Dashed Strings for String Constraint Solving

Roberto Amadini[a], Graeme Gange[b], Peter J. Stuckey[b]

[a] *University of Bologna, Bologna, Italy*
[b] *Monash University, Melbourne, Victoria, Australia*

Abstract

String processing is ubiquitous across computer science, and arguably more so in web programming — where it is also a critical part of security issues such as injection attacks. In recent years, a number of string solvers have been developed to solve combinatorial problems involving string variables and constraints. We examine the dashed string approach to string constraint solving, which represents an unknown string as a sequence of blocks of characters with bounds on their cardinalities. The solving approach relies on propagation of information about the blocks of characters that arise from reasoning about the constraints in which they occur. This approach shows promising performance on many benchmarks involving constraints like string length, equality, concatenation, and regular expression membership. In this paper, we formally review the definition, the properties and the use of dashed strings for string constraint solving, and we provide an empirical validation that confirms the effectiveness of this approach.

Keywords: Artificial Intelligence, Constraint Programming, String Solving

1. Introduction

String constraint solving (in short *string solving*) is an emerging important field, given the ubiquity of strings in different domains such as test-case generation [1], program analysis [2], model checking [3], web security [4], and bioinformatics [5]. The string solving approaches that have been proposed so far are mainly based on: *automata* [6, 7, 8], *word-equations* [9, 10] and *unfolding* (using either bit-vector solvers [11, 12] or Constraint Programming (CP) [13]).

Automata and word-equations handle *unbounded-length* strings, but have limitations when combining string and integer constraints, and suffer from scalability issues: automata growth in the first case, disjunctive case-splitting in the second. Unfolding approaches are based on *bounded-length* strings: they fix a length bound λ and then map each string variable to a vector of λ elements (either by compiling down to integer/bit-vector constraints [14, 11, 12] or using dedicated propagators [13]). This approach adds flexibility but sacrifices high-level relationships between strings, and can become very expensive when the maximum string length λ is large — even if generated solutions are very short.

A recent CP approach introduced the *dashed string* abstraction [15, 16, 17, 18]. This approach can be seen as a *lazy* unfolding over bounded-length strings, where the goal is to mitigate the dependency on the length bound λ. Informally, a dashed string consists of a concatenation of distinct set of strings called *blocks*. The core idea is to encode in a compact way potentially very long strings, and very big sets of strings, by discriminating between the (number of) characters that *may* appear in a string, and those that *must* appear. Ideally we aim to find, if there exists, the dashed string that better abstracts a given set of concrete strings. The empirical evaluations conducted in [15, 16, 17, 18] demonstrate the effectiveness of this approach, which has been implemented in the G-Strings [19] string solver.

In this paper, we provide a comprehensive review of the dashed strings approach for string constraint solving. In particular, we:

- formalise the model and the theoretical properties of dashed strings, by correcting and refining the notions first introduced in [15];
- explain in much more depth, and formally prove correctness of, the Equate algorithm first introduced in [16], which constitutes the basis of most propagators for dashed string solvers;
- provide an overview of the propagators and the branchers we devised for string solving, and the tools we implemented based on dashed strings;

- report a new empirical evaluation where we compare the dashed string approach against state-of-the-art string solvers on the recently introduced *StringFuzz* string benchmarks [20].

Empirical results confirm the effectiveness of the dashed string approach, often able to significantly outperform other state-of-the-art string solvers.

Paper structure. In Section 2 we give preliminary notions. In Section 3 we formalise the dashed string model. Section 4 thoroughly describes the equation algorithm for dashed string. In Section 5 we give an overview of the propagators and branchers we implemented, while Section 6 covers the dashed string implementation. In Section 7 we show the empirical evaluation we performed, before reporting the related works in Section 8 and concluding in Section 9.

2. Preliminaries

Let us fix a finite alphabet $\Sigma = \{a_1, \dots, a_m\}$ of $m > 1$ symbols. We denote with Σ^* the set of all the strings over Σ. 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 use the 1-based array notation to lookup the symbols in a string: $w[i]$ is the i-th symbol of string w, with $1 \leq i \leq |w|$.

The concatenation of $v, w \in \Sigma^*$ is denoted by $v \cdot w$ (or simply with vw) while w^n is the iterated concatenation (i.e., $w^0 = \epsilon$ and $w^n = ww^{n-1}$ for $n > 0$). Analogously, we define the concatenation between sets of strings: given $V, W \subseteq \Sigma^*$, we denote with $V \cdot W = \{vw \mid v \in V, w \in W\}$ (or simply with VW) their concatenation and with W^n the iterated concatenation (where $W^0 = \{\epsilon\}$).

This work is focused on *bounded-length* strings. We fix an upper bound $\lambda \in \mathbb{N}$ on the maximum string length, and we consider only strings $w \in \Sigma^*$ such that $|w| \leq \lambda$. We define $\mathbb{S}_\Sigma^\lambda = \bigcup_{i=0}^{\lambda} \Sigma^i$. Clearly $\mathbb{S}_\Sigma^\lambda$ is not closed under concatenation but is finite: $|\mathbb{S}_\Sigma^\lambda| = \sum_{i=0}^{\lambda} n^i = \dfrac{n^{\lambda+1} - 1}{n - 1}$, where $n = |\Sigma|$.

We extend the canonical definition of *Constraint Satisfaction Problem* (CSP) by including string variables and constraints. Formally, a CSP is a triple $\langle \mathcal{X}, \mathcal{D}, \mathcal{C} \rangle$

where: $\mathcal{X} = \{x_1, \ldots, x_n\}$ are the *variables*; $\mathcal{D} = \{D(x_1), \ldots, D(x_n)\}$ are the *domains*, where for $i = 1, \ldots, n$ each $D(x_i)$ is a (super-)set of values that x_i can take; $\mathcal{C} = \{C_1, \ldots, C_m\}$ are the *constraints* over the variables of $\mathcal{X}$. The goal is to find a solution, i.e., an assignment $\xi \in D(x_1) \times \cdots \times D(x_n)$ of domain values to the corresponding variables that satisfies all of the constraints of $\mathcal{C}$. To do so, we have to define suitable *propagators* for each constraint.

A propagator for a constraint $C \in \mathcal{C}$ is basically a function that —according to the semantics of C— tries to prune the domain of the variables involved in C, until it becomes *subsumed* (i.e., $\mathcal{D}$ implies C), a domain becomes empty (the CSP is unsatisfiable) or it reaches a *fixpoint* (no domain can be further pruned). The level of pruning is a compromise between the efficacy (how much we are able to prune) and the efficiency of the propagator. Most of the propagators are not complete (a fixpoint is reached even if the domains could in principle be further pruned) and a *branching* strategy is often used to search for a solution.

Virtually all the CSPs referred in the literature have *finite domains*, i.e., the cardinality of each set of $\mathcal{D}$ is finite. This guarantees the decidability of these problems—that are in general NP-complete—by enumeration. Typically, only CSPs with integer variables are considered but the literature also presents some variants (e.g., CSPs involving sets or floating point variables [21]).

In this work, in addition to constraints over integers, we also consider constraints over bounded-length strings. Given finite alphabet Σ and maximum string length λ, a CSP with bounded-length strings contains a number $k > 0$ of string variables $\{x'_1, \ldots, x'_k\} \subseteq \mathcal{X}$ such that $D(x'_i) \subseteq \Sigma^*$ and $|x'_i| \leq \lambda$ for $i = 1, \ldots, k$. The set $\mathcal{C}$ may include well-known string constraints, e.g., string length, (dis-)equality, membership in a regular language, lexicographic ordering, concatenation, substring selection, and finding/replacing. We will refer to the constraint solving involving string variables as *string (constraint) solving*.

3. Dashed Strings

Dashed strings are special cases of regular expressions. In this work, because we limit the string length by λ (and because Σ is finite), we only consider dashed strings denoting a finite set of strings. However, the following definitions can be easily generalised to unbounded-length strings where $\lambda = +\infty$. Dashed strings are inspired by the *Bricks* string abstract domain defined in [22].

The rationale behind dashed strings is to facilitate a compact and flexible representation for set of strings having unknown (yet bounded) length, without statically pre-allocating an arbitrary large number of elements for each potential character of the string. Dashed strings divide similarly behaved regions of a partially specified string into a sequence of concatenated blocks. Each block discriminates between the number of characters that *must* appear in a string and those that *may* appear. Let us now formalise dashed strings and their properties.

3.1. Formal model

Definition 1. A *dashed string* X of size k is a concatenation of $k > 0$ blocks $X = S_1^{l_1,u_1} S_2^{l_2,u_2} \cdots S_k^{l_k,u_k}$ such that, for $i = 1, \dots, k$:

(i) $S_i \subseteq \Sigma$

(ii) $0 \leq l_i \leq u_i \leq \lambda$

(iii) $u_i + \sum_{j \neq i} l_j \leq \lambda$

For each block $S_i^{l_i,u_i}$ we call S_i the *base* and (l_i, u_i) the *cardinality.* In particular, $l_i = lb(S_i^{l_i,u_i})$ is the *lower bound* while $u_i = ub(S_i^{l_i,u_i})$ is the *upper bound* (on the cardinality) of a block.

Condition *(i)* constrains the characters of the base to be in Σ, while *(ii)* ensures the consistency of the lower and upper bounds. Condition *(iii)* restricts the cardinality values: the sum of an upper bound and all the other lower bounds cannot exceed λ. The i-th block of a dashed string X will be denoted by $X[i]$, and the size of X by $|X|$. We denote by $\mathbb{DS}_\Sigma^\lambda$ the set of all the dashed strings. We do not distinguish blocks and dashed strings of unary size.

As previously mentioned, the primary goal of a dashed string $X \in \mathbb{DS}_\Sigma^\lambda$ is to compactly represent a set of strings $W \subseteq \mathbb{S}_\Sigma^\lambda$. We therefore define a "concretisation"[1] operator γ such that $\gamma(S^{l,u}) = \{w \in S^* \mid l \leq |w| \leq u\} \subseteq \mathbb{S}_\Sigma^\lambda$ is the *language denoted* by block $S^{l,u}$, i.e., the set of all the strings of $\mathbb{S}_\Sigma^\lambda$ having characters in S and length in $[l, u]$. In particular, the *null block* denotes the empty string only: $\gamma(\emptyset^{0,0}) = \{\epsilon\}$. With a little abuse of notation, we then extend γ to handle dashed strings: $\gamma(S_1^{l_1,u_1} \cdots S_k^{l_k,u_k}) = (\gamma(S_1^{l_1,u_1}) \dots \gamma(S_k^{l_k,u_k})) \cap \mathbb{S}_\Sigma^\lambda$ (the intersection with $\mathbb{S}_\Sigma^\lambda$ is necessary to limit the size of the denoted strings).

Note that, while $\mathbb{S}_\Sigma^\lambda$ is finite, $\mathbb{DS}_\Sigma^\lambda$ is *countable*: λ bounds the cardinalities (l_i, u_i) of the blocks, but not the number k of blocks that may appear in a dashed string. In fact, for each $X \in \mathbb{DS}_\Sigma^\lambda$ of size k we can get a dashed string $X' \in \mathbb{DS}_\Sigma^\lambda$ of size $k + 1$ by appending to X a block of the form $S^{0,u}$ with $S \subseteq \Sigma$ and $u \leq \lambda - \sum_{i=1}^k l_i$. This is because if a block $S^{0,u}$ has lower bound 0 then $\epsilon \in \gamma(S^{0,u})$. Blocks of this form will be called *nullable*. A dashed string X is said to be *known* if $|\gamma(X)| = 1$, i.e., it represents a single, "concrete" string of $\mathbb{S}_\Sigma^\lambda$. There does not exist a dashed string X such that $\gamma(X) = \emptyset$.

Note that we could make $\mathbb{DS}_\Sigma^\lambda$ finite by setting a limit on the upper bounds, i.e., $\sum_{i=1}^k u_i \leq \lambda$. However, this can lead to a precision loss. Consider for example $X = \{a\}^{0,1}\{b\}^{\lambda-1,\lambda-1}\{a\}^{0,1}$. This denotes the set of strings $\gamma(X) = \{ab^{\lambda-1}, b^{\lambda-1}, b^{\lambda-1}a\}$. By forcing $\sum_{i=1}^k u_i \leq \lambda$ we would have $X \notin \mathbb{DS}_\Sigma^\lambda$: an over-approximation X' of X with $\gamma(X') \supset \gamma(X)$ would be required.

Given a block $S^{l,u}$, we define its *dimension* $\|S^{l,u}\|$ as:

$$\|S^{l,u}\| = \begin{cases} u - l + 1 & \text{if } |S| \leq 1 \\ \dfrac{|S|^{u+1} - |S|^l}{|S| - 1} & \text{otherwise} \end{cases}$$

and we generalise this definition to dashed strings: $\|X\| = \Pi_{i=1}^{|X|} \|X[i]\|$ for each $X \in \mathbb{DS}_\Sigma^\lambda$. The dimension of a dashed string gives a measure of the number of concrete strings it represents. For a block $S^{l,u}$ we have that $\|S^{l,u}\| = |\gamma(S^{l,u})|$, while for a dashed string $X \in \mathbb{DS}_\Sigma^\lambda$ we have $\|X\| \geq |\gamma(X)|$ but not $\|X\| = |\gamma(X)|$.

[1] Note the analogy with the concretisation function γ for Abstract interpretation [23].

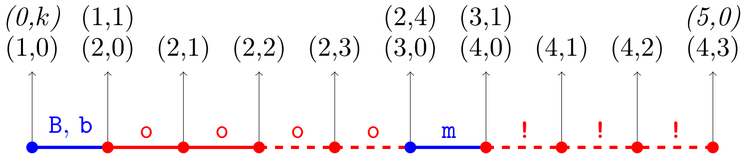

Figure 1: Graphical representation of $X = \{\texttt{B,b}\}^{1,1}\{\texttt{o}\}^{2,4}\{\texttt{m}\}^{1,1}\{\texttt{!}\}^{0,3}$ and corresponding offset positions.

For example, if $X = \{a\}^{0,1}\{a\}^{0,1}$, then $|\gamma(X)| = |\{\epsilon, a, aa\}| = 3$ and $\|X\| = \|\{a\}^{0,1}\| \cdot \|\{a\}^{0,1}\| = 4$.

Given the potentially huge dimension of a dashed string, it can be convenient to consider the *logarithm* of its dimension: $\log \|X\| = \sum_{i=1}^{|X|} \log \|X[i]\|$. To avoid overflows, for each block $S^{l,u}$ such that $n = |S| > 1$, we compute it as:

$$\begin{aligned}\log \|S^{l,u}\| = \log \frac{n^{u+1} - n^l}{n-1} &= \log \frac{n^l(n^{u-l+1} - 1)}{n-1} \\ &= \log(n^l(n^{u-l+1} - 1)) - \log(n-1) \\ &= l \cdot \log n + \log(n^{u-l+1} - 1) - \log(n-1).\end{aligned}$$

3.1.1. Graphical interpretation

We now give an informal intuition of what a dashed string is. The name "dashed" comes from a graphical interpretation of $X = S_1^{l_1,u_1} S_2^{l_2,u_2} \cdots S_k^{l_k,u_k}$ where we imagine a block $S_i^{l_i,u_i}$ as a continuous segment of length l_i followed by a dashed segment of length $u_i - l_i$. The continuous segment indicates that exactly l_i characters of S_i *must* occur in each concrete string of $\gamma(X)$; the dashed segment indicates that k characters of S_i, with $0 \leq k \leq u_i - l_i$, *may* occur. Consider dashed string $X = \{\texttt{B,b}\}^{1,1}\{\texttt{o}\}^{2,4}\{\texttt{m}\}^{1,1}\{\texttt{!}\}^{0,3}$ in Figure 1. Each string of $\gamma(X)$ must start with '`B`' or '`b`', followed by 2 to 4 '`o`'s, one '`m`', and 0 to 3 '`!`'s.

A convenient way to refer a dashed string is through its *(offset) positions*. Given $X = S_1^{l_1,u_1} S_2^{l_2,u_2} \cdots S_k^{l_k,u_k}$, 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 Figure 1).

We also add extra notation for convenience. We define $(k+1, 0)$ to be equivalent to the position (k, u_k), that is the end of the dashed string. Similarly we define $(0, k)$ for any k to be equivalent to position $(1, 0)$, i.e., the beginning of the dashed string.[1]

We indicate with $\preceq$ the total, lexicographic order over equivalent positions: $(i,j) \prec (i',j') \iff i < i' \vee (i = i' \wedge j < j')$, and $(i,j) \preceq (i',j')$ iff $(i,j) \prec (i',j')$ or (i,j) and (i',j') are equivalent. Position (i,j) of dashed string X will be denoted by $X[i,j]$. Note that $X[i,j]$ does not represent a character but a position in X *in between* characters. Given $X = S_1^{l_1,u_1} S_2^{l_2,u_2} \cdots S_k^{l_k,u_k}$ and positions $(i,j), (i',j')$, we indicate with $X[(i,j),(i',j')]$ the *region* of X between (i,j) and (i',j'), i.e., $X[(i,j),(i',j')] =$

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

Each block $X[i]$ has a *mandatory part*, defined by $X[(i,0),(i,l_i)]$, and an *optional part*, defined by $X[(i,l_i),(i,u_i)]$. For brevity, we indicate with $X[P, ...]$ the region $X[P, (k, u_k)]$ between a position P and the last position (k, u_k) of X.

For example, in Figure 1 the region $X[(2,1),(2,4)]$ corresponds to $\{\texttt{o}\}^{1,3}$ while $X[(2,3),(4,2)] = \{\texttt{o}\}^{0,1}\{\texttt{m}\}^{1,1}\{\texttt{!}\}^{0,2}$. The mandatory part of block $\{\texttt{o}\}^{2,4}$ is $\{\texttt{o}\}^{2,2} = X[(2,0),(2,2)]$ while its optional part is $\{\texttt{o}\}^{0,2} = X[(2,2),(2,4)]$. Region $X[(2,2), ...]$ is $\{\texttt{o}\}^{0,2}\{\texttt{m}\}^{1,1}\{\texttt{!}\}^{0,3}$.

3.2. Normalisation

It is easy to note that $\gamma : \mathbb{DS}_\Sigma^\lambda \to \mathcal{P}(\mathbb{S}_\Sigma^\lambda)$ is not injective: there may exist two distinct dashed strings X, Y such that $\gamma(X) = \gamma(Y)$. In other terms, the same

[1] For convenience, one could also use negative offsets where position $(i, -j)$ refers to j characters *from the end* of block i (counting from left to right), as done in [16].

concrete string $w \in \mathbb{S}_{\Sigma}^{\lambda}$ can be denoted by different dashed strings of $\mathbb{DS}_{\Sigma}^{\lambda}$. For example, there are infinite ways of denoting the empty string ϵ by concatenating an arbitrary number of null blocks. To remove this unpleasant redundancy, and minimise the dashed string size, we introduce a notion of *normalisation*.

Definition 2. A dashed string $X = S_1^{l_1,u_1} S_2^{l_2,u_2} \cdots S_k^{l_k,u_k}$ is said *normalised* if and only if:

(i) $S_i \neq S_{i+1}$, for $i = 1, \dots, k-1$

(ii) $S_i = \emptyset \iff l_i = u_i = 0$, for $i = 1, \dots, k$

(iii) $X = \emptyset^{0,0} \vee S_i \neq \emptyset$, for $i = 1, \dots, k$

We define $\textsc{norm} : \mathbb{DS}_{\Sigma}^{\lambda} \to \mathbb{DS}_{\Sigma}^{\lambda}$ such that $\textsc{norm}(X)$ is the normalisation of X. We denote the set of normalised dashed strings as $\overline{\mathbb{DS}}_{\Sigma}^{\lambda} = \{\textsc{norm}(X) \mid X \in \mathbb{DS}_{\Sigma}^{\lambda}\}$.

Condition *(i)* forces each adjacent base to be distinct, since $\gamma(S^{l,u}\, S^{l',u'}) = \gamma(S^{l+l',u+u'})$. Condition *(ii)* avoids multiple configurations for $\emptyset^{0,0}$, while *(iii)* forbids its redundant use, since in general $\gamma(X\,\emptyset^{0,0}) = \gamma(\emptyset^{0,0}\,X) = \gamma(X)$. We omit the algorithm for $\textsc{norm}$, that unsurprisingly has linear cost $O(|X|)$ for normalising a dashed string X. From now on, if not further specified, we will always refer to normalised dashed strings.

Normalisation is fundamental to remove a large number of spurious dashed strings from $\mathbb{DS}_{\Sigma}^{\lambda}$. However, some issues still occur. For example, $\overline{\mathbb{DS}}_{\Sigma}^{\lambda}$ remains countable: e.g., given two distinct characters $a, b \in \Sigma$ we can generate an infinite sequence of dashed strings in $\overline{\mathbb{DS}}_{\Sigma}^{\lambda}$ by alternating blocks $\{a\}^{0,1}$ and $\{b\}^{0,1}$:

$$\{a\}^{0,1}, \quad \{a\}^{0,1}\{b\}^{0,1}, \quad \{a\}^{0,1}\{b\}^{0,1}\{a\}^{0,1}, \quad \{a\}^{0,1}\{b\}^{0,1}\{a\}^{0,1}\{b\}^{0,1}, \quad \dots$$

The dimension $\|X\|$ of a normalised dashed string X is still less precise than $|\gamma(X)|$, for example if $X = \{a\}^{0,1}\{a,b\}^{0,1}$ then $\|X\| = 6 > |\gamma(X)| = |\{\epsilon, a, b, aa, ab\}| = 5$. Most importantly, as shown in Proposition 1*(i)*, γ restricted to $\overline{\mathbb{DS}}_{\Sigma}^{\lambda}$ is *still not injective*. The reason is the intersection with $\mathbb{S}_{\Sigma}^{\lambda}$, needed to limit the maximum string length. However, as proven in Proposition 1*(ii)*, normalisation makes γ injective for most cases — precisely when the sum of the upper bounds does not exceed λ.

Proposition 1. The following properties hold for γ:

(i) γ is not injective: $\gamma(X) = \gamma(Y)$ does not always implies $X = Y$

(ii) If $X, Y \in \overline{\mathbb{DS}}_{\Sigma}^{\lambda}$ are such that $\sum_{i=1}^{|X|} ub(X[i]) \leq \lambda$ and $\sum_{i=1}^{|Y|} ub(Y[i]) \leq \lambda$, then $\gamma(X) = \gamma(Y)$ implies $X = Y$.

Proof. See Appendix.

As a corollary of Proposition 1 it is easy to see that for normalised unbounded-length strings, where $\lambda = +\infty$, the γ function is always injective.

3.3. The $\sqsubseteq$ relation

Let us define a binary relation $\sqsubseteq$ on $\overline{\mathbb{DS}}_{\Sigma}^{\lambda}$ such that, for each $X, Y \in \overline{\mathbb{DS}}_{\Sigma}^{\lambda}$,

$$X \sqsubseteq Y \iff (X = Y \vee \gamma(X) \subset \gamma(Y)).^{1}$$

Intuitively, $\sqsubseteq$ models the relation *"is more precise than or equal to"* between dashed strings. This property is important because ideally we aim to find, if there exists, the dashed string that better abstracts a given set of concrete strings. It is easy to see that the algebraic structure $(\overline{\mathbb{DS}}_{\Sigma}^{\lambda}, \sqsubseteq)$ is a partially ordered set (*poset*). A nice property is that $(\overline{\mathbb{DS}}_{\Sigma}^{\lambda}, \sqsubseteq)$ is *well-founded*, as proven in Theorem 1. A less welcome property is that $(\overline{\mathbb{DS}}_{\Sigma}^{\lambda}, \sqsubseteq)$ is not a lattice, as proven in Theorem 2.

Theorem 1. The poset $(\overline{\mathbb{DS}}_{\Sigma}^{\lambda}, \sqsubseteq)$ has no infinite descending chains.

Proof. See Appendix.

