Explaining Propagators for String Edit Distance Constraints

Felix Winter,[1] **Nysret Musliu,**[1] **Peter J. Stuckey**[2]
[1]Christian Doppler Laboratory for Artificial Intelligence and Optimization for Planning and Scheduling
DBAI, TU Wien, Vienna, Austria
{winter,musliu}@dbai.tuwien.ac.at
[2]Monash University, Melbourne, Australia
peter.stuckey@monash.edu

Abstract

The computation of string similarity measures has been thoroughly studied in the scientific literature and has applications in a wide variety of different areas. One of the most widely used measures is the so called string edit distance which captures the number of required edit operations to transform a string into another given string. Although polynomial time algorithms are known for calculating the edit distance between two strings, there also exist NP-hard problems from practical applications like scheduling or computational biology that constrain the minimum edit distance between arrays of decision variables. In this work, we propose a novel global constraint to formulate restrictions on the minimum edit distance for such problems. Furthermore, we describe a propagation algorithm and investigate an explanation strategy for an edit distance constraint propagator that can be incorporated into state of the art lazy clause generation solvers. Experimental results show that the proposed propagator is able to significantly improve the performance of existing exact methods regarding solution quality and computation speed for benchmark problems from the literature.

Introduction

String comparison and matching are well studied topics in computer science which have spawned a large number of publications in the past e.g. (Wagner and Fischer 1974; Ukkonen 1985; Navarro 2001). One of the most widely used methods to quantify the similarity of two given strings is the so called string edit distance (Wagner and Fischer 1974), that counts the number of required edit operations to transform a string into another given string.

Algorithms that can compute the edit distance between two given strings have been thoroughly studied, and several methods that use dynamic programming have been suggested to efficiently calculate the minimum edit distance in polynomial time e.g. (Ukkonen 1985). However, there also exist NP-hard combinatorial optimization problems that aim to minimize the edit distance between strings in the literature (Nicolas and Rivals 2003; Winter and Musliu 2019). Solving such problems requires the repeated computation of the string edit distance, and although an exact algorithm using dynamic programming for one of these problems has been proposed by Kruskal (1983), such an approach has a run-time complexity which is exponential to the instance size in the worst case. Therefore, several other techniques have been proposed in the literature to tackle practically sized instances e.g. (Hayashida and Koyano 2016; Jiang et al. 2003; Olivares-Rodríguez and Oncina 2008). Although, most of these algorithms rely on approximations or heuristic techniques (Jiang et al. 2003; Olivares-Rodríguez and Oncina 2008) an exact approach using integer linear programming has been recently proposed by Hayashida and Koyano (2016). An important open research question is if methods using constraint programming can improve the results of existing approaches.

In this paper, we propose a novel global constraint propagator that can be used to efficiently solve subproblems of constraint satisfaction and combinatorial optimization problems that aim to minimize the string edit distance between arrays of decision variables. Furthermore, we investigate the incorporation of this constraint into a lazy clause generation based solver (Ohrimenko, Stuckey, and Codish 2009) and study how constraint propagation can be explained. To show the practical applicability of the proposed propagator, we solve two problems from the literature that make use of edit distance constraints (Kohonen 1985; Winter et al. 2019) using lazy clause generation and our novel propagator. We evaluate the performance of the proposed method by performing a large number of experiments for these two problems and compare the outcomes with results produced by state of the art exact methods from the literature.

Preliminaries

In this section we briefly provide background on the string edit distance and lazy clause generation, as we will later assume that the reader is familiar with these topics.

String Edit Distance

The notion of edit distance was first introduced by Wagner and Fischer (1974) and has been thoroughly studied in the literature ever since e.g. (Ukkonen 1985; Navarro 2001). In the following we give a short review of the definition as well as the traditional dynamic programming routine to calculate the edit distance.

The *edit distance* between two given strings s and t over alphabet Σ is defined as the minimum number of editing operations required to transform s into t (or vice versa). Let a given string s consist of n characters $s_1, s_2, \ldots s_n$ and another string t consist of m characters $t_1, t_2, \ldots, t_m$, then we distinguish between three different editing operations:

1. $s_i \rightarrow t_j$ denotes a change of the character at position i in string s to the character at position j in the second string t.
2. $s_i \rightarrow \epsilon$ denotes a removal of the character at position i in string s.
3. $\epsilon \rightarrow t_j$ denotes an insertion of the character at position j in string t.

The minimum edit distance between two strings s and t can be calculated with the use of dynamic programming (where ϵ denotes an empty string, $s(i)$ denotes the sub-string $s[1:i]$ of s, that is the first i characters of the string s or ϵ if $i = 0$, γ denotes the cost of an edit operation, and $d(i,j)$ denotes the minimum edit distance between $s(i)$ and $t(j)$):

$$d(0,0) = 0$$
$$d(i,j) = \min \begin{cases} d(i-1,j-1) + \gamma(s_i \rightarrow t_j) \\ d(i,j-1) + \gamma(\epsilon \rightarrow t_j) \\ d(i-1,j) + \gamma(s_i \rightarrow \epsilon) \end{cases} \quad (1)$$

The dynamic programming routine can be used to compute the edit distance between two strings as long as the following triangle inequality holds for the costs of the edit operations:

$$\gamma(s_i \rightarrow t_j) \leq \gamma(\epsilon \rightarrow t_j) + \gamma(s_i \rightarrow \epsilon) \quad (2)$$

We assume that if $s_i = t_j$ then $\gamma(s_i \rightarrow t_j) = 0$, that is the edit distance of making no change is 0.

