular languages, the number of Brzozowski's derivatives has a linear upper bound on the size of the regular expressions [19].

Antimirov [2] generalized Brzozowski's results and introduced the partial derivatives to construct an NFA - the partial derivative automaton, or the equation automaton. In [15] it is proved that the automaton construction based on the notion of Mirkin's prebase [47] is equivalent to Antimirov's construction.

Definition 5. *Given a regular expression E and a symbol a, the partial derivative w.r.t. a, denoted by $\partial_a(E)$, is defined inductively as follows:*

$$
\begin{aligned}
\partial_a(\varnothing) &= \partial_a(\varepsilon) = \varnothing && (1)\\
\partial_a(b) &= \begin{cases} \varepsilon, & if\ b = a \\ \varnothing, & otherwise \end{cases} && (2)\\
\partial_a(F+G) &= \partial_a(F) \cup \partial_a(G) && (3)\\
\partial_a(FG) &= \begin{cases} \partial_a(F)G, & if\ nullable(F) = false \\ \partial_a(F)G \cup \partial_a(G), & otherwise \end{cases} && (4)\\
\partial_a(F^*) &= \partial_a(F)F^* && (5)
\end{aligned}
$$

The partial derivative w.r.t. a word is computed by: $\partial_\varepsilon(E) = \{E\}$, $\partial_{wa}(E) = \bigcup_{p \in \partial_w(E)} \partial_a(p)$. Denote $PD(E)$ as $\bigcup_{w \in \Sigma^*} \partial_w(E)$ and we have the definition of the equation automaton as follows:

Definition 6. *The equation automaton $M_e(E)$ of a regular expression E is defined by a 5-tuple $(Q_e, \Sigma, \delta_e, s_e, F_e)$, where*
$Q_e = PD(E)$,
$\delta_e(q, a) = \partial_a(q)$, *for* $q \in Q_e$ *and* $a \in \Sigma$,
$s_e = E$,
$F_e = \{q \in PD(E) \mid \varepsilon \in L(q)\}$.

It was proved in [2] that the size of $PD(E)$ is less than or equal to $\|E\|+1$. It is known that the equation automaton $M_e(E)$ is a quotient of the position automaton $M_{pos}(E)$ [16, 24, 40].

3 Position-Based Algorithms

In [9], the authors proposed novel position and c-continuation constructions of regular expressions extended with intersection operators. In [23], the authors proposed an M_{E_1}-directed search algorithm on position automata that avoids the explicit construction of complement automata. These inspired us to develop intersection non-emptiness checking algorithms based on the *first*, *follow*, and *last* sets defined in Chapter 2 without explicit construction of automata. We first propose an algorithm based on simulating the construction of position automata, and further optimize it with follow automata.

The algorithm `Pos_intersect` is listed in Algorithm 1, which simulates intersection non-emptiness checking on the position automata. The algorithm first turns regular expressions into star normal form and linearizes them. Then in line 5 and line 8, it checks the nullability and the intersection non-emptiness of their *last* sets as heuristics to bail out the algorithm earlier, i.e. if both expressions are nullable, then ε is in the intersection, and if all of the last characters between both languages are disjointed, then from Definition 2 we know the final state sets of the position automata of the expressions are disjoint, so the intersection is empty. Next in line 11, it starts the recursive searching procedure with the $first$ sets and the tuple of both the starting states as initial inputs. In the recursive function (from line 12 to line 23), the set v_1 and v_2 are position sets which implies the transition of the position automata and Q is introduced to store the position tuples visited, i.e. states of the intersection of the position automata. The function first performs an intersection non-emptiness check on the symbols of input sets, thus deciding the next positions to search, effectively cumulating transitions that have the same symbols from both the current states in the position automata. For every tuple of positions (p_1, p_2) from v_1 and v_2 that represent the same symbol (i.e., $p_1^\natural = p_2^\natural$), it checks if both of them are in the *last* sets in each expression respectively. If so it returns true. Otherwise, if the position tuple has been reached before, it returns false, otherwise, it memoizes the tuple into the set Q and continue the search with the $follow$ sets of positions, simulating a transition from this state to a next state in both position automata.

Theorem 1. *Given two regular expressions E_1 and E_2,* `Pos_intersect` *returns true if and only if $L(E_1) \cap L(E_2) \neq \varnothing$.*

Proof. $\Longleftarrow$: Assume $L(E_1) \cap L(E_2) \neq \varnothing$, i.e. there exists a word w with $w \in L(E_1)$ and $w \in L(E_2)$. Then we have $w_1 \in L(E_1^{\#})$, $w_2 \in L(E_2^{\#})$ where $w_1^\natural = w_2^\natural = w$. We can select a pair of marked prefixes u_1 and u_2 from w_1 and w_2 respectively, which are denoted as $u_1 = xp_1$ and $u_2 = yp_2$ respectively, such that the following conditions are satisfied: $x^\natural = y^\natural$ and $p_1^\natural = p_2^\natural$, where x, y are words and

Algorithm 1: `Pos_intersect`

Input: two regular expressions E_1 and E_2.
Output: *true* if $L(E_1) \cap L(E_2) \neq \varnothing$ or *false* otherwise.

`Pos_intersect` :: (Expression E_1, Expression E_2) $\to$ Boolean
begin
  $E_1^{\bullet} \leftarrow snf(E_1)$; $E_2^{\bullet} \leftarrow snf(E_2)$;
  $E_1^{\#} \leftarrow linearize(E_1^{\bullet})$; $E_2^{\#} \leftarrow linearize(E_2^{\bullet})$;
  if $nullable(E_1^{\#}) = true \wedge nullable(E_2^{\#}) = true$ **then**
    return *true*;
  else
    if $last(E_1^{\natural}) \cap last(E_2^{\natural}) = \varnothing$ **then**
      return *false*;
    else
      return `Pos_recur`$(first(E_1^{\#}), first(E_2^{\#}), \{(s_{pos_1}, s_{pos_2})\})$;

`Pos_recur` :: (Set v_1, Set v_2, Set Q) $\to$ Boolean
begin
  if $v_1^{\natural} \cap v_2^{\natural} = \varnothing$ **then**
    return *false*;
  forall $p_1 \in v_1 \wedge p_2 \in v_2 \wedge p_1^{\natural} = p_2^{\natural}$ **do**
    if $p_1 \in last(E_1^{\#}) \wedge p_2 \in last(E_2^{\#})$ **then**
      return *true*;
    if $(p_1, p_2) \notin Q$ **then**
      if
       `Pos_recur`$(follow(E_1^{\#}, p_1), follow(E_2^{\#}, p_2), Q \cup \{(p_1, p_2)\}) = true$
       then
        return *true*;
    else
      return *false*;

p_1, p_2 are positions, and $|x| = |y| \geq 0$. Since $p_1^{\natural} = p_2^{\natural}$, `Pos_intersect` first checks if p_1, p_2 are in $last(E_1^{\#})$ and $last(E_2^{\#})$ correspondingly, if so `Pos_intersect` returns $true$ in advance before the end of w. If not, `Pos_intersect` checks whether (p_1, p_2) was reached before to ensure termination and continue to check the positions in their $follow$ sets. Trivially for every last character a of a word u, there must exist positions that $p_1^{'\natural} = p_2^{'\natural} = a$, $p_1^{'} \in last(E_1^{\#})$ and $p_2^{'} \in last(E_2^{\#})$. And `Pos_intersect` returns $true$.

