Theoretical Computer Science 660 (2017) 57–74

Contents lists available at ScienceDirect

Theoretical Computer Science

www.elsevier.com/locate/tcs

On the parameterized complexity of associative and commutative unification ☆

Tatsuya Akutsu [a], Jesper Jansson [a], Atsuhiro Takasu [b], Takeyuki Tamura [a,*]

[a] Bioinformatics Center, Institute for Chemical Research, Kyoto University, Gokasho, Uji, Kyoto, 611-0011, Japan
[b] National Institute of Informatics, 2-1-2 Hitotsubashi, Chiyoda-ku, Tokyo 101-8430, Japan

ARTICLE INFO

Article history:
Received 1 March 2016
Received in revised form 13 September 2016
Accepted 22 November 2016
Available online 28 November 2016
Communicated by A. Kucera

Keywords:
Unification
Parameterized algorithm
Dynamic programming
Tree edit distance

ABSTRACT

This article studies the parameterized complexity of the unification problem with associative, commutative, or associative-commutative functions with respect to the parameter "number of variables". It is shown that if every variable occurs only once then both of the associative and associative-commutative unification problems can be solved in polynomial time, but that in the general case, both problems are $W[1]$-hard even when one of the two input terms is variable-free. For commutative unification, an algorithm whose time complexity depends exponentially on the number of variables is presented; moreover, if a certain conjecture is true then the special case where one input term is variable-free belongs to FPT. Some related results are also derived for a natural generalization of the classic string and tree edit distance problems that allows variables.

1. Introduction

Unification is a useful concept in many areas of computer science such as automated theorem proving, program verification, natural language processing, logic programming, and database query systems [5,17,19,20]. In its fundamental form, the unification problem is to find a substitution for all variables in two given *terms* that makes the terms identical, where terms are constructed from function symbols, variables, and constants [20]. As an example, the two terms $f(x, y)$ and $f(g(a), f(b, x))$, where f and g are functions, x and y are variables, and a and b are constants, become identical by substituting x by $g(a)$ and y by $f(b, g(a))$.

Unification has a long history beginning with the seminal work of Herbrand in 1930 (see, e.g., [20]). It is becoming an active research area again because of *math search*, an information retrieval (IR) task where the objective is to find all documents containing a specified mathematical formula and/or all formulas similar to a query formula [18,23,24]; for example, math search has been adopted as a pilot task in the IR evaluation conference NTCIR [24]. Also, math search systems such as Wolfram Formula Search and Wikipedia Formula Search have been developed. Since mathematical formulas are typically represented by rooted trees, it may seem reasonable to measure the similarity between formulas simply by measuring the

☆ A preliminary version of this article appeared in *Proceedings of the 9th International Symposium on Parameterized and Exact Computation* (IPEC 2014), volume 8894 of *Lecture Notes in Computer Science*, pp. 15–27, Springer International Publishing Switzerland, 2014.

* Corresponding author.
E-mail addresses: takutsu@kuicr.kyoto-u.ac.jp (T. Akutsu), jj@kuicr.kyoto-u.ac.jp (J. Jansson), takasu@nii.ac.jp (A. Takasu), tamura@kuicr.kyoto-u.ac.jp (T. Tamura).
URLs: http://www.bic.kyoto-u.ac.jp/takutsu/members/takutsu/index.html (T. Akutsu), http://sunflower.kuicr.kyoto-u.ac.jp/~jj/ (J. Jansson), http://www.nii.ac.jp/en/faculty/digital_content/takasu_atsuhiro/ (A. Takasu), http://sunflower.kuicr.kyoto-u.ac.jp/~tamura/index.html.en (T. Tamura).

http://dx.doi.org/10.1016/j.tcs.2016.11.026
0304-3975/

structural similarity of their trees. However, methods directly based on approximate tree matching such as the *tree edit distance* (see, e.g., the survey in [7]) alone are not sufficient if every label is treated as a constant. For example, under unit cost edit operations, the query $x^2 + x$ has the same tree edit distance to each of the formulas $y^2 + z$ and $y^2 + y$, although $y^2 + y$ is mathematically the same as $x^2 + x$ while $y^2 + z$ is not.

Because of the practical importance of the unification problem and its variants, heuristic algorithms have been proposed, some of which incorporate approximate tree matching techniques [15,17]. On the negative side, their worst-case performance may be poor. To make the study of the computational complexity of unification more formal, this article examines some natural variants of the unification problem from the viewpoint of parameterized complexity and presents several new algorithms for them. The parameterized complexity is considered with respect to the parameter "number of variables appearing in the input"; we choose this parameter because the number of variables is often much smaller than the length of the terms.

1.1. Related work

An exponential-time algorithm for the unification problem was given in [26]. In the twenty years that followed, numerous faster and more practical algorithms were published (see [20] for a comprehensive survey), and in particular, a linear-time algorithm for the problem was developed [11,25].

Various extensions of unification have also been considered [5,6,19,20]. Three such extensions are unification with associative, commutative, and associative-commutative functions (where a function f is called *associative* if $f(x, f(y, z)) = f(f(x, y), z)$ always holds, *commutative* if $f(x, y) = f(y, x)$ always holds, and *associative-commutative* if it is both associative and commutative). These are especially relevant for math search since many functions encountered in practice have one of these properties. Interestingly, when allowing such functions, there are more ways to map nodes in the two corresponding trees to each other, and as a result, the computational complexity of unification may *increase*. Indeed, each of the associative, commutative, and associative-commutative unification problems is NP-hard [6,12].

The special case of unification where one of the two input terms contains no variables is also known as *matching*. Unfortunately, all of the associative, commutative, and associative-commutative matching problems remain NP-hard [6,12], and polynomial-time algorithms are known only for very restricted cases [2,6,19]. E.g., associative-commutative matching can be done in polynomial time if every variable occurs exactly once [6].

We remark that associative unification is in PSPACE and both commutative unification and associative-commutative unification are in NP (see, e.g., the references in Section 3.4 in [5]). Although this means that all three problems can be solved in single exponential-time in the size of the input, it does not necessarily mean single exponential-time algorithms with respect to the number of variables.

For an introduction to parameterized complexity, the reader is referred to the textbook [14]. When a problem is proved to be W[1]-hard or W[2]-hard, it is strongly believed that developing an FPT algorithm is impossible. To prove W[i]-hardness, we often use reduction from Longest Common Subsequence (LCS). LCS is, given a set of strings $R = \{r_1, r_2, \ldots, r_q\}$ over an alphabet Σ_0 and an integer l, to determine whether there exists a string r of length l such that r is a subsequence of r_i for every $r_i \in R$, where r is called a *subsequence* of r' if r can be obtained by performing deletion operations on r'. LCS is $W[1]$-hard with respect to the parameter (q, l) (this problem variant is called "LCS-3" in [8]). On the other hand, LCS is $W[2]$-hard with respect to the parameter l (this problem variant is called "LCS-2" in [8]). The definitions of FPT and W[i] by [8] are given in Appendix A.

Other previous work on associative and/or commutative unification has focused on central aspects such as termination, soundness, and completeness of algorithms (see, for example, [28]) as well as implementations [27]. In computational experiments done in the 1980s, associative-commutative unification was not efficiently computable in general [9], but computable for DO-terms for very small instances [21].

1.2. Summary of new results

We present a number of new results on the parameterized complexity of associative, commutative, and associative-commutative unification with respect to the parameter "number of variables appearing in the input", denoted from here on by k. See Table 1 for an overview. Most notably:

- Both associative and associative-commutative matching are $W[1]$-hard.
- Commutative matching can be done in $O(2^k poly(m, n))$ time, where m and n are the sizes of the two input terms, if a certain conjecture is true.
- Both associative and associative-commutative unification can be done in polynomial time if every variable occurs exactly once.
- Commutative unification can be done in polynomial time if the number of variables is bounded by a constant.

In addition, we show that generalizing the classic string and tree edit distance problems [7,16] to also allow variables yields $W[1]$-hard problems.

Table 1
Summary of the new results in this article. SEDV = the string edit distance problem with variables, OTEDV = the ordered tree edit distance problem with variables, and DO = distinct occurrences of all variables. $W[1]$-hard and FPT mean with respect to the parameter k. The result marked by * is true if Conjecture 1 in Section 4.2 holds.

	matching	unification	DO-matching	DO-unification
Associative	$W[1]$-hard (Theorem 1) NP-complete (Ref. [6])	–	P (Ref. [6])	P (Theorem 2)
Commutative	NP-hard (Ref. [6]) FPT* (Theorem 3)	XP (Theorem 5)	P (Ref. [6])	P (Theorem 4)
Associative-commutative	$W[1]$-hard (Theorem 6) NP-hard (Ref. [6])	–	P (Ref. [6])	P (Proposition 4)
SEDV	$W[2]$-hard (Theorem 7)	$W[2]$-hard/$W[1]$-hard (Theorem 7/Theorem 8) $O(\|\Sigma\|^k poly)$ (Proposition 5)	–	P (Theorem 11)
OTEDV	–	$W[1]$-hard (Theorem 10)	–	P (Theorem 11)

To simplify the presentation, the algorithms described in this article only determine if two terms are unifiable, but they may be modified to output the corresponding substitutions (when unifiable) by using standard traceback techniques. We remark that it is not possible to always output a most general unifier (mgu) [22] because there does not necessarily exist an mgu for either associative unification or commutative unification. For example, in commutative unification, $f(x, y)$ and $f(a, b)$ have two unifiers $\theta = \{x/a, y/b\}$ and $\theta = \{x/b, y/a\}$ and thus no mgu, and in associative unification, $f(x, y)$ and $f(a, f(b, c))$ have two unifiers $\theta = \{x/a, y/f(b, c)\}$ and $\theta = \{x/f(a, b), y/c\}$ and thus no mgu.

2. Basic definitions

Suppose that Σ is a set of function symbols, where each function symbol has an associated *arity*, which is an integer describing how many arguments the function takes, and that Γ is a set of variables. No function symbol is allowed to be a variable, i.e., $\Sigma \cap \Gamma = \emptyset$. A function symbol with arity 0 is called a *constant*. A *term* over $\Sigma \cup \Gamma$ is defined recursively as:

1. A constant is a term,
2. A variable is a term,
3. If $t_1, \cdots, t_d$ are terms and f is a function symbol with arity $d > 0$ then $f(t_1, \ldots, t_d)$ is a term.

Every term is identified with a rooted, ordered, node-labeled tree in which every internal node is labeled by a function symbol with nonzero arity and every leaf is labeled by either a constant or a variable. The tree identified with a term t is also denoted by t. For any term t, $N(t)$ is the set of all nodes in its tree t, $r(t)$ is the root of t, and $\gamma(t)$ is the function symbol of $r(t)$. The *size* of t is defined as $|N(t)|$. For any $u \in N(t)$, t_u denotes the subtree of t rooted at u and hence corresponds to a subterm of t. Any variable that occurs only once in a term is called a *DO-variable*, where "DO" stands for "distinct occurrences", and a term in which all variables are DO-variables is called a *DO-term* [6]. A term that consists entirely of elements from Σ is called *variable-free* or a *ground term*.

