A Novel Approach to String Constraint Solving

Roberto Amadini[1], Graeme Gange[1], Peter J. Stuckey[1], and Guido Tack[2]

[1] University of Melbourne, Victoria, Australia
[2] Monash University, Australia

Abstract. String processing is ubiquitous across computer science, and arguably more so in web programming. In order to reason about programs manipulating strings we need to solve constraints over strings. In Constraint Programming, the only approaches we are aware for representing string variables—having bounded yet possibly unknown size—degrade when the maximum possible string length becomes too large. In this paper, we introduce a novel approach that decouples the size of the string representation from its maximum length. The domain of a string variable is dynamically represented by a simplified regular expression that we called a *dashed string*, and the constraint solving relies on propagation of information based on equations between dashed strings. We implemented this approach in G-Strings, a new string solver—built on top of Gecode solver—that already shows some promising results.

1 Introduction

Strings are fundamental datatypes in all the modern programming languages. String analysis [10, 23, 25] is needed in several real-life applications such as test-case generation [12], program analysis [8], model checking [17], web security [5], and bioinformatics [4]. Reasoning over strings requires the processing of constraints such as (in-)equality, concatenation, length, and so on.

A natural candidate to tackle string constraints is the *Constraint Programming* (CP) paradigm [19]. Unfortunately, practically no CP solver natively supports string constraints. To the best of our knowledge, the only exception is Gecode+$\mathbb{S}$ [29, 31], an extension of Gecode solver [18]. Gecode+$\mathbb{S}$ relies on *Bounded-Length Sequence* (BLS) string variables [31], implemented with dynamic lists of bitsets. Empirical results shows that Gecode+$\mathbb{S}$ is usually better than dedicated string solvers such as Hampi [22], Kaluza [28], and Sushi [14].

The *MiniZinc* [26] modelling language was recently extended to include string variables and constraints [1]. A MiniZinc library for converting MiniZinc models with strings into equivalent FlatZinc instances containing only integer variables has also been provided. In this way every solver supporting FlatZinc can now solve a MiniZinc model with strings, by converting each string of maximum length n into an array of n integer variables. This allowed the comparison of native string solvers like Gecode+$\mathbb{S}$ against state-of-the-art CP solvers using a decomposition. Results indicate that native support for string variables usually pays off, but not always, in which case the technology of the best solver varies.

Having bounded-length strings is reasonable (note that satisfiability with unbounded-length strings is not decidable in general [16]) and enables finite-domain variables. The crucial issue here is to decide a maximum length ℓ for string variables. On the one hand, too small a value for ℓ may exclude solutions for important classes of string applications, e.g., where a variable represents a long XML string or part of a DNA string. On the other hand, too large a value for ℓ can significantly worsen performance even for relatively simple problems.

A common drawback, shared by both the Gecode+$\mathbb{S}$ solver and the approaches statically mapping string variables to arrays of integer variables, is that the solving process is coupled to the maximum string length ℓ. Indeed, the performance of these approaches degrade when ℓ becomes bigger and bigger even for relatively simple problems.

In this paper we address this problem by proposing a novel approach to string representation in CP solvers. The new representation is based on a restricted class of regular expressions, which we refer to as *dashed strings*. Given an alphabet Σ and a maximum string length ℓ, a dashed string consists of an ordered sequence $S_1^{l_1,u_1} S_2^{l_2,u_2} \cdots S_k^{l_k,u_k}$ of $0 < k \leq \ell$ *blocks*, where $S_i \subseteq \Sigma$ and $0 \leq l_i \leq u_i \leq \ell$ for $i = 1, \ldots, k$, and $\Sigma_{i=1}^{k} l_i \leq \ell$. Each block $S_i^{l_i,u_i}$ represents the set of all the strings of Σ having length in $[l_i, u_i]$ and characters in S_i. The idea of dashed strings takes inspiration from the *Bricks* abstract domain of [10]. In that paper, however, each block refers to a set of strings of Σ^* (while in our representation refers to a set of characters of Σ) and some workarounds are used in order to make the abstract domain a lattice.

We use dashed strings to model the domain of string variables. The propagators for string constraints rely on the notion of *equation* between dashed strings in order to possibly narrow each string domain to a concrete string (i.e., to a dashed string representing a single string of Σ^*). We also define a branching strategy that aims to select the strings with minimal length that satisfy all the constraints, using the lexicographic order for breaking ties.

Following the Gecode+$\mathbb{S}$ approach, we use Gecode [18] as a starting point for implementing our solver. The resulting solver, that we called G-Strings, already shows promising results. We compared its performance against: the aforementioned Gecode+$\mathbb{S}$; the state-of-the-art CP solvers Chuffed, Gecode, iZplus; the SMT solver Z3str3 [34], a string theory plug-in built on top of Z3 solver. Results indicate that, despite still being in a preliminary stage, G-Strings often outperforms all such solvers. However, there are class of problems where it has worse performance. This leaves room for future enhancements.

The original contributions of this paper are: *(i)* new abstractions and algorithms for modelling and manipulating the domain of string variables; *(ii)* new propagators and branchers for string constraint solving; *(iii)* the implementation and the evaluation of a new string solver.

Paper Structure. Section 2 gives preliminary notions. Section 3 defines the dashed strings and the algorithms we used in Section 4 for implementing string variables and constraints. Section 5 provides an evaluation of our approach, before we conclude in Section 6.

2 Preliminaries

Given a finite alphabet $\Sigma = \{a_1, \ldots, a_n\}$, a string $x \in \Sigma^*$ is a finite sequence of $|x| \geq 0$ characters of Σ, where $|x|$ is the length of x. We omit the distinction between characters and strings of unary length. The interval $[a, b]$ will be denoted also with $\{a..b\}$.

The concatenation of $x, y \in \Sigma$ is denoted by $x \cdot y$ (or simply xy when not ambiguous) while x^n denotes the iterated concatenation of x for n times (where x^0 is the empty string ϵ). We generalise this definition to set of strings: given $X, Y \subseteq \Sigma$, we denote with $X \cdot Y = \{xy \mid x \in X, y \in Y\}$ (or simply with XY) their concatenation and with X^n the iterated concatenation of X for n times (where $X^0 = \{\epsilon\}$).

In this work we focus on bounded-length strings: fixed a maximum length ℓ, we consider only strings in the universe $\mathbb{S} = \bigcup_{i=0}^{\ell} \Sigma^i$. Clearly $\mathbb{S}$ is not closed under concatenation. We extend the canonical definition of *Constraint Satisfaction Problem* (CSP) by including string variables and constraints. Formally, a CSP is a triple $\langle \mathcal{V}, \mathcal{D}, \mathcal{C} \rangle$ consisting of a set of variables $\mathcal{V}$, each of which associated with a domain $\mathcal{D}(x) \in \mathcal{D}$ of values that $x \in \mathcal{V}$ could take, and a set of constraints $\mathcal{C}$ defining all the feasible assignments of values to variables. The goal is to find a solution, i.e., a variable assignment satisfying all the constraints of $\mathcal{C}$.

In addition to "standard" integer variables and constraints, in this paper we consider string variables x having domain $D(x) \subseteq \mathbb{S}$, and string constraints over string variables. We also consider constraints involving both string and integer variables, e.g., the length constraint $|x| = n$ or the power constraint $x^n = y$ where x, y are string variables and n is an integer variable.

3 Dashed Strings

A *dashed string* is a restricted regular expression denoting a finite set of concrete strings. The rationale behind this representation is to facilitate a compact and dynamic representation of set of strings of unknown length, without statically pre-allocating an arbitrarily large number of elements. Moreover, as we shall see later, dashed strings enable us to deal with concatenation—arguably the most common string operation—in a natural way.

