A Regular Matching Constraint for String Variables

Roberto Amadini[1] , **Peter J. Stuckey**[2,3]
[1]University of Bologna, Bologna, Italy
[2]Monash University, Melbourne, Victoria, Australia
[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 $\mathsf{regular}(x, R)$ which only considers the *membership* of x to the regular language $\mathcal{L}(R)$ 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 $\mathsf{match}(x, \rho)$ returning the *position* in x of the leftmost possible match of a string of $\mathcal{L}(\rho)$. 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 Σ and a maximum string length λ, 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 λ 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 Σ be a finite, non-empty alphabet and Σ^* the set of all its strings. A string $w \in \Sigma^*$ is a finite sequence of $|w| \geq 0$ characters of Σ, where $|w|$ is the length of w (the empty string is denoted with ϵ). 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 $w^n = ww^{n-1}$ 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 VW) 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 $l..u = \{x \in \mathbb{Z} \mid l \leq x \leq u\}$ and denote with $w[i..j]$ the substring of w from character i to character j, i.e., $w[i]w[i+1]\cdots w[j]$ (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 $\mathcal{L}(R)$ accepted by a *finite-state automaton* R, the language $\mathcal{L}(r)$ denoted by *regular expression* r, or the language $\mathcal{L}(\Gamma)$ generated by regular grammar Γ.[1] We call *regular pattern* (shortly, pattern) a formalism ρ describing a regular language $\mathcal{L}(\rho)$.

A *deterministic* finite-state automaton (DFA) is a tuple $R = (\Sigma, Q, \delta, q_0, F)$ where: Σ 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 $s_0 \to \cdots \to s_n$ where $n = |w|$ and $s_0, \ldots, s_n \in Q$ such that $s_i = \delta(s_{i-1}, w[i])$ for $i = 1, \ldots, n$. We denote with $\widehat{\delta} : Q \times \Sigma^* \to Q$ the extension of δ to strings such that $\widehat{\delta}(q, w) = q'$ if and only if $q \to \cdots \to q'$ is a computation for w. A string $w \in \Sigma^*$ is accepted by R if $\widehat{\delta}(q_0, w) \in F$. The language of R is $\mathcal{L}(R) = \{w \in \Sigma^* \mid \widehat{\delta}(q_0, w) \in F\}$.

If δ is a total function, the DFA is *complete*. If δ is partial, we can get a complete DFA by extending Q to $Q' = Q \cup \{q_\perp\}$, and δ to $\delta \cup \{(q, c, q_\perp) \mid q \in Q', c \in \Sigma, \delta(q, c) = \perp\}$. The complement automaton $\overline{R}$ having $L(\overline{R}) = \Sigma^* \setminus L(R)$ is easily computed from a complete DFA R by complementing its final states. If there does not exist a DFA R' with less than $|Q|$ states such that $\mathcal{L}(R') = \mathcal{L}(R)$, then R is *minimal*. Given a regular pattern ρ, we denote by $\mathsf{DFA}(\rho)$ the minimal and complete DFA such that $\mathcal{L}(\rho) = \mathcal{L}(\mathsf{DFA}(\rho))$.

If $R = (Q, \Sigma, \delta, q_0, F)$ is complete, a state $q \in F$ is said *universally accepting* if *all* computations from q bring to a state $q' \in F$. Dually, $q \in Q \setminus F$ is *universally rejecting* if all computations from q bring to a $q' \in Q \setminus F$. 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 $\mathsf{minlen}(R)$ 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 $\mathcal{L}(r)$ denoted by each $r \in \mathcal{RE}$, as: *(i)* $\varnothing \in \mathcal{RE}$, denoting $L(\varnothing) = \emptyset$; *(ii)* if $c \in \Sigma \cup \{\epsilon\}$, then $c \in \mathcal{RE}$ denoting $\mathcal{L}(c) = \{c\}$; *(iii)* if $r, r' \in \mathcal{RE}$, then $r \cdot r' \in \mathcal{RE}$ denoting $\mathcal{L}(r \cdot r') = \mathcal{L}(r)\mathcal{L}(r')$, and $r|r' \in \mathcal{RE}$ denoting $\mathcal{L}(r|r') = \mathcal{L}(r) \cup \mathcal{L}(r')$; *(iv)* if $r \in \mathcal{RE}$ then $r^* \in \mathcal{RE}$ denoting $\mathcal{L}(r^*) = \mathcal{L}(r)^*$.

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 C defined over variables $x_1, \ldots, x_k$ is an algorithm that aims to prune the most inconsistent values from $\mathcal{D}(x_1), \ldots, \mathcal{D}(x_k)$ according to the semantics of 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 Σ 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 $D(x_i) \subseteq \Sigma^*$ 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; Tateishi *et al.*, 2013], word equations [Berzish *et al.*, 2017; Barbosa *et al.*, 2022], unfolding (using either bit-vector solvers [Kieżun *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* $S_1^{l_1,u_1} S_2^{l_2,u_2} \cdots S_k^{l_k,u_k}$, where $S_i \subseteq \Sigma$ and $0 \leq l_i \leq u_i \leq \lambda$ for $i = 1, \ldots, k$ and $\Sigma_{i=1}^k l_i \leq \lambda$. For each block $S^{l,u}$, we call S its base and l, u its lower/upper bound, respectively. We denote with $\mathsf{maxlen}(X)$ the sum of the upper bounds of each block of X, with $X[i]$ the i-th block 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 $\emptyset^{0,0}$ occurs only to denote the empty string.

