Propagating lex, find and replace with dashed strings

Roberto Amadini, Graeme Gange, and Peter J. Stuckey
{roberto.amadini,gkgange,pstuckey}@unimelb.edu.au

Department of Computing and Information Systems
The University of Melbourne, Australia

Abstract. Dashed strings have been recently proposed in Constraint Programming to represent the domain of string variables when solving combinatorial problems over strings. This approach showed promising performance on some classes of string problems, involving constraints like string equality and concatenation. However, there are a number of string constraints for which no propagator has yet been defined. In this paper, we show how to propagate lexicographic ordering (lex), find and replace with dashed strings. All of these are fundamental string operations: lex is the natural total order over strings, while find and replace are frequently used in string manipulation. We show that these propagators, that we implemented in G-Strings solver, allows us to be competitive with state-of-the-art approaches.

1 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 [8, 7], model checking [11], and web security [6, 21].

Various approaches to string constraint solving have been proposed, falling in three rough families: automata-based [13, 15, 20], word-equation based [5, 16] and unfolding-based (using either bit-vector solvers [14, 17] or constraint programming (CP) [18]). Automaton and word-equation approaches allow reasoning about unbounded strings, but are limited to constraints supported by the corresponding calculus – they have particular problems combining string and integer constraints. These approaches both have scalability problems: from automata growth in the first case, and disjunctive case-splitting in the second. Unfolding approaches first select a length bound k, then substitute each string variable with a fixed-width vector (either by compiling down to integer/bit-vector constraints [2, 14, 17] or using dedicated propagators [18]). This adds flexibility but sacrifices high-level relationships between strings, and can become very expensive when the length bound is large – even if generated solutions are very short.

A recent CP approach [4, 3] introduced the *dashed-string* representation for string variables, together with efficient propagators for dashed-string equality and related constraints (concatenation, reversal, substring). However, numerous other string operations arise in programs and constraint systems involving

strings. In this paper, we focus on two: lexicographic ordering and find. Lexicographic ordering is the most common total ordering over strings, and is frequently used to break variable symmetries in combinatorial problems. Find (or *indexOf*) identifies the first occurrence of one query string in another. It appears frequently in problems arising from verification or symbolic execution, and is a convenient building-block for expressing other constraints (e.g., replacement and non-occurrence).

The original contributions of this paper are: *(i)* new algorithms for propagating lexicographic order, find and replace with dashed strings; *(ii)* the implementation of these algorithms in the G-Strings solver, and a performance evaluation against the state-of-the-art string solvers CVC4 [16] and Z3str3 [5]. Empirical results show that our approach is highly competitive – it often outperforms these solvers in terms of solving time and solution quality.

Paper structure. In Section 2 we give background notions about dashed strings. In Section 3 we show how we propagate lexicographic ordering. Section 4 explains find and replace propagators. In Section 5 we validate our approach with different experimental evaluations, before concluding in Section 6.

2 Preliminaries

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 $i = 1, \ldots, k$, 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 $\{a\}^{l,u}$ as $a^{l,u}$, and $\{a\}^{l,l}$ as a^l. 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 and we consider only *normalised* dashed strings, where the *null element* $\emptyset^{0,0}$ occur at most once and adjacent blocks have distinct bases. For each pair $C = (l, u)$, we define $\mathsf{lb}(C) = l$ and $\mathsf{ub}(C) = u$. We indicate with $<$ the total order over characters of Σ, and with $\prec$ the lexicographic order over Σ^*.

Let $\gamma(S^{l,u}) = \{x \in S^* \mid l \leq |x| \leq u\}$ be the language denoted by block $S^{l,u}$. We extend γ to dashed strings: $\gamma(S_1^{l_1,u_1} \cdots S_k^{l_k,u_k}) = (\gamma(S_1^{l_1,u_1}) \cdots \gamma(S_k^{l_k,u_k})) \cap \mathbb{S}$ (intersection with $\mathbb{S}$ excludes the strings with length greater than ℓ). A dashed string X is *known* if it denotes a single string: $|\gamma(X)| = 1$. A block of the form $S^{0,u}$ is called *nullable*, i.e. $\epsilon \in \gamma(S^{0,u})$. There is no upper bound on the number of blocks in a dashed string since an arbitrary number of nullable blocks may occur. 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 overestimate 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 $\sqsubseteq$.

Fig. 1: Representation of $X = \{\texttt{B,b}\}^{1,1}\{\texttt{o}\}^{2,4}\{\texttt{m}\}^{1,1}\{\texttt{!}\}^{0,3}$. Each string of $\gamma(X)$ starts with `B` or `b`, followed by 2 to 4 `o`s, one `m`, then 0 to 3 `!`s.

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 n characters of S_i, with $0 \leq n \leq u_i - l_i$, *may* occur. Consider, for example, dashed string $X = \{\texttt{B,b}\}^{1,1}\{\texttt{o}\}^{2,4}\{\texttt{m}\}^{1,1}\{\texttt{!}\}^{0,3}$ illustrated in Fig. 1.

Given a dashed string X, we shall refer to *offset positions* (i, o), where i refers to block $X[i]$ and o is its *offset*, indicating how many characters from the beginning of $X[i]$ we are considering. Note that, while positions are 1-based, offset are 0-based to better represent the beginning (or dually the end) of a block. Positive offsets denote positions relative to the beginning of $X[i]$, and negative offsets are relative to the end of $X[i]$. For example, position $(2, 3)$ of $X = a^{1,2}b^{0,4}c^3$ refers to 3 characters after the beginning of second block $b^{0,4}$; this position can be equivalently expressed as $(2, -1)$.