Below we give the formal definition of dashed string, and then show how we propagate information over equations between dashed strings. Before that, we give an informal intuition of what a dashed string is. The name "dashed" comes from a graphical interpretation of $S = S_1^{l_1,u_1} S_2^{l_2,u_2} \cdots S_k^{l_k,u_k}$ 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 denoted by S; the dashed segment indicates that k characters of S_i, with $0 \leq k \leq u_i - l_i$, *may* occur. Consider Fig. 1, illustrating dashed string $S = \{\texttt{B,b}\}^{1,1}\{\texttt{o}\}^{2,4}\{\texttt{m}\}^{1,1}\{\texttt{!}\}^{0,3}$. Each string represented by S starts with `B` or `b`, followed by 2 to 4 `o`s, one `m`, then 0 to 3 `!`s.

Fig. 1. Graphical representation of $\{\texttt{B,b}\}^{1,1}\{\texttt{o}\}^{2,4}\{\texttt{m}\}^{1,1}\{\texttt{!}\}^{0,3}$.

3.1 Definition

Let us fix the alphabet Σ, the maximum length ℓ, and the universe $\mathbb{S} = \bigcup_{i=0}^{n} \Sigma^{\ell}$. A *dashed string* of *length* k is defined by a concatenation of $0 < k \leq \ell$ blocks $S_1^{l_1,u_1} S_2^{l_2,u_2} \cdots S_k^{l_k,u_k}$, where $S_i \subseteq \Sigma$ and $0 \leq l_i \leq u_i \leq \ell$ for $i = 1, \ldots, k$, and $\Sigma_{i=1}^{k} l_i \leq \ell$. For block $S_i^{l_1,u_1}$, we call S_i the *base* and (l_i, u_i) the *cardinality*. $S[i]$ indicates the i-th block of dashed string S, and $|S|$ the number of blocks (i.e., the length of S). $\mathbb{DS}$ denotes the set of all dashed strings. We do not distinguish blocks from dashed strings of unary length.

Let $\gamma(S^{l,u}) = \{x \in S^* \mid l \leq |x| \leq u\}$ be the language denoted by block $S^{l,u}$. In particular the *null element* $\emptyset^{0,0}$ is such that $\gamma(\emptyset^{0,0}) = \{\epsilon\}$. We extend γ to dashed strings: $\gamma(S_1^{l_1,u_1} \cdots S_k^{l_k,u_k}) = (\gamma(S_1^{l_1,u_1}) \cdots \gamma(S_k^{l_k,u_k})) \cap \mathbb{S}$. A dashed string S is *known* if $|\gamma(S)| = 1$, i.e., it represents a single string.

We say $S^{l,u}$ is *coverable by* $T^{l',u'}$ if some string in $\gamma(S^{l,u})$ is a prefix of a string in $\gamma(T^{l',u'})$ (formally, if $l = 0 \vee (l \leq u' \wedge S \cap T \neq \emptyset)$). S and T are *incompatible* if neither S nor T is coverable by the other.

Given $S, T \in \mathbb{DS}$ we define the relation $S \sqsubseteq T \iff \gamma(S) \subseteq \gamma(T)$. Intuitively, operator $\sqsubseteq$ models the relation "is more precise than" between dashed strings.

Unfortunately, although $\sqsubseteq$ is a partial order over $\mathbb{DS}$, the structure $(\mathbb{DS}, \sqsubseteq)$ does not form in general a lattice. This means that it might not exist a greatest lower bound (or a least upper bound) for two given dashed strings $S, T \in \mathbb{DS}$. Proposition 1 proves this statement. Unlike other frameworks (e.g., Abstract Interpretation [11]), Constraint Programming does not require lattice structures to preserve the soundness of constraint solving. However, as we shall see, care must be taken in order to avoid leaks of feasible solutions or infinite propagations.

Proposition 1 *The structure* $(\mathbb{DS}, \sqsubseteq)$ *is not a lattice.*

Proof. Let $\Sigma = \{a, b\}$, $S = \{a\}^{1,1}\{b\}^{1,1}$, and $T = \{b\}^{1,1}\{a\}^{1,1}$. We prove that there is no least upper bound in $(\mathbb{DS}, \sqsubseteq)$ for S and T, nor a greatest lower bound for $S' = \{a\}^{0,1}\{b\}^{1,1}\{a\}^{0,1}$ and $T' = \{b\}^{0,1}\{a\}^{1,1}\{b\}^{0,1}$.

We first observe that S', T' and $\{a, b\}^{2,2}$ are the minimal elements greater than S, T according to $\sqsubseteq$. However, they are incomparable with $\sqsubseteq$ since $\gamma(S') = \{b, ab, aba\}$, $\gamma(T') = \{a, ba, bab\}$ and $\gamma(\{a, b\}^{2,2}) = \{aa, ab, ba, bb\}$. Thus, there not exist a least upper bound for S, T. The greatest lower bound of S', T' does not exists because the maximal elements smaller than S', T' are $\{a, b\}^{1,1}\{a, b\}^{0,2}$ and $\{a, b\}^{0,2}\{a, b\}^{1,1}$, which are incomparable according to $\sqsubseteq$. □

The γ function is not injective. For example, for $S = \{a\}^{0,1}\{a\}^{0,1}$ and $T = \{a\}^{0,2}$ we have $\gamma(S) = \gamma(T) = \{\epsilon, a, aa\}$. To remove redundant configurations, and minimise the length of a dashed string, we introduce the notion of *normalisation*. A dashed string $S = S_1^{l_1,u_1} \cdots S_k^{l_k,u_k}$ is normalised if and only if:

(i) $S_i \neq S_{i+1}, \quad \text{for } i = 1, \ldots, k-1.$
(ii) $S_i = \emptyset \iff l_i = u_i = 0, \quad \text{for } i = 1, \ldots, k;$
(iii) $S = \emptyset^{0,0} \vee S_i \neq \emptyset, \quad \text{for } i = 1, \ldots, k;$

Condition (i) says that each adjacent base has to be distinct, since blocks $S^{l,u}$ and $S^{l',u'}$ are equivalent to $S^{l+l',u+u'}$. Condition (ii) avoid multiple configurations for the null element $\emptyset^{0,0}$. Condition (iii) forbids the redundant use of $\emptyset^{0,0}$, being in general $\gamma(B \cdot \emptyset^{0,0}) = \gamma(\emptyset^{0,0} \cdot B) = \gamma(B)$.

We omit the definition of the normalisation, that unsurprisingly has linear cost $O(|S|)$ for normalising a dashed string S. Note that if $S, S' \in \mathbb{DS}$ are normalised then $S = S' \iff \gamma(S) = \gamma(S')$.

Finally, we define the *size* $\|S^{l,u}\|$ of a block $S^{l,u}$ 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, i.e., $\|S\| = \Pi_{i=1}^{k} \|S_i^{l_i,u_i}\|$ for each dashed string $S = S_1^{l_1,u_1} \cdots S_k^{l_k,u_k}$.

The size of a dashed string gives a measure of the number of concrete strings it represents. Note that, while for a block $S^{l,u}$ we have that $\|S^{l,u}\| = |\gamma(S^{l,u})|$, for a generic dashed string $S \in \mathbb{DS}$ we have that $\|S\| \geq |\gamma(S)|$ but not $\|S\| = |\gamma(S)|$. For example, if $S = \{a\}^{0,1}\{a,b\}^{0,1}$, we have $|\gamma(S)| = |\{\epsilon, a, b, aa, ab\}| = 5$ while $\|S\| = \|\{a\}^{0,1}\| \cdot \|\{a,b\}^{0,1}\| = 2 \cdot 3 = 6$.

3.2 Equating Dashed Strings

We use dashed strings as a domain abstraction for string variables. Following the standard CP framework, each variable domain is iteratively narrowed until it becomes a single value, that will be assigned to the variable, or it becomes empty, meaning that the problem is unsatisfiable.