Let $\gamma(S^{l,u}) = \{x \in S^* \mid l \leq |x| \leq u\}$ be the language denoted by block $S^{l,u}$.[2] We extend γ to dashed strings: $\gamma(S_1^{l_1,u_1} \cdots S_k^{l_k,u_k}) = (\gamma(S_1^{l_1,u_1}) \cdots \gamma(S_k^{l_k,u_k})) \cap \mathbb{S}$, where the intersection with $\mathbb{S} = \{w \in \Sigma^* \mid |w| \leq \lambda\}$ excludes the strings with length greater than λ. Normalisation entails that each string $w \in \mathbb{S}$ has a unique dashed string X such that $w = \gamma(X)$.

Given dashed strings X and Y we define the relation $X \sqsubseteq Y \Leftrightarrow \gamma(X) \subseteq \gamma(Y)$ 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 $\{\mathtt{ab}, \mathtt{ba}\} \subseteq \Sigma^*$: three maximally accurate but incomparable dashed strings that represent this set are $\{a, b\}^{2,2}$, $\{a\}^{0,1}\{b\}^{1,1}\{a\}^{0,1}$ and $\{b\}^{0,1}\{a\}^{1,1}\{b\}^{0,1}$, each represents a different over approximation. This means that workarounds have to be used to preserve the soundness

[1] We assume that these formalism are familiar to the reader.

[2] $\gamma(S^{l,u})$ is the set of strings having length in $l..u$ and characters in S. We use the notation $\gamma(X)$ instead of $\mathcal{L}(X)$ to be consistent with the original dashed string definition [Amadini *et al.*, 2020].

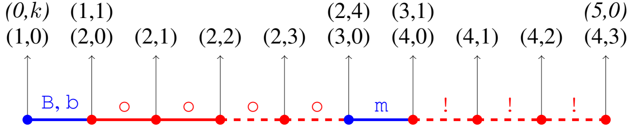

Figure 1: Graphical representation of the dashed string $\{\texttt{B,b}\}^{1,1}\{\texttt{o}\}^{2,4}\{\texttt{m}\}^{1,1}\{\texttt{!}\}^{0,3}$ and its positions.

of propagation.

From a graphical point of view, we can see each block $S_i^{l_i,u_i}$ as a solid segment of length l_i followed by a dashed segment of length $u_i - l_i$. The solid segment indicates that exactly l_i characters of S_i *must* occur in each string of $\gamma(S_1^{l_1,u_1} \cdots S_k^{l_k,u_k})$, and defines the *mandatory part* S^{l_i,l_i}. The dashed segment indicates that n characters of S_i, with $0 \leq n \leq u_i - l_i$, *may* occur and defines the *optional part* S_i^{0,u_i-l_i}. Consider, e.g., the graphical representation of dashed string $X = \{\texttt{B,b}\}^{1,1}\{\texttt{o}\}^{2,4}\{\texttt{m}\}^{1,1}\{\texttt{!}\}^{0,3}$ in Fig. 1. Each string of $\gamma(X)$ starts with B or b, followed by 2 to 4 os, one m, then 0 to 3 !s.

A convenient way to refer a dashed string is through its *positions*. Given $X = S_1^{l_1,u_1} \cdots S_k^{l_k,u_k}$, we can define $1 + \sum_{i=1}^k u_i$ positions (i,j) where index $i \in \{1,\ldots,k\}$ refers to block $X[i]$ and offset j indicates how many characters from the beginning of $X[i]$ we are considering. To better represent the beginning (or the end) of a block, offsets are 0-based. In particular, $(i,0)$ refers to the beginning of block i, and can be equivalently identified with $(i-1, u_{i-1})$, i.e., the end of block $i-1$ (see Fig. 1). For convenience, we also consider $(k+1,0)$ to be equivalent to the position (k,u_k), and $(0,k)$ for any k to be equivalent to position $(1,0)$, the beginning of the dashed string.

Given $X = S_1^{l_1,u_1} \cdots S_k^{l_k,u_k}$ and positions $(i,j),(i',j')$, we denote with $X[(i,j),(i',j')]$ the *region* of X between (i,j) and (i',j'), i.e., $X[(i,j),(i',j')]$ is equal to:

$$\begin{cases} \emptyset^{0,0} & \text{if } (i,j) \succeq (i',j') \\ S_i^{\max(0,l_i-j),j'-j} & \text{else if } i = i' \\ S_i^{\max(0,l_i-j),u_i-j} S_{i+1}^{l_{i+1},u_{i+1}} \cdots S_{i'-1}^{l_{i'-1},u_{i'-1}} S_{i'}^{\min(l_{i'},j'),j'} & \text{otherwise} \end{cases}$$

where $\preceq$ is the lexicographic ordering between pairs.