The *index* into a string $w \in \Sigma^*$ indicates a character position in range $1..|w|$, assuming that character positions are 1-based. For example, the index of the first occurrence of *"abc"* in "*dfababcdeabc*" is 5.

Converting between *indices* and *positions* is relatively straightforward (see the pseudo-code below), though this conversion might lose precision when dealing with blocks $S^{l,u}$ having non-fixed cardinality, i.e., having $l < u$. Indeed, we may have $\textsc{pos-to-min-ind}(X, \textsc{ind-to-min-pos}(X, i)) < i$. Consider for example $X = a^{0,1}b^{0,2}$ and $i = 3$: we have $\textsc{ind-to-min-pos}(X, i) = (2, 1)$ and $\textsc{pos-to-min-ind}(2, 1) = 2$.

function IND-TO-MIN-POS($X = S_1^{l_1,u_1} \cdot \ldots \cdot S_n^{l_n,u_n}, idx$)
  $i \leftarrow 1$
  $o \leftarrow idx - 1$
  while $i \leq n \wedge o \geq u_i$ **do**
    $(i, o) \leftarrow (i + 1, o - u_i)$
  end while
  return (i, o)
end function
function POS-TO-MIN-IND($X, (i, o)$)
  return $1 + o + \sum_{j=1}^{i-1} l_j$
end function

2.1 Sweep Algorithm

Equating dashed strings X and Y requires determining two dashed strings X' and Y' such that: *(i)* $X' \sqsubseteq X$, $Y' \sqsubseteq Y$; and *(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 [3] the authors propose Sweep, an algorithm for equating dashed strings. Sweep works analogously to the sweep algorithm for timetable reasoning of cumulative [1]: 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 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 $Y[j]$.

2.2 G-Strings Solver

The Sweep algorithm is implemented in G-Strings,[1] an extension of Gecode solver [12]. It implements the domain of every string variable X with a dashed string $dom(X)$, and defines a *propagator* for each string constraint. Propagators take advantage of Sweep for refining the representations of the involved variables. For example, string equality $X = Y$ is simply propagated by equating $dom(X)$ and $dom(Y)$ with Sweep; the propagator for $Z = X \cdot Y$ is implemented by equating $dom(Z)$ and $dom(X) \cdot dom(Y)$, where $dom(X) \cdot dom(Y)$ is the concatenation of the blocks of $dom(X)$ and $dom(Y)$, taking care of properly projecting the narrowing of $dom(X) \cdot dom(Y)$ to $dom(X)$ and $dom(Y)$.

G-Strings implements constraints like 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 strategies for branching on variables (e.g., the one with smallest domain size or having the domain with the minimum number of blocks) and domain values (by heuristically selecting first a block, and then a character of its base).

3 Lexicographic ordering

The constraint $\text{lex}_{\preceq}(X, Y)$ enforces a lexicographic ordering between X and Y. Propagating lex on an unfolded sequence is largely straightforward: roughly speaking, we walk pointwise along X and Y to find the first possible difference, and impose the ordering on that element. An extension of this procedure which enforces domain consistency is given in [10]. Unfortunately, dashed strings represent sequences where we do not know the exact cardinality of each block.

3.1 Lexicographic bounds on dashed strings

Propagation of $\text{lex}_{\preceq}(X, Y)$ essentially reduces to propagating two unary constraints: $X \preceq \max_{\preceq}(D(Y))$, and $\min_{\preceq}(D(X)) \preceq Y$. However, the behaviour of

[1] G-Strings is publicly available at `https://bitbucket.org/robama/g-strings`.

X	$\min_{\preceq}(D(X))$	$\max_{\preceq}(D(X))$
$c^{0,10} \cdot a^{0,10} \cdot d^{0,10}$	$c^0 a^0 d^0$	$c^0 a^0 d^{10}$
$c^{0,10} \cdot a^{1,10} \cdot d^{0,10}$	$c^0 a^1 d^0$	$c^{10} a^1 d^{10}$
$c^{0,10} \cdot a^{1,10} \cdot d^{1,10}$	$c^0 a^{10} d^1$	$c^{10} a^1 d^{10}$
$a^{1,10} \cdot c^{0,10} \cdot d^{1,10}$	$a^{10} c^{10} d^1$	$a^1 a^0 d^{10}$
$c^{1,10} \cdot d^{0,10} \cdot a^{1,10}$	$c^1 d^0 a^1$	$c^1 d^{10} a^{10}$
$a^{0,10} \cdot d^{0,10} \cdot a^{1,10} \cdot d^{0,10}$	$a^0 d^0 a^1 d^0$	$a^0 d^{10} a^1 d^{10}$
$a^{0,10} \cdot d^{0,10} \cdot a^{1,10} \cdot d^{1,10}$	$a^{10} d^0 a^{10} d^1$	$a^0 d^{10} a^1 d^{10}$

(a)

	$a^{0,10}$	$d^{0,10}$	$a^{1,10}$	$e^{1,10}$
min. succ	a^+	a^+	e^-	\$
lex. min.	a^{10}	d^0	a^{10}	e^1
max. succ	d^-	a^+	e^-	\$
lex. max.	a^0	d^{10}	a_1	e^{10}

(b)

Fig. 2: (a) Lexicographic bounds for several dashed strings. (b) Least and greatest successors for each block of $a^{0,10} \cdot d^{0,10} \cdot a^{1,10} \cdot e^{1,10}$. The notation d^+ (resp. d^-) indicates the least/greatest successor of the block begins with a finite sequence of d's, followed by some character $x > d$ (resp. $x < d$).

max and min under $\preceq$ is perhaps counterintuitive. In the lexicographic minimum (resp. maximum), each block $S^{l,u}$ takes its least (greatest) character, and *either* its minimum cardinality l *or* its maximum cardinality u: but which cardinality is chosen depends on the following blocks, i.e., the *suffix* of the string.

