Scientific Annals of Computer Science vol. 23 (2), 2013, pp. 229–249
doi: 10.7561/SACS.2013.2.229

Algorithmics of Posets Generated by Words Over Partially Commutative Alphabets (Extended Version)

Łukasz Mikulski[1], Marcin Piątkowski[1], Sebastian Smyczyński[1]

Abstract

It is natural to relate partially ordered sets (posets in short) and classes of equivalent words over partially commutative alphabets. Their common graphical representation are Hasse diagrams. We investigate this relation in detail and propose an efficient online algorithm that decompresses a concurrent word to its Hasse diagram. The lexicographically minimal representative of a trace (an equivalence class of words) is called its lexicographical normal form. We give an algorithm which enumerates, in the lexicographical order, all distinct traces identified by their lexicographical normal forms. The two presented algorithms are the main contribution of this paper.

Keywords: poset, Hasse diagram, partially commutative alphabets, algorithms, generations

Introduction

Many practical problems related to partially ordered sets have a very high time complexity. Examples of such problems are the #P-complete problem of counting the number of posets linear extensions [1] or the NP-complete problem of computing the minimal number of jumps [3].

Among less complex problems one can mention a problem of computing the Hasse diagram of a poset (the transitive reduction of its graph) which has cubic time complexity. We consider a language-theoretic approach to

[1] Faculty of Mathematics and Computer Science, Nicolaus Copernicus University, Chopina 12/18, 87-100 Toruń, Poland.
Email: {lukasz.mikulski, marcin.piatkowski, sebastian.smyczynski}@mat.umk.pl

posets that uses words over partially commutative alphabets. It allows us to exploit the inner structure of a given poset to develop new algorithms. The complexity of these algorithms depends not only on the number of elements of a poset, but also on the complexity of its structure (the size of the concurrent alphabet used to represent the poset). The basic theory together with some algorithms can be found in [4] and [5]. However, most of ideas presented there is based on the projection representation of traces which results $\mathrm{O}(nk)$ memory complexity.

In the first section we give some basic notions related to the formal languages, partial orders and concurrency theories. In Section 2 we look more closely at the relation between words over partially commutative alphabets and posets. We analyse the dependence graphs of concurrent words and their relation to the Hasse diagrams of posets. We also summarise the situation when Hasse diagram has a special structure. Particularly, we show that every poset can be generated by the word over the partially commutative alphabet. Moreover we prove that $P4$-freeness of dependence relation of the concurrent alphabet guarantees N-freeness of the Hasse diagram.

In the following section we deal with a decoding of the Hasse diagram from an arbitrary concurrent word and give an online algorithm for its construction. The presented algorithm works in time of $\mathrm{O}(nk^2)$, where n denotes the size of the poset, and k the size of the alphabet. Note that the presented algorithm has memory complexity of $\mathrm{O}(k^2)$. Together with the possibility of immediate output of partial results it allows us to process long words.

The study of the properties of words over partially commutative alphabets requires efficient tools for the enumeration of distinct classes of equivalent words (in the sense of the independence relation). We deal with this practical problem in the fourth section. Basically, we identify classes of equivalent words with their lexicographical normal forms [5]. Further, we show how to compute the considered representatives of all classes in the lexicographical order. For a given concurrent word (that is canonical), the single step of our algorithm computes the next (in the lexicographical order) word that is canonical. Moreover, if we consider possible blocks of identical letters instead of their individual occurrences, we can achieve the better time complexity of a single step.

The preliminary version of this paper was published in Proceedings of Prague Stringology Conference 2011 ([13]). The new version is revised, contains some additional facts, proofs, new and extended examples, and

improved description of generation algorithm (with better time complexity).

1 Basic Notions

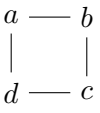
We use some basic notions of formal languages theory. By Σ we denote a finite set, called the *alphabet*. Elements of the alphabet are called *letters*. *Words* are sequences over the alphabet Σ. The sets of all finite words is denoted by Σ^*, while by $Alph(w) \subseteq \Sigma$ we denote the set of all letters contained in the word w.

A *concurrent alphabet* is a pair (Σ,D), where Σ is an alphabet and $D \subseteq \Sigma \times \Sigma$ is a reflexive and symmetric relation, called *dependence relation*. With dependence we associate, as another relation, an *independence relation* $I = \Sigma \times \Sigma \setminus D$. Having the concurrent alphabet we define a relation that identifies similar words. We say that a word $\sigma \in \Sigma^*$ is in relation $\equiv_D$ with a word $\tau \in \Sigma^*$ if there exists a finite sequence of commutations of subsequent and independent letters that leads from σ to τ. Relation $\equiv_D \subseteq \Sigma^* \times \Sigma^*$ is a congruence (whenever it causes no confusion, relation symbol D will be omitted).

To emphasise that considered word $w \in \Sigma^*$ is over a concurrent alphabet (Σ, D) (an alphabet equipped with a dependence relation) we call it a *partially commutative word*. On the other hand, dividing the set Σ^* by the relation $\equiv$ we get a quotient monoid. The elements of $\Sigma^*/_{\equiv}$ are often called *traces* (see [6, 11, 12]). This way, every partially commutative word σ determines a trace $\alpha = [\sigma]$.