[1] We do not define $\sqsubseteq$ simply as $X \sqsubseteq Y \iff \gamma(X) \subseteq \gamma(Y)$ because otherwise $\sqsubseteq$ would be a *pre-order* but not a partial order, e.g., if $X = \{a, b, c\}^{0,2}$, and $Y = \{a, b\}^{0,2}\{b, c\}^{0,2}\{a, c\}^{0,2}$ are such that $\gamma(X) \subseteq \gamma(Y)$ and $\gamma(Y) \subseteq \gamma(X)$ so $X \sqsubseteq Y$ and $Y \sqsubseteq X$ but $X \neq Y$.

Theorem 2. Let Σ be a finite alphabet with at least two symbols and $\lambda \in \mathbb{N} \cup \{+\infty\}$ a maximum string length. The poset $(\overline{\mathbb{DS}}_{\Sigma}^{\lambda}, \sqsubseteq)$ is *not a lattice.*

Proof. See Appendix.

Theorem 2 shows a weakness of dashed strings: in general, we might not be able to determine the dashed string which best represents two or more concrete strings. This holds even for unbounded-length strings: the proof of Theorem 2 does not make any assumption on the upper bound λ. We can say that the nature of general dashed strings is particularly suitable to handle very common string operations such as string length, equality, concatenation and related operations. However, dashed strings might lose precision when performing disjunctive reasoning of the form $x = w \lor x = w'$ if $w \neq w' \land \max(|w|, |w'|) > 1$.

Note that, unlike other frameworks (e.g., Abstract Interpretation [23]), Constraint Programming does not require a lattice structures to preserve the soundness of constraint solving. However, as we shall see, some workarounds are necessary to avoid leaking feasible solutions or causing infinite propagations.

4. Dashed String Equation

In this section we introduce a fundamental operation on $\overline{\mathbb{DS}}_{\Sigma}^{\lambda}$: the *equation* between dashed strings. As we shall see in Section 5, equating dashed strings is crucial for propagating equality between string variables, as well as related constraints like reified equality, disequality, domain, (iterated) concatenation, reverse, substring, and so on.

We can see the equation between dashed string X and Y as a semantic unification where we want to find a *refinement* of X and Y including *all* the strings of $\gamma(X) \cap \gamma(Y)$ and removing *the most* values not belonging to $\gamma(X) \cap \gamma(Y)$. In other terms, we are looking for a minimal —or at least small enough— dashed string denoting all the concrete strings of $\gamma(X)$ and $\gamma(Y)$. This operation is a surrogate of the meet operation on $\overline{\mathbb{DS}}_{\Sigma}^{\lambda}$, which cannot be defined since, as seen in Theorem 2*(ii)*, $(\overline{\mathbb{DS}}_{\Sigma}^{\lambda}, \sqsubseteq)$ is not a meet-semilattice. Clearly, this only makes sense for *equatable* dashed strings such that $\gamma(X) \cap \gamma(Y) \neq \emptyset$. Determining if two dashed strings are equatable is always decidable since γ always returns a finite set of strings. Let us now formalise the concept of dashed string equation.

Definition 3. Given two pairs of dashed strings $(X, Y), (X', Y') \in \overline{\mathbb{DS}}_\Sigma^\lambda \times \overline{\mathbb{DS}}_\Sigma^\lambda$ we say that (X', Y') *refines* (X, Y) if and only if:

(i) $X' \sqsubseteq X \wedge Y' \sqsubseteq Y$

(ii) $\gamma(X') \cap \gamma(Y') = \gamma(X) \cap \gamma(Y)$

We define $\mathit{Ref}_{X,Y} = \{(X', Y') \mid (X', Y') \text{ } \mathit{refines} \text{ } (X, Y)\}$.

Let us extend $\sqsubseteq$ to pairs of dashed string in a pointwise fashion: $(X, Y) \sqsubseteq (X', Y') \iff X \sqsubseteq X' \wedge Y \sqsubseteq Y'$. The following properties hold:

Proposition 2. For each $X, Y \in \overline{\mathbb{DS}}_\Sigma^\lambda$, the following properties hold:

(i) $(X, Y) = \max_\sqsubseteq \left(\mathit{Ref}_{X,Y}\right)$

(ii) $(\mathit{Ref}_{X,Y}, \sqsubseteq)$ has a minimal element

(iii) $\min_\sqsubseteq \left(\mathit{Ref}_{X,Y}\right)$ does not always exist

Proof. See Appendix.

Proposition 2 says that finding a minimal refinement according to $\sqsubseteq$ is decidable, but the *best* refinement does not always exist.

Definition 4. A function $\rho : (\overline{\mathbb{DS}}_\Sigma^\lambda \times \overline{\mathbb{DS}}_\Sigma^\lambda) \to ((\overline{\mathbb{DS}}_\Sigma^\lambda \times \overline{\mathbb{DS}}_\Sigma^\lambda) \cup \{\bot\})$ is an *equation refinement* if and only if, for each $X, Y \in \overline{\mathbb{DS}}_\Sigma^\lambda$:

(i) if $\rho(X, Y) = \bot$, then X and Y are not equatable

(ii) if $\rho(X, Y) = (X', Y')$, then (X', Y') refines (X, Y)

An equation refinement ρ is *optimal* if $\rho(X, Y) = (X', Y')$ implies that (X', Y') is a minimal element of $\mathit{Ref}_{X,Y}$ according to $\sqsubseteq$.

An algorithm $\mathcal{A}$ implementing an equation refinement ρ is called an *equation algorithm*. Given an equation algorithm $\mathcal{A}$, equating dashed strings $X, Y \in \overline{\mathbb{DS}}_\Sigma^\lambda$ means computing $\mathcal{A}(X, Y)$.

As per Proposition 2, an equation algorithm always exists (e.g., consider the identity function on $\overline{\mathbb{DS}}_{\Sigma}^{\lambda} \times \overline{\mathbb{DS}}_{\Sigma}^{\lambda}$) but this might be not unique. Finding an optimal refinement is surely decidable, but this can have high complexity. Hence, equating dashed strings implies a compromise between the precision of the refinement and its computational cost.

In [15], a sub-optimal dynamic programming algorithm was proposed to equate $X, Y \in \overline{\mathbb{DS}}_{\Sigma}^{\lambda}$ in $O(|X||Y|(|X|+|Y|))$. This complexity is however too high for a CP propagator, especially when a large portion of a dashed string is known. In [16] a sub-optimal sweep-based algorithm was proposed. Although potentially less precise, it turns out to be much more efficient for string constraint solving. This sweep-based algorithm, from now on referred as EQUATE, is revised and detailed in the next section.[1]

4.1. The EQUATE *algorithm*

The EQUATE equation algorithm takes inspiration from the propagator of the CUMULATIVE global constraint [24]: given two input dashed strings X and Y, for each block $X[i]$ we wish to find the earliest and the latest positions in Y where $X[i]$ could be matched. This information is then used to either return $\bot$ (if X and Y are not equatable) or to refine $X[i]$. This process is repeated symmetrically to refine each block of Y.

In a nutshell, for $i = 1, \dots, |X|$ and given an initial position (j, k) in Y, we try to match each block $X[i]$ against region $Y[(j, k), ...]$. If we succeed, then (j, k) is a *start position* for $X[i]$ in Y and the end of this match identifies the corresponding *end position.* However, if while doing so we reach a block $Y[j']$, with $j \leq j' \leq n$, that is "not compatible" with $X[i]$, then $X[i]$ must necessarily be matched somewhere *after* block $Y[j']$. The formal definition of compatibility is given as follows:

Definition 5. A block $S^{l,u}$ is *compatible* with block $S'^{l',u'}$ if it is nullable or

[1] In [16] to distinguish between the two equation algorithms, the sweep-based algorithm was called SWEEP while the dynamic programming algorithm was called COVER.

$S^{l,u}$ and $S'^{l',u'}$ are equatable, i.e., $l > 0 \implies \gamma(S^{l,u}) \cap \gamma(S'^{l',u'}) \neq \emptyset$.

Given dashed string $X = S_1^{l_1,u_1} \cdots S_k^{l_k,u_k}$, two positions $(i,j),(i',j')$ in X, and a block $S^{l,u}$, we say that block $S^{l,u}$ *matches* region $X[(i,j),(i',j')]$ if and only if $l > 0 \implies \gamma(S^{l,u}) \cap \gamma(X[(i,j),(i',j')]) \neq \emptyset$.

Let us consider again Figure 1. For example, we have that block $\{\texttt{o}, !, ?\}^{1,2}$ is compatible with blocks $\{\texttt{o}\}^{2,4}$ and $\{!\}^{0,3}$ but not with $\{\texttt{B},\texttt{b}\}^{1,1}$ and $\{\texttt{m}\}^{1,1}$. Block $\{\texttt{b},\texttt{o},!\}^{4,8}$ matches $X[(1,0),(2,4)]$ because it is compatible with $\{\texttt{B},\texttt{b}\}^{1,1}\{\texttt{o}\}^{2,4}$. However, $\{\texttt{b},\texttt{o},!\}^{4,8}$ is not compatible with $X[(3,0),(4,3)] = \{\texttt{m}\}^{1,1}\{!\}^{0,3}$ because it does not contain any $\texttt{m}$'s (necessary to match $\{\texttt{m}\}^{1,1}$).

4.1.1. Pushing and Stretching

For each block $X[i]$ of X we wish to determine the *"earliest matching region"* in Y. To do so, Equate uses a push$^+$ operation that, given block $X[i]$, a dashed string Y and an initial position (j,k) of Y, attempts to find a matching region $Y[(p,q),(p',q')]$ for $X[i]$ such that $(j,k) \preceq (p,q)$ and (p',q') is minimal according to $\preceq$. Dually, Equate also uses a function push$^-$ which works "backwards" across the blocks, i.e., it tries to find a matching region $Y[(p,q),(p',q')]$ for block $X[i]$ such that $(p',q') \preceq (j,k)$ and (p,q) is maximal.

The algorithm for push$^+$ is given in Figure 2 and explained with practical examples in Example 1. This algorithm searches for the earliest feasible position for a block $S^{l,u}$ in Y, assuming that $S^{l,u}$ starts no earlier than (i,j). It returns a pair (P,Q) where P is the earliest start position of $S^{l,u}$ in $Y[(i,j),...]$ and $Q \succeq P$ is the corresponding earliest end. Note that if $P = Q = (|Y|+1,0)$ there is no feasible match for $S^{l,u}$ in $Y[(i,j),...]$, i.e., as proven in Lemma 1, for each $w \in \gamma(S^{l,u})$ there is no $w' \in \gamma(Y[(i,j),...])$ such that w is a *substring* of w'.

Example 1. Consider finding the earliest matching of block $B = \{a,b\}^{3,4}$ into the dashed string $Y = \{a\}^{2,3}\{c\}^{1,2}\{b\}^{1,1}\{c\}^{0,2}\{a\}^{3,4}$ starting from $Y[1,1]$, as illustrated in Figure 3. We begin trying to fit as much as possible of B into the remainder of $\{a\}^{2,3}$. However, the remnant of current block $Y[1]$ only has at most 2 characters, which leaves 1 character to be matched in following blocks.

```
function PUSH$^+$($S^{l,u}$, $Y$, $(i,j)$)
    $(i_0, j_0) \leftarrow (i, j)$
    $k \leftarrow l$
    while $k > 0$ do                          ▷ Repeat until $l$ characters are consumed...
        if $i > |Y|$ then                                  ▷ ...or the end of $Y$ is reached
            return $(i,j), (i,j)$
        $S'^{l',u'} \leftarrow Y[i]$
        if $S \cap S' = \emptyset$ then                               ▷ Incompatible blocks
            $(i,j) \leftarrow (i+1, 0)$
            if $l' > 0$ then                     ▷ If not nullable, restart from $(i+1,0)$
                $(i_0, j_0) \leftarrow (i, j)$
                $k \leftarrow l$
        else if $k \leq (u' - j)$ then                          ▷ Remainder fits in $Y[i]$
            return $(i_0, j_0), (i, j+k)$
        else                                                ▷ Fill $Y[i]$ and continue
            $k \leftarrow k - (u' - j)$
            $(i,j) \leftarrow (i+1, 0)$
    return $(i_0, j_0), (i, j)$
```

Figure 2: PUSH$^+$ algorithm.

Unfortunately, B is incompatible with $Y[2]$. Since B cannot be completely placed *before* $Y[2]$, it must be placed entirely after: we restart at the beginning of $Y[3]$.

After consuming $Y[3]$, we reach $Y[4]$ – another incompatible block, but a nullable one since it has lower bound 0. In this case, B may still cross $Y[4]$ (by setting $Y[4]$ to the null element), so we can skip it and continue matching $Y[5]$. Block $Y[5]$ consumes the remaining characters, and we terminate: we identified the earliest matching region $Y[(3,0),(5,2)]$. □

Lemma 1. Let $S^{l,u}$ be a block, Y a dashed string with $m = |Y|$, and $P \prec (m+1, 0)$ a position. If PUSH$^+(S^{l,u}, Y, P) = ((m+1,0),(m+1,0))$, then for each $w \in \gamma(S^{l,u})$ there is no $w' \in \gamma(Y[P, ...])$ such that w is substring of w'.

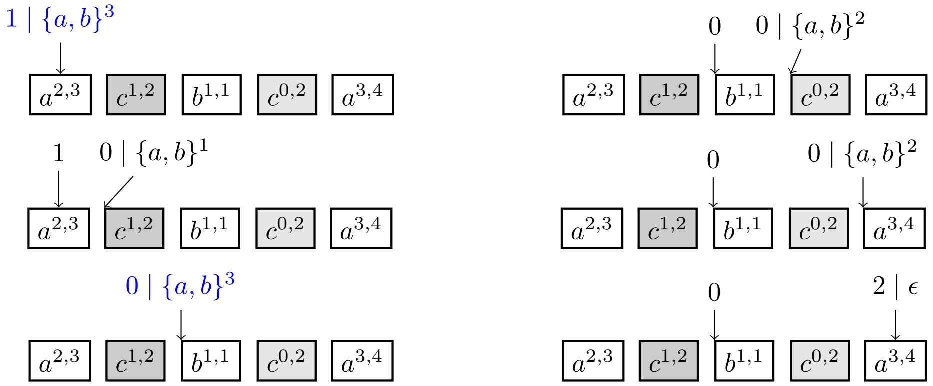
$1 \mid \{a,b\}^3$
$a^{2,3}$ $c^{1,2}$ $b^{1,1}$ $c^{0,2}$ $a^{3,4}$
1 $0 \mid \{a,b\}^1$
$a^{2,3}$ $c^{1,2}$ $b^{1,1}$ $c^{0,2}$ $a^{3,4}$
$0 \mid \{a,b\}^3$
$a^{2,3}$ $c^{1,2}$ $b^{1,1}$ $c^{0,2}$ $a^{3,4}$
0 $0 \mid \{a,b\}^2$
$a^{2,3}$ $c^{1,2}$ $b^{1,1}$ $c^{0,2}$ $a^{3,4}$
0 $0 \mid \{a,b\}^2$
$a^{2,3}$ $c^{1,2}$ $b^{1,1}$ $c^{0,2}$ $a^{3,4}$
0 $2 \mid \epsilon$
$a^{2,3}$ $c^{1,2}$ $b^{1,1}$ $c^{0,2}$ $a^{3,4}$

Figure 3: Applying PUSH$^+$ to find the earliest start of $\{a,b\}^{3,4})$ from $Y[1,1]$. End-point markers show current offset, and characters yet to be consumed. Blocks incompatible with $\{a,b\}^{3,4}$ are in grey colour. Arrows delimits the current matching region.

Proof. See Appendix.

Let us now introduce a notion of matching which we shall use to prove the soundness of EQUATE. If we take a concrete string $z \in \gamma(X) \cap \gamma(Y)$ where $X, Y \in \overline{\mathbb{DS}}^{\lambda}_{\Sigma}$, then because $z \in \gamma(X)$ there must be $n = |X|$ substrings $z_1, \dots, z_n$ such that $z = z_1 \cdots z_n$ and $z_i \in \gamma(X[i])$. But because $z \in \gamma(Y)$ there must be also n *regions* of Y (not necessarily blocks of Y) $Y_1, \dots, Y_n$ such that $Y = Y_1 \cdots Y_n$ and $z_i \in \gamma(Y_i)$. These regions define what we call a *concrete matching* for z in Y. Formally:

Definition 6. Let $X, Y \in \overline{\mathbb{DS}}^{\lambda}_{\Sigma}$ be equatable dashed strings with $n = |X|, m = |Y|$ and let $z = z_1 \cdots z_n \in \gamma(X) \cap \gamma(Y)$ such that $z_i \in \gamma(X[i])$ for $i = 1, \dots, n$. A *concrete matching* for z in Y is given by $n+1$ positions $\mathbb{P}_1, \dots, \mathbb{P}_{n+1}$ such that:

(i) $\mathbb{P}_1 = (1, 0) \preceq \mathbb{P}_2 \preceq \cdots \preceq \mathbb{P}_n \preceq \mathbb{P}_{n+1} = (m+1, 0)$

(ii) $z_i \in \gamma(Y[\mathbb{P}_i, \mathbb{P}_{i+1}])$ for $i = 1, \dots, n$

By definition, if $z \in \gamma(X) \cap \gamma(Y)$ then a concrete matching for z in Y (and dually in X) always exists. However, this may be not unique. For instance, if $X =$

$\{a\}^{1,1}\{b,c\}^{1,1}$, $Y = \{a,b\}^{1,2}\{b\}^{0,1}$ and $z = ab$ then both $(1,0),(1,1),(3,0)$ and $(1,0),(1,2),(3,0)$ are concrete matchings for z in Y because character a matches both $Y[(1,0),(1,1)] = \{a,b\}^{1,1}$ and $Y[(1,0),(1,2)] = \{a,b\}^{1,2}$, while character b matches both $Y[(1,1),(3,0)] = \{a,b\}^{0,1}\{b\}^{0,1}$ and $Y[(1,2),(3,0)] = \{b\}^{0,1}$.

Lemma 2 uses the notion of concrete matching to show a property of $\textsc{push}$ later on used to prove the soundness of $\textsc{Equate}$. It essentially proves that, given any concrete matching $\mathbb{P}_1,\ldots,\mathbb{P}_{n+1}$, if $\textsc{push}^+(X[i],Y,P)$ starts from a position P *before* $\mathbb{P}_i$ then it cannot finish *after* $\mathbb{P}_{i+1}$.

Lemma 2. Let $X, Y \in \overline{\mathbb{DS}}^{\lambda}_{\Sigma}$ with $n = |X|$ and $z = z_1 \cdots z_n \in \gamma(X) \cap \gamma(Y)$ such that $z_i \in \gamma(X[i])$ for $i = 1,\ldots,n$. Let $\mathbb{P}_1,\ldots,\mathbb{P}_{n+1}$ be a concrete matching for z in Y. Then, for each $P \preceq \mathbb{P}_i$ we have that $(P', P'') = \textsc{push}^+(X[i],Y,P)$ implies $P' \preceq \mathbb{P}_i$ and $P'' \preceq \mathbb{P}_{i+1}$.

Proof. See Appendix.

The algorithm for $\textsc{push}^-$ is dual: it just considers blocks from left to right instead of from right to left and if returns $(0,0)$ then there is no matching for $S^{l,u}$ in Y. Unsurprisingly, the following lemmas hold for $\textsc{push}^-$:

Lemma 3. Let $S^{l,u}$ be a block, Y a dashed string and P a position. If $\textsc{push}^-(S^{l,u},Y,P) = ((0,0),(0,0))$, then for each $w \in \gamma(S^{l,u})$ there is no $w' \in \gamma(Y[(1,0),P])$ such that w is substring of w'.

Proof. See Appendix.

Lemma 4. Let $X, Y \in \overline{\mathbb{DS}}^{\lambda}_{\Sigma}$ with $n = |X|$ and $z = z_1 \cdots z_n \in \gamma(X) \cap \gamma(Y)$ such that $z_i \in \gamma(X[i])$, and $\mathbb{P}_1 = (1,0) \preceq \mathbb{P}_2 \preceq \cdots \preceq \mathbb{P}_n \preceq \mathbb{P}_{n+1} = (|Y|+1)$ such that $z_i \in \gamma(Y[\mathbb{P}_i, \mathbb{P}_{i+1}])$ for $i = 1,\ldots,n$. For each $P \succeq \mathbb{P}_{i+1}$ we have that $(P', P'') = \textsc{push}^-(X[i],Y,P)$ implies $P' \succeq \mathbb{P}_i$ and $P'' \succeq \mathbb{P}_{i+1}$.

Proof. See Appendix.

To improve the precision of $\textsc{Equate}$, we also introduce a suitable "*stretch*" procedure (pseudo-code in Figure 4). The $\textsc{stretch}^+$ function is in some sense a

```
function STRETCH$^+$($S^{l,u}$, $Y$, $(i,j)$)
    $k \leftarrow u$, $n \leftarrow |Y|$
    while $i \leq n$ do
        $S'^{l',u'} \leftarrow Y[i]$
        $b \leftarrow l' - j$                          ▷ Minimum number of characters to consume
        if $(b \leq 0)$ then                                          ▷ Nothing to consume
            $(i,j) \leftarrow (i+1, 0)$
        else if $S \cap S' = \emptyset$ then                                ▷ Incompatible blocks
            return $(i,j)$
        else if $k < b$ then                                           ▷ No more characters
            return $(i, j+k)$
        else                                                      ▷ $b$ characters consumed
            $k \leftarrow k - b$
            $(i,j) \leftarrow (i+1, 0)$
    return $(i,j)$
```

Figure 4: STRETCH$^+$ attempts to find the latest end position of $S^{l,u}$, assuming it starts no later than (i,j). It consumes the minimum possible of each successive block, and stops when it finds an incompatible block, or has consumed all possible characters, or reaches the end of Y.

dual of PUSH$^+$: where PUSH$^+$ attempts to squeeze as much of $X[i]$ into a block of Y as possible, STRETCH$^+$ consumes only the minimum amount of $X[i]$ before moving onto the next block of Y.

A fundamental difference between PUSH and STRETCH is that, unlike PUSH, the STRETCH procedure does not skip incompatible blocks: it returns the *latest position* where a block $S^{l,u}$ match a region of Y. In other terms, the start position for STRETCH is fixed: if a block $S'^{l',u'}$ of Y is incompatible with $S^{l,u}$, the algorithm terminates (while PUSH updates the start position of $S^{l,u}$ by restarting the search from the following block). Example 2 shows how STRETCH works in practice.

Example 2. Recall the dashed string Y from Example 1. Consider the case (a)

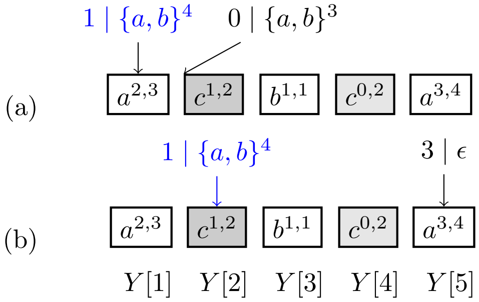

Figure 5: Applying STRETCH^{+} on $\{a,b\}^{3,4}$ from two possible start positions in Y. End-point markers show current offset, and characters still available.

shown in Figure 5, where the predecessors of the current block $B = \{a,b\}^{3,4}$ can only reach position $Y[1,1]$. We have 4 characters available, and start walking along successive blocks of Y, decreasing our budget by the lower bound (i.e., the mandatory characters) of each block we cross. As we start from $Y[1,1]$, we consume $2-1=1$ character and we reach position $Y[2,0]$ with 3 characters remaining. However, $Y[2]$ is incompatible with B. In this case we terminate immediately, concluding that X must end before the beginning of $Y[2]$ – and accordingly, the successor of X must begin no later than $Y[2,0]$.