Consider the string $a^n \cdot b \cdot X$, for some characters $a, b \in \Sigma$ and string $X \in \Sigma^*$. If $a < b$, increasing the number n of a's produces a smaller string under $\preceq$. If instead $a > b$, adding successive a's can only result in bigger strings under $\preceq$. If $a = b$, then we must recursively consider the prefix of X.

But we do not need to perform this recursion explicitly. All we need to know is *(i)* the first character of the suffix, and *(ii)* whether the sequence increases or decreases afterwards. In essence, we need to know whether the suffix is above or below the infinite sequence a^∞. We use the notation a^+ to denote a value infinitesimally greater than a^∞, but smaller than the successor $succ_{\leq}(a)$ of a in Σ under $\leq$; similarly, a^- denotes a value infinitesimally smaller than a but greater than $(pred_{\leq}(a))^\infty$, where $pred_{\leq}(a)$ is the predecessor of a in Σ under $\leq$.

Example 1. Figure 2(a) shows the behaviour of the lexicographic min- and max- for several dashed strings. The last two instances highlight a critical point: if the minimum value of the base of a block matches the minimum *immediate* successor, we must consult the character appearing *after* the contiguous run. Figure 2(b) illustrates the computation of least and greatest successors for a dashed string.

The computation processes the blocks in reverse order, updating the current suffix for each block. The suffix is initially \$, a special terminal character such that $\$ \prec x$ for each $x \in \Sigma^*$. The final block is non-empty, so the suffix is updated; and is greater than the current suffix, so becomes e^-. The next block is again non-empty, and below the current suffix, so it is updated to a^+. The next block is nullable, and its value d is greater than the current suffix. So, for computing the lex-min we omit this block, carrying the suffix backwards. We then reach $a^{0,10}$ with a successor of a^+. This gives us successors $[a^+, a^+, e^-, \$]$. We can then compute the lex-minimising value for each block by looking at its successor: the first block $a^{0,10}$ has successor a^+. As $a < a^+$, the block takes maximum

Call	Result	X'
LEX-STEP($\{d,e\}^{0,1}$,a,Y_{max},(1,0))	**EQ**((1,0))	ϵ
LEX-STEP($\{f,g\}^{0,4}$,d,Y_{max},(1,0))	**EQ**((1,0))	ϵ
LEX-STEP($\{d,e,f\}^{2,4}$,a,Y_{max},(1,0))	**EQ**((1,2))	$\{d\}^{2,2}$
LEX-STEP($\{a,b,c\}^{0,3}$,a,Y_{max},(1,2))	**LT**	$\{d\}^{2,2}\{a,b,c\}^{0,3}\{f,g\}^{1,4}$

Table 1: Calling LEX-STEP($\{d,e\}^{0,1}$,a,Y_{max},(1,0)).

cardinality a^{10}. The next block $d^{0,10}$ again has successor a^+. But this time, since $d > a^+$, the block takes its minimum cardinality 0. This process is repeated for each block to obtain the lexicographically minimum string. The pseudo-code for computing the minimum successor and the lex-min value is shown in Fig. 3. □

```
function MIN-SUCC(X = S_1^{l_1,u_1} · ... S_n^{l_n,u_n})
    M_succ ← ∅
    suff ← $
    for i ∈ n ... 1 do
        M_succ[i] ← suff
        c ← min(S_i)
        if c < suff then
            suff ← c^+
        else if l_i > 0 then
            suff ← c^-
        end if
    end for
    return M_succ
end function
```

```
function LEX-MIN(S^{l,u}, c_succ)
    c ← min(S)
    if c < c_succ then
        return c^u
    else
        return c^l
    end if
end function
```

Fig. 3: Computing the minimum successor for each block of X, and the lex-min value for block $S^{l,u}$ given the computed successor.

The LEX propagation for $X \preceq \max_{\preceq}(D(Y))$ is implemented by LEX-X shown in Figure 4. Essentially it walks across the blocks of X, by comparing them with $\max_{\preceq}(D(Y))$. If it finds that the block must violate LEX, then it returns **UNSAT**. If the block may be strictly less than $\max_{\preceq}(D(Y))$ we terminate, the constraint can be satisfied. If the lower bound of the block equates to $\max_{\preceq}(D(Y))$ we force it to take its lower bound value, and continue processing the next block.

The *strict* parameter of LEX-X is a Boolean flag which is true if and only if we are propagating the strict ordering $\prec$. The propagation of $\prec$ is exactly the same of $\preceq$, with the only difference that if dashed string X is completely consumed after the loop then we raise a failure (this means that $\min_{\preceq}(D(X)) \geq \max_{\preceq}(D(Y))$).

Example 2. Let X be $\{d,e\}^{0,1}\{f,g\}^{0,4}\{d,e,f\}^{2,4}\{a,b,c\}^{0,3}\{f,g\}^{1,4}$ and $Y = \{a,b\}^{0,3}\{c,d\}^{0,3}\{a\}^{1,3}\{d,e\}^{1,3}$. For propagating $X \preceq Y$, we first compute X_{succ} as $[a,d,a,f,\$]$. Then, $Y_{max} = \max_{\preceq}(D(Y)) = dddaeee = \{d\}^{3,3}\{a\}^{1,1}\{e\}^{3,3}$.

```
function LEX-X(X, Y, strict)
    Let X = X_1 · ... · X_n.
    X_succ := MIN-SUCC(X)
    Y_max ← max_⪯(Y)
    pos ← ⟨1, 0⟩
    X' ← [ ]
    for j ∈ 1...n do
        match LEX-STEP(X_j, X_succ[j], Y_max, pos) with
            case UNSAT ⇒
                return UNSAT
            case LT ⇒
                X' ← X' · X_j · ... · X_n
                break
            case EQ(pos') ⇒
                X' ← X' · LEX-MIN(X_j, X_succ[j])
                pos ← pos'
        end
    end for
    if strict ∧ j = n then
        return UNSAT
    end if
    POST(X ← X')
    return SAT
end function
```

Fig. 4: Algorithm for propagating $X \preceq \max_{\preceq}(D(Y))$.