Let $\mathcal{T}$ be a set of terms over $\Sigma \cup \Gamma$. A *substitution* θ is defined as any partial mapping from Γ to $\mathcal{T}$ (where we let x/t indicate that the variable x is mapped to the term t), under the constraint that if $x/t \in \theta$ then t is not allowed to contain the variable x. For any term $t \in \mathcal{T}$ and substitution θ, $t\theta$ is the term obtained by simultaneously replacing its variables in accordance with θ. For example, $\theta = \{x/y, y/x\}$ is a valid substitution, and in this case, $f(x, y)\theta = f(y, x)$. Two terms $t_1, t_2 \in \mathcal{T}$ are said to be *unifiable* if there exists a θ such that $t_1\theta = t_2\theta$, and such a θ is called a *unifier*.

Example 1. Let $\Sigma = \{a, b, f, g\}$, where a and b are constants, f has arity 2, and g has arity 3, and let $\Gamma = \{w, x, y, z\}$. Define the terms $t_1 = f(g(a, b, a), f(x, x))$, $t_2 = f(g(y, b, y), z)$, and $t_3 = f(g(a, b, a), f(w, f(w, w)))$. Then t_1 and t_2 are unifiable since $t_1\theta = t_2\theta = f(g(a, b, a), f(x, x))$ holds for $\theta = \{y/a, z/f(x, x)\}$. Similarly, t_2 and t_3 are unifiable since $t_2\theta' = t_3\theta' = f(g(a, b, a), f(w, f(w, w)))$ with $\theta' = \{y/a, z/f(w, f(w, w))\}$. However, t_1 and t_3 are not unifiable because it is impossible to simultaneously satisfy $x = w$ and $x = f(w, w)$. □

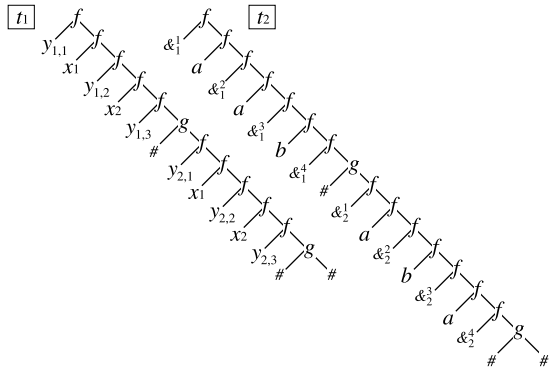

Fig. 1. Illustrating the reduction in Theorem 1. Here, $r_1 = aab$, $r_2 = aba$, and $l = 2$. The two terms t_1 and t_2 can be matched since there exists a unifier of the form $\theta = \{x_1/a,\ x_2/b,\ y_{1,1}/\&_1^1,\ y_{1,2}/f(f(\&_1^2, a), \&_1^3),\ y_{1,3}/\&_1^4, \ldots\}$, corresponding to the common subsequence ab of length 2.

In this paper, the *unification problem* is to determine whether two input terms t_1 and t_2 are unifiable. (Other versions of the unification problem have also been studied in the literature, but will not be considered here.) Unless otherwise stated, m and n denote the sizes of the two input terms t_1 and t_2. The unification problem can be solved in linear time [11, 25]. The important special case of the unification problem where one of the two input terms is variable-free is called the *matching problem*. Another important special case of the unification problem is *unification for DO-terms*, where both terms are DO-terms and no variable occurs in both of them.

Throughout the paper, k refers to the number of variables occurring in the input.

3. Associative unification

A function f with arity 2 is called *associative* if $f(x, f(y, z)) = f(f(x, y), z)$ always holds. Associative unification is a variant of unification in which functions may be associative. In this section, we assume that all functions are associative. However all results are valid by appropriately modifying the details even if usual (non-associative) functions are included.

3.1. Hardness

Associative matching was shown to be NP-hard in [6] by a simple reduction from 3SAT. However, the proof in [6] does not show the parameterized hardness. We therefore present the following new result.

Theorem 1. *Associative matching is $W[1]$-hard with respect to the number of variables even for a fixed Σ.*

Proof. We present an FPT-reduction [14] from the longest common subsequence problem (LCS, explained in Section 1.1) to associative matching.

First consider the case of an unrestricted Σ. Let $(\{r_1, \ldots, r_q\}, l)$ be any given instance of LCS. For each $i = 1, \ldots, q$, create a term u^i as follows: $u^i = f(y_{i,1}, f(x_1, f(y_{i,2}, f(x_2, \cdots f(y_{i,l}, f(x_l, f(y_{i,l+1}, g(\#, \#))) \cdots))))$, where # is a character not appearing in $r_1, \ldots, r_q$. Create a term t_1 by replacing the last occurrence of # in each u^i by u^{i+1} for $i = 1, \ldots, q-1$, thus concatenating $u^1, \ldots, u^q$, as shown in Fig. 1. Next, transform each r_i into a string r'_i of length $1 + 2 \cdot |r_i|$ by inserting a special character $\&_i^j$ in front of the jth in r_i, and appending $\&_i^{|r_i|+1}$ to the end of r_i, where each $\&_i^j$ is considered to be a distinct constant, meaning that $\&_i^j$ does not match any symbol (in particular, $\&_i^j \neq \&_{i'}^{j'}$ holds for all $i \neq i'$ and for $j \neq j'$) but can match any variable. Represent each r'_i by a term t^i defined by: $t^i = f(r'_i[1], f(r'_i[2], f(r'_i[3], f(\cdots, f(r'_i[1 + 2 \cdot |r'_i|], g(\#, \#)) \cdots))))$. Finally, create a term t_2 by concatenating $t^1, \ldots, t^q$. (Again, see Fig. 1.) Now, t_1 and t_2 can be matched if and only if there exists a common subsequence of $\{r_1, \ldots, r_q\}$ of length l. Since the number of variables in t_1 is $(l+1)q + l = lq + l + q$, it is an FPT-reduction and thus the problem is $W[1]$-hard.

For the case of a fixed Σ, represent each constant by a distinct term using a special function symbol h and binary encoding (e.g., the 10th symbol among 16 symbols can be represented as $h(1, h(0, h(1, 0)))$). □

3.2. Algorithms

For any term t, define the *canonical form* of t (called the "flattened form" in [6]) as the term obtained by contracting all edges in t whose two endpoints are labeled by the same function symbol. For example, the canonical form of $f(g(a, b), f(c, d))$ is $f(g(a, b), c, d)$. It is easy to see that the canonical form of t can be computed in linear time [6].

We begin with the simplest case in which every term is variable-free.

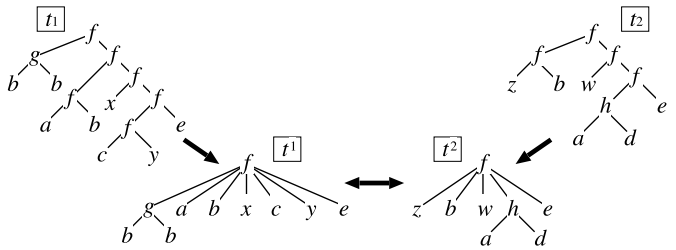

Fig. 2. An example of associative unification. The DO-terms t_1, t_2 are transformed into their canonical forms and then unified by $\theta = \{y/h(a,d),\ z/f(g(b,b),a),\ w/f(x,c)\}$.

Proposition 1. *Associative unification for variable-free terms can be done in linear time.*

Proof. Transform the two terms into their canonical forms in linear time as above. Then it suffices to test if the canonical forms are isomorphic. The rooted ordered labeled tree isomorphism problem is trivially solvable in linear time by traversing the trees [10]. □

We next consider associative unification for DO-terms, which has some similarities to DO-associative-commutative matching [6]. To handle the more general case of two DO-terms t_1 and t_2, we transform them into their canonical forms t^1 and t^2 and apply the following procedure, which returns 'true' if and only if t^1 and t^2 are unifiable. See Fig. 2 for an illustration. The procedure considers all pairs $u \in N(t^1)$, $v \in N(t^2)$ in bottom-up order, and assigns $D[u,v] = 1$ if and only if $(t^1)_u$ and $(t^2)_v$ are unifiable.

```
Procedure AssocUnifDO(t^1, t^2)
  for all u ∈ N(t^1) do                    /* in post-order */
    for all v ∈ N(t^2) do                  /* in post-order */
      if (t^1)_u or (t^2)_v is a variable then                    (#1)
        D[u, v] ← 1;
      else if (t^1)_u or (t^2)_v is a constant then               (#2)
        if (t^1)_u and (t^2)_v are the same constant
        then D[u, v] ← 1 else D[u, v] ← 0;
      else   (#3) /* (t^1)_u = f_1((t^1)_{u_1}, ..., (t^1)_{u_p}), (t^2)_v = f_2((t^2)_{v_1}, ..., (t^2)_{v_q}) */
        if f_1 = f_2 and ⟨(t^1)_{u_1}, ..., (t^1)_{u_p}⟩ and ⟨(t^2)_{v_1}, ..., (t^2)_{v_q}⟩ are compatible
        then D[u, v] ← 1 else D[u, v] ← 0;
  if D[r(t^1), r(t^2)] = 1 then return true else return false.
```

Algorithm 1. AssocUnifDO.

When both u and v are internal nodes, we need to check if $\langle (t^1)_{u_1}, \ldots, (t^1)_{u_p} \rangle$ and $\langle (t^2)_{v_1}, \ldots, (t^2)_{v_q} \rangle$ are *compatible*, where compatible implicitly means unifiable. This compatibility is defined and efficiently tested by regarding the two sequences of nodes as two strings and applying string matching with variable-length don't-care symbols [4], while setting the difference to 0 and allowing don't-care symbols in both strings. Until the end of this section, *matching* for strings means this kind of string matching (not a special case of unification).

Let $s_a = a_1 \ldots a_p$ and $s_b = b_1 \ldots b_q$ be two strings including variable-length don't-care symbols '*'. Intuitively, '*' in s_a (resp., in s_b) can match any non-empty substring of s_b (resp., s_a), where the substring can include '*'. In addition, matching between constants (i.e., symbols except '*') is not necessarily symmetric and we can define any $M \subseteq (\{a_1, \ldots, a_p\} - \{*\}) \times (\{b_1, \ldots, b_q\} - \{*\})$ as a set of allowed matching constant pairs. Then, we say that s_a matches s_b (and also s_b matches s_a) if there exists a set of pairs of indices $L \subseteq \{1, \ldots, p\} \times \{1, \ldots, q\}$ satisfying the following conditions:

- for all $a_i \neq *$, $|\{j|(i,j) \in L\}| = 1$,
- for all $b_j \neq *$, $|\{i|(i,j) \in L\}| = 1$,
- for all $a_i = *$, $\{j|(i,j) \in L\}$ is a non-empty set of consecutive numbers,
- for all $b_j = *$, $\{i|(i,j) \in L\}$ is a non-empty set of consecutive numbers,
- for all $(i,j) \in L$, $|\{k|(i,k) \in L\}| = 1$ or $|\{k|(k,j) \in L\}| = 1$ holds,
- there are no pairs $(i_1, j_1), (i_2, j_2) \in L$ such that $(i_2 - i_1) \cdot (j_2 - j_1) < 0$.

