The Thirty-Second AAAI Conference
on Artificial Intelligence (AAAI-18)

Sweep-Based Propagation for String Constraint Solving

Roberto Amadini, Graeme Gange, Peter J. Stuckey
Department of Computing and Information Systems
The University of Melbourne, Australia
{roberto.amadini,gkgange,pstuckey}@unimelb.edu.au

Abstract

Solving constraints over strings is an emerging important field. Recently, a Constraint Programming approach based on *dashed strings* has been proposed to enable a compact domain representation for potentially large bounded-length string variables. In this paper, we present a more efficient algorithm for propagating equality (and related constraints) over dashed strings. We call this propagation *sweep-based*. Experimental evidences show that sweep-based propagation is able to significantly outperform state-of-the-art approaches for string constraint solving.

Introduction

Constraint solving over strings is an important field, given the ubiquity of strings in different domains such as, e.g., software verification and testing (Emmi, Majumdar, and Sen 2007; Bjørner, Tillmann, and Voronkov 2009), model checking (Gange et al. 2013), and web security (Bisht et al. 2011; Thomé et al. 2017).

Various approaches of disparate nature have been proposed, based for instance on bit-vectors (Kiezun et al. 2012), automata (Hooimeijer and Weimer 2012; Li and Ghosh 2013; Tateishi, Pistoia, and Tripp 2013), SMT extensions (Berzish, Zheng, and Ganesh 2017; Liang et al. 2014; Abdulla et al. 2015; Trinh, Chu, and Jaffar 2014; Yu, Alkhalaf, and Bultan 2010; Saxena et al. 2010), and constraint programming (Scott et al. 2017; Amadini et al. 2017).

Typically, we aim to solve problems with *bounded-length* strings, i.e., strings having a maximum length. These problems are mainly addressed with some variant of *unfolding*: a string s of length (up to) k is translated into a sequence of single-character variables $s_1, \ldots, s_k$, and string constraints are imposed in terms of these variables. However, methods which unfold eagerly face scalability problems when the upper bounds on string length becomes large, and committing to absolute character positions makes it difficult to reason precisely when combining strings of non-fixed length.

Dashed strings, an alternative representation for modelling the domain of string variables through sub-sequences of uncertain length, were described in (Amadini et al. 2017).

Copyright © 2018, Association for the Advancement of Artificial Intelligence (www.aaai.org). All rights reserved.

In a nutshell, given an alphabet Σ, a dashed string is a concatenation $S_1^{l_1,u_1} S_2^{l_2,u_2} \cdots S_k^{l_k,u_k}$ of k *blocks*, where 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 $S_i \subseteq \Sigma$. Unfortunately, the dynamic programming algorithm used in (Amadini et al. 2017) for propagating equality, here referred as COVER, suffers its own scalability troubles: its worst-case behaviour is cubic in the number of blocks, so performance degrades rapidly when dealing with long fixed strings.

In this paper, we introduce the SWEEP algorithm for propagating equality between dashed strings. In essence, the algorithm performs linear-time sweeps across a dashed string, collecting the earliest and latest positions a given block could be matched. The SWEEP algorithm provides weaker propagation than COVER, but has much lower worst-case complexity: as we shall see, this can dramatically improve the resolution when the number of blocks becomes large.

We implemented the SWEEP algorithm by extending G-STRINGS, a string solver introduced in (Amadini et al. 2017) which represents string variables with dashed strings, and we compared its performance against the COVER algorithm (already implemented in G-STRINGS) and other state-of-the-art approaches such as: the CP-based string solver GECODE+𝕊 (Scott et al. 2017); the CP solvers CHUFFED, GECODE, IZPLUS (by mapping strings into array of integers); the SMT-based string solvers Z3STR3 (Berzish, Zheng, and Ganesh 2017) and CVC4 (Liang et al. 2014).

To make the empirical evaluation more meaningful, we extended the benchmarks used in (Amadini et al. 2017) by generating new instances of the SQL problem introduced in (Amadini et al. 2016). Empirical results indicate that sweep-based propagation allows us to considerably outperform all the aforementioned approaches.

Dashed Strings

In this Section we give some preliminary notions about dashed strings, including the COVER algorithm introduced in (Amadini et al. 2017) for equating dashed strings,[1] and the CP solver G-STRINGS, which implements COVER.

[1] In (Amadini et al. 2017) the COVER algorithm is actually called EQUATE. Here we call it COVER to differentiate it from the SWEEP algorithm, since both COVER and SWEEP are used for equating dashed strings.

Figure 1: Representation of $\{\texttt{B,b}\}^{1,1}\{\texttt{o}\}^{2,4}\{\texttt{m}\}^{1,1}\{\texttt{!}\}^{0,3}$.

Let us fix an alphabet Σ, a maximum string length $\ell \in \mathbb{N}$, and the universe $\mathbb{S} = \bigcup_{i=0}^{\ell} \Sigma^i$. 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 $i = 1, \dots, k$, and $\Sigma_{i=1}^{k} l_i \leq \ell$.

For each block $S_i^{l_i,u_i}$, we call S_i the *base* and (l_i, u_i) the *cardinality*. For brevity we will sometimes write a block with singleton base $\{a\}^{l,u}$ as $a^{l,u}$. The i-th block of a dashed string X is denoted by $X[i]$, and $|X|$ is the number of blocks of X. We do not distinguish blocks from dashed strings of unary length. For each pair of the form $C = (l, u)$, we define $\mathsf{lb}(C) = l$ and $\mathsf{ub}(C) = u$.