For example, with X as in Fig. 1, $X[(2,1),(4,2)] = \{\texttt{o}\}^{1,3}\{\texttt{m}\}^{1,1}\{\texttt{!}\}^{0,2}$. For brevity, we indicate with $X[P,...]$ the "suffix region" $X[P,(k,u_k)]$ between a position P and the last position (k,u_k) of X, and with $X[...,P]$ the "prefix region" between $(0,0)$ and P. For example, with X as in Fig. 1 and $P = (2,3)$ we have $X[...,P] = \{\texttt{B,b}\}^{1,1}\{\texttt{o}\}^{2,3}$ and $X[P,...] = \{\texttt{o}\}^{0,1}\{\texttt{m}\}^{1,1}\{\texttt{!}\}^{0,3}$.

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

$$\begin{cases} \{0\} & \text{if } \nexists i,j.\ w[i..j] \in \mathcal{L}(\rho) \\ \{1\} & \text{else if } w = \epsilon \text{ and } \epsilon \in \mathcal{L}(\rho) \\ \{i \in 1..|w| \mid \exists j.\ w[i..j] \in \mathcal{L}(\rho)\} & \text{otherwise} \end{cases}$$

In practice, $\mathcal{MI}(w,\rho)$ is the set of all the positions in w where ρ *matches* a substring w' of w, i.e., $w' \in \mathcal{L}(\rho)$. For example, if $\Sigma = \{\texttt{a},\texttt{b},\texttt{c}\}$, $\rho = \texttt{ab(a|c)}^*$ and $w = \texttt{cabbcab}$ then $\mathcal{MI}(w,\rho) = \{2,6\}$. If instead $w = \texttt{caacb}$, 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 $\mathcal{MI}(w,\rho) = 1..|w|$ because $w[i..i-1] = \epsilon \in L(\rho)$ for $i = 1,\ldots,|w|$. In this way, we capture the fact that ϵ occurs at each position of a non-empty string, being the empty string substring of every string.

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

Lemma 1. *Let ρ be a regular pattern with $\mathcal{L}(\rho) \neq \emptyset$, $R = \mathsf{DFA}(\rho)$ and $\widehat{R} = \mathsf{DFA}(\star\rho\star)$. The following properties hold:*

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

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

Proof. (i): Suppose $\widehat{R}$ has a universally rejecting state $q_\perp$. $\widehat{R}$ is minimal, so there is a $\widehat{R}$-computation $q_0 \to \cdots \to q_\perp$ for a certain $w_0 \in \Sigma^*$. Hence, for each $w \in \Sigma^*$ we would have $w_0 w \notin \mathcal{L}(\widehat{R})$. In particular, if $w \in \mathcal{L}(R)$ we would have $w_0 w \notin \mathcal{L}(\widehat{R}) = \mathcal{L}(\star R\star) \supseteq \mathcal{L}(\star R)$ which is impossible being $w_0 w \in \mathcal{L}(\star R) = \Sigma^*\mathcal{L}(R)$.

(ii): Because $\mathcal{L}(\rho) \neq \emptyset$, there is at least a final state. If $\widehat{R}$ has no universally accepting states, there exist two states $q_i \in F, q_j \notin F$ and a character $a \in \Sigma$ such that $\delta(q_i,a) = q_j$. So there is a $w \in \mathcal{L}(\widehat{R})$ such that $wa \notin \mathcal{L}(\widehat{R})$ which is impossible being $\mathcal{L}(\widehat{R}) = \Sigma^*\mathcal{L}(R)\Sigma^*$ closed under concatenation. So, $\widehat{R}$ has at least a universally accepted state. Because it is minimal, it must have at most one universally accepting state. So, $\widehat{R}$ has exactly one universally accepting state. □

Lemma 2. *Let ρ be a regular pattern with $\epsilon \notin \mathcal{L}(\rho)$ and $R = \mathsf{DFA}(\star\rho\star) = (\Sigma,Q,\delta,q_0,F)$. For each $w \in \Sigma^*$, if $\hat{\delta}(q_0,w) = q_0$ then $w \notin \mathcal{L}(\rho)$.*

Proof. Let $w \in \Sigma^*$ such that $\hat{\delta}(q_0,w) = q_0$ and suppose $w \in \mathcal{L}(\rho)$. Because $\mathcal{L}(\rho) \subseteq \mathcal{L}(\star\rho\star)$, then $w \in \mathcal{L}(\star\rho\star) = \mathcal{L}(R)$ so it must be $q_0 \in F$ and thus $\epsilon \in \mathcal{L}(\star\rho\star)$. Because by hypothesis $\epsilon \notin \mathcal{L}(\rho)$, it must be $\epsilon \in \mathcal{L}(\star\rho\star) \setminus \mathcal{L}(\rho)$ which is impossible. □

3.1 Propagating match

Let us now explain how we propagate $i = \mathsf{match}(x,\rho)$, 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 $\mathcal{D}(x)$ is a fixed sequence of integer variables

as in [Pesant, 2004; Scott *et al.*, 2017]. In fact, the sequence $x_1 \cdots x_n$ where $\mathcal{D}(x_i) \subseteq \Sigma$ ($x_i = j$ iff $x[i]$ is the j-th symbol of Σ) corresponds to dashed string $\mathcal{D}(x_1)^{1,1} \cdots \mathcal{D}(x_n)^{1,1}$.