The calls to LEX-STEP, its returning values, and the progression of values for X' are shown in Table 1 (as a result of the propagation, X is replaced by X').

4 Find and replace

Find and replace are important and widely used string operations, since much string manipulation is achieved using them. Formally, $\text{FIND}(Q, X)$ returns the start index of the *first* occurrence of Q in X, or 0 if Q does not occur (assuming 1-based indexing). Similarly, $\text{REPLACE}(Q, R, X)$ returns X with the first occurrence of Q replaced by R (returning X if Q does not occur in X).

FIND is surprisingly difficult to express as a decomposition. An attempt might start with encoding the occurrence as $\exists a, b.\ X = a \cdot Q \cdot b \wedge |a| = idx - 1$. This ensures that Q occurs at index idx. However, it omits two important aspects: *(i)* $\text{FIND}(Q, X)$ identifies the *first* (rather than any) occurrence of Q; *(ii)* Q can be absent from X. Both problems arise from the same cause: the difficulty of encoding *non-occurrence*. To do so, we would essentially need to add a disequality constraint between Q and every $|Q|$-length substring of X (this problem will recur when we discuss encodings for unfolding-based solvers in Section 4.3). Thus, developing a specialised propagator for FIND appears prudent. As we shall see in Section 4.2, it also serves as useful primitive for implementing other constraints.

```
function LEX-STEP(X_j = S^{l,u}, succ, Y, ⟨i, o⟩)
    Let Y = {a_1^{α_1} · ... · a_m^{α_m}}.        ▷ Greatest word in the domain of variable Y
    c_S ← min(S)
    if succ > c_S then                            ▷ Better than our best successor, so saturate
        cap ← u
    else                                          ▷ Use only what we must.
        cap ← l
    end if
    while cap > 0 do
        if i > m then                             ▷ No more Y to match.
            return UNSAT
        end if
        c_Y ← a_i
        card_Y ← α_i − o
        if cap > 0 ∧ c_S > c_Y then               ▷ Greater than the next character in Y
            return UNSAT
        else if cap > 0 ∧ c_S < c_Y then
            return LT                             ▷ Globally satisfied
        else
            if card_Y ≤ cap then
                cap ← cap − card_Y
                (i, o) ← (i + 1, 0)
            else
                return EQ(⟨i, o + cap⟩)           ▷ Bounds coincide so far.
            end if
        end if
    end while
    return EQ(⟨i, o⟩)
end function
```

Fig. 5: Processing a single block X_j of the left operand of $\text{LEX}_{\leq}$.

4.1 Find

The constraint $I = \text{FIND}(X, Y)$ returns the index I of the character in Y where the string X appears first, or 0 if X is not a substring of Y. Note the use of 1-based indexing, which is standard in maths (and mathematical programming systems), but not so standard in computer science.[2]

The propagator for $I = \text{FIND}(X, Y)$ is implemented using the PUSH and STRETCH algorithms used by the G-STRINGS solver to implement equality of dashed strings. $\text{PUSH}^+(B, Y, (i, o))$ attempts to find the earliest possible match of B after o characters into $Y[i]$ (the i^{th} block of Y), while $\text{STRETCH}^+(B, Y, (i, o))$ 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.

[2] We use this indexing to comply with MiniZinc indexing [2]. However, translating between this and the corresponding 0-indexed operations (e.g., the Java `indexOf` method or the C++ `find` method) is trivial.

```
function PROP-FIND-MIN(I, [X_1, ..., X_n], [Y_1, ..., Y_m])
    start ←IND-TO-MIN-POS([Y_1, ..., Y_m], min(D(I) − {0}))
    nochanges ← true
    repeat
        for i ∈ {1, ..., n} do                                  ▷ Scanning forwards
            if ¬ nochanges then
                start ← est(X_i)
            end if
            start, end ← PUSH^+(X_i, Y, start)
            est(X_i) ← start
            start ← end
        end for
        if end > (m, +ub(Y_m)) then
            D(I) ← D(I) ∩ {0}                                   ▷ Prefix cannot fit
            return D(I) ≠ ∅
        end if
        for i ∈ {n, n − 1, ..., 1} do                           ▷ Scanning backwards
            end ← STRETCH^−(X_i, Y, end)
            if end > est(X_i) then
                est(X_i) ← end
                nochanges ← false
            end if
        end for
        start ← end
    until nochanges
    D(I) ← D(I) ∩ ({0} ∪ {POS-TO-MIN-IND([Y_1, ..., Y_m], end).. + ∞})
    return D(I) ≠ ∅
end function
```

Fig. 6: Algorithm for propagating the earliest possible start position I of string X in the string Y.

The algorithm works by first converting the minimal index of the current domain of I into a earliest possible starting position in X. This is achieved by consuming characters from the upper bound of length of X blocks. We then use push to find the earliest position in Y where X_i can occur, for each block X_i in turn, recording this in $est(X_i)$. If the earliest end position for this process is after the end of Y we know that no match is possible, we update the domain of I, and return $false$ if it becomes empty.

The PUSH procedure may have introduced gaps between blocks. Hence we stretch backwards to pull blocks forward to their actual earliest start position assuming all later blocks are matched. This may update the earliest start positions for each block Y_i. If we found any earliest start positions changed during the stretch operation, we repeat the whole push/stretch loop until there are no changes. At the end of this, *end* holds the earliest match position for X in Y. We convert *end* to an index and update the domain of I to reflect this.