In this context, we have to iteratively "narrow" a dashed string S until it becomes known or the unsatisfiability is detected. Things are tricky here since $(\mathbb{DS}, \sqsubseteq)$ does not form a lattice. Consider for example two string variables x and y, having domain S' and T' as in the proof of Proposition 1. There is not an unique way to prune the domain of x and y when it comes to propagate the equality constraint $x = y$, since there is no greatest lower bound for S' and T'.

Regardless of the choice of how pruning, a propagator for a string constraint must be at least *sound* (it never prunes values that can appear in a solution) and *contracting* (is only allowed to remove values).

The core algorithm that we adopted for string constraint propagation is based on the *equation* of two dashed strings. Informally, equating two dashed strings S and T means, firstly, to verify that there exists at least a concrete string shared by both $\gamma(S)$ and $\gamma(T)$ and, if so, to find a representation for S and T that includes all the strings of $\gamma(S) \cap \gamma(T)$ and removes the most values not belonging to $\gamma(S) \cap \gamma(T)$. More formally, this problem consists in finding,

Algorithm 1 EQUATE algorithm.

```
1: function EQUATE(S, T)
2:     Input: Dashed strings S = S_1^{a_1,b_1} ··· S_n^{a_n,b_n} and T = T_1^{c_1,d_1} ··· T_m^{c_m,d_m}.
3:     Output: true if S and T are equatable; false otherwise.
4:     Matches ← NoGoods ← ∅
5:     CHECK(S, 1, S_1^{a_1,b_1}, T, 1, T_1^{c_1,d_1}, Matches, NoGoods)
6:     if Matches = ∅ then
7:         return false
8:     SplitS, SplitT ← SPLIT(S, T, Matches)
9:     S̃, T̃ ← MERGE(SplitS, SplitT)
10:    UPDATE(S, T, S̃, T̃)
11:    return true
```

if feasible, two dashed strings S' and T' such that: *(i)* $S' \sqsubseteq S$, $T' \sqsubseteq T$; *(ii)* $\gamma(S') \cap \gamma(T') = \gamma(S) \cap \gamma(T)$. We could add a third condition stating that there not exist two dashed strings S'', T'' such that $S'' \sqsubset S'$ and $T'' \sqsubset T$. However, this requirement makes the propagation too difficult.

We address this equation problem—that can be seen as a semantic unification problem—with a multiphase strategy, where dashed strings S are T in input are processed and possibly updated with two "refined" dashed strings S' and T'. These phases, namely *checking*, *splitting*, *merging*, and *updating*, are explained below. We use pseudo-code and we abstract as much as possible the technicalities, referring to a running example rather than going into the implementation details. The actual code we developed integrates and optimise these four stages that, for the sake of readability, here we present simplified and separately.

The main algorithm is summarised in Algorithm 1. Taking as input two dashed strings $S = S_1^{a_1,b_1} \cdots S_n^{a_n,b_n}$ and $T = T_1^{c_1,d_1} \cdots T_m^{c_m,d_m}$, that we assume already normalised, EQUATE initialises variables *Matches* and *NoGoods* to the empty set (we shall explain their meaning below) and then CHECK is called.

Checking CHECK (Algorithm 2) both tests if S and T are equatable, and constructs a directed acyclic graph *Matches* encoding the set of solutions. SPLIT will then reconstruct *Matches* into dashed strings for S and T.

CHECK uses a top-down dynamic programming approach, recursively matching suffixes of S and T. In any matching, the first block of either S or T must finish first. If S, we compute what remains available of the T-block, and match the tail of S with the remnant of T (similarly for T) – this is done in lines 11–18. Lines 2–10 cover early termination, where S or T reached the end or have incompatible initial blocks. FAIL saves failed computations in *NoGoods* before returning *false*.

For a successful computation, the sequence of partial blocks consumed by calls to CHECK encode possible solutions to $S = T$. CHECK builds a directed acyclic graph representing the set of such sequences. Each sequence will be called a *match*. For simplicity, we elide details of how *Matches* is maintained – essentially, it amounts to recording the graph of successful CHECK calls.

Algorithm 2 CHECK algorithm.

```
1: function CHECK(S, i, S_i^{l_i,u_i}, T, j, T_j^{l_j,u_j}, Matches, NoGoods)
2:     if (i, l_i, u_i, j, l_j, u_j) ∈ NoGoods then return false
3:     if i = |S| + 1 then                                   ▷ Reached end of S
4:         if l_j = c_{j+1} = ··· = c_m = 0 then return NEWMATCH(Matches)
5:         else return FAIL(NoGoods, S_i^{l_i,u_i}, T_j^{l_j,u_j})
6:     else if j = |T| + 1 then                              ▷ Reached end of T
7:         if l_i = a_{i+1} = ··· = a_n = 0 then return NEWMATCH(Matches)
8:         else return FAIL(NoGoods, S_i^{l_i,u_i}, T_j^{l_j,u_j})
9:     else if l_i > 0 ∧ l_j > 0 ∧ S_i ∩ T_j = ∅ then        ▷ Incompatible blocks
10:        return FAIL(NoGoods, S_i^{l_i,u_i}, T_j^{l_j,u_j})
11:    if l_i = 0 ∨ (S_i ∩ T_j ≠ ∅ ∧ l_i ≤ u_j) then          ▷ S_i^{l_i,u_i} coverable
12:        Rem_T ← S_i ∩ T_j ≠ ∅ ? T_j^{max(0,l_j−u_i),u_j−l_i} : T_j^{l_j,u_j}
13:        Checks_S ← CHECK(S, i + 1, S[i + 1], T, j, Rem_T, Matches, NoGoods)
14:    else Checks_S ← false
15:    if l_j = 0 ∨ (S_i ∩ T_j ≠ ∅ ∧ l_j ≤ u_i) then          ▷ T_j^{l_j,u_j} coverable
16:        Rem_S ← S_i ∩ T_j ≠ ∅ ? S_i^{max(0,l_i−u_j),u_i−l_j} : S_i^{l_i,u_i}
17:        Check_T ← CHECK(S, i, Rem_S, T, j + 1, T[j + 1], Matches, NoGoods)
18:    else Check_T ← false
19:    if ¬(Checks_S ∨ Check_T) then
20:        return FAIL(NoGoods, S_i^{l_i,u_i}, T_j^{l_j,u_j})
21:    return Checks_S ∨ Check_T
```

CHECK defines a *match-tree*, i.e., a binary tree where: *(i)* each node is a pair of blocks (the root is $\langle S_1^{l_1,u_1}, T_1^{l_1,u_1}\rangle$); *(ii)* there is a branch from $\langle S_i^{l_i,u_i}, T_j^{l_j,u_j}\rangle$ to left child $\langle S_{i+1}^{l_{i+1},u_{i+1}}, T_j^{l'_j,u'_j}\rangle$ if $S_i^{l_i,u_i}$ is coverable by $T_j^{l_j,u_j}$ and $T_j^{l'_j,u'_j}$ is the corresponding remnant (the dual definition applies to the right child); *(iii)* a leaf is either a success (a match is found) or a failure (due to incompatible blocks).

A match tree for $S = \{a..c\}^{0,30}\{d\}^{5,5}\{c..f\}^{0,2}$ and $T = \{b..d\}^{26,26}\{f\}^{1,1}$ is shown in Fig. 2 (ignoring for now dashed arrows). Failures are denoted with $\times$, while successes with $\Diamond$. A match identifies a path from root to $\Diamond$ representable with the coordinates $\langle i, j\rangle$ of each node $\langle S_i^{l_i,u_i}, T_j^{l_j,u_j}\rangle$. For each transition $\langle i, j\rangle \to \langle i', j'\rangle$ the invariant $(i' = i \wedge j' = j + 1) \vee (j' = j \wedge i' = i + 1)$ holds. We can thus see each transition as a move of length 1 in a $n \times m$ grid.