The first and second lines mean that each constant symbol can match exactly one symbol (constant or variable). The third and fourth lines mean that each variable symbol can match a consecutive substring in the other string. The fifth line means that there is no overlap among matching pairs. The last line means that matching must be order-preserving. It is to be noted that *match* here means that for the whole parts of two input strings. For example, $ab*d$ can match $a*d$ by $L = \{(1,1),(2,2),(3,2),(4,3)\}$, where $M = \{(a,a),(b,b),(c,c),(d,d)\}$. Further, we can see that $ab*c*d$ can match $a*d$, but $ab*c$ cannot match $a*d$ or $ac*c$.

The following pseudocode decides whether or not s_a matches s_b by successively assigning $E[i, j] = 1$ if and only if the prefix $a_1 \dots a_i$ matches the prefix $b_1 \dots b_j$ for increasing values of i and j. We define and examine whether $(t^1)_u = f((t^1)_{u_1}, \dots, (t^1)_{u_p})$ and $(t^2)_v = f((t^2)_{v_1}, \dots, (t^2)_{v_q})$ are compatible by constructing the strings $s_a = (t^1)_{u_1} \dots (t^1)_{u_p}$ and $s_b = (t^2)_{v_1} \dots (t^2)_{v_q}$ where each variable is replaced by '*', defining a matching relation by $M = \{((t^1)_{u_i}, (t^2)_{v_j}) | (t^1)_{u_i} \neq *, (t^2)_{v_j} \neq *, (t^1)_{u_i}$ and $(t^2)_{v_j}$ are unifiable$\}$, and testing whether s_a matches s_b.

```
Procedure StrMatchVDC(s_a, s_b)
  for all i, j ∈ {0, ..., p} × {0, ..., q} do E[i, j] ← 0;
  E[0, 0] ← 1;
  for i = 1 to p do
    for j = 1 to q do
      if a_i = * and b_j = * then    (#1)
        E[i, j] ← max{max_{i'<i} {E[i', j − 1]}, max_{j'<j} {E[i − 1, j']}}
      else if a_i = * then E[i, j] ← max_{j'<j} {E[i − 1, j']}    (#2)
      else if b_j = * then E[i, j] ← max_{i'<i} {E[i', j − 1]}    (#3)
      else if a_i matches b_j then E[i, j] ← E[i − 1, j − 1];    (#4)
  if E[p, q] = 1 then return true else return false.
```

Algorithm 2. StrMatchVDC.

Note that there exists a case in which one variable partially matches two variables in associative unification. For example, consider two terms $f(t_1, x, t_2)$ and $f(y, z)$. Here, $\{x/f(t_3, t_4), y/f(t_1, t_3), z/f(t_4, t_2)\}$ is a unifier. However, we can have a simpler unifier $\{x/t_3, y/f(t_1, t_3), z/t_2\}$ because each variable occurs only once. Therefore, we can use string matching with variable-length don't-care symbols. This issue is discussed in the proof of Theorem 2 and referred to as "overlap removal".

Lemma 1. *String matching with variable-length don't-care symbols can be solved in $O(pq(p + q))$ time with Algorithm 2.*

Proof. First we prove the soundness and the completeness of the algorithm by showing that $E[i, j] = 1$ holds if and only if $a_1 \dots a_i$ matches $b_1 \dots b_j$, using mathematical induction on i and j.

As the base step, we note that $E[i, 0] = E[0, j] = 0$ holds for any $i = 1, \dots, m$ and $j = 1, \dots, n$ because the **for**-loops begin from $i = 1$ and $j = 1$. Furthermore, we can see the following from $E[0, 0] = 1$ and the corresponding lines:

- if $a_1 = *$ and $b_1 = *$, $E[1, 1] = 1$ holds from part (#1),
- if $a_1 = *$ and $b_1 \neq *$, $E[1, 1] = 1$ holds from part (#2),
- if $a_1 \neq *$ and $b_1 = *$, $E[1, 1] = 1$ holds from part (#3),
- if $a_1 \neq *$, $b_1 \neq *$, and $a_1 = b_1$, $E[1, 1] = 1$ holds from part (#4),
- otherwise $E[1, 1] = 0$ holds because there is no change on $E[1, 1]$ after the initialization.

Since all the cases of $i = 1$ or $j = 1$ are covered, this proves the claim for the cases of $i \leq 1$ or $j \leq 1$.

As the induction step, we assume that the claim holds for $i = 0, \dots, I - 1$ and $j = 0, \dots, J - 1$, and consider the case of $i = I$ and $j = J$, where $I > 0$, $J > 0$, and $I + J > 2$. Since '*' can match any string (including '*'s) with length greater than 0, $a_1 \dots a_I$ can match $b_1 \dots b_J$ if and only if one of the following holds:

(i) $a_I = *$, $b_J = *$, and $b_1 \dots b_{J-1}$ matches $a_1 \dots a_h$ for some $h \in \{1, 2, \dots, I - 1\}$,
(ii) $a_I = *$, $b_J = *$, and $a_1 \dots a_{I-1}$ matches $b_1 \dots b_h$ for some $h \in \{1, 2, \dots, J - 1\}$,
(iii) $a_I = *$, $b_J \neq *$, and $a_1 \dots a_{I-1}$ matches $b_1 \dots b_h$ for some $h \in \{1, 2, \dots, J - 1\}$,
(iv) $a_I \neq *$, $b_J = *$, and $b_1 \dots b_{J-1}$ matches $a_1 \dots a_h$ for some $h \in \{1, 2, \dots, I - 1\}$,
(v) $a_I \neq *$, $b_J \neq *$, $a_I = b_J$, and $a_1 \dots a_{I-1}$ matches $b_1 \dots b_{J-1}$.

In cases (i) and (ii), $E[I, J] = 1$ holds from part (#1), In case (iii), $E[I, J] = 1$ holds from part (#2), In case (iv), $E[I, J] = 1$ holds from part (#3), In case (v), $E[I, J] = 1$ holds from part (#4). Otherwise, $E[I, J] = 0$ holds. Since all the cases of $i = I$ and $j = J$ are covered, this proves the claim for the case of $i = I$ and $j = J$. Therefore, the algorithm is sound and complete.

Next, we analyze the time complexity. The most time-consuming part is clearly (#1), which takes $O(p + q)$ time. Since (#1) is repeated $O(pq)$ times, the total time complexity is $O(pq(p + q))$. □

Theorem 2. *Associative unification for DO-terms can be done in $O(m^2 n^2 (m + n))$ time.*

Proof. First we prove the soundness and the completeness of Algorithm 1. For that purpose, it is enough to prove that $D[u, v] = 1$ holds if and only if $(t^1)_u$ and $(t^2)_v$ are unifiable, using mathematical induction of the size of $(t^1)_u$ and $(t^2)_v$.

As the base step, we assume that either $(t^1)_u$ or $(t^2)_v$ is a variable or a constant. Then, $(t^1)_u$ and $(t^2)_v$ are unifiable if and only if one of the following holds:

(i) $(t^1)_u$ is a variable,
(ii) $(t^2)_v$ is a variable,
(iii) $(t^1)_u$ and $(t^2)_v$ are the same constant,

because each variable occurs at most once and thus can match any term (including variables). In cases (i) and (ii), $D[u, v] = 1$ holds from part (#1). In case (iii) $D[u, v] = 1$ holds from part (#2). Otherwise, $D[u, v] = 0$ holds. This proves the claim for the base step.

As the induction step, we assume that $(t^1)_u = f_1((t^1)_{u_1}, \ldots, (t^1)_{u_p})$, $(t^2)_v = f_2((t^2)_{v_1}, \ldots, (t^2)_{v_q})$, and $D[u_i, v_j] = 1$ holds if and only if $t^1_{u_i}$ and $t^2_{v_j}$ are unifiable. Since $(t^1)_u$ and $(t^2)_v$ are not unifiable if $f_1 \neq f_2$ and this property is also correctly handles in part (#3), we assume w.l.o.g. that $f_1 = f_2 = f$.

For the soundness, we prove that if s_a matches s_b, there exists a unifier θ such that $(t^1)_u\theta = (t^2)_v\theta$. We construct such a θ by applying the following rules beginning with $\theta = \emptyset$:

- If '*' in s_a corresponding to a variable x in $(t^1)_u$ matches a symbol in s_b corresponding to a term or variable $(t^2)_{v_j}$, we add $x/(t^2)_{v_j}$ to θ.
- If '*' in s_a corresponding to a variable x in $(t^1)_u$ matches two or more consecutive symbols in s_b corresponding to consecutive terms $(t^2)_{v_j} \ldots (t^2)_{v_{j+l}}$, we add $x/f((t^2)_{v_j}, \ldots, (t^2)_{v_{j+l}})$ to θ.
- If '*' in s_b corresponding to a variable y in $(t^2)_v$ matches a symbol in s_a corresponding to a term $(t^1)_{u_i}$, we add $y/(t^1)_{u_i}$ to θ.
- If '*' in s_b corresponding to a variable y in $(t^2)_v$ matches two or more consecutive symbols in s_a corresponding to consecutive terms $(t^1)_{u_i} \ldots (t^1)_{u_{i+l}}$, we add $y/f((t^1)_{u_i}, \ldots, (t^1)_{u_{i+l}})$ to θ.

Since each variable occurs at most once, θ is well defined and thus $(t^1)_u\theta = (t^2)_v\theta$ holds.

For the completeness, we prove that if there exists θ satisfying $(t^1)_u\theta = (t^2)_v\theta$, s_a matches s_b. We assume that $t = (t^1)_u\theta = (t^2)_v\theta$ is a ground term (i.e., a term not containing any variable) because otherwise each variable can be replaced by any constant. We assume w.l.o.g. that t is in canonical form and has the form $t = f(t_1, t_2, \ldots, t_r)$. It is to be noted that none of t_i, $(t^1)_{u_i}$, and $(t^2)_{v_i}$ have the form $f(\ldots)$ because all of these are in canonical form. Since t does not contain any variable, the following relations hold for some k_i and h_j:

- $(t^1)_{u_i}\theta = t_{k_i}$ (if $k_i + 1 = k_{i+1}$),
- $(t^1)_{u_i}\theta = f(t_{k_i}, \ldots, t_{k_{i+1}-1})$ (otherwise),
- $(t^2)_{v_j}\theta = t_{h_j}$ (if $h_j + 1 = h_{j+1}$),
- $(t^2)_{v_j}\theta = f(t_{h_j}, \ldots, t_{h_{j+1}-1})$ (otherwise),

where $k_1 = h_1 = 1$ and $k_{p+1} = h_{q+1} = r + 1$. Then, we construct a matching between $s_a = s_1 \ldots a_p$ and $s_b = b_1 \ldots b_q$ as follows. We assign intervals to each of $(t^1)_{u_i}$ and $(t^2)_{v_j}$. Initially, we assign $[k_i, k_{i+1} - 1]$ to $(t^1)_{u_i}$ and $[h_j, h_{j+1} - 1]$ to $(t^2)_{v_j}$. First we remove overlaps between $[k_i, k_{i+1} - 1]$ and $[h_j, h_{j+1} - 1]$ if $k_i < h_j < k_{i+1} < h_{j+1}$ or $h_j < k_i < h_{j+1} < k_{i+1}$ holds, in a greedy manner from the smaller positions to larger positions. In the former case, we replace $[k_i, k_{i+1} - 1]$ and $[h_j, h_{j+1} - 1]$ with $[k_i, h_j - 1]$ and $[k_{i+1}, h_{j+1} - 1]$, respectively. In the latter case, we replace $[k_i, k_{i+1} - 1]$ and $[h_j, h_{j+1} - 1]$ with $[h_{j+1}, k_{i+1} - 1]$ and $[h_j, k_i - 1]$, respectively. Then, the resulting intervals do not have overlaps where it is allowed one interval is identical or completely included in another interval. It is to be noted that if $k_i < h_j < k_{i+1} < h_{j+1} < k_{i+2}$ holds, $[h_j, h_{j+1} - 1]$ has overlaps between $[k_i, k_{i+1} - 1]$ and $[k_{i+1}, k_{i+2} - 1]$, but $[k_{i+1}, h_{j+1} - 1]$ remains because the overlap removal is applied in a greedy manner (see y_2 in Fig. 3).