Throughout the paper in our examples we assume $\gamma(\epsilon \rightarrow c) = \gamma(c \rightarrow \epsilon) = 1$ forall $c \in \Sigma$ and $\gamma(c_1 \rightarrow c_2) = 2$ if $c_1 \neq c_2$ forall $\{c_1, c_2\} \subseteq \Sigma$. Nevertheless, the presented techniques can be applied also on any other cost assignments, as long as the triangle inequality holds.

Example 1: As an example, consider two given strings s and t where $s = ABBC$ and $t = ACB$ and assume that costs for insertion/deletion are set to 1 while substitution cost is set to 2. We can use the dynamic programming approach shown in Equation 1 to find out that the minimum edit distance between s and t is 3. To visualize the calculation of the dynamic programming routine it is helpful to illustrate intermediate results for all recursion steps as a matrix, where each line in the matrix represents a letter of string s while each column represents a letter of string t (ϵ represents an empty string). Each cell of such a dynamic programming matrix will be set to the value of $d(i,j)$ where i is the associated row number and j is the associated column number. Figure 1 illustrates the full dynamic programming matrix for Example 1, and shows that finding the minimum edit distance corresponds to finding the shortest path through the dynamic programming matrix if one imagines a directed arc network that connects adjacent single cells of the matrix. Horizontal/vertical arcs will go in rightwards/downwards direction and have a length of one in our example, as they represent single character insertion/removal. Diagonal arcs represent single character substitution and have a length of two if the characters mismatch or a length of zero if two characters are equal. □

	ϵ	A	C	B
ϵ	0	1	2	3
A	1	0	1	2
B	2	1	2	1
B	3	2	3	2
C	4	3	2	3

(a) DP Matrix

A	$-$	B	B	C
A	C	B	$-$	$-$

(b) Alignment of Strings

Figure 1: The dynamic programming matrix for calculating the edit distance between two given strings $s = ABBC$ and $t = ACB$ is shown in Figure 1a, where each insertion/deletion causes a cost of one and each substitution causes a cost of 2. It also shows the shortest path through the dynamic programming matrix that leads to the minimum edit distance of 3 in this case. The small number next to each arrow denotes the cost for a single edit operation (a diagonal move denotes keeping a single character, a downwards move will delete a single character from string s and a rightwards move will insert a single character to string t). Figure 1b shows the alignment corresponding to the edits.

Lazy Clause Generation

In this paper we propose an explaining propagator that can be used with a lazy clause generation (LCG) solver (Ohrimenko, Stuckey, and Codish 2009). A LCG solver tracks information about the reasons for any propagated domain changes during search and stores explanations for each propagation. In case of a failure, these explanations can then be used to compute so called *nogoods*, which record the reason for the failure in form of novel constraints. These *nogoods* can then prevent the search from making similar sets of faulty decisions later.

A LCG solver furthermore uses Boolean variables to represent integer variables. For example, a variable x with a domain $D(x) = [l \ldots u]$ will be represented by Boolean variables $[\![x = d]\!], l \leq d \leq u$ and $[\![x \leq d]\!], l \leq d < u$. We use $[\![x \neq d]\!]$ to represent $\neg[\![x = d]\!]$ and $[\![x \geq d]\!]$ to represent $\neg[\![x \leq d-1]\!]$. To explain a propagation, a LCG solver will define clauses over these Boolean variables. We will provide

some examples later when we describe how to explain propagation for the constraint propagator that we propose in this paper.

Propagating Lower Bounds on the Minimum Edit Distance

In this section we propose a novel global constraint propagator that propagates lower bounds on the minimum edit distance between two strings that are represented by positive integer arrays.

Assuming two positive integer variable arrays X and Y of length n ($X = [x_1, \ldots, x_n], [y_1, \ldots, y_n]$) and a positive integer variable ed, we introduce the edit distance constraint $ED(X, Y, ed)$ that will constrain ed to any value greater or equal to the minimum edit distance between X and Y ($ed \geq d(x, y)$). In the following we refer to the domain of any variable x as $D(x)$.

The constraint ED additionally constrains all values $x \in X$ and $y \in Y$ to be in the range $0..|\Sigma|$, where a value of zero represents the end of a string and a positive value c the c^{th} character in an alphabet Σ. We specifically allow the use of zero values so that the arrays X and Y can hold any string of length $\leq n$ including the empty string. For reasons of simplicity and to avoid symmetries, we further specify that the constraint ED (separately) enforces that whenever a variable x_i is set to 0, all variables $x_j, j > i$ have to be set to 0 as well, similarly for y_i.