All the three matches of Fig. 2 are coloured in green. In particular the (partial) match $[\langle 1, 1\rangle, \langle 2, 1\rangle, \langle 2, 2\rangle, \langle 3, 2\rangle]$ is truncated. This is because the pair $\langle\{c..f\}_3^{0,2}, \{f\}_2^{1,1}\rangle$ has already been examined before and thus there is no need to rebuild the subtree again. Even if not explicitly detailed, our actual implementation defines a mechanism—similar to the recording of failures—that enables the caching of already visited nodes, and hence to prune redundant computations.

From Fig. 2 we can see for example that the rightmost subtree rooted in $\langle 1, 1\rangle$ always fails. This is because if the block $\{b..d\}^{26,26}$ is entirely covered by $\{a..c\}^{0,30}$, then there is no other block in S that can cover $\{f\}^{1,1}$.

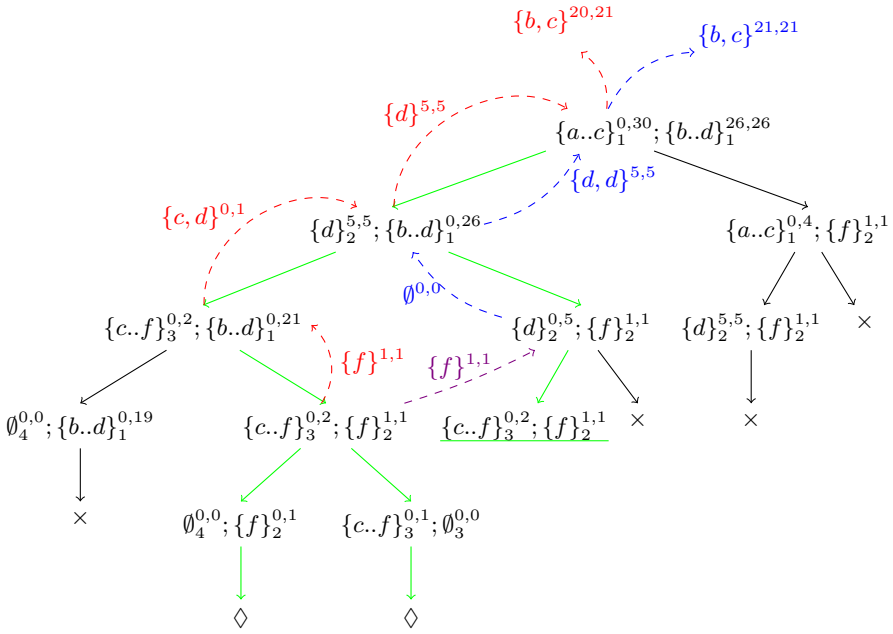

Fig. 2. Match tree for $S = \{a..c\}^{0,30}\{d\}^{5,5}\{c..f\}^{0,2}$ and $T = \{b..d\}^{26,26}\{f\}^{1,1}$.

Lemma 1. *The worst case complexity of* EQUATE *is* $O(nm\max(n,m))$.

Proof. Each recursive call in $\textsc{Check}(S, i, S_i^{l_i,u_i}, T, j, T_j^{l_j,u_j}, \mathit{Matches}, \mathit{NoGoods})$ removes one block completely from S or from T so one between $S_i^{l_i,u_i}$ and $T_j^{l_j,u_j}$ is an original block and the other one is a remnant block. If it is a remnant block it can only be changed $\max(n,m)$ times, since it runs out of blocks to cover. Hence the total number of different calls is $O(nm\max(n,m))$. □

Splitting Suppose CHECK returned *true* (otherwise EQUATE terminates). Thus $\gamma(S)\cap\gamma(T) \neq \emptyset$. However, we would like to refine S and T in order to prune the most values not belonging to $\gamma(S)\cap\gamma(T)$. In this second phase we take advantage of the matches collected in *Matches* for possibly *splitting* the blocks of S and T. We aim to find (partial) maps $\sigma_S : [1, |S|] \to \mathbb{DS}$ such that $\sigma_S(i) \sqsubseteq S[i]$. In this way, by definition, $\sigma_S(1)\cdots\sigma_S(n) \sqsubseteq S$ (same applies for T).

As mentioned, a match for $S = S_1^{a_1,b_1}\cdots S_n^{a_n,b_n}$ and $T = T_1^{c_1,d_1}\cdots T_m^{c_m,d_m}$ can be described by a path in the match tree. In particular, a sub-path of the form $[\langle i,j\rangle, \langle i,j+1\rangle, \ldots, \langle i,j+k\rangle]$ enables us to split block $S_i^{l_i,u_i}$ of S into a concatenation of k blocks $(S_i \cap T_j)^{\alpha_k,\beta_k}(S_i \cap T_{j+1})^{\alpha_{k-1},\beta_{k-1}}\cdots(S_i \cap T_{j+k})^{\alpha_1,\beta_1}$ where cardinalities α_h, β_h are computed iteratively by a bottom-up approach that we explain below.

Algorithm 3 SPLIT algorithm.

```
1: function SPLIT(S, T, Matches)
2:     k ← 1;   splitS ← splitT ← [ ];   Matches' ← TRIM(Matches)
3:     for M_k ∈ Matches' do
4:         sl_i ← su_i ← sl_j ← su_j ← 0;   split_S^k ← split_T^k ← { };   last_i ← −1
5:         for ⟨S_i^{l_i,u_i}, T_j^{l_j,u_j}⟩ ∈ M_k do
6:             R ← S_i ∩ T_j
7:             if i = last_i then                                   ▷ direction ↖
8:                 l ← max(l_i − su_j, l_j);   u ← min(u_i − sl_j, u_j)
9:                 if l > u then  l ← u ← 0
10:                split_S^k[i] ← NORM([R^{l,u}] + split_S^k[i]);   split_T^k[j] ← [R^{l,u}]
11:                sl_i ← l;   su_i ← u;   sl_j ← sl_j + l;   su_j ← su_j + u
12:            else                                                 ▷ direction ↗
13:                l ← max(l_j − su_i, l_i);   u ← min(u_j − sl_i, u_i)
14:                if l > u then  l ← u ← 0
15:                split_T^k[j] ← NORM([R^{l,u}] + split_T^k[j]);   split_S^k[i] ← [R^{l,u}]
16:                sl_j ← l;   su_j ← u;   sl_i ← sl_i + l;   su_i ← su_i + u
17:            last_i ← i
18:        splitS ← splitS + [split_S^k];   splitT ← splitT + [split_T^k];   k ← k + 1
19:    return splitS, splitT
```

Informally speaking, we "climb back up" the match-tree from the leaves to the root. Each move from a child node to its father has a *direction* that can be top-right (if it is a left child) or top-left (for a right child). If we "walk straight" in the same direction, for each node of the path there is always one block B that stays fixed, while the other blocks $B', B'', B''', \ldots$ vary along the way. So we can split B into sub-blocks thanks to the information given by $B', B'', B''', \ldots$, i.e., by all the blocks covered by B along the way. Care must be taken when computing the cardinality of the sub-blocks: we have to consider the cumulative cardinality of $B', B'', B''', \ldots$ and not only the block currently being examined. When the direction changes, we "turn" in the new direction. This process is repeated until the root is reached.

