A Regular Matching Constraint for String Variables

Roberto Amadini\textsuperscript{1}, Peter J. Stuckey\textsuperscript{2,3}

\textsuperscript{1}University of Bologna, Bologna, Italy
\textsuperscript{2}Monash University, Melbourne, Victoria, Australia
\textsuperscript{3}OPTIMA ARC Industrial Transformation and Training Centre, Melbourne, Australia
roberto.amadini@unibo.it, peter.stuckey@monash.edu

Abstract
Using a regular language as a pattern for string matching is nowadays a common—and sometimes unsafe—operation, provided as a built-in feature by most programming languages. A proper constraint solver over string variables should support most of the operations over regular expressions and related constructs. However, state-of-the-art string solvers natively support only the membership relation of a string variable to a regular language. Here we take a step forward by defining a specialised propagator for the match operation, returning the leftmost position where a pattern can match a given string. Empirical evidences show the effectiveness of our approach, implemented within the constraint programming framework, and tested against state-of-the-art string solvers.

1 Introduction
Regular expressions are a powerful and useful way of processing strings. Nowadays they are commonly used in languages such as Python, Java and JavaScript for data parsing and validation. The widespread use of regular expressions arguably coincides with the increasing development of web applications heavily using string operations both explicitly (i.e., by directly invoking string functions) and implicitly (i.e., by silently converting objects of different type into strings). Unsurprisingly, this led to the development of a number of both static and dynamic string analysis frameworks especially targeted to cybersecurity issues [Bultan et al., 2017].

In this paper we focus on the dynamic side of string analysis, and more precisely on string constraint solving [Amadini, 2021], to show how the Constraint Programming (CP) paradigm [Rossi et al., 2006] can be used to tackle string constraints where a pattern —be it a regular expression, an automaton or a grammar— denoting a regular language is matched against a string variable. The key practical motivation of this work is the constraint-based reasoning about string manipulating programs, e.g., JavaScript web applications, where match constructs, and related operations like string replacement, are heavily used.

CP is a well-established framework for general-purpose constraint solving and optimization. Handling constraints over regular expressions is not new to CP, having been introduced in the early 2000’s by Pesant [2004] for fixed sequences of integer variables, and more recently retrofitted by Amadini et al. [2018]. However, these works refer to the constraint regular\((\cdot, R)\) which only considers the membership of \(x\) to the regular language \(L(R)\) denoted by the finite-state automaton \(R\), and not what is, nowadays, probably the most common operation: using \(R\) as a pattern to be matched against \(x\). Instead, in this paper we will show how to handle in CP the constraint match\((x, \rho)\) returning the position in \(x\) of the leftmost possible match of a string of \(L(\rho)\). To our knowledge, at present no CP propagator exists for match.

To represent the finite domain of bounded-length string variables, we consider the dashed string abstraction as done in [Amadini et al., 2018]. Given a finite alphabet \(\Sigma\) and a maximum string length \(\lambda\), dashed strings (over-)represent subsets of \(\{w \in \Sigma^* \mid |w| \leq \lambda\}\) through the concatenation of distinct sets of characters called blocks. Using blocks of characters prevents the eager unfolding of the domain of a string variable into \(O(\lambda)\) integer variables, which is definitely a big plus when \(\lambda\) is large.

We compared the performance of our approach against different state-of-the-art string solving technologies. Empirical results underline the importance of implementing native support for the match constraint.

2 Preliminaries
In this section we give the necessary background notions about strings, regular languages, constraint programming and (dashed) string solving.

2.1 Strings and Regular Languages
Let \(\Sigma\) be a finite, non-empty alphabet and \(\Sigma^*\) the set of all its strings. A string \(w \in \Sigma^*\) is a finite sequence of \(|w| \geq 0\) characters of \(\Sigma\), where \(|w|\) is the length of \(w\) (the empty string is denoted with \(\epsilon\)). We focus on bounded-length strings: fixed \(\lambda \in \mathbb{N}\), we only consider strings \(w \in \Sigma^*\) such that \(|w| \leq \lambda\).

The concatenation of \(v, w \in \Sigma^*\) is denoted by \(v \cdot w\) (or simply by \(vw\)) while \(w^n\) is the iterated concatenation: \(w^0 = \epsilon\) and \(w^n = w w^{n-1}\) for \(n > 0\). Similarly, given \(V, W \subseteq \Sigma^*\), we denote by \(V \cdot W = \{vw \mid v \in V, w \in W\}\) (or simply by \(V W\)) their concatenation and with \(W^n\) the iterated concatenation \((W^0 = \{\epsilon\})\). We adopt a 1-based array notation: \(w[i]\) is the \(i\)-th symbol of \(w\), with \(1 \leq i \leq |w|\).
We define \( l.u = \{ x \in \mathbb{Z} \mid 1 \leq x \leq u \} \) and denote with \( w[i..j] \) the substring of \( w \) from character \( i \) to character \( j \), i.e., \( w[i..j] = w[i]w[i+1] \cdots w[j] \) if \( i > j \), then \( w[i..j] = \epsilon \).

A subset \( L \subseteq \Sigma^* \) is called a language. A regular language can be defined in several equivalent ways. For example, as the language \( L(R) \) accepted by a finite-state automaton \( R \), the language \( L(r) \) denoted by regular expression \( r \), or the language \( L(T) \) generated by regular grammar \( \Gamma \).\(^1\) We call regular pattern (shortly, pattern) a formalism \( \rho \) describing a regular language \( L(\rho) \).