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 X is *known* if $|\gamma(X)| = 1$, i.e., it represents a single string. A dashed-string X is *nullable* if $\epsilon \in \gamma(X)$; or equivalently, each of its blocks has minimum cardinality 0. We assume *normalised* dashed strings, where $\emptyset^{0,0}$ can occur at most once and adjacent blocks always have distinct bases.

The *size* $\|S^{l,u}\|$ of a block is the number of concrete strings it denotes, i.e., $\|S^{l,u}\| = |\gamma(S^{l,u})|$. The size of dashed string $X = S_1^{l_1,u_1} S_2^{l_2,u_2} \cdots S_k^{l_k,u_k}$ is instead an over-estimate of $|\gamma(X)|$, given by $\|X\| = \Pi_{i=1}^{k} \|S_i^{l_i,u_i}\|$.

Given two dashed strings X and Y we define the relation $X \sqsubseteq Y \iff \gamma(X) \subseteq \gamma(Y)$. Intuitively, $\sqsubseteq$ models the relation "is more precise than" between dashed strings. Unfortunately, the set of dashed strings does not form a lattice according to $\sqsubseteq$. This implies that some workarounds have to be used to determine a reasonable lower/upper bound of two dashed strings according to $\sqsubseteq$.

For example, it is sometimes useful to over-approximate a dashed string $X = S_1^{l_1,u_1} S_2^{l_2,u_2} \cdots S_k^{l_k,u_k}$ into a single block. We define an upper closure operator CRUSH such that $\text{CRUSH}(X) = S^{l,u}$ where $S = \bigcup_{i=1}^{k} S_i$, $l = \sum_{i=1}^{k} l_i$, and $u = \sum_{i=1}^{k} u_i$. Moreover, we also consider a restricted version of CRUSH only crushing blocks in X that are compatible with a set of characters T, that is, $\text{CRUSH}_T(X) = \text{CRUSH}([(S_i \cap T)^{l_i,u_i} \mid i \in \{1, \dots, k\}, S_i \cap T \neq \emptyset])$.

Intuitively, we can imagine each block $S_i^{l_i,u_i}$ of $X = S_1^{l_1,u_1} S_2^{l_2,u_2} \cdots S_k^{l_k,u_k}$ 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 Fig. 1, illustrating dashed string $X = \{\texttt{B,b}\}^{1,1}\{\texttt{o}\}^{2,4}\{\texttt{m}\}^{1,1}\{\texttt{!}\}^{0,3}$. Each string of $\gamma(X)$ starts with B or b, followed by 2 to 4 os, one m, then 0 to 3 !s.

COVER Algorithm

Equating dashed strings X and Y requires determining, if possible, two dashed strings X' and Y' such that: *(i)* $X' \sqsubseteq X, Y' \sqsubseteq Y$; *(ii)* $\gamma(X') \cap \gamma(Y') = \gamma(X) \cap \gamma(Y)$.

Informally, we can see this problem 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 $\gamma(X) \cap \gamma(Y)$ (note that there may not exist a greatest lower bound for X, Y according to $\sqsubseteq$).

In (Amadini et al. 2017) the authors propose a multiphase equation algorithm for equating dashed strings. We now call it COVER because it is based on the notion of *coverable* blocks, where block $S^{l,u}$ 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 $l = 0 \vee (l \leq u' \wedge S \cap T \neq \emptyset)$). If block B is coverable by B', or B' is coverable by B, then B and B' are *compatible*; otherwise, they are incompatible.

The COVER algorithm is difficult to summarize: we refer the reader to (Amadini et al. 2017) for a full explanation. In a nutshell, COVER first checks if dashed strings X and Y are equatable. This is performed by a top-down dynamic programming approach, recursively matching pairs of compatible blocks. This phase returns a directed acyclic graph encoding a set of *matchings*, i.e., computations that successfully consume all the blocks of X and Y.

Then, all the matchings are processed and merged together (often by means of the CRUSH operation) to possibly update X and Y with refined dashed strings X' and Y'.

As an example, consider $X = \{a..c\}^{0,30}\{d\}^{5,5}\{c..f\}^{0,2}$ and $Y = \{b..d\}^{26,26}\{f\}^{1,1}$. The COVER algorithm refines X in $X' = \{b, c\}^{20,21}\{d\}^{5,5}\{c, d\}^{0,1}\{f\}^{1,1}$, while $Y' = Y$. Note that COVER does not aim to find an optimal refinement, since this requirement is too expensive to meet.

The main argument against COVER is its worst case complexity $O(nm \max(n, m))$, where $n = |X|, m = |Y|$. Indeed, despite having good results when the number of blocks is small (regardless of the the maximum length a string may have) COVER can be expensive when a string *must* be very long (this naturally implies a large number of blocks).

G-STRINGS Solver

The COVER algorithm was implemented in G-STRINGS, an extension of GECODE solver (Gecode 2017). G-STRINGS is a *copying solver*, i.e., during the search the domains are copied and possibly restored. In (Amadini et al. 2017) the authors demonstrate that, despite being a prototype, G-STRINGS already shows promising performance.

G-STRINGS implements the domain of every string variable x with a dashed string $\mathcal{D}(x)$, and defines a *propagator* for each string constraint. Propagators take advantage of COVER for refining the representations of the involved variables. For example, string equality $x = y$ is simply propagated by equating $\mathcal{D}(x)$ and $\mathcal{D}(y)$ with COVER.