Example 3. Consider the constraint $I = \text{FIND}(X, Y)$ where $X = \{a\}^{1,1}\{b\}^{2,2}$ and $Y = \{b, c\}^{0,12}\{a\}^{3,3}\{d\}^{1,2}\{b, c\}^{2,4}\{a\}^{5,5}\{b\}^{3,3}\{a, c\}^{0,8}$, and $D(I) = \{0..38\}$. The starting position *start* from index 1 is (1,0). We begin by PUSHing the block $\{a\}^{1,1}$ to position (2,0), we then PUSH the block $\{b\}^{2,2}$ to position (4,0). Starting from $end = (4, 2)$ we STRETCH the block $\{b\}^{2,2}$ to position (4,0), but STRETCHing $\{a\}^{1,1}$ we simply return $end = (4, 0)$ because the previous block is incompatible. We repeat the main loop by PUSHing the block $\{a\}^{1,1}$ starting from (4,0) to position (5,0), and PUSH the block $\{b\}^{2,2}$ to position (6,0). Starting from $end = (6, 2)$ we STRETCH the block $\{b\}^{2,2}$ to position (6,0), and STRETCHing $\{a\}^{1,1}$ to (5,4). We repeat the main loop this time finding no change. The earliest index for the match is hence POS-TO-MIN-IND(Y,(5,4)) $= 0 + 3 + 1 + 2 + 5 =$ 11. We thus set the domain of I to $\{0\} \cup \{11..35\}$. □

The algorithm for propagating the upper bound on the index value is similar. It uses PUSH$^-$ to find the latest start position for each block, then STRETCH$^+$ to improve these. This process continues until fixpoint. Finally the index of the latest start position of first block X_1 is used to update $D(I)$.

If we have determined $0 \notin D(I)$, from either a definite match or propagation on I, we know there is some occurrence of X in Y. In this case, we know that $Y = \Sigma^{\mathsf{lb}(I)-1,\mathsf{ub}(I)-1} X \Sigma^*$, and we propagate between X and Y accordingly.

We check for definite matches only if the string X is completely fixed. We build the fixed components of Y by replacing blocks $S^{l,u}$ having non-singleton character sets ($|S| > 1$) by special character \$ $\notin \Sigma$, and singleton character blocks where $S = \{a\}$ by a string $a^l\$a^l$, unless $l = u$ in which case we use a^l. We do this because if a block is singleton character but has unfixed cardinality (e.g., $a^{3,6}$), then the block may participate in matches to either side, but cannot form matches *across* the block since its cardinality is not fixed. Thus it splits the component string, contributing its lower bound to either side (e.g. $a^3\$a^3$). We then do a substring search for X in this string. If there is a match we convert the resulting index of this string match into a position in Y, and then compute the latest possible index in Y corresponding to this position. We propagate this as an upper bound of I, and remove 0 from the domain of I.

Example 4. Consider the constraint $I = \text{FIND}(X, Y)$ where $X = \{a\}^{1,1}\{b\}^{2,2}$ and $Y = \{b, c\}^{0,12}\{a\}^{3,4}\{d\}^{1,2}\{b, c\}^{2,4}\{a\}^{5,5}\{b\}^{2,3}\{a, c\}^{0,8}$. X is fixed to string "*abb*", so we create the fixed components of Y as "\$*aaa*\$*aaad*\$*d*\$*aaaaabb*\$*bb*\$" and search for X in this string. We find a match at the 17^{th} character, which corresponds to Y position (5,4). The last possible index for this position is $27 = 12 + 4 + 2 + 4 + 5$. We update the domain of I to $\{1..27\}$. □

4.2 Replace

The constraint $Y_2 = \text{REPLACE}(X_1, X_2, Y_1)$ requires that Y_2 is the string resulting from replacing the first occurence of X_1 in Y_1 by X_2. If X_1 does not occur in Y_1, then $Y_2 = Y_1$. We encode this constraint by using concatenation and FIND as follows:

$$\exists n, A_1, A_2.\ \begin{pmatrix} n = \textsc{find}(X_1, Y_1) \\ \wedge\ |A_1| = \max(0, n-1) \\ \wedge\ Y_1 = A_1 \cdot X_1^{(n>0)} \cdot A_2 \\ \wedge\ Y_2 = A_1 \cdot X_2^{(n>0)} \cdot A_2 \\ \wedge\ \textsc{find}(X_1, A_1) = (|X_1| = 0) \end{pmatrix}$$

The encoding ensures that Y_2 is Y_1 with some part $Z_1 \in \{X_1, \epsilon\}$ replaced by $Z_2 \in \{X_2, \epsilon\}$. Index n encodes the position where we find X_1 in Y_1. If it does not occur we force $Z_1 = Z_2 = \epsilon$ which makes $Y_1 = Y_2 = A_1 \cdot A_2$. If it does occur we force $X_1 = Z_1$ and $X_2 = Z_2$, and the length of A_1 to be $n - 1$. The last constraint is redundant: it ensures X_1 appears in A_1 if and only if $X_1 = \epsilon$. It is included to strengthen propagation (due to the loss of information when converting between indices and positions).

The FIND constraint allows us to encode other sub-string like constraints, e.g., $\textsc{startsWith}(Q, X) \iff (\textsc{find}(X, Q) = 1)$ and $\textsc{contains}(Q, X) \iff (\textsc{find}(X, Q) > 0)$. Note that because FIND is functional, these decompositions can be used in any context (e.g., in negated form).