A deterministic finite-state automaton (DFA) is a tuple \( R = (\Sigma, Q, \delta, q_0, F) \) where: \( \Sigma \) is the alphabet, \( Q \) the finite set of states, \( \delta : Q \times \Sigma \to Q \) the transition function, \( q_0 \in Q \) the initial state and \( F \subseteq Q \) the accepting or final states. A computation for \( w \in \Sigma^* \) is a sequence of transitions \( s_0 \to s_1 \to \cdots s_n \) where \( n = |w| \) and \( s_1, \ldots, s_n \in Q \) such that \( s_i = \delta(s_{i-1}, w[i]) \) for \( i = 1, \ldots, n \). We denote with \( \delta : Q \times \Sigma \to Q \) the extension of \( \delta \) to strings such that \( \delta(q, w) = q' \) if and only if \( q \to \cdots \to q' \) is a computation for \( w \). A string \( w \in \Sigma^* \) is accepted by \( R \) if \( \delta(q_0, w) \in F \). The language of \( R \) is \( L(R) = \{ w \in \Sigma^* \mid \delta(q_0, w) \in F \} \).

If \( \delta \) is a total function, the DFA is complete. If \( \delta \) is partial, we can get a complete DFA by extending \( \Sigma \) to \( \Sigma' = \Sigma \cup \{ q_1 \} \), and \( \delta \) to \( \delta \cup \{ (q, q_1, q_1) \mid q \in \Sigma', c \in \Sigma, \delta(q, c) = \bot \} \).

The complement automaton \( \overline{R} \) having \( L(\overline{R}) = \Sigma^* \setminus L(R) \) is easily computed from a complete DFA \( \overline{R} \) by complementing its final states. If there does not exist a DFA \( R' \) with less than \( |Q| \) states such that \( L(R') = L(R) \), then \( R \) is minimal. Given a regular pattern \( \rho \), we denote by \( DFA(\rho) \) the minimal and complete DFA such that \( L(\rho) = L(DFA(\rho)) \).

If \( R = (Q, \Sigma, \delta, q_0, F) \) is complete, a state \( q \in Q \) is said universally accepting if all computations from \( q \) bring to a state \( q' \in F \). Dually, \( q \in \overline{F} \) is universally rejecting if all computations from \( q \) bring to \( q' \in \overline{F} \). If \( R \) is minimal, it has at most one universally accepting state, and at most one universally rejecting state (if they have two or more universally accepting/rejecting states then those states can be merged, so \( R \) would not be minimal). We denote with \( minlen(R) \) the minimum length of a string accepted by \( R \).

A well-known way to identify regular languages is using regular expressions. We define inductively the set \( \mathcal{RE} \) of regular expressions, and the language \( L(\cdot) \) denoted by each \( r \in \mathcal{RE} \), as: (i) \( \emptyset \in \mathcal{RE} \) denoting \( L(\emptyset) = \emptyset \); (ii) if \( c \in \Sigma \cup \{ \epsilon \} \), then \( c \in \mathcal{RE} \) denoting \( L(c) = \{ c \} \); (iii) if \( r, r' \in \mathcal{RE} \), then \( r \cdot r' \in \mathcal{RE} \) denoting \( L(r \cdot r') = L(r) \cdot L(r') \), and \( r|r' \in \mathcal{RE} \) denoting \( L(r|r') = L(r) \cup L(r') \); (iv) if \( r \in \mathcal{RE} \) then \( r^* \in \mathcal{RE} \) denoting \( L(r^*) = L(r)^* \).

### 2.2 Constraint Programming and String Solving

Constraint programming [Rossi et al., 2006] solves Constraint Satisfaction Problems (CSPs), defined by a triple \((\mathcal{X}, \mathcal{D}, \mathcal{C})\) where: \( \mathcal{X} = \{ x_1, \ldots, x_n \} \) is a finite set of variables, \( \mathcal{D} = \{ \mathcal{D}(x_1), \ldots, \mathcal{D}(x_n) \} \) is a set of domains, where each \( \mathcal{D}(x_i) \) is the set of the values that \( x_i \) can take, and \( \mathcal{C} \) is a set of constraints over the variables of \( \mathcal{X} \) defining the feasible assignments. The goal is to find an assignment of domain values to corresponding variables satisfying all of the constraints in \( \mathcal{C} \). To do so, CP heavily relies on propagators. A propagator for a constraint \( \mathcal{C} \) defined over variables \( x_1, \ldots, x_n \) is an algorithm that aims to prune the most inconsistent values from \( \mathcal{D}(x_1), \ldots, \mathcal{D}(x_n) \) according to the semantics of \( \mathcal{C} \). The level of pruning is a compromise between efficacy (how many values we prune) and efficiency. Most of the propagators are not complete: a fixpoint is reached even if the domains could in principle be further pruned.

Most CSPs are defined over finite domains, i.e., \( \mathcal{D} \) only contains finite sets, and \( \mathcal{X} \) only contains integer variables. In this work, we also consider constraints over bounded-length strings. Fixing a finite alphabet \( \Sigma \) and an upper bound \( \lambda \in \mathbb{N} \), a CSP with bounded-length strings contains \( k > 0 \) string variables \( \{ x_1, \ldots, x_k \} \subseteq \mathcal{X} \) such that \( D(x_i) \subseteq \Sigma^\lambda \) and \( |x_i| \leq \lambda \). The set \( \mathcal{C} \) may contain well-known string constraints, such as string length, (dis-)equality, concatenation, substring selection, and finding/replacing. We refer to constraint solving involving string variables as string constraint solving.