The SPLIT algorithm listed in Algorithm 3 performs the backward propagation from the leaves to the root. We consider each match $M_k \in \mathit{Matches}'$ where $\mathit{Matches}' = \textsc{Trim}(\mathit{Matches})$ and the TRIM function removes from *Matches* all the pairs of the form $\langle \emptyset^{0,0}, B\rangle$ and $\langle B, \emptyset^{0,0}\rangle$ (useless in this context). For each M_k we have a map split_S^k (resp., split_T^k) mapping each index $i \in [1, n]$ to a list of blocks $\mathit{split}_S^k[i]$ defining a splitting of $S[i]$ (resp., mapping each index $j \in [1, m]$ to $\mathit{split}_T^k[j]$). SPLIT returns two lists $\mathit{splitS} = [\mathit{split}_S^1, \ldots, \mathit{split}_S^p]$ and $\mathit{splitT} = [\mathit{split}_T^1, \ldots, \mathit{split}_T^p]$ where $p = |\mathit{Matches}'|$.

Each match M_k is already in reversed order, i.e., from leaf to root, since each match is registered following the stack of recursive calls to CHECK.

If we are going top-right (lines 7–11) then we are splitting on S_i. We then add at the head of the current split $\mathit{split}_S^k[i]$ the element $R^{l,u}$ with $R = S_i \cap T_j$, $l = \max(l_i - \mathit{su}_j, l_j)$ and $u = \min(u_i - \mathit{sl}_j, u_j)$. We store in variable sl_j (resp., su_j)

the cumulative sum of the lower bounds (resp., upper bounds) encountered when walking in the same direction. The + operator is the concatenation between lists.

Note that splitting a block $S^{l,u}$ into $S' = (S \cap T_1)^{l_1,u_1} \dots (S \cap T_k)^{l_k,u_k}$ always refines the base S, since $(S \cap T_1) \cup \dots \cup (S \cap T_k) \subseteq S$, but in general does not ensure that S' is normalised and, most important, that $\gamma(S') \subseteq \gamma(S^{l,u})$. Consider matching $S = \{a, b\}^{2,3}$ with $T = \{a, c\}^{0,2}\{b, c\}^{0,2}$. After matching, we would obtain a split $S' = \{a\}^{0,2}\{b\}^{0,2}$ for $S[1]$. While S' refines the base of $S[1]$, the loss of cardinality information introduces new (spurious) strings (e.g., the string $aaaa \in \gamma(S') \setminus \gamma(S)$). We must therefore consider the cardinality of the original block when splitting. This is performed by a function NORMthat, when splitting $S^{l,u}$ into $S' = (S \cap T_1)^{l_1,u_1} \dots (S \cap T_1)^{l_k,u_k}$, first checks if $\Sigma_{i=1}^{k} l_i \geq l$ and $\Sigma_{i=1}^{k} u_i \leq u$. If so, it returns the normalisation of S'. Otherwise, it returns the block $((S \cap T_1) \cup \dots \cup (S \cap T_k))^{l,u}$.

The opposite direction (lines 12–16) is totally symmetric. To identify the direction it is enough to check the value of $last_i$, which is updated at each loop iteration at line 17. Line 18 updates the lists of the split for each new match; these lists are then returned in line 19.

To better understand how SPLIT works, consider again the match tree in Fig. 2. After CHECK algorithm, we have $Matches' = \{M_1, M_2\}$ where M_1 and M_2 correspond to paths $[\langle 3,2\rangle, \langle 3,1\rangle, \langle 2,1\rangle, \langle 1,1\rangle]$ and $[\langle 3,2\rangle, \langle 2,2\rangle, \langle 2,1\rangle, \langle 1,1\rangle]$ respectively. Let us consider M_1 (see the red dashed arrows). Its first node $\langle \{c..f\}^{0,2}, \{f\}^{1,1}\rangle$ propagates upward the block $(\{c..f\} \cap \{f\})^{\max(0,1),\min(2,1)} = \{f\}^{1,1}$. Then we change direction. Node $\langle \{c..f\}^{0,2}, \{b..d\}^{0,21}\rangle$ propagates upward $(\{c..f\} \cap \{b..d\})^{\max(0-1,0),\min(2-1,21)} = \{c, d\}^{0,1}$. Node $\langle \{d\}^{5,5}, \{b..d\}^{0,26}\rangle$ propagates $(\{d\} \cap \{b..d\})^{\max(5,0-0),\min(5,26-1)} = \{d\}^{5,5}$ and finally the root propagates $(\{a..c\} \cap \{b..d\})^{\max(0,26-5-1),\min(30,26-5-0)} = \{b, c\}^{20,21}$. The corresponding splits are then $split_S^1 = \{1 : \{b, c\}^{20,21}, 2 : \{d\}^{5,5}, 3 : \{c, d\}^{0,1}\{f\}^{1,1}\}$ and $split_T^1 = \{1 : \{b..d\}^{26,26}, 2 : \{f\}^{1,1}\}$. We observe that $split_T^1[1] = T[1]$ instead of $T' = \{b, c\}^{20,21}\{d\}^{5,5}\{c, d\}^{0,1}$ since, as explained above, $T' \not\sqsubseteq T[1]$ (in particular, T' would compromise the soundness by allowing strings of length 25 and 27).

Similarly, we can construct $split_S^2 = \{1 : \{b, c\}^{21,21}, 2 : \{d\}^{5,5}\}$ and $split_T^2 = \{1 : \{b, c\}^{21,21}\{d\}^{5,5}, 2 : \{f\}^{1,1}\}$. Note that in the actual implementation the element $\{f\}^{1,1}$ coloured in violet in Fig. 2 does not need to be recomputed by SPLIT because it is already cached.

Merging At this stage, we have two lists of splits $splitS = [split_S^1, \dots, split_S^p]$ and $splitT = [split_T^1, \dots, split_T^p]$ that can be used to refine S and T respectively. The question now is: how to actually refine each $S[i]$ and $T[j]$, having different splitting $split_S^k[i]$ and $split_T^k[j]$ for $k = 1, \dots, p$? We have somehow to *merge* each split $split_S^1[i], \dots, split_S^p[i]$ into a minimal dashed string $\widetilde{S_i}$ that "contains" each split, i.e., such that $\widetilde{S_i} \sqsupseteq split_S^1[i], \dots, split_S^p[i]$ (analogously for each $\widetilde{T_j}$).

Unfortunately, we remark that $(\mathbb{DS}, \sqsubseteq)$ is not a lattice so there might not exist a least upper bound for $split_S^1[i], \dots, split_S^p[i]$ (see Proposition 1). Even here we have thus to settle for a relaxed "join" operation $\sqcup$ returning a dashed string $\widetilde{S_i} = split_S^1[i] \sqcup \dots \sqcup split_S^p[i]$ that over-approximates each split and it is a

good compromise between precision and efficiency (same thing for $\widetilde{T_j}$). If some $split_S^k[i]$ is not defined, we simply ignore it.

In the general case, given $S = S_1^{a_1,b_1} \cdots S_n^{a_n,b_n}$ and $T = T_1^{c_1,d_1} \cdots T_m^{c_m,d_m}$ we define $S \sqcup T = R^{l,u}$ where $R = \bigcup_{i=1}^{n} \bigcup_{j=1}^{m} (S_i \cup T_j)$, $l = \min(\Sigma_{i=1}^{n} a_i, \Sigma_{j=1}^{m} c_j)$, and $u = \max(\Sigma_{i=1}^{n} b_i, \Sigma_{j=1}^{m} d_j)$. However, we also deal with particular cases to improve the precision (e.g., when $S = T$).