Dashed strings naturally support concatenation: e.g., the propagator for $z = x \cdot y$ is implemented by equating $\mathcal{D}(z)$ and $\mathcal{D}(x) \cdot \mathcal{D}(y)$, where $\mathcal{D}(x) \cdot \mathcal{D}(y)$ is the concatenation of the blocks of $\mathcal{D}(x)$ and $\mathcal{D}(y)$, taking care of properly projecting the narrowing of $\mathcal{D}(x) \cdot \mathcal{D}(y)$ on $\mathcal{D}(x)$ and $\mathcal{D}(y)$.

G-STRINGS implements the following constraints: string (dis-)equality, (half-)reified equality, (iterated) concatenation, string domain, length, reverse, substring selection.

Since propagation is in general not complete, G-STRINGS also defines a search strategy that first chooses the string variable x with smallest domain size $\|\mathcal{D}(x)\|$ and then, 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 $S_i^{l_{i+1},u_i}$). Otherwise, if the first non-nullable block $S_i^{l_i,l_i}$ is of length $l_i > 1$ it splits it into two fixed length blocks $S_i^{1,1} S_i^{l_i-1,l_i-1}$. If the first non-nullable 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 $S_i - \{a\}$).

Sweep-based Propagation

As noted in (Amadini et al. 2017), the dynamic programming method for equality propagation performs poorly when long fixed strings are interleaved with unknown regions. In this section we present SWEEP: a more robust, though weaker, approach for enforcing equality.

The core idea of SWEEP is similar to timetable reasoning for CUMULATIVE (Aggoun and Beldiceanu 1993): to equate X and Y, for each block $X[i]$, we wish to find the earliest and latest positions in Y where $X[i]$ could be matched. Once these positions are computed, they are used to refine the block: roughly, $X[i]$ may only contain a content between its earliest start and latest end, and any content between the *latest* start and *earliest* end must be included in $X[i]$. This process is repeated symmetrically to refine each block $Y[j]$.

Given a dashed string X, we shall refer to *positions* (i, o), where i refers to block $X[i]$ and o is its *offset*, indicating how many characters of $X[i]$ have already been consumed. Positive offsets denote positions relative to the beginning of $X[i]$, and negative offsets are relative to the end of $X[i]$.

From an initial position p_i in Y, we try to match $X[i]$ against the blocks of Y immediately following p_i. If we succeed, p_i is the *earliest start time* of $X[i]$ denoted $est(X[i])$, and the *end* of this match identifies the *earliest end time* of $X[i]$, denoted $eet(X[i])$. However, if while doing so we reach a block $Y[j]$ incompatible with $X[i]$, then $X[i]$ must necessarily be matched somewhere *after* $Y[j]$.

SWEEP uses two (pairs of) complementary operations, PUSH and STRETCH. PUSH$^+(B, Y, (i, o))$ attempts to find the earliest possible match of B after o characters into $Y[i]$ (the ith block of Y), while STRETCH$^+(B, Y, (i, o))$ finds the *latest* position B could finish, assuming B begins at most o characters into $Y[i]$. There are analogous versions PUSH$^-$ and STRETCH$^-$ which work backwards across the blocks. They simply invert the blocks of Y and then convert the returned results back to the original offset descriptions.

The algorithm for PUSH$^+$ is given in Figure 2, and explained in Example 1 with a running example.

Example 1 *Consider finding the earliest matching of block $B = \{a, b\}^{3,4}$ into the dashed string $Y = Y_1 Y_2 Y_3 Y_4 Y_5 = \{a\}^{2,3}\{c\}^{1,2}\{b\}^{1,1}\{c\}^{0,2}\{a\}^{3,4}$ given an initial offset of 1 into Y_1, illustrated in Figure 3. We begin trying to fit as much as possible of B into the remainder of $\{a\}^{2,3}$. However the*

function PUSH$^+(S^{l,u}, Y, (i, o))$
    $(i_s, o_s) \leftarrow (i, o)$
    $k \leftarrow l$
    while $k > 0 \wedge i \leq |Y|$ **do**
        $T^{l',u'} \leftarrow Y[i]$
        if $S \cap T = \emptyset$ **then** ▷ Incompatible blocks
            $(i, o) \leftarrow (i + 1, 0)$
            if $l' > o$ **then** ▷ If not nullable,
                $(i_s, o_s) \leftarrow (i, o)$ ▷ restart from here
                $k \leftarrow l$
            end if
        else if $k \leq (u' - o)$ **then** ▷ Remainder fits in $Y[i]$
            return $(i_s, o_s), (i, o + k)$
        else ▷ Fill $Y[i]$ and continue
            $k \leftarrow k - (u' - o)$
            $(i, o) \leftarrow (i + 1, 0)$
        end if
    end while
    return $(i_s, o_s), (i, o)$
end function

Figure 2: PUSH$^+$ searches for the earliest feasible location to match $S^{l,u}$ in Y, assuming it starts no earlier than (i, o). The pair (i_s, o_s) tracks the candidate start position, while (i, o) is the corresponding end.