Let $[k^L_i, k^R_i]$ and $[h^L_j, h^R_j]$ be the resulting intervals assigned to $(t^1)_{u_i}$ and $(t^2)_{v_j}$, respectively. It is seen from the construction that each interval is non-empty, and the union of $[k^L_i, k^R_i]$s and the union of $[h^L_j, h^R_j]$s are equivalent (see also Fig. 3 (C)). Then, we construct a matching between s_a and s_b by the following rule:

- if $[k^L_i, k^R_i] = [h^L_j, h^R_j]$, a_i matches b_j,
- if $[k^L_i, k^R_i] \supseteq [h^L_j, h^R_j]$ and $a_i = *$, a_i matches a consecutive subsequence including b_j,
- if $[k^L_i, k^R_i] \subseteq [h^L_j, h^R_j]$ and $b_j = *$, b_j matches a consecutive subsequence including a_i.

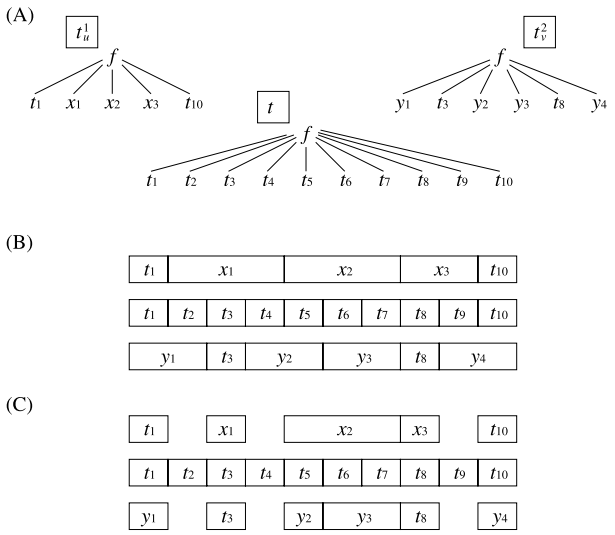

Fig. 3. Illustration of construction of a pairing from a unifier. (A) $t_u^1\theta = t_v^2\theta = t$. (B) Assignment of intervals. (C) Removal of overlaps.

Since each position i in an interval corresponds to t_i and only identical consecutive intervals are removed, this is a valid matching and thus the completeness is proved.

Next, we analyze the time complexity. The most time-consuming part is clearly the string matching with variable-length don't-care symbols. As shown in Lemma 1, it can be done in $O(mn(m+n))$ time. Since the **for**-loops are iterated $O(mn)$ times, the total time complexity is $O(m^2n^2(m+n))$. □

4. Commutative unification

A function f with arity 2 is called *commutative* if $f(x, y) = f(y, x)$ always holds. Commutative unification is a variant of unification in which functions are allowed to be commutative. W.l.o.g., all functions are assumed to be commutative in this section. Free function symbols are not allowed. Consequently, the trees that represent terms as explained in Section 2 now become *unordered* and *binary* trees. Commutative matching was shown to be NP-hard in [6] (by another reduction from 3SAT than the one referred to above). Here, we present a parameterized algorithm for commutative matching for DO-terms and a polynomial-time algorithm for commutative unification with a bounded number of variables.

We consider the following four cases in the rest of this section. (1) When both terms are variable-free. (2) When one term is variable-free. (3) When both terms are DO-terms. (4) The general case where both terms may contain non-DO-variables.

4.1. Case 1: Both terms are variable-free

First note that commutative unification is easy to solve when both t_1 and t_2 are variable-free because in this case, it reduces to the rooted unordered labeled tree isomorphism problem which is solvable in linear time (see, e.g., p. 86 in [1]):

Proposition 2. *Commutative unification for variable-free terms can be done in linear time.*

4.2. Case 2: One term is variable-free

Next, we consider commutative matching. We will show how to construct a 0–1 table $D[u, v]$ for all node pairs $(u, v) \in N(t_1) \times N(t_2)$, such that $D[u, v] = 1$ if and only if $(t_1)_u$ and $(t_2)_v$ can be matched, by applying bottom-up dynamic programming. It is enough to compute these table entries for pairs of nodes with the same depth only. We also construct a table $\Theta[u, v]$, where each entry holds a set of possible substitutions θ such that $(t_1)_u\theta = (t_2)_v$.

Let $\theta_1 = \{x_{i_1}/t_{i_1}, \ldots, x_{i_p}/t_{i_p}\}$ and $\theta_2 = \{x_{j_1}/t_{j_1}, \ldots, x_{j_p}/t_{j_q}\}$ be substitutions. θ_1 is said to be *compatible* with θ_2 if there exists no variable x such that $x = x_{i_a} = x_{j_b}$ but $t_{i_a} \neq t_{j_b}$. Let Θ_1 and Θ_2 be sets of substitutions. We define $\Theta_1 \bowtie \Theta_2 = \{\theta_i \cup \theta_j : \theta_i \in \Theta_1$ is compatible with $\theta_j \in \Theta_2\}$. For any node u, u_L and u_R denote the left and right child of u. The algorithm is as follows:

Procedure *CommutMatch*(t_1, t_2)
  for all pairs $(u, v) \in N(t_1) \times N(t_2)$ with the same depth
  do /* in bottom-up order */
    if $(t_1)_u$ is a variable **then** (#1)
      $\Theta[u, v] \leftarrow \{\{(t_1)_u/(t_2)_v\}\}$; $D[u, v] \leftarrow 1$
    else if $(t_1)_u$ is a constant **then** (#2)
      $\Theta[u, v] \leftarrow \emptyset$;
      if $(t_1)_u = (t_2)_v$ **then** $D[u, v] \leftarrow 1$ **else** $D[u, v] \leftarrow 0$
    else if $\gamma((t_1)_u) \neq \gamma((t_2)_v)$ **then** (#3)
      $\Theta[u, v] \leftarrow \emptyset$; $D[u, v] \leftarrow 0$ /* recall: $\gamma(t)$ is a function symbol of $r(t)$ */
    else (#4)
      $\Theta[u, v] \leftarrow \emptyset$; $D[u, v] \leftarrow 0$;
      for all $(u_1, u_2, v_1, v_2) \in \{(u_L, u_R, v_L, v_R), (u_R, u_L, v_L, v_R)\}$ **do**
        if $D[u_1, v_1] = 1$ and $D[u_2, v_2] = 1$ and $\Theta_1[u_1, v_1] \bowtie \Theta_2[u_2, v_2] \neq \emptyset$
        then $\Theta[u, v] \leftarrow \Theta[u, v] \cup (\Theta_1[u_1, v_1] \bowtie \Theta_2[u_2, v_2])$; $D[u, v] \leftarrow 1$;
  if $D[r(t_1), r(t_2)] = 1$ **then return** true **else return** false.

Algorithm 3. CommutMatch.

Let B_i denote the maximum size of $\Theta[u, v]$ when the number of (distinct) variables in $(t_1)_u$ is i. Then, we have the following conjecture.

Conjecture 1. *$B_1 = 1$ and $B_{i+j} = 2B_iB_j$ hold, from which $B_i = 2^{i-1}$ follows.*

Theorem 3. *If Conjecture 1 holds, then commutative matching can be done using $O(2^k poly(m, n))$ time, where k is the number of variables in t_1.*

Proof. First, we prove the soundness and the completeness of Algorithm 3. For that purpose, it is enough to prove that $D[u, v] = 1$ holds if and only if $(t^1)_u$ matches $(t^2)_v$, and $\Theta[u, v]$ is a set of all possible substitutions θ such that $(t_1)_u\theta = (t_2)_v$ (otherwise $\Theta[u, v] = \emptyset$), using mathematical induction on the size of $(t_1)_u$.

As the base step, we assume that $(t_1)_u$ is a variable or a constant. In the former case, $(t^1)_u$ always matches $(t^2)_v$, and $\Theta[u, v] = \{\{(t_1)_u/(t_2)_v\}\}$ holds. In this case, $D[u, v] = 1$ and $\Theta[u, v] = \{\{(t_1)_u/(t_2)_v\}\}$ hold from part (#1). In the latter case, $(t_1)_u$ matches $(t_2)_v$ if and only if $(t_1)_u = (t_2)_v$. This condition is correctly tested in part (#2). Furthermore, $\Theta[u, v] = \emptyset$ holds. This proves the claim for the base step.

As the induction step, we assume that $(t_1)_u = f((t_1)_{u_L}, (t_1)_{u_R})$. If $\gamma(t_2)_r \neq f$, $(t_1)_u$ does not match $(t_2)_v$. In this case, $D[u, v] = 0$ and $\Theta[u, v] = \emptyset$ hold from part (#3). Then, we assume w.l.o.g. that $(t_2)_v = f((t_2)_{v_L}, (t_2)_{v_R})$ and $D[u_a, v_b]$ and $\Theta[u_a, v_b]$ have already been obtained for all $a, b \in \{L, R\}$. In this case, $(t_1)_u\theta = (t_2)_v$ holds if and only if one of the following holds:

- $(t_1)_{u_L}\theta_L = (t_2)_{v_L}$ and $(t_1)_{u_R}\theta_R = (t_2)_{v_R}$ hold, and θ_L and θ_R are compatible (i.e., the same variables are substituted by the same terms),
- $(t_1)_{u_L}\theta_L = (t_2)_{v_R}$ and $(t_1)_{u_R}\theta_R = (t_2)_{v_L}$ hold, and θ_L and θ_R are compatible.

In each case, $D[u, v] = 1$ holds from part (#4). Furthermore, $\Theta[u, v]$ consists of all θ such that $(t_1)_u\theta = (t_2)_v$ because it is constructed from only compatible pairs. Otherwise, $D[u, v] = 0$ and $\Theta[u, v] = \emptyset$ hold from part (#4). This proves the claim for the induction step.