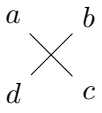
Example. Let $\Sigma = \{a, b, c, d\}$ and (Σ, D) be the concurrent alphabet, where

D



while the independence relation is

I



The words $abb\mathbf{aac}d$ and $abb\mathbf{caa}d$ are equivalent.

Note that dependence relation D is reflexive. However, here and through the paper loops in graphical representation of the relation are omitted.

A *partial order* on the set X is a reflexive, antisymmetric and transitive relation $\leq \subseteq X \times X$. If additionally every pair of elements from X is comparable, the relation $\leq_t$ is called the *total order*. A pair $(X,\leq)$ is called the *partially ordered set*, (*poset* in short). Observe that in the case of a

totally ordered set $(X,\leq_t)$ elements of X form a sequence (denoted by $w_{\leq_t}$). A *linearisation* of a partial order $(X,\leq)$ is a sequence $w_{\leq_t}$ for any total order $(X,\leq_t)$ containing $(X,\leq)$, which means that $\leq \subseteq \leq_t$.

With every poset we can associate its directed graph (digraph in short) $G = (X, E)$. The vertices of G are elements of the poset. There is an arc between two vertices $x, y \in X$ if $x < y$ (i.e. $x \leq y$ but $x \neq y$). Such a graph is always acyclic. We can also define the Hasse diagram of the poset $(X,\leq)$ as a transitive reduction of the graph G. More general, the graph of every relation on X which transitive closure is equal to $\leq$ is called a *diagram* of $\leq$.

Definition 1 *Let $G = (X, E)$ be an acyclic graph. The* Hasse diagram *of G is the acyclic graph $H = (X, E' \subseteq E)$, such that an arc $(x, y) \in E'$ if there is no $z \in X$ (different than x and y) for which there are both paths (in G) from x to z and from z to y.*

The example of a poset's graph and its Hasse diagram is shown on Figure 1. We can observe that the size of the Hasse diagram is significantly smaller than the size of the poset's graph. Therefore, Hasse diagrams can be seen as a compact representations of posets. Another efficient representation of a poset is discussed in the following section.

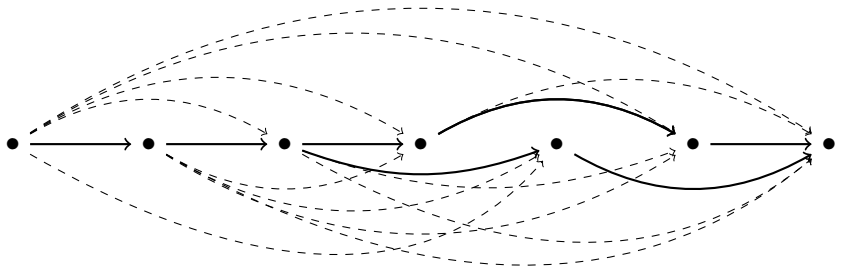


Figure 1: The graph of an example poset. The dashed edges are not contained in its Hasse diagram.

2 From Partially Commutative Words to Posets

With every word w over partially commutative alphabet (Σ, D) we can associate a poset. One of the diagrams of this poset is induced by the dependence graph of a word w. An element v_j associated with the letter w_j is greater than an element v_i associated with the letter w_i if $i < j$ and $w_i D w_j$. The label of the element (vertex) v_i is denoted by $\ell(v_i) = w_i$. It is worth noting that two words are equivalent if and only if their dependence graphs are the same (isomorphic and respecting labelling).

By the definition of a diagram, reflexive transitive closure of the dependence graph of a word is basically a graph of a poset associated with the word. Additionally, transitive reduction of this dependence graph is exactly the Hasse diagram of the considered poset, see Figure 2.

Remark 1 *For an arbitrary concurrent word, its Hasse diagram representation is unique. On the other hand, two different words over the same concurrent alphabet can lead to the same Hasse diagram structure (without taking into account the labelling of the nodes).*

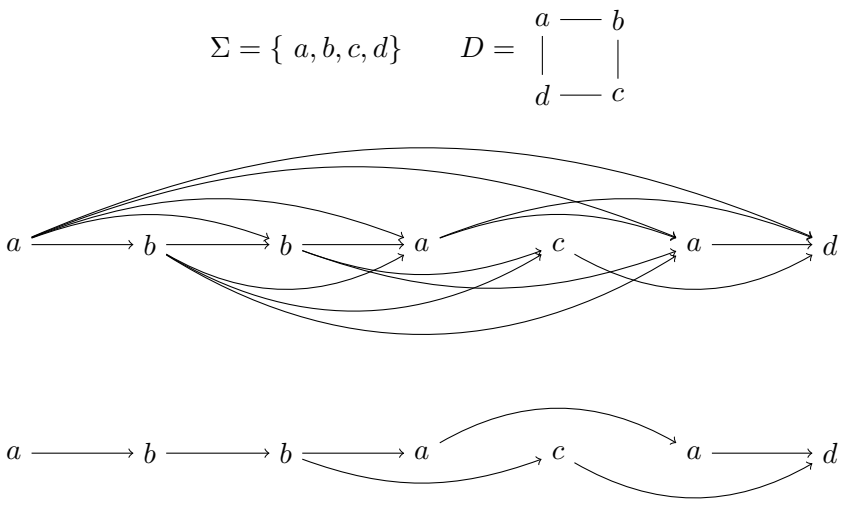


Figure 2: A concurrent alphabet (Σ, D), dependence graph and Hasse diagram of word *abbacad* over that alphabet.