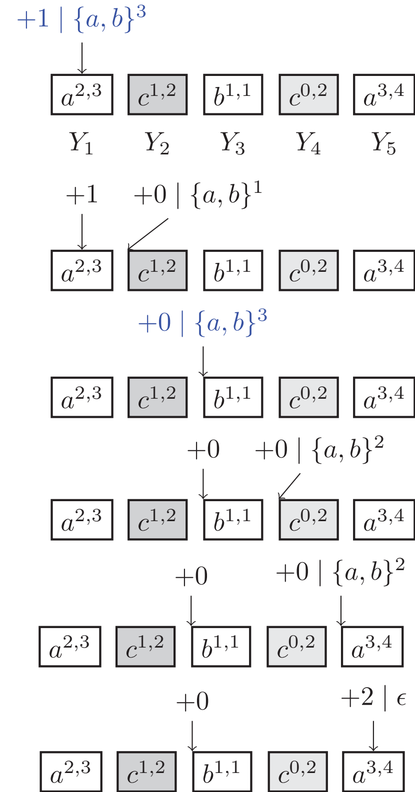

Figure 3: Applying PUSH$^+$ to find the earliest match for $\{a, b\}^{3,4}$, given initial offset of 1 into Y_1. End-point markers show current offset, and characters yet to be consumed. Incompatible blocks are in grey colour.

```
function STRETCH+(S^{l,u}, Y, (i,o))
    k ← u
    while i ≤ |Y| do
        T^{l',u'} ← Y[i]
        if (l' ≤ o) then                    ▷ Nothing to consume
            (i,o) ← (i+1, 0)
        else if S ∩ T = ∅ then              ▷ Incompatible blocks
            return (i,o)
        else if k < l' − o then             ▷ No more characters
            return (i, o+k)
        else                                ▷ l' − o characters consumed
            k ← k − (l' − o)
            (i,o) ← (i+1, 0)
        end if
    end while
    return (i,o)
end function
```

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

current block only has 2 *capacity remaining, which leaves* 1 *character to be matched in following blocks. However* B *is incompatible with* Y_2*. Since* B *cannot be placed* before Y_2, *it must be placed after; so we restart at the beginning of* Y_3.

After consuming Y_3*, we reach* Y_4 *– another incompatible block, but one with a lower bound of* 0*. In this case,* B *may still cross* Y_4 *(by setting* Y_4 *to null), so we skip it and continue matching* Y_5*. Block* Y_5 *consumes the remaining characters, and we terminate.*

From this, we conclude the earliest start for B *is* $(3, +0)$*, and the earliest start of the following block is at least* $(5, +2)$*, corresponding to the earliest end of* B*. The earliest start of* B *also gives us the earliest end of the predecessor of* B*: namely* $(3, -1)$*, or equivalently* $(2, -0)$.

As the matching may not contain gaps, the earliest start of B also determines the earliest end of its predecessor in X. So, for successive blocks in X, we may run PUSH$^+$ from the earliest end of its predecessor. However, we can do better by observing that a start position is infeasible if its suffix *cannot reach the end* of Y. Thus, we use the STRETCH$^+$ procedure to identify the earliest possible matching for the suffix.

STRETCH$^+$ is in some sense a dual of PUSH$^+$: where PUSH$^+$ attempts to squeeze as much of B into each block as possible, STRETCH$^+$ consumes only the minimum amount before moving onto the next block of Y. STRETCH$^+$ computes a bound on the *latest end time* of block B, denoted $let(B)$, from which we may also infer the latest start time of its successor. Analogously, we also define the *latest start time* of block B as $lst(B)$.

Pseudo-code for STRETCH$^+$ is given in Figure 4, and explained in Example 2 with a running example.

Example 2 *Recall the dashed string* Y *from Example 1. To find initial bounds on the latest ends of blocks in* X*, we run*

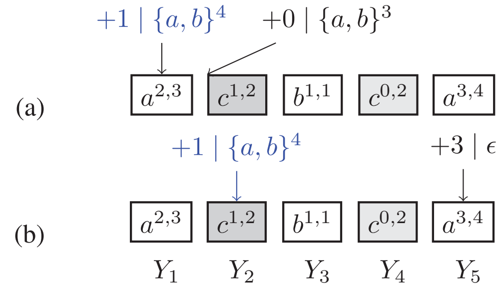

Figure 5: Applying STRETCH$^+$ on $\{a, b\}^{3,4}$ from two possible start positions in Y. End-point markers show current offset, and characters still available..

STRETCH$^+$ *progressively from the beginning of* Y.

Consider the case (a) *shown in Figure 5, where the predecessors of the current block* $B = \{a, b\}^{3,4}$ *can only reach position* $(1, +1)$*. 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 with offset* $+1$ *into* Y_1*, its effective lower bound is* $2-1 = 1$*, so we reach the beginning of* Y_2 *with* 3 *characters remaining. However,* Y_2 *is incompatible with* $\{a, b\}$*, and has a non-zero lower bound. 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* $(2, +0)$.

For case (b), we start inside *an incompatible block. However our starting offset is already at the lower bound; thus its effective lower bound is* 0*, and we may continue unopposed. We continue as before, reaching* Y_4 *with 3 characters available.* Y_4 *is again incompatible with* $\{a, b\}$*, but it is nullable and it can be skipped. Our remaining budget is consumed in* Y_5*, yielding a final position of* $(5, +3)$.

Note that while $(5, +3)$ *remains 'inside'* Y*, it nevertheless denotes a feasible end position, as the remainder of the dashed string is potentially null.*

Running STRETCH$^-$ from the *end* of Y allows us to compute initial (lower) bounds for the earliest starts of X-blocks. If the corresponding earliest start of the first block $X[1]$ is strictly later than first position $(1, +0)$, then we have detected infeasibility: X and Y can not be equated.