$\Longrightarrow$: If `Pos_intersect` returns true, it is whether $\varepsilon \in L(E_1) \wedge \varepsilon \in L(E_2)$, i.e. $\varepsilon \in L(E_1) \cap L(E_2)$, or there is a tuple (p_1, p_2) that $p_1 \in last(E_1^{\#}) \wedge p_2 \in last(E_2^{\#})$, where from the tuple set Q we have two marked words $w_1 = u_1 p_1$ and $w_2 = u_2 p_2$, where $w_1^{\natural} = w_2^{\natural} = w$. And assume $u_1 = u_1^{'} p_1^{'}$, $u_2 = u_2^{'} p_2^{'}$ and $|u_1^{'}| = |u_2^{'}| = i$ for $0 \leq i \leq |w|$, we have $u_1^{\natural} = u_2^{\natural}$, $p_1 \in follow(E_1^{\#}, p_1^{'})$ and $p_2 \in follow(E_2^{\#}, p_2^{'})$. By induction on i, $w_1^\natural \in L(E_1)$ and $w_2^\natural \in L(E_2)$. Overall we have $w \in L(E_1)$ and $w \in L(E_2)$ and thus $L(E_1) \cap L(E_2) \neq \varnothing$. □

Next, we show how the follow automaton [40] can be integrated into Algorithm 1 as an optimization. Ilie and Yu proposed a new quadratic algorithm to construct ε-free NFAs from regular expressions named follow automaton, denoted as $M_f(E)$. The authors proposed a novel constructive method based on removal of ε transitions from an small ε-automaton similiar to Thompson's automaton [45]. We refer to [40] for details of the construction.

Definition 7. *The follow automaton $M_f(E)$ of a regular expression E is defined by a 5-tuple $(Q_f, \Sigma, \delta_f, s_f, F_f)$ where*
$Q_f = \{(follow(E^\#, x), x \in last_0(E^\#))\}$, *for* $x \in pos_0(E)$,
$\delta_f(x, a) = \{(follow(E^\#, y), y \in last_0(E^\#)) \mid y \in follow(E^\#, x), y^\natural = a\}$, *for* $x \in pos_0(E)$ *and* $a \in \Sigma$,
$s_f = (first(E^\#), 0 \in last_0(E^\#))$,
$F_f = \{(follow(E^\#, x), true) \mid x \in last_0(E^\#)\}$.

We define the right invariant equivalence relation $\equiv_f \subseteq Q_{pos}^2$ [40]:

Definition 8. *Given two states a_1 and a_2 in Q_{pos}, we have:*

$$a_1 \equiv_f a_2 \Longleftrightarrow \begin{cases} a_1 \in last_0(E^\#) \Leftrightarrow a_2 \in last_0(E^\#) \\ follow(E^\#, a_1) = follow(E^\#, a_2) \end{cases} \tag{1}$$

Proposition 2. *(See [40].)* $M_f(E) \backsimeq M_{pos}(E)/_{\equiv_f}$.

Since the follow automaton is a quotient of its position automaton, then simulating the follow automaton instead of the position automaton in Algorithm 1 can reduce the size of the solution space. This motivated us to develop a new algorithm `Follow_intersect`. Specifically, we adapt Algorithm 1 to using the equivalence relation $\equiv_f$, which is mainly achieved by substituting the function in line 11 to $\texttt{Pos\_recur}(first(E_1^\#), first(E_2^\#), \{(first(E_1^\#), 0 \in last_0(E_1)), (first(E_2^\#), 0 \in last_0(E_2)))\})$, the condition in line 19 to $((follow(E_1^\#, p_1), p_1 \in last_0(E_1)), (follow(E_2^\#, p_2), p_2 \in last_0(E_2))) \notin Q$ and the condition in line 20 to $\texttt{Pos\_recur}(follow(E_1^\#, p_1), follow(E_2^\#, p_2), Q \cup \{((follow(E_1^\#, p_1), p_1 \in last_0(E_1)), (follow(E_2^\#, p_2), p_2 \in last_0(E_2)))\}) = true$. After the substitution, Algorithm 1 starts recursive search with $first$ sets as in Definition 7 and select the positions whose symbols are identical, combine them pairwise and check if they are in both $last_0$ sets, i.e. indicating the final states of follow automata, if not, check if their $follow$ sets and the Boolean value of whether they are in the $last_0$ sets are reached before, if not, memoize them into Q and continue to check their $follow$ sets, simulating a transition from the current state to the next state in both follow automata. Overall the search space of the checking algorithm is reduced to simulating the search procedure on the states of the product automaton of two follow automata.

Theorem 2. `Follow_intersect` *preserves the right-invariant property of* $\equiv_f$.

Proof. We shall prove the theorem by showing `Follow_intersect` implies two conditions of right invariant property of $\equiv_f$.

Firstly, we will show this condition won't lead to an equivalence between a final state and a non-final state. Note that if `Follow_intersect` reaches the point of determining whether the condition $(follow(E_1^{\#}, p_1), p_1 \in last_0(E_1)), (follow(E_2^{\#}, p_2), p_2 \in last_0(E_2)) \notin Q$ is valid, it means that `Follow_intersect` not returns true in the previous procedure. Accordingly, in the previous searching path, the search on tuple $(follow(E_1^{\#}, p_1), p_1 \in last_0(E_1)), (follow(E_2^{\#}, p_2), p_2 \in last_0(E_2))$ will not lead to a successful termination, thus the condition only result in merging between non-final states w.r.t. simulated intersection automaton.

Secondly, when `Follow_intersect` reaches the step of judging whether the condition $((follow(E_1^{\#}, p_1), p_1 \in last_0(E_1)), (follow(E_2^{\#}, p_2), p_2 \in last_0(E_2))) \notin Q$ is valid, the algorithm actually make a judgement on if the next searching space is already searched. We denote by S the state tuple $(follow(E_1^{\#}, p_1), follow(E_2^{\#}, p_2))$, and suppose that $S = T = ((follow(E_1^{\#}, p_1), p_1 \in last_0(E_1)), (follow(E_2^{\#}, p_2), p_2 \in last_0(E_2))) \notin Q$, next we shall show the follow equivalence between S and T. Since $S \in Q$ now, `Follow_intersect` just skip the searching on $(follow(E_1^{\#}, p_1), follow(E_2^{\#}, p_2))$, which actually simulates the merging of states in follow automata. There we have $(follow(E_1^{\#}, p_1) = follow(E_2^{\#}, t_1))$ and $(follow(E_1^{\#}, p_2) = follow(E_2^{\#}, t_2))$, so we have $p_1 \equiv_f t_1$ w.r.t $M_{pos}(E_1)$ and $p_2 \equiv_f t_2$ w.r.t $M_{pos}(E_2)$. $\square$

Theorem 3. *Given two regular expressions E_1 and E_2,* `Follow_intersect` *returns true if and only if $L(E_1) \cap L(E_2) \neq \varnothing$.*

Proof. $\Longleftarrow$: Assume $L(E_1) \cap L(E_2) \neq \varnothing$, i.e. there exists a word u with $u \in L(E_1)$ and $u \in L(E_2)$. Then we have $u_1 \in L(E_1^{\#})$, $u_2 \in L(E_2^{\#})$ where $u_1^{\natural} = u_2^{\natural} = u$. We can select a pair of marked prefixes p_1 and p_2 from u_1 and u_2 respectively, which are denoted as $p_1 = xa$ and $p_2 = yb$ respectively, such that the following conditions are satisfied: $x^{\natural} = y^{\natural}$ and $a^{\natural} = b^{\natural}$, where x, y are words and a, b are characters, and $|x| = |y| \geq 0$. Since $a^{\natural} = b^{\natural}$, `Follow_intersect`$(E_1, E_2)$ will move on to check the next character. Note that the condition $(follow(E_1^{\#}, p_1), follow(E_2^{\#}, p_2)) \notin Q$ won't affect eventually to the end of u_1 and u_2 by induction as the fore-checked positions has identical characters. Trivially for every last unmarked character a of a word u, the corresponding positions must belong to $last_0(E_1^{\#})$ and $last_0(E_2^{\#})$. Thus `Follow_intersect`(E_1, E_2) must return true.