4.3 Encoding find and replace in unfolding-based solvers

As discussed in Section 4.1, expressing the *non*-existence of target strings poses difficulties. In unfolding-based solvers (e.g., Gecode+$\mathbb{S}$, or bit-vector solvers), this manifests in an encoding of $\textsc{find}(Q, R)$ which is of size $\mathsf{ub}(|Q|) \times \mathsf{ub}(|R|)$: essentially, we compute the set I of all the indices where Q and R align, then we return $\min(I)$ if $I \neq \emptyset$, otherwise we return 0:

$$\textsc{findAt}(Q, R, i) = \begin{cases} 1 & \textbf{if } \forall j = 1, \ldots, |Q| : Q[j] = R[i + j - 1] \\ 0 & \textbf{otherwise} \end{cases}$$

$$\textsc{find}(Q, R) = \begin{cases} \min(I) & \textbf{if } I \neq \emptyset \\ 0 & \textbf{otherwise} \end{cases}$$

where $I = \{i \in \{1, \ldots, |R|\} \mid \textsc{findAt}(Q, R, i) = 1\}$.

5 Experimental Evaluation

We have extended the G-Strings solver with the proposed filtering algorithms. We evaluated the propagators on three classes of problems and compared the performance of G-Strings against CVC4 [16] and Z3str3 [5], two state-of-the-art SMT solvers supporting the theory of strings.[3]

First, we consider the PISA suite, consisting of instances arising from web-application analysis. However the PISA instances are rather small and, being

[3] We used Ubuntu 15.10 machines with 16 GB of RAM and 2.60 GHz Intel® i7 CPU. Experiments available at: `https://bitbucket.org/robama/exp_cpaior_2018`

instance	sat	Z3str3	CVC4	G-Strings			
				500	1000	5000	10000
pisa-000	✓	0.02	0.12	0.00	0.00	0.00	0.00
pisa-001	✓	0.04	0.00	0.00	0.00	0.00	0.00
pisa-002	✓	0.02	0.01	0.00	0.00	0.00	0.00
pisa-003	×	0.01	0.00	0.00	0.00	0.00	0.00
pisa-004	×	0.01	0.56	0.05	0.11	0.55	1.07
pisa-005	✓	0.02	0.03	0.00	0.00	0.00	0.00
pisa-006	×	0.01	0.58	0.06	0.11	0.56	1.1
pisa-007	×	0.01	0.68	0.05	0.11	0.54	1.09
pisa-008	✓	0.03	0.01	1.44	1.44	1.45	1.44
pisa-009	✓	2.68	0.00	0.00	0.00	0.00	0.00
pisa-010	✓	0.01	0.00	0.00	0.00	0.00	0.00
pisa-011	✓	0.01	0.00	0.00	0.00	0.00	0.00
total (sat)		2.83	**0.17**	1.44	1.44	1.45	1.44
total (unsat)		**0.04**	1.82	0.16	0.33	1.65	3.26
total		2.87	1.99	**1.60**	**1.77**	3.10	4.70

Table 2: Comparison of solvers in the PISA instances. Times are given in seconds. Satisfiable problems are marked ✓ while unsatisfiable are marked ×.

heavily used SMT benchmarks, CVC4 and Z3str3 are well tuned for these; they are included here largely as a sanity check. We then evaluate scalability with two sets of constructed combinatorial instances.

PISA The PISA benchmark suite, described in [22], consists of a number of problems derived from web-application analysis. It contains find-related constraints like indexOf, lastIndexOf, replace, contains, startsWith, endsWith. We compare CVC4 and Z3str3 and G-Strings using different maximum string lengths $\ell \in \{500, 1000, 5000, 10000\}$; results are given in Table 2.

Unsurprisingly, the current versions of CVC4 and Z3str3 solve the PISA instances near instantly. For G-Strings, performance on satisfiable instances is very efficient, and independent of maximum string length. Results on the unsatisfiable instances highlight the importance of search strategy. Using a naive input-order search strategy, several of these time out because the search branches first on variables irrelevant to the infeasibility; but by specifying the first decision variable, these instances terminate quickly. This suggests integrating dashed strings into a learning solver may be worthwhile, as the limitation is in the enumeration procedure, rather than the propagators themselves. It appears the unsatisfiable instances scale linearly with maximum string length for G-Strings.

Partial de Bruijn sequences The de Bruijn sequence $B(k, n)$ (see, e.g., [9]) is a cyclic sequence containing exactly one occurrence of each string of length n in an alphabet of size k. We consider instead the following variant: given a set S of fixed strings in the ASCII alphabet, what is the shortest string X containing

solver	opt	sat	unk	ttf	time	score	borda	iborda
CVC4	0	**20**	**0**	14.2	600.0	10.0	9.6	11.5
Z3STR3	0	6	14	484.2	600.0	1.5	0.0	0.0
G-STRINGS	**6**	**20**	**0**	**0.0**	**446.0**	**15.0**	**22.4**	**20.5**

Table 3: Comparative results on the partial de Bruijn sequence problems.

solver	opt	sat	unk	ttf	time	score	borda	iborda
Z3STR3	2	9	11	358.5	546.5	5.0	3.6	8.0
CVC4	2	18	2	60.1	564.4	8.0	4.6	9.0
G-STRINGS $\lvert\cdot\rvert \leq \lvert\cdot\rvert$	15	**20**	**0**	0.1	189.8	18.4	31.3	31.0
G-STRINGS $\preceq$	**18**	**20**	**0**	**0.05**	**80.5**	**19.5**	**42.5**	**34.0**

Table 4: Comparative results on the substring selection problems.

at least one occurrence of each string in S:[4]

$$\begin{aligned} \underset{X}{\text{minimize}} \quad & |X| \\ \text{subject to} \quad & \text{CONTAINS}(X, s),\ s \in S. \end{aligned}$$

We generated sets of n random strings having total length l, for $n \in \{5, 10, 15, 20\}$ and $l \in \{100, 250, 500, 750, 1000\}$ (one instance per pair of parameters). Results are given for G-STRINGS, CVC4 and Z3STR3. We also attempted an unfolding approach, using a MiniZinc implementation [2] of the decomposition of Section 4.3 to compile down to integer constraints; however the conversion failed due to memory exhaustion even for $l = 100$. The maximum string length for G-STRINGS was set to l, being an upper bound on the minimum sequence length.