The overall algorithm for computing the earliest starts for X-blocks in Y is given in Figure 6. PROP-EST runs in $O(m + n)$ time (assuming set operations are constant time). The algorithm for computing the latest ends is analogous, swapping the sign (i.e., direction) of STRETCH and PUSH.

Filtering from matches

Given the earliest and latest matchings for a block $B = S^{l,u}$, we then attempt to tighten the base S and cardinality (l, u). The range between $est(B)$ and $let(B)$ is the *feasible* region – B must be fully contained in this region.

Similarly, the content between $lst(B)$ and $eet(B)$ forms a sub-region of the feasible region that we call the *mandatory* region.

```
function PROP-EST([X_1, ..., X_n], Y = [Y_1, ..., Y_m])
    end ← (m, +ub(Y_m))
    for i ∈ {n, n−1, ..., 1} do              ▷ Scanning backwards
        end ← STRETCH⁻(X_i, Y, end)
        est(X_i) ← end
    end for
    if est(X_1) > (1, +0) then
        return FAIL                           ▷ Suffix cannot reach
    end if
    end ← (1, 0)
    for i ∈ {1, ..., n} do                    ▷ Scanning forwards
        start ← MAX(end, est(X_i))
        start, end ← PUSH⁺(X_i, Y, start)
        if start > est(X_i) then
            est(X_i) ← start
        end if
    end for
    if est(X_n) > (m, +ub(Y_m)) then
        return FAIL                           ▷ Prefix cannot fit
    end if
    return est                 ▷ est is the array of earliest starts
end function
```

Figure 6: Algorithm for computing the earliest start position in Y of each X-block. We run STRETCH backwards (from the last block) to compute initial starting positions, then run PUSH forward to eliminate infeasible positions.

From the feasible region, we can tighten upper bounds on cardinality and the base characters. From the mandatory, we can update lower-bounds on block cardinality. We cannot generally refine feasible values using the mandatory region, as a base expresses only *possible*, rather than *necessary*, characters.

Consider ranges in Figure 7 showing the earliest and latest matchings of a block $S^{l,u}$ into a dashed string Y. The mandatory region starts one character into Y_2, and reaches the end of Y_3: the minimum cardinality l is at least $l' = \max(0, \mathsf{lb}(C_2) - 1) + \mathsf{lb}(C_3)$. We can use similar reasoning over the feasible region to prune the upper bound of cardinality, but we need to consider only those blocks which are

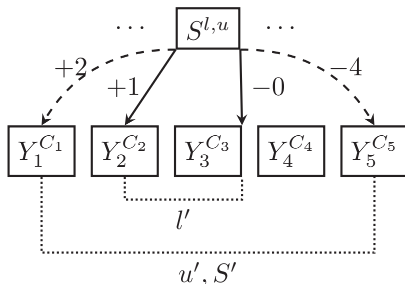

Figure 7: Filtering a block using matching bounds. Feasible bounds (*est* and *let*) are indicated with dashed lines, labelled with the offset into the specified block. Mandatory bounds (*lst* and *eet*) are indicated by solid lines.

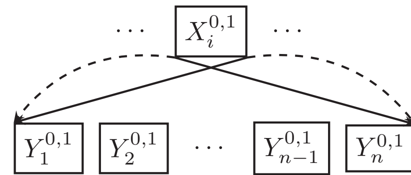

Figure 8: Matching two long sequences of possibly null blocks. As all X- and Y-blocks are nullable, the *est* and *let* of the X-blocks are never forced to shift. As such, processing the feasible region of X_i requires scanning all Y-blocks.

compatible with $S^{l,u}$. Similarly, we may intersect S with the union S' of all the bases of the blocks Y_j in the feasible region. But where the dashed strings contain many nullable blocks, the feasible region of each X_i may cover all of Y (see Fig. 8). In such cases, performing a complete sweep of the feasible region for each block is too expensive. Instead, we maintain a *cache* of previously processed feasible regions; before scanning each region, we first check the cache, and if present we re-use the previous result.

Rather than merely pruning a block B, we may instead *replace* it with the sequence of blocks constituting the feasible region. However, as explained in Example 3, we must take care when doing so not to lose any cardinality information; otherwise, our propagation may inadvertently *increase* the domain size.

Example 3 *Consider matching $B = \{a, b, c\}^{0,4}$ against dashed string Y, and finding the corresponding match to be $Z = \{a, c\}^{0,2}\{b, c\}^{3,4}$ (with mandatory and feasible regions coinciding).*

Replacing B with Z is tempting, as it provides more precise information about the sequencing *of characters. However, the maximum cardinality of Z is greater than that of B, so there are strings in $\gamma(Z)$ which were not in $\gamma(B)$.*

In fact, in this case the search space would be larger after the replacement (e.g., ccccc $\in \gamma(Z) - \gamma(B)$). Even if this were not the case, non-monotonicity may cause problems – e.g., constraints which were found to be satisfied (and thus safe to deactivate) may no longer be so after this update.

We adopt a simple strategy to avoid this behaviour when refining $B = S^{l,u}$. Let $[l', u']$ be the lower and upper cardinality bounds, derived respectively from the mandatory and feasible regions, and S' the union of all the blocks of the feasible region. If $l > l' \vee u' > u$, or the mandatory region is empty (that is, $eet(B) < lst(B)$), we restrict ourselves to refine B as we shall see later, without any replacement.

Otherwise, it is safe to replace B with the matching range from Y (setting the lower bound of optional blocks to 0). However, this is not necessarily beneficial: replacing $S^{0,10}$ with $S^{0,1} \cdots S^{0,1}$ preserves the set of models, but increases the search space by introducing symmetries.