For case (b), we start at position $Y[2,1]$. Note that while $Y[2]$ is incompatible with B, position $Y[2,1]$ indicates that we are actually considering the *optional part* of $\{c\}^{1,2}$, so its effective lower bound is 0, so we may continue unopposed. We continue as before, by reaching $Y[4]$ with 3 characters available after consuming 1 character for $Y[3]$. Block $Y[4]$ is again incompatible with B, but it is nullable and thus it can be skipped. The remaining budget is consumed in $Y[5]$: the latest end position for B is $Y[5,3]$. Note that $Y[5,3]$ is not the last position Y, but it nevertheless denotes a feasible end position, as the remainder of the dashed string $Y[5,4]$ is optional. □

Lemma 5 is analogous to Lemma 2. Let $z = z_1 \cdots z_n \in \gamma(X) \cap \gamma(Y)$ with $n = |X|$ such that $z_i \in \gamma(X[i])$, and $\mathbb{P}_1, \ldots, \mathbb{P}_{n+1}$ a concrete matching for z in Y. Lemma 5 basically says that for any position P not before $\mathbb{P}_i$ we have that

$\textsc{stretch}^+(X[i], Y, P)$ returns a position not before $\mathbb{P}_{i+1}$.

Lemma 5. Let $X, Y \in \overline{\mathbb{DS}}^{\lambda}_{\Sigma}$ with $n = |X|$ and $z = z_1 \cdots z_n \in \gamma(X) \cap \gamma(Y)$ such that $z_i \in \gamma(X[i])$ for $i = 1, \ldots, n$, and let $\mathbb{P}_1, \ldots, \mathbb{P}_{n+1}$ a concrete matching for z in Y. For each position $P \succeq \mathbb{P}_i$ we have $\textsc{stretch}^+(X[i], Y, P) \succeq \mathbb{P}_{i+1}$.

Proof. See Appendix.

$\textsc{stretch}^+$ is useful to find initial bounds on the end positions: we run progressively the algorithm for $X[0], X[1], \ldots$ from the beginning of Y. Analogously to $\textsc{push}^-$, we define $\textsc{stretch}^-$, the "backward" version of $\textsc{stretch}^+$ considering blocks from right to left to compute initial bounds for the start positions (see Figure 6).

Lemma 6. Let $X, Y \in \overline{\mathbb{DS}}^{\lambda}_{\Sigma}$ with $n = |X|$ and $z = z_1 \cdots z_n \in \gamma(X) \cap \gamma(Y)$ such that $z_i \in \gamma(X[i])$, and let $\mathbb{P}_1, \ldots, \mathbb{P}_{n+1}$ a concrete matching for z in Y. For each position P we have that $P \preceq \mathbb{P}_{i+1}$ implies $\textsc{stretch}^-(X[i], Y, P) \preceq \mathbb{P}_i$.

Proof. See Appendix.

The Init algorithm in Figure 7 shows how $\textsc{stretch}$ is used to determine initial bounds for the *earliest start* and *latest end* positions for the blocks of X when equating X and Y. If $\textsc{init}(X, Y) = (\bot, \bot)$, then X and Y are not equatable as proved in Lemma 8 (the reverse implication is instead not true in general). Otherwise, $\textsc{init}(X, Y)$ returns a pair (ESP, LEP) of arrays such that for $i = 1, \ldots, |X|$ the position $ESP[i]$ (resp., $LEP[i]$) is a sound start position (resp., end position) for block $X[i]$ in Y. Precisely, Lemma 7 proves that, given $z = z_1 \cdots z_n \in \gamma(X) \cap \gamma(Y)$ such that $z_i \in \gamma(X[i])$ for $i = 1, \ldots, n$ we have that $Y[ESP[i], LEP[i]]$ is the *biggest feasible region* of Y possibly matching z_i.

Lemma 7. Let $X, Y \in \overline{\mathbb{DS}}^{\lambda}_{\Sigma}$ with $n = |X|$ and $\textsc{Init}(X, Y) = (ESP, LEP) \neq (\bot, \bot)$. Let $z = z_1 \cdots z_n \in \gamma(X) \cap \gamma(Y)$ such that $z_i \in \gamma(X[i])$ for $i = 1, \ldots, n$, and let $\mathbb{P}_1, \ldots, \mathbb{P}_{n+1}$ be a concrete matching for z in Y. Then, $ESP[i] \preceq \mathbb{P}_i$ and $\mathbb{P}_{i+1} \preceq LEP[i]$ for $i = 1, \ldots, n$.

```
function stretch$^-$($S^{l,u}$, $Y$, $(i,j)$)
    $k \leftarrow u$
    if $j > lb(Y[i])$ then
        $j \leftarrow lb(Y[i])$                                ▷ Skip the optional part of $Y[i]$
    while $i > 0$ do
        $S'^{l',u'} \leftarrow Y[i]$
        if $l' = 0$ then                                       ▷ Skip block $Y[i]$
            if $i > 1$ then
                $(i,j) \leftarrow (i-1, lb(Y[i-1]))$
            else
                return $(0,0)$
        else if $S \cap S' = \emptyset$ then                   ▷ We cannot go further
            return $(i,j)$
        else if $k < l'$ then
            return $(i, j-k)$                                  ▷ No more characters
        else
            $k \leftarrow k - l'$                              ▷ Consuming the least characters of $Y[i]$
            if $i > 1$ then
                $(i,j) \leftarrow (i-1, lb(Y[i-1]))$
            else
                return $(0,0)$
    return $(i,j)$
```

Figure 6: stretch$^-$ attempts to find the earliest start position of $S^{l,u}$, assuming it starts no earlier than (i,j).

Proof. See Appendix.

Lemma 8. Let $X, Y \in \overline{\mathbb{DS}}^{\lambda}_{\Sigma}$. If Init$(X,Y) = (\bot,\bot)$ then $\gamma(X) \cap \gamma(Y) = \emptyset$.

Proof. See Appendix.

Now, assuming init$(X,Y) \neq (\bot,\bot)$, we want to refine the initial values of

```
function INIT($X = [X_1, \ldots, X_n]$, $Y = [Y_1, \ldots, Y_m]$)
    $curr \leftarrow (1, 0)$
    for $i \leftarrow 1, \ldots, n$ do                    ▷ Stretching forward for latest ends
        $LEP[i] \leftarrow$ STRETCH$^+(X[i], Y, curr)$
        $curr \leftarrow LEP[i]$
    if $LEP[n] \prec (m, ub(Y_m))$ then
        return $(\bot, \bot)$                            ▷ The end of $Y$ cannot be reached
    $curr \leftarrow (m, ub(Y_m))$
    for $i \in \{n, n-1, \ldots, 1\}$ do        ▷ Stretching backward for earliest starts
        $ESP[i] \leftarrow$ STRETCH$^-(X[i], Y, curr)$
        $curr \leftarrow ESP[i]$
    if $(1, 0) \prec ESP[1]$ then
        return $(\bot, \bot)$                      ▷ The beginning of $Y$ cannot be reached
    return $(ESP, LEP)$
```

Figure 7: Algorithm for initialising the earliest starts and the latest ends for the blocks of X when equating X and Y.

ESP and LEP (from which we will be able to extract also the *latest start* and *earliest end* positions). To do so, we use PUSH in both directions. The algorithm PUSHESP(X, Y, ESP, i) shown in Figure 8 aims to *improve* the earliest start positions by pushing forward $X[i]$ from the current $ESP[i]$ position. It returns the possibly updated ESP array, or $\bot$ if $X[i]$ cannot match any block of Y: in this case X and Y are not equatable (see Lemma 9).

Lemma 9. Let $X, Y \in \overline{\mathbb{DS}}^{\lambda}_{\Sigma}$ such that INIT$(X, Y) = (ESP, LEP)$. Then, for each $i = 1, \ldots, |X|$, PUSHESP$(X, Y, ESP, i) = \bot$ implies $\gamma(X) \cap \gamma(Y) = \emptyset$.

Proof. See Appendix.

Note that after updating $ESP[i]$ in line 12, we may want to perform a *stretch backward* from *start* position for possibly improving the earliest end of the previous blocks $X[i-1], X[i-2], \ldots$ (or returning $\bot$). However, despite being

```
function $\textsc{push}\mathrm{ESP}(X = [X_1, \ldots, X_n],\ Y = [Y_1, \ldots, Y_m],\ ESP,\ i)$
    if $lb(X[i]) = 0$ then                    ▷ $X[i]$ nullable, not pushing $ESP[i]$
        if $i < n \ \wedge\ ESP[i+1] \prec ESP[i]$ then
            $ESP[i+1] \leftarrow ESP[i]$            ▷ $X[i+1]$ cannot start before $X[i]$
        return $ESP$
    $(start, end) \leftarrow \textsc{push}^{+}(X[i], Y, ESP[i])$
    if $start = (m+1, 0)$ then               ▷ $X[i]$ cannot match any block of $Y$
        return $\bot$
    if $i < n \ \wedge\ ESP[i+1] \prec end$ then   ▷ $X[i+1]$ cannot start before $end$
        $ESP[i+1] \leftarrow end$
    if $ESP[i] \prec start$ then                      ▷ Pushing $ESP[i]$ forward
        $ESP[i] \leftarrow start$
    return $ESP$
```

Figure 8: Algorithm for computing the earliest start position in Y of each X-block.

potentially more precise, this approach requires an additional layer of complexity (it implies reprocessing blocks we already considered, including $X[i]$ itself) and we experimentally verified that this is almost never worthwhile.

Lemma 10 proves that $\textsc{push}$ESP possibly refines the precision of ESP positions while preserving their soundness.

Lemma 10. Let $X, Y \in \overline{\mathbb{DS}}^{\lambda}_{\Sigma}$ with $n = |X|$. Let $z = z_1 \cdots z_n \in \gamma(X) \cap \gamma(Y)$ such that $z_i \in \gamma(X[i])$ for $i = 1, \ldots, n$, and let $\mathbb{P}_1, \ldots, \mathbb{P}_{n+1}$ a concrete matching for z in Y. Let ESP such that $ESP[i] \preceq \mathbb{P}_i$ for $i = 1, \ldots, n$. If $ESP' = \textsc{push}\mathrm{ESP}(X, Y, ESP, i) \neq \bot$ then $ESP'[i] \preceq \mathbb{P}_i$ for $i = 1, \ldots, n$.

Proof. See Appendix.

The algorithm $\textsc{push}$LEP for computing the latest ends is symmetrical: it works right to left by using $\textsc{push}^{-}$ to update $LEP[n], LEP[n-1], \ldots, LEP[1]$. Unsurprisingly, the following properties hold for $\textsc{push}$LEP:

Lemma 11. Let $X, Y \in \overline{\mathbb{DS}}^{\lambda}_{\Sigma}$ such that $\textsc{Init}(X, Y) = (ESP, LEP)$. Then, for each $i = 1, \ldots, |X|$, $\textsc{push}\text{LEP}(X, Y, LEP, i) = \bot$ implies $\gamma(X) \cap \gamma(Y) = \emptyset$.

Proof. See Appendix.

Lemma 12. Let $X, Y \in \overline{\mathbb{DS}}^{\lambda}_{\Sigma}$ with $n = |X|$. Let $z = z_1 \cdots z_n \in \gamma(X) \cap \gamma(Y)$ such that $z_i \in \gamma(X[i])$ for $i = 1, \ldots, n$, and let $\mathbb{P}_1, \ldots, \mathbb{P}_{n+1}$ a concrete matching for z in Y. Let LEP such that $\mathbb{P}_{i+1} \preceq LEP[i]$. If $LEP' = \textsc{push}\text{LEP}(X, Y, LEP, i) \neq \bot$ then $\mathbb{P}_{i+1} \preceq LEP'[i]$ for $i = 1, \ldots, n$.

Proof. See Appendix.

Once computed the earliest start and the latest end positions, we can derive (an over-approximation of) the latest starts and the earliest ends. Figure 9 shows the $\textsc{Sweep}$ algorithm for computing the earliest/latest start/end positions (ESP, LSP, EEP, LEP) of each block $X[i]$.

$\textsc{Sweep}$ first computes (lines 2–12) the arrays for earliest start positions (ESP) and latest end positions (LEP). Then, ESP and LEP are used to initialise the arrays for latest starts (LSP) and earliest ends (EEP).

The loop in lines 14–17 tries to update the latest start $LSP[i]$ for each block $X[i]$ with the latest end of its predecessor $X[i-1]$. This makes sense only if $ESP[i] \preceq LEP[i-1]$, i.e., if the earliest start of a block is not after its latest start. If not, $\bot$ is returned.

The loop in lines 18–21 computes the earliest end positions (and possibly updates the latest ends). This procedure is symmetrical to the latest ends computation of lines 14–17.

Lemma 13 proves that if $\bot$ is returned then X and Y are surely not equatable, while Lemma 14 asserts that $\textsc{Sweep}$ preserves the soundness of ESP and LEP positions.

Lemma 13. Let $X, Y \in \overline{\mathbb{DS}}^{\lambda}_{\Sigma}$. If $\textsc{Sweep}(X, Y) = \bot$, then $\gamma(X) \cap \gamma(Y) = \emptyset$.

Proof. See Appendix.

```
function SWEEP($X = [X_1, \ldots, X_n]$, $Y = [Y_1, \ldots, Y_m]$)
    $(ESP, LEP) \leftarrow$ INIT$(X, Y)$
    if $(ESP, LEP) = (\bot, \bot)$ then
        return $\bot$
    for $i \leftarrow 1, \ldots, n$ do                    ▷ Pushing forward for earliest starts
        $ESP \leftarrow$ pushESP$(X[i], Y, ESP, i)$
        if $ESP[i] = \bot$ then
            return $\bot$
    for $i \leftarrow n, \ldots, 1$ do                    ▷ Pushing backward for latest ends
        $LEP \leftarrow$ pushLEP$(X[i], Y, LEP, i)$
        if $LEP[i] = \bot$ then
            return $\bot$
    $LSP \leftarrow ESP$,    $EEP \leftarrow LEP$
    for $i \leftarrow 2, \ldots, n$ do                              ▷ Updating latest starts
        $LSP[i] \leftarrow LEP[i-1]$
        if $LSP[i] \prec ESP[i]$ then
            return $\bot$
    for $i \leftarrow n-1, \ldots, 1$ do                           ▷ Updating earliest ends
        $EEP[i] \leftarrow ESP[i+1]$
        if $LEP[i] \prec EEP[i]$ then
            return $\bot$
    return $(ESP, LSP, EEP, LEP)$
```

Figure 9: SWEEP algorithm for computing earliest/latest start/end positions.

Lemma 14. Let $X, Y \in \overline{\mathbb{DS}}^{\lambda}_{\Sigma}$ with $n = |X|$ and $z = z_1 \ldots z_n \in \gamma(X) \cap \gamma(Y)$ such that $z_i \in \gamma(X[i])$ for $i = 1, \ldots, n$. Let $\mathbb{P}_1, \ldots, \mathbb{P}_{n+1}$ be a concrete matching for z in Y. If SWEEP$(X, Y) = (ESP, LEP, EEP, LEP)$ then $ESP[i] \preceq \mathbb{P}_i \preceq LSP[i]$ and $EEP[i] \preceq \mathbb{P}_{i+1} \preceq LEP[i]$ for $i = 1, \ldots, n$.

Proof. See Appendix.

Lemma 15 concludes this section by proving that the worst-case complexity of Sweep algorithm is linear in the number of blocks.

Lemma 15. Let $X, Y \in \overline{\mathbb{DS}}_{\Sigma}^{\lambda}$ with $n = |X|$ and $m = |Y|$. The worst-case complexity of Sweep (X, Y) is $O(D(n+m))$ where $O(D)$ is the worst-case complexity of checking that the base of X and the base of Y are disjoint.

Proof. See Appendix.

4.1.2. Refining

Let us assume that $\textsc{Sweep}(X, Y) \neq \bot$. Now, we attempt to refine each block $X[i]$ into a (sequence of) block(s) X'_i such that $X'_i \sqsubseteq X[i]$. In this way, $X' = \textsc{norm}(X'_1 \cdots X'_n) \sqsubseteq X$. However, we must ensure that $\gamma(X') \cap \gamma(Y) = \gamma(X) \cap \gamma(Y)$, i.e., that we do not remove feasible values from $\gamma(X)$. To do so, we introduce the notions of feasible, mandatory and optional regions.

Definition 7. Let $\textsc{Sweep}(X, Y) = (ESP, LSP, EEP, LEP)$. For each block $X[i]$ of X we define its:

- *feasible region* (w.r.t. Y) as $feas_Y(X[i]) = Y[ESP[i], LEP[i]]$
- *mandatory region* (w.r.t. Y) as $mand_Y(X[i]) = Y[LSP[i], EEP[i]]$
- *left-optional region* (w.r.t. Y) as $lopt_Y(X[i]) = Y[ESP[i], LSP[i]]$
- *right-optional region* (w.r.t. Y) as $ropt_Y(X[i]) = Y[EEP[i], LEP[i]]$

The feasible region delimits the *biggest region* $Y[(p, q), (p', q')]$ where a block $B = S^{l,u}$ of X *may* match a dashed string Y. Hence, from $feas_Y(B)$ we may refine the base S into $S' = S \cap (S_p \cup S_{p+1} \cup \cdots \cup S_{p'-1} \cup S_{p'})$. Similarly, we may refine the upper bound u into $u' = \min(u, (u_p - q) + u_{p+1} + \cdots + u_{p'-1} + q')$. However we can be more precise by only considering the upper bounds of the blocks $S_j^{l_j, u_j}$ having $S \cap S_j \neq \emptyset$ for $j = p, p+1, \ldots, q$, i.e.:

$$u' = \min(u, \nabla_{S,S_p}(u_p - q) + \nabla_{S,S_{p+1}}(u_{p+1}) + \cdots + \nabla_{S,S_{p'-1}}(u_{p'-1}) + \nabla_{S,S'_q}(q'))$$

where $\nabla_{S,S'}(x) = \begin{cases} x & \text{if } S \cap S' \neq \emptyset \text{ and } x > 0 \\ 0 & \text{otherwise} \end{cases}$

If $p = p'$, we can be even more precise: $u' = \min(u, \nabla_{S,S_p}(q' - q))$.

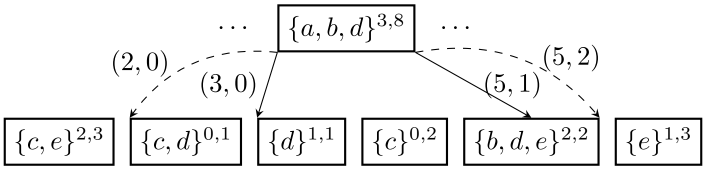

Figure 10: Filtering block $\{a, b, d\}^{3,8}$ using matching bounds. Feasible bounds are indicated with dashed lines, mandatory bounds are indicated by solid lines.

The mandatory region $Y[(h, k),(h', k')]$ delimits a region that *must* be matched by a block of X. It does not help to refine the base nor the upper bound of a block but, if $h < h'$, we can refine lower bound l into:

$$l' = \max(l, \nabla_{S,S_h}(l_h - k) + \nabla_{S,S_{h+1}}(l_{h+1}) + \cdots + \nabla_{S,S_{h'-1}}(l_{h'-1}) + \nabla_{S,S_k}(k'))$$

If $h = h'$, then simply $l' = \max(l, \nabla_{S,S_h}(k' - k))$.

Note that l' can be used to further refine the upper bound u of the block. Indeed, for $j = p, \ldots, q$ it must be that $u_j + (l' - l_j) \leq u$ because the optional characters of block $S_j^{l_j,u_j}$ plus the mandatory characters of all the other blocks of the feasible region cannot exceed u. Hence, $u_j \leq u - l' + l_j$: instead of $\nabla_{S,S_j}(u_j)$ we can consider $\nabla_{S,S_j}(\min(u_j, u - l' + l_j))$.

Example 3. Consider Figure 10, where block $B = \{a, b, d\}^{3,8}$ is matched against dashed string $Y = \{c, e\}^{2,3}\{c, d\}^{0,1}\{d\}^{1,1}\{c\}^{0,2}\{b, d, e\}^{2,2}\{e\}^{1,3}$. We have $\mathit{feas}_Y(B) = Y[(2, 0),(5, 2)]$ because B is not compatible with $Y[1]$ and $Y[6]$. The mandatory region is $\mathit{mand}_Y(B) = Y[(3, 0),(5, 1)]$, because B cannot start after block $Y[3]$, and must consume at least a character of $Y[5]$. The optional regions are $\mathit{lopt}_Y(B) = [(2, 0),(3, 0)]$ and $\mathit{ropt}_Y(B) = [(5, 1),(5, 2)]$. From $\mathit{feas}_Y(B)$ we can refine the base of B into $S' = \{a, b, d\} \cap (\{c, d\} \cup \{d\} \cup \{c\} \cup \{b, d, e\}) = \{b, d\}$. The upper bound of B becomes $u' = \min(8, 1 + 1 + 0 + 2) = 4$, while its lower bound does not change: $l' = \max(3, 1 + 0 + 1) = 3$. Hence, we can safely refine $B = \{a, b, d\}^{3,8}$ into $B' = \{b, d\}^{3,4}$. □

Now, the question is: can we do better than this? Rather than merely pruning block $S^{l,u}$ into $S'^{l'u'}$ we may instead *replace* it with the normalised dashed string

$\textsc{norm}((S_1 \cap S)^{l_1,u_1} \cdots (S_k \cap S)^{l_k,u_k})$ where $\mathit{feas}_Y(S^{l,u}) = S_1^{l_1,u_1} \cdots S_k^{l_k,u_k}$. In the example above, we would refine B with $B'' = \textsc{norm}(\{d\}^{0,1}\{d\}^{1,1}\emptyset^{0,2}\{b,d\}^{2,2}) = \{d\}^{1,2}\{b,d\}^{2,2}$ instead of $B' = \{b,d\}^{3,4}$. This would be a tighter refinement: $\|\{d\}^{1,2}\{b,d\}^{2,2}\| = 2 \cdot 4 = 8$ while $\|\{b,d\}^{3,4}\| = 24$. However, as we shall explain in Example 4*(i)*, care must be taken not to lose any cardinality information. For instance, if $\sum_{i=1}^{k} u_i > u$ it is not sound replace block $S^{l,u}$ with $R = \textsc{norm}((S_1 \cap S)^{l_1,u_1} \cdots (S_k \cap S)^{l_k,u_k})$ because $\gamma(R) \not\subseteq \gamma(S^{l,u})$. Moreover, even when it is sound (see Example 4*(ii)*) replacing a block with its feasible region is not necessarily beneficial because this may introduce a lot of *symmetries*.

Example 4. *(i)* Consider matching block $B = \{a,b,c\}^{0,4}$ against dashed string Y such that $\mathit{feas}_Y(B) = B' = \{a,c\}^{0,2}\{b,c\}^{3,4}$. Replacing B with B' is tempting, as this would provide more precise information about the *sequencing* of characters (e.g., if both a and b occur in $\gamma(B)$, then a must appear before b). However, $\mathit{ub}(B'[1]) + \mathit{ub}(B'[2]) = 6 > 4 = \mathit{ub}(B)$, so $\gamma(B') \not\subseteq \gamma(B)$ (e.g., $ccccc \in \gamma(B') - \gamma(B)$). Thus, we cannot refine B with $\mathit{feas}_Y(B)$.

(ii) Consider matching block $B = \{a,b,c\}^{0,4}$ against dashed string Y such that $\mathit{feas}_Y(B) = B' = \{a,b\}^{0,1}\{a,b,c\}^{0,1}\{a,b\}^{0,1}\{a,b,c\}^{0,1}$. We clearly have $\gamma(B') \subset \gamma(B)$ (e.g., $cccc \in \gamma(B) - \gamma(B')$). However, $\|B\| = \sum_{i=0}^{4} 3^i = 121 < \|B'\| = 3 \cdot 4 \cdot 3 \cdot 4 = 144$. This is because, unlike B, $\mathit{feas}_Y(B)$ contains a number of symmetries (e.g., there are many ways to denote the string aa, namely: $\epsilon\epsilon aa, \epsilon a\epsilon a, \epsilon aa\epsilon, a\epsilon\epsilon a, a\epsilon a\epsilon, aa\epsilon\epsilon$). □

To overcome the above drawbacks, we adopt the following strategy. Let $B = S^{l,u}$ be a block of X to be refined, $\mathit{feas}_Y(B) = S_1^{l_1,u_1} \cdots S_k^{l_k,u_k}$ its (non-empty) feasible region, and $\mathit{mand}_Y(B) = S_i^{l_i,u_i} \cdots S_j^{l_j,u_j}$ its (possibly empty) mandatory region. We compute $S' = \bigcup_{h=1}^{k} S_h$, $l' = \sum_{h=i}^{j} \nabla_{S,S_h}(l_h)$, and then $u' = \sum_{h=1}^{k} \nabla_{S,S_h}(\min(u_h, u - l' + l_h))$.