Different approaches to string constraint solving have been proposed, based on: automata [Li and Ghosh, 2013; Tabei et al., 2013], word equations [Berzish et al., 2017; Barbosa et al., 2022], unfolding (using either bit-vector solvers [Kiežun et al., 2012; Saxena et al., 2010] or CP [Scott et al., 2017]), and dashed strings [Amadini et al., 2020]. In this work we focus on the latter approach, which can be considered the state-of-the-art for CP string solving.

Dashed strings are special cases of regular expressions, defined by \( k > 0 \) blocks \( S_1^{i_1..u_1} \cdots S_k^{i_k..u_k} \), where \( S_i \subseteq \Sigma \) and \( 0 \leq i_1 \leq u_1 \leq \lambda \) for \( i = 1, \ldots, k \) and \( \sum_{i=1}^k i_1 \leq \lambda \). For each block \( S_i^{i_1..u_i} \), we call \( S_i \) its base and \( i_1, u_i \) its lower/upper bound, respectively. We denote with \( \text{maxlen}(X) \) the sum of the upper bounds of each block of \( X \), with \( \text{len}(X) \) the length of a dashed string \( X \), and with \( |X| \) the number of blocks of \( X \). We consider only normalised dashed strings, where the adjacent blocks have distinct bases and the null block \( \emptyset \) occurs only to denote the empty string.

Let \( \gamma(S_1^{i_1..u_1} \cdots S_k^{i_k..u_k}) = \gamma(S_1^{i_1..u_1}) \cdots \gamma(S_k^{i_k..u_k}) \cap S \), where the intersection \( S = \{ w \in \Sigma^* \mid |w| \leq \lambda \} \) excludes the strings with length greater than \( \lambda \). Normalisation entails that each string \( w \in S \) has a unique dashed string \( X \) such that \( w = \gamma(X) \).

Given dashed strings \( X \) and \( Y \) we define the relation \( X \sqsubseteq Y \iff \gamma(X) \subseteq \gamma(Y) \) denoting a relation "is more precise than" between dashed strings. Unfortunately, the set of dashed strings does not form a lattice according to \( \sqsubseteq \) [Amadini et al., 2020]. For example, there is no single "best" dashed string denoting \( [ab, ba] \subseteq \Sigma \): three maximally accurate but incomparable dashed strings that represent this set are \( \{ a, b \}^{2..2} \setminus \{ a \}^{1..1} \setminus \{ b \}^{0..0} \setminus \{ a \}^{1..1} \setminus \{ b \}^{0..0} \), each represents a different over approximation. This means that workarounds have to be used to preserve the soundness.\(^2\)
Let \( L \) be a regular pattern with \( \Sigma^* \leq L \), which semantics is 
\[
\sigma(q_0, a) = \begin{cases} 
q_1 & \text{if } a \in F \\
q_0 & \text{otherwise}
\end{cases}
\]
for each \( a \in \Sigma \). We represent the domain of \( x \) with a dashed string. However, this propagator also works when \( D(x) \) is a fixed sequence of integer variables.

3 Matching Regular Expressions

In this section we show how we propagate the match constraint. To do so, we need to introduce some notions and properties exploited by our propagator.

**Definition 1.** Let \( \Sigma \) be a finite alphabet and \( \rho \) a pattern for \( \Sigma \). For each \( w \in \Sigma^* \) we define the set \( \mathcal{MI}(w, \rho) \leq 0,|w| \) of matching indexes of \( \rho \) in \( w \) as:

\[
\begin{align*}
\{0\} & \quad \text{if } \exists i, j \suchthat w[i:j] \in L(\rho) \\
\{1\} & \quad \text{else if } w = \epsilon \in L(\rho) \\
\{i \in 1..|w| \suchthat \exists j \suchthat w[i:j] \in L(\rho) \} & \quad \text{otherwise}
\end{align*}
\]

In practice, \( \mathcal{MI}(w, \rho) \) is the set of all the positions in \( w \) where \( \rho \) matches a substring \( w' \) of \( w \), i.e., \( w' \in L(\rho) \). For example, if \( \Sigma = \{a, b, c\} \), \( \rho = ab(a|c)^* \) and \( w = cabcaab \) then \( \mathcal{MI}(w, \rho) = \{2, 6\} \). If instead \( w = caabcaab \), then \( \mathcal{MI}(w, \rho) = \{0\} \) because no match occurs. Note that with this definition if \( \epsilon \in L(\rho) \) then for each \( w \in \Sigma^* \setminus \{\epsilon\} \) we have \( \mathcal{MI}(w, \rho) = 1 \cdot |w| \) because \( w[i:i-1] = \epsilon \in L(\rho) \) for \( i = 1, \ldots, |w| \). In this way, we capture the fact that \( \epsilon \) occurs at each position of a non-empty string, being the empty string substring of every string.