Lemma 1 *Every finite poset $(P, \leq)$ can be generated by a word over concurrent alphabet.*

Proof: For a given finite poset $(P, \leq)$, let us define a concurrent alphabet (Σ, D) in such a way that $\Sigma = P$ and $p_1 D p_2$ if and only if $p_1 \leq p_2$ or $p_2 \leq p_1$. An arbitrary linearisation of the poset $(P, \leq)$ corresponds in a natural way with a word $v \in \Sigma^*$ which generates a poset equal to $(P, \leq)$. □

The above observations allow us to represent every poset in a compressed way by a pair consisting of concurrent alphabet and a single word over that alphabet. In the next section we will provide an efficient algorithm that produces a Hasse diagram by decompressing a given word to its associated poset.

Further optimisation, possible only for Hasse diagrams which are minimal series-parallel graphs [18], leads us to another data structure which can be used to solve many problems in a simpler way (for instance, the #P-complete problem of counting the number of linear expansions [1] can be solved in a linear time for such posets). In what follows, by a *sink* of a directed graph we mean any vertex v_1 that has no outgoing arc, while by a *source* we mean any vertex v_2 that has no ingoing arc. Note that every acyclic graph has at least one source and one sink, while a path may be considered as a special kind of graph with all vertices having at most one ingoing and at most one outgoing arc, and exactly one sink and one source.

Definition 2 *A minimal Series-Parallel digraph (MSP) is a graph consisting of a single vertex and no arcs or is constructed from two disjoint MSPs – $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$ – by the following operations:*

- *Parallel composition: $G_P = (V_1 \cup V_2, E_1 \cup E_2)$;*
- *Serial composition: $G_P = (V_1 \cup V_2, E_1 \cup E_2 \cup T_1 \times S_2)$;*

where T_1 is the set of sinks of G_1 and S_2 is a set of sources of G_2. In other words, series-parallel graphs can be represented as an expression built by series and parallel composition of graphs with single-vertex graphs as atoms.

The example of the graphical representation of the composition operations is shown on Figure 3.

The properties of series-parallel graphs are deeply studied (see for instance [2, 15, 18]). A very useful determinant for sequential parallel graphs is their N-freeness [17].

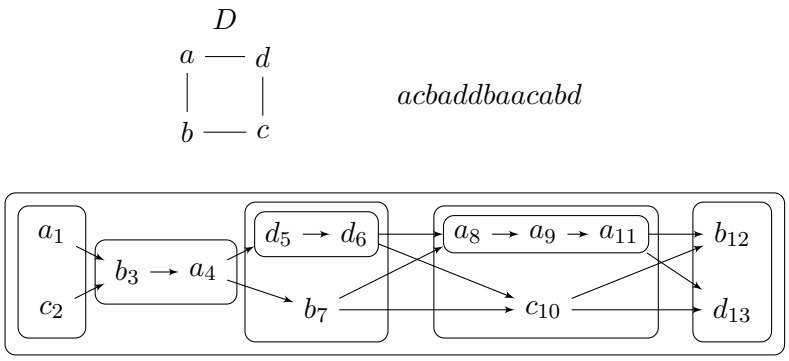


Figure 3: The dependent alphabet D, the word w and its Hasse diagram divided to series-parallel blocks.

Definition 3 *An* N-poset *is a poset consisting of four elements* a, b, c, d *with relations* $a < c, b < c$ *and* $b < d$ *(drawing a graph of such poset with greater elements higher brings to mind capital letter* N*).[10]*

Definition 4 *A poset is* N*-free if its graph does not contain an induced subgraph isomorphic with Hasse diagram of* N*-poset.*

Remark 2 *In the case of undirected graphs, analogue is* $P4$*-free graph (a graph that does not contain an induced path of length 3).*

The example of the graphical interpretation of the above mentioned notions can be seen on Figure 4.

In general, this type of graphs, also in the context of partial orders, is deeply studied (see [9, 16, 18] and the references therein). However, observation worth mentioning is the following:

Lemma 2 *If a dependence graph* D *of an alphabet* Σ *is* $P4$*-free then the Hasse diagram of every partially commutative word* $w \in (\Sigma^*, D)$ *is* N*-free.*

Proof: We prove this lemma by contradiction. Let us suppose that there exists a word w over concurrent alphabet (Σ, D) with $P4$-free graph of relation D which Hasse diagram $H(w)$ has induced digraph $N = (V_n, E_n)$ of N shape.