$\Longrightarrow$: If `Follow_intersect`(E_1, E_2) returns true, from the tuple set Q we have two marked words w_1 and w_2, and note that the characters of the same position in w_1 and w_2 are identical because we continue the loop under the condition of $p_1^{\natural} = p_2^{\natural} = u$. And here we shall show we get a marked word by increasingly appending characters from their follow sets under condition $((follow(E_1^{\#}, p_1), p_1 \in last_0(E_1)), (follow(E_2^{\#}, p_2), p_2 \in last_0(E_2))) \notin Q$. First, we can easily obtain a position automaton from a linearized regular expression, the elements in the follow set of a character a can directly be interpreted as a transition from the former state. From Theorem 2, under the condition $((follow(E_1^{\#}, p_1), p_1 \in last_0(E_1)), (follow(E_2^{\#}, p_2), p_2 \in last_0(E_2))) \notin Q$ we actually merge those states in position automata with the same follow set, here we get the corresponding follow automata. Accordingly, the procedure in `Follow_intersect`(E_1, E_2) effectively simulates a search on the transition graph of the production automaton from two follow automata, the word we get is definitely accepted by both the follow automata. Thus the word w_1 and w_2 belong to $L(E_1^{\#})$ and $L(E_2^{\#})$. Above all, we have $w \in L(E_1)$ and $w \in L(E_2)$ and thus $L(E_1) \cap L(E_2) \neq \varnothing$. □

Complexity. The computation of the intersection of two position sets can be done in time $O(\|E_1\|+\|E_2\|)$ with the help of an auxiliary hash table in the size of $O(\|E_1\|)$. The identification of condition $(p_1, p_2) \notin Q$ takes time $O(\|E_1\|\|E_2\|)$, since the set of position tuples Q has a size of $O(\|E_1\|\|E_2\|)$. This condition is checked $\|E_1\|\|E_2\|$ times the worst time. Then `Pos_intersect` have time complexity of $O(\|E_1\|^2\|E_2\|^2)$ and space complexity of $O(\|E_1\|\|E_2\|)$. For the case of `Follow_intersect`, the intersection of two position sets is computed the same as above. The identification of condition $((follow(E_1^{\#}, p_1), p_1 \in last_0(E_1)), (follow(E_2^{\#}, p_2), p_2 \in last_0(E_2))) \notin Q$ takes time $O((\|E_1\| + \|E_2\|)\|E_1\|\|E_2\|)$ since the set of follow set tuple Q has a size of $O((\|E_1\|+\|E_2\|)\|E_1\|\|E_2\|)$. This condition is checked $\|E_1\|\|E_2\|$ times. Then we have the overall time complexity of $O((\|E_1\|+\|E_2\|)\|E_1\|^2\|E_2\|^2)$ and space complexity of $O((\|E_1\| + \|E_2\|)\|E_1\|\|E_2\|)$. For a deterministic regular expression E, the size of position sets has a upper bound of $|\Sigma_E|$, thus the time complexity of `Pos_intersect` is $O((|\Sigma_{E_1} \cap \Sigma_{E_2}|)(\|E_1\| + \|E_2\|)\|E_1\|\|E_2\|)$ and the space complexity is $O(\|E_1\|\|E_2\|)$. For `Follow_intersect`, the time complexity is $O((|\Sigma_{E_1}| + |\Sigma_{E_2}|)|\Sigma_{E_1} \cap \Sigma_{E_2}|(\|E_1\| + \|E_2\|)\|E_1\|\|E_2\|)$ and space complexity is $O((|\Sigma_{E_1}| + |\Sigma_{E_2}|)\|E_1\|\|E_2\|)$.

4 C-Continuation-based Algorithms

The notion of continuation is developed by Berry and Sethi [5], by Champarnaud and Ziadi [16], by Ilie and Yu [40], and by Chen and Yu [24]. In [20, 25], the author gave a novel construction of derivatives on deterministic regular expressions and proved its linear cardinality. However when arbitrary regular expressions is considered, exponential search space of the algorithm is inevitable. To avoid the exponential blow-up, we exploit the notion of c-continuation proposed in [16] to check the intersection non-emptiness of two regular expressions.

Definition 9. *Given a regular expression E and a symbol a, the c-derivative w.r.t. a, denoted by $d_a(E)$, is defined inductively as follows [16]:*

$$d_a(\varnothing) = d_a(\varepsilon) = \varnothing, \tag{1}$$

$$d_a(b) = \begin{cases} \varepsilon, & if\ b = a \\ \varnothing, & otherwise \end{cases} \tag{2}$$

$$d_a(F+G) = \begin{cases} d_a(F), & if\ d_a(F) \neq \varnothing \\ d_a(G), & otherwise \end{cases} \tag{3}$$

$$d_a(FG) = \begin{cases} d_a(F)G, & if\ d_a(F) \neq \varnothing \\ d_a(G), & if\ d_a(F) \neq \varnothing \wedge nullable(F) = true \\ \varnothing, & otherwise \end{cases} \tag{4}$$

$$d_a(F^*) = d_a(F)F^* \tag{5}$$

The c-derivative w.r.t. a word is computed by: $d_\varepsilon(E) = E$, $d_{wa} = d_a(d_w(E))$, for $a \in \Sigma$ and $w \in \Sigma^*$.

Lemma 1. *(see [16].) For a linearized regular expression $E^\#$, for every symbol a and every word u, the c-derivative $d_{ua}(E^\#)$ w.r.t the word ua is either $\varnothing$ or unique.*

Definition 10. *Given a linearized regular expression $E^\#$ and a symbol $a \in \Sigma_E$, the c-continuation of $E^\#$ w.r.t. a symbol, denoted as $c_a(E^\#)$, is defined inductively as follows:*

$$c_a(a) = \varepsilon \tag{1}$$

$$c_a(F+G) = \begin{cases} c_a(F), & if\ c_a(F) \neq \varnothing, \\ c_a(G), & otherwise \end{cases} \tag{2}$$

$$c_a(FG) = \begin{cases} c_a(F)G, & if\ c_a(F) \neq \varnothing, \\ c_a(G), & otherwise \end{cases} \tag{3}$$

$$c_a(F^*) = c_a(F)F^* \tag{4}$$

Also we let $c_0(E^\#) = d_\varepsilon(E^\#) = E^\#$.