If $l' < l$ or $u' > u$, we cannot replace B with $\mathit{feas}_Y(B)$, so we refine B with $\textsc{norm}\left((S \cap S')^{l,\min(u,u')}\right)$. A normalisation is needed to uniquely represent $\emptyset^{0,0}$. To limit the symmetries this refinement is also applied when $l' = 0$, i.e., when the whole mandatory region is nullable.

If instead $l' > 0$ and $l \le l' \le u' \le u$, we refine B with $\textsc{norm}(L_B \cdot M_B \cdot R_B)$, where $L_B = \left(S \cap \left(\bigcup_{h=1}^{i-1} S_h\right)\right)^{0,\sum_{h=1}^{i-1} u_h}$, $M_B = (S \cap S_i)^{l_i,u_i} \cdots (S \cap S_j)^{l_j,u_j}$, and $R_B = \left(S \cap \left(\bigcup_{h=j+1}^{k} S_h\right)\right)^{0,\sum_{h=j+1}^{k} u_h}$.

In the "central" part M_B we take the mandatory region and we intersect each of its bases with S. The block L_B (resp., R_B) is obtained by "crushing" the left (right) optional region of $S^{l,u}$ into a single block over-approximating $\mathit{lopt}_Y(B)$ ($\mathit{ropt}_Y(B)$). Note that the lower bounds of L_B and R_B (i.e., the characters that may appear) are set to 0 to preserve the soundness of the refinement. We crush the optional blocks instead of considering $(S \cap S_1)^{0,u_1} \cdots (S \cap S_{i-1})^{0,u_{i-1}}$ and $(S \cap S_{j+1})^{0,u_{j+1}} \cdots (S \cap S_k)^{0,u_k}$ to prevent the proliferation of symmetries, and to minimise the dashed string size.

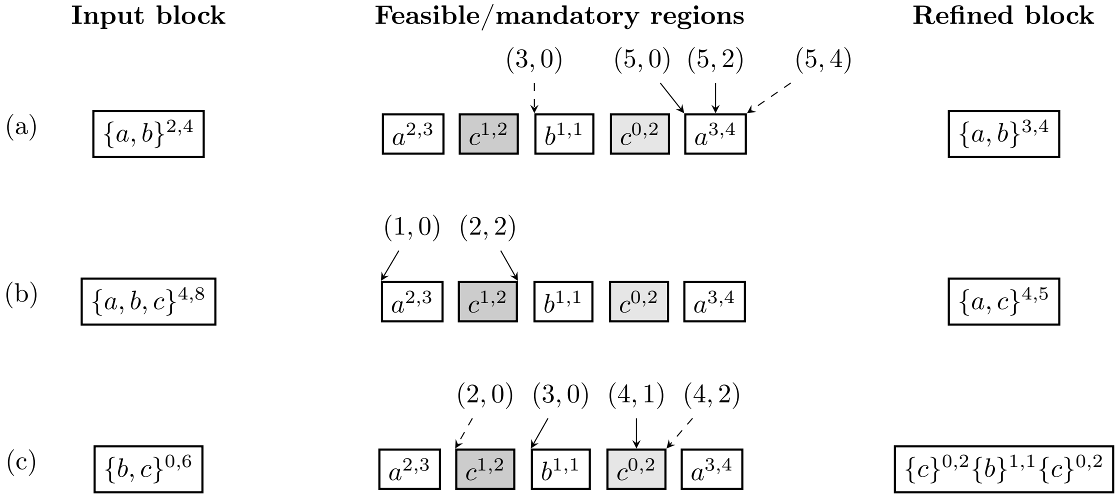

Figure 11: Refining different blocks, given different possible matchings in Y.

Example 5. Recall the dashed string $Y = \{a\}^{2,3}\{c\}^{1,2}\{b\}^{1,1}\{c\}^{0,2}\{a\}^{3,4}$ from Example 1. Figure 11 illustrates how the blocks in the first column are refined, according to the matching regions provided in the second column (dashed arrows indicate the feasible region, solid arrows the mandatory region).

In case (a) we try to refine $\{a,b\}^{2,4}$. We have $S' = \{a,b,c\}$, $l' = 2$, and $u' = 1 + 4 = 5$. Because $u' > 4$, we crush the feasible region into $(\{a,b\} \cap$

$\{a, b, c\})^{\max(3,2),\min(4,5)} = \{a, b\}^{3,4}$. This refinement enables us to infer that each concrete string w denoted by block (a) contains at least 3 characters, however the information that w always contains the substring aaa is lost.

In case (b) the feasible region $Y[(1, 0), (2, 2)]$ coincides with the mandatory region. Because $l' = lb(Y[1]) + lb(Y[2]) = 3 < 4$ we refine block $\{a, b, c\}^{4,8}$ with $(\{a, b, c\} \cap \{a, c\})^{\max(4,1+2),\min(8,3+2)} = \{a, c\}^{4,5}$.

In case (c), since $l' = 1 \geq 0$ and $u' = 5 \leq 6$, we can safely replace $\{b, c\}^{0,6}$ with $\textsc{norm}((\{b, c\} \cap \{c\})^{0,2} \cdot \{b\}^{1,1}\{c\}^{0,1} \cdot (\{b, c\} \cap \{c\})^{0,1}) = \{c\}^{0,2}\{b\}^{1,1}\{c\}^{0,2}$.

□

Figure 12 shows the pseudo-code of Refine, implementing the strategy defined above to refine each block $X[h]$ of X into (a sequence of) block(s) X'_h given the matching arrays returned by Sweep. We use variables ES, LS, EE, LE to avoid recomputing the matching regions for two blocks having the same matching regions. If block $X[h]$ is known, there is nothing to refine.

The if statement between lines 6–14 computes S' and l' from the matching region of a block $X[h]$, and possibly computes a dashed string M for refining $X[h]$ without crushing the matching region into a single block. Note that the value of S', l', M does not depend on a specific block $X[h]$, but only on the values of ES, LS, EE, LE: in this way, if another block $X[h']$ has the same matching region, we reuse those values without recomputing S', l', M.

We then compute u' as explained above (its value depends instead on $X[h]$). If l' is greater than u there is an inconsistency: the candidate lower bound for the refined block(s) X'_h is greater than the upper bound of $X[h]$ (e.g., by equating $X = \{a\}^{1,1}\{x\}^{1,1}\{b\}^{1,1}\Sigma^{1,1}\{z\}^{1,1}\{y\}^{1,1}\{z\}^{1,1}$ and $Y = \Sigma^{1,3}\{x\}^{1,1}\{y\}^{1,1}\Sigma^{0,3}$ we have that $l' = 2 > 1 = u$ for block $X[4]$). In lines 19–22 we refine $X[h]$ according to the values of l', u' by filtering each base of the feasible region with S'. Finally, we normalise the resulting dashed string $X' = X'[1] \cdots X'[n]$ and we return it.

Let us conclude the section by giving some important properties of Refine. Lemma 16 states that if Refine returns $\perp$ then X and Y are not equatable. Lemma 17 proves that Refine(X, M) always return a dashed string $X' \sqsubseteq X$,

```
function Refine($X = [X_1, \dots, X_n], (ESP, LSP, EEP, LEP)$)
    $ES \leftarrow LS \leftarrow EE \leftarrow LE \leftarrow \bot, \quad X' \leftarrow X$
    for $h \in \{1, \dots, n\}$ do
        if $\|X[h]\| = 1$ then                                        ▷ Known block
            continue
        if $(ES, LS, EE, LE) \neq (ESP[h], LSP[h], EEP[h], LEP[h])$ then
            $ES \leftarrow ESP[h],\ LS \leftarrow LSP[h],\ EE \leftarrow EEP[h],\ LE \leftarrow LEP[h]$
            $S_1^{l_1,u_1} \cdots S_k^{l_k,u_k} \leftarrow [ES, LE]$                  ▷ Feasible region
            $S_i^{l_i,u_i} \cdots S_j^{l_j,u_j} \leftarrow [LS, EE]$                  ▷ Mandatory region
            $S' \leftarrow \bigcup_{p=1}^{k} S_p, \quad l' \leftarrow \sum_{p=i}^{j} \nabla_{S,S_j}(l_p)$
            if $l' > 0 \ \wedge \ l' \geq l$ then
                $L \leftarrow \left(\bigcup_{p=1}^{i-1} S_p\right)^{0,\sum_{p=1}^{i-1} u_p}$
                $R \leftarrow \left(\bigcup_{p=j+1}^{k} S_p\right)^{0,\sum_{p=j+1}^{k} u_p}$
                $M \leftarrow \left[L, S_i^{l_i,u_i}, \dots, S_j^{l_j,u_j}, R\right]$
        $S^{l,u} \leftarrow X[h]$
        if $u < l'$ then
            return $\bot$
        $u' \leftarrow \sum_{p=1}^{k} \nabla_{S,S_p}(\min(u_p, u - l' + l_p))$
        if $l' = 0 \ \vee \ l' < l \ \vee \ u' > u$ then
            $X'_h \leftarrow (S \cap S')^{\max(l,l'),\min(u,u')}$             ▷ Crushing into a single block
        else
            $X'_h \leftarrow \left[(S \cap S_p)^{l_p,\min(u_p,u-l'+l_p)} \mid S_p^{l_p,u_p} = M[p],\ p = 1, \dots, |M|\right]$
    return norm($X'_1, \dots, X'_n$)
```

Figure 12: Refine algorithm for refining a block given its matching regions.

while Lemma 18 demonstrates the soundness of the refinement, i.e., that no concrete string is "lost" when refining a block.

Lemma 16. Let $X, Y \in \overline{\mathbb{DS}}_\Sigma^\lambda$ such that $M = \text{Sweep}(X, Y) \neq \bot$. Then, $\text{Refine}(X, M) = \bot$ implies $\gamma(X) \cap \gamma(Y) = \emptyset$.

Proof. See Appendix.

Lemma 17. Let $X, Y \in \overline{\mathbb{DS}}_{\Sigma}^{\lambda}$ with $n = |X|$ such that $M = \textsc{Sweep}(X, Y) \neq \bot$ and $X' = \textsc{Refine}(X, M) \neq \bot$. Then, $X' \sqsubseteq X$.

Proof. See Appendix.

Lemma 18. Let $X, Y \in \overline{\mathbb{DS}}_{\Sigma}^{\lambda}$ with $n = |X|$ such that $M = \textsc{Sweep}(X, Y) \neq \bot$, and let $X' = \textsc{Refine}(X, M) \neq \bot$. For each $z = z_1 \cdots z_n \in \gamma(X) \cap \gamma(Y)$ such that $z_i \in \gamma(X[i])$ for $i = 1, \ldots, n$ we have that $z_i \in \gamma(X'_i)$ where X'_i is the refinement of block $X[i]$.

Proof. See Appendix.

Unlike Sweep, as proven in Lemma 19, the worst-case complexity of Refine is more than linear in the number of blocks. This can happen in a scenario where, for $h = 1, \ldots, n$, there is a *maximal overlap* between the matching region R_h of $X[h]$ and the matching region $R_{h+1} \neq R_h$ of the next block $X[h + 1]$. Consider for example the equation of $X = \{a_1\}^{1,n}\{a_2\}^{1,n} \cdots \{a_n\}^{1,n}$ and $Y = \{a_1, \ldots, a_n\}^{1,n}\{a_2, \ldots, a_n\}^{1,n} \cdots \{a_n\}^{1,n}$ with $a_1 < a_2 < \cdots < a_n$. We have that $\mathit{feas}_Y(X[1]) = Y[(1, 0), (2, 0)]$, $\mathit{feas}_Y(X[2]) = Y[(1, 1), (3, 0)]$, ..., $\mathit{feas}_Y(X[i]) = Y[(1, n - 1), (n + 1, 0)]$. This means that for $h = 1, \ldots, n$ each block $X[h]$ requires performing $O(h)$ set unions to compute S', so in total we perform a quadratic number of set unions. We can generalise this example for $n \neq m$ (e.g., $Y = \{a_1, \ldots, a_m\}^{0,m}\{a_2, \ldots, a_m\}^{0,m} \cdots \{a_m\}^{0,m}$ with $n < m$). However, this scenario is really a corner case that rarely happens in practice.

Lemma 19. Let $X, Y \in \overline{\mathbb{DS}}_{\Sigma}^{\lambda}$ and $M = \textsc{Sweep}(X, Y) \neq \bot$. The worst-case complexity of Refine (X, M) is:

$$O((D + U)nm + Im)$$

where $n = |X|$, $m = |Y|$, and $O(D), O(I), O(U)$ are respectively the worst-case complexity of checking set disjointness, computing set intersection, and set union between a base of X and a base of Y.

Proof. See Appendix.

4.1.3. Properties of Equate

```
function Equate($X = [X_1, \dots, X_n], Y = [Y_1, \dots, Y_m]$)
    $M \leftarrow$ Sweep($X, Y$)
    if $M = \bot$ then
        return $\bot$
    $X' \leftarrow$ Refine($X, M$)
    if $X' = \bot$ then
        return $\bot$
    $M \leftarrow$ Sweep($Y, X'$)
    if $M = \bot$ then
        return $\bot$
    $Y' \leftarrow$ Refine($Y, M$)
    if $Y' = \bot$ then
        return $\bot$
    return $(X', Y')$
```

Figure 13: equate algorithm.

The overall Equate algorithm is given in Figure 13. Unsurprisingly, the worst-case complexity of Equate is dominated by the cost of Refine (see Theorem 3). However, note that: *(i)* in this context it is totally reasonable to assume the cost of set operations constant, because for most applications the bases of the blocks are either bounded by a small constant (i.e., ASCII alphabet) or convex (so, a single range) — we can thus consider the worst case complexity $O(m^2 + nm)$ (see Corollary 1); *(ii)* in most cases we do not have a maximum overlap between matching regions, so the complexity of Equate is often linear in the number of blocks: $O(n + m)$.

Theorem 3. Let $X, Y \in \overline{\mathbb{DS}}^{\lambda}_{\Sigma}$. The worst-case complexity of equate(X, Y) is:

$$O((D + U)(m^2 + nm) + I(n + m))$$

where $n = |X|$, $m = |Y|$, and $O(D), O(I), O(U)$ are respectively the worst-case complexity of checking set disjointness, computing set intersection, and set union between a base of X and a base of Y.

Proof. See Appendix.

Corollary 1. Let $X, Y \in \overline{\mathbb{DS}}_{\Sigma}^{\lambda}$. If $n = |X|$, $m = |Y|$, and the worst-case complexity of checking set disjointness, computing set intersection, and set union between a base of X and a base of Y is $O(1)$, then the worst-case complexity of $\textsc{equate}(X, Y)$ is $O(m^2 + nm)$.

Proof. See Appendix.

Theorem 4 proves the *soundness* of Equate, i.e., that Equate meets the definition of equation algorithm.

Theorem 4. Equate is an equation algorithm, i.e., for each $X, Y \in \overline{\mathbb{DS}}_{\Sigma}^{\lambda}$:

(i) if $\textsc{Equate}(X, Y) = \bot$, then $\gamma(X) \cap \gamma(Y) = \emptyset$

(ii) if $\textsc{Equate}(X, Y) = (X', Y')$, then (X', Y') refines (X, Y)

Proof. See Appendix.

At present, the completeness of Equate (i.e., if $\gamma(X) \cap \gamma(Y) = \emptyset$ implies $\textsc{Equate}(X, Y) = \bot$ for any $X, Y \in \overline{\mathbb{DS}}_{\Sigma}^{\lambda}$) is still an open problem.[1] However, this is not a problem because we can prove the *eventual completeness* of Equate, i.e., if all the blocks of X and Y have fixed cardinality then $\textsc{Equate}(X, Y)$ is always able to detect when $\gamma(X) \cap \gamma(Y) = \emptyset$. This property is important because it basically says that a dashed string solver does not have to wait until X and Y becomes known ($|\gamma(X)| = |\gamma(Y)| = 1$) to detect that $\gamma(X) \cap \gamma(Y) = \emptyset$: a sufficient condition is that $lb(X[i]) = ub(X[i])$ and $lb(Y[j]) = ub(Y[j])$ for $i = 1, \ldots, |X|$ and $j = 1, \ldots, |Y|$ as proven in Theorem 5.

[1] We evaluated the complete Cover algorithm used in [15, 16] on several different benchmarks — including those evaluated in this paper — to look for a counterexample where $\textsc{Cover}(X, Y) = \bot \wedge \textsc{Equate}(X, Y) \neq \bot$. However, this situation did not occur in any of the problems we considered.

Theorem 5. Let $X, Y \in \overline{\mathbb{DS}}^{\lambda}_{\Sigma}$. If $lb(X[i]) = ub(X[i])$ and $lb(Y[j]) = ub(Y[j])$ for $i = 1, \ldots, |X|$ and $j = 1, \ldots, |Y|$ then:

$$\textsc{Equate}(X, Y) = \bot \iff \gamma(X) \cap \gamma(Y) = \emptyset.$$

Proof. See Appendix.

Unfortunately, although sound and eventually complete, Theorem 6 proves that Equate is not optimal and not idempotent. However, because Equate is called several times during propagation, we realised that it is usually far better to use a "lightweight" algorithm like Equate, often linear in the number of blocks, rather than a more precise but computationally expensive approach (e.g., the dynamic programming algorithm used in [15, 16]). In particular, even if a fixpoint always exists (see Proposition 2) it does not appear convenient to iterate Equate until $\textsc{Equate}(X, Y) = (X, Y)$.

Theorem 6. The equation algorithm Equate is:

(i) *not optimal*: $\textsc{Equate}(X, Y)$ is not always minimal element of $(\mathit{Ref}_{X,Y}, \sqsubseteq)$

(ii) *not idempotent*: $\textsc{Equate}(X, Y) = (X', Y')$ does not always imply that $\textsc{Equate}(X', Y') = (X', Y')$.

Proof. See Appendix.

5. String Constraint Solving

The dashed string formalism introduced in Section 3 and the Equate algorithm described in Section 4 constitute the foundation on which we build our string constraint solving model.

Given a finite alphabet Σ and a maximum string length λ, let $\mathcal{P} = \langle \mathcal{X}, \mathcal{D}, \mathcal{C} \rangle$ be a CSP containing a number $k > 0$ of *string variables* $\mathcal{X}_{str} = \{x_1, \ldots, x_k\} \subseteq \mathcal{X}$ such that $D(x_i) \subseteq \Sigma^*$ and $|x_i| \leq \lambda$ for $i = 1, \ldots, k$. We will model each domain $D(x_i)$ with a dashed string $DS(x_i) \in \overline{\mathbb{DS}}^{\lambda}_{\Sigma}$ such that $D(x_i) \subseteq \gamma(DS(x_i))$.

The goal is to find, if it does exist, an assignment $\xi \in D(x_1) \times \cdots \times D(x_n)$ of domain values to the corresponding variables satisfying all the constraints of $\mathcal{C}$. In particular, we have to narrow the domain $DS(x)$ of each string variable $x \in \mathcal{X}_{str}$ until $DS(x)$ becomes known (or we detect that $\mathcal{P}$ is unsatisfiable). To do so, we have to define suitable *propagators* for the string constraints, i.e., for each constraint of $\mathcal{C}$ involving at least a variable of $\mathcal{X}_{str}$.

As we shall see, most of the propagators we defined are based on the Equate algorithm. However, for a few others we had to devise different algorithms. In the following we will provide an overview of the propagators we defined, covering virtually all the most common string constraints. One thing to note is that it is hard to define and maintain consistency notions for dashed string propagators because, as seen in Section 3, dashed strings do not form a lattice w.r.t. the $\sqsubseteq$ relation and thus there is not always a best approximation for a set of concrete strings. Moreover, it is not trivial to define incremental propagators because in general dashed strings can both shrink and expand due to normalization, hence a position (i, j) for a dashed string X may refer to different parts of X in subsequent calls of the propagator.

5.1. Propagation Equate*-based*

The Equate algorithm is the core of most of the propagators we defined. This is also due to the nature of dashed strings, that naturally handles well-known string operations such as (iterated) concatenation, indexing and reverse.

For conciseness, for any integer variable x, we define $lb(x) = \min(D(x))$ and $ub(x) = \max(D(x))$. We will always assume that $\textsc{Equate}(X, Y) \neq \bot$ (otherwise, the problem is unsatisfiable).

5.1.1. Equality, disequality, and reified equality

Given string variables $x, y \in \mathcal{X}_{str}$ the propagation of the equality constraint $x = y$ is straightforward: we compute $(X', Y') = \textsc{Equate}(DS(x), DS(y))$ and we update the corresponding domains $DS(x) \leftarrow X', DS(y) \leftarrow Y'$. Similarly, we handle the domain constraint $x :: D$ where $D \in \overline{\mathbb{DS}}_{\Sigma}^{\lambda}$ is a dashed string constant representing a set of concrete strings.

Apart from some corner cases, the disequality constraint $x \neq y$ does not involve any pruning: we basically just wait until x, y become known. However, we can detect when the constraint is redundant or *subsumed*. We define a predicate $\textsc{CheckEquate}(X, Y) = (\textsc{Sweep}(X, Y) \neq \bot \wedge \textsc{Sweep}(Y, X) \neq \bot)$, so $x \neq y$ is subsumed if $\textsc{CheckEquate}(DS(X), DS(Y))$ is false. $\textsc{CheckEquate}$ is actually a lightweight version of $\textsc{Equate}$ operating in $\mathrm{O}(|X| + |Y|)$: we do not perform any refinement, we just check if we can show that $\gamma(X) \cap \gamma(Y) = \emptyset$.

$\textsc{CheckEquate}$ is also helpful for *reified* equality: $b \iff x = y$ where $b \in \mathcal{X}$ is a Boolean variable. If $b = \textit{true}$ ($b = \textit{false}$), then we rewrite the constraint into $x = y$ ($x \neq y$). If $\textsc{CheckEquate}(x, y) = \textit{false}$, we update $D(b) \leftarrow \textit{false}$. Similarly we handle the *half-reifications*: $b \implies x = y$ and $x = y \implies b$.

5.1.2. Concatenation, iterated concatenation, and reverse

A key property of dashed strings is that we can easily concatenate them: if $X = S_1^{l_1,u_1} \cdots S_n^{l_n,u_n}$ and $Y = {S'}_1^{l'_1,u'_1} \cdots {S'}_m^{l'_m,u'_m}$ we can define as $XY = S_1^{l_1,u_1} \cdots S_n^{l_n,u_n} {S'}_1^{l'_1,u'_1} \cdots {S'}_m^{l'_m,u'_m}$. Note that this operation is *not closed* in $\mathbb{DS}_{\Sigma}^{\lambda}$ if $\sum_{i=1}^{n} l_i + \sum_{j=1}^{m} l'_j > \lambda$, and that if $S_n = {S'}_1$ then XY is *not normalised*. However, we must not normalise when propagating $z = xy$, because we do not want to merge the domains of x and y.

The concatenation constraint $z = xy$ is propagated by computing $(Z', Q) = \textsc{Equate}(DS(z), DS(x)DS(y))$ first. Then, if $Q = Q[1] \cdots Q[h] \cdots Q[k]$ where the first h blocks come from the refinement of the X-blocks and the last $k - h$ are relative to the Y-blocks, we can update the domains: $DS(x) \leftarrow Q[1] \cdots Q[h]$, $DS(y) \leftarrow Q[h+1] \cdots Q[k]$, $DS(z) \leftarrow Z'$. Note that the concatenation, together with the string length and the domain constraints, allows us to decompose the substring constraint $y = x[i..j] = x[i]x[i+1] \cdots x[j]$ into basic constraints (in particular, this is useful to select the i-th element $x[i]$ of a string x).