The results are shown in Table 3. It shows for each solver the number of problems where an optimal solution is proven (opt), the number where a solution was found (sat), and the number where no solution was found (unk). It then gives the average time to first solution (ttf), and solve time (where 600s is the timeout). The last three columns are comparative scores across the solvers: score gives 0 for finding no solution, 0.25 for finding the worst known solution, 0.75 for finding the best known solution, a linear scale value in (0.25, 0.75) for finding other solutions, and 1 for proving the optimal solution. borda is the MiniZinc Challenge score [19], which gives a Borda score where each pair of solvers is compared on each instance, the better solver gets 1, the weaker 0, and if they are tied the point is split inversely proportional to their solving time. iborda uses a similar border score but the comparison is just on the objective value (proving optimality is not considered better than finding the optimal solution – this is actually the incomplete score of MiniZinc challenge, devised to evaluate local search solvers).

[4] This model considers only non-cyclic sequences. For cyclic sequences, we need only to replace each occurrence of $\text{FIND}(X, s)$ with $\text{FIND}(\text{CONCAT}(X, X), s)$.

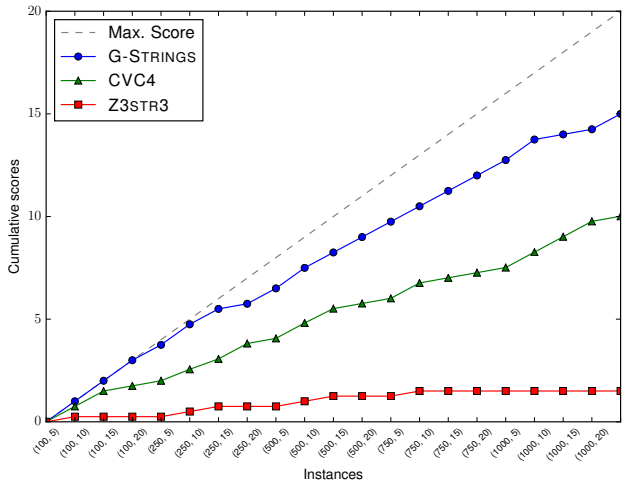

Fig. 7: Cumulative score for partial de Bruijn sequence problem.

Clearly the G-STRINGS solver significantly outperforms the other two solvers. It instantly finds a solution to all problems, and is the only solver capable of proving optimality. For these benchmarks Z3STR3 is dominated by the other two solvers (see the 0 score for Borda-based metrics).

Substring selection To test a typical application of LEX, we generated instances of a *substring selection* problem: given a set S of n strings, find the k-longest distinct substrings appearing in at least m original strings. LEX is used to break symmetries among selected substrings.

$$\begin{aligned} \underset{x_1,\ldots,x_k}{\text{maximize}} \quad & |x_1| + \ldots + |x_k| \\ \text{subject to} \quad & \sum_{s \in S} \text{CONTAINS}(s, x_i) \geq m,\ i \in 1, \ldots, k \\ & x_{i-1} \preceq x_i,\ i \in 2, \ldots, k \\ & |x_{i-1}| > 0 \rightarrow x_{i-1} \prec x_i,\ i \in 2, \ldots, k. \end{aligned}$$

As for the partial de Bruijn problem, we selected a set S of n random strings, with $n \in \{5, 10, 15, 20\}$, and we tuned the total length $l \in \{100, 250, 500, 750, 1000\}$. We then fixed $k = \frac{|S|}{2}$, and selected uniformly in $m \in \left[\frac{|S|}{2}, |S| - 1\right]$. We give results for G-STRINGS, CVC4 and Z3STR3; for unfolding approaches, the conversion again failed due to memory exhaustion.

As the SMT string theory lacks terms for lexicographic ordering, we replace the LEX symmetry breaking using $\preceq$ (the last two lines of the model) by length symmetry breaking using $|x_{i-1}| \leq |x_i|$. We used pair-wise inequalities $\bigwedge_{1 \leq i < j \leq k} x_i \neq x_j$ to have distinct strings. For G-STRINGS, we report performance with symmetry breaking either using LEX, or just on string lengths.

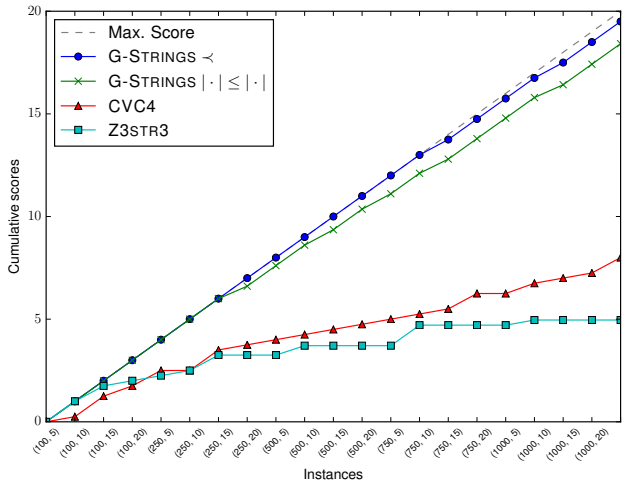

Fig. 8: Cumulative score for maximum substring selection problem.