We therefore split the feasible region into three parts: the mandatory region $M = M_1 \cdot \ldots \cdot M_k$, and a (possibly empty) prefix/suffix on either side. From M we discard all blocks incompatible with B, yielding M_B, then use CRUSH_S to make the prefix and suffix into single blocks P and Q (compatible with B), finally replacing B with $P \cdot M_B \cdot Q$.

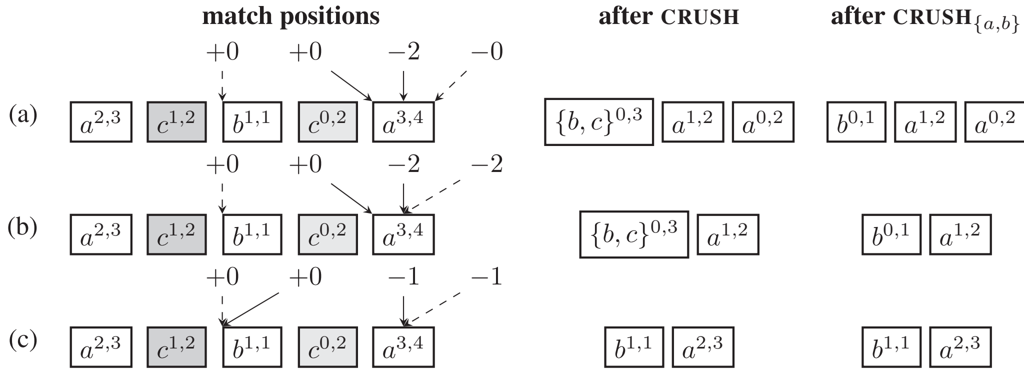

Figure 9: Refining $B = \{a, b\}^{3,4}$, given several possible matchings.

Example 4 *Recall the earliest matching computed in Example 1. Figure 9 illustrates several possible completions of this matching. In each case, we show two versions of the possible matching: using* CRUSH*, and the more precise* CRUSH$_{\{a,b\}}$*. In case (a), the computed matches are too weak to perform any filtering. In case (b), computing match cardinality without considering B yields an upper bound of 5, which is too weak to allow filtering. However, discarding blocks incompatible with B refines this to $\{b\}^{0,1}\{a\}^{1,2}$, which allows us to reduce the maximum cardinality of B to 3. In case (c), the cardinality of the matching is at least as precise as the original block. In this case, we replace B with corresponding matching $\{b\}^{1,1}\{a\}^{2,3}$.*

Note that this procedure does not exploit all possibilities for propagation. In case (b), since the maximum cardinality is now 3, we can replace block $\{a, b\}^{3,4}$ by $\{b\}^{1,1}\{a\}^{2,2}$.

Evaluation

We implemented the SWEEP algorithm as an extension of G-STRINGS solver. To distinguish this version of G-STRINGS from that introduced in (Amadini et al. 2017), which uses the COVER algorithm for propagating equality, we use the notation G-SWEEP and G-COVER respectively.

We first evaluated the performance of G-COVER and G-SWEEP approaches by using the same benchmarks of (Amadini et al. 2017), consisting of six problems, namely: a^{n}b^n, ChunkSplit, HammingDistance, Levenshtein, SQLInjection, StringReplace. For each of these problems we derived 5 instances by varying the maximum string length $\ell \in \{250, 500, 1000, 5000, 10000\}$.

In addition to G-COVER and G-SWEEP, we also evaluated state-of-the-art string solvers such as: the SMT solvers CVC4 (Liang et al. 2014) and Z3STR3 (Berzish, Zheng, and Ganesh 2017),[2] and the CP solver GECODE+$\mathbb{S}$ (Scott et al. 2017). Furthermore, thanks to the MiniZinc translation to integers defined in (Amadini et al. 2016) statically mapping string variables into arrays of integer variables, we were able to evaluate state-of-the-art constraint solvers over integers such as: GECODE (Gecode 2017), a CP solver on Finite Domains; CHUFFED (Chu 2011), a CP solver with lazy clause generation (Ohrimenko, Stuckey, and Codish 2009); and IZPLUS (Fujiwara 2016), a CP solver that also exploits local search.

Average solving times over (Amadini et al. 2017) benchmark are shown in Table 1 (solving time is set to the timeout $T = 600$ seconds if a solver can not solve a problem).[3] We ignore the model construction time, that for CP solvers using MiniZinc translations (the first three rows in the table) can be too expensive.

On the left side of the table, we report the average times by varying the maximum string length ℓ. Clearly G-SWEEP has the best performance, giving an instantaneous answer for all the problems regardless of ℓ. G-COVER still provides good results, although performance deteriorates as ℓ grows. The CP solver GECODE+$\mathbb{S}$ follows the same trend, but with worse performance. Although the performance of SMT solvers CVC4 and Z3STR3 is rather independent from ℓ, their effectiveness seems to be worse than the CP string approaches. Unsurprisingly, CP solvers that statically translate into arrays of integer variables are not very effective. This is particularly true when ℓ becomes big.