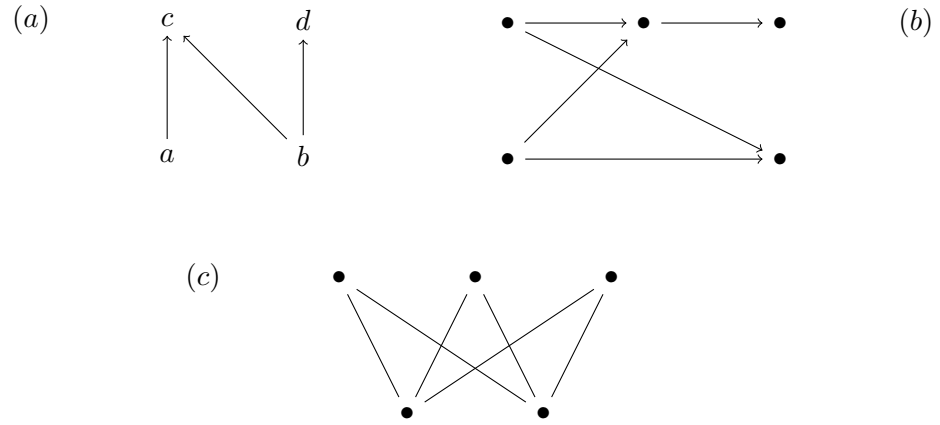


Figure 4: N-poset, simple N-free poset and $P4$-free graph.

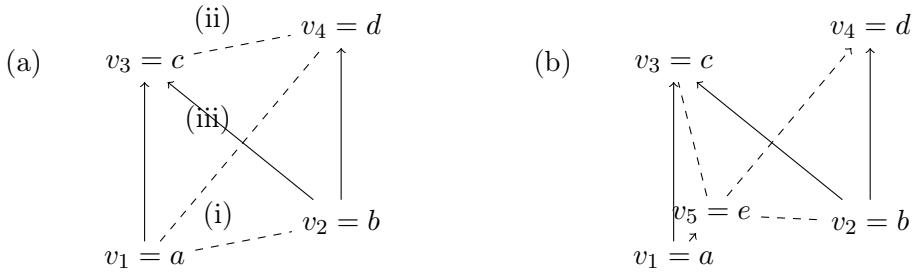


Figure 5: Proof situation.

Without loss of generality we can assume that graph N is the first graph in Figure 4. It means that $V_n = \{v_1, v_2, v_3, v_4\}$ and labels of these vertices are $v_1 = a, v_2 = b, v_3 = c, v_4 = d$.

Let us consider the situation depicted in Figure 5a. We start from relation (i) and claim that letters a and b are independent. Indeed, otherwise, there has to be a path p in $H(w)$ between vertices v_1 and v_2. Let us suppose that v_1 is source of path p. Then there is a path from v_1 to v_3, so there should not be an arc (v_1, v_3) in Hasse diagram.

We proceed by deducing that letters c and d (relation (ii)) are also independent. Otherwise, there is a path p between vertices v_3 and v_4 in graph $H(w)$. If v_3 is a source of path p then arc (v_2, v_4) should not be present

in $H(w)$. If v_4 is a source of path p then arc (v_2, v_3) should not be present in $H(w)$.

The relations aIb and cId shows that $\ell(v_1) \neq \ell(v_2)$ and $\ell(v_3) \neq \ell(v_4)$. We also know that bDc so $\ell(v_1) \neq \ell(v_3)$ because one letter can not be at once dependent and independent with another. For similar reasons $b \neq d$ and $b \neq c$.

Now we consider the relation (iii) from Figure 5a. Firstly let us suppose that aId. Then also $a \neq d$ and we have a subalphabet $\{a, b, c, d\} \subseteq \Sigma$ with a dependence graph of shape $P4$. It is in contradiction with the assumption that D is $P4$-free.

The last situation to consider is aDd. Then there should be a path p between v_1 and v_4. If the vertex v_4 is a source of path p then we have a path form v_2 to v_3 of length greater than 1, so the arc (v_2, v_3) should not be present in graph $H(w)$. Let us suppose that vertice v_1 is a source of path p. Let the first arc of path p be (v_1, v_5) and the label of v_5 is denoted by e (see Figure 5b). Then, the letter e is independent with c (otherwise, one of arcs (v_1, v_3) or (v_1, v_5) should not be present in $H(w)$) and independent with b (otherwise, one of arcs (v_1, v_3) or (v_2, v_4) should not be present in $H(w)$). It means that $\ell(v_5) \neq \ell(v_2)$, $\ell(v_5) \neq \ell(v_3)$ and $\ell(v_5) \neq \ell(v_1)$, so we have a subalphabet $\{a, b, c, e\} \subseteq \Sigma$ with a dependence graph of shape $P4$. It is in contradiction with the assumption that D is $P4$-free and the proof is complete. □

3 Construction of Hasse Diagram

This section is devoted to the problem of constructing the Hasse diagram (see Definition 1) for an arbitrary concurrent word. At the beginning, we give an algorithm and its pseudo-code. After that, we discuss the complexity of our solution.