The approach for *iterated* concatenation is similar. Given $X \in \overline{\mathbb{DS}}_{\Sigma}^{\lambda}$, we define $X^0 = \emptyset^{0,0}$ and $X^k = XX^{k-1}$ for each $k \in \mathbb{N}$. However, handling the constraint

$y = x^n$ where $n \in \mathcal{X}$ is an integer variable adds a level of complexity because we do not know how many times x must be repeated. Given integers l, u with $0 \leq l \leq u$ and $X = S_1^{l_1,u_1} \cdots S_k^{l_k,u_k}$, we define $X^{l..u} = X^l(S_1 \cup \cdots \cup S_k)^{0,(u-l)\sum_{i=1}^k u_i}$. In practice we repeat X for l times, and then we approximate the remaining (optional) $u - l$ repetitions with a nullable block. We then propagate $y = x^n$ by computing $\textsc{Equate}(DS(x)^{lb(n)..ub(n)}, DS(y))$.

Given $X = S_1^{l_1,u_1} \cdots S_k^{l_k,u_k}$, we define its *reverse* as $X^{-1} = S_k^{l_k,u_k} \cdots S_1^{l_1,u_1}$. In this way, handling the string reverse constraint $y = x^{-1}$ is straightforward: $(DS(x), DS(y)) \leftarrow \textsc{Equate}(DS(x)^{-1}, DS(y))$. This propagator is useful to detect palindrome strings for which $x = x^{-1}$.

5.1.3. Find and replace

A widely used string operation is finding a sub-string x into a query string y. The constraint $i = \textsc{find}(x, y)$ holds if i is the index of the *first* occurrence of x in y (assuming $i = 0$ if x is not substring of y). The propagator for FIND does not explicitly make use of EQUATE, but implements a variant for determining the earliest and the latest start of x in y. The domain of i is refined accordingly, thanks to suitable functions to convert between indexes of $D(i)$ and corresponding positions in the dashed string $DS(y)$. For more details about the propagation of FIND with dashed strings we refer the reader to [17].

From FIND we can derive a number of other constraints. For example, we may want to find the *last* occurrence of x in y, or to determine is x is a prefix, a suffix or in general a sub-string of y. Another common operation is the string *replacement*: the constraint $y = \textsc{replace}(x, x', y)$ holds if y is the string resulting from replacing x with x' in y (if x is not substring of y, then $y' = y$). As detailed in [17], this constraint can be rewritten into basic string constraints including FIND and concatenation. In [**?**], a dedicated dashed string propagator has been defined for REPLACE and REPLACE-ALL (which replaces *all* the occurrences of a query string in a target string).

5.1.4. Other constraints

There are a number of other string constraints relying on Equate algorithm. For example, the conversion to lower/upper case characters, or the channelling to numbers. The general idea is to approximate a set of concrete strings with a dashed string (eventually by crushing into single blocks) and then to use Equate to possibly refine the domain. For example, to convert an integer variable $n \in [50, 1200]$ to the corresponding string x, we can define a dashed string $N = \{0\}^{\lambda-2}\{1\}^{0,1}\{0,\dots,9\}^{3,3}$ (we admit leading zeros) over-approximating $D(n)$ and apply $\textsc{Equate}(DS(x), N)$ to possibly refine the domain of x.

In particular, Equate is also used to propagate the well-known element global constraint stating that $x = A[i]$, where x a string variable, A is an array of n string variables and $i \in [1, n]$ is an integer variable. element is particularly useful to propagate logical disjunctions of the form $x = x_1 \vee \cdots \vee x = x_n$, that are rewritten into $x = [x_1, \dots, x_n][i] \wedge i :: [1, n]$. In this case, we can use CheckEquate to remove an index j from $D(i)$ if $\textsc{CheckEquate}(DS(x), DS(x_j)) = \mathit{false}$. We can also derive a dashed string X over-approximating all the $DS(x_j)$ such that $j \in D(i)$, and then use $\textsc{Equate}(DS(x), X)$ to refine $DS(x)$ (in the worst case, we crush all the dashed strings $DS(x_j)$ into a single block as seen in Section 4).

5.2. Propagation not based on Equate

We conclude the section by showing some string constraints whose propagation is not based on the Equate algorithm.

5.2.1. Length

A suitable propagation for the string length constraint is fundamental for string solving. If $X = DS(x)$, we propagate $n = |x|$ by basically propagating the constraint $n = n_1 + \cdots + n_k$ where $k = |X|$ and each n_i is an integer variable having domain $[lb(X[i]), ub(X[i])]$ for $i = 1, \dots, k$. For example, if $DS(x) = \{a\}^{1,2}\{b\}^{0,3}\{a\}^{1,2}$ and $D(n) = [0, 2]$ the constraint $|x| = n$ refines $DS(x)$ into $\textsc{norm}(\{a\}^{1,1}\{b\}^{0,0}\{a\}^{1,1}) = \{a\}^{2,2}$. This in turn sets $n \leftarrow 2$.

String length information is very useful for rewriting string constraints and to strengthen their propagation. For example, we have that $x = y \Rightarrow |x| = |y|$, $z = xy \Rightarrow |z| = |x| + |y|$, $y = x^n \Rightarrow |y| = n|x|$, and $y = x^{-1} \Rightarrow |y| = |x|$.

5.2.2. Lexicographic ordering

Given a total order $<$ over the characters of Σ, the lexicographic ordering $\preceq$ over Σ^* is given by:

$$x \preceq y \iff x = y \ \vee \ \exists i \in [1, |x|].\, (x[i] < y[i] \ \wedge \ (\forall j \in [1, i-1] : x[j] = y[j]).$$

Given that $|x|$ is generally unknown and potentially very big, decomposing $\preceq$ into basic constraints can be highly inefficient. For this constraint, neither the Equate algorithm nor the string length information are helpful. We thus devised a propagator that basically enforces the unary constraints $x \preceq \max_{\preceq} \gamma(DS(y))$ and $\min_{\preceq} \gamma(DS(x)) \preceq y$.

Care must be taken because in the lexicographic minimum (resp., maximum), each block $S^{l,u}$ takes its least (resp., greatest) character, and *either* its minimum cardinality l *or* its maximum cardinality u: but which cardinality is chosen depends on the following blocks, i.e., the *suffix* of the string. For more details about the propagation of the lexicographic ordering with dashed strings we refer the reader to [17].

5.2.3. Regular membership

The constraint $x \in \mathcal{L}(R)$ enforces x to belong to the regular language $\mathcal{L}(R)$ denoted by automata R. This constraint occurs frequently in problems derived from security analysis and model checking. The CP global constraint regular(X, R) is not nicely applicable in the string solving context because it relies on a *fixed-length*[1] sequence X of integer variables (see [25, 26, 27, 28]), and therefore is strongly coupled to the maximum string length λ.

[1] The dashed string approach is instead *bounded-length*: each string variable x has length $|x| \leq \lambda$, while for fixed-length approaches $|x|$ is always fixed.

In [18] an approach to propagate the reified version $b \iff x \in \mathcal{L}(R)$ with dashed string has been proposed (this is more general, and enables to deal with constraints like $x \notin \mathcal{L}(R)$ or `if` $x \in \mathcal{L}(R)$ `then` $A(x)$ `else` $B(x)$, frequently arising from program analysis). This approach essentially works in two steps: *(i)* a *forward* pass, where the set of reachable states is computed (potentially detecting inconsistency); *(ii)* a *backward* pass, where only feasible end-states are considered and the domains of the variables are possibly pruned.

When dealing with problems derived from program analysis, often the regular membership constraint is given in terms of a *regular expression* r. Clearly we can always transform r into an equivalent automaton. However, if r is simple enough, it is beneficial to *decompose* it into basic string constraints. For example, $x \in L((\texttt{fee}|\texttt{foo})\texttt{bar})$ can be reformulated into $x = yz \wedge y \in \{\texttt{fee}, \texttt{foo}\} \wedge z = \texttt{bar}$. We use simple syntactic rules of the form $x \in L(r) \models C_1 \wedge \ldots \wedge C_k$ to identify opportunities for decomposing $x \in L(r)$ into a conjunction $C_1 \wedge \ldots \wedge C_k$ of basic string constraints.

For more details about the propagation and the decomposition of regular membership with dashed strings, we refer the reader to [18].

5.3. Branching

The propagators described above are *incomplete*, because they not remove all the infeasible values from the variables' domains. This also means that, when the propagation reaches a fixpoint without returning a solution or an inconsistency, we have to search for a solution by *branching* on the string variables.

Branching is important for string solving because the size of the string domains is often very big. We adopt the following hierarchical strategy. We first select a variable $x \in \mathcal{X}$ with a given *variable heuristic*. Then, we use a *length heuristic* to fix the length of x, i.e., we branch on $|x| = n$ or $|x| \neq n$ with $n \in D(|x|)$. Fixing the string length is important for boosting the propagation and for the early detection of unsatisfiability (see Theorem 5).

Once $|x|$ is fixed, we use a *block heuristic* to choose a block $S^{l,u}$ of $DS(x)$ (this is not a branch point), a *cardinality heuristic* to set its cardinality to a

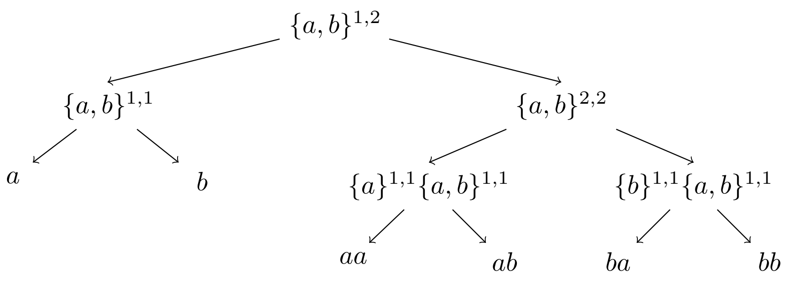

Figure 14: Example of search tree.

value $k \in \{l, u\}$ (we branch on $S^{k,k}$ or $S^{l+b,u-(1-b)}$ with $b = 1$ if $k = l$, $b = 0$ otherwise), and finally a *base heuristic* to select a value $a \in S$ (we branch on $\{a\}^{1,1}S^{k-1,k-1}$ or $(S - \{a\})^{1,1}S^{k-1,k-1}$). By iterating this procedure, we either get a solution or detect the unsatisfiability of the problem.

For example, a reasonable variable heuristic can be to choose the variable x with minimum domain dimension $\|DS(x)\|$ or the minimum length $|x|$. For most cases, it makes sense to set $|x|$ to its minimum value (i.e., searching for a minimum-length solution). Note that the above strategy always work "from left to right", i.e., the base heuristic incrementally assigns the *prefix* of x. One may want to introduce yet another heuristic to determine a "split point" $h \in [1, k]$ for $S^{k,k}$ in order to branch on $S^{h-1,h-1}\{a\}^{1,1}S^{k-h,k-h}$ or $S^{h-1,h-1}(S-\{a\})^{1,1}S^{k-h,k-h}$. However, to avoid a further level of complexity, we always assume $h = 1$.

For instance, to enumerate all the values of a string domain $D(x)$ in *shortlex order* (first by length, then lexicographically) we can adopt a strategy that: *(i)* selects the minimum length for x; *(ii)* chooses the leftmost unknown block $S^{l,u}$ of $DS(x)$; *(iii)* sets its cardinality to the lower bound l; *(iv)* branches first on $\{a\}^{1,1}S^{l-1,l-1}$ where $a = \min(S)$. Consider for example Figure 14, where we show the search tree for a string variable x with domain $DS(x) = \{a, b\}^{1,2}$.

6. Implementation

String constraint solving and dashed strings are fairly new concepts in Constraint Programming. It is therefore not surprising that their implementations are still at an early stage. However, some prototypical tools already exist. In this section we give an overview of them.

6.1. MiniZinc with Strings

A number of modelling languages have been proposed to encode CP problems into a format understandable by constraint solvers, that is, for translating a mathematical formulation $\mathcal{P} = (\mathcal{X}, \mathcal{D}, \mathcal{C})$ into a format processable by a solver.

Although the CP community has not yet agreed on a standard language, one of the most popular nowadays is *MiniZinc* [29]. This language is solver-independent (the motto is *"model once, solve anywhere"*) and enables the separation between model and data, i.e., a MiniZinc model can be defined as a generic template to be instantiated by different data.

Each MiniZinc model (together with corresponding data, if any) is translated into *FlatZinc*—the solver-specific target language for MiniZinc—in the form required by a solver. MiniZinc also allows one to define annotations to communicate with the underlying solver. From the same MiniZinc model, different FlatZinc instances can therefore be derived.

In order to deal with string solving, MiniZinc was equipped with *string variables* and constraints in [14]. A MiniZinc model with strings can be solved "directly" by CP solvers natively supporting string variables (Gecode+$\mathbb{S}$ [13] and G-Strings [16]) or "indirectly" via a static unfolding into integer variables. Basically, if $\Sigma = \{a_1, \dots, a_n\}$ each string variable x in the MiniZinc model is translated into an array A_x of λ integer variables such that $A_x[i] \in [0, n]$ and $A_x[i] = 0 \iff i > |x|$ for $i = 1, \dots, \lambda$. For each constraint over string variables $x_1, \dots, x_k$, we define a corresponding constraint over $A_{x_1}, \dots, A_{x_k}$. Clearly, this unfolding is generally inefficient—especially as λ grows. Moreover, some string constraints are not nicely expressible without string variables. For example, as seen in [17], it is not straightforward to translate the constraint $i = \textsc{find}(x, y)$ because it is not trivial to express that x cannot occur in y before index i.

6.2. G-Strings

G-Strings [19] is the first implementation of a dashed strings approach in Constraint Programming. This tool is an extension of Gecode [30], a mature CP solver over finite domains written in C++, with string variables.

G-Strings is a *copying solver*, i.e., during the search the domains are copied (and possibly restored) before a choice is committed. In this context the memory management becomes critical. Note that implementing a trailing solver with dashed string is not trivial, because dashed string normalization and the non-lattice nature of dashed strings make hard to record for each state-changing operation the information necessary to undo its effect.

G-Strings mainly relies on the Gecode built-in data structures: a dashed string is currently implemented as a `DynamicArray` of blocks, where the base of each block is a `BndSet` representing finite set of integers as unions of disjoint ranges (see [30] for more details about these data structures). Each dashed string object internally stores the length information.

G-Strings implements the Equate algorithm and all the propagators listed in Section 5. It also supports the aforementioned string extension of MiniZinc language. Despite being still a prototype, the empirical evaluations performed in [15, 16, 17, 18, 31] against state-of-the-art string solvers witness the effectiveness of the dashed string approach.

Clearly, G-Strings has great room for improvements in terms of both software engineering and new features. For example, *trailing* might be more efficient than copying during the search: rather than cloning dashed strings we keep track of their changes. Furthermore, it would be very interesting to implement *learning* features. This is not trivial because a dashed string continually changes its shape during the search: it can both tighten (via normalisation) or widen (via branching or refinement).

6.3. Aratha

In [31] a *dynamic symbolic execution* (DSE) tool for the analysis of JavaScript programs called Aratha [32] has been proposed.

DSE, also known as concolic execution/testing, or directed automated random testing [33, 34], is a hybrid technique that integrates the *dynamic*, or concrete, execution of a program (i.e., testing a particular input value) together with its *symbolic* execution [35] (i.e., testing a symbolic, or abstract, input value).

In a nutshell, DSE collects the constraints (or *path conditions*) encountered at conditional statements during the concrete execution. Then, a constraint solver is used to detect alternative execution paths by systematically negating the path conditions. This process is repeated until all the feasible paths are covered or a given limit is exceeded.

A key factor for the success of DSE is obviously the efficiency and the expressiveness of the underlying constraint solver(s). In particular, the nature and the applications of JavaScript language requires its DSE to process and solve different string constraints. This means that efficient string solvers are needed.

Aratha allows one to use both SMT and CP solvers to solve path conditions, and in particular it enables the use of G-Strings through the generation of MiniZinc path conditions. The empirical evaluation in [31] proves that the dashed string based approach can be easily competitive with the SMT approaches, and in particular that both these techniques can be used in conjunction.

7. Evaluation

The dashed string approach has already been empirically validated in [15, 16, 17, 18]. Most of the benchmarks used in those experiments are derived from real-world applications (in particular analysis of web applications).

In this paper we perform a new evaluation on the *StringFuzz* benchmarks [36]. No dashed string approach has been tested before on these benchmarks. *StringFuzz* is a problem generator and fuzzer for SMT string solvers. It also contains a repository of several SMT-LIB 2.0/2.5 string and regex instances generated and transformed with the fuzzer. We conducted our evaluation on the suite of 1065 generated instances introduced in [36]. Table 1 summarizes the nature of

Table 1: *StringFuzz* benchmarks composition

Class	Description	Quantity
Concats-{Small,Big}	Right-heavy, deep tree of concats	120
Concats-Balanced	Balanced, deep tree of concats	100
Concats-Extracts-{Small,Big}	Single concat tree, with character extractions	120
Lengths-{Long,Short}	Single, large length constraint on a variable	200
Lengths-Concats	Tree of fixed-length concats of variables	100
Overlaps-{Small,Big}	Formula of the form $ax = xb$	80
Regex-{Small,Big}	Complex regex membership test	120
Many-Regexes	Multiple random regex membership tests	40
Regex-Deep	Regex membership test with many nested operators	45
Regex-Pair	Test for membership in one regex, but not another	40
Regex-Lengths	Regex membership test, and a length constraint	40
Different-Prefix	Equality of two deep concats with different prefixes	60

these instances, grouped into twelve different classes.[1]

To make the evaluation as comprehensive as possible, we evaluated all the state-of-the-art string solving approaches we could. Precisely, we compared the performance of G-Strings against the following SMT solvers:

- CVC4 [10], an efficient SMT solver supporting several theories, including the theory of strings;
- Z3str3 [9], the primary string solver used by the well-known SMT theorem prover Z3 [37];
- Z3seq, a Z3 plug-in for the theory of sequences, that can also be used for string solving;
- Norn [38], a solver for word equations over unbounded length string variables and membership constraints;

We also attempted to run other solvers. For example, the S3# solver [39] cannot properly process the SMT-LIB format and therefore could not be tested. The

[1] See `http://stringfuzz.dmitryblotsky.com/suites/generated` for more details.

same applies for the recently introduced Trau [40] solver. We also ran the Sloth [41] solver, but we realised that it is unstable on these benchmarks (about 160 incorrect answers).

As already observed in [18] the other CP string solver, namely Gecode+$\mathbb{S}$ [13], is no longer actively developed: it cannot support some classes of constraints (e.g., negated regular expressions) and its SMT-LIB/FlatZinc interfaces are incomplete. We do not include it in the evaluation because it could solve only 304/1065 instances, and for these instances it was never the overall best solver.

We did not include in the evaluation any automata-based approach because we are not aware of any state-of-the-art automata-based string solver able to process the *StringFuzz* benchmarks.

Unlike SMT solvers, G-Strings needs a maximum string length λ. We therefore evaluated different versions of G-Strings with $\lambda = 1000, 5000, 10000$. For brevity, we will call them G-Str$_{1k}$, G-Str$_{5k}$, G-Str$_{10k}$ respectively. The default branching strategy used by G-Strings is the following: we first select the string variable x minimising $\|X[k]\|$, where $X = DS(x)$ is the domain of x and $X[k]$ is the *leftmost unknown block* of X. We first fix $|x|$ to its minimum length, i.e., $|x| = \sum_{i=1}^{|X|} lb(X[i])$. Then, if $X[k] = S^{l,u}$, we branch on $\{a\}^{1,1}S^{l-1,l-1}$ where a is the minimal lexicographic element of the set S' defined as:

$$S' = \begin{cases} S \cap T & \text{if } S \cap T \neq \emptyset \\ S & \text{otherwise} \end{cases}$$

where T is a set of characters that *must* appear in each solution of the problem (we compute T only once during the first round of propagation by considering the string constants and the known blocks of each domain).

To process the *StringFuzz* instances, we developed a suitable compiler from SMT-LIB to MiniZinc with strings, and then we converted the resulting MiniZinc models into corresponding FlatZinc instances. We set with a timeout of $T = 300$ seconds for each problem instance.

Table 2: Overall results.

Solver	SAT	UNS	OUT	INC	UNK	%Solved	Avg. Time	Borda
Norn	582	107	149	0	227	64.69	109.42	523.71
Z3seq	494	218	310	0	43	66.85	113.07	1390.47
CVC4	645	266	154	0	0	85.54	46.08	3152.29
Z3str3	680	240	123	0	22	86.38	44.29	2305.41
G-Str$_{1k}$	682	**292**	0	78	13	91.46	25.85	4447.66
G-Str$_{5k}$	**760**	**292**	0	0	13	**98.78**	**4.00**	**4744.94**
G-Str$_{10k}$	**760**	**292**	0	0	13	**98.78**	4.36	4676.70
VBS	*761*	*302*	*2*	*0*	*0*	*99.81*	*0.62*	*N/A*

The overall results[1] are shown in Table 2, where we report: the number of satisfiable (SAT) and unsatisfiable (UNS) instances solved; the number of timeouts (OUT), incorrect (INC) and unknown (UNK) answers — UNK is set if a solver terminates its execution before the timeout without saying SAT nor UNS (e.g., due to memory exhaustion); the percentage of solved instances; the average time (in seconds) to solve an instance — we set the solving time to the timeout T if a solver cannot solve an instance; and the total *Borda* score. The latter is a comparative measure used in the MiniZinc Challenge [42] where each pair of solvers is compared on each problem instance. Let $\mathit{time}(s,p) \in [0,T]$ be the time taken by solver s to solve problem p (we set $\mathit{time}(s,p) = T$ if s cannot solve p). For each problem p, a solver s is compared against each other solver s', and it scores: 0 points, if $\mathit{time}(s,p) = T$; 1 point, if $\mathit{time}(s,p) < T$ and $\mathit{time}(s',p) = T$; $\frac{\mathit{time}(s',p)}{\mathit{time}(s,p)+\mathit{time}(s',p)}$ points otherwise (if $\mathit{time}(s,p) = \mathit{time}(s',p) = 0$ both s and s' score 0.5 points).

The last row of Table 2 refers to the *Virtual Best Solver* (VBS), a fictitious solver such that $\mathit{time}(\text{VBS},p) = \min\{\mathit{time}(s_i,p) \mid i = 1,\dots,k\}$ given a problem p and a set of solvers $\{s_1,\dots,s_k\}$. In our benchmarks, the VBS can solve 1063

[1] All the experiments have been conducted on a Ubuntu 15.10 machine with 16 GB of RAM and 2.60 GHz Intel® i7 CPU. We run both CVC4 an Z3 with their last stable release (i.e., version 1.7 and 4.8.6 respectively

Table 3: Instance solved by class

Class	CVC4	Norn	Z3seq	Z3str3	G-Str$_{1k}$	G-Str$_{5k}$	G-Str$_{10k}$
Concats-Balanced	**100**	70	**100**	70	**100**	**100**	**100**
Concats-Big	**60**	16	**60**	14	**60**	**60**	**60**
Concats-Extracts-Big	58	0	58	**60**	**60**	**60**	**60**
Concats-Extracts-Small	27	0	58	**60**	**60**	**60**	**60**
Concats-Small	**60**	**60**	**60**	**60**	**60**	**60**	**60**
Different-Prefix	**60**	**60**	**60**	**60**	**60**	**60**	**60**
Lengths-Concats	**100**	**100**	72	**100**	**100**	**100**	**100**
Lengths-Long	**100**	**100**	10	**100**	22	**100**	**100**
Lengths-Short	**100**	**100**	**100**	**100**	**100**	**100**	**100**
Many-Regexes	**40**	**40**	**40**	**40**	**40**	**40**	**40**
Overlaps-Big	**20**	2	0	3	10	10	10
Overlaps-Small	**60**	3	0	15	**60**	**60**	**60**
Regex-Big	14	11	11	**60**	**60**	**60**	**60**
Regex-Deep	34	37	23	41	**42**	**42**	**42**
Regex-Lengths	27	**40**	9	37	**40**	**40**	**40**
Regex-Pair	5	13	13	**40**	**40**	**40**	**40**
Regex-Small	46	37	38	**60**	**60**	**60**	**60**

instances, i.e., there are two instances not solvable by any solver.[1]

There is no single dominant solver. Eight instances were solved by CVC4 only (precisely, instances of the *Overlaps-Big* class). In one case each, G-Strings and Z3str3 were the only solver to solve an instance (precisely, an instance of the *Regex-Deep* class). If we consider the solving time, with a tolerance of 0.2 seconds to mitigate the effect of negligible time differences, we have that (a version of) G-Strings is the fastest for 82 instances, Z3seq for 44 instances, Z3str3 for 26 instances, CVC4 for 10 instances.