In the example of Fig. 2, having $split_S^1 = \{1 : \{b, c\}^{20,21}, 2 : \{d\}^{5,5}, 3 : \{c, d\}^{0,1}\{f\}^{1,1}\}$ and $split_S^2 = \{1 : \{b, c\}^{21,21}, 2 : \{d\}^{5,5}\}$, we get $\widetilde{S_1} = \{b, c\}^{20,21}$, $\widetilde{S_2} = \{d\}^{5,5}$, and $\widetilde{S_3} = \{c, d\}^{0,1}\{f\}^{1,1}$. For T instead we simply get $\widetilde{T_1} = T_1$ and $\widetilde{T_2} = T_2$. Finally, we return $\widetilde{S} = \widetilde{S_1} \ldots \widetilde{S_n}$ and $\widetilde{T} = \widetilde{T_1} \ldots \widetilde{T_m}$.

Updating In the last stage, we update the original dashed strings S and T trying to refine their blocks thanks to the information given by $\widetilde{S}$ and $\widetilde{T}$. To do so, we use a simple block-wise approach that compares each S_i with $\widetilde{S_i}$ and, in case $\|\widetilde{S_i}\| < \|S_i\|$, updates S_i with $\widetilde{S_i}$. For avoiding overflows, instead of $\|S\|$ we consider its logarithm $\log \|S\| = \sum_{i=1}^{n} \log \|S_i^{a_i,b_i}\|$. In particular, if $x = |S| > 1$, we compute $\log \|S^{l,u}\|$ as $\log \dfrac{x^{u+1} - x^l}{x - 1} = \log \dfrac{x^l(x^{u-l+1} - 1)}{x - 1} = \log(x^l(x^{u-l+1} - 1)) - \log(x - 1) = l \cdot \log x + \log(x^{u-l+1} - 1) - \log(x - 1)$. In the same way we possibly update each T_j with $\|\widetilde{T_j}\|$.

Considering again the example in Fig. 2, from the original dashed strings $S = \{a..c\}^{0,30}\{d\}^{5,5}\{c..f\}^{0,2}$ we get $S' = \{b, c\}^{20,21}\{d\}^{5,5}\{c, d\}^{0,1}$, while T remains unchanged. However, we observe that while $\|S\|$ is in the order of 10^{15}, the size of S' is 9437184. Note the size difference which results if we equate $S'' = \{a..c\}^{0,30M}\{d\}^{5M,5M}\{c..f\}^{0,2M}$ and $T'' = \{b..d\}^{26M,26M}\{f\}^{M,M}$, where M is an arbitrarily big parameter. A nice property of EQUATE algorithm is that in this case the complexity is totally independent from M: both EQUATE(S, T) and EQUATE(S'', T'') are solved instantaneously.

Finally, note that we could run EQUATE on S and T with the blocks reversed to determine different information. We do not consider this in our implementation since we will focus on extracting information about the earliest blocks which will be the most helpful when aligned with the search we perform.

4 Constraint Solving

In this Section we give an overview of how we applied the notions introduced in Section 3 in order to solve a CSP with string variables and constraints.

Given a CSP $\langle \mathcal{V}, \mathcal{D}, \mathcal{C} \rangle$, the domain of each string variable $x \in \mathcal{V}$ is a dashed string $\mathcal{D}(x) \in \mathbb{DS}$. Each constraint $C \in \mathcal{C}$ on string variables $x_1, \ldots, x_k$ has an associated *propagator* that aims to remove the inconsistent values from domains $\mathcal{D}(x_1), \ldots, \mathcal{D}(x_k)$. Since propagation is incomplete, we have to define *search* strategies that split the domain of strings to cause more propagation.

4.1 Constraints

The key property of dashed strings that makes them useful is that we can concatenate dashed strings in a natural way: given $S = S_1^{a_1,b_1} \cdots S_n^{a_n,b_n}$ and $T = T_1^{c_1,d_1} \cdots T_m^{c_m,d_m}$ we get $S{\cdot}T = S_1^{a_1,b_1} \cdots S_n^{a_n,b_n} T_1^{c_1,d_1} \cdots T_m^{c_m,d_m}$ without any effort. Analogously, we can easily define the iterated concatenation $S^k = S{\cdot}S^{k-1}$, where $S^0 = \emptyset^{0,0}$, and the reverse $S^{-1} = S_n^{a_n,b_n} \cdots S^{1,1}$. Hence we can define many propagators by simply relying on the dashed string concatenation and the Equate algorithm described in Section 3. To lighten the load of propagation, we defined CheckEquate, a simplified version of Equate(S,T) that returns *true* if S and T have a match (and immediately returns), and *false* otherwise. CheckEquate neither stores nor computes the matches.

We consider the following constraints, and the corresponding propagators:[1]

- *equality* $x = y$. Implemented by Equate$(\mathcal{D}(x), \mathcal{D}(y))$;
- *disequality* $x \neq y$. If CheckEquate$(\mathcal{D}(x), \mathcal{D}(y)) =$ *false* the constraint is subsumed; otherwise we wait until both $\mathcal{D}(x)$ and $\mathcal{D}(y)$ are known;
- *half-reified* [13] equality $b \Rightarrow (x = y)$. If $b =$ *true*, the constraint is rewritten into $x = y$. If $b =$ *false*, the constraint is subsumed. Otherwise, if CheckEquate$(\mathcal{D}(x), \mathcal{D}(y)) =$ *false* then b is set to *false*. We treat $b \Rightarrow (x \neq y)$ similarly. Full reification $b \Leftrightarrow (x = y)$ is encoded as the conjunction $(b \Rightarrow x = y) \wedge (\neg b \Rightarrow x \neq y)$.
- *length* $|x| = n$. If $\mathcal{D}(x) = S_1^{l_1,u_1} \cdots S_k^{l_k,u_k}$, it is implemented analogously to $x_1 + \cdots + x_k = n$ where x_i is an integer variable with domain $[l_i, u_i]$.
- *domain* $x :: S$, where $S \in \mathbb{DS}$. Implemented by a version of Equate$(\mathcal{D}(x), S)$ that only updates $\mathcal{D}(x)$.
- *concatenation* $z = x \cdot y$. Implemented by Equate$(\mathcal{D}(z), \mathcal{D}(x) \cdot \mathcal{D}(y))$, taking care of properly projecting the narrowing of $\mathcal{D}(x) \cdot \mathcal{D}(y)$ on $\mathcal{D}(x)$ and $\mathcal{D}(y)$.
- *iterated concatenation* $y = x^n$. If $\mathcal{D}(x) = S_1^{l_1,u_1} \cdots S_k^{l_k,u_k}$, it is propagated by Equate$(\mathcal{D}(y), \mathcal{D}(x)^{\underline{n}} \cdot (\bigcup_{i=1}^{k} S_i)^{0,\overline{n}-\underline{n}})$.
- *reverse* $y = x^{-1}$. Implemented by Equate$(\mathcal{D}(y), \mathcal{D}(x)^{-1})$.
- *sub-string* $y = x[i..j]$. Rewritten in $l = |x| \wedge n = \max(1, i) \wedge m = \min(l, j) \wedge |y| = \max(0, m-n+1) \wedge x = y' \cdot y \cdot y'' \wedge y' :: \Sigma^{\underline{n}-1,\overline{n}-1} \wedge y'' :: \Sigma^{\max(0,\underline{l}-\overline{m}),\overline{l}-\underline{m}}$.

This set of constraints is not fully exhaustive. In particular, the lack of *regular* constraint limits its expressiveness since we can not fully encode the Kleene star S^* when S is a set of strings with length greater than one. However, thanks to reification and (iterated) concatenation we can often compensate this lack (and also define constraints that are not expressible with regular, i.e., see the SQL Injection problem firstly introduced in [1] and evaluated in Section 5).

Each propagator is scheduled by *propagator events* that occur if and when the domain of a variable in the constraint changes. We consider the following events: *fail* (a domain became empty), *none* (domains unchanged), *value* (a domain became a singleton), *cardinality* (the cardinality of some blocks changed),

[1] For conciseness, for integer variable x, we define $\underline{x} = \min(\mathcal{D}(x))$ and $\overline{x} = \max(\mathcal{D}(x))$.