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

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

This rewriting, especially when i is not fixed, is inefficient, as it involves a universal quantification over $1..i-1$ and many *reifications*.[1] This implies a rather weak propagation. Note that $\neg\mathsf{regular}(x[1..i-1], \mathsf{DFA}(\star\rho\star))$ is redundant, but it enables a stronger propagation. The universal quantification is instead necessary. Imagine, e.g., $\rho = \mathtt{aa}$ and $x = \mathtt{baaa}$. Without $\forall j \in 1..i-1 : \neg\mathsf{regular}(x[j..|x|], \mathsf{DFA}(\rho\star))$, the solution ($x = \mathtt{baaa}, i = 3$) would be sound: $x[1..2] = \mathtt{ba} \notin \mathcal{L}(\star\mathtt{aa}\star)$ and $x[3..4] = \mathtt{aa} \in \mathcal{L}(\mathtt{aa}\star)$. But it must be $\mathsf{match}(\mathtt{baaa}, \mathtt{aa}) = 2$.

PROPMATCH

Let us now describe with pseudo-code our propagator. The function PROPMATCH described by Algorithm 1 propagates $i = \mathsf{match}(x, \rho)$. 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 $R = \mathsf{DFA}(\star\rho\star)$ and $S = \mathsf{DFA}(\rho\star)$, and the minimal length $r = \mathsf{minlen}(\rho)$ of a word of $\mathcal{L}(\rho)$, computed with a breadth-first search over the states of $\mathsf{DFA}(\rho)$. 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 ρ 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 $X[1] \cdots X[i]$, and j is the offset within $X[i]$ such that if $Q_i = \{q_0\}$ then $\gamma(X[..., (i,j)]) \cap \mathcal{L}(\rho) = \emptyset$. Indeed, if

[1] The reified version of regular is $b \Leftrightarrow \mathsf{regular}(x, R)$ where b is a Boolean variable. If $b = \mathit{false}$, this is equivalent to $\mathsf{regular}(x, \overline{R})$, where $\overline{R}$ is the complement of R.

Algorithm 1 PROPMATCH function.

1: **function** PROPMATCH(x, i, $R = (\Sigma, Q, \delta, q_0, \{q_F\})$, S, r)
2: $\quad X \leftarrow \mathcal{D}(x), \quad I \leftarrow \mathcal{D}(i)$
3: $\quad I \leftarrow I \cap$ CHECK $(0..\mathsf{maxlen}(X) - r + 1)$
4: $\quad$ **if** $I = \{k\} \wedge k \leq 1$ **then** $\quad \triangleright\ i = 0 \vee i = 1$
5: $\quad\quad$ **if** $k = 0$ **then**
6: $\quad\quad\quad$ REWRITE($\neg\mathsf{regular}(x, R)$) $\quad \triangleright\ x \notin \mathcal{L}(R)$
7: $\quad\quad$ **else**
8: $\quad\quad\quad$ REWRITE($\mathsf{regular}(x, S)$) $\quad \triangleright\ x \in \mathcal{L}(S)$
9: $\quad\quad$ **return** X, I
10: $\quad i \leftarrow h \leftarrow k \leftarrow 0, \quad n \leftarrow |X|$
11: $\quad Q_0 \leftarrow \{q_0\}$
12: $\quad$ **while** $i < n \wedge q_F \notin Q_i$ **do**
13: $\quad\quad i \leftarrow i + 1$
14: $\quad\quad (Q_i, j) \leftarrow$ CHECKBLOCK($X[i], Q_{i-1}, R$)
15: $\quad\quad$ **if** $Q_i = \{q_0\}$ **then**
16: $\quad\quad\quad (h, k) \leftarrow (i, j)$
17: $\quad$ **if** $q_F \notin Q_i$ **then** $\quad \triangleright$ No match
18: $\quad\quad I \leftarrow$ CHECK $(I \cap \{0\})$
19: $\quad\quad$ **return** X, I
20: $\quad$ **if** $Q_i = \{q_F\}$ **then** $\quad \triangleright$ Surely a match
21: $\quad\quad I \leftarrow$ CHECK $(I \cap 1..\max(I))$
22: $\quad\quad$ **if** $i < n$ **then**
23: $\quad\quad\quad I \leftarrow$ CHECK $\left(I \cap 1..\sum_{j=1}^{i} u_j - r + 1\right)$
24: $\quad \ell \leftarrow \sum_{j=1}^{h-1} l_j + k + 1$
25: $\quad$ **if** $0 \in I$ **then**
26: $\quad\quad$ **return** $X, \{0\} \cup (I \cap \ell..\max(I))$
27: $\quad$ **if** $\ell < \min(I)$ **then**
28: $\quad\quad h \leftarrow 1, k \leftarrow \min(I) - 1$ $\quad \triangleright$ Update (h, k)
29: $\quad\quad$ **while** $h \leq n \wedge k \geq u_h$ **do**
30: $\quad\quad\quad (h, k) \leftarrow (h + 1, k - u_h)$
31: $\quad\quad X' \leftarrow$ PROPAGATE $(\neg\mathsf{regular}(X[..., (h,k)], R))$
32: $\quad\quad$ **if** $|I| > 1$ **then**
33: $\quad\quad\quad X'' \leftarrow$ PROPAGATE $(\mathsf{regular}(X[(h,k), ...], R))$
34: $\quad$ **else**
35: $\quad\quad I \leftarrow$ CHECK $(I \cap \ell..\max(I))$
36: $\quad\quad X' \leftarrow X[..., (h,k)], X'' \leftarrow X[(h,k), ...]$
37: $\quad$ **if** $|I| = 1$ **then**
38: $\quad\quad X'' \leftarrow$ PROPAGATE $(\mathsf{regular}(X[(h,k), ...], S))$
39: $\quad X \leftarrow$ NORMALIZE($X' \cdot X''$)
40: $\quad$ **return** $X, I \cap$ CHECK $(1..\mathsf{maxlen}(X) - r + 1)$