Let \( \rho \) be a regular pattern. The (full) wrapping of \( \rho \) is the minimal pattern \( \rho^w \) denoting the language \( \Sigma^* \cdot L(\rho) \cdot \Sigma^* \). We also define the left-wrapping \( \rho^l \), denoting \( \Sigma^* \cdot L(\rho) \), and the right-wrapping \( \rho^r \), denoting \( L(\rho) \cdot \Sigma^* \). For example, if \( \Sigma = \{a, b, c\} \) and \( \rho = ab(a|c)^* \) its wrapping is \( \rho^w = (a|b)c^* ab(a|b)c^* \), the left-wrapping is \( \rho^l = (a|b)c^* ab(a|b)c^* \) and the right-wrapping is \( \rho^r = ab(a|b)c^* \).

**Lemma 1.** Let \( \rho \) be a regular pattern with \( L(\rho) \neq \emptyset \). \( R = \text{DFA}(\rho) \) and \( \hat{R} = \text{DFA}(\rho^w) \). The following properties hold:

(i) \( \hat{R} \) has no universally rejecting states

(ii) \( \hat{R} \) has exactly one universally accepting state

**Proof.** (i) Suppose \( \hat{R} \) has a universally rejecting state \( q_\perp \). \( \hat{R} \) is minimal, so there is a \( R \)-computation \( q_0 \to \cdots \to q_\perp \) for a certain \( u_0 \in \Sigma^* \). Hence, for each \( w \in \Sigma^* \) we would have \( w_0w \notin L(\hat{R}) \). In particular, if \( w \in L(\hat{R}) \) we would have \( w_0w \notin L(\hat{R}) = L(\Sigma R^\ast) \subseteq L(*R) \) which is impossible being \( w_0w \in L(*R) \). Specifically, if \( \epsilon \in L(\hat{R}) \) such that \( \epsilon \notin L(\hat{R}) \). It is possible being \( L(R) = \Sigma^* \cdot L(\hat{R}) \cdot \Sigma^* \) closed under concatenation.

(ii) Because \( L(\rho) \neq \emptyset \), there is at least a final state. If \( \hat{R} \) has no universally accepting states, there exist two states \( q_i, q_j \) such that \( \delta(q_i, a) = q_j \). So there is a \( w \in L(\hat{R}) \) such that \( wa \notin L(\hat{R}) \) which is impossible because \( L(\hat{R}) = \Sigma^* \cdot L(\hat{R}) \cdot \Sigma^* \) closed under concatenation.

3.1 Propagating match

Let us now explain how we propagate \( i = \text{match}(x, \rho) \), which semantics is \( i = \min(\mathcal{MI}(x, \rho)) \). We represent the domain of \( x \) with a dashed string. However, this propagator also works when \( D(x) \) is a fixed sequence of integer variables.
as in [Pesant, 2004; Scott et al., 2017]. In fact, the sequence
\( x_1 \cdots x_q \) where \( D(x_i) \subseteq X = \{ 0, 1 \} \) is the j-th symbol
of \( \Sigma \) corresponds to dashed string \( D(x_1)D(x_2) \cdots D(x_q) \).

It is also important to note that, because we assume \( \epsilon \) occ-
curring at each position of a string, we have \( \epsilon \in \mathcal{L}(\rho) \) \( \Rightarrow \)
match\( (x, \rho) = 1 \) for any string variable \( x \). Hence, before propa-
gating match we should check if \( \epsilon \in \mathcal{L}(\rho) \). If so, we can
simply rewrite \( i = \text{match}(x, \rho) \) into \( i = 1 \). Because we focus on
the more interesting cases, in the following we consider only patterns \( \rho \) such that \( \epsilon \notin \mathcal{L}(\rho) \).

A specialized match propagator is not strictly necessary, e.g., we can impose \( 0 \leq i \leq |x| \) and rely on the regular
propagator to rewrite \( i = \text{match}(x, \rho) \) into:

\[
\begin{align*}
\neg \text{regular} (x, DFA(\varphi_\rho)) & \quad \text{if } i = 0 \\
\neg \text{regular} (x[1..i-1], DFA(\varphi_\rho)) \\
\wedge \text{regular} (x[i..|x|], DFA(\varphi_\rho)) \\
\wedge \forall j \in 1..i-1 : \neg \text{regular} (x[j..|x|], DFA(\varphi_\rho)) & \quad \text{if } i \neq 0
\end{align*}
\tag{1}
\]

This rewriting, especially when \( i \) is not fixed, is inefficient, as it involves a universal quantification over \( 1..i-1 \) and many

reflections.\(^\dagger\) This implies a rather weak propagation. Note that \( \neg \text{regular} (x[1..i-1], DFA(\varphi_\rho)) \) is redundant, but it en-
ables a stronger propagation. The universal quantification is instead necessary. Imagine, e.g., \( \rho = aa \) and \( x = baaa \).
Without \( \forall j \in 1..i-1 : \neg \text{regular} (x[j..|x|], DFA(\varphi_\rho)) \), the solution \( (x = baaa, i = 3) \) would be sound: \( x[1..2] = ba \notin \mathcal{L}(aa*a) \) and \( x[3..4] = aa \in \mathcal{L}(aa*a) \). But it must be
match\( (baa,aa) = 2 \).