Next, we analyze the time complexity. We consider the number of elements in $\Theta[u, v]$. A crucial observation is that if $(t_1)_{u_L}$ does not contain a variable then $|\Theta[u, v]| \leq \max(|\Theta[u_R, v_L]|, |\Theta[u_R, v_R]|)$ holds (and analogously for $(t_1)_{u_R}$). Let B_i denote the maximum size of $\Theta[u, v]$ when the number of (distinct) variables in $(t_1)_u$ is i. Then, the following relations hold: $B_1 = 1$, $B_{i+j} = 2B_iB_j$, from which $B_i = 2^{i-1}$ follows. Therefore, $\Theta_1[u_1, v_1] \bowtie \Theta_2[u_2, v_2]$ can be computed in $O(2^k poly(m, n))$ time by using 'sorting' as in usual 'join' operations. Thus, the total running time is also $O(2^k poly(m, n))$. □

Lemma 2. *If the total number of occurrences of variables in t_1 is i, $i \geq 1$, and t_2 is variable-free then the number of different substitutions θ such that $(t_1)\theta = t_2$ is at most 2^{i-1}.*

Proof. Fix an arbitrary left-to-right ordering of the children at each node in the unordered, rooted, node-labeled tree identified with the term t_2 and denote the resulting ordered tree by t'_2. Let X be the set of all ordered trees that: (i) can be obtained by ordering the children at each internal node in the unordered tree identified with t_1; and (ii) can be matched with t'_2. For every $t'_1 \in X$, define $S(t'_1)$ as the substitution θ for which $(t'_1)\theta = t'_2$. Also define $Y = \{S(t'_1) : t'_1 \in X\}$. Then the number of different substitutions θ satisfying $(t_1)\theta = t_2$ equals $|Y|$. (Note that $|X| \geq |Y|$ and that strict inequality is possible; e.g., if $t_1 = f(a, x)$ and $t_2 = f(a, a)$ then $|X| = 2$ while $|Y| = 1$). The rest of this proof derives an upper bound on $|Y|$.

Say that an internal node in a tree is an *lca-node* if it is the lowest common ancestor of two or more leaves corresponding to variables. Consider the following randomized procedure that transforms the unordered tree identified with the term t_1 into an ordered tree t'_1 and outputs the substitution θ such that $(t'_1)\theta = t'_2$ if t'_1 and t'_2 can be matched:

Procedure *GenerateMatching* (t_1)
1. Let t'_1 be the (initially unordered) tree identified with t_1.
2. **for each** lca-node u in t'_1 **do**
   choose one of the two children of u uniformly at random and make it the left child of u, and make the other one the right child.
3. Do a depth-first traversal of t'_1 starting at the root and whenever encountering an internal node u whose children have not yet been ordered **do**
   let u_A be any child of u such that the subtree rooted at u_A is variable-free and let u_B be the other child, and assign left and right among u_A and u_B as well as among all descendants of u_A so the subtree rooted at u_A becomes isomorphic to the corresponding subtree in t'_2, if possible; otherwise, assign left and right to u_A and u_B arbitrarily.
4. **if** t'_1 and t'_2 can be matched **then output** the substitution θ such that $(t'_1)\theta = t'_2$; **else output** the empty set.

Algorithm 4. GenerateMatching.

Since there are i occurrences of variables in t_1 and $i \geq 1$, there are $i-1$ lca-nodes and the procedure has to make $i-1$ choices in Step 2. In Step 3, any choices made regarding the assignment of left and right to u_A and u_B and to the descendants of u_A do not affect the output substitution because whenever such a decision is required, either the two subtrees rooted at the children of the corresponding node in t'_2 are identical to each other or the substitution output in Step 4 will be the empty set. Therefore, Algorithm 4 yields one of at most 2^{i-1} different possible outcomes. Finally, each substitution in Y can be generated by the procedure by making some (not necessarily unique) set of choices. Thus, $|Y| \leq 2^{i-1}$. □

By Lemma 2, Conjecture 1 is true when every variable in t_1 occurs exactly once. In other words, according to Theorem 3, the special case of the commutative matching problem where t_1 is a DO-term is in FPT with respect to the parameter k. We have not been able to prove that Conjecture 1 holds in the more general case where t_1 is not a DO-term.

4.3. Case 3: Both terms are DO-terms

Next, we consider commutative unification when both input terms are DO-terms. In this case, we can use a technique similar to Algorithm 3. However, we do not keep sets of tentative unifiers here since each variable occurs only once. In the following procedure, $D[u, v]$ is determined in a bottom-up manner so that $D[u, v] = 1$ if and only if $(t_1)_u$ and $(t_2)_v$ are unifiable.

Procedure *CommuteUnifyDO* (t_1, t_2)
  for all pairs $(u, v) \in N(t_1) \times N(t_2)$ **do** /* in post-order */
    if $(t_1)_u$ or $(t_2)_v$ is a variable **then** (#1)
      $D[u, v] \leftarrow 1$;
    else if $(t_1)_u$ or $(t_2)_v$ is a constant **then** (#2)
      if $(t_1)_u = (t_2)_v$ **then** $D[u, v] \leftarrow 1$ **else** $D[u, v] \leftarrow 0$;
    else
      Let $(t_1)_u = f_1((t_1)_{u_L}, (t_1)_{u_R})$ and $(t_2)_v = f_2((t_2)_{v_L}, (t_2)_{v_R})$;
      if $f_1 = f_2$ **then** (#3)
        if $(D[u_L, v_L] = 1$ and $D[u_R, v_R] = 1)$ or
            $(D[u_L, v_R] = 1$ and $D[u_R, v_L] = 1)$
        then $D[u, v] \leftarrow 1$ **else** $D[u, v] \leftarrow 0$
      else $D[u, v] \leftarrow 0$;
  if $D[r_1, r_2] = 1$ **then return** true **else return** false.

Algorithm 5. CommuteUnifyDO.

Theorem 4. *Commutative unification for DO-terms can be done in $O(mn)$ time.*

Proof. First, we prove the correctness of Algorithm 5 by showing that $D[u, v] = 1$ holds if and only if $(t_1)_u$ and $(t_2)_v$ are unifiable.

As the base case, we assume that $(t_1)_u$ or $(t_2)_v$ is a constant or a variable. In this case, two terms are unifiable if and only if one of the following holds:

- $(t_1)_u$ is a variable,
- $(t_2)_v$ is a variable,
- $(t_1)_u = (t_2)_v$,

because each variable occurs at most once. These cases are correctly handled in parts (#1) and (#2). Therefore, $D[u, v] = 1$ holds if and only if $(t_1)_u$ and $(t_2)_v$ are unifiable.

As the induction step, we assume that $(t_1)_u = f_1((t_1)_{u_L}, (t_1)_{u_R})$ and $(t_2)_v = f_2((t_2)_{v_L}, (t_2)_{v_R})$ and the values of $D[u_L, v_L]$, $D[u_R, v_R]$, $D[u_L, v_R]$, $D[u_R, v_L]$ have already been determined. In this case, two terms are unifiable if and only if $f_1 = f_2$ and one of the following holds:

- $(t_1)_{u_L}$ and $(t_2)_{v_L}$ are unifiable, and $(t_1)_{u_R}$ and $(t_2)_{v_R}$ are unifiable,
- $(t_1)_{u_L}$ and $(t_2)_{v_R}$ are unifiable, and $(t_1)_{u_R}$ and $(t_2)_{v_L}$ are unifiable.

These cases are correctly handled in part (#3). Therefore, $D[u, v] = 1$ holds if and only if $(t_1)_u$ and $(t_2)_v$ are unifiable. This proves the correctness of the algorithm.

Next, we analyze the time complexity. It is clear that each line in the **for**-loop is executed in a constant time. Since the **for**-loop is iterated $O(mn)$ times, the total time complexity is $O(mn)$. □

4.4. Case 4: General case

Finally, we consider the general case in which both t_1 and t_2 contain variables of any type. Although the linear-time unification algorithm by Paterson and Wegman [25] is not for commutative unification, we can use their basic idea of representing two variable-free terms t_1 and t_2 by a directed acyclic graph (DAG) having distinguished vertices r_1 and r_2 of indegree 0. Let $G(V, E)$ be a directed acyclic graph satisfying the following:

- only r_1 and r_2 have indegree 0,
- each vertex has outdegree 2 or 0,
- for each vertex v with outdegree 0, a constant symbol c_v is assigned to v,
- for each vertex v with outdegree 2, a function symbol f_v with arity 2 is assigned to v,

where $c_u = c_v$ and $f_u = f_v$ are allowed for $u \neq v$.

Then, we can associate a variable-free term t_v to any $v \in V$ by the following rule:

- if v is a vertex with outdegree 0, the term c_v is assigned to t_v,
- if v is a vertex having outgoing edges to v_1 and v_2, the term $f_v(t_{v_1}, t_{v_2})$ is assigned to t_v.

See Fig. 4 for an example of such a $G(V, E)$. Then, testing whether r_1 and r_2 represent the same term takes polynomial time (in the size of G) by using the following procedure:

```
Procedure TestCommutIdent(r1, r2, G(V, E))
  for all (u, v) ∈ V × V do
    if u = v then D[u, v] ← 1 else D[u, v] ← 0;
  for all (u, v) ∈ V × V such that u ≠ v do          /* in post-order */
    if t_u or t_v is a constant then    (#2)
      if t_u = t_v then D[u, v] ← 1 else D[u, v] ← 0;
    else
      Let t_u = f1(t_{u_L}, t_{u_R}) and t_v = f2(t_{v_L}, t_{v_R});
      if f1 = f2 then    (#3)
        if (D[u_L, v_L] = 1 and D[u_R, v_R] = 1) or
           (D[u_L, v_R] = 1 and D[u_R, v_L] = 1)
        then D[u, v] ← 1 else D[u, v] ← 0
      else D[u, v] ← 0;
if D[r1, r2] = 1 then return true else return false.
```

Algorithm 6. TestCommutIdent.

Proposition 3. *Algorithm 6 correctly tests in $O(|V|^2)$ time whether two commutative terms rooted at r_1 and r_2 are identical.*

Proof. First we prove the soundness and the completeness of the algorithm by showing that $D[u, v] = 1$ holds if and only if t_u and t_v are identical.

As the base step, we assume that either t_u or t_v is a constant. If $u = v$, $D[u, v] = 1$ is assigned in part (#1). Otherwise, $D[u, v] = 1$ is assigned in part (#2) if and only if both are the same constant. This proves the claim for the base step.

As the induction step, we assume that $t_u = f_1(t_{u_L}, t_{r_R})$ and $t_v = f_2(t_{v_L}, t_{v_R})$ (otherwise either t_u or t_v is a constant). t_u and t_v are identical if and only if $f_1 = f_2$ holds and one of the following holds

- t_{u_L} and t_{v_L} are identical, and t_{u_R} and t_{v_R} are identical,
- t_{u_L} and t_{v_R} are identical, and t_{u_R} and t_{v_L} are identical.

Clearly, these conditions are checked in part (#3). Therefore, t_u and t_v are identical if and only if $D[u, v] = 1$ is assigned in part (#3). This proves the claim for the induction step and thus the correctness of the algorithm.

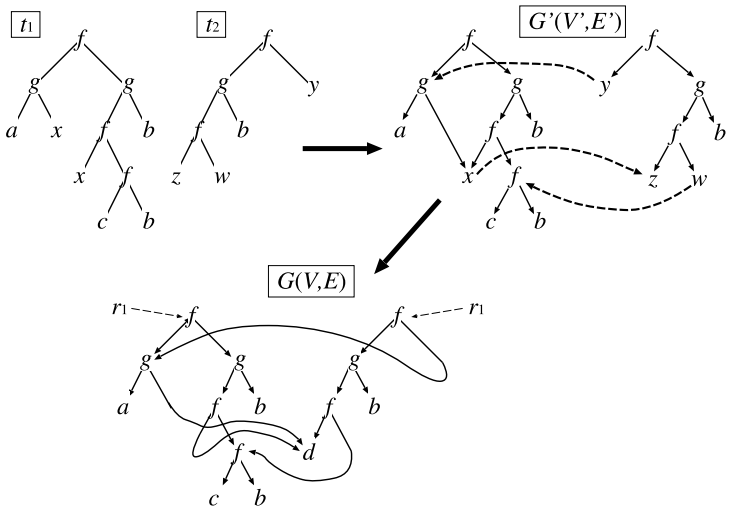

Fig. 4. Example of DAGs $G(V, E)$ and $G'(V', E')$ for the algorithm and proof of Theorem 5. $G'(V', E')$ is drawn so that the orderings of siblings in t'_1 and t'_2 are reflected, and arcs in A are represented by dotted curves. d is a newly introduced constant to represent variable z. In this case, $t_{r_1} = t_{r_2} = f(g(a, d), g(f(d, f(c, b)), b))$.