$Q_i = \{q_0\}$ and (Q_i, j) is returned by CHECKBLOCK, then $\widehat{\delta}(q_0, w) = q_0$ for each $w \in \gamma(X[..., (i,j)])$. By Lemma 2, this means that no $w \in \gamma(X[..., (i,j)])$ can be in $\mathcal{L}(\rho)$. In other terms, if we feed R with any string $w \in \gamma(X[..., (i,j)])$ we always go back to the initial state. Position (h, k) is meant to be the *leftmost* position where ρ can match X, as anytime we find that $Q_i = \{q_0\}$ at line 16 we update (h, k) with (i, j).

We exit the while loop if all the X-blocks are consumed, or the unique and universally accepting state q_F (see Lemma 1) is reached. If q_F is never reached, it must be $\mathsf{match}(x, \rho) = 0$ (line 17). If instead $Q_i = \{q_F\}$, then surely $\mathsf{match}(x, \rho) \geq 1$ (line 20). In this case, we may also refine the upper bound of I because we know that $w \in \mathcal{L}(\rho)$ for each $w \in \gamma(X[..., (i, u_i)])$, hence the match cannot happen after position $\sum_{j=1}^{i} u_j - r + 1$ (line 23). Note that if q_F is not the only reachable state we cannot refine the bounds of I: there may

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 $X^{-1} = X[n] \cdots X[1]$ and the non-deterministic automaton $R^{-1} = (\Sigma, Q, \{q_F\}, \{(q', c, q) \mid (q, c, q') \in \delta\}, \{q_0\})$ 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 ρ 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 $\mathcal{MI}(x, \rho)$.

In line 24 from (h, k) we get the leftmost position $\ell = \sum_{j=1}^{h-1} l_j + k + 1$ 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 $1..\ell - 1$ (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 $X[..., (h, k)]$. The while loop at line 29 computes the rightmost position (h, k) such that $|w| < \min(I)$ for each $w \in \gamma(X[..., (h, k)])$. Then, we impose that no string of $\gamma(X[..., (h, k)])$ belongs to $\mathcal{L}(R)$ 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 $\mathcal{D}(x)$ is a fixed-size array of integer variables, or the one by [Amadini *et al.*, 2018], where $\mathcal{D}(x)$ is a dashed string.

Similarly, at line 33 we may propagate regular over the suffix X'' with $R = \mathsf{DFA}(\star\rho\star)$. 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 $X[(h, k), ...]$ because any character of Σ can occur in a string of $\mathcal{L}(R)$, 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 ρ can match x at position ℓ. 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 $\mathcal{L}(S) = \mathcal{L}(\rho\star)$. 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 $X = \{\mathtt{a}, \mathtt{b}\}^{0,5}$, $\rho = \mathtt{a}$ and $I = 3..5$ then the leftmost position for a match is $(1, 2)$. Propagating $\neg\mathsf{regular}(X[..., (1, 2)], \star\mathtt{a}\star)$ fixes prefix X' to $\{\mathtt{b}\}^{2,2}$. However, propagating $\mathsf{regular}(X[(1, 2), ...], \mathtt{a}\star)$ sets suffix X'' to $\{\mathtt{a}\}^{1,1}\{\mathtt{a}, \mathtt{b}\}^{0,2}$. This refines X to $\{\mathtt{b}\}^{2,2}\{\mathtt{a}\}^{1,1}\{\mathtt{a}, \mathtt{b}\}^{0,2}$, and so i 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 $\emptyset^{0,0}$ 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.

```
 1: function CHECKBLOCK(S^{l,u}, Q_in,
    R = (Σ, Q, δ, q_0, {q_F}))
 2:     δ_S ← {q ↦ {δ(q, a) | a ∈ S} | q ∈ Q}
 3:     Q_0 ← Q_in,   j ← 0
 4:     for i ∈ 1, 2, ..., l do                    ▷ Mandatory region
 5:         Q_i ← ⋃_{q∈Q_{i-1}} δ_S(q)
 6:         if Q_i = {q_0} then
 7:             j ← i                              ▷ All paths lead to q_0
 8:         else if q_F ∈ Q_i then
 9:             return (Q_i, j)                    ▷ Final state reached
10:         if Q_i = Q_{i-1} then
11:             if Q_i = {q_0} then                ▷ Fixpoint
12:                 return (Q_i, l)
13:             else
14:                 return (Q_i, j)
15:     dist ← {q ↦ l       if q ∈ Q_l
                     +∞      if q ∈ Q − Q_l }
16:     U ← QUEUE(Q_l)            ▷ U = queue of all the states in Q_l
17:     while U ≠ [ ] do
18:         q ← POP(U)                             ▷ Breadth-first search.
19:         d ← dist[q] + 1
20:         if d ≤ u then
21:             for q' ∈ δ_S(q) where dist[q'] > d do
22:                 PUSH(U, q')
23:                 dist[q'] = d
24:             if q_F ∈ δ_S(q) then               ▷ Final state reached
25:                 break
26:     return ({q ∈ Q | dist[q] ≤ u}, j)
```