The algorithm exploits the knowledge of the structure of resulting diagram. We can summarise it in the following facts:

Lemma 3 *Let $w \in \Sigma^*$ be a word and $H(w) = (V, E_H)$ be a Hasse diagram of w. If there exists the arc connecting vertices v_i and v_j (labelled $a = w_i$ and $b = w_j$ respectively) then letters a and b do not appear in word w between indexes i and j.*

Proof: Let $G = (V, E)$ be the dependence graph of the word w over the

concurrent alphabet (Σ, D). The existence of an arc between v_i and v_j in graph H implies that there is also an arc in the graph G, hence letters a and b are dependent (formally aDb). Let us suppose that there exists a letter $c = \ell(v_k)$ (for $i < k < j$) that is dependent both with a and b. Then, by Definition 1, there is a path in graph G between vertices v_i and v_j of length greater than one, so there is no arc between v_i and v_j in graph $H(w)$, which leads to a contradiction, and completes the proof. □

Lemma 4 *Let $w \in \Sigma^*$ be a word and $H(w)$ be a Hasse diagram of w. Then, in $H(w)$, for each vertex there are no more than $k = |\Sigma|$ outgoing arcs and no more than k ingoing arcs.*

Proof: Let $G = (V, E)$ be the dependence graph of the word w over concurrent alphabet (Σ, D). Let us suppose that there is a vertex v_i which has $k+1$ outgoing arcs. There are k letters in alphabet Σ, so two of these outgoing arcs lead to two distinct vertices v_j and v_k $(i < j < k)$ labelled with the same letter. Without loss of generality we can assume that $\ell(v_i) = a$ and $\ell(v_j) = \ell(w_k) = b$. From Lemma 3 there is no arc in graph $H(w)$ between vertices v_i and v_k, which proves that there are at most k outgoing arcs. Similar reasoning allows us to prove the second part of the lemma on the number of ingoing arcs. □

Lemma 5 *Let $w \in \Sigma^*$ be a word and $H(w) = (V, E)$ be a Hasse diagram of w. Ingoing arcs of a given vertex v_i are fully determined by the vertices associated with last occurrences of letters dependent with $\ell(v_i)$. More formally, $(v_j, v_i) \in E$ if and only if $j < i$ and $\ell(v_i)D\ell(v_j)$ and there is no vertex v_k, such that $\ell(v_k)D\ell(v_i)$ and $j < k < i$ and there is a path from v_j to v_k.*

Proof: Let (v_j, v_i), where $\ell(v_j) = a$, be an arc in $H(w)$. Lemma 3 implies that v_j must be the last occurrence of letter a in word w that precedes w_i. Second part of the lemma follows directly from Definition 1 (see proof of the Lemma 3). □

Using foregoing observations we propose an additional structure that saves information about last occurrences of each letter processed so far. It

allows us to immediately add a new vertex to Hasse diagram, with all of its ingoing arcs. Our structure consists of a list of dependencies, a set of visibility (both of size at most k) and a pointer to the last occurrence, for each letter of the alphabet Σ. The list D_a contains all letters dependent with a in LIFO (last in – first out) order of their last occurrence in the currently constructed part of the diagram. The set V_a contains all letters b whose last occurrences are visible from the last vertex labelled with a. In other words, there exists a path from v_i to v_j where v_i and v_j are the last occurrences of letters $\ell(v_i)$ and $\ell(v_j)$ in hitherto diagram. Such elements v_i will be called sources of v_j. The last element is a pointer L_a which is basically a pointer to the last vertex labelled with the letter a in processed diagram. We will also use a temporary set V.

Before we start generating Hasse diagram, we set all pointers to *null* and all sets to be empty. The lists of dependencies should be complete with all dependent letters, but the initial order does not matter. With such data we are ready to process a new letter a of a word w in online manner, updating the proposed structure after each step and creating a new vertex and new arcs. During the addition of the new vertex labelled with letter a we clear set V and browse the list D_a. For each letter b from that list we check if the pointer L_b is not empty and if b does not belong to V (its last occurrence is not already visible from the new vertex). If we succeed, we add a new arc from the vertex v_b pointed by L_b to the newly created vertex. Addition of a new arc implies that there is also a path from every source of v_b to the recently created vertex. Therefore, we add set V_b to our temporary set V. It is worth noting that the order of processing letters form list D_a is important because of the dynamically changing set V.

After adding new arcs, we have to update our structure. Firstly, we remove the letter a from each set V_b – the new vertex is now the last occurrence of letter a. Next, we switch the position of the letter a in every list D_b – the letter a is the most recent letter now. The last operation is the update of the set V_a to $V \cup a$ and pointer L_a to the position of the new vertex. Note that in the rest of the generation process we need only the most recent vertex labelled with a and we do not have to store other vertices labelled with the same letter.