Example 2: Let arrays $X = [1, 1, 2, 0, 0]$ and $Y = [1, 2, 2, 1, 1]$ represent two strings AAB and $ABBAA$. The ED constraint would then propagate $ed \geq 4$ and remove any values smaller or equal to 3 in $D(ed)$, as 4 is the minimum edit distance in this example. Another example array $X = [1, 0, 1, 2, 0]$ would violate the constraint independently of the values assigned to Y and ed, since the zero values of array X are not properly aligned at the end. □

We now propose an adaption of the standard dynamic programming routine to propagate lower bounds on the edit distance between two variable arrays. The idea is to build an "optimistic" dynamic programming matrix similar to the example shown in Figure 1, where we assume the best case for variables that are unfixed (in other words we will assume a zero cost diagonal move is possible when any character still appears in both corresponding variable domains). In addition to the standard dynamic programming routine previously defined in 1, we also have to include exceptional cases for insertion and removals of empty string characters as the variable arrays X and Y may contain less than n characters. Therefore, whenever a variable domain contains the value zero which denotes an empty string character, we will assume that the insertion or removal costs of an empty character will be 0, i.e. $\gamma(\epsilon \to 0) = \gamma(0 \to \epsilon) = 0$ and $\gamma(0 \to c) = \gamma(c \to 0) = 1, \forall c \in \Sigma$. Algorithm 1 describes the detailed propagation function and Figure 2 further shows how a dynamic programming matrix can be used to calculate a lower bound on the edit distance between two integer variable arrays.

Algorithm 1: Propagate Edit Distance Lower Bound

fn *PropagateEditDistance*(X, Y, ed)
- $d = \text{CalculateDpMatrix}(X, Y)$
- $lb = d(length(X), length(Y))$
- $D(ed) = \{x | x \in D(ed) \wedge x \geq lb\}$

fn *CalculateDpMatrix*(X, Y)
- $n = length(X)$; $m = length(Y)$
- $d(0, 0) = 0$ ▷ d is a $(n + 1 \times m + 1)$ matrix
- **for** $j = 1$ *to* m **do**
  - $insCost = \min\{\gamma(\epsilon \to c) \mid c \in D(y_j)\}$
  - $d(0, j) = d(0, j - 1) + insCost$
- **for** $i = 1$ *to* n **do**
  - $remCost = \min\{\gamma(c \to \epsilon) \mid c \in D(x_i)\}$
  - $d(i, 0) = d(i - 1, 0) + remCost$
- **for** $i = 1$ *to* n; $j = 1$ *to* m **do**
  - $insCost = \min\{\gamma(\epsilon \to c) \mid c \in D(y_j)\}$
  - $remCost = \min\{\gamma(c \to \epsilon) \mid c \in D(x_i)\}$
  - $subCost = \min\{\gamma(c_x \to c_y) \mid c_x \in D(x_i) \setminus \{0\}, c_y \in D(y_j) \setminus \{0\}\}$
  - $d(i, j) = \min \begin{cases} d(i, j - 1) + insCost \\ d(i - 1, j) + remCost \\ d(i - 1, j - 1) + subCost \end{cases}$
- **return** d

Explaining Propagation

In a LCG solver we need to provide an explanation clause which the solver can use to build an inference graph. When a conflict occurs during search, the solver can then find nogood constraints that are automatically created by analyzing the inference graph. In the following we will describe how inferences made by the edit distance propagator can be represented as an explanation clause.

Whenever a lower bound lb on the ed variable is propagated, essentially what we have to achieve is to enforce the corresponding Boolean variable $[\![ed \geq lb]\!]$ to be set to true. A correct explanation $expl$ therefore consists of a set of literals so that the following proposition is valid:

$$\bigwedge_{l \in expl} l \to [\![ed \geq lb]\!] \tag{3}$$

Furthermore, an explanation is considered to be minimal, whenever it is not possible to remove any single literal l from $expl$ without invalidating Equation 3. In the following we show how a minimal explanation can be generated for inferences on the lower bound of the edit distance.

If we consider the example shown in Figure 2 we can see that in this case a lower bound of 3 has been determined for the edit distance and therefore the associated Boolean variable $[\![ed \geq 3]\!]$ would be set to true as a result of the propagation. To explain this inference, we can think about what could possibly be changed in the domains of variable arrays X and Y to allow an edit distance ≤ 2, and then negate such changes in our explanation.

If we look at the possible shortest paths in Figure 2, we

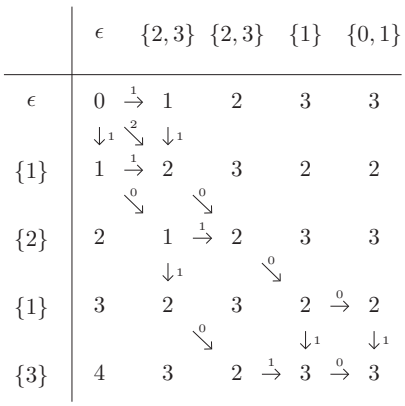

Figure 2: The dynamic programming matrix for calculating a lower bound for the edit distance between two given variable arrays $X = [x_1 = \{1\}, x_2 = \{2\}, x_3 = \{1\}, x_4 = \{3\}]$ and $Y = [y_1 = \{2,3\}, y_2 = \{2,3\}, y_3 = \{1\}, y_4 = \{0,1\}]$. Each line in the matrix represents a variable of array X while each column represents a variable of array Y (ϵ represents an empty string). In the first column/line of the matrix the domains of the corresponding variables are shown. All possible shortest paths through the matrix that lead to the lower bound for the edit distance of 3 in this case, are also shown on the figure as arrows. As not all variables are fixed, the best case (i.e. a possible match in characters, or a 0 cost insertion) is assumed several times in this example.

can observe that we need to reduce the cost of at least one of the edges towards the end of the matrix that have a cost ≥ 1. For example if x_4 and y_4 would allow the same value assignment in their domain we could take another diagonal 0 cost move and improve the lower bound to 2. More generally speaking, improvements to the edit distance can be achieved if moves are possible that reduce the length of the shortest path through the matrix.