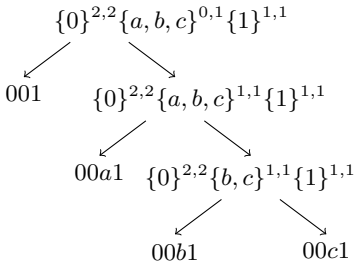

Fig. 3. Example of search tree.

character (the characters of some bases changed), *domain* (cardinality or characters changed). For example, the propagator for $|x| = n$ can only narrow the length of $\mathcal{D}(x)$ and not its characters, hence it does not need to wake on character changes.

4.2 Search

Searching in string problems is very important since there are typically a very large number of solutions for each string variable x. The search strategy we implemented first chooses the string variable x with smallest domain (minimizing $\log \|\mathcal{D}(x)\|$).

If the length of x is unknown it branches on the first unknown length block $S_i^{l_i,u_i}$ being equal to its minimal length or not (i.e., $S_i^{l_i,l_i}$ or $S_i^{l_i+1,u_i}$). This branching wakes up propagators dependent on the length of x.

Otherwise if the first non-zero length block $S_i^{l_i,l_i}$ is of length $l_i > 1$ it splits it into two fixed length blocks $S_i^{1,1}S_i^{l_i-1,l_i-1}$ (note this is not a branch point). If the first non-zero length block $S_i^{l_i,l_i}$ is of length 1 it branches on setting the block to its least value $a = \min(S_i)$ or not (i.e., $\{a\}$ or $S_i - \{a\}$). This branching wakes up propagators dependent of the contents of x.

Overall this search has the effect of enumerating the solutions of x in lexicographic order, as shown in Figure 3 where we show the search tree when $\mathcal{D}(x) = \{0\}^{2,2}\{a,b,c\}^{0,1}\{1\}^{1,1}$. However, the branching can be generalised by defining proper heuristic to choose how to split an unknown-length block, and how to pick a value from the base of a known-length block.

5 Evaluation

We implemented our approach as an extension of Gecode [18], a mature CP solver written in C++. The resulting solver, that we called G-Strings, is publicly available at `https://bitbucket.org/robama/g-strings`.

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

Table 1. Results in seconds. 'n/a' indicates an abnormal termination while 't/o' means timeout. Unsatisfiable problems are marked with *, best performance are in bold font.

	Chuffed			Gecode			iZplus			Z3str3			Gecode+𝕊			G-Strings		
ℓ	250	1000	10000	250	1000	10000	250	1000	10000	250	1000	10000	250	1000	10000	250	1000	10000
$a^n b^n$ *	0.1	1.3	483.83	1.81	129.32	t/o	0.74	16.45	t/o	t/o	t/o	t/o	0.31	29.46	t/o	**0.0**	**0.0**	**0.0**
ChunkSplit	1.62	t/o	26.33	0.39	12.93	61.91	1.81	11.56	116.61	0.6	0.6	0.6	1.28	182.24	t/o	**0.0**	**0.0**	**0.0**
Hamming *	0.32	1.69	61.27	0.16	0.77	19.58	0.24	1.89	49.95	1.22	1.2	1.2	**0.0**	0.12	129.8	**0.0**	**0.0**	**0.0**
Levenshtein	0.18	0.89	63.67	0.08	0.36	18.05	2.32	2.64	306.39	0.01	0.01	0.01	**0.0**	**0.0**	**0.0**	**0.0**	**0.0**	**0.0**
StringRep.	1.46	43.75	n/a	0.56	19.28	n/a	0.7	8.54	673.18	2.58	2.59	2.62	0.06	2.25	t/o	**0.0**	**0.0**	**0.0**
SQLInj.	10.78	t/o	n/a	0.81	375.17	t/o	130.68	613.16	n/a	t/o	t/o	t/o	**0.01**	**0.2**	**299.99**	0.09	72.58	t/o

management becomes critical. We underline that G-Strings is still a prototype, and mainly relies on the Gecode built-in data structures. In a nutshell, a dashed string is currently implemented as a `DynamicArray` of blocks, where the base of each block is encoded by a `BndSet`, which represents finite set of integers as unions of disjoint ranges (see [18] for more details about this data structures). As a future work we plan to improve this implementation.

We compared G-Strings against the string CP solver Gecode+𝕊 [29, 31], the string SMT solver Z3str3 [34][2] and three state-of-the-art constraint solvers, namely: the aforementioned Gecode [18]; Chuffed [9], a CP solver with lazy clause generation [27]; and iZplus [15], a CP solver that also exploits local search. For these three solvers we used the MiniZinc translation to integers [1] that statically maps string variables into arrays of integer variables. We did not compare against automata-based approaches like [21, 24, 32, 33] since their limited effectiveness in our context (as an example, every single block $S^{l,u}$ has to be encoded by an automaton of exactly $u + 1$ states).

As already noted in [1,20,29–31] there is unfortunately a lack of standardised and challenging string benchmarks. We decided to use the same string problems used in the evaluation of [1], namely: $a^n b^n$, ChunkSplit, HammingDistance, Levenshtein, StringReplace, SQLInjection. The only differences are: *(i)* the "HammingDistance" problem has been simplified since G-Strings does not yet support the regular constraint (for the other problems we have overcome this lack with (iterated) concatenation and reified equality); *(ii)* the "Palindrome" problem is omitted since neither G-Strings nor Gecode+𝕊 supports the new global cardinality constraint introduced in [1].

All these problems have no parameters, except for the maximum string length ℓ that we varied in $\{250, 1000, 10000\}$. We ran the experiments on Ubuntu 15.10 machines with 16 GB of RAM and 2.60 GHz Intel® i7 CPU by setting a solving timeout of $T = 1200$ seconds.

Comparative solving times are shown in Table 1. We ignore model construction time, which for the first three solvers using MiniZinc can be quite expensive (for SQL and $\ell = 10000$ this is almost 20 minutes). The results show that the G-Strings solver is (almost) independent of the maximum string length, and in particular it provides an instantaneous answer in all the Norn benchmarks.

[2] We used the last stable release: `https://sites.google.com/site/z3strsolver/`.

The performance of GECODE+𝕊 and the other CP solvers clearly degrade when increasing ℓ. Although being independent from ℓ, also Z3STR3 performs worse than G-STRINGS.

Conversely, the SQL benchmark illustrates a weakness of the current implementation. This problem involves a long fixed string of length ℓ, and our solver has worse performance than GECODE+𝕊. In particular, G-STRINGS runs out of time when $\ell = 10000$. This points out that we need to specialise the EQUATE algorithm for parts of strings representable as fixed strings, and also switch to asymptotically faster propagation algorithms when the number of blocks becomes large.

6 Conclusions

In this work we introduced the dashed string representation to enable possibly very long strings to be represented succinctly, trying to decouple the complexity of constraint solving from the maximum length a string *may* have. Propagation of dashed strings is very efficient when the number of its blocks is small. Moreover, while dealing with large alphabets might be a problem for some approaches [31], this representation is weakly coupled to the size of the alphabet we are using.

Clearly dashed strings are not a universal panacea, since equating long dashed string representations can be too expensive. In other terms, this approach might fail when a string *must* be very long. Hence we need to develop weaker propagation algorithms to gracefully handle this case.

Using multiple representations for a string variable may be highly advantageous, where we choose the propagator for each string constraint which is most efficient to propagate. String abstract domains often combine different representations in this way (see, e.g., [3,10]). Although this may clearly reduce the propagation of information, it can avoid worst case behaviour. This hybrid approach can be implemented "internally", i.e., by building a *channeling* propagator between the representations, or "externally" via a *portfolio* approach [2] combining different solving strategies.