The correctness of the algorithm presented above relies on lemmas formulated at the beginning of this section. Let us discuss the memory and time complexity of our solution. The proposed data structure consists of k lists D of at most k items each. It gives us k^2 elements. The k sets V can

Algorithm 1: Hasse diagram

```
1  Input: a word w = w_1 w_2 ··· w_n over a concurrent alphabet (Σ, D)
2  Output: a graph G representing Hasse diagram

3  foreach a ∈ Σ do
4      L_a := 0; V_a := ∅;

5  for i := 1 to n do
6      a := w_i; V := ∅;
7      foreach b ∈ D_a in order of the last occurrence do
8          if L_b ≠ 0 and b ∉ V then
9              Insert an arc w_{L_b} → w_i into G ;
10             V := V ∪ V_b;
11     foreach b ∈ Σ do
12         V_b := V_b/{a};
13     foreach b ∈ D_a do
14         Move a to the beginning D_b;
15     V_a := V ∪ a; L_a := i;
```

be implemented using $\mathrm{O}(k^2)$ memory, we also need k pointers L. Summing up, the most significant part of this data structure is a set of lists and the memory complexity is $O(k^2)$.

The presented algorithm is online, which gives a linear factor in time complexity. Let us analyse a single step of extending the diagram with a new vertex (processing a new letter). We can see there a sequence of three loops. The first one is the most significant. We have to compute at most k sums of subsets of set Σ. It gives us a factor k^2. Every of k operations in the second loop (line 11) can be done in constant time. Furthermore, the operations in last loop (line 13) has logarithmic time complexity if we make use of priority queue but it can be implemented in constant time. Summarising, we have a complexity of $O(k^2)$ for each step of algorithm that in total gives $O(nk^2)$ time complexity for processing the whole word. See Figure 6 for the detailed step by step example of the Hasse diagram generation.

Current letter	Dependence lists	Visibility sets	Pointers	Hasse diagram
ε	$D_a = (a, b, d)$ $D_b = (b, a, c)$ $D_c = (c, b, d)$ $D_d = (d, a, c)$	$V_a = \emptyset$ $V_b = \emptyset$ $V_c = \emptyset$ $V_d = \emptyset$	$L_a = 0$ $L_b = 0$ $L_c = 0$ $L_d = 0$	
$\boldsymbol{a}$	$D_a = (a, b, d)$ $D_b = (a, b, c)$ $D_c = (c, b, d)$ $D_d = (a, d, c)$	$V_a = \{a\}$ $V_b = \emptyset$ $V_c = \emptyset$ $V_d = \emptyset$	$L_a = 1$ $L_b = 0$ $L_c = 0$ $L_d = 0$	a_1
$a\boldsymbol{d}$	$D_a = (d, a, b)$ $D_b = (b, a, c)$ $D_c = (d, c, b)$ $D_d = (d, a, c)$	$V_a = \{a\}$ $V_b = \emptyset$ $V_c = \emptyset$ $V_d = \{a, d\}$	$L_a = 1$ $L_b = 0$ $L_c = 0$ $L_d = 2$	$a_1 \longrightarrow d_2$
$ad\boldsymbol{b}$	$D_a = (b, a, d)$ $D_b = (b, a, c)$ $D_c = (b, c, d)$ $D_d = (d, a, c)$	$V_a = \{a\}$ $V_b = \{a, b\}$ $V_c = \emptyset$ $V_d = \{a, d\}$	$L_a = 1$ $L_b = 3$ $L_c = 0$ $L_d = 2$	$a_1 \longrightarrow d_2 \quad b_3$
$adb\boldsymbol{c}$	$D_a = (a, b, d)$ $D_b = (c, b, a)$ $D_c = (c, b, d)$ $D_d = (c, d, a)$	$V_a = \{a\}$ $V_b = \{a, b\}$ $V_c = \{a, b, c, d\}$ $V_d = \{a, d\}$	$L_a = 1$ $L_b = 3$ $L_c = 4$ $L_d = 2$	$a_1 \longrightarrow d_2 \quad b_3 \longrightarrow c_4$
$adbc\boldsymbol{b}$	$D_a = (b, a, d)$ $D_b = (b, a, c)$ $D_c = (b, c, d)$ $D_d = (d, a, c)$	$V_a = \{a\}$ $V_b = \{a, b, c, d\}$ $V_c = \{a, c, d\}$ $V_d = \{a, d\}$	$L_a = 1$ $L_b = 5$ $L_c = 4$ $L_d = 2$	$a_1 \longrightarrow d_2 \quad b_3 \longrightarrow c_4 \longrightarrow b_5$

Figure 6: Example of the Hasse diagram generation for the word *adbcb* using Algorithm 1.

4 Generation of All Disjoint Traces

The problem with the compressed presentation of a poset discussed in the previous sections is that it is not unique, see Remark 1. For a given ordered concurrent alphabet $(\Sigma = \{a_1 < a_2 < \ldots < a_k\}, D)$ and a word w, every other word v equivalent with w represents the same poset. To overcome this disadvantage we can use the notion of *lexicographic normal form* [5]. Basically, from all the representatives we choose the lexicographically minimal one as a normal form. All words that are in such normal form are called *canonical words*. The natural problem that arises, is to enumerate all nonequivalent words (in fact lexicographic normal forms of traces) of length n for a given concurrent alphabet. In this section we deal with this problem.