The algorithm for generating a minimal explanation is shown in Algorithm 2. It works backwards over the matrix of (i, j) values starting from (n, m) collecting the constraints C which must hold to ensure the lower bound lb. It stores in s the minimal edit cost from a position to reach (n, m) under the current set of constraint C in the explanation. The algorithm is based on a priority queue (heap) which stores node positions that are reachable under the current assumptions. We take the (lexicographically) largest node position (i, j) off the heap, and then considers what characters c in the domain of y_j would allow a smaller lower bound via a path from $(i, j-1)$ we add a constraint $y_j \neq c$ preventing this. The remaining characters are used to update the cost s for position $(i, j-1)$ and it is pushed onto the heap. Note if a node is pushed multiple times it only appears once on the heap. Afterwards, we consider paths via $(i-1, j)$ similarly. We only consider 0 edit operations $(\gamma(\epsilon \to 0), \gamma(0 \to \epsilon))$ as long as no constraint $[\![x_i \neq 0]\!], [\![y_i \neq 0]\!]$ has been added to C since the rules of correct string representation would not allow any additional 0 operations then. Finally we consider paths via $(i-1, j-1)$. Here when a substitution $(c_x \to c_y)$ would lead to a path which is shorter than lb we have a choice we can enforce $x_i \neq c_x$ or $y_j \neq c_y$. In practice we make choices dependent on the current domains of x_i and y_j. If x_i is fixed and $D(x_i) = \{c_x\}$ then we choose the restriction on y_j and if $D(x_i) = \{c\}, c \neq c_x$ we choose the restriction on x_i. Similarly if y_j is fixed. If only one of the disequations holds in the current domain we choose that (note that its impossible that both do not hold, otherwise lb would not be the lower bound). In the remaining cases we can choose arbitrarily.

Algorithm 2: Generate disequalities for minimal explanation

fn *GenerateExpl*(X, Y, d, lb)
- $n = length(X); m = length(Y)$
- **for** $i = 0$ *to* $n; j = 0$ *to* m **do**
  - $s(i, j) = lb + 1$
- $s(n, m) = 0$; $C = \{\}$; $H = []$
- $H.push(n, m)$
- **while** H *is not empty* **do**
  - $(i, j) = H.popMax()$
  - **if** $s(i, j) \geq lb$ **then continue**
  - **if** $j - 1 \geq 0$ **then**
    - $T = \Sigma \cup \{0 \mid \neg\exists x_k \neq 0 \in C, k \geq j\}$
    - **for** $c \in T$ **do**
      - **if** $d(i, j-1) + \gamma(\epsilon \to c) + s(i, j) < lb$ **then**
        - $C.add([\![y_j \neq c]\!])$
      - **else**
        - $s(i, j-1) = \min(s(i, j-1), s(i, j) + \gamma(\epsilon \to c))$
        - $H.push(i, j-1)$
  - **if** $i - 1 \geq 0$ **then**
    - $T = \Sigma \cup \{0 \mid \neg\exists x_k \neq 0 \in C, k \geq i\}$
    - **for** $c \in T$ **do**
      - **if** $d(i-1, j) + \gamma(c \to \epsilon) + s(i, j) < lb$ **then**
        - $C.add([\![x_i \neq c]\!])$
      - **else**
        - $s(i-1, j) = \min(s(i-1, j), s(i, j) + \gamma(\epsilon \to c))$
        - $H.push(i-1, j)$
  - **if** $i - 1 \geq 0 \wedge j - 1 \geq 0$ **then**
    - **for** $c_x \in \Sigma, c_y \in \Sigma$ **do**
      - **if** $d(i-1, j-1) + \gamma(c_x \to c_y) + s(i, j) < lb$ **then**
        - $C.add([\![x_i \neq c_x]\!])$ **OR** $C.add([\![y_j \neq c_y]\!])$
      - **else**
        - $s(i-1, j-1) = \min(s(i-1, j-1), s(i, j) + \gamma(c_x \to c_y))$
        - $H.push(i-1, j-1)$
- **return** C