Tables 3 and 4 show respectively the number of solved instances and the average solving time per class of problems. We can observe that G-Strings performs very well in almost all the classes. Its worse performance is given by G-Str$_{1k}$ in the *Lengths-Long* class, because here the string constraints may

[1] Namely, `regex-deep-00013-1` and `regex-deep-00014-1`.

Table 4: Average solving time by class

Class	CVC4	Norn	Z3seq	Z3str3	G-Str$_{\text{1k}}$	G-Str$_{\text{5k}}$	G-Str$_{\text{10k}}$
Concats-Balanced	0.01	9.33	**0.00**	8.66	0.01	0.01	0.01
Concats-Big	0.03	13.94	**0.00**	13.85	0.04	0.04	0.04
Concats-Extracts-Big	0.71	16.90	1.00	**0.00**	**0.00**	**0.00**	**0.00**
Concats-Extracts-Small	10.02	16.90	0.57	**0.00**	**0.00**	**0.00**	**0.00**
Concats-Small	**0.00**	0.22	**0.00**	0.66	**0.00**	**0.00**	**0.00**
Different-Prefix	**0.00**	0.07	**0.00**	**0.00**	**0.00**	**0.00**	**0.00**
Lengths-Concats	**0.01**	0.26	13.91	0.03	**0.01**	**0.01**	**0.01**
Lengths-Long	**0.00**	0.11	25.91	0.02	21.97	**0.00**	**0.00**
Lengths-Short	**0.00**	0.11	1.36	0.01	**0.00**	**0.00**	**0.00**
Many-Regexes	**0.00**	0.13	0.31	0.07	**0.00**	**0.00**	**0.00**
Overlaps-Big	**0.00**	5.07	5.63	4.92	2.82	2.83	2.89
Overlaps-Small	**0.00**	16.06	16.90	13.60	**0.00**	0.10	0.41
Regex-Big	13.42	13.82	14.43	0.08	**0.01**	**0.01**	**0.01**
Regex-Deep	3.12	2.30	6.74	1.39	**0.86**	**0.86**	**0.86**
Regex-Lengths	4.21	0.05	9.43	0.97	**0.00**	**0.00**	**0.00**
Regex-Pair	9.86	7.63	8.64	**0.01**	0.12	0.12	0.12
Regex-Small	4.68	6.53	8.24	0.01	**0.00**	**0.00**	**0.00**

have solutions with strings longer than 1000, and thus G-Str$_{\text{1k}}$ wrongly detects unsatisfiability 78 times. However, the length of these solutions is never bigger than 5000 and therefore G-Str$_{\text{5k}}$ and G-Str$_{\text{10k}}$ solve instantaneously all the instances of the *Lengths-Long* class.

In the *Overlaps-** class CVC4 is better than G-Strings because this class contains constraints of the form $ax = xb$ with $a \neq b$ and $|a| = |b|$, and the string reformulation rules for CVC4 directly reason about this class of unsatisfiability.

In these cases for G-Strings it is difficult to detect the unsatisfiability without enumerating many possible values for x of increasing length. Note that for these problems the answer is "UNK" instead of "OUT" because we exhausted the memory before the timeout. Clearly incorporating string reformulation rules into G-Strings could overcome the problems with this particular benchmark, although the benchmark is somewhat artificial.

Finally, Table 5 and Table 6 show the comparative performance of the solvers

Table 5: Number of instances where s_i solves an instance that s^j cannot.

Solvers	CVC4	Norn	Z3seq	Z3str3	G-Str$_{\text{1k}}$	G-Str$_{\text{5k}}$	G-Str$_{\text{10k}}$
CVC4	—	250	245	140	88	10	10
Norn	28	—	171	11	79	1	1
Z3seq	46	194	—	77	0	0	0
Z3str3	149	242	285	—	80	2	2
G-Str$_{\text{1k}}$	151	**364**	262	134	—	0	0
G-Str$_{\text{5k}}$	151	**364**	340	134	78	—	0
G-Str$_{\text{10k}}$	151	**364**	340	134	78	0	—

Table 6: Number of instances where s_i is faster than s^j when they both solve an instance.

Solvers	CVC4	Norn	Z3seq	Z3str3	G-Str$_{\text{1k}}$	G-Str$_{\text{5k}}$	G-Str$_{\text{10k}}$
CVC4	—	632	266	248	16	85	86
Norn	28	—	223	40	0	4	4
Z3seq	95	287	—	96	46	44	45
Z3str3	104	633	280	—	26	25	27
G-Str$_{\text{1k}}$	100	610	298	260	—	70	71
G-Str$_{\text{5k}}$	100	**683**	298	309	0	—	70
G-Str$_{\text{10k}}$	100	**683**	298	308	0	0	—

in terms of solved instances and solving time respectively. If s_i is the solver on the i-th row of the table and s^j is the solver on the j-th column, we have that the cell (i, j) of Table 5 contains the number of instances where s_i can solve an instance that s^j cannot solve, i.e., $\big|\{p \mid \mathit{time}(s_i, p) < \mathit{time}(s^j, p) = T\}\big|$. The cell (i, j) of Table 6 contains the number of instances where s_i is faster than s^j when they both solve that instance, i.e., $\big|\{p \mid \tau + \mathit{time}(s_i, p) < \mathit{time}(s^j, p) < T\}\big|$ where $\tau = 0.2$ seconds is the tolerance. These numbers also confirms the good performance of the dashed string approach.

8. Related Work

String analysis is an important emerging field, given the ubiquity of strings in different domains such as, e.g., abstract interpretation [22, 43, 44], software verification and testing [1, 2, 31, 45], model checking [3], and web security [4, 46]. Although string solvers are still in their infancy, various string solving approaches have been proposed. We can classify them into three rough families:

Automata-based. This approach uses automaton to handle string variables and related operations. Stranger [47] is a tool for detecting vulnerabilities in PHP applications that uses both string and arithmetic automata. PASS [7] combines automata and parameterised arrays for the treatment of unsatisfiable cores. StrSolve [6] constructs the search space lazily based on automata representation. PISA [8] relies on the BDD-based automaton representation of Monadic Second-order Logic formulae. The advantage of automaton is that they can handle unbounded-length strings and precisely represents infinite sets of strings. However, this approach faces performance issues due to state explosion and the integration with other domains (e.g., integers). This is the main reason why general purpose string solving is not automata-based nowadays.

Word-based. According to [48] definition, word-based solvers are SMT solvers treating strings without abstraction or representation conversions. The most known word-based solvers, mainly based on the DPLL(T) paradigm [49], are: CVC4 [10], the family of solvers Z3str [50], Z3str2 [51], and Z3str3 [9] based on the Z3 solver [37], S3 [52] and its evolutions S3P [53] and S3# [39], Norn [54]. More recent proposals are Sloth [41] and Trau [40]. These solvers allow one to reason about unbounded strings and can take advantage of already defined theories. However, most of them are incomplete and they can face scalability issues due to disjunctive case-splitting.

Unfolding-based. An unfolding approach basically selects a length bound k, and then substitute each string variable with a vector having fixed length k. String constraints can be compiled down to bit-vector constraints (e.g., Hampi [11] and Kaluza [12] solvers) or integer constraints (see the $\mathcal{F}^{int}$ FlatZinc decomposition of [14]). Gecode+$\mathbb{S}$ [13] is instead a CP-based approach that defines dedicated propagators for the string constraints. The unfolding approach can add flexibility but may sacrifice high-level relationships between strings, it cannot deal with unbounded-length strings and especially it can become very inefficient when the length bound is large—even if the generated solutions have short length.

Dashed strings could be seen as a *lazy unfolding* approach that, before unfolding a string variable into k elements, performs an high-level reasoning over the blocks of the corresponding dashed string $DS(x)$. This allows it to efficiently reason with potentially very large strings.

9. Conclusions

In this paper we defined the dashed string approach to string constraint solving. Dashed strings provide an expressive class of strings that can be used as a basis for propagating string constraints. Because they natively support concatentation and reversal, they provide strong support for common string operations. Because they provide compact representations of potentially very long strings, they provide an efficient way of reasoning about strings. Overall the dashed string approach provides a completely new and very efficient approach to string solving which is able to handle all common string operations relatively efficiently. The resulting solver G-Strings proves to be highly competitive with other existing state-of-the-art string solving approaches.

Possible future directions concern the propagation of more complex string constraints like pattern-matching, back-references or replacing all the occurrences of a string in another string. It would be also interesting to study new branching strategies, and to extend the dashed string approach with nogood learning.

Acknowledgments

This work is supported by the Australian Research Council (ARC) through Linkage Project Grant LP140100437 and Discovery Early Career Researcher Award DE160100568.

References

[1] M. Emmi, R. Majumdar, K. Sen, Dynamic test input generation for database applications, in: Proceedings of the ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA), ACM, 2007, pp. 151–162.

[2] N. Bjørner, N. Tillmann, A. Voronkov, Path feasibility analysis for string-manipulating programs, in: Proceedings of the International Conference on Tools and Algorithms for the Construction and Analysis of Systems, Vol. 5505 of LNCS, Springer, 2009, pp. 307–321.

[3] G. Gange, J. A. Navas, P. J. Stuckey, H. Søndergaard, P. Schachte, Unbounded model-checking with interpolation for regular language constraints, in: Proceedings of the International Conference on Tools and Algorithms for the Construction and Analysis of Systems, Vol. 7795 of LNCS, Springer, 2013, pp. 277–291.

[4] P. Bisht, T. L. Hinrichs, N. Skrupsky, V. N. Venkatakrishnan, WAPTEC: Whitebox analysis of web applications for parameter tampering exploit construction, in: Proceedings of ACM Conference on Computer and Communications Security, ACM, 2011, pp. 575–586.

[5] P. Barahona, L. Krippahl, Constraint programming in structural bioinformatics, Constraints 13 (1-2) (2008) 3–20.

[6] P. Hooimeijer, W. Weimer, StrSolve: Solving string constraints lazily, Automated Software Engineering 19 (4) (2012) 531–559.

[7] G. Li, I. Ghosh, PASS: String solving with parameterized array and interval automaton, in: V. Bertacco, A. Legay (Eds.), Proc. 9th Int. Haifa Verification Conf., Vol. 8244 of LNCS, Springer, 2013, pp. 15–31.

[8] T. Tateishi, M. Pistoia, O. Tripp, Path- and index-sensitive string analysis based on monadic second-order logic, ACM Trans. Software Engineering Methodology 22 (4) (2013) article 33.

[9] M. Berzish, V. Ganesh, Y. Zheng, Z3str3: A string solver with theory-aware heuristics, in: D. Stewart, G. Weissenbacher (Eds.), Proc. 17th Conf. Formal Methods in Computer-Aided Design, FMCAD Inc, 2017, pp. 55–59.

[10] T. Liang, A. Reynolds, C. Tinelli, C. Barrett, M. Deters, A DPLL(T) theory solver for a theory of strings and regular expressions, in: A. Biere, R. Bloem (Eds.), Computer Aided Verification: Proc. 26th Int. Conf., Vol. 8559 of LNCS, Springer, 2014, pp. 646–662.

[11] A. Kieżun, V. Ganesh, S. Artzi, P. J. Guo, P. Hooimeijer, M. D. Ernst, HAMPI: A solver for word equations over strings, regular expressions, and context-free grammars, ACM Trans. Software Engineering and Methodology 21 (4) (2012) article 25.

[12] P. Saxena, D. Akhawe, S. Hanna, F. Mao, S. McCamant, D. Song, A symbolic execution framework for JavaScript, in: Proc. 2010 IEEE Symp. Security and Privacy, IEEE Comp. Soc., 2010, pp. 513–528.

[13] J. D. Scott, P. Flener, J. Pearson, C. Schulte, Design and implementation of bounded-length sequence variables, in: M. Lombardi, D. Salvagnin (Eds.), Proc. 14th Int. Conf. Integration of Artificial Intelligence and Operations Research Techniques in Constraint Programming, Vol. 10335 of LNCS, Springer, 2017, pp. 51–67.

[14] R. Amadini, P. Flener, J. Pearson, J. D. Scott, P. J. Stuckey, G. Tack, MiniZinc with strings, in: M. Hermenegildo, P. López-García (Eds.), LOPSTR 2016: Revised Selected Papers, Vol. 10184 of LNCS, Springer, 2017, pp. 59–75.

[15] R. Amadini, G. Gange, P. J. Stuckey, G. Tack, A novel approach to string constraint solving, in: J. C. Beck (Ed.), Proc. 23rd Int. Conf. Principles and Practice of Constraint Programming, Vol. 10416 of LNCS, Springer, 2017, pp. 3–20.

[16] R. Amadini, G. Gange, P. J. Stuckey, Sweep-based propagation for string constraint solving, in: Proc. 32nd AAAI Conf. Artificial Intelligence, AAAI Press, 2018, pp. 6557–6564.

[17] R. Amadini, G. Gange, P. J. Stuckey, Propagating *Lex*, *Find* and *Replace* with dashed strings, in: W.-J. van Hoeve (Ed.), Proc. 15th Int. Conf. Integration of Artificial Intelligence and Operations Research Techniques in Constraint Programming, Vol. 10848 of LNCS, Springer, 2018.

[18] R. Amadini, G. Gange, P. J. Stuckey, Propagating regular membership with dashed strings, in: J. Hooker (Ed.), Proc. 24th Conf. Principles and Practice of Constraint Programming, Vol. 11008 of LNCS, Springer, 2018, pp. 13–29.

[19] Roberto Amadini, G-Strings: Gecode with String Variables, available at `https://bitbucket.org/robama/g-strings` (2019).

[20] D. Blotsky, F. Mora, M. Berzish, Y. Zheng, I. Kabir, V. Ganesh, Stringfuzz: A fuzzer for string solvers, in: Computer Aided Verification - 30th International Conference, CAV 2018, Held as Part of the Federated Logic Conference, FloC 2018, Oxford, UK, July 14-17, 2018, Proceedings, Part II, 2018, pp. 45–51.

[21] C. Michel, M. Rueher, Y. Lebbah, Solving constraints over floating-point numbers, in: Principles and Practice of Constraint Programming - CP 2001, 7th International Conference, CP 2001, Paphos, Cyprus, November 26 - December 1, 2001, Proceedings, 2001, pp. 524–538.

[22] G. Costantini, P. Ferrara, A. Cortesi, A suite of abstract domains for static analysis of string values, Software: Practice and Experience 45 (2) (2015) 245–287.

[23] P. Cousot, R. Cousot, Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints, in:

Proceedings of the Fourth ACM Symposium on Principles of Programming Languages, ACM, 1977, pp. 238–252.

[24] A. Aggoun, N. Beldiceanu, Extending CHIP in order to solve complex scheduling and placement problems, Mathematical and Computer Modelling 17 (7) (1993) 57–73.

[25] R. Barták, Modelling resource transitions in constraint-based scheduling, in: SOFSEM 2002: Theory and Practice of Informatics, 29th Conference on Current Trends in Theory and Practice of Informatics, Milovy, Czech Republic, November 22-29, 2002, Proceedings, 2002, pp. 186–194.

[26] G. Pesant, A regular language membership constraint for finite sequences of variables, in: M. Wallace (Ed.), Proceedings of the 10th International Conference on Principles and Practice of Constraint Programming, Vol. 3258 of LNCS, Springer-Verlag, 2004, pp. 482–495.

[27] K. Cheng, R. Yap, Maintaining generalized arc consistency on ad hoc r-ary constraints, in: 14th International Conference on Principles and Process of Constraint Programming, Vol. 5202 of LNCS, 2008, pp. 509–523.

[28] G. Perez, J. Régin, Improving GAC-4 for table and MDD constraints, in: B. O'Sullivan (Ed.), Principles and Practice of Constraint Programming - 20th International Conference, CP 2014, Lyon, France, September 8-12, 2014. Proceedings, Vol. 8656 of Lecture Notes in Computer Science, Springer, 2014, pp. 606–621.

[29] N. Nethercote, P. J. Stuckey, R. Becket, S. Brand, G. J. Duck, G. Tack, MiniZinc: Towards a standard CP modelling language, in: Proceedings of the 13th International Conference on Principles and Practice of Constraint Programming, Vol. 4741 of LNCS, Springer, 2007, pp. 529–543.

[30] Gecode Team, Gecode: Generic constraint development environment, available at `http://www.gecode.org` (2016).

[31] R. Amadini, M. Andrlon, G. Gange, P. Schachte, H. Søndergaard, P. J. Stuckey, Constraint programming for dynamic symbolic execution of javascript, in: To appear in Proc. 16th Int. Conf. Integration of Artificial Intelligence and Operations Research Techniques in Constraint Programming, LNCS, Springer, 2019.

[32] Mak Andrlon and Roberto Amadini, Aratha DSE tool, available at `https://github.com/ArathaJS/aratha` (2019).

[33] P. Godefroid, N. Klarlund, K. Sen, DART: Directed automated random testing, in: Proc. ACM SIGPLAN Conf. Programming Language Design and Implementation (PLDI'05), ACM, 2005, pp. 213–223.

[34] R. Majumdar, K. Sen, Hybrid concolic testing, in: Proc. 29th Int. Conf. Software Engineering (ICSE 2007), IEEE, 2007, pp. 416–426.

[35] J. C. King, Symbolic execution and program testing, Communications of the ACM 19 (7) (1976) 385–394.

[36] D. Blotsky, F. Mora, M. Berzish, Y. Zheng, I. Kabir, V. Ganesh, Stringfuzz: A fuzzer for string solvers, in: Computer Aided Verification - 30th International Conference, CAV 2018, Held as Part of the Federated Logic Conference, FloC 2018, Oxford, UK, July 14-17, 2018, Proceedings, Part II, 2018, pp. 45–51.

[37] L. M. de Moura, N. 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, 2008, pp. 337–340.

[38] P. A. Abdulla, M. F. Atig, Y. Chen, L. Holík, A. Rezine, P. Rümmer, J. Stenman, Norn: An SMT solver for string constraints, in: CAV, Vol. 9206 of LNCS, Springer, 2015, pp. 462–469.

[39] M. Trinh, D. Chu, J. Jaffar, Model counting for recursively-defined strings, in: Computer Aided Verification - 29th International Conference, CAV 2017, Heidelberg, Germany, July 24-28, 2017, Proceedings, Part II, 2017, pp. 399–418.

[40] P. A. Abdulla, M. F. Atig, Y. Chen, B. P. Diep, L. Holík, A. Rezine, P. Rümmer, Flatten and conquer: a framework for efficient analysis of string constraints, in: Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2017, Barcelona, Spain, June 18-23, 2017, 2017, pp. 602–617.

[41] L. Holík, P. Janku, A. W. Lin, P. Rümmer, T. Vojnar, String constraints with concatenation and transducers solved efficiently, PACMPL 2 (POPL) (2018) 4:1–4:32.

[42] P. J. Stuckey, T. Feydy, A. Schutt, G. Tack, J. Fischer, The minizinc challenge 2008-2013, AI Magazine (2) (2014) 55–60.

[43] R. Amadini, A. Jordan, G. Gange, F. Gauthier, P. Schachte, H. Søndergaard, P. J. Stuckey, C. Zhang, Combining string abstract domains for JavaScript analysis: An evaluation, in: A. Legay, T. Margaria (Eds.), Proc. 23rd Int. Conf. Tools and Algorithms for the Construction and Analysis of Systems, Part I, Vol. 10205 of LNCS, Springer, 2017, pp. 41–57.

[44] R. Amadini, G. Gange, F. Gauthier, A. Jordan, P. Schachte, H. Søndergaard, P. J. Stuckey, C. Zhang, Reference abstract domains and applications to string analysis, Fundam. Inform. 158 (4) (2018) 297–326.

[45] B. Loring, D. Mitchell, J. Kinder, ExpoSE: Practical symbolic execution of standalone JavaScript, in: Proc. 24th ACM SIGSOFT Int. SPIN Symp. Model Checking of Software (SPIN'17), ACM Press, 2017, pp. 196–199.

[46] J. Thomé, L. K. Shar, D. Bianculli, L. C. Briand, Search-driven string constraint solving for vulnerability detection, in: ICSE 2017, Buenos Aires, Argentina, May 20-28, 2017, 2017, pp. 198–208.

[47] F. Yu, M. Alkhalaf, T. Bultan, Stranger: An automata-based string analysis tool for PHP, in: TACAS, Vol. 6015 of LNCS, Springer, 2010, pp. 154–157.

[48] Y. Zheng, V. Ganesh, S. Subramanian, O. Tripp, J. Dolby, X. Zhang, Effective search-space pruning for solvers of string equations, regular expressions and length constraints, in: Computer Aided Verification - 27th International Conference, CAV 2015, San Francisco, CA, USA, July 18-24, 2015, Proceedings, Part I, 2015, pp. 235–254.

[49] H. Ganzinger, G. Hagen, R. Nieuwenhuis, A. O. a, C. Tinelli, Dpll(t): Fast decision procedures, in: R. Alur, D. A. Peled (Eds.), Computer Aided Verification: Proc. 16th Int. Conf., Vol. 3114 of LNCS, Springer, 2004, pp. 175–188.

[50] Y. Zheng, X. Zhang, V. Ganesh, Z3-str: A Z3-based string solver for web application analysis, in: Proc. 9th Joint Meeting on Foundations of Software Engineering, ACM, 2013, pp. 114–124.

[51] Y. Zheng, V. Ganesh, S. Subramanian, O. Tripp, J. Dolby, X. Zhang, Effective search-space pruning for solvers of string equations, regular expressions and length constraints, in: CAV, Vol. 9206 of LNCS, Springer, 2015, pp. 235–254.

[52] M.-T. Trinh, D.-H. Chu, J. Jaffar, S3: A symbolic string solver for vulnerability detection in web applications, in: Proc. 2014 ACM SIGSAC Conf. Computer and Communications Security, ACM, 2014, pp. 1232–1243.

[53] M.-T. Trinh, D.-H. Chu, J. Jaffar, Progressive reasoning over recursively-defined strings, in: Computer Aided Verification, Vol. 9779 of Lecture Notes in Computer Science, Springer, 2016, pp. 218–240.

[54] P. A. Abdulla, M. F. Atig, Y.-F. Chen, L. Holík, A. Rezine, P. Rümmer, J. Stenman, Norn: An SMT solver for string constraints, in: D. Kroening, C. Păsăreanu (Eds.), Computer Aided Verification: Proc. 27th Int. Conf. Part I, Vol. 9206 of LNCS, Springer, 2015, pp. 462–469.

Appendix

Proofs of Section 3

Proposition 1. The following properties hold for γ:

(i) γ is not injective: $\gamma(X) = \gamma(Y)$ does not always implies $X = Y$

(ii) If $X, Y \in \overline{\mathbb{DS}}_\Sigma^\lambda$ are such that $\sum_{i=1}^{|X|} ub(X[i]) \leq \lambda$ and $\sum_{i=1}^{|Y|} ub(Y[i]) \leq \lambda$, then $\gamma(X) = \gamma(Y)$ implies $X = Y$.