Proposition 3. *If E is a regular expression in star normal form, then $c_a(E^\#)$ is in star normal form, for each a in $\Sigma^\#$.*

The proof is a straightforward induction on the structure of $c_a(E^\#)$.

Here, given two states a_1 and a_2 in Q_{pos}, we can define the following equivalence relations $=_c, \equiv_c \subseteq Q_{pos}^2$:

$$a_1 =_c a_2 \iff c_{a_1}(E^\#) = c_{a_2}(E^\#) \tag{1}$$

$$a_1 \equiv_c a_2 \iff c_{a_1}(E^\#)^\natural = c_{a_2}(E^\#)^\natural \tag{2}$$

It has been shown both $=_c$ and $\equiv_c$ are right-invariant w.r.t. $M_{pos}(E)$ [16]. From the definition of c-continuation, the c-continuation automaton $M_{ccon}(E)$ is obtained.

Lemma 2. *(see [16].) For a regular expression E, $M_{ccon}(E)$ and $M_{pos}(E)$ are identical.*

Definition 11. *Automaton $M_{ccon}(E)/_{=_c}$ of a regular expression E is defined by a 5-tuple $(Q_{ccon}, \Sigma, \delta_{ccon}, s_{ccon}, F_{ccon})$ where*
$Q_{ccon} = \{c_x(E^\#) \mid x \in \Sigma^\# \cup \{0\}\}$,
$\delta_{ccon}(c_x(E^\#), a) = \{d_y(c_x(E^\#)) \mid y^\natural = a\}$, *for* $x \in \Sigma^\# \cup \{0\}$ *and* $a \in \Sigma$,
$s_{ccon} = c_0(E^\#)$,
$F_{ccon} = \{c_x(E^\#) \mid nullable(c_x(E^\#)) = true\}$.

Proposition 4. *For a deterministic regular expression E, $M_{ccon}(E)/_{=_c}$ is deterministic.*

Proof. As a direct consequence of Lemma 2, for a deterministic expression E, since $M_{pos}(E)$ is deterministic, and $M_{ccon}(E)/_{=_c}$ is a quotient of $M_{pos}(E)$, it is implied that $M_{ccon}(E)/_{=_c}$ is deterministic.

We have the following relations between c-continuation automaton and position automaton.

Lemma 3. *For any $a \in \Sigma_E$, the following two relations holds: $first(c_a(E^{\#})) = follow(E^{\#}, a)$ and $a \in last_0(E^{\#}) \iff nullable(c_a(E^{\#})) = true$.*

Algorithm 2: `CCon_intersect`

Input: two regular expressions E_1 and E_2.
Output: *true* if $L(E_1) \cap L(E_2) \neq \varnothing$ or *false* otherwise.

`CCon_intersect` :: (Expression E_1, Expression E_2) $\to$ Boolean
begin
    $E_1^{\bullet} \leftarrow snf(E_1)$; $E_2^{\bullet} \leftarrow snf(E_2)$;
    $E_1^{\#} \leftarrow linearize(E_1^{\bullet})$; $E_2^{\#} \leftarrow linearize(E_2^{\bullet})$;
    if $nullable(E_1^{\#}) = true \wedge nullable(E_2^{\#}) = true$ **then**
        return *true*;
    else
        if $last(E_1^{\natural}) \cap last(E_2^{\natural}) = \varnothing$ **then**
            return *false*;
        else
            return `CCon_recur`$(E_1^{\#}, E_2^{\#}, \{(E_1^{\#}, E_2^{\#})\})$;

`CCon_recur` :: (Expression $r_1^{\#}$, Expression $r_2^{\#}$, Set C) $\to$ Boolean
begin
    if $first(r_1^{\natural}) \cap first(r_2^{\natural}) = \varnothing$ **then**
        return *false*;
    forall $a_1 \in first(E_1^{\#}) \wedge a_2 \in first(E_2^{\#}) \wedge a_1^{\natural} = a_2^{\natural}$ **do**
        $c_1 \leftarrow c_{a_1}(E_1^{\#})$; $c_2 \leftarrow c_{a_2}(E_2^{\#})$;
        if $nullable(c_1) = true \wedge nullable(c_2) = true$ **then**
            return *true*;
        if $(c_1, c_2) \notin C$ **then**
            if `CCon_recur`$(c_1, c_2, C \cup \{(c_1, c_2)\}) = true$ **then**
                return *true*;
        else
            return *false*;

While the preprocessing procedure of Algorithm 2 before the recursive search follows the same technique as Algorithm 1, the recursive function starts the search with the linearized expressions and a tuple of them as inputs. The linearized expressions correspond to elements in $CC(E_1)$ and $CC(E_2)$, then the tuples are states of $M_{ccon}(E_1)/_{=_c} \cap M_{ccon}(E_2)/_{=_c}$, which starts from $(E_1^{\#}, E_2^{\#})$ as in Definition 11, $E^{\#}$ is a start state of $M_{ccon}(E)/_{=_c}$. The recursive search performs an intersection non-emptiness check on the symbols of $first$ mappings of the input expressions. For a_1 and a_2 in $first$ sets of each expressions that have the same symbol, i.e. $a_1^\natural = a_2^\natural$, we calculate c-continuations of the input expressions w.r.t the positions and check the nullability of their c-continuations as from Definition 11, a nullable c-continuation corresponds to a final state. If the c-continuations are not nullable simultaneously, we first check if this c-continuation tuple is reached before, we terminate this branch, if not, we memoize the tuple into a set C as a reached state in the intersection automaton and continue the search with these c-continuations simulating a transition of the identical symbol from positions used for calculating those c-continuations in both $M_{ccon}(E)/_{=_c}$.

Theorem 4. *Given two regular expressions E_1 and E_2,* `CCon_intersect` *returns true if and only if $L(E_1) \cap L(E_2) \neq \varnothing$.*

Proof. $\Longleftarrow$: Assume $L(E_1) \cap L(E_2) \neq \varnothing$, i.e. there exists a word w with $w \in L(E_1)$ and $w \in L(E_2)$. Then we have $w_1 \in L(E_1^{\#})$, $w_2 \in L(E_2^{\#})$ where $w_1^\natural = w_2^\natural = w$. We can select a pair of marked prefixes u_1 and u_2 from w_1 and w_2 respectively, which are denoted as $u_1 = xp_1$ and $u_2 = yp_2$ respectively, such that the following conditions are satisfied: $x^\natural = y^\natural$ and $p_1^\natural = p_2^\natural$, where x, y are words and p_1, p_2 are positions, and $|x| = |y| \geq 0$. Since $p_1^\natural = p_2^\natural$, denote $d_x(E_1^{\#})$ and $d_y(E_2^{\#})$ as c_1 and c_2, `CCon_intersect` first checks if $d_{p_1}(c_1)$, $d_{p_2}(c_2)$ are nullable simultaneously, if so `CCon_intersect` returns $true$ in advance before the end of w. If not, `CCon_intersect` checks whether $(d_{p_1}(c_1), d_{p_2}(c_2))$ was reached before to ensure termination and continue to check the positions in their $first$ sets. For every last character a of a word w, there must exist positions that $p_1^\natural = p_2^\natural$, $d_{w_1}(E_1^{\#})$ and $d_{w_2}(E_2^{\#})$. And `CCon_intersect` returns $true$.