**PropMatch**

Let us now describe with pseudo-code our propagator. The function **PropMatch** described by Algorithm 1 propagates

\( i = \text{match}(x, \rho) \). It is called after the constraint is posted
and anytime the domain of \( x \) or \( i \) gets updated. It takes as input
the variables \( x \) and \( i \), the automata \( R = DFA(\varphi_\rho) \) and 
\( S = DFA(\varphi_\rho) \), and the minimal length \( r = \min \{ \rho \} \) of a
word of \( \mathcal{L}(\rho) \), computed with a breadth-first search over the
states of Dfa(\( \rho \)). It returns a pair \( (X, I) \) of possibly refined
domains for \( x \) and \( i \), i.e., \( \gamma(X) \subseteq \mathcal{D}(x) \) and \( I \subseteq \mathcal{D}(i) \).

At line 2 we initialize \( X \) and \( I \) with the domains of \( x \) and \( i \), respectively. At line 3 we possibly refine the upper bound of
\( I \), which cannot be bigger than the length of the longest string of
\( \gamma(X) \) minus the length of the shortest string accepted by
\( \rho \) plus one (we recall that, in our notation, the \( i \)-th leftmost
character of a string has index \( i \)). The function **Check** \( (A) \)
throws a failure if \( A = \emptyset \), otherwise \( A \) is returned. The ‘if’
statement at line 4 handles a particular case: if \( i \) is fixed and
its value is at most 1, then we can safely rewrite match into a single
regular constraint.

The while loop starting at line 12 calls **CheckBlock**
(explained below) on the blocks of \( X \) to get a pair \( (Q_i, j) \),
where \( Q_i \) is the set of states reachable after consuming blocks
\( X[1..i] \) and \( j \) is the offset within \( X[i] \) such that if
\( Q_i = \{ q_0 \} \) then \( \gamma(X[1..i]) \cap \mathcal{L}(\rho) = \emptyset \). Indeed, if

\[\\]
\(^\dagger\)The reified version of regular is \( b \Leftrightarrow \text{regular}(x, R) \) where \( b \) is
a Boolean variable. If \( b = \text{false} \), this is equivalent to \( \text{regular}(x, \overline{R}) \),
where \( \overline{R} \) is the complement of \( R \).
be no match.

A more general, and computationally expensive, way of refining the upper bound of \( I \) would be to reverse \( X \) and \( R \) to get \( X^{-1} = \prod_{i=1}^{n} X[i] \) and the non-deterministic automaton \( R^{-1} = (\Sigma, Q, \delta, q_0) \) having \( q_F \) as initial state and \( q_0 \) as the only final state. In this way, by running the propagator in reverse, we can find the rightmost position where \( \rho \) can match \( x \) and therefore an upper bound for \( I \). However, the benefits of this approach are overshadowed by its higher computational cost, and the fact that refining the upper bound of \( I \) is not that important as we are seeking the smallest index in \( M\hat{T}(x, \rho) \).

In line 24 from \((h, k)\) we get the leftmost position \( \ell = \sum_{i=1}^{n} l_i + k + 1 \) where a match can occur. Then, if we cannot refine \( X \) because \( 0 \in I \), we may still refine \( I \) by removing all the values in \( 1 \cdots \ell - 1 \) (line 26). If \( \ell < \min(I) \) we cannot refine \( I \) (line 27) but we can impose that no match occurs before \( \min(I) \) by considering the prefix region \( X[\ldots, (h, k)] \). The while loop at line 29 computes the rightmost position \( (h, k) \) such that \( |w| < \min(I) \) for each \( w \in \gamma(X[\ldots, (h, k)]) \). Then, we impose that no string of \( \gamma(X[\ldots, (h, k)]) \) belongs to \( L(R) \) by applying a regular filtering algorithm returning a possibly refined prefix \( X' \) (line 31). Here we deliberately allow flexibility about which regular algorithm to use, e.g., it may be the one defined in [Pesant, 2004], where \( D(x) \) is a fixed-size array of integer variables, or the one by [Amadini et al., 2018], where \( D(x) \) is a dashed string.

Similarly, at line 33 we may propagate regular over the suffix \( X'' \) with \( R = DFA(\star \rho \ast) \). This is sound, but only applied when \( I \) is not fixed (otherwise we can do better, as explained below) and \( \ell < \min(I) \), i.e., the lower bound of \( I \) is greater than the position of the leftmost possible match. In this case, it makes sense to look "to the right" of position \( \min(I) \) to basically check for inconsistencies. Indeed, this propagation will hardly result in a refinement of \( X[\ldots, (h, k)] \) because any character of \( \Sigma \) can occur in a string of \( L(R) \), which implies a likely weak propagation at the expense of a higher computational cost. This is why if \( \ell \geq \min(I) \) we do not apply this reasoning: we already know that \( \rho \) can match \( x \) at position \( \ell \). In this case we possibly refine \( I \) (line 35) and we only use \((h, k)\) as a pivot to split \( X \) into \( X' \) and \( X'' \) (line 36).