CHECKBLOCK

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 $S^{l,u}$. However, here we are not interested in recording $Q_0, \ldots, Q_{l+1}$ 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 $S^{l,u}$ so that if $S^{l,u}$ is the k-th block of dashed string X and $(_, j)$ is returned, then no match can occur in $X[..., (k, j)]$.

The function $\delta_S : Q \to \mathcal{P}(Q)$ 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 $S^{l,u}$, 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 $S^{l,u} = X[k]$ and Q_{in} is the set of states reachable after consuming the prefix $X[..., (k, 0)]$, then by Lemma 2 no string of $\mathcal{L}(R)$ can match $\gamma(X[..., (k, j)])$.

The second part of CHECKBLOCK, from line 15, performs a breadth-first search over the optional part $S^{0,u-l}$ to possibly update the set of reachable states. If we reach the final

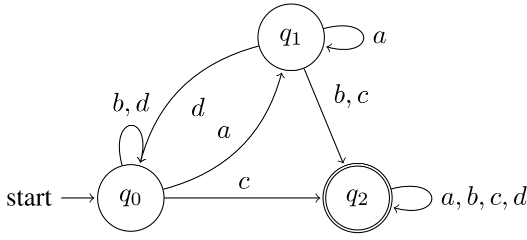

Figure 2: DFA of $\star(ab|c)\star$

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* $\Sigma = \{\mathsf{a},\mathsf{b},\mathsf{c},\mathsf{d}\}$, $\rho = (\mathsf{ab}|\mathsf{c})$, $X = \{\mathsf{d}\}^{2,3}\{\mathsf{c}\}^{0,5}\{\mathsf{a},\mathsf{b}\}^{3,5}$ *and* $I = 10..100$. *Then,* $\mathsf{maxlen}(X) = 3+5+5 = 13$, $R = \mathsf{DFA}(\star\rho\star)$ *as in Fig. 2, and* $r = \mathsf{minlen}(R) = 1$. *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* $\neg\mathsf{regular}(\{\mathsf{d}\}^{2,3}\{\mathsf{c}\}^{0,5}\{\mathsf{a},\mathsf{b}\}^{1,1}, R)$ *to get* $X' = \{\mathsf{d}\}^{2,3}\{\mathsf{a},\mathsf{b}\}^{1,1}$. *Because* $|I| > 1$, *we also propagate* $\mathsf{regular}(X[(3,1),...], R)$, *but in this case the propagation has no effect so* X'' *will be* $X[(3,1),...] = \{\mathsf{a},\mathsf{b}\}^{2,4}$.

We cannot further refine the suffix, so X *becomes* NORMALIZE$(\{\mathsf{d}\}^{2,3}\{\mathsf{a},\mathsf{b}\}^{1,1}\{\mathsf{a},\mathsf{b}\}^{2,4}) = \{\mathsf{d}\}^{2,3}\{\mathsf{a},\mathsf{b}\}^{3,5}$. *Now,* $\mathsf{maxlen}(X) = 8$, *hence* CHECK $(10..8-1+1)$ = CHECK $(\emptyset)$ *will throw a failure before returning.*

Example 2. *Let* Σ *and* ρ *as in Example 1,* $I = 0..10$, *and* $X = \{\mathsf{b},\mathsf{d}\}^{2,5}\{\mathsf{a},\mathsf{c}\}^{0,3}$. *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* $0 \in I$. *However, we can refine* I *into* $\{0\} \cup 3..8$.