Next, we analyze the time complexity. Each line in the **for**-loops can be processed in a constant time. Since the **for**-loops are iterated $O(mn)$ times, the total time complexity is $O(mn)$. □

To cope with terms involving variables, we need to consider all possible mappings from the set of variables to $N(t_1) \cup N(t_2)$. For each such mapping, we replace all appearances of the variables by the corresponding nodes, resulting in a DAG to which we apply Algorithm 6. The following pseudocode describes the procedure for terms with variables:

```
Procedure CommutUnify(t1, t2)
  for all mappings M from a set of variables to nodes in t1 and t2 do
    if there exists a directed cycle (excluding a self-loop) then continue;
    Replace each variable having a self-loop with a distinct constant symbol;
    Replace each occurrence of a variable node u with node M(u);
        /* if M(u) = v and M(v) = w then u is replaced by w */
    Let G(V, E) be the resulting DAG;
    Let r1 and r2 be the nodes of G corresponding to t1 and t2;
    if TestCommutIdent(r1, r2, G(V, E)) = true then return true;
  return false.
```

Algorithm 7. CommutUnify.

For an illustration of how it works, see Fig. 4. In summary, we have the following theorem, which means that commutative unification is in the class XP [14]. Note that when the number of variables k is a constant, the time complexity is polynomial.

Theorem 5. *Commutative unification can be done in* $O((m+n)^{k+2})$ *time.*

Proof. First, we prove the soundness of Algorithm 7. Note that it checks whether each graph contains a directed cycle, which performs *occur check* [22]. If the graph does not contain a directed cycle, it returns 'true' only if Algorithm 6 returns true. Since the correctness of Algorithm 6 is proved in Proposition 3, the soundness of Algorithm 7 is proved. It should be noted that a unifier θ can be obtained from M: if variable x is mapped to node v, then $x/t_v \in \theta$.

Next, we prove the completeness. Suppose that t_1 and t_2 are commutatively unifiable. Then, there exist unifiable non-commutative terms t'_1 and t'_2 that are obtained by exchanging left and right arguments in some terms in t_1 and t_2. Furthermore, t'_1 and t'_2 have the most general unifier because they are non-commutative. Let $G'(V', E')$ be a graph obtained by identifying nodes corresponding to the same variable in the union of t'_1 and t'_2. Paterson and Wegman [25] showed that the mgu is represented by a set of arcs A from variable nodes to nodes in $G'(V', E')$, where each node has at most one outgoing arc (there may exist some nodes without outgoing arcs). Since A corresponds to the mgu, $G'(V', E' \cup A)$ is acyclic. We let $M(u) = v$ in $G(V, E)$ if and only $(u, v) \in A$. Then, $G(V, E)$ is also acyclic. Each variable without an outgoing arc is replaced by a distinct constant symbol not appearing in t_1 or t_2. Since the mgu is the most general, this replacement does not affect the unifiability and thus t_{r_1} and t_{r_2} represent identical terms. Therefore, Algorithm 6 returns true. Since such an M is examined in the **for**-loop, Algorithm 7 outputs 'true'. This proves the completeness.

Next, we analyze the time complexity. Recall that k is the number of (distinct) variables in t_1 and t_2. Since the number of possible mappings is bounded by $(m+n)^k$ and Algorithm 6 can be done in $O((m+n)^2)$ time, Algorithm 7 runs in $O((m+n)^{k+2})$ time. □

5. Associative-commutative unification

Associative-commutative unification is the variant of unification in which some functions can be both associative and commutative. The next theorem shows that associative-commutative matching is $W[1]$-hard even if every function is associative and commutative.

Theorem 6. *Matching is $W[1]$-hard with respect to the number of variables even if every function symbol is associative and commutative.*

Proof. We show an FPT-reduction from LCS (variant LCS-3 in [8]). Let (S, l) be an instance of LCS where $S = \{s_1, \ldots, s_k\}$ is a set of strings and l is an integer.

We cannot represent the order of the characters in the strings directly. Instead, we represent the position of each character by the size of the corresponding term.

Let f_1, f_2, f_3, f_4 be distinct functional symbols and a be a constant not appearing in S. For each $s_i[j]$, we define the term $\hat{s}_i[j]$ by

$$\hat{s}_i[j] = f_1(s_i[j], f_2(\overbrace{a, a, \cdots, a}^{j})).$$

Then, we define the term t_2 by

$$t_2 = f_3(f_4(\hat{s}_1[1], \hat{s}_1[2], \cdots, \hat{s}_1[|s_1|]), f_4(\hat{s}_2[1], \hat{s}_2[2], \cdots, \hat{s}_2[|s_2|]),$$
$$\cdots, f_4(\hat{s}_k[1], \hat{s}_k[2], \cdots, \hat{s}_k[|s_k|])).$$

Next, we define the term t^i_j $(i = 1, \ldots, k,\ j = 1, \ldots, l)$ by

$$t^i_j = f_1(x_j, f_2(y_{i,1}, y_{i,2}, \cdots, y_{i,j})),$$

where x_j and $y_{i,h}$s are variables. Then, we define t^i $(i = 1, \ldots, l)$ and t_1 by

$$t^i = f_4(z_i, t^i_1, t^i_2, \cdots, t^i_k),$$
$$t_1 = f_3(t^1, t^2, \cdots, t^l),$$

where z_i is a variable.

We show that t_1 and t_2 are unifiable if and only if LCS of S has length at least l. First we show the 'if' part. Let s_c be a common subsequence of S such that $|s_c| = l$. We consider a partial substitution θ' defined by

$$\theta' = \{x_1/s_c[1], x_2/s_c[2], \ldots, x_l/s_c[l]\}.$$

Then, it is straightforward to see that θ' can be extended to a substitution θ such that $t_2 = t_1\theta$.

Conversely, suppose that there exists a substitution θ satisfying $t_2 = t_1\theta$. We can see from the construction of t_1 and t_2 that if x_j matches $s_i[h]$ and $x_{j'}$ matches $s_i[h']$ for some $i \in \{1, \ldots, k\}$ where $j < j' \leq l$, then $h < h'$ must hold. Let $x_j/a_j \in \theta$ $(j = 1, \ldots, l)$. Then, we can see from the above property that $s_c = a_1 a_2 \ldots a_l$ is a common subsequence of S.

Since the reduction can be done in polynomial time and the number of variables is bounded by a polynomial in k and l, the theorem holds. □

Thus, it is unlikely that an FPT-algorithm for associative-commutative matching exists. On the other hand, associative-commutative matching can be done in polynomial time if t_1 is a DO-term [6]. We can extend this algorithm to the special case of unification where both terms are DO-terms. In the extended algorithm, due to the definition of associative-commutative unification, $f((t_1)_{u_1}, \ldots, (t_1)_{u_p})$ and $f((t_2)_{v_1}, \ldots, (t_2)_{v_q})$ can be unified by $\theta = \{(t_1)_{u_i}/f((t_2)_{v_1}, \ldots, (t_2)_{v_{j-1}}, (t_2)_{v_{j+1}}, \ldots, (t_2)_{v_q}), (t_2)_{v_j}/f((t_1)_{u_1}, \ldots, (t_1)_{u_{i-1}}, (t_1)_{u_{i+1}}(t_1)_{u_p})\}$ if $(t_1)_{u_i}$ and $(t_2)_{v_j}$ are variables for some i, j.

This yields:

Proposition 4. *Associative-commutative unification can be done in polynomial time if both t_1 and t_2 are DO-terms.*

6. String and tree edit distance with variables

6.1. String edit distance with variables

In this subsection, we introduce string edit distance with variables based on unification for strings. Let Σ be an alphabet and Γ a set of variables. A *substitution* is a mapping from Γ to Σ. For any string s over $\Sigma \cup \Gamma$ and substitution θ, let $s\theta$ denote the string over Σ obtained by replacing every occurrence of a variable $x \in \Gamma$ in s by the symbol $\theta(x)$. (We write x/a to express that x is substituted by a.) Two strings s_1 and s_2 over $\Sigma \cup \Gamma$ are called *unifiable* if there exists a substitution θ such that $s_1\theta = s_2\theta$.

Example 2. Suppose $\Sigma = \{a, b, c\}$ and $\Gamma = \{x, y, z\}$. Let $s_1 = abxbx$, $s_2 = ayczc$, and $s_3 = axcby$. Then s_1 and s_2 are unifiable since $s_1\theta = s_2\theta = abcbc$ holds for $\theta = \{x/c,\ y/b,\ z/b\}$. Also, s_2 and s_3 are unifiable since $s_2\theta' = s_3\theta' = accbc$ holds for $\theta' = \{x/c,\ y/c,\ z/b\}$. On the other hand, s_1 and s_3 are not unifiable since there does not exist any θ'' with $s_1\theta'' = s_3\theta''$. □

We shall use the following notation. For any string s, $|s|$ is the length of s. The *string edit distance* (see, e.g., [16]) between two strings s_1, s_2 over Σ, denoted by $d_S(s_1, s_2)$, is the length of a shortest sequence of edit operations that transforms s_1 into s_2, where an *edit operation* on a string is one of the following three operations: a *deletion* of the character at some specified position, an *insertion* of a character at some specified position, or a *replacement* of the character at some specified position by a specified character.[1] For example, $d_S(bcdfe, abgde) = 3$ because $abgde$ can be obtained from $bcdfe$ by the deletion of f, the replacement of c by g, and the insertion of an a, and no shorter sequence can accomplish this. By definition, $d_S(s_1, s_2) = \min_{ed\,:\,ed(s_1)=s_2} |ed| = \min_{ed\,:\,ed(s_2)=s_1} |ed|$ holds, where ed is a sequence of edit operations.

We generalize the string edit distance to two strings s_1, s_2 over $\Sigma \cup \Gamma$ by defining

$$\hat{d}_S(s_1, s_2) = \min_{ed\,:\,(\exists\theta)\,(ed(s_1)\theta = s_2\theta)} |ed|.$$

The *string edit distance problem with variables* takes as input two strings s_1, s_2 over $\Sigma \cup \Gamma$, and asks for the value of $\hat{d}_S(s_1, s_2)$. (To the authors' knowledge, this problem has not been studied before. Note that it differs from the *pattern matching with variables problem* [13], in which one of the two input strings contains no variables and each variable may be substituted by any string over Σ, but no insertions or deletions are allowed.) Let k be the number of variables appearing in at least one of s_1 and s_2. Although $d_S(s_1, s_2)$ is easy to compute in polynomial time (see [16]), computing $\hat{d}_S(s_1, s_2)$ is $W[2]$-hard with respect to the parameter k according to the next theorem:

Theorem 7. *The string edit distance problem with variables is $W[2]$-hard with respect to k when the number of occurrences of every variable is unrestricted, even if one of the two strings has no variables.*

Proof. As in the proof of Theorem 1, we reduce from LCS. It is known that LCS is $W[2]$-hard with respect to the parameter l (problem "LCS-2" in [8]).