Let $\Sigma = \{a_1 < a_2 < \ldots < a_k\}$ be an ordered alphabet and X a set of words over Σ. For a word $w \in X$ we define its X-successor as the lexicographically minimal word $v \in X$ such that $v \neq w$ and $w \leq v$.

The proposed algorithm is motivated by the well known SEPA algorithm, see [7, 8]. We consider a set X of lexicographically minimal representatives of all nonequivalent traces of length n. For a given word w we identify and modify only its *working suffix* – the suffix of w which makes it different from its X-successor. We begin enumeration with lexicographically minimal word $w = a_1 a_1 \ldots a_1$. Then, we consecutively modify the current word to its X-successor. The correctness of proposed procedure follows from several corollaries to the following fact:

Proposition 1 *Let (Σ, D) be a concurrent alphabet and $<$ be a linear ordering of Σ. Then, a word $w \in \Sigma^*$ is the lexicographic normal form of a trace over (Σ, D) (is canonical) if and only if for each factor aub of w with $a, b \in \Sigma$, $u \in \Sigma^*$, $\forall_{c \in Alph(au)}(c, b) \in I$ it holds $a < b$.*

The proof of Proposition 1 can be found in [4]. For self consistency we include the following corollaries equipped with independent proofs.

Corollary 1 *If wv is a canonical word then both words w and v also are canonical. In other words prefixes and suffixes of canonical words are canonical.*

Proof: Observe that each factor of w is also a factor of wv, hence by Proposition 1, each factor aub of w with $a, b \in \Sigma$, $u \in \Sigma^*$, $\forall_{c \in Alph(au)}(c, b) \in I$ it holds $a < b$. Using Proposition 1 we achieve that w is canonical.

Note that the same arguments can be applied to any factor of wv, particularly for v.

Corollary 2 *In every canonical word w if there exists i such that letters w_i and w_{i+1} are independent then $w_i < w_{i+1}$.*

Proof: We apply Proposition 1 for $u = \varepsilon$.

Corollary 3 *If there exists a substring $w_i w_{i+1} \dots w_{j-1} w_j$ of canonical word w such that letter w_j is independent with all letters $w_i, w_{i+1}, \dots, w_{j-1}$ then w_j is the maximal amongst these letters. More precisely,*

$$\forall_{l \in \{i, i+1, \dots, j-1\}} w_j > w_l.$$

Proof: We apply Proposition 1 for $w_l w_{l+1} \dots w_{j-1} w_j$ for $i \leq l < j$.

Definition 5 *Let $a \in \Sigma$ be a letter. By $\mathbf{C}_a^n$ we denote the set of all canonical words of length n which start with the letter a.*

It is an easy observation that the set $\mathbf{C}_a^n$ is nonempty. It contains at least the word a^n. Moreover, $\mathbf{C}_a^1 = \{a\}$.

Lemma 6 *Let $w_1 \in \Sigma$ be an arbitrary but fixed letter and $w = w_1 w_2 \dots w_n$ be the lexicographically smallest word from $\mathbf{C}_{w_1}^n$ (for $n > 1$). Then the letter w_2 is the smallest letter dependent with the letter w_1 and the word $w_2 \dots w_n$ is the lexicographically smallest word from $\mathbf{C}_{w_2}^{n-1}$. Moreover, the sequence of letters $w_1, w_2, \dots, w_n$ is nonincreasing and every two consecutive letters from this sequence are dependent.*

Proof: We give the proof by induction on the length n.

Let $w \in \mathbf{C}_{w_1}^2$. Then w is of the form $w_1 w_2$, where w_2 is dependent with w_1 or strictly greater then w_1. Therefore, the smallest element of $\mathbf{C}_{w_1}^2$ is the word $w_1 w_2$, where w_2 is the smallest letter dependent with w_1 (maybe w_1 itself). Other parts of the lemma are clearly satisfied.

Let us suppose that the lemma holds for all letters and lengths smaller than k. We prove the case of letter w_1 and length k. Let us suppose, that word $w = w_1 w_2 \dots w_k$ is the lexicographically smallest word from $\mathbf{C}_{w_1}^k$. Then the letter w_2 is (similarly to the case of length 2) dependent with w_1 and not greater than w_1. Moreover, from Corollary 1 the word $w_2 \dots w_k$ is canonical. If it were not the smallest word from the set $\mathbf{C}_{w_2}^{k-1}$, we could change it to the word of such property achieving better candidate for minimum, and the proof is complete. □

The foregoing facts provide us enough information on the structure of the canonical words to design the algorithm transforming a given canonical word w into its X-successor. The algorithm consists of three steps:

1. Finding the last index i such that $w_i \neq a_k$. We know that index i is the starting position of the working suffix.
2. Computing the minimal letter a greater than w_i such that $w_1 w_2 \dots w_{i-1} a$ is canonical. It is implied by Corollary 1.
3. Generating the rest of the working suffix to obtain the minimal canonical word that starts from the letter a (at position i).