$\Longrightarrow$: If `CCon_intersect` returns true, it is whether $\varepsilon \in L(E_1) \wedge \varepsilon \in L(E_2)$, i.e. $\varepsilon \in L(E_1) \cap L(E_2)$, or there are $|w_1| = |w_2|$, where $nullable(d_{w_1}(E_1^{\#})) = true \wedge nullable(d_{w_2}(E_1^{\#})) = true$. And assume $u_1 = u_1^{'} p_1^{'}$, $u_2 = u_2^{'} p_2^{'}$ and $|u_1^{'}| = |u_2^{'}| = i$ for $0 \leq i \leq |w|$, we have $u_1^\natural = u_2^\natural$, $p_1 \in first(d_{u_2^{'}}(E_1^{\#}))$ and $p_2 \in first(d_{u_2^{'}}(E_2^{\#}))$. By induction on i, when $w_1^\natural \in L(E_1)$ and $w_2^\natural \in L(E_2)$ hold. Overall we have $w \in L(E_1)$ and $w \in L(E_2)$ and thus $L(E_1) \cap L(E_2) \neq \varnothing$. □

Proposition 5. *(See [16].)* $M_e(E) \simeq M_{ccon}(E)/_{\equiv_c}$.

From the fact above we know equation automaton is a quotient of its ccontinuation automaton. We can improve Algorithm 2 by substituting the code in line 11 with **return** `CCon_recur`$(E_1^{\#}, E_2^{\#}, \{(E_1^\natural, E_2^\natural)\})$, line 20 with condition $(c_1^\natural, c_2^\natural) \notin C$ and line 21 with `CCon_recur`$(c_1, c_2, C \cup \{(c_1^\natural, c_2^\natural)\}) = true$. And algorithm `Equa_intersect` is obtained. By dropping the labels in c-continuations, each expressions in the tuples stored in C corresponds to states in $M_{ccon}(E)/_{\equiv_c}$, the search space of the checking algorithm is reduced to simulating the search procedure on the states of the product automaton of two $M_{ccon}(E)/_{\equiv_c}$.

Theorem 5. `Equa_intersect` *preserves the right-invariant property of $\equiv_c$.*

Proof. This proof is based on the discussion on the properties of right-invariant equivalence relation(see section 2.3), we shall show $(c_1^\natural, c_2^\natural) \notin C$ preserves those two properties.

Firstly, we shall show this condition won't lead to a merge between a final state and a non-final state. Note that the condition for terminating the search is $nullable(c_1) = true$, $nullable(c_2) = true$, which indicates one of the states in a non-final intersection state pair must not be final. Therefore the input tuple must be a non-final state from the discussion above. As a consequence, any merge of states in a tuple C must be the merging of two non-final states. In summary, the condition indicates the first property of right-invariant equivalence relation, i.e. $a_1 \notin last(E^{\#})$ and $a_2 \notin last(E^{\#})$.

Secondly, we shall show the condition of $(c_1^\natural, c_2^\natural) \notin C$ indicates the merging between $(c_1^\natural, c_2^\natural)$ and an equivalent element $(c_1^{'\natural}, c_2^{'\natural})$ in C. For $(c_1^\natural, c_2^\natural)$, suppose our algorithm have marked $(c_1^{'\natural}, c_2^{'\natural})$ in precedent procedures, when $(c_1^\natural, c_2^\natural)$ is reached, the branch of search terminates at $(c_1^{'\natural}, c_2^{'\natural})$ thus indicates a loop on the intersection automaton (or simply fails). So the further search from $(c_1^\natural, c_2^\natural)$ will be pruned to ensure the termination and all elements are unique in C. If the search reaches the condition $(c_1^\natural, c_2^\natural) \notin C$, the algorithm expand the set C. And if $(c_1^\natural, c_2^\natural)$ and $(c_1^{'\natural}, c_2^{'\natural})$ are equvialent, suppose that $x_1^\natural = x_2^\natural = x_1^{'\natural} = x_2^{'\natural}$, for any position $p_1 \in E_1^{\#}$, whose character $p_1^\natural \in first(d_{x_1}^\natural(c_1)) \cap first(d_{x_2}^\natural(c_2))$, there must at least exist an equivalent position $p_2 \in E_1^{\#}$, whose character $p_2^\natural \in first(d_{x_1'}^\natural(c_1)) \cap first(d_{x_2'}^\natural(c_2))$. From the hypothesis we have $c_1^\natural = c_1^{'\natural}$, thus we have $p_1 \equiv_c p_2$. Evidently the same property holds for E_2. □

Theorem 6. *Given two regular expressions E_1 and E_2,* `Equa_intersect` *returns true if and only if $L(E_1) \cap L(E_2) \neq \varnothing$.*

Proof. $\Longleftarrow$: From the assumption we have that there exists a word u that $u \in L(E_1)$ and $u \in L(E_2)$. From the marked regular expression we have $u_1 \in L(E_1^{\#})$, $u_2 \in L(E_2^{\#})$ where $u_1^\natural = u_2^\natural = u$. Suppose some marked prefixes p_1 and p_2 of u_1 and u_2 are denoted as $p_1 = xa$ and $p_2 = yb$ for E_1 and E_2, while $x^\natural = y^\natural$ and $a^\natural = b^\natural$, where x, y are words and a, b are characters, and $|x| = |y| \geq 0$. Since $a^\natural = b^\natural$, `Equa_intersect`(E_1, E_2) will move on to compute the next c-continuation of the marked character in u, and to the end of u by induction as the fore-checked positions has identical characters. Trivially for every last unmarked character of a word u, the corresponding positions must belong to $last(E_1^{\#})$ and $last(E_2^{\#})$, so two corresponding c-continuation must be nullable, thus `Equa_intersect`(E_1, E_2) must return true.

$\Longrightarrow$: If `Equa_intersect`(E_1, E_2) returns true, from the tuple set C we have a unmarked word w. However, the corresponding possible marked words may not be unique, according to the arbitrariness of the possible words, we can just take two of which w_1 and w_2 to discuss. Note that the characters of the same position in w_1 and w_2 are identical because we continue the loop under the condition of $a_1^\natural = a_2^\natural$. And here we will show we can get a marked word by appending character inductively from its c-continuation. First, we obtain a c-continuation automaton by computing the c-continuation of marked characters, the cardinality of a c-continuation set is lower than $\|E\|+1$. The identification of condition $(c_1^{\natural}, c_2^{\natural}) \notin C$ effectively simulates $\equiv_c$ relation in an online manner, eventually the procedure in `Equa_intersect`(E_1, E_2) in fact simulates a search on the transition graph of the production automaton of $M_{ccon}(E_1)/_{\equiv_c}$ and $M_{ccon}(E_2)/_{\equiv_c}$, which are isomorphic to the corresponding equation automata by Lemma 2, the word we obtain is definitely accepted by both the equation automata. Thus the word w_1 and w_2 belong to $L(E_1^{\#})$ and $L(E_2^{\#})$. Above all, we have $w \in L(E_1)$ and $w \in L(E_2)$ and thus $L(E_1) \cap L(E_2) \neq \varnothing$. $\square$