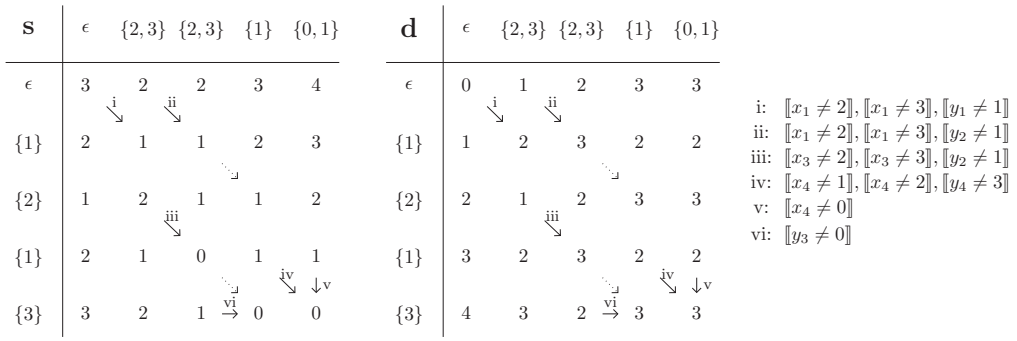

Figure 3: The matrix on the left visualizes the contents of matrix s at the end of the explanation algorithm (Algorithm 2) that is called for two variable arrays $X = [x_1 = \{1\}, x_2 = \{2\}, x_3 = \{1\}, x_4 = \{3\}]$ and $Y = [y_1 = \{2,3\}, y_2 = \{2,3\}, y_3 = \{1\}, y_4 = \{0,1\}]$, an edit distance lower-bound of 3 and the matrix d which is shown in the middle. Additionally, six sets of disequalities that are determined by the explanation algorithm are listed on the right. For each set of generated disequalities the corresponding edit operation is also visualized in the two matrices by solid arrows highlighting the operations that could lower the edit distance bound if their cost would be reduced. The dotted arrows on the other hand indicate operations that also would need to be lowered together with operations (ii or iii), to reach a reduced lower-bound. However, since the explanation algorithm aims to minimize the number of generated explanation clauses and it is sufficient to only produce disequalities for the operations that actually cause a shortest path lower than the current lower-bound in Algorithm 2, no sets of disequalities are inserted for the dotted arrows.

The result of the code is to return C such that $C \rightarrow [\![ed \geq lb]\!]$. Note that we can simplify C by replacing a set of disequations $[\![x_i \neq c]\!], 0 \leq c < l$ by $[\![x_i \geq l]\!]$ and similarly a set of disequations $[\![x_i \neq c]\!], u < c < |\Sigma|$ by $[\![x_i \leq u]\!]$.

Example 3: Consider generating the explanation for the case shown in Figure 2. We start by setting $s(i,j) = lb+1 = 4$ everywhere, then resetting $s(4,4) = 0$ and pushing (4,4). We pop off $(4,4)$. Since $d(4,3) = 3$ we cannot get a path of length ≤ 2 via it We set $s(4,3) = 0$ (for the case that $y_4 = 0$ and push $(4,3)$). Since $d(3,4) = 2$ we can get a path of length ≤ 2 via it if $x_4 = 0$ so we add $[\![x_4 \neq 0]\!]$ to C. We set $s(3,4) = 1$ since all other deletions cost 1 and push $(3,4)$. Since $d(3,3) = 2$ we can get paths of length ≤ 2 via this position if the characters for x_4 and y_4 are the same. We need to add one of each pair $[\![x_4 \neq c]\!]$ or $[\![y_4 \neq c]\!]$ for all $c \in 1..3$. We choose $[\![x_4 \neq 1]\!]$, $[\![y_4 \neq 3]\!]$, because of the values in the current domains of x_4 and y_4. The remaining choice is arbitrary: say we add $[\![x_4 \neq 2]\!]$ to C. We set $s(3,3) = 2$ and push $(3,3)$. We pop off (4,3). Since $d(3,3) = 2$ we could get a path of length ≤ 2 if $x_4 = 0$ but we already inserted a constraint of the form $[\![x_4 \neq 0]\!]$, so we do not consider 0 edit operations in that direction any longer and do not change the cost to of $s(3,3)$. Since $d(3,2) = 3$ there is no path less than lb possible, we set $s(3,2) = 0$ and push it. Since $d(4,2) = 2$ we can get a path of length ≤ 2 via it if $y_3 = 0$ so we add $[\![y_3 \neq 0]\!]$ to C and push (4,2) as other insertions all cost 1. We pop off $(4,2)$ and its treatment is similar, followed by $(4,1)$ and $(4,0)$. Next we pop $(3,4)$ and will set $s(2,4) = 2$ and push $(2,4)$ as we cannot directly improve the lb. Similarly we set $s(2,3) = 1$ and $s(3,3)$ will have its assigned value changed to 1, since we can reach (4,4) quicker via (3,4). The process continues eventually collecting $C = \{[\![x_1 \neq 2]\!], [\![x_1 \neq 3]\!], [\![x_3 \neq 2]\!], [\![x_3 \neq 3]\!], [\![x_4 \neq 0]\!], [\![x_4 \neq 1]\!], [\![x_4 \neq 2]\!], [\![y_1 \neq 1]\!], [\![y_2 \neq 1]\!], [\![y_3 \neq 0]\!], [\![y_4 \neq 3]\!]\}$. It can be simplified to $\{[\![x_1 \leq 1]\!], [\![x_3 \leq 1]\!], [\![x_4 \geq 3]\!], [\![y_1 \geq 2]\!], [\![y_2 \geq 2]\!], [\![y_3 \geq 1]\!], [\![y_4 \leq 2]\!]\}$. □

Figure 3 visualizes matrix s after generating the explanation for Example 3 and summarizes which constraints have been produced.