Proof. *(i)* Let $X = \{a\}^{0,1}\{b\}^{0,1}\{a\}^{0,1}$ and $Y = \{b\}^{0,1}\{a\}^{0,1}\{b\}^{0,1}$ with $\Sigma = \{a, b\}$ and $\lambda = 2$. We have that $\mathbb{S}_\Sigma^\lambda = \{\epsilon, a, b, aa, ab, ba, bb\}$ and $\gamma(X) = \{\epsilon, a, b, aa, ab, ba, bb, aba\} \cap \mathbb{S}_\Sigma^\lambda = \mathbb{S}_\Sigma^\lambda = \{\epsilon, a, b, aa, ab, ba, bb, bab\} \cap \mathbb{S}_\Sigma^\lambda = \gamma(Y)$.

(ii) Let $X = S_1^{l_1,u_1} \cdots S_n^{l_n,u_n}$ with $U = \sum_{i=1}^n u_i \leq \lambda$, so $|w| \leq \lambda$ for each $w \in \gamma(X)$, therefore $\gamma(X) \subseteq \mathbb{S}_\Sigma^\lambda$ and hence $\gamma(X) = \gamma(S_1^{l_1,u_1}) \cdots \gamma(S_n^{l_n,u_n})$. Let $Y = {S'_1}^{l'_1,u'_1} \cdots {S'_m}^{l'_m,u'_m}$ with $U' = \sum_{i=1}^n u'_i \leq \lambda$, so $\gamma(Y) = \gamma({S'_1}^{l'_1,u'_1}) \cdots \gamma({S'_m}^{l'_m,u'_m})$. Let us assume $\gamma(X) = \gamma(Y)$. We have to prove that $X = Y$.

First, note that $U = U'$ (otherwise there is at least a string with length $\max(U, U')$ that does not belong to $\gamma(X) \cap \gamma(Y)$). If $U = 0$ the proof is trivial ($X = Y = \emptyset^{0,0}$) so let us assume $U > 0$ (since X and Y are normalised, no $\emptyset^{0,0}$ occurs in them). Now, let $p = \min(n, m)$ and let us suppose to the contrary that there exists an index $i \in \{1, \ldots, p\}$ such that $X[i] \neq Y[i]$. We have three cases:

(a) if $S_i \neq S'_i$, let us assume w.l.o.g. that $S_i \not\subset S'_i$ (if $S_i \subset S'_i$ the proof is symmetrical because this implies $S'_i \not\subset S_i$);

(b) if $S_i = S'_i$ and $u_i \neq u'_i$, let us assume w.l.o.g. $u_i > u'_i$;

(c) if $S_i = S'_i$, $u_i = u'_i$, and $l_i \neq l'_i$, let us assume w.l.o.g. $l_i < l'_i$;

In case (a) consider $w = a_1^{u_1} \cdots a_{i-1}^{u_{i-1}} a_i^{u_i} a_{i+1}^{u_{i+1}} \cdots a_n^{u_n}$ with $a_i \in S_i - S'_i$ and $a_j \in S_j$ for $j \neq i$. We have that $w \in \gamma(X)$ with $|w| = U$ but $w \notin \gamma(Y)$ because, being $a_i \notin S'_i$ and $u_i > 0$, the maximum length of a string of $\gamma(Y)$ having prefix $a_1^{u_1} \cdots a_i^{u_i}$ is $U - u'_i < U = |w|$.

In case (b) consider $w = a_1^{u_1} \cdots a_n^{u_n}$ with $a_j \in S_j$ for $j = 1, \ldots, p$. We have that $w \in \gamma(X)$ with $|w| = U$ but $w \notin \gamma(Y)$ because, since $u'_i < u_i$, the maximum length of a string of $\gamma(Y)$ would be $U - (u_i - u'_i) < U = |w|$.

In case (c) consider $w = a_1^{u_1} \cdots a_{i-1}^{u_{i-1}} a_i^{l_i} a_{i+1}^{u_{i+1}}$ where $a_i \in S_i$, $a_{i+1} \in S_{i+1}$ and $a_i \neq a_{i+1}$ (if $i = p$, just consider $w = a_1^{u_1} \cdots a_{i-1}^{u_{i-1}} a_i^{l_i}$). We have that w is a prefix of a number of strings of $\gamma(X)$, but w cannot be prefix of any string of $\gamma(Y)$ because there are $l'_i - l_i > 0$ characters a_i of S'_i that cannot match $a_{i+1}^{u_{i+1}}$.

So, we proved that $X[i] = Y[i]$ for $i = 1, \dots, \min(n, m)$. Now, if $n < m$ for each string $w \in \gamma(X)$ and for each $w' \in \gamma(Y[n+1] \cdots Y[m])$ we have that $ww' \in \gamma(Y) - \gamma(X)$. So, $n \geq m$ and symmetrically we can prove that $m \geq n$: we have that $n = m$ and $X[i] = Y[i]$ for $i = 1, \dots, n$. Hence, $X = Y$. □

Theorem 1. The poset $(\overline{\mathbb{DS}}_{\Sigma}^{\lambda}, \sqsubseteq)$ has no infinite descending chains.

Proof. Let us suppose, to the contrary, that there exists an infinite number of dashed strings $X_0, X_1, \dots, X_k, \dots \in \overline{\mathbb{DS}}_{\Sigma}^{\lambda}$ such that $X_{n+1} \sqsubset X_n$ for each $n \in \mathbb{N}$. By definition this would imply $\gamma(X_{n+1}) \subset \gamma(X_n)$, which is clearly impossible. □

Theorem 2. Let Σ be a finite alphabet with at least two symbols and $\lambda \in \mathbb{N} \cup \{+\infty\}$ a maximum string length. The poset $(\overline{\mathbb{DS}}_{\Sigma}^{\lambda}, \sqsubseteq)$ is *not a lattice.*

Proof. We actually prove something stronger, i.e., that $(\overline{\mathbb{DS}}_{\Sigma}^{\lambda}, \sqsubseteq)$ is neither: *(i)* a *join-semilattice*, nor *(ii)* a *meet-semilattice.*

(i) We prove that there exist two dashed strings $X, Y \in \overline{\mathbb{DS}}_{\Sigma}^{\lambda}$ not having a least upper bound in $\overline{\mathbb{DS}}_{\Sigma}^{\lambda}$ according to $\sqsubseteq$. Let $\Sigma = \{a, b\}$, $X = \{a\}^{1,1}\{b\}^{1,1}$, and $Y = \{b\}^{1,1}\{a\}^{1,1}$. The minimal elements greater than both X and Y according to $\sqsubseteq$ (i.e., those denoting the smallest set containing $\{ab, ba\}$) are $Z = \{a, b\}^{2,2}$, $Z' = \{a\}^{0,1}\{b\}^{1,1}\{a\}^{0,1}$, and $Z'' = \{b\}^{0,1}\{a\}^{1,1}\{b\}^{0,1}$. However, Z, Z', Z'' are incomparable according to $\sqsubseteq$ because $\gamma(Z) = \{aa, ab, ba, bb\}$, $\gamma(Z') = \{ab, aba, b, ba\}$ and $\gamma(Z'') = \{a, ab, ba, bab\}$.

(ii) Dually, we prove that there exist two dashed strings in $\overline{\mathbb{DS}}_{\Sigma}^{\lambda}$ not having a greatest lower bound. Let us take Z, Z' as above. We have that the greatest element smaller than both Z, Z' according to $\sqsubseteq$ is the one denoting the biggest set contained in both $\gamma(Z), \gamma(Z')$, i.e., contained in $\gamma(Z) \cap \gamma(Z') = \{ab, ba\}$. As seen in *(i)*, there does not exist a dashed string denoting $\{ab, ba\}$: the biggest

elements smaller than Z, Z' are thus $X = \{a\}^{1,1}\{b\}^{1,1}$ and $Y = \{b\}^{1,1}\{a\}^{1,1}$, denoting $\{ab\}$ and $\{ba\}$ respectively. Unfortunately, $X \not\sqsubseteq Y \wedge Y \not\sqsubseteq X$. □

Proofs of Section 4

Proposition 2. For each $X, Y \in \overline{\mathbb{DS}}^{\lambda}_{\Sigma}$, the following properties hold:

(i) $(X, Y) = \max_{\sqsubseteq}\left(\mathit{Ref}_{X,Y}\right)$

(ii) $(\mathit{Ref}_{X,Y}, \sqsubseteq)$ has a minimal element

(iii) $\min_{\sqsubseteq}\left(\mathit{Ref}_{X,Y}\right)$ does not always exist

Proof. *(i)* By definition $(X, Y) \in \mathit{Ref}_{X,Y}$. If $(X', Y') \in \mathit{Ref}_{X,Y}$ such that $(X', Y') \not\sqsubseteq (X, Y)$, then $X' \not\sqsubseteq X \vee Y' \not\sqsubseteq Y$ so $(X', Y') \notin \mathit{Ref}_{X,Y}$.

(ii) It is sufficient to prove that $\mathit{Ref}_{X,Y}$ is finite. We proved in Theorem 1 that $\overline{\mathbb{DS}}^{\lambda}_{\Sigma}$ has no infinite descending chains, so the set of lower bounds $\mathit{LB}_{\sqsubseteq}(X) = \{X' \mid X' \sqsubseteq X\}$ is finite, and so it is $\mathit{LB}_{\sqsubseteq}(Y)$. Hence, $\mathit{LB}_{\sqsubseteq}(X) \times \mathit{LB}_{\sqsubseteq}(Y)$ is finite. By definition, $\mathit{Ref}_{X,Y} \subseteq \mathit{LB}_{\sqsubseteq}(X) \times \mathit{LB}_{\sqsubseteq}(Y)$.

(iii) Let $X = \{a, b\}^{1,1}$ and $Y = \{b\}^{0,1}\{a\}^{0,1}\{b\}^{0,1}$. We have that the minimal elements of $\mathit{Ref}_{X,Y}$ are $(X, \{a\}^{0,1}\{b\}^{0,1})$ and $(X, \{b\}^{0,1}\{a\}^{0,1})$ which are incomparable. □

Lemma 1. Let $S^{l,u}$ be a block, Y a dashed string with $m = |Y|$, and $P \prec (m + 1, 0)$ a position. If $\textsc{push}^{+}(S^{l,u}, Y, P) = ((m + 1, 0), (m + 1, 0))$, then for each $w \in \gamma(S^{l,u})$ there is no $w' \in \gamma(Y[P, ...])$ such that w is substring of w'.

Proof. If $P \prec (m+1, 0)$ then it must be that $l > 0$ (otherwise (P, P) is returned). The only case when $((m + 1, 0), (m + 1, 0))$ is returned is when line 6 of $\textsc{push}^{+}$ is reached: the while loop scans all the blocks of Y (because $i > |Y|$ in the pseudo code of $\textsc{push}^{+}$) but less than l characters of $\gamma(S^{l,u})$ are consumed (because $k > 0$ in the pseudo code of $\textsc{push}^{+}$), i.e., there is no sub-region $Y[P', Q']$ of $Y[P, ...]$ matched by $S^{l,u}$. This means that for each $w \in \gamma(S^{l,u})$ there cannot be $w' \in \gamma(Y[P, ...])$ such that w is a substring of w'. □

Lemma 2. Let $X, Y \in \overline{\mathbb{DS}}_{\Sigma}^{\lambda}$ with $n = |X|$ and $z = z_1 \cdots z_n \in \gamma(X) \cap \gamma(Y)$ such that $z_i \in \gamma(X[i])$ for $i = 1, \dots, n$. Let $\mathbb{P}_1, \dots, \mathbb{P}_{n+1}$ be a concrete matching for z in Y. Then, for each $P \preceq \mathbb{P}_i$ we have that $(P', P'') = \textsc{push}^{+}(X[i], Y, P)$ implies $P' \preceq \mathbb{P}_i$ and $P'' \preceq \mathbb{P}_{i+1}$.

Proof. Let us suppose that $P' \succ \mathbb{P}_i$, i.e., $\textsc{push}^{+}$ pushes block $X[i]$ from $P \preceq \mathbb{P}_i$ to a position P' after $\mathbb{P}_i$. This means that block $Y[j] = S_j^{l_j, u_j}$ where $\mathbb{P}_i = (j, k)$ is incompatible with $X[i] = S_i^{l_i, u_i}$, i.e., $S_i \cap S_j = \emptyset$, $l_i > 0$, and $l_j > 0$. Since $z_i \in \gamma(Y[\mathbb{P}_i, \mathbb{P}_{i+1}])$ we would have $|z_i| > 0$ and $z_i[1] \in S_j - S_i$, which is impossible because $z_i \in \gamma(X[i])$.

So, $P' \preceq \mathbb{P}_i$. Let us suppose now $P'' \succ \mathbb{P}_{i+1}$. According to the definition of $\textsc{push}^{+}$, this would imply that block $X[i]$ starts at position P' and finishes after $\mathbb{P}_{i+1}$, i.e., the $lb(X[i])$ mandatory characters of $X[i]$ exceed the sum of the upper bounds u_j of the blocks $Y[j] = S_j^{l_j, u_j}$ in $Y[P', \mathbb{P}_{i+1}]$ such that $S_i \cap S_j \neq \emptyset$. In other words, any string in $\gamma(X[i])$ would be longer than any string in $\gamma(Y[P', \mathbb{P}_{i+1}])$, and thus longer than any string in $\gamma(Y[\mathbb{P}_i, \mathbb{P}_{i+1}])$ because $P' \preceq \mathbb{P}_i$. This is impossible because $z_i \in \gamma(X[i]) \cap \gamma(Y[\mathbb{P}_i, \mathbb{P}_{i+1}])$ by hypothesis. □

Lemma 3. Let $S^{l,u}$ be a block, Y a dashed string and P a position. If $\textsc{push}^{-}(S^{l,u}, Y, P) = ((0,0),(0,0))$, then for each $w \in \gamma(S^{l,u})$ there is no $w' \in \gamma(Y[(1,0), P])$ such that w is substring of w'.

Proof. Symmetric to the proof of Lemma 1. □

Lemma 4. Let $X, Y \in \overline{\mathbb{DS}}_{\Sigma}^{\lambda}$ with $n = |X|$ and $z = z_1 \cdots z_n \in \gamma(X) \cap \gamma(Y)$ such that $z_i \in \gamma(X[i])$, and $\mathbb{P}_1 = (1,0) \preceq \mathbb{P}_2 \preceq \cdots \preceq \mathbb{P}_n \preceq \mathbb{P}_{n+1} = (|Y| + 1)$ such that $z_i \in \gamma(Y[\mathbb{P}_i, \mathbb{P}_{i+1}])$ for $i = 1, \dots, n$. For each $P \succeq \mathbb{P}_{i+1}$ we have that $(P', P'') = \textsc{push}^{-}(X[i], Y, P)$ implies $P' \succeq \mathbb{P}_i$ and $P'' \succeq \mathbb{P}_{i+1}$.

Proof. Symmetric to the proof of Lemma 2. □

Lemma 5. Let $X, Y \in \overline{\mathbb{DS}}_{\Sigma}^{\lambda}$ with $n = |X|$ and $z = z_1 \cdots z_n \in \gamma(X) \cap \gamma(Y)$ such that $z_i \in \gamma(X[i])$ for $i = 1, \dots, n$, and let $\mathbb{P}_1, \dots, \mathbb{P}_{n+1}$ a concrete matching for z in Y. For each position $P \succeq \mathbb{P}_i$ we have $\textsc{stretch}^{+}(X[i], Y, P) \succeq \mathbb{P}_{i+1}$.

Proof. If $P \succ \mathbb{P}_{i+1}$ the lemma is trivially true, so let us assume $\mathbb{P}_i \preceq P \preceq \mathbb{P}_{i+1}$. Let $X[i] = S_i^{l_i,u_i}$. Suppose to the contrary $\textsc{stretch}^+(X[i], Y, P) \prec \mathbb{P}_{i+1}$. We have two cases: (a) $\textsc{stretch}^+$ encounters a block $Y[j] = S_j^{l_j,u_j}$ such that $S_i \cap S_j = \emptyset$ and $l_j > 0$ before reaching position $\mathbb{P}_{i+1}$ (line 9); or (b) all the u_i characters of $X[i]$ are consumed before reaching $\mathbb{P}_{i+1}$ (line 11).

In case (a), if $z_i = \epsilon$ then region $Y[\mathbb{P}_i, \mathbb{P}_{i+1}]$ would be nullable, and thus $\textsc{stretch}^+$ would jump to the beginning of next block until we consume all the blocks in $Y[\mathbb{P}_i, \mathbb{P}_{i+1}]$. But this is impossible because we assumed that $\textsc{stretch}^+(X[i], Y, P) \prec \mathbb{P}_{i+1}$. So, $z_i \neq \epsilon$, i.e., $l_i > 0$. If $S_i \cap S_j = \emptyset$ and $l_j > 0$ then z_i contains at least a character in $S_j - S_i$, which is impossible because $z_i \in \gamma(X[i])$ by hypothesis.

In case (b), we would have that the maximum number of characters for a string in $\gamma(X[i])$ is smaller than the minimum number of characters for a string in $\gamma(Y[P, \mathbb{P}_{i+1}])$, i.e., $u_i < \sum_{B \in Y[P,\mathbb{P}_{i+1}]} lb(B) \leq \sum_{B \in Y[\mathbb{P}_i,\mathbb{P}_{i+1}]} lb(B)$. This means that $|w| < |w'|$ for each $w \in \gamma(X[i]), w' \in \gamma(Y[\mathbb{P}_i, \mathbb{P}_{i+1}])$ which is impossible because $z_i \in \gamma(X[i]) \cap \gamma(Y[\mathbb{P}_i, \mathbb{P}_{i+1}])$. □

Lemma 6. Let $X, Y \in \overline{\mathbb{DS}}_\Sigma^\lambda$ with $n = |X|$ and $z = z_1 \cdots z_n \in \gamma(X) \cap \gamma(Y)$ such that $z_i \in \gamma(X[i])$, and let $\mathbb{P}_1, \ldots, \mathbb{P}_{n+1}$ a concrete matching for z in Y. For each position P we have that $P \preceq \mathbb{P}_{i+1}$ implies $\textsc{stretch}^-(X[i], Y, P) \preceq \mathbb{P}_i$.

Proof. Symmetrical to the proof of Lemma 5. □

Lemma 7. Let $X, Y \in \overline{\mathbb{DS}}_\Sigma^\lambda$ with $n = |X|$ and $\textsc{Init}(X, Y) = (ESP, LEP) \neq (\bot, \bot)$. Let $z = z_1 \cdots z_n \in \gamma(X) \cap \gamma(Y)$ such that $z_i \in \gamma(X[i])$ for $i = 1, \ldots, n$, and let $\mathbb{P}_1, \ldots, \mathbb{P}_{n+1}$ a concrete matching for z in Y. Then, $ESP[i] \preceq \mathbb{P}_i$ and $\mathbb{P}_{i+1} \preceq LEP[i]$ for $i = 1, \ldots, n$.

Proof. If $\textsc{Init}(X, Y) = (ESP, LEP) \neq (\bot, \bot)$ then $LEP[n] = (m, ub(Y[m])) = \mathbb{P}_{n+1}$ where $m = |Y|$, so $\mathbb{P}_{n+1} \preceq LEP[n]$. Let us assume to the contrary that there exists $k < n$ such that $LEP[k] \prec \mathbb{P}_{k+1}$ i.e., that there exists $k < n$ such that $\textsc{stretch}^+(X[k], Y, LEP[k-1]) \prec \mathbb{P}_{k+1}$. By the contrapositive of Lemma 5 this would imply $LEP[k-1] \prec \mathbb{P}_k$, i.e., $\textsc{stretch}^+(X[k-1], Y, LEP[k-2]) \prec \mathbb{P}_k$.

By reapplying Lemma 5 we have $LEP[k-2] \prec \mathbb{P}_{k-1}$ and so we can iterate this process until we get $LEP[1] = \textsc{stretch}^{+}(X[1], Y, (1,0)) \prec \mathbb{P}_2$ and thus by Lemma 5 we conclude $(1,0) \prec \mathbb{P}_1 = (1,0)$ which is impossible. Hence, $\mathbb{P}_{i+1} \preceq LEP[i]$ for $i = 1, \ldots, n$ and dually, by means of Lemma 6, it can be proven that $ESP[i] \preceq \mathbb{P}_i$. □

Lemma 8. Let $X, Y \in \overline{\mathbb{DS}}_{\Sigma}^{\lambda}$. If $\textsc{Init}(X, Y) = (\bot, \bot)$ then $\gamma(X) \cap \gamma(Y) = \emptyset$.

Proof. Let us assume that $\gamma(X) \cap \gamma(Y) \neq \emptyset$, i.e., there exists $z \in \gamma(X) \cap \gamma(Y)$. Let $n = |X|$ and $z = z_1 \cdots z_n \in \gamma(X) \cap \gamma(Y)$ such that $z_i \in \gamma(X[i])$, and let $\mathbb{P}_1, \ldots, \mathbb{P}_{n+1}$ a concrete matching for z in Y. If $\textsc{Init}(X, Y) = (\bot, \bot)$ then $LEP[n] \prec (|Y| + 1, 0) = \mathbb{P}_{n+1}$ or $ESP[1] \succ (1, 0) = \mathbb{P}_1$. By Lemma 7, both these conditions are impossible. □

Lemma 9. Let $X, Y \in \overline{\mathbb{DS}}_{\Sigma}^{\lambda}$ such that $\textsc{Init}(X, Y) = (ESP, LEP)$. Then, for each $i = 1, \ldots, |X|$, $\textsc{push}\mathrm{ESP}(X, Y, ESP, i) = \bot$ implies $\gamma(X) \cap \gamma(Y) = \emptyset$.

Proof. Let us assume that $\gamma(X) \cap \gamma(Y) \neq \emptyset$, i.e., that exists $z \in \gamma(X) \cap \gamma(Y)$. Let $n = |X|$ and $z = z_1 \cdots z_n \in \gamma(X) \cap \gamma(Y)$ such that $z_i \in \gamma(X[i])$, and let $\mathbb{P}_1, \ldots, \mathbb{P}_{n+1}$be a concrete matching for z in Y.

$\textsc{push}$ESP returns $\bot$ only when $lb(X[i]) > 0$ and $\textsc{push}^{+}(X[i], Y, ESP[i]) = ((m{+}1, 0), (m{+}1, 0))$ with $m = |Y|$. By Lemma 1 this means that any $w \in \gamma(X[i])$ is not substring of any $w' \in \gamma(Y[ESP[i], ...])$ and thus is not substring of any $w' \in \gamma(Y[\mathbb{P}_i, \mathbb{P}_{i+1}])$ because by Lemma 7 we have $ESP[i] \preceq \mathbb{P}_i$ and $\mathbb{P}_{i+1} \preceq LEP[i] \preceq (m+1, 0)$. So there would not exist any $z_i \in \gamma(X[i]) \cap \gamma(Y[\mathbb{P}_i, \mathbb{P}_{i+1}])$. □

Lemma 10. Let $X, Y \in \overline{\mathbb{DS}}_{\Sigma}^{\lambda}$ with $n = |X|$. Let $z = z_1 \cdots z_n \in \gamma(X) \cap \gamma(Y)$ such that $z_i \in \gamma(X[i])$ for $i = 1, \ldots, n$, and let $\mathbb{P}_1, \ldots, \mathbb{P}_{n+1}$ a concrete matching for z in Y. Let ESP such that $ESP[i] \preceq \mathbb{P}_i$ for $i = 1, \ldots, n$. If $ESP' = \textsc{push}\mathrm{ESP}(X, Y, ESP, i) \neq \bot$ then $ESP'[i] \preceq \mathbb{P}_i$ for $i = 1, \ldots, n$.

Proof. Note that $ESP'[j] = ESP[j] \preceq \mathbb{P}_j$ for $j = 1, \ldots, i-1, i+2, \ldots, n$ because $\textsc{push}$ESP can only modify either $ESP[i]$ or $ESP[i{+}1]$. Let $(start, end) =$

$\textsc{push}^{+}(X[i], Y, ESP[i])$. There are three mutually exclusive cases where $ESP \neq ESP'$:

(a) If $lb(X[i]) = 0$ and $ESP[i+1] \prec ESP[i]$ then $ESP'[i+1] = ESP[i]$ (see line 4 of Figure 8). By hypothesis we have $ESP[i] \preceq \mathbb{P}_i \preceq \mathbb{P}_{i+1}$ so $ESP'[i+1] \preceq \mathbb{P}_{i+1}$.