Complexity. The computation of the $first$ sets of c-continuations takes an $O(\|E_1\|^2|E_1| + \|E_2\|^2|E_2|)$ time and $O(\|E_1\|^2 + \|E_2\|^2)$ space complexity. The calculation of the intersection of two position sets can be done in linear time and space with the same technique of Algorithm 1. The computation of c-continuations of both expressions costs $O(\|E_1\||E_1|^2 + \|E_2\||E_2|^2)$ time and space [16]. Computation of *nullable* on the resulted c-continuations costs $O(|E_1|^2 + |E_2|^2)$ time and $O(\|E_1\| + \|E_2\|)$ space as in the worst case, the size of a c-continuation of E is $|E|^2$ [35]. The identification of condition $(c_1, c_2) \notin C$ takes $O(\|E_1\||E_1|^2 + \|E_2\||E_2|^2)$ time, and $O(\|E_1\||E_1|^2 + \|E_2\||E_2|^2)$ space is required for representation of the list C of c-continuation tuples. Finally we have the time complexity of `CCon_intersect`: $O((\|E_1\||E_1|^2 + \|E_2\||E_2|^2) \times (\|E_1\|^2|E_1| + \|E_2\|^2|E_2|))$ and the space complexity: $O(|E_1|^2\|E_1\| + |E_2|^2\|E_2\|)$. In the case of deterministic regular expressions, the time complexity of `CCon_intersect` is $O((\|E_1\||E_1|^2 + |\|E_2\|||E_2|^2) \times (|\Sigma_{E_1}|\|E_1\||E_1| + |\Sigma_{E_2}|\|E_2\||E_2|))$ and space complexity is $O(|E_1|^2\|E_1\| + |E_2|^2\|E_2\|)$ because the $first$ sets of c-continuation has an $O(|\Sigma_E|)$ size. The time and space complexity of `Equa_intersect` is exactly the same no matter the input expressions are deterministic or not, since dropping the labels requires linear time and no additional space.

5 Experimental Evaluation

In this section, we evaluate the effectiveness and efficiency of our algorithms on regular expression datasets. In the following, `Pos_intersect`, `Follow_intersect`, `CCon_intersect` and `Equa_intersect` are abbreviated as PO, FO, CC and EQ respectively.

Benchmarks. SRE is a dataset of standard regular expressions randomly generated on alphabets of $1 \leq |\Sigma| \leq 10$ and symbol occurence ranges from 1 to 1000 with step 10. For every step of symbol occurences, we generate an expression as E_1. And generate 100 expressions as E_2 whose symbol occurence ranges from 1 to 1000 with step 10, giving a total of 10000 pairs of expressions.

DRE is a dataset of 27129 pairs of deterministic regular expressions used in practical applications selected from [21], which are collected and normalized from XSD, DTD and Relax NG schema files. DRE is evaluated for multiple excellent properties of deterministic regular expressions in their position automaton and its quotients [11, 19].

Baselines. To evaluate the effectiveness and efficiency of our algorithms, we selected six state-of-the-art SMT solvers for comparison: Z3str3 [7], Z3-Trau [1], Z3seq [52], Ostrich [25], CVC4 [43] and Z3str3RE [6]. For discussions of the tools, see Chapter 6.

Configurations. We implemented a prototype of our algorithms in Ocaml. Our experiments were run on a machine with 3.40GHz Intel i7-6700 8 CPU and 8G RAM, running Ubuntu 20. All baselines were configured in the settings reported in their original documents. A timeout of 20 seconds is used.

Efficiency and Effectiveness. In the following tables, True Positive denotes E_1 and E_2 intersect and the algorithm reported true. True Negative denotes E_1 and E_2 do not intersect and the algorithm reported false. False Positive denotes E_1 and E_2 do not intersect but the algorithm reported true. False Negative denotes E_1 and E_2 do not intersect but the algorithm reported true. Unknown is the sum of "unknown" responses, which can be resulted from when non-termination in their algorithms are detected or a resource limit is met. Program Crash denotes the sum of crashes. Timeout denotes the sum of reaching the time limit and Time is the total runtime of each algorithm. The best results achieved by the algorithms are shown in bold. In the cactus plots, algorithms that are further to the right and closer to the bottom of the plot have better performance.

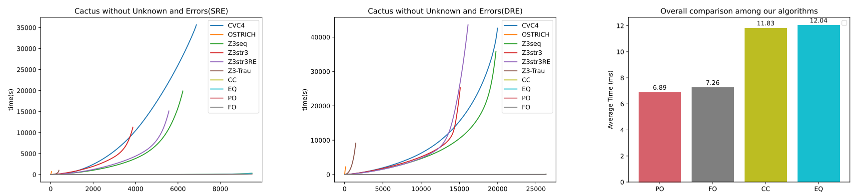
Table 1. Detailed results for the SRE benchmark.

	Z3-Trau	OSTRICH	Z3seq	CVC4	Z3str3	Z3str3RE	PO	FO	CC	EQ
True Positive	2	1	3486	1581	3740	4095	**4642**	**4642**	**4642**	**4642**
True Negative	386	38	2747	5288	137	1481	**5358**	**5358**	**5358**	**5358**
False Positive	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**
False Negative	36	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**
Program Crash	37	**0**	**0**	**0**	9	**0**	**0**	**0**	**0**	**0**
Unknown	**0**	36	**0**	**0**	396	**0**	**0**	**0**	**0**	**0**
Timeout	9539	9925	3767	3131	5727	4423	**0**	**0**	**0**	**0**
Time(s)	192353	199863	95236	98283	126844	103611	**157**	171	340	348

Table 2. Detailed results for the DRE benchmark.

	Z3-Trau	OSTRICH	Z3seq	CVC4	Z3str3	Z3str3RE	PO	FO	CC	EQ
True Positive	972	11	18536	12982	14226	14891	**19252**	**19252**	**19252**	**19252**
True Negative	484	108	1241	6995	889	1235	**7877**	**7877**	**7877**	**7877**
False Positive	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**
False Negative	288	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**
Program Crash	103	**0**	**0**	**0**	9	**0**	**0**	**0**	**0**	**0**
Unknown	**0**	4302	**0**	**0**	348	**0**	**0**	**0**	**0**	**0**
Timeout	25282	22708	7352	7152	11657	11003	**0**	**0**	**0**	**0**
Time(s)	518433	501129	182877	185641	260525	263633	98.8	**98.6**	99.4	99.0

The results for SRE benchmark are shown in Table 1 and Figure 2(a). All of our algorithms solved all of the instances correctly. Including timeouts, the fastest algorithm PO achieves a speedup of 606x over Z3seq, 626x over CVC4,



(a) Results for the SRE benchmark (b) Results for the DRE benchmark (c) Overall comparison among our algorithms

Fig. 2. Plots showing cumulative results for each benchmark.

659x over Z3str3RE, 807x over Z3str3, 1225x over Z3-Trau, and 1273x over OSTRICH. The difference among curves of our algorithms are minor compared to the performance of the baselines on Figure 2(a) and 2(b), thus close to coincide.

The results for DRE benchmark are shown in Table 2 and Figure 2(b). All of our algorithms solved all of the instances correctly and 7152 instances uniquely solved by ours. Including timeouts, the fastest algorithm FO achieves a speedup of 1855x over Z3seq, 1883x over CVC4, 2642x over Z3str3, 2673x over Z3str3RE, 5082x over OSTRICH and 5257x over Z3-Trau. The experimental results reveal that our algorithms are more efficient for deterministic regular expressions, since at line 16 of all our algorithms, the position tuple corresponding to a symbol is always unique for deterministic regular expressions.