We can argue that the explanation produced by GenerateExpl is minimal since the only time we add constraints to C is when otherwise there would be a path to (n,m) of length less than lb. Removing any explanation would cause such a path to exist, thus making the explanation incorrect, hence it is minimal.

Experimental Evaluation

We implemented the constraint propagation and explanation algorithms proposed in this paper for use with a recent version of the lazy clause generation solver Chuffed (Chu 2011). Afterwards, we evaluated our constraint propagator on two NP-hard problems that utilize the edit distance constraint described in this paper.

All of our experiments have been conducted on an Intel Xeon E5345 2.33 GHz CPU with 48 GB RAM, using a single CPU core.

Paint Shop Scheduling

One of the problems we consider is a real life scheduling problem that we have recently described (Winter et al. 2019; Winter and Musliu 2019). The aim of this scheduling problem is to find a feasible production plan that minimizes setup times and costs. Production in the paint shop is organized in cyclic production rounds, where each round has its own scheduling sequence of different jobs that are produced within that cycle. The evaluation of setup costs for paint shop scheduling requires calculating the edit distance between two consecutive cycles, as the required change in

	CP		CP+global	
Instance	Cost	Runtime	Cost	Runtime
I1	**775***	9.95	**775***	**3.63**
I2	**842***	1.30	**842***	**0.57**
I3	**961***	3.76	**961***	**1.75**
I4	**918***	178.46	**918***	**25.05**
I5	**530***	126.64	**530***	**51.03**
I6	**842***	9.76	**842***	**5.31**
I7	1046	∞	**1040**	∞
I8	**1237***	2915.83	**1237***	**445.43**
I9	1006	∞	**992**	∞
I10	973	∞	**966**	∞
I11	—	∞	—	∞
I12	—	∞	—	∞

Table 1: Results for the experiments conducted on benchmark instances 1-12 for the paint shop scheduling problem (Winter et al. 2019). Columns 2 and 3 show the best objective value achieved within one hour as well as the runtime needed to prove an optimal solution in seconds for the Constraint Programming model from (Winter and Musliu 2019) (CP). Similarly, Columns 4 and 5 show the results achieved with the Constraint Programming model that uses the global constraint propagator proposed in this paper instead of the constraint decomposition for the edit distance constraint (CP+global). The best result within each line is formatted in bold face and results marked with a * denote proven optimal solutions. ∞ represent a time out (1 hour), while — means that no solution at all could be found within the time limit.

	CP nolearn		CP+Global nolearn	
Instance	Cost	Runtime	Cost	Runtime
I1	3282	∞	**775***	**1617.15**
I2	**842***	13.24	**842***	**0.57**
I3	**961***	519.41	**961***	**1.76**
I4	—	∞	**918***	**79.77**
I5	—	∞	—	∞
I6	17234	∞	**842***	**6.25**
I7	—	∞	—	∞
I8	—	∞	—	∞
I9	—	∞	—	∞
I10	—	∞	**973**	∞
I11	—	∞	—	∞
I12	—	∞	—	∞

Table 2: Results for the experiments conducted on benchmark instances 1-12 for the paint shop scheduling problem (Winter et al. 2019) without clause learning. Columns 2 and 3 show the best objective value achieved within one hour as well as the run-time needed to prove an optimal solution in seconds for the Constraint Programming model from (Winter and Musliu 2019) (CP nolearn). Similarly, Columns 4 and 5 show the results achieved with the Constraint Programming model that uses the global constraint propagator proposed in this paper instead of the constraint decomposition for the edit distance constraint (CP+global nolearn).

production utilities corresponds to the minimum edit operations between the two scheduling sequences. Paint shop scheduling defines the following edit operation costs: $\gamma(\epsilon \rightarrow c) = \gamma(c \rightarrow \epsilon) = 1$ forall $c \in \Sigma$ and $\gamma(c_1 \rightarrow c_2) = 2$ if $c_1 \neq c_2$ forall $\{c_1, c_2\} \subseteq \Sigma$. The full problem definition is unfortunately too extensive to include in this paper, we therefore refer the reader to (Winter et al. 2019) for further details about the problem.

For our experiments we used 12 benchmarks instances that we have previously published (Winter et al. 2019). We used the constraint programming model from (Winter and Musliu 2019) and replaced the edit distance constraint decomposition with the global constraint propagator proposed in this paper. For each of the instances we used the same programmed search strategies that have been used for the final experiments in (Winter and Musliu 2019). Afterwards, we ran the Chuffed solver with both the existing decomposition model and a model that uses the propagator proposed in this paper on all 12 instances within a time limit of 1 hour. The results of these experiments are shown in Table 1. We see in the table that the model using the global constraint produces equally good or improved results for all of the benchmark instances compared to the results produced by the previously proposed constraint decomposition. Both models are able to prove optimality for 7 of the 12 instances and can produce 3 upper-bounds within 1 hour, however all of the upper bounds produced with the global constraint propagator are improved compared to the upper bounds achieved with the decomposition. When we compare run-times for instances where both solvers could prove optimality, we can clearly see that the global propagator requires significantly less run-time to find optimal solutions. Both methods are not able to produce any solutions for the two largest instances (I11 and I12) within one hour.