Example 3. *Let* Σ, ρ *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* $X' = \{\mathsf{b},\mathsf{d}\}^{2,2}$ *and* $X'' = \{\mathsf{b},\mathsf{d}\}^{0,3}\{\mathsf{a},\mathsf{c}\}^{0,3}$. *Finally,* $|I| = 1$ *so we can safely propagate* $\mathsf{regular}(X[(2,1),...], (\mathsf{ab}|\mathsf{c})\star)$ *to refine* X'' *into* $\{c\}^{1,1}\{a,c\}^{0,2}$. *Then, we update* X *to* $\{\mathsf{b},\mathsf{d}\}^{2,2}\{\mathsf{c}\}^{1,1}\{\mathsf{a},\mathsf{c}\}^{0,2}$.

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 δ at most $l+1$ times (one for each of the l mandatory characters of block $S^{l,u}$, plus one for the BFS over the optional part of $S^{l,u}$). So, the worst case time complexity of CHECKBLOCK is $O((l+1)\times|\delta|)$. If $X = S_1^{l_1,u_1} \cdots S_n^{l_n,u_n}$ the overall complexity of the first part of PROPMATCH is therefore $O(\sum_{i=1}^n l_i \times |\delta|)$, 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 $O(\sum_{i=1}^n l_i \times |\delta|)$, as in [Amadini *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-STRINGS$_{dec}$. We also include in the evaluation the state-of-the-art SMT solvers CVC5 [Barbosa *et al.*, 2022] and Z3 [de Moura and Bjørner, 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 $\{a_1, \dots, a_n\}$ using string variables only. We use $2n$ string variables: $r_1, \dots, r_n$ for the rows and $c_1, \dots, c_n$ for the columns, and we impose for $k = 1, \dots, n$ that:

- $|r_k| = |c_k| = n$ and $r_k, c_k \in \mathcal{L}(a_1|\dots|a_n)^*$
- $\mathsf{match}(r_i, a_k) > 0$ for $i = 1, \dots, n$
- $\mathsf{match}(c_j, a_k) > 0$ for $j = 1, \dots, n$
- $\mathsf{match}(r_i, a_k) = j \Leftrightarrow \mathsf{match}(c_j, a_k) = i$ for $i, j = 1, \dots, n$

We encoded the problem in MiniZinc [Nethercote *et al.*, 2007] for G-STRINGS and G-STRINGS$_{dec}$, 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 `str.indexof` function, returning the first occurrence of a pattern string in a target string (we can soundly use `str.indexof` because in this case the patterns are always fixed strings $a_1, \dots, a_k$ of length 1).[1]

Fig. 3 shows the runtimes in seconds for $n = 2, 3, \dots, 26$. 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-STRINGS$_{dec}$ 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.

solver	opt	sat	unk	ttf	time	score	borda
CVC5	6.25	87.5	12.5	38.44	292.5	5.25	0.51
Z3	37.5	50.0	50.0	174.0	196.94	7.5	14.52
G-STRINGS	**43.75**	**100.0**	**0.0**	**4.56**	**172.19**	**13.75**	**26.16**

Table 1: Results for the shortest matching superstring problem. Best results in bold font.

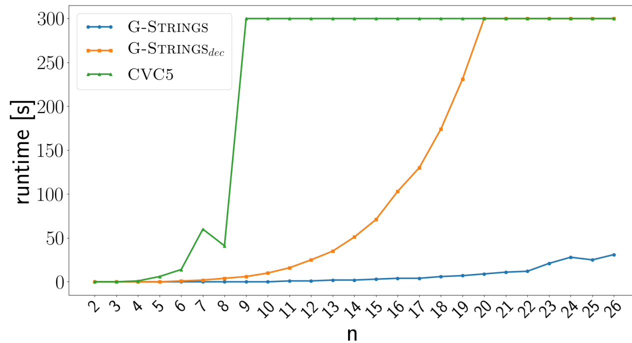

Figure 3: Latin square runtimes for $n = 2, 3, \ldots, 26$.

300s timeout (note that we are not charging for the flattening time into FlatZinc). Conversely, G-STRINGS equipped with match propagator can solve all the problems within 32s.

4.2 Shortest Matching Superstring

The second problem we consider is the *shortest matching superstring*: given patterns $\rho_1, \ldots, \rho_n$, the goal is to find the shortest string matched by all patterns, i.e., a minimal-length string x such that match$(x, \rho_i) > 0$ for $i = 1, \ldots, n$. For the SMT solvers, instead of match we used the regular expression membership function `str.in_re` applied to each wrapped pattern $\star\rho_i\star$ 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 effectibely dominated by the other approaches. It is important to note the performance difference between G-STRINGS with G-STRINGS$_{dec}$, 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 match(x, ρ), returning the position in x of the leftmost possible match of a string belonging to the language denoted by the regular pattern ρ. 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

[Amadini *et al.*, 2018] Roberto Amadini, Graeme Gange, and Peter J. Stuckey. Propagating regular membership with dashed strings. In J. Hooker, editor, *Proc. 24th Conf. Principles and Practice of Constraint Programming*, volume 11008 of *LNCS*, pages 13–29. Springer, 2018.

[Amadini *et al.*, 2020] Roberto Amadini, Graeme Gange, and Peter J. Stuckey. Dashed strings for string constraint solving. *Artif. Intell.*, 289:103368, 2020.