At line 38, we possibly perform a stronger suffix propagation by applying regular on \( L(S) = L(\rho \ast) \). But this is only sound when \( |I| = 1 \), i.e., \( i \) is fixed to a certain value. If instead \( |I| > 1 \), we cannot know at which position in \( I \) the match should happen, and therefore we cannot invoke regular on \( S \). For example, if \( X = \{a, b\} \cdot \{0, 5, 6\}, \rho = a \) and \( I = 3, 5 \) then the leftmost position for a match is \((1, 2) \). Propagating \( \ast\) regular \((X[\ldots, (1, 2)], \ast \star\ast) \ast\) fixes prefix \( X' \) to \( \{b\} \cdot \{2, 2\} \). However, propagating regular \((X[(1, 2), \ldots], \ast \star\ast) \ast\) sets suffix \( X'' \) to \( \{a\} \cdot \{1, 1\} \cdot \{a, b\} \cdot \{0, 2\} \). This refines \( X \) to \( \{b\} \cdot \{2, 2\} \cdot \{a\} \cdot \{1, 1\} \cdot \{a, b\} \cdot \{0, 2\} \), and so \( m \) must be 3, which is in general not sound as the leftmost occurrence of character \( a \) could also be at position 4 or 5 depending on the actual prefix of \( x \).

At line 39 we concatenate the blocks of \( X' \) and \( X'' \). If the last block of \( X' \) and the first block of \( X'' \) have the same base, or \( |0^{\ast} \ast| \) occurs in \( X' \) or \( X'' \), then a normalization is performed via the normalize function. Finally, we return the possibly refined domains.

### Algorithm 2 CHECKBLOCK function.

```plaintext
1: function CHECKBLOCK(S\textsuperscript{i}\textsuperscript{\textsubscript{in}}, Q\textsubscript{in},
   \( R = (\Sigma, Q, \delta, q_0, \{q_F\}) \))
2: \( \delta_S \leftarrow \{q \rightarrow \{\delta(q, a) | a \in S \} | q \in Q \} \)
3: \( Q_0 \leftarrow Q_{in}, j \leftarrow 0 \)
4: for \( i \in 1, 2, \ldots, l \) do \( \triangleright \) Mandatory region
5: \( Q_i \leftarrow \bigcup_{q \in Q_{in}, 0 < j} \delta_S(q) \)
6: if \( Q_S = \{q_0\} \) then \( \triangleright \) Fixpoint
7: \( j \leftarrow i \)
8: else if \( q_F \in Q_i \) then
9: \( \triangleright \) Final state reached
10: if \( Q_i = \{q_i\} \) then
11: \( \triangleright \) Fixpoint
12: \( \triangleright \) Final state reached
13: else
14: \( \triangleright \) Fixpoint
15: \( \text{dist} \leftarrow \begin{cases} q \rightarrow i & \text{if } q \in Q_i \\ +\infty & \text{if } q \in Q - Q_i \end{cases} \)
16: \( U \leftarrow \text{queue}(Q_i) \)
17: while \( U \neq \emptyset \) do \( \triangleright \) Breadth-first search.
18: \( q \leftarrow \text{POP}(U) \)
19: \( d \leftarrow \text{dist}[q] + 1 \)
20: if \( d \leq u \) then \( \triangleright \) Final state reached
21: \( \text{for } q' \in \delta_S(q) \text{ where } \text{dist}[q'] > d \text{ do} \)
22: \( \triangleright \) Final state reached
23: \( \text{dist}[q'] = d \)
24: if \( q_F \in \delta_S(q) \) then \( \triangleright \) Final state reached
25: \( \text{break} \)
26: \( \text{return } \{q \in Q | \text{dist}[q] \leq u, j\} \)
```

The function CHECKBLOCK described by 2 is similar to the forward pass of the regular propagator described in [Amadini et al., 2018], because it explores the sets \( Q_i \) of states reachable from \( Q_{in} \), after consuming \( i \) characters of block \( S\textsuperscript{i}\textsuperscript{\textsubscript{in}} \). However, here we are not interested in recording \( Q_0, \ldots, Q_{i+1} \) for each block because no backward pass is needed, while in [Pesant, 2004; Amadini et al., 2018] the domains' refinement is performed "backwards" starting from a feasible set of reachable states. Moreover, here we stop as soon as a candidate match is found, i.e., \( q_F \in Q_i \). Furthermore, CHECKBLOCK keeps track of the offset \( j \) within block \( S\textsuperscript{i}\textsuperscript{\textsubscript{in}} \), so that if \( S\textsuperscript{i}\textsuperscript{\textsubscript{in}} \) is the \( k \)-th block of dashed string \( X \) and \( (\_, j) \) is returned, then no match can occur in \( X[\ldots, (k, j)] \).

The function \( \delta_S : Q \rightarrow \mathcal{P}(Q) \) at line 2 returns the set of states reachable from a given state after consuming any of the characters in \( S \). We explore the reachable states \( Q_i \), for each of the \( l \) mandatory characters of input block \( S\textsuperscript{i}\textsuperscript{\textsubscript{in}} \), and we stop earlier when either the final state \( q_F \) (line 9) or a fixpoint \( Q_i \) (line 10) is reached. At each step, we update \( j \) with \( i \) if we find that \( Q_i = \{q_0\} \) (line 7), i.e., all the possible paths we get from a state of \( Q_{in} \) by consuming \( i \) characters of \( S \) lead to \( q_0 \). In this way, if \( S\textsuperscript{i}\textsuperscript{\textsubscript{in}} = X[k] \) and \( Q_{in} \) is the set of states reachable after consuming the prefix \( X[\ldots, (k, 0)] \), then by Lemma 2 no string of \( L(R) \) can match \( \gamma(X[\ldots, (k, j)]) \).