Given any instance of LCS, we construct an instance of the string edit distance problem with variables as follows. Let $\Sigma = \Sigma_0 \cup \{\#\}$, where $\#$ is a symbol not appearing in $r_1, r_2, \ldots, r_q$, and $\Gamma = \{x_1, x_2, \ldots, x_l\}$. Clearly, R has a common subsequence of length l if and only if there exists a θ such that $x_1x_2\cdots x_l\theta$ is a common subsequence of R. Now, construct s_1 and s_2 by setting:

$$s_1 = x_1x_2\cdots x_l\#x_1x_2\cdots x_l\#\cdots\#x_1x_2\cdots x_l$$

$$s_2 = r_1\#r_2\#\cdots\#r_q$$

where the substring $x_1x_2\cdots x_l$ occurs q times in s_1. By the construction, there exists a θ such that $x_1x_2\cdots x_l\theta$ is a common subsequence of R if and only if there exists a θ such that $s_1\theta$ is a subsequence of s_2. The latter statement holds if and only if $\hat{d}_S(s_1, s_2) = (\sum_{i=1}^{q} |r_i|) - ql$. Since $k = l$, this is an FPT-reduction. □

The above proof can be extended to prove the $W[1]$-hardness of a restricted case with a bounded number of occurrences of each variable as follows.

Theorem 8. *The string edit distance problem with variables is $W[1]$-hard with respect to k, even if the total number of occurrences of every variable is 2.*

[1] In the literature, "replacement" is usually referred to as "substitution". Here, we use "replacement" to distinguish it from the "substitution" of variables defined above.

Proof. We use the same basic idea for the reduction as in the proof of Theorem 7. As before, let (R, l) be any given instance of LCS, where $R = \{r_1, r_2, \ldots, r_q\}$ is a set of strings over an alphabet Σ_0 and l is an integer. Recall that LCS is $W[1]$-hard with respect to the parameter (q, l) (the problem variant LCS-3 in [8]).

To bound the number of occurrences of each variable by 2, we replace the jth occurrence $(1 \le j \le q)$ of each variable x_i in the string s_1 in the proof of Theorem 7 by $q-1$ new consecutive variables of the form $x_i^{j,h}$, where $h \in \{1, 2, \ldots, q\} \setminus \{j\}$, and then force $x_i^{1,2}\theta = x_i^{2,1}\theta = x_i^{1,3}\theta = \ldots = x_i^{q-1,q}\theta = x_i^{q,q-1}\theta$ in any substitution θ. For this purpose, we stretch out each input string r_j inside the constructed string s_2 by duplicating its symbols and inserting a @-symbol to represent the boundaries between successive positions in r_j. To prevent two variables of the form $x_i^{j,a}$ and $x_i^{j,b}$ from being paired to different positions of r_j, we enclose all $x_i^{j,*}$-variables in the resulting s_1 by a pair of special y_i^j-variables that have to be paired to some symbol of the form $\alpha_{j,i}$ that only occurs twice in s_2. Also, to ensure that every $x_i^{j,h}\theta = x_i^{h,j}\theta$, we place one of $x_i^{j,h}$ and $x_i^{h,j}$ in s_1 and the other one at the corresponding position in s_2. More precisely, the modified construction is as follows.

- Let $\Sigma = \Sigma_0 \cup \{@, \#\} \cup \{\alpha_{j,h} : 1 \le j \le q,\ 1 \le h \le |r_j|\}$, where @, #, and all $\alpha_{j,h}$ are symbols not in Σ_0. Introduce the following variables: $\Gamma = \{x_i^{j,h} : 1 \le i \le l,\ 1 \le j \le q,\ 1 \le h \le q,\ j \ne h\} \cup \{y_i^j : 1 \le i \le l,\ 1 \le j \le q\}$. For each $j \in \{1, 2, \ldots, q\}$, create a string $\tilde{x}^j$ by:
$$\begin{aligned}\tilde{x}^j = {} & y_1^j x_1^{j,1} x_1^{j,2} \cdots x_1^{j,j-1} x_1^{j,j+1} \cdots x_1^{j,q}\, y_1^j\, @ \\ & y_2^j x_2^{j,1} x_2^{j,2} \cdots x_2^{j,j-1} x_2^{j,j+1} \cdots x_2^{j,q}\, y_2^j\, @ \\ & \cdots @\, y_l^j x_l^{j,1} x_l^{j,2} \cdots x_l^{j,j-1} x_l^{j,j+1} \cdots x_l^{j,q}\, y_l^j\end{aligned}$$
- For each $r_j \in R$, express it as $r_j = r_{j,1} r_{j,2} \cdots r_{j,p_j}$, where $p_j = |r_j|$ and each $r_{j,h} \in \Sigma_0$, and create a string $\tilde{r}^j$ by taking $q-1$ copies of the symbol $r_{j,1}$ and enclosing them by a pair of $\alpha_{j,1}$-symbols, followed by an @-symbol, $q-1$ copies of $r_{j,2}$ enclosed by a pair of $\alpha_{j,2}$-symbols, followed by @, etc.:
$$\begin{aligned}\tilde{r}^j = {} & \alpha_{j,1} r_{j,1} r_{j,1} \cdots r_{j,1} \alpha_{j,1}\, @\, \alpha_{j,2} r_{j,2} r_{j,2} \cdots r_{j,2} \alpha_{j,2}\, @ \\ & \cdots @\, \alpha_{j,p(j)} r_{j,p_j} r_{j,p_j} \cdots r_{j,p_j} \alpha_{j,p(j)}\end{aligned}$$
- Partition Γ into two sets $\Gamma_<$ and $\Gamma_>$ by defining $\Gamma_< = \{x_i^{j,h} \in \Gamma : j < h\}$ and $\Gamma_> = \{x_i^{j,h} \in \Gamma : j > h\}$. Let t be a string over $\Gamma_<$ obtained by concatenating the elements of $\Gamma_<$ in any arbitrary order, and let u be the corresponding string over $\Gamma_>$ obtained from t by replacing each symbol $x_i^{j,h}$ by $x_i^{h,j}$. E.g., let $t = x_1^{1,2} x_1^{1,3} \cdots x_i^{a,b} \cdots x_l^{q-1,q}$ and $u = x_1^{2,1} x_1^{3,1} \cdots x_i^{b,a} \cdots x_l^{q,q-1}$.
- Finally, construct s_1 and s_2 by setting:
$$\begin{aligned}s_1 &= \tilde{x}^1 \# \tilde{x}^2 \# \cdots \# \tilde{x}^q \# t \\ s_2 &= \tilde{r}^1 \# \tilde{r}^2 \# \cdots \# \tilde{r}^q \# u\end{aligned}$$

Then each y_i^j-variable occurs twice in s_1, and each $x_i^{j,h}$-variable occurs once in the substring $\tilde{x}^1 \# \tilde{x}^2 \# \cdots \# \tilde{x}^q$ and once in either t or u, i.e., twice in total in s_1 and s_2.

It follows from the construction that there exists a common subsequence of R of length l if and only if there exists some θ such that $s_1\theta$ is a subsequence of $s_2\theta$, which in turn holds if and only if $\hat{d}_S(s_1, s_2) = ((\sum_{i=1}^q |r_i|) - ql) \cdot (q+2)$. Since the number of variables is $q(q-1)l + ql = q^2 l$, which is still a polynomial in q and l, it is an FPT-reduction. □

Furthermore, the following $W[1]$-hardness result is also obtained.

Theorem 9. *The string edit distance problem with variables is $W[1]$-hard with respect to k, even if variables occur only in s_1 and the number of occurrences of every variable is at most 3.*

Proof. We show an FPT-reduction from the maximum clique problem, which is $W[1]$-hard [14]. Let $(G(V, E), l)$ be an instance of the maximum clique problem (i.e., asking whether there exists an l-clique in $G(V, E)$), where $V = \{v_1, \ldots, v_n\}$. We construct two strings s_1 and s_2 as follows.

Let $\Gamma = \{y_1, \ldots, y_l\} \cup \{x_{i,j} \mid i = 1, \ldots, l,\ \ j = 1, \ldots, l,\ \ i \ne j\}$, where we identify $x_{i,j}$ with $x_{j,i}$ (i.e., $x_{i,j} = x_{j,i}$). Let $s^i = y_i x_{i,1} x_{i,2} \cdots x_{i,l} y_i y_i$. Then, s_1 is defined as $s_1 = s^1 s^2 \cdots s^l$. s_1 represents an l-clique. Notice that each variable occurs at most three times.

Let $N(v_i) = \{v_{i,1}, \ldots, v_{i,d_i}\}$ be the set of neighbors of v_i, arranged in the order of $v_1, v_2, \ldots, v_n$. We define t^i $(i = 1, \ldots, n)$ by $t^i = a_i b_{i,1} \cdots, b_{i,d_i} a_i a_i$ where a_i and $b_{i,j}$ are constants. We let $b_{i,p} = b_{j,q}$ iff $v_{i,p} = v_j$ and $v_{j,q} = v_i$. Otherwise, any two symbols are distinct. Finally, we let $s_2 = t^1 t^2 \cdots t^n$. s_2 represents $G(V, E)$.

Then, we can see that $G(V, E)$ has an l-clique iff $\hat{d}_S(s_1, s_2) = |s_2| - |s_1|$. Since the number of variables in s_1 is $\frac{1}{2}l(l-1)+l$, this is an FPT reduction. □

On the positive side, the number of possible θ is bounded by $|\Sigma|^k$. This immediately yields a fixed-parameter algorithm with respect to k when Σ is fixed:

Proposition 5. *The string edit distance problem with variables can be solved in $O(|\Sigma|^k\ poly(m, n))$ time, where m and n are the lengths of the two input strings.*

Also note that in the special case where every variable in the input occurs exactly once, the problem is equivalent to approximate string matching with don't-care symbols, which can be solved in polynomial time [3].

6.2. Tree edit distance with variables

Similar to what was done in Section 6.1, we can combine the *tree edit distance* with unification to get what we call the *tree edit distance problem with variables*. Let $d_T(t_1, t_2)$ be the tree edit distance between two node-labeled (ordered or unordered) trees t_1 and t_2, whose definition is as follows.

To simplify the presentation, we assume that the root $r(T)$ of any tree T has an imaginary parent node $p(T)$ labeled by a unique symbol that does not appear anywhere else in T and that $p(T) \notin V(T)$, where $V(T)$ is the set of nodes in T. Let t_1 and t_2 be two rooted ordered (or unordered) trees. The *tree edit distance* between t_1, t_2 denoted by $d_T(t_1, t_2)$, is the length of a shortest sequence of edit operations that transforms t_1 into a tree isomorphic to t_2, where an *edit operation* on a tree T is one of the following three operations.