The right side of the table, where results are averaged for each problem class, better explains the reasons for such performance. Both G-COVER and G-SWEEP are instantaneous on all the problems from the NORN (Abdulla et al. 2015) benchmarks, i.e., all the problems except SQLInjection. This problem was introduced in (Amadini et al. 2016) and basically consists in solving the string equation $\omega = p \cdot e \cdot b_1 \cdot = \cdot b_2 \cdot e \cdot s$ where:

- ω is a random-generated fixed string of parametric length ℓ, meant to be an input string;
- p, s are string variables representing the prefix and the suffix of the right hand side of the equation;
- = is a fixed string of length 1, representing character '=';
- b_1, b_2 are string variables representing (possibly empty) sequences of white spaces;
- e is a string variable representing a non-empty expression.

[2]We used the last stable releases Z3STR3 1.0.0 and CVC4 1.5.

[3]We ran the experiments on Ubuntu 15.10 machines with 16 GB of RAM and 2.60 GHz Intel® i7 CPU. The source code is publicly available at (G-Strings 2017)

Table 1: Average results in seconds. Unsatisfiable problem are marked with *. Best performance in bold font.

	ℓ					Problems					
	250	500	1000	5000	10000	$a^n b^n$*	Chunk.	Leven.	Hamm.*	SQL.	St.Rep.
CHUFFED	1.76	94.46	208.01	312.47	415.04	117.62	361.59	17.8	18.8	471.77	250.51
GECODE	0.63	21.04	89.07	403.14	406.38	269.57	243.14	6.21	5.59	335.25	244.55
IZPLUS	22.55	104.74	106.8	440.22	460.57	244.34	243.49	17.81	186.64	505.9	163.68
Z3STR3	200.83	202.54	200.78	200.65	204.29	600.0	0.28	3.24	0.01	600.0	7.38
CVC4	100.05	100.04	100.05	100.04	100.04	0.02	0.2	0.01	0.01	600.0	0.01
GECODE+$\mathbb{S}$	0.29	2.9	35.66	269.22	371.24	246.23	279.78	28.28	**0.0**	66.1	194.79
G-COVER	0.01	1.94	5.63	5.22	100.0	**0.0**	**0.0**	**0.0**	**0.0**	135.35	**0.0**
G-SWEEP	**0.0**	**0.0**	**0.0**	**0.01**	**0.01**	**0.0**	**0.0**	**0.0**	**0.0**	**0.02**	**0.0**

Table 2: Average solving time, in seconds, on new SQL benchmarks. Best performance in bold font.

	SAT					UNS					ALL				
ℓ	250	500	1000	5000	10000	250	500	1000	5000	10000	250	500	1000	5000	10000
GECODE+$\mathbb{S}$	270.09	251.75	356.96	600.0	600.0	0.17	3.41	130.69	600.0	600.0	135.13	127.58	243.82	600.0	600.0
G-COVER	61.28	285.34	469.96	600.0	600.0	306.32	600.0	600.0	600.0	600.0	183.8	442.67	534.98	600.0	600.0
G-SWEEP	**0.01**	**0.08**	**0.05**	**0.52**	**1.05**	**0.04**	**0.35**	**1.84**	**69.58**	**327.92**	**0.03**	**0.21**	**0.95**	**35.05**	**164.49**

Table 3: Percentage of solved instances on new SQL benchmarks. Best performance in bold font.

	SAT					UNS					ALL				
ℓ	250	500	1000	5000	10000	250	500	1000	5000	10000	250	500	1000	5000	10000
GECODE+$\mathbb{S}$	55.0	60.0	45.0	0.0	0.0	**100.0**	**100.0**	90.0	0.0	0.0	77.5	80.0	67.5	0.0	0.0
G-COVER	90.0	55.0	25.0	0.0	0.0	50.0	0.0	0.0	0.0	0.0	70.0	27.5	12.5	0.0	0.0
G-SWEEP	**100.0**	**100.0**	**100.0**	**100.0**	**100.0**	**100.0**	**100.0**	**100.0**	**100.0**	**100.0**	**100.0**	**100.0**	**100.0**	**100.0**	**100.0**

As the name suggests, solving this problem means finding, if there exists, a potentially dangerous pattern in a SQL query (e.g., the 'classic' string "`OR 0=0`" appended at the end of the query). Note that this problem can not be solved with regular expressions, since the language denoted by $p \cdot e \cdot b_1 \cdot = \cdot b_2 \cdot e \cdot s$ is not even context free.

The main difficulty for the SQLInjection problem consists in dealing with a potentially very long fixed string ω. This means that ω is represented by a long sequence of known blocks. As previously mentioned, this is the worst-case scenario for COVER, and indeed its performance can severely degrade when $\ell = |\omega|$ grows. Consequently, for this problem COVER performs worse than both SWEEP and GECODE+$\mathbb{S}$. This behaviour was observed also in (Amadini et al. 2017). Conversely, SWEEP performs very well even for this problem since the sweep-based propagation is able to scan the blocks of the dashed strings in linear time.

Unfortunately, as also pointed out in (Amadini et al. 2017; 2016; Scott et al. 2017), there is a lack of challenging string benchmarks. As can be seen from Table 1, the most difficult problem is SQLInjection. We then randomly generated 200 harder instances from this problem; in particular, for each $\ell \in \{250, 500, 1000, 5000, 10000\}$ we generated 20 satisfiable and 20 unsatisfiable instances.

Results are shown in Table 2 (average solving time, with timeout $T = 600$ seconds) and Table 3 (percentage of solved instances), where we only compare the three best solvers from the first experiment.

These harder problems confirm the results of the initial experiments. Since these problem involve long strings GECODE+$\mathbb{S}$ is actually better that G-COVER particularly on unsatisfiable instances. But G-SWEEP is clearly superior to both, particularly on satisfiable instances.