[Amadini, 2021] Roberto Amadini. A survey on string constraint solving. *ACM Comput. Surv.*, 55(1), nov 2021.

[Barbosa *et al.*, 2022] Haniel Barbosa, Clark W. Barrett, Martin Brain, Gereon Kremer, Hanna Lachnitt, Makai Mann, Abdalrhman Mohamed, Mudathir Mohamed, Aina Niemetz, Andres Nötzli, Alex Ozdemir, Mathias Preiner, Andrew Reynolds, Ying Sheng, Cesare Tinelli, and Yoni Zohar. cvc5: A versatile and industrial-strength SMT solver. In Dana Fisman and Grigore Rosu, editors, *Tools and Algorithms for the Construction and Analysis of Systems - 28th International Conference, TACAS 2022, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2022, Munich, Germany, April 2-7, 2022, Proceedings, Part I*, volume 13243 of *Lecture Notes in Computer Science*, pages 415–442. Springer, 2022.

[Barrett *et al.*, 2010] Clark Barrett, Aaron Stump, and Cesare Tinelli. The SMT-LIB Standard: Version 2.0. In A. Gupta and D. Kroening, editors, *Proceedings of the 8th International Workshop on Satisfiability Modulo Theories (Edinburgh, UK)*, 2010.

[Berzish *et al.*, 2017] Murphy Berzish, Vijay Ganesh, and Yunhui Zheng. Z3str3: A string solver with theory-aware heuristics. In D. Stewart and G. Weissenbacher, editors, *Proc. 17th Conf. Formal Methods in Computer-Aided Design*, pages 55–59. FMCAD Inc, 2017.

[Bultan *et al.*, 2017] Tevfik Bultan, Fang Yu, Muath Alkhalaf, and Abdulbaki Aydin. *String Analysis for Software Verification and Security*. Springer, 2017.

[de Moura and Bjørner, 2008] Leonardo Mendonça de Moura and Nikolaj Bjørner. Z3: an efficient SMT solver. In *Tools and Algorithms for the Construction and Analysis of Systems, 14th International Conference, TACAS 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29-April 6, 2008. Proceedings*, pages 337–340, 2008.

[Gecode Team, 2023] Gecode Team. Gecode: Generic constraint development environment. http://www.gecode.org, 2023. Accessed: 2023-29-05.

[Kieżun *et al.*, 2012] Adam Kieżun, Vijay Ganesh, Shay Artzi, Philip J. Guo, Pieter Hooimeijer, and Michael D. Ernst. HAMPI: A solver for word equations over strings, regular expressions, and context-free grammars. *ACM Trans. Software Engineering and Methodology*, 21(4):article 25, 2012.

[Li and Ghosh, 2013] Guodong Li and Indradeep Ghosh. PASS: String solving with parameterized array and interval automaton. In V. Bertacco and A. Legay, editors, *Proc. 9th Int. Haifa Verification Conf.*, volume 8244 of *LNCS*, pages 15–31. Springer, 2013.

[Nethercote *et al.*, 2007] Nicholas Nethercote, Peter J. Stuckey, Ralph Becket, Sebastian Brand, Gregory J. Duck, and Guido Tack. MiniZinc: Towards a standard CP modelling language. In *Proceedings of the 13th International Conference on Principles and Practice of Constraint Programming*, volume 4741 of *LNCS*, pages 529–543. Springer, 2007.

[Pesant, 2004] G. Pesant. A regular language membership constraint for finite sequences of variables. In M. Wallace, editor, *Proceedings of the 10th International Conference on Principles and Practice of Constraint Programming*, volume 3258 of *LNCS*, pages 482–495. Springer-Verlag, 2004.

[Rossi *et al.*, 2006] F. Rossi, P. van Beek, and T. Walsh, editors. *Handbook of Constraint Programming*. Elsevier, 2006.

[Saxena *et al.*, 2010] Prateek Saxena, Devdatta Akhawe, Steve Hanna, Feng Mao, Stephen McCamant, and Dawn Song. A symbolic execution framework for JavaScript. In *Proc. 2010 IEEE Symp. Security and Privacy*, pages 513–528. IEEE Comp. Soc., 2010.

[Scott *et al.*, 2017] Joseph D. Scott, Pierre Flener, Justin Pearson, and Christian Schulte. Design and implementation of bounded-length sequence variables. In M. Lombardi and D. Salvagnin, editors, *Proc. 14th Int. Conf. Integration of Artificial Intelligence and Operations Research Techniques in Constraint Programming*, volume 10335 of *LNCS*, pages 51–67. Springer, 2017.

[Stuckey *et al.*, 2014] Peter J. Stuckey, Thibaut Feydy, Andreas Schutt, Guido Tack, and Julien Fischer. The minizinc challenge 2008-2013. *AI Magazine*, (2):55–60, 2014.

[Tateishi *et al.*, 2013] Takaaki Tateishi, Marco Pistoia, and Omer Tripp. Path- and index-sensitive string analysis based on monadic second-order logic. *ACM Trans. Software Engineering Methodology*, 22(4):article 33, 2013.