To implement the second step we introduce an oracle $V : 1 \dots n \times \Sigma \to \Sigma$. For every position i and every letter a the $V(i,a) = V_i(a)$ answers to the question - is there a substring $w_j w_{j+1} \dots w_{i-1}$ such that all letters $w_j, w_{j+1}, \dots w_{i-1}$ are independent from a and at least one letter from this substring is greater than a? In the case of positive answer, $V_i(a)$ gives the maximal witness (the maximal letter from all substring of considered property), otherwise it simply returns a. Such an oracle can be constructed in linear time (with respect to n) using the following formula:

$$\begin{aligned} V_1(a) &= a \\ V_i(a) &= \begin{cases} a & : \ aDw_{i-1} \\ \max(w_{i-1}, V_{i-1}(a)) & : \ \text{otherwise} \end{cases} \end{aligned}$$

For every letter a such that $V_i(a) = a$, the string $w_1 w_2 \dots w_{i-1} a$ is canonical.

For the efficient generation of the working suffix in step three we use a precomputed table $D_{\min}$ such that $\forall_{a \in \Sigma} \ D_{\min}(a) = \min\{b \in \Sigma : aDb\}$.

After generating a new canonical word, we have to update the oracle V. The value of $V_j(a)$ depends only on $V_{j-1}(a)$ and letter w_{j-1}. Therefore, we only have to update oracle from V_{i+1} to V_n (for the whole working suffix). Moreover, if there exists such an index l in the working suffix that $w_l = w_{l+1}$, then the rest of the suffix is constant (all foregoing letters are equal to w_l) and computation of missing oracle values are trivial ($V_{l+2} = V_{l+3} = \dots = V_n = V_{l+1}$). The example of the enumeration process described above is shown in Figure 7.

The canonical word *abeceeee* is transformed into the next canonical word *abeecbbb* in the lexicographic order. The first letter of the working suffix c is changed to e; it cannot be increased to d due to the oracle $V_4(d) \neq d$. Then the rest of the suffix is generated using $D_{\min}$ table. $D_{\min}(e) = c$, $D_{\min}(c) = b$

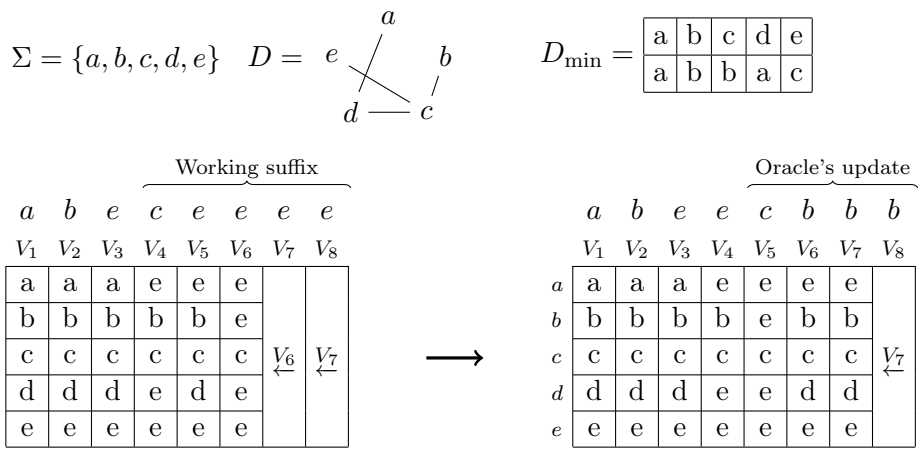


Figure 7: The example of X-successor computation.

and $D_{\min}(b) = b$. Finally the working suffix *ceeee* is transformed into *ecbbb*. The oracle V_5, V_6, V_7, V_8 is updated afterwards.

The observations mentioned above lead us to the Algorithms 2 and 3. Let us discuss their memory and time complexity. The used memory is obviously $\mathrm{O}(nk)$, mostly used for oracle V. The time complexity of steps needed for generating the next canonical word depends on the length $\#SUFF$ of the working suffix (lines from 6 to 13 of Algorithm 2). The line 6 is linear with respect to $\#SUFF$. Loop in lines 7 – 9 perform at most k iterations. The next loop (lines 10 – 11), which generates a suffix, makes exactly $\#SUFF$ operations. The most complex work is done in the last loop, which updates the oracle. At most k times the execution of the procedure Update Oracle is nontrivial and computes whole V_i. The rest of computation (at maximum $\#SUFF$ times) will end up at line 4 of the Update Oracle procedure, which can by simply implemented as a reference copying. It gives $\mathrm{O}(k^2 + \#SUFF)$ complexity of the last loop.

Note that instead of reference copying we can make use of blocks of the same letters (like in Run Length Encoding compressed representation, see [14]). Such blocks may appear also when the algorithm changes the first element of the working suffix to the last letter of the preceding prefix. Such a solution needs more careful implementation but enables the reduction