The results shown in Table 4 clearly show that G-Strings is superior for these problems, and that using lex for symmetry breaking is significantly better than simply breaking on length, particularly for proving unsatisfiability.

Finally in Figures 7 and 8 we show the cumulative score for each solver by sorting the instances lexicographically over (l, m) parameters. We can see that G-Strings is dominant, except for the larger de Bruijn sequence problems where CVC4 is quite competitive. On the maximum substring selection problem, the lex constraints make the G-Strings performance almost perfect.

6 Conclusions

In this paper we have the continued work on dashed-string representation for string constraints, defining propagation algorithms for two constraints which lack compact decompositions: lexicographic ordering and substring search. On small verification instances, our approach is competitive with highly-tuned SMT solvers. On constructed combinatorial instances, our dashed-string based propagators substantially outperform current SMT-based string solvers. Results on the verification instances also highlight the importance of autonomous search. Future directions include therefore the study of suitable string search heuristics and the development of learning string solvers.

Acknowledgments

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

References

1. Aggoun, A., Beldiceanu, N.: Extending CHIP in order to solve complex scheduling and placement problems. Mathematical and Computer Modelling 17(7), 57–73 (Dec 1993)
2. Amadini, R., Flener, P., Pearson, J., Scott, J.D., Stuckey, P.J., Tack, G.: Minizinc with strings. In: Logic-Based Program Synthesis and Transformation - 25th International Symposium, LOPSTR 2016 (2016), https://arxiv.org/abs/1608.03650
3. Amadini, R., Gange, G., Stuckey, P.J.: Sweep-based propagation for string constraint solving. In: To appear in AAAI 2018 (2018)
4. Amadini, R., Gange, G., Stuckey, P.J., Tack, G.: A novel approach to string constraint solving. In: Beck, J.C. (ed.) Principles and Practice of Constraint Programming - 23rd International Conference, CP 2017, Melbourne, VIC, Australia, August 28 - September 1, 2017, Proceedings. Lecture Notes in Computer Science, vol. 10416, pp. 3–20. Springer (2017)
5. Berzish, M., Zheng, Y., Ganesh, V.: Z3str3: A string solver with theory-aware branching. CoRR abs/1704.07935 (2017), `http://arxiv.org/abs/1704.07935`
6. Bisht, P., Hinrichs, T.L., Skrupsky, N., Venkatakrishnan, V.N.: WAPTEC: Whitebox analysis of web applications for parameter tampering exploit construction. In: Proceedings of ACM Conference on Computer and Communications Security. pp. 575–586. ACM (2011)
7. Bjørner, N., Tillmann, N., Voronkov, A.: Path feasibility analysis for string-manipulating programs. In: Proceedings of the International Conference on Tools and Algorithms for the Construction and Analysis of Systems. LNCS, vol. 5505, pp. 307–321. Springer (2009)
8. Emmi, M., Majumdar, R., Sen, K.: Dynamic test input generation for database applications. In: Proceedings of the ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA). pp. 151–162. ACM (2007)
9. Fredricksen, H.: A survey of full length nonlinear shift register cycle algorithms. 24(2), 195–221 (1982)
10. Frisch, A.M., Hnich, B., Kiziltan, Z., Miguel, I., Walsh, T.: Propagation algorithms for lexicographic ordering constraints. Artif. Intell. 170(10), 803–834 (2006)
11. Gange, G., Navas, J.A., Stuckey, P.J., Søndergaard, H., Schachte, P.: 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. LNCS, vol. 7795, pp. 277–291. Springer (2013)
12. Gecode Team: Gecode: Generic constraint development environment (2016), available at `http://www.gecode.org`
13. Hooimeijer, P., Weimer, W.: StrSolve: Solving string constraints lazily. Automated Software Engineering 19(4), 531–559 (2012)
14. Kiezun, A., Ganesh, V., Artzi, S., Guo, P.J., Hooimeijer, P., Ernst, M.D.: 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)
15. Li, G., Ghosh, I.: PASS: String solving with parameterized array and interval automaton. In: Proceeedings of the Haifa Verification Conference. LNCS, vol. 8244, pp. 15–31. Springer (2013)
16. Liang, T., Reynolds, A., Tinelli, C., Barrett, C., Deters, M.: A DPLL(T) theory solver for a theory of strings and regular expressions. In: CAV. LNCS, vol. 8559, pp. 646–662. Springer (2014)

17. Saxena, P., Akhawe, D., Hanna, S., Mao, F., McCamant, S., Song, D.: A symbolic execution framework for JavaScript. In: S&P. pp. 513–528. IEEE Computer Society (2010)
18. Scott, J.D., Flener, P., Pearson, J., Schulte, C.: 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)
19. Stuckey, P.J., Feydy, T., Schutt, A., Tack, G., Fischer, J.: The minizinc challenge 2008-2013. AI Magazine (2), 55–60 (2014)
20. Tateishi, T., Pistoia, M., Tripp, O.: Path- and index-sensitive string analysis based on monadic second-order logic. ACM Transactions on Software Engineering Methodology 22(4), 33 (2013)
21. Thomé, J., Shar, L.K., Bianculli, D., Briand, L.C.: Search-driven string constraint solving for vulnerability detection. In: ICSE 2017, Buenos Aires, Argentina, May 20-28, 2017. pp. 198–208 (2017)
22. Zheng, Y., Ganesh, V., Subramanian, S., Tripp, O., Berzish, M., Dolby, J., Zhang, X.: Z3str2: an efficient solver for strings, regular expressions, and length constraints. Formal Methods in System Design 50(2-3), 249–288 (2017)