Also we found CVC4's implementation of `firstChars` function is overapproximated when handling intersection between regular expressions, which partially explains their performance in our experiments. Furthermore, for extreme cases when both the symbol occurrence of the expressions up to 3000, we also observed all of the baselines reported timeout while our algorithms can find the correct solutions.

Relations among algorithms. To investigate the relation among our algorithms, relations among automata is necessary. From [14, 22], we have:

Lemma 4. *For regular expressions in star normal form,* $=_c \subseteq \equiv_f \subseteq \equiv_c$.

In [38], authors showed mn states are sufficient and necessary for an NFA to accept the intersection of an m-state NFA and an n-state NFA in the worst case. According to the commutativity of intersection, denote ordering $M_1 \succeq_{nsc} M_2$ iff automaton M_1 has more or equal states than automaton M_2, then we can conclude:

Theorem 7. *For regular expressions* $E_1, \ldots, E_m$ *in star normal form, we have:*

$$\bigcap_{i=1}^{m} M_{pos}(E_i) \succeq_{nsc} \bigcap_{i=1}^{m} M_{ccon}(E_i)/_{=_c} \succeq_{nsc} \bigcap_{i=1}^{m} M_f(E_i) \succeq_{nsc} \bigcap_{i=1}^{m} M_e(E_i). \quad (1)$$

From the theorem above, we can easily deduce the relations among the worst case search space of our algorithms. Figure 2(c) shows the average time of our

algorithms to solve a instance in all our benchmarks, where PO is the fastest and EQ is the slowest. We also observed the efficiency of PO and FO is higher than that of c-continuation-based algorithms. This is due to a position tuple (of PO) has constant size (recall in Chapter 1 when the number of input expressions is fixed as two) and a position set tuple (of FO) have linear sizes while partial derivatives or c-continuations has sizes at worst-case quadratic [2, 16, 35]. This fact reveals smaller cost in identification of states can significantly accelerate regular expression intersection non-emptiness checking algorithms. In general, FO is recommended for smaller solution space and average case performance.
Summary. Overall, all our algorithms outperforms all baselines in both effectiveness and efficiency in solving intersection non-emptiness problems for regular expressions.

6 Related Work

Computational Complexity. Bala [4] showed PSPACE-completeness of intersection non-emptiness problem on regular expressions without + operators whose star height is 2 and NP-completeness for those star height is at most 1. Martens et al. [44] showed even for very innocent fragments of regular expressions, intersection non-emptiness problem is intractable. Gelade and Neven [29, 30] showed that the intersection of regular expressions are double exponentially more succinct than ordinary regular expressions. Arrighi et al. [3] investigated complexity of intersection non-emptiness problem within hierarchies in star-free language classes, namely Staubing-Thérien hierarchy and Brzozowski dot-depth hierarchy. Fernau et al. [28] provided a systematic study of intersection non-emptiness problem of regular languages under the (Strong) Exponential Time Hypothesis. Recently, Fernau et al. [27] investigated parameterized complexity of intersection non-emptiness problem of various models for regular languages.

Z3-Trau [1] is based on Z3, which depends on parametric flat automata to handle string constraints, with both under- and over-approximations. The evaluation of Z3-Trau exposed 324 soundness errors and 140 crashes on our datasets. OSTRICH [25] is a string solver implementing a transducer model and handle regular language intersection via cross product algorithm based on [48]. OSTRICH reported 4338 "unknown" resposes on our benchmarks. Experimentally we found pure automata methods are inefficient in solving regular expression intersection non-emptiness. Z3seq [52] is a hybrid solver which reasons on sequences of characters serving as the default string solver in current Z3. For regular language constraints, Z3seq uses symbolic Boolean derivatives based on Brzozowski's [10] and Antimirov's [2] derivatives without constructing automata. CVC4's the decision procedure [43] for regular expression constraints extends Antimirov's partial derivatives [2] similar to [13]. Experimentally derivative-based solvers show advantages over the other solvers, however outperformed by our algorithms: we utilized derivatives in a different manner from the derivative-based solvers, firstly our algorithms are based on linearization technique, also we simultate cross product on derivatives instead of integrating intersection operation into derivatives. Z3str3 [7] handles Boolean combinations of regular expressions with reduction to word equations. On our benchmarks, Z3str3 reported unknown and crashes on 762 instances. Z3str3RE [6] is based on Z3str3 with the length-aware automata-based algorithm and heuristics. In the experiments we found Z3str3RE's optimizations and bug-fixes to Z3str3 are effective, however the cost from intersection of automata has adverse impact in its efficiency compared to our algorithms.

7 Concluding Remarks

In this paper, we have given four algorithms based on online automata construction simulation to solve the intersection non-emptiness problem of regular expressions, which are compared against six state-of-the-art SMT solvers (namely, CVC4, Z3seq, Z3str3, Z3-Trau, OSTRICH and Z3str3RE) over synthetic and real-world datasets. Overall we show that our algorithms out-performed all of the solvers mentioned above.

Our algorithms also show high extension prospects: algorithms can be easily modified to output the whole intersection automaton to handle get-model constraints instead of only checking non-emptiness, add mechanisms for extended features in real-world regular expressions such as character classes, matching precedence and capturing groups, and introduce local search or real-time heuristic search strategies to evaluate the cost during the searching procedure for improving practical performance.

References

1. Abdulla, P.A., Atig, M.F., Chen, Y.F., Diep, B.P., Dolby, J., Janků, P., Lin, H.H., Holík, L., Wu, W.C.: Efficient handling of string-number conversion. In: PLDI 2020. pp. 943–957 (2020)
2. Antimirov, V.M.: Partial derivatives of regular expressions and finite automaton constructions. Theoretical Computer Science **155**, 291–319 (1996)
3. Arrighi, E., Fernau, H., Hoffmann, S., Holzer, M., Jecker, I., de Oliveira Oliveira, M., Wolf, P.: On the complexity of intersection non-emptiness for star-free language classes. CoRR **abs/2110.01279** (2021)
4. Bala, S.: Intersection of regular languages and star hierarchy **2380**, 159–169 (2002)
5. Berry, G., Sethi, R.: From regular expressions to deterministic automata. Theoretical Computer Science **48**, 117–126 (1986)
6. Berzish, M., Day, J.D., Ganesh, V., Kulczynski, M., Manea, F., Mora, F., Nowotka, D.: Towards more efficient methods for solving regular-expression heavy string constraints. Theoretical Computer Science **943**, 50–72 (2023)
7. Berzish, M., Ganesh, V., Zheng, Y.: Z3str3: A string solver with theory-aware heuristics. In: FMCAD 2017. pp. 55–59 (2017)
8. Broda, S., Holzer, M., Maia, E., Moreira, N., Reis, R.: A mesh of automata. Information and Computation **265**, 94–111 (2019)
9. Broda, S., Machiavelo, A., Moreira, N., Reis, R.: Position automaton construction for regular expressions with intersection. In: DLT 2016. pp. 51–63 (2016)
10. Brzozowski, J.A.: Derivatives of regular expressions. J. ACM **11**(4), 481–494 (1964)
11. Brüggemann-Klein, A.: Regular expressions into finite automata. Theoretical Computer Science **120**, 197–213 (1993)
12. Brüggemann-Klein, A., Wood, D.: One-unambiguous regular languages. Information and Computation **140**(2), 229–253 (1998)
13. Caron, P., Champarnaud, J.M., Mignot, L.: Partial derivatives of an extended regular expression. In: LATA 2011. pp. 179–191 (2011)
14. Champarnaud, J.M., Ouardi, F., Ziadi, D.: Normalized expressions and finite automata. International Journal of Algebra and Computation **17**, 141–154 (2007)
15. Champarnaud, J.M., Ziadi, D.: From mirkin's prebases to antimirov's word partial derivatives. Fundam. Informaticae **45**, 195–205 (2001)
16. Champarnaud, J.M., Ziadi, D.: Canonical derivatives, partial derivatives and finite automaton constructions. Theoretical Computer Science **289**(1), 137–163 (2002)
17. Chang, C.H., Paige, R.: From regular expressions to DFA's using compressed NFA's. In: CPM 1992. pp. 90–110 (1992)
18. Chapman, C., Stolee, K.T.: Exploring regular expression usage and context in Python. In: ISSTA 2016. pp. 282–293 (2016)
19. Chen, H.: Finite automata of expressions in the case of star normal form and one-unambiguity. In: Technical report. ISCAS-LCS-10-11 (2010)
20. Chen, H., Chen, L.: Inclusion test algorithms for one-unambiguous regular expressions. In: ICTAC 2008. vol. 5160, pp. 96–110 (2008)
21. Chen, H., Li, Y., Dong, C., Chu, X., Mou, X., Min, W.: A large-scale repository of deterministic regular expression patterns and its applications. In: PAKDD 2019, vol. 11441, pp. 249–261 (2019)
22. Chen, H., Lu, P.: Derivatives and finite automata of expressions in star normal form. In: LATA 2017. pp. 236–248 (2017)
23. Chen, H., Xu, Z.: Inclusion algorithms for one-unambiguous regular expressions and their applications. Science of Computer Programming **193**, 102436 (2020)