The second part of CHECKBLOCK, from line 15, performs a breadth-first search over the optional part \( S^{\ast} \) to possibly update the set of reachable states. If we reach the final
state, we interrupt the search (line 24). The returned set will be the input set of states of CHECKBLOCK when called by PROPMATCH on the following block. To be sound, we can only update the offset \( j \) when considering the mandatory part of the block because, by definition, the optional part may not be present.

**Example 1.** Consider \( \Sigma = \{a, b, c, d\} \), \( \rho = (ab)c \). \( X = \{d\}^{2.3} \{c\}^{0.5} \{a, b\}^{3.5} \) and \( I = 10 \ldots 10 \). Then, \( \text{maxlen}(X) = 3 + 5 + 5 = 13 \), \( R = \text{DFA}(\ast \rho \ast) \) as in Fig. 2, and \( r = \text{milen}(R) = 1 \). Hence, \( I \) is refined into 10..13. The call of CHECKBLOCK on \( X[1] \) returns \( \{(q_0), 2\} \). The call on \( X[2] \) returns \( \{(q_0, q_2), 0\} \). So, after the while loop of PROPMATCH we have \((h, k) = (1, 2)\) and at line 24 we get \( \ell = 3 \). Because \( \ell < \min(I) = 10 \) we update \((h, k)\) to \((3, 1)\) and propagate \( \neg \text{regular}(\{d\}^{2.3} \{c\}^{0.5} \{a, b\}^{3.1} \{R\}) \) to get \( X' = \{d\}^{2.3} \{a, b\}^{1.2} \). Because \( |I| > 1 \), we also propagate \( \text{regular}(X[3, 1], \ldots, R) \), but in this case the propagation has no effect so \( X'' \) will be \( X[3, 1], \ldots = \{a, b\}^{2.4} \).

We cannot further refine the suffix, so \( X \) becomes \( \text{normalize}(\{d\}^{2.3} \{a, b\}^{1.2}) = \{d\}^{2.3} \{a, b\}^{3.5} \).

Now, \( \text{milen}(X) = 8 \), hence \( \text{CHECK}(10..8 - 1 + 1) = \text{CHECK}(\emptyset) \) will throw a failure before returning.

**Example 2.** Let \( \Sigma \) and \( \rho \) as in Example 1, \( I = 0..10 \), and \( X = \{b, d\}^{2.5} \{a, c\}^{0.3} \). Then \( I \) is refined into 0..8. The call of CHECKBLOCK on \( X[1] \) returns \( \{(q_0), 2\} \) so \((h, k) = (1, 2)\). The call on \( X[2] \) returns \( \{(q_1, q_2), 0\} \) so \((h, k)\) is not updated. So again \( \ell = 3 \), but in this case we cannot refine \( X \) because 0 \( \notin I \). However, we can refine \( I \) into \( \{0\} \cup 3..8 \).

**Example 3.** Let \( \Sigma, \rho \) and \( X \) as in Example 2, and \( I = \{3\} \). As above, \((h, k) = (2, 1)\) and \( \ell = 3 \). But this time \( \ell < \min(I) \), so we do not update \((h, k)\) and we split \( X \) into \( X' = \{b, d\}^{2.2} \) and \( X'' = \{b, d\}^{0.3} \{a, c\}^{0.2} \). Finally, \( |I| = 1 \) so we can safely propagate \( \text{regular}(X[2, 1], \ldots, (ab)\ast) \) to refine \( X'' \) into \( \{c\}^{1.1} \{a, c\}^{0.2} \). Then, we update \( X \) to \( \{b, d\}^{2.2} \{c\}^{1.1} \{a, c\}^{0.2} \).

### 3.2 Complexity and Consistency

Let us analyze the worst case complexity of PROPMATCH. The cost of the first part of the algorithm (lines 1–26) is dominated by the while loop at line 12, which calls CHECKBLOCK at most \( n = |X| \) times. Each call to CHECKBLOCK considers each transition of \( \delta \) at most \( l + 1 \) times (one for each of the \( l \) mandatory characters of block \( S^I \cdots \), plus one for the BFS over the optional part of \( S^I \)). So, the worst case time complexity of CHECKBLOCK is \( O((l+1) \times |\delta|) \). If \( X = S^I_1 \cdots S^I_n \), the overall complexity of the first part of PROPMATCH is therefore \( O(\sum^n_{i=1} l_i \times |\delta|) \), which depends on the characters that must occur (the mandatory part of each block) but not on those that may occur (the optional part).

The complexity of the second part of PROPMATCH (from line 27) depends on the regular propagator adopted, which is called at most twice. It is however reasonable to assume that regular complexity is also bounded by \( O(\sum^n_{i=1} l_i \times |\delta|) \), as in [Jafar et al., 2018].

Unfortunately, we can say very little about the consistency notions ensured by the propagator. This is because, in general, dashed strings cannot guarantee a unique best representation of a set of strings (see the example in Sect. 2). Hence, it is infeasible to maintain consistency notions such as, e.g., the generalized arc-consistency.

### 4 Validation

We implemented our match propagator in G-STRINGS, a string solver based on CP solver GECODE [Gecode Team, 2023]. We compare its performance against the approach using decomposition (1), which we will call G-STRINGSdec.

We also include in the evaluation the state-of-the-art SMT solvers CVC5 [Barbosa et al., 2022] and Z3 [de Moura and Björner, 2008]. Although these solvers support the theory of strings, they do not natively support the match constraint, so we had to use some workarounds.