To investigate the effect of the global propagator without lazy clause learning, we further repeated all experiments with the paint shop scheduling instances without clause learning (we set the *nolearn* parameter for chuffed). The results produced without the explanation algorithm are shown in Table 2. Without clause learning, the constraint propagator produced improved results compared to the constraint decomposition for four instances and could further reduce the required runtime to prove optimality for two instances. The results indicate the effectiveness of the constraint propagator even without lazy clause generation.

The Median String Problem

To evaluate our constraint propagator we also consider the median string problem, which has been thoroughly studied in the literature (e.g. (Kohonen 1985; Hayashida and Koyano 2016)).

The median string problem is formulated as follows: Given a set of n strings S (all strings of length $\leq k$) over a finite alphabet Σ, find a string that minimizes the global edit distance to each of the given strings.

The *global edit distance* $D(s, S)$ between a string s and a set of strings S over the finite alphabet Σ is defined as follows:

$$D(s, S) = \sum_{s' \in S} d(s, s') \tag{4}$$

Then the median string is defined as any string m over Σ where $D(m, S) \leq D(w, S)$ holds for any string w over the alphabet Σ.

We generated a large number of instances following the instance generation procedure that has been proposed in (Hayashida and Koyano 2016) to evaluate the performance of different exact solution approaches: Our instance generation routine considered different numbers of strings $n = [2, 4, 6, 10, 15]$ as well as different maximum string lengths $k = [3, 5, 8, 13, 20]$. We used a simple alphabet consisting of 4 different characters $\Sigma = \{1, 2, 3, 4\}$ to randomly generate 10 different instances for each of the possible $|n \times k|$ configurations, totaling in 250 benchmark instances. To randomly generate 10 different instances per configuration we implemented two procedures: Five of the instances are generated by randomly assigning the letters s^i_j at positions $j \in \{1, \ldots, k\}$ for each string $i \in \{1, \ldots, n\}$. Each letter is assigned to $s^i_j = \min(1 + \lfloor |\alpha| \rfloor, |\Sigma|)$, where α follows a normal distribution with mean 0 and variance 1. The other five instances of each configuration are generated by performing 100 random single character edit operations on an initial string of length k that initially contains only the first letter of the alphabet Σ. In each of the 100 edit iterations we randomly select a feasible single character insertion, single character removal or single character substitution.

We compare the performance of our edit distance constraint propagator (CP+global) on the median string problem with an existing mixed integer programming (MIP) formulation from (Hayashida and Koyano 2016) as well as a Constraint Programming (CP) model that uses the same edit distance constraint decomposition as for the paint shop scheduling to solve the median string problem. In our experiments we use the edit operations costs that are known as *levensthein distance*, as these costs have often been used for median string problems e.g. (Hayashida and Koyano 2016): $\gamma(\epsilon \to c) = \gamma(c \to \epsilon) = 1$ forall $c \in \Sigma$ and $\gamma(c_1 \to c_2) = 1$ if $c_1 \neq c_2$ forall $\{c_1, c_2\} \subseteq \Sigma$.

We performed experiments with all of the 250 benchmark instances under a time limit of 10 minutes, using a recent version of the Chuffed solver (Chu 2011) for the CP model and the CP model that uses our propagator, as well as Gurobi 8.0.1 (Gurobi Optimization 2019) for experiments with the MIP model that was previously proposed in (Hayashida and Koyano 2016).

Table 3 and Figures 4 and 5 summarize the results of our experiments with the median string benchmarks.

Looking at the results shown in Table 3, we can see that the CP model that uses the global edit distance propagator produces the largest number of best found solutions as well as the largest number of optimal solutions found. Furthermore, the approach can prove optimality faster than the MIP model and the CP model without the global propagator for all instances that can be solved to optimality. When comparing the decomposition based CP model with the MIP model, the results show that the MIP model requires less runtime to prove optimality and can find better solutions for a larger number of instances.

	MIP	CP	CP+global
# opt	208	180	**227**
# proven opt	197	169	**227**
# best	213	180	**246**
# fastest proof	0	0	**227**
avg time	143.16	219.12	**61.61**
std dev time	243.37	277.52	**175.66**

Table 3: Summarized experimental results for the median string problem. Column 2 displays results achieved with the MIP formulation from (Hayashida and Koyano 2016), while column 3 displays results achieved with the edit distance constraint decomposition from (Winter and Musliu 2019). Column 4 shows results for the global constraint propagator proposed in this paper. Line 1 shows the number of optimal solutions found, line 2 shows for how many instances optimality could be proven within the time limit, and line 3 shows the number of solutions that have the overall best found objective value. Line 4 displays for how many instances the method could provide the fastest optimality proof and lines 5 and 6 show the average required runtime, as well as the standard deviation of all required runtimes in the experiments.