24. Chen, H., Yu, S.: Derivatives of regular expressions and an application. In: Computation, Physics and Beyond, vol. 7160, pp. 343–356 (2012)
25. Chen, T., Flores-Lamas, A., Hague, M., Han, Z., Hu, D., Kan, S., Lin, A.W., Ruemmer, P., Wu, Z.: Solving string constraints with regex-dependent functions through transducers with priorities and variables. POPL 2022 **6**, 1–31 (2022)
26. De Moura, L., Bjørner, N.: Z3: An efficient SMT solver. In: TACAS 2008. pp. 337–340 (2008)
27. Fernau, H., Hoffmann, S., Wehar, M.: Finite automata intersection non-emptiness: Parameterized complexity revisited. ArXiv **abs/2108.05244** (2021)
28. Fernau, H., Krebs, A.: Problems on finite automata and the exponential time hypothesis. Algorithms **10**(1), 24 (2017)
29. Gelade, W.: Succinctness of regular expressions with interleaving, intersection and counting. Theoretical Computer Science **411**(31), 2987–2998 (2010)
30. Gelade, W., Neven, F.: Succinctness of the complement and intersection of regular expressions. ACM Trans. Comput. Logic **13**(1) (2012)
31. Glushkov, V.M.: The abstract theory of automata. Russian Mathematical Surveys **16**, 1–53 (1961)
32. GNU: "Gawk: Effective AWK Programming" (2022), `https://www.gnu.org/software/gawk/manual/`
33. GNU: "GNU Grep: Print lines matching a pattern" (2022), `https://www.gnu.org/software/grep/manual/`
34. GNU: "GNU Sed: a stream editor" (2022), `https://www.gnu.org/software/sed/manual/`
35. Gruber, H., Holzer, M.: Language operations with regular expressions of polynomial size. Theoretical Computer Science **410**(35), 3281–3289 (2009)
36. Guanciale, R., Gurov, D., Laud, P.: Private intersection of regular languages. In: PST 2014. pp. 112–120 (2014)
37. Henriksen, J.G., Jensen, J., Jørgensen, M., Klarlund, N., Paige, R., Rauhe, T., Sandholm, A.: Mona: Monadic second-order logic in practice. In: TACAS. pp. 89–110 (1995)
38. Holzer, M., Kutrib, M.: State complexity of basic operations on nondeterministic finite automata. In: CIAA 2003, vol. 2608, pp. 148–157 (2003)
39. Hunt, H.B., Rosenkrantz, D.J., Szymanski, T.G.: On the equivalence, containment, and covering problems for the regular and context-free languages. Journal of Computer and System Sciences **12**(2), 222–268 (1976)
40. Ilie, L., Yu, S.: Follow automata. Information and Computation **186**(1), 140–162 (2003)
41. Kozen, D.: Lower bounds for natural proof systems. In: SFCS 1977. pp. 254–266 (1977)
42. Li, Y., Chen, Z., Cao, J., Xu, Z., Peng, Q., Chen, H., Chen, L., Cheung, S.C.: ReDoSHunter: A combined static and dynamic approach for regular expression DoS detection. In: USENIX Security '21. pp. 3847–3864 (2021)
43. Liang, T., Tsiskaridze, N., Reynolds, A., Tinelli, C., Barrett, C.: A decision procedure for regular membership and length constraints over unbounded strings. In: FroCoS 2015, vol. 9322, pp. 135–150 (2015)
44. Martens, W., Neven, F., Schwentick, T.: Complexity of decision problems for simple regular expressions. pp. 889–900 (2004)
45. McNaughton, R., Yamada, H.: Regular expressions and state graphs for automata. IRE Trans. Electron. Comput. **9**, 39–47 (1960)
46. Middleton, J., Toro Icarte, R., Baier, J.: Real-Time Heuristic Search with LTLf Goals. In: IJCAI 2022. pp. 4785–4792 (2022)

47. Mirkin, B.G.: An algorithm for constructing a base in a language of regular expressions. Journal of Symbolic Logic **36**(4), 694–694 (1971)
48. Møller, A.: dk.brics.automaton, `https://www.brics.dk/automaton/`
49. de Oliveira Oliveira, M., Wehar, M.: On the fine grained complexity of finite automata non-emptiness of intersection. In: DLT 2020, vol. 12086, pp. 69–82 (2020)
50. Rabin, M.O., Scott, D.: Finite automata and their decision problems. IBM Journal of Research and Development **3**(2), 114–125 (1959)
51. Sakarovitch, J.: Elements of automata theory. Cambridge University Press (2009)
52. Stanford, C., Veanes, M., Bjørner, N.: Symbolic Boolean derivatives for efficiently solving extended regular expression constraints. In: PLDI 2021. pp. 620–635 (2021)
53. Todd Wareham, H.: The parameterized complexity of intersection and composition operations on sets of finite-state automata. In: CIAA 2001, pp. 302–310 (2001)
54. Wehar, M.: Hardness results for intersection non-emptiness. In: ICALP 2014, vol. 8573, pp. 354–362 (2014)
55. Yu, S.: Regular languages. In: Handbook of Formal Languages, Vol. 1: Word, Language, Grammar. pp. 41–110 (1997)
56. Yu, S., Zhuang, Q.: On the state complexity of intersection of regular languages. ACM SIGACT News **22**(3), 52–54 (1991)
57. Ziadi, D., Ponty, J.L., Champarnaud, J.M.: Passage d'une expression rationnelle à un automate fini non-déterministe. Bulletin of The Belgian Mathematical Society-simon Stevin **4**, 177–203 (1997)