Deletion: Delete a node v in $V(T)$ with parent u, making the children of v become children of u. The children are inserted in the place of v into the set of the children of u.
Insertion: Inverse of delete. Insert a node v as a child of any node u in $V(T) \cup \{p(T)\}$, making v the parent of a (possibly empty) subset of the children of u.
Replacement: Change the label (function symbol) of a node v.

Note that the definition of the *isomorphism* differs between ordered and unordered trees: the order of children must be preserved in the ordered tree isomorphism whereas it does not need to be preserved in the unordered tree isomorphism. See [7] for more details of the definitions.

We generalize $d_T(t_1, t_2)$ to two trees, i.e., two terms, over $\Sigma \cup \Gamma$ by defining $\hat{d}_T(t_1, t_2) = \min_{ed\,:\,(\exists\theta)\,(ed(t_1)\theta = t_2\theta)} |ed|$. The *tree edit distance problem with variables* takes as input two (ordered or unordered) trees t_1, t_2 over $\Sigma \cup \Gamma$, and asks for the value of $\hat{d}_T(t_1, t_2)$.

As before, let k be the number of variables appearing in at least one of t_1 and t_2. By applying Theorem 8, we obtain:

Theorem 10. *The tree edit distance problem with variables is $W[1]$-hard with respect to k, both for ordered and unordered trees, even if the number of occurrences of every variable is bounded by 2.*

As demonstrated in [6], certain matching problems are easy to solve for DO-terms. The next theorem states that the ordered tree edit distance problem with variables also becomes polynomial-time solvable for DO-terms. (In contrast, the classic *unordered* tree edit distance problem is already NP-hard for variable-free terms; see, e.g., [7]).

Theorem 11. *The ordered tree edit distance problem with variables can be solved in polynomial time when t_1 and t_2 are DO-terms.*

Proof. Let F_1 and F_2 be two ordered forests. Let T_1 (resp., T_2) be the rightmost tree of F_1 (resp., F_2). It is known (see, e.g., [7]) that the rooted ordered tree edit distance can be computed by the following dynamic programming procedure in $O(m^2n^2)$ time:

$$
\begin{aligned}
D[\epsilon, \epsilon] &\leftarrow 0,\\
D[F_1, \epsilon] &\leftarrow D[F_1 - r(T_1), \epsilon] + \delta(r(T_1), -),\\
D[\epsilon, F_2] &\leftarrow D[\epsilon, F_2 - r(T_2)] + \delta(-, r(T_2)),\\
D[F_1, F_2] &\leftarrow \min \begin{cases} D[F_1 - r(T_1), F_2] + \delta(r(T_1), -),\\ D[F_1, F_2 - r(T_2)] + \delta(-, r(T_2)),\\ D[F_1 - T_1, F_2 - T_2] + D[T_1 - r(T_1), T_2 - r(T_2)] + \delta(r(T_1), r(T_2)), \end{cases}
\end{aligned}
$$

where ϵ denotes the empty tree, $F - v$ (resp., $F - T$) is the forest obtained by deleting v (resp., T) from F, $\delta(x, x) = 0$ and $\delta(x, y) = 1$ for $x \neq y$, and $D[t_1, t_2]$ is the tree edit distance between the two trees t_1 and t_2.

To cope with DO-variables, it is enough to add the following when taking the minimum in the recursive formula for computing $D[F_1, F_2]$ above:

$$D[F_1 - T_1, F_2 - T_2], \quad \text{if } T_1 \text{ or } T_2 \text{ consists of a variable node.}$$

It is clear that the time complexity is the same as that of the original dynamic programming procedure, and hence polynomial. (More sophisticated techniques for further reducing the time complexity of computing the tree edit distance mentioned in [7] and elsewhere may also be applied here.) □

7. Concluding remarks

In this paper, we have studied the parameterized complexity of unification with associative and/or commutative functions with respect to the number of variables. See Table 1 in Section 1.2 for a summary.

Some remaining open problems include

1. Determining whether each of the commutative unification problem and the matching version of Theorem 8 (i.e., where all variables occur in one of the strings and the number of occurrences of each variable is at most 2) is $W[1]$-hard or FPT.
2. Determining whether associative unification is in XP.
3. Resolving Conjecture 1.

Acknowledgements

This work was partially supported by the Collaborative Research Programs of National Institute of Informatics (1-2 (2013–2014)) and of Institute for Chemical Research, Kyoto University (2016-27). TA was partially supported by Grant-in-Aid #26240034 from JSPS, Japan. JJ was supported by the Hakubi Project at Kyoto University. TT was partially supported by JSPS, Japan (Grant-in-Aid for Young Scientists (B) 25730005).

Appendix A. Definitions of FPT and W[i] by [8]

A *parameterized problem* L is a subset of $\Sigma^* \times N$, where Σ is a fixed alphabet and N is a natural number. For L and $k \in N$, we write L_k to denote the associated fixed-parameter problem $L_k = \{x | (x, k) \in L\}$.

Definition 1. A parameterized problem L is fixed parameter tractable if there is a constant α and an algorithm Φ such that Φ decides if $(x, k) \in L$ in time $f(k)|x|^{\alpha}$ where $f : N \to N$ is an arbitrary function.

The classes related to FPT and $W[i]$ are intuitively based on the complexity of the circuits required to check a solution, or alternatively, the "natural logical depth" of the problem.

Definition 2. A mixed Boolean circuit consists of the following two kinds of gates.

1. Small gates: "not" gates, "and" gates and "or" gates with bounded fan-in. We will usually assume that the bound of fan-in is 2 for "and" gates and "or" gates, and 1 for "not" gates.
2. Large gates: "and" gates and "or" gates with unrestricted fan-in

Definition 3. The depth of a circuit C is defined to be the maximum number of gates (small or large) on an input–output path in C. The weft of a circuit C is the maximum number of large gates on an input–output path in C.

Definition 4. A family of decision circuits F has bounded depth if there is a constant h such that every circuit in the family F has depth at most h. F has bounded weft if there is a constant t such that every circuit in the family F has weft at most t. The weight of a Boolean vector x is the number of 1's in the vector.

Definition 5. Let F be a family circuit of decision circuits. We allow F to have many different circuits with a given number of inputs. To F we associate the parameterized circuit problem $L_F = \{(C, k) : C \text{ accepts an input vector of weight } k\}$.

Definition 6. A parameterized problem L belongs to $W[t]$ if L reduces to the parameterized circuit problem $L_{F(t,h)}$ for the family $F(t, h)$ of mixed type decision circuits of weft at most t, and depth at most h, for some constant h.

Definition 7. A parameterized problem L belongs $W[P]$ if L reduces to the circuit problem L_F, where F is the set of all circuits without restrictions.

The class of fixed parameter tractable problems is denoted by FPT. The framework above describes a hierarchy of parameterized complexity classes

$$\mathrm{FPT} \subseteq W[1] \subseteq W[2] \subseteq \cdots \subseteq W[P]$$

for which there are many natural hard or complete problems. For example, INDEPENDENT SET, CLIQUE, and LCS-3 are known to be $W[1]$-complete. DOMINATING SET is known to be $W[2]$-complete, and LCS-2 is $W[2]$-hard.

References

[1] A.V. Aho, J.E. Hopcroft, J.D. Ullman, The Design and Analysis of Computer Algorithms, Addison-Wesley, 1974.
[2] K. Aikou, Y. Suzuki, T. Shoudai, T. Uchida, T. Miyahara, A polynomial time matching algorithm of ordered tree patterns having height-constrained variables, in: Combinatorial Pattern Matching, Springer, 2005, pp. 346–357.
[3] T. Akutsu, Approximate string matching with don't care characters, Inform. Process. Lett. 55 (5) (1995) 235–239.
[4] T. Akutsu, Approximate string matching with variable length don't care characters, IEICE Trans. Inf. Syst. E79D (1996) 1353–1354.
[5] F. Baader, W. Snyder, Unification theory, in: Handbook of Automated Reasoning, 2001, pp. 447–533.
[6] D. Benanav, D. Kapur, P. Narendran, Complexity of matching problems, J. Symbolic Comput. 3 (1) (1987) 203–216.
[7] P. Bille, A survey on tree edit distance and related problems, Theoret. Comput. Sci. 337 (1) (2005) 217–239.
[8] H. Bodlaender, R.G. Downey, Fellows, H.T. Wareham, The parameterized complexity of sequence alignment and consensus, Theoret. Comput. Sci. 147 (1995) 31–54.
[9] H.-J. Bürckert, A. Herold, D. Kapur, J.H. Siekmann, M.E. Stickel, M. Tepp, H. Zhang, Opening the AC-unification race, J. Automat. Reason. 4 (4) (1988) 465–474.
[10] T.H. Cormen, C.E. Leiserson, R.L. Rivest, C. Stein, Introduction to Algorithms, MIT Press, 2009.
[11] D. De Champeaux, About the Paterson–Wegman linear unification algorithm, J. Comput. System Sci. 32 (1) (1986) 79–90.
[12] S. Eker, Single elementary associative-commutative matching, J. Automat. Reason. 28 (1) (2002) 35–51.
[13] H. Fernau, M.L. Schmid, Pattern matching with variables: a multivariate complexity analysis, Inform. and Comput. 242 (2015) 287–305.
[14] J. Flum, M. Grohe, Parameterized Complexity Theory, Springer Verlag, Berlin, 2006.
[15] D. Gilbert, M. Schroeder, Fuzzy unification and resolution based on edit distance, in: Proc. 1st IEEE International Symposium on Bioinformatics and Biomedical Engineering, 2000, pp. 330–336.
[16] N.C. Jones, P. Pevzner, An Introduction to Bioinformatics Algorithms, MIT Press, 2004.
[17] P. Julián-Iranzo, C. Rubio-Manzano, An efficient fuzzy unification method and its implementation into the Bousi∼Prolog system, in: Proc 2010 IEEE International Conference on Fuzzy Systems, 2010, pp. 1–8.
[18] S. Kamali, F.W. Tompa, A new mathematics retrieval system, in: Proceedings of the 19th ACM International Conference on Information and Knowledge Management, 2010, pp. 1413–1416.
[19] D. Kapur, P. Narendran, Complexity of unification problems with associative-commutative operators, J. Automat. Reason. 9 (2) (1992) 261–288.
[20] K. Knight, Unification: a multidisciplinary survey, ACM Comput. Surv. 21 (1) (1989) 93–124.
[21] P. Lincoln, J. Christian, Adventures in associative-commutative unification, J. Symbolic Comput. 8 (1–2) (1989) 217–240.
[22] J.W. Lloyd, Foundations of Logic Programming, Springer, 1984.
[23] T.T. Nguyen, K. Chang, S.C. Hui, A math-aware search engine for math question answering system, in: Proceedings of the 21st ACM International Conference on Information and Knowledge Management, 2012, pp. 724–733.
[24] NTCIR, http://research.nii.ac.jp/ntcir/ntcir-10/conference.html, 2013.
[25] M.S. Paterson, M.N. Wegman, Linear unification, J. Comput. System Sci. 16 (1978) 158–167.
[26] J.A. Robinson, A machine-oriented logic based on the resolution principle, J. ACM 12 (1) (1965) 23–41.
[27] K. Slind, AC unification in HOL90, in: Higher Order Logic Theorem Proving and Its Applications, Springer, 1994, pp. 437–449.
[28] M.E. Stickel, A unification algorithm for associative-commutative functions, J. ACM 28 (3) (1981) 423–434.