The introduction of dashed strings immediately opens several research branches. One of these concerns the definition of new propagators. We have already devised algorithms for propagating lexicographic comparisons, global cardinality, and regular constraints, although they are not implemented yet in G-STRINGS. Furthermore, the potentially huge search space suggests the exploration of different search approaches such as, e.g., Local Search [6, 7]. Another interesting directions concerns the definition of *trailing* string solvers, i.e., solvers that store the domain changes in a stack instead of copying the entire domain during the search.

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

References

1. R. Amadini, P. Flener, J. Pearson, J. D. Scott, P. J. Stuckey, and G. Tack. Minizinc with strings. *To appear in LOPSTR post-proceedings*, 2016. https://arxiv.org/abs/1608.03650.
2. R. Amadini, M. Gabbrielli, and J. Mauro. A multicore tool for constraint solving. In *Proceedings of the International Joint Conference on Artificial Intelligence*, pages 232–238. AAAI Press, 2015.
3. R. Amadini, A. Jordan, G. Gange, F. Gauthier, P. Schachte, H. Søndergaard, P. J. Stuckey, and C. Zhang. Combining string abstract domains for javascript analysis: An evaluation. In *Tools and Algorithms for the Construction and Analysis of Systems - 23rd International Conference, TACAS 2017, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2017, Uppsala, Sweden, April 22-29, 2017, Proceedings, Part I*, pages 41–57, 2017.
4. P. Barahona and L. Krippahl. Constraint programming in structural bioinformatics. *Constraints*, 13(1-2):3–20, 2008.
5. P. Bisht, T. L. Hinrichs, N. Skrupsky, and V. N. Venkatakrishnan. WAPTEC: Whitebox analysis of web applications for parameter tampering exploit construction. In *Proceedings of ACM Conference on Computer and Communications Security*, pages 575–586. ACM, 2011.
6. G. Björdal. String variables for constraint-based local search. Master's thesis, Department of Information Technology, Uppsala University, Sweden, August 2016. Available at `http://urn.kb.se/resolve?urn=urn:nbn:se:uu:diva-301501`.
7. G. Björdal, J.-N. Monette, P. Flener, and J. Pearson. A constraint-based local search backend for MiniZinc. *Constraints*, 20(3):325–345, July 2015.
8. N. Bjørner, N. Tillmann, and 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*, volume 5505 of *LNCS*, pages 307–321. Springer, 2009.
9. G. Chu. *Improving Combinatorial Optimization*. PhD thesis, Department of Computing and Information Systems, University of Melbourne, Australia, 2011.
10. G. Costantini, P. Ferrara, and A. Cortesi. A suite of abstract domains for static analysis of string values. *Software: Practice and Experience*, 45(2):245–287, 2015.
11. P. Cousot and 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*, pages 238–252. ACM, 1977.
12. M. Emmi, R. Majumdar, and K. Sen. Dynamic test input generation for database applications. In *Proceedings of the ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*, pages 151–162. ACM, 2007.
13. T. Feydy, Z. Somogyi, and P. Stuckey. Half-reification and flattening. In J. Lee, editor, *Proceedings of the 17th International Conference on Principles and Practice of Constraint Programming*, volume 6876 of *LNCS*, pages 286–301. Springer, 2011.
14. X. Fu, M. C. Powell, M. Bantegui, and C. Li. Simple linear string constraints. *Formal Aspects of Computing*, 25(6):847–891, 2013.
15. T. Fujiwara. iZplus description. `http://www.minizinc.org/challenge2016/description_izplus.txt`, 2016.
16. V. Ganesh, M. Minnes, A. Solar-Lezama, and M. C. Rinard. Word equations with length constraints: What's decidable? In *HVC*, volume 7857 of *LNCS*, pages 209–226. Springer, 2013.

17. G. Gange, J. A. Navas, P. J. Stuckey, H. Søndergaard, and 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*, volume 7795 of *LNCS*, pages 277–291. Springer, 2013.
18. Gecode Team. Gecode: Generic constraint development environment, 2016. Available at `http://www.gecode.org`.
19. K. Golden and W. Pang. Constraint reasoning over strings. In *Proceedings of the Ninth International Conference on Principles and Practice of Constraint Programming*, pages 377–391, 2003.
20. J. He, P. Flener, and J. Pearson. Solving string constraints: The case for constraint programming. In *Proceedings of the 19th International Conference on Principles and Practice of Constraint Programming*, volume 8124 of *LNCS*, pages 381–397. Springer, 2013.
21. P. Hooimeijer and W. Weimer. StrSolve: Solving string constraints lazily. *Automated Software Engineering*, 19(4):531–559, 2012.
22. A. Kiezun, V. Ganesh, S. Artzi, P. J. Guo, P. Hooimeijer, and M. D. Ernst. HAMPI: A solver for word equations over strings, regular expressions, and context-free grammars. *ACM Transactions on Software Engineering and Methodology*, 21(4):article 25, 2012.
23. S. Kim, W. Chin, J. Park, J. Kim, and S. Ryu. Inferring grammatical summaries of string values. In *12th Asian Symposium on Programming Languages and Systems*, volume 8858 of *LNCS*, pages 372–391. Springer, 2014.
24. G. Li and I. Ghosh. PASS: String solving with parameterized array and interval automaton. In *Proceeedings of the Haifa Verification Conference*, volume 8244 of *LNCS*, pages 15–31. Springer, 2013.
25. M. Madsen and E. Andreasen. String analysis for dynamic field access. In *23rd International Conference on Compiler Construction*, volume 8409 of *LNCS*, pages 197–217. Springer, 2014.
26. N. Nethercote, P. J. Stuckey, R. Becket, S. Brand, G. J. Duck, and G. Tack. MiniZinc: Towards a standard CP modelling language. In *Proceedings of the 13th International Conference on Principles and Practice of Constraint Programming*, volume 4741 of *LNCS*, pages 529–543. Springer, 2007.
27. O. Ohrimenko, P. J. Stuckey, and M. Codish. Propagation via lazy clause generation. *Constraints*, 14(3):357–391, 2009.
28. P. Saxena, D. Akhawe, S. Hanna, F. Mao, S. McCamant, and D. Song. A symbolic execution framework for JavaScript. In *S&P*, pages 513–528. IEEE Computer Society, 2010.
29. J. D. Scott. *Other Things Besides Number: Abstraction, Constraint Propagation, and String Variable Types*. PhD thesis, Department of Information Technology, Uppsala University, Sweden, 2016. Available at `http://urn.kb.se/resolve?urn=urn:nbn:se:uu:diva-273311`.
30. J. D. Scott, P. Flener, and J. Pearson. Constraint solving on bounded string variables. In *Twelth International Conference on Integration of Artificial Intelligence and Operations Research techniques in Constraint Programming*, volume 9075 of *LNCS*, pages 375–392. Springer, 2015.
31. J. D. Scott, P. Flener, J. Pearson, and C. Schulte. Design and implementation of bounded-length sequence variables. In *Fourteenth International Conference on Integration of Artificial Intelligence and Operations Research techniques in Constraint Programming*, LNCS. Springer, 2017.

32. T. Tateishi, M. Pistoia, and O. Tripp. Path- and index-sensitive string analysis based on monadic second-order logic. *ACM Transactions on Software Engineering Methodology*, 22(4):33, 2013.
33. F. Yu, M. Alkhalaf, and T. Bultan. Stranger: An automata-based string analysis tool for PHP. In *TACAS*, volume 6015 of *LNCS*, pages 154–157. Springer, 2010.
34. Y. Zheng, V. Ganesh, S. Subramanian, O. Tripp, J. Dolby, and X. Zhang. Effective search-space pruning for solvers of string equations, regular expressions and length constraints. In *CAV*, volume 9206 of *LNCS*, pages 235–254. Springer, 2015.