(b) If $i < n$ and $ESP[i+1] \prec end$ then $ESP'[i+1] = end$ (see line 10). By Lemma 2 since $ESP[i] \prec \mathbb{P}_i$ we have that $end \preceq \mathbb{P}_{i+1}$.

(c) If $ESP[i] \prec start$ then $ESP'[i] = start$ (see line 12). By Lemma 2 we have that $start \preceq \mathbb{P}_i$. □

Lemma 11. Let $X, Y \in \overline{\mathbb{DS}}_{\Sigma}^{\lambda}$ such that $\textsc{Init}(X, Y) = (ESP, LEP)$. Then, for each $i = 1, \ldots, |X|$, $\textsc{push}\text{LEP}(X, Y, LEP, i) = \bot$ implies $\gamma(X) \cap \gamma(Y) = \emptyset$.

Proof. Symmetric to proof of Lemma 9. □

Lemma 12. Let $X, Y \in \overline{\mathbb{DS}}_{\Sigma}^{\lambda}$ with $n = |X|$. Let $z = z_1 \cdots z_n \in \gamma(X) \cap \gamma(Y)$ such that $z_i \in \gamma(X[i])$ for $i = 1, \ldots, n$, and let $\mathbb{P}_1, \ldots, \mathbb{P}_{n+1}$ a concrete matching for z in Y. Let LEP such that $\mathbb{P}_{i+1} \preceq LEP[i]$. If $LEP' = \textsc{push}\text{LEP}(X, Y, LEP, i) \neq \bot$ then $\mathbb{P}_{i+1} \preceq LEP'[i]$ for $i = 1, \ldots, n$.

Proof. Symmetric to proof of Lemma 10. □

Lemma 13. Let $X, Y \in \overline{\mathbb{DS}}_{\Sigma}^{\lambda}$. If $\textsc{Sweep}(X, Y) = \bot$, then $\gamma(X) \cap \gamma(Y) = \emptyset$.

Proof. If $\textsc{Sweep}(X, Y) = \bot$ because $\textsc{Init}(X, Y) = (\bot, \bot)$, then $\gamma(X) \cap \gamma(Y) = \emptyset$ by Lemma 7.

Let $(ESP^0, LEP^0) = \textsc{Init}(X, Y)$. Because ESP is updated at each iteration in loop 5, we define $ESP^i = \textsc{push}\text{ESP}(X[i], Y, ESP^{i-1}, i)$ and we prove that if there exists $i \in \{1, \ldots, n\}$ such that $ESP^i = \bot$, then $\gamma(X) \cap \gamma(Y) = \emptyset$.

Let $k = \min\{i \mid \textsc{push}\text{ESP}(X, Y, ESP^{i-1}, i) = \bot\}$ and suppose to the contrary that there exists $z = z_1 \cdots z_n \in \gamma(X) \cap \gamma(Y)$ with $z_i \in \gamma(X[i])$. Let $\mathbb{P}_1, \ldots, \mathbb{P}_{n+1}$ be a corresponding concrete matching in Y.

By applying repeatedly Lemma 10, we have $ESP^1[i] \preceq \mathbb{P}_i, ESP^2[i] \preceq \mathbb{P}_i, \ldots, ESP^{k-1}[i] \preceq \mathbb{P}_i$ for $i = 1, \ldots, n$. So, in particular $ESP^{k-1}[k] \preceq \mathbb{P}_k$. But if $\textsc{push}\mathrm{ESP}(X, Y, ESP^{k-1}, k) = \bot$ then there is no string in $\gamma(X[k])$ that can be substring of a string in $\gamma(Y[ESP^{k-1}[k], ...])$ (see Lemma 9). Hence, it must be that $\mathbb{P}_k \prec ESP^{k-1}[k]$ which is impossible.

By applying the dual reasoning, by means of Lemmas 11 and 12 we can prove that $\textsc{push}\mathrm{LEP}(X[i], Y, LEP^j, i) = \bot$ implies $\gamma(X) \cap \gamma(Y) = \emptyset$.

Now assume $\textsc{Sweep}(X, Y) = \bot$ because after n iterations of loops of line 5 and line 9 the latest start $LSP^n[i]$ of block $X[i]$ comes before its earliest start $ESP^n[i]$ for a given $i \in \{2, \ldots, n\}$ (it cannot be $i = 1$ because $LSP[1] = ESP^n[1]$). Let us assume that there exists $z = z_1 \cdots z_n \in \gamma(X) \cap \gamma(Y)$ with $z_i \in \gamma(X[i])$. Let $\mathbb{P}_1, \ldots, \mathbb{P}_{n+1}$ be a corresponding concrete matching in Y. By Lemmas 10 and 12, we have $ESP^n[i] \preceq \mathbb{P}_i$ and $\mathbb{P}_{i+1} \preceq LEP^n[i]$ so $LSP^n[i] \prec ESP^n[i]$ implies $\mathbb{P}_i \preceq LEP^n[i-1] = LSP^n[i] \prec ESP^n[i] \preceq \mathbb{P}_i$ which is clearly impossible.

When $\textsc{Sweep}(X, Y) = \bot$ because the latest end $LEP[i]$ precedes its earliest end $EEP[i]$ the proof is dual: if $\mathbb{P}_1, \ldots, \mathbb{P}_{n+1}$ is a concrete matching for $z \in \gamma(X) \cap \gamma(Y)$ then $\mathbb{P}_{i+1} \preceq LEP^n[i] \prec EEP^n[i] = ESP^n[i+1] \preceq \mathbb{P}_{i+1}$. □

Lemma 14. Let $X, Y \in \overline{\mathbb{DS}}_{\Sigma}^{\lambda}$ with $n = |X|$ and $z = z_1 \ldots z_n \in \gamma(X) \cap \gamma(Y)$ such that $z_i \in \gamma(X[i])$ for $i = 1, \ldots, n$. Let $\mathbb{P}_1, \ldots, \mathbb{P}_{n+1}$ be a concrete matching for z in Y. If $\textsc{Sweep}(X, Y) = (ESP, LEP, EEP, LEP)$ then $ESP[i] \preceq \mathbb{P}_i \preceq LSP[i]$ and $EEP[i] \preceq \mathbb{P}_{i+1} \preceq LEP[i]$ for $i = 1, \ldots, n$.

Proof. As done above, let $(ESP^0, LEP^0) = \textsc{Init}(X, Y)$ and $ESP^i = \textsc{push}\mathrm{ESP}(X[i], Y, ESP^{i-1}, i)$ for $i = 1, \ldots, n$. As proven in Lemma 13, by applying repeatedly Lemmas 9—12 after loops of line 5 and 9 we get $ESP^n[i] \preceq \mathbb{P}_i$ and $\mathbb{P}_{i+1} \preceq LEP^n[i]$ for $i = 1, \ldots, n$.

After the loop of line 14 the invariant $ESP^n[i] \preceq LSP[i]$ holds and we have $\mathbb{P}_i \preceq LEP^n[i-1] = LSP[i]$ for $i = 2, \ldots, n$ (while $\mathbb{P}_1 = ESP^n[1] = LSP[1] = (1, 0)$). Analogously, after the loop of line 18 we have $EEP[i] \preceq LEP^n[i]$ and $\mathbb{P}_{i+1} \succeq ESP^n[i+1] = EEP[i]$ for $i = 1, \ldots, n-1$ (while $\mathbb{P}_{n+1} = LEP^n[n] =$

$EEP^n[n] = (|Y| + 1, 0))$. □

Lemma 15. Let $X, Y \in \overline{\mathbb{DS}}_\Sigma^\lambda$ with $n = |X|$ and $m = |Y|$. The worst-case complexity of SWEEP (X, Y) is $O(D(n + m))$ where $O(D)$ is the worst-case complexity of checking that the base of X and the base of Y are disjoint.

Proof. The INIT function calls STRETCH$^+$ n times, which costs $O(Dm)$ in the worst case. However, for $i = 2, \ldots, n$, the i-th call to STRETCH$^+$ considers only the blocks of Y *after* position $LEP[i - 1]$; in other terms, there is no overlap between regions $Y[LEP[i-1], LEP[i]]$. So in the worst case we consider each block of X and Y exactly once: initialising LEP costs $O(D(n + m))$. Since initialising ESP costs the same, the worst case complexity of INIT is $O(D(n + m))$.

Following the same reasoning, we have that pushing forward (backward) the earliest starts (latest ends) costs $O(D(n + m))$. The loops in lines 14–21 cost $O(n)$, so the overall cost of SWEEP is $O(D(n + m))$. □

Lemma 16. Let $X, Y \in \overline{\mathbb{DS}}_\Sigma^\lambda$ such that $M = \text{SWEEP}(X, Y) \neq \bot$. Then, $\text{REFINE}(X, M) = \bot$ implies $\gamma(X) \cap \gamma(Y) = \emptyset$.

Proof. Let us suppose $z = z_1 \ldots z_n \in \gamma(X) \cap \gamma(Y)$ with $z_i \in \gamma(X[i])$ for $i = 1, \ldots, n$ and let $\mathbb{P}_1, \ldots, \mathbb{P}_{n+1}$ be a concrete matching for z.

If $\text{REFINE}(X, M) = \bot$ there exists a block $X[h] = S^{l,u}$ such that $l' = \sum_{h=i}^{j} \nabla_{S,S_j}(l_h) > u$ where $Y[LSP[h], EEP[h]] = S_i^{l_i,u_i} \cdots S_j^{l_j,u_j}$ is the mandatory region of $X[h]$ in Y. This means $|z_h| > u$ because $z_h \in \gamma(Y[\mathbb{P}_i, \mathbb{P}_{i+1}])$ and, by Lemma 14, we have $\mathbb{P}_i \preceq LSP[h], \mathbb{P}_{i+1} \succeq EEP[h]$. But this is impossible because, being $z_h \in \gamma(X[h])$, has to be $|z_h| \leq u$. □

Lemma 17. Let $X, Y \in \overline{\mathbb{DS}}_\Sigma^\lambda$ with $n = |X|$ such that $M = \text{SWEEP}(X, Y) \neq \bot$ and $X' = \text{REFINE}(X, M) \neq \bot$. Then, $X' \sqsubseteq X$.

Proof. Let $\text{NORM}(X'_1, \ldots, X'_n) = \text{REFINE}(X, M) \neq \bot$. Given a generic block $X[h] = S^{l,u}$ of X, we have prove that $X'_h \sqsubseteq X[h]$, i.e., that $\gamma(X'_h) \subseteq \gamma(X[h])$.

If $\mathit{feas}_Y(X[h]) = S_1^{l_1,u_1} \cdots S_k^{l_k,u_k}$ and $\mathit{mand}_Y(X[h]) = S_i^{l_i,u_i} \cdots S_j^{l_j,u_j}$, let $S' = \bigcup_{p=1}^{k} S_i$, $l' = \sum_{p=i}^{j} \nabla_{S,S_j}(l_h)$, and $u' = \sum_{p=1}^{k} \nabla_{S,S_p}(\min(u_p, u - l' + l_p))$ as defined in the pseudo-code of Refine. There are two ways of refining $X[h]$:

(a) $X'_h = (S \cap S')^{\max(l,l'),\min(u,u')}$ if $l' = 0 \vee l' < l \vee u' > u$ (see line 20)

(b) $X'_h = \left[(S \cap S_i)^{l_i,\min(u_i,u-l'+l_i)} \mid S_i^{l_i,u_i} = M[i],\ i = 1,\ldots,|M|\right]$ with $M = \left(\bigcup_{p=1}^{i-1} S_h\right)^{0,\sum_{p=1}^{i-1} u_h} S_i^{l_i,u_i}, \ldots, S_j^{l_j,u_j} \left(\bigcup_{p=j+1}^{k} S_h\right)^{0,\sum_{p=j+1}^{k} u_p}$ otherwise.

In case (a) we have $\gamma(X'_h) = \{w \in (S \cap S')^* \mid \max(l,l') \leq |w| \leq \min(u,u')\} \subseteq \{w \in S^* \mid l \leq |w| \leq u\} = \gamma(X[h])$.

In case (b), where $l' \geq l$, we have $\gamma(X'_h) \subseteq \{w \in (S \cap S')^* \mid l' \leq |w| \leq u\} \subseteq \{w \in S^* \mid l \leq |w| \leq u\} = \gamma(X[h])$. □

Lemma 18. Let $X, Y \in \overline{\mathbb{DS}}_{\Sigma}^{\lambda}$ with $n = |X|$ such that $M = \text{Sweep}(X,Y) \neq \bot$, and let $X' = \text{Refine}(X, M) \neq \bot$. For each $z = z_1 \cdots z_n \in \gamma(X) \cap \gamma(Y)$ such that $z_i \in \gamma(X[i])$ for $i = 1, \ldots, n$ we have that $z_i \in \gamma(X'_i)$ where X'_i is the refinement of block $X[i]$.

Proof. Let $\text{norm}(X'_1, \ldots, X'_n) = \text{Refine}(X, M) \neq \bot$ and let $z = z_1 \ldots z_n \in \gamma(X) \cap \gamma(Y)$ with $z_i \in \gamma(X[i])$ for $i = 1, \ldots, n$. Let $\mathbb{P}_1, \ldots, \mathbb{P}_{n+1}$ be a concrete matching for z. Given a generic block $X[h] = S^{l,u}$ of X we have to prove that $z_h \in \gamma(X[h])$ implies $z_h \in \gamma(X'_h)$.

Let $\mathit{feas}_Y(X[h]) = S_1^{l_1,u_1} \cdots S_k^{l_k,u_k}$ and $\mathit{mand}_Y(X[h]) = S_i^{l_i,u_i} \cdots S_j^{l_j,u_j}$, let $S' = \bigcup_{p=1}^{k} S_i$, $l' = \sum_{p=i}^{j} \nabla_{S,S_j}(l_h)$, and $u' = \sum_{p=1}^{k} \nabla_{S,S_p}(\min(u_p, u - l' + l_p))$ as defined in the pseudo-code of Refine. As we seen above there are two ways of refining $X[h]$ with X'_h (see cases (a) and (b) in the proof of Lemma 17).

Let $Z_h = \{z_h[i] \mid i = 1, \ldots, |z_h|\}$ the set of characters occurring in z_h. Because $z_h \in \gamma(X[h]) \cap \gamma(\mathit{feas}_Y(X[h]))$, we have that $Z_h \subseteq S \cap S'$ and $l, l' \leq |z_h| \leq u, u'$ so if we refine $X[h]$ with $X'_h = (S \cap S')^{\max(l,l'),\min(u,u')}$, as we do in case (a), we surely have $z_h \in \gamma(X'_h)$.

In case (b) we have that $l' > 0$ and $l \leq l' \leq u' \leq u$. In this case z_h has the form $z_h = z'z''z'''$ where $z' \in \gamma(Y[\mathbb{P}_h, \mathit{LSP}[h]])$, $z'' \in \gamma(Y[\mathit{LSP}[h], \mathit{EEP}[h]])$

with $z'' \neq \epsilon$ and $z''' \in \gamma(Y[EEP[h], \mathbb{P}_{h+1}])$. So, $z' \in \gamma\left(\left(\bigcup_{p=1}^{i-1} S_p\right)^{0, \sum_{p=1}^{i-1} u_p}\right)$, $z'' \in \gamma(mand_Y(X[h]))$, and $z''' \in \gamma\left(\left(\bigcup_{p=j+1}^{k} S_p\right)^{0, \sum_{p=j+1}^{k} u_p}\right)$. Then, surely $z_h = z'z''z''' \in \gamma(M)$ (where M is the array of blocks defined in line 14 of Figure 12). Now, we have that $Z_h \subseteq S$, and for $p = 1, \dots, k$ the maximum number of characters u_p of $X[p] = S_p^{l_p, u_p}$ plus the mandatory characters $l' - l_p$ of all the other blocks cannot exceed u so $u_p + l' - l_p \leq u$ and thus $u_p \leq u - l' + l_p$. Hence, $z_h \in \left[(S \cap S_i)^{l_i, \min(u_i, u - l' + l_i)} \mid S_i^{l_i, u_i} = M[i],\ i = 1, \dots, |M|\right] = Z'_h$. □

Lemma 19. Let $X, Y \in \overline{\mathbb{DS}}_{\Sigma}^{\lambda}$ and $M = \textsc{Sweep}(X, Y) \neq \bot$. The worst-case complexity of $\textsc{Refine}$ (X, M) is:

$$O((D + U)nm + Im)$$

where $n = |X|$, $m = |Y|$, and $O(D), O(I), O(U)$ are respectively the worst-case complexity of checking set disjointness, computing set intersection, and set union between a base of X and a base of Y.

Proof. When there is maximum overlap between distinct and contiguous matching regions, we need to perform $O(m)$ set unions for each block of X in order to compute S'. Similarly, computing u' needs to check $O(m)$ set disjunctions for each block of X. Since mandatory regions never overlap, the cost of computing l' is instead $O(m)$.[2] The cost for computing S', l', u' is therefore $O(Unm) + O(Dnm) + O(m) = O((D + U)nm)$. The maximum number of intersections performed is $O(m)$ (no overlaps between mandatory regions) so the overall cost of $\textsc{Refine}$ is $O((D + U)nm + Im)$. □

Theorem 3. Let $X, Y \in \overline{\mathbb{DS}}_{\Sigma}^{\lambda}$. The worst-case complexity of $\textsc{equate}(X, Y)$ is:

$$O((D + U)(m^2 + nm) + I(n + m))$$

where $n = |X|$, $m = |Y|$, and $O(D), O(I), O(U)$ are respectively the worst-case complexity of checking set disjointness, computing set intersection, and set union between a base of X and a base of Y.

[2] We always assume that basic arithmetic operations like + and − cost $O(1)$.

Proof. As we seen in Lemma 19, the worst case complexity for computing X' is $O((D+U)nm + Im)$. Similarly, the worst case complexity for computing Y' is $O((D+U)mn' + In')$ where $n' = |X'|$. Now, because of the refining strategy defined in Section 4.1.2, we have $n' \leq n + m$ so Y' can be computed in $O((D+U)(m(n+m)) + I(n+(n+m))) = O((D+U)(m^2+mn) + I(n+m))$, which dominates the worst case complexity for X'. □

Corollary 1. Let $X, Y \in \overline{\mathbb{DS}}_{\Sigma}^{\lambda}$. If $n = |X|$, $m = |Y|$, and the worst-case complexity of checking set disjointness, computing set intersection, and set union between a base of X and a base of Y is $O(1)$, then the worst-case complexity of $\textsc{equate}(X, Y)$ is $O(m^2 + nm)$.

Proof. Straightforward from Theorem 3. □

Theorem 4. $\textsc{Equate}$ is an equation algorithm, i.e., for each $X, Y \in \overline{\mathbb{DS}}_{\Sigma}^{\lambda}$:

(i) if $\textsc{Equate}(X, Y) = \bot$, then $\gamma(X) \cap \gamma(Y) = \emptyset$

(ii) if $\textsc{Equate}(X, Y) = (X', Y')$, then (X', Y') refines (X, Y)

Proof. *(i)* If $\textsc{Equate}(X, Y) = \bot$ then either $\textsc{Sweep}$ or $\textsc{Refine}$ returns $\bot$, and this implies $\gamma(X) \cap \gamma(Y) = \emptyset$ by Lemma 13 and Lemma 16 respectively.

(ii) If $\textsc{Equate}(X, Y) = (X', Y') \neq \bot$ then by Lemma 17 $X' \sqsubseteq X$ and $Y' \sqsubseteq Y$ so $\gamma(X') \cap \gamma(Y') \subseteq \gamma(X) \cap \gamma(Y)$. We have to prove that $\gamma(X) \cap \gamma(Y) \subseteq \gamma(X') \cap \gamma(Y')$. Let $z = z_1 \cdots z_n \in \gamma(X) \cap \gamma(Y)$ where $z_i \in \gamma(X[i])$ for $i = 1, \ldots, n$. If $X' = \textsc{Refine}(X, \textsc{Sweep}(X, Y))$ then by Lemma 18 we have $z_i \in X'_i$ where X'_i is the refinement of $X[i]$ so $z = z_1 \cdots z_n \in \gamma(X'_1 \cdots X'_n) = \gamma(\textsc{norm}(X'_1 \cdots X'_n)) = \gamma(X')$. So, $z \in \gamma(X')$ and being also $z \in \gamma(Y)$ we have $z \in \gamma(Y) \cap \gamma(X')$. Therefore, if $Y' = \textsc{Refine}(Y, \textsc{Sweep}(Y, X'))$ by Lemma 18 we have $z \in \gamma(Y')$. Hence, $z \in \gamma(X') \cap \gamma(Y')$. □

Theorem 5. Let $X, Y \in \overline{\mathbb{DS}}_{\Sigma}^{\lambda}$. If $lb(X[i]) = ub(X[i])$ and $lb(Y[j]) = ub(Y[j])$ for $i = 1, \ldots, |X|$ and $j = 1, \ldots, |Y|$ then:

$$\textsc{Equate}(X, Y) = \bot \iff \gamma(X) \cap \gamma(Y) = \emptyset.$$

Proof. The implication ($\textsc{Equate}(X, Y) = \bot$) $\Rightarrow \gamma(X) \cap \gamma(Y) = \emptyset$ is proven in Theorem 4*(i)*. Let us prove the reverse implication.

Let $n_i = lb(X[i]) = ub(X[i])$ for $i = 1, \ldots, |X|$ and $n = \sum_{i=1}^{|X|} n_i$; let $m_j = lb(Y[j]) = ub(Y[j])$ for $j = 1, \ldots, |Y|$ and $m = \sum_{j=1}^{|Y|} m_j$. If $n \neq m$, then either $n < m$ or $n > m$. The first case is trivial because $\textsc{stretch}^+$ cannot reach the end of Y so $\textsc{Init}(X, Y) = (\bot, \bot)$. In the second case, there are $n - m > 0$ characters of X that cannot be matched. So, if $\textsc{Init}(X, Y) \neq (\bot, \bot)$, then surely $\textsc{Equate}(X, Y) = \bot$ because there is at least a block $X[i]$ for which $\textsc{push}$ESP returns $\bot$ (it reaches position $(|Y| + 1, 0)$).

So, let us assume $n = m$. Also in this case $\textsc{Init}(X, Y) = (\bot, \bot)$, because if $\gamma(X) \cap \gamma(Y) = \emptyset$ there are at least two blocks $X[i]$, $Y[j]$ such that $X[i]$ cannot consume entirely $Y[j]$, i.e., there are $0 < k \leq n_i$ characters of the base of $X[i]$ that cannot be matched. So $\textsc{stretch}^+$ cannot reach the end of Y (the latest end of the last block of X is at least k positions before the last position of Y). □

Theorem 6. The equation algorithm $\textsc{Equate}$ is:

(i) *not optimal*: $\textsc{Equate}(X, Y)$ is not always minimal element of $(\mathit{Ref}_{X,Y}, \sqsubseteq)$

(ii) *not idempotent*: $\textsc{Equate}(X, Y) = (X', Y')$ does not always imply that $\textsc{Equate}(X', Y') = (X', Y')$.

Proof. *(i)* Consider block $B = \{a, b\}^{3,3}$ such that $\mathit{feas}_Y(B) = \mathit{mand}_Y(B) = \{a\}^{1,1}\{a, b\}^{0,1}\{b\}^{0,1}\{a\}^{1,1}$. Because $lb(Y[1]) + \cdots + lb(Y[4]) = 2 < 3$, we cannot further refine B. However, an optimal refinement would be $\{a\}^{1,1}\{a, b\}^{1,1}\{a\}^{1,1}$.

(ii) Consider $X = \{a, b\}^{0,1}\{b\}^{0,1}$ and $Y = \{b, c\}^{0,1}\{b\}^{1,1}$. We have that $\textsc{Equate}(X, Y) = (\{b\}^{0,2}, \{b\}^{1,2}) \neq \textsc{Equate}(\{b\}^{0,2}, \{b\}^{1,2}) = (\{b\}^{1,2}, \{b\}^{1,2})$. □