Conclusions

We examined the problem of string constraint solving over bounded-length string variables. We started from a recently introduced Constraint Programming approach, based on a restricted class of regular expressions called dashed strings.

We improved on previous works by defining SWEEP, a new sweep-based approach for propagating the equality between dashed strings. The strength of SWEEP does not reside in the amount of pruning it performs. Its effectiveness is since it can check satisfiability in $O(n + m)$, and while filtering is more complex, in practice it is also linear. This allows the approach to scale to dashed strings that consist of a large number of blocks.

The SWEEP algorithm, that we implemented as an extension of the G-STRINGS solver, substantially outperforms other state-of-the-art approaches for string constraint solving, whether based on dashed strings or not.

There are many future directions for interesting research. Extending the set of constraints supported by the string solver to include constraints such as regular, global cardinality and lexicographic constraints is clearly of interest. Exploring new kinds of search strategy is also important because of the potentially huge search spaces that arise in string problems. Also challenging is to add explanations to dashed string constraint solving in order to make use of nogood learning approaches.

Acknowledgments

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

References

Abdulla, P. A.; Atig, M. F.; Chen, Y.; Holík, L.; Rezine, A.; Rümmer, P.; and Stenman, J. 2015. Norn: An SMT solver for string constraints. In *CAV*, volume 9206 of *LNCS*, 462–469. Springer.

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

Amadini, R.; Flener, P.; Pearson, J.; Scott, J. D.; Stuckey, P. J.; and Tack, G. 2016. Minizinc with strings. In *LOPSTR 2016*, 41–57.

Amadini, R.; Gange, G.; Stuckey, P. J.; and Tack, G. 2017. A novel approach to string constraint solving. In *CP*, Lecture Notes in Computer Science, 3–20. Springer.

Berzish, M.; Zheng, Y.; and Ganesh, V. 2017. Z3str3: A string solver with theory-aware branching. *CoRR* abs/1704.07935.

Bisht, P.; Hinrichs, T. L.; Skrupsky, N.; and Venkatakrishnan, V. N. 2011. WAPTEC: Whitebox analysis of web applications for parameter tampering exploit construction. In *CCS*, 575–586. ACM.

Bjørner, N.; Tillmann, N.; and Voronkov, A. 2009. Path feasibility analysis for string-manipulating programs. In *TACAS*, volume 5505 of *LNCS*, 307–321. Springer.

Chu, G. 2011. *Improving Combinatorial Optimization*. Ph.D. Dissertation, Department of Computing and Information Systems, University of Melbourne.

Emmi, M.; Majumdar, R.; and Sen, K. 2007. Dynamic test input generation for database applications. In *ISSTA*, 151–162. ACM.

Fujiwara, T. 2016. iZplus. http://www.minizinc.org/challenge2016/description_izplus.txt. Accessed Sept 2017.

G-Strings. 2017. Source code. Accessed November 2017. https://bitbucket.org/robama/g-strings.

Gange, G.; Navas, J. A.; Stuckey, P. J.; Søndergaard, H.; and Schachte, P. 2013. Unbounded model-checking with interpolation for regular language constraints. In *TACAS*, volume 7795 of *LNCS*, 277–291. Springer.

Gecode. 2017. Source code. www.gecode.org. Accessed Sept 2017.

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

Kiezun, A.; Ganesh, V.; Artzi, S.; Guo, P. J.; Hooimeijer, P.; and Ernst, M. D. 2012. HAMPI: A solver for word equations over strings, regular expressions, and context-free grammars. *ACM Trans. Softw. Eng. Methodol.* 21(4):25:1–25:28.

Li, G., and Ghosh, I. 2013. PASS: String solving with parameterized array and interval automaton. In *HVC*, volume 8244 of *LNCS*, 15–31. Springer.

Liang, T.; Reynolds, A.; Tinelli, C.; Barrett, C.; and Deters, M. 2014. A DPLL(T) theory solver for a theory of strings and regular expressions. In *CAV*, volume 8559 of *LNCS*, 646–662. Springer.

Ohrimenko, O.; Stuckey, P. J.; and Codish, M. 2009. Propagation via lazy clause generation. *Constraints* 14(3):357–391.

Saxena, P.; Akhawe, D.; Hanna, S.; Mao, F.; McCamant, S.; and Song, D. 2010. A symbolic execution framework for JavaScript. In *S&P*, 513–528. IEEE Computer Society.

Scott, J. D.; Flener, P.; Pearson, J.; and Schulte, C. 2017. Design and implementation of bounded-length sequence variables. In *CPAIOR*, volume 10335 of *LNCS*, 51–67. Springer.

Tateishi, T.; Pistoia, M.; and Tripp, O. 2013. Path- and index-sensitive string analysis based on monadic second-order logic. *ACM Trans. Softw. Eng. Methodol.* 22(4):33:1–33:33.

Thomé, J.; Shar, L. K.; Bianculli, D.; and Briand, L. C. 2017. Search-driven string constraint solving for vulnerability detection. In *ICSE 2017, Buenos Aires, Argentina, May 20-28, 2017*, 198–208.

Trinh, M.; Chu, D.; and Jaffar, J. 2014. S3: A symbolic string solver for vulnerability detection in web applications. In *SIGSAC*, 1232–1243. ACM.

Yu, F.; Alkhalaf, M.; and Bultan, T. 2010. Stranger: An automata-based string analysis tool for PHP. In *TACAS*, volume 6015 of *LNCS*, 154–157. Springer.