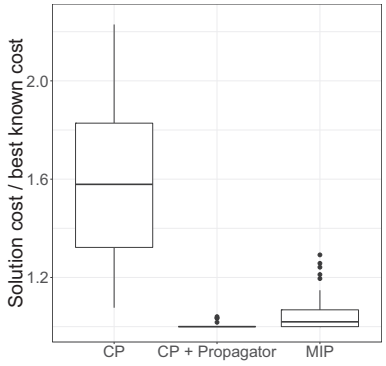

Figure 4: Box plot displaying the differences in quality of solutions for median string experiments, omitting 70 instances where all three approaches gave equal results. The vertical axis represents the relative objective value (objective value of solution divided by best found objective). Except for three outliers, the best solution cost was produced with the global propagator. Some outliers with values higher than 2.5 for the CP approach without the propagator have been omitted for a better visualization.

Figure 4 compares the quality of solutions where not all of the three considered approaches produced an equal result within 10 minutes. The results show, that except for a few outliers the approach using the constraint propagator always

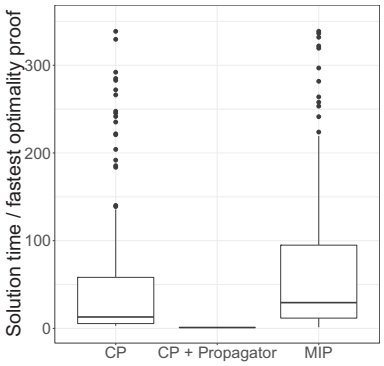

Figure 5: Box plot comparing the run-times for the 178 instances where all methods proved optimality. The vertical axis represents the required run-time divided by the overall shortest optimality proof time for each instance. The fastest optimality proof was in all cases achieved with the CP model that uses the constraint propagator proposed in this paper. Some outliers with values higher than 300 for the CP approach without the propagator and the MIP approach have been omitted for a better visualization.

produced the best results, while the MIP model seems to overall produce better results regarding solution quality than the CP model which uses a constraint decomposition.

Figure 5 compares the run-time required to prove optimality for those instances where all three approaches could prove optimality within 10 minutes. The box plots show that the approach using the propagator always proved optimality within the shortest run-time in our experiments. Furthermore, it seems that the decomposition based CP model performs similarly to the MIP model when it comes to proving optimality, although the majority of instances have a shorter run-time with the CP model.

Conclusion

In this paper, we propose a novel edit distance constraint propagator which can be used to model constraints appearing in NP-hard problems that limit the edit distance between arrays of decision variables. We define algorithms to perform constraint propagation and to explain propagated inferences for the edit distance constraint, thus allowing its use in powerful lazy clause generation solvers, that define the state of the art for many NP-hard combinatorial optimization problems. Our experimental results show that a lazy clause generation based solver that uses an edit distance constraint propagator is able to outperform state of the art MIP and CP techniques for these problems in terms of solution quality for the majority of the considered benchmark instances, and can significantly reduce the required run-time for proving optimal solutions.

In future work we will extend the propagator to use an upper bound on edit distance ed to reduce the domains of string variables x_i, y_j.

Acknowledgments The financial support by the Austrian Federal Ministry for Digital and Economic Affairs and the National Foundation for Research, Technology and Development is gratefully acknowledged. Special thanks go to Georg Faustmann and Jakob Preininger who helped discovering the appearance of string edit distance constraints in paint shop scheduling problems.

References

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

Gurobi Optimization, L. 2019. Gurobi optimizer reference manual.

Hayashida, M., and Koyano, H. 2016. Finding median and center strings for a probability distribution on a set of strings under levenshtein distance based on integer linear programming. In *BIOSTEC (Selected Papers)*, volume 690 of *Communications in Computer and Information Science*, 108–121. Springer.

Jiang, X.; Abegglen, K.; Bunke, H.; and Csirik, J. 2003. Dynamic computation of generalised median strings. *Pattern Anal. Appl.* 6(3):185–193.

Kohonen, T. 1985. Median strings. *Pattern Recognition Letters* 3(5):309–313.

Kruskal, J. B. 1983. An overview of sequence comparison: Time warps, string edits, and macromolecules. *SIAM Review* 25(2):201–237.

Navarro, G. 2001. A guided tour to approximate string matching. *ACM Computing Surveys* 33(1):31–88.

Nicolas, F., and Rivals, E. 2003. Complexities of the centre and median string problems. In *CPM*, volume 2676 of *Lecture Notes in Computer Science*, 315–327. Springer.

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

Olivares-Rodríguez, C., and Oncina, J. 2008. A stochastic approach to median string computation. In *SSPR/SPR*, volume 5342 of *Lecture Notes in Computer Science*, 431–440. Springer.

Ukkonen, E. 1985. Algorithms for approximate string matching. *Information and Control* 64(1-3):100–118.

Wagner, R. A., and Fischer, M. J. 1974. The string-to-string correction problem. *J. ACM* 21(1):168–173.

Winter, F., and Musliu, N. 2019. Constraint based modeling for scheduling paint shops in the automotive supply industry. Technical report, TU Wien, CD-TR, 2019/1.

Winter, F.; Musliu, N.; Demirovic, E.; and Mrkvicka, C. 2019. Solution approaches for an automotive paint shop scheduling problem. In *ICAPS*, 573–581. AAAI Press.