#### 4.1 Latin Square

We first compared the above approaches on a “stringy” version of the latin square problem: given \( n > 1 \), generate a \( n \times n \) latin square over alphabet \( \{a_1, \ldots, a_n\} \) using string variables only. We use 2\( n \) string variables: \( r_1, \ldots, r_n \) for the rows and \( c_1, \ldots, c_n \) for the columns, and we impose for \( k = 1, \ldots, n \) that:

- \(|r_k| = |c_k| = n \) and \( r_k, c_k \in \{a_1| \cdots |a_n\}^* \)
- \( \text{match}(r_i, a_k) > 0 \) for \( i = 1, \ldots, n \)
- \( \text{match}(c_j, a_k) > 0 \) for \( j = 1, \ldots, n \)
- \( \text{match}(r_i, a_k) = j \iff \text{match}(c_j, a_k) = i \) for \( i, j = 1, \ldots, n \)

We encoded the problem in MiniZinc [Nethercote et al., 2007] for G-STRINGS and G-STRINGSdec, while for SMT solvers we used the SMT-LIB [Barrett et al., 2010] language. For SMT solvers, we replaced each match constraint with a corresponding \( \text{str.indexof} \) function, returning the first occurrence of a pattern string in a target string (we can soundly use \( \text{str.indexof} \) because in this case the patterns are always fixed strings \( a_1, \ldots, a_k \) of length 1).

Fig. 3 shows the runtimes in seconds for \( n = 2, 3, \ldots, 26 \). Z3 is not included because it could not find any solution. CVC5 can only solve small size instances \( n \leq 8 \). As \( n \) gets bigger the performance of G-STRINGSdec clearly deteriorates: for \( n \geq 20 \) it cannot solve the problem within the

\(^{1}\)We used CVC5 1.0.2 and Z3 4.11.2 with Z3str3 string solver (we also tried Z3seq, which did not improve Z3’s performance). We run all the experiments on a Ubuntu 22.04 machine with 8 GB of RAM and 1.60 GHz Intel i5 CPU, with a timeout of 300s. We will disclose the source code in case of acceptance of the paper.
The second problem we consider is the shortest matching superstring problem. Given patterns $p_1, \ldots, p_n$, the goal is to find the shortest string matched by all patterns, i.e., a minimal-length string $x$ such that $\text{match}(x, p_i) > 0$ for $i = 1, \ldots, n$. For the SMT solvers, instead of match we used the regular expression membership function $\text{str.in.re}$ applied to each wrapped pattern $*p_i*$ for $i = 1, \ldots, n$.

We generated 16 sets $S_{l,n}$ of patterns for $n \in \{32, 64, 128, 256\}$ and $l \in \{5, 10, 15, 20\}$. Each $S_{l,n}$ contains $n$ random patterns such that there exists a string of length $l$ matched by all patterns (but there could be a shorter one). So, $l$ is an upper bound on the length of the shortest string matched by all patterns. We use this information to provide an initial bound for each problem.

The results are shown in Tab. 1, showing for each solver the percentage of problems solved to optimality (opt), where a solution was found (sat), no solution was found (unk). Then it reports the average time to first solution (ttf) when one is found, and solve time (which equals the time limit for unknown cases). Last two columns are comparative scores: score gives 0 for finding no solution, 0.25 for finding the worst known solution, 0.75 for finding the best known solution, a linear scale value in (0.25, 0.75) for finding other solutions, and 1 for proving the optimal solution. borda is the MiniZinc Challenge score [Stuckey et al., 2014], which gives a Borda score where each pair of solvers is compared on each instance, the better solver gets 1, the weaker 0, and if they are tied the point is split inversely proportional to their solving time.

SMT solvers perform better on this problem, but G-STRINGS still achieves the best results. It is the quickest to find solutions and the most robust overall, always finding a solution and proving optimality more often than other approaches. The score shows that its solutions are significantly better than other approaches, while borda shows that CVC5 is effectively dominated by the other approaches. It is important to note the performance difference between G-STRINGS and G-STRINGSdec, which is not in Tab. 1 because it could not find any solution. This witnesses the scalability issues of the decomposition approach, and hence the importance of having a specialized match propagator.

5 Conclusions

We introduce a propagation algorithm for the constraint $\text{match}(x, \rho)$, returning the position in $x$ of the leftmost possible match of a string belonging to the language denoted by the regular pattern $\rho$. This generalizes existing propagators and decision procedures for the regular constraint, which only enforce the membership of a string variable to the language denoted by a regular pattern. We provided empirical evidence demonstrating the benefits of a specialized propagator for match, in contrast to a less efficient rewriting of this constraint in terms of regular.

Possible future directions include the extension of the replace constraint where the string to be replaced is defined by a regular pattern instead of a string variable, and other harder constraints such as, e.g., finding and replacing all the occurrences or considering non-regular patterns such as back-references.

Acknowledgments

Research partly funded by PNRR - M4C2 - Investimento 1.3, Partenariato Esteso PE00000013 - “FAIR – Future Artificial Intelligence Research” - Spoke 1 “Human-centered AI”, funded by the European Commission under the NextGeneration EU programme. This research was partially funded by the Australian Government through the Australian Research Council Industrial Transformation Training Centre in Optimisation Technologies, Integrated Methodologies, and Applications (OPTIMA), Project ID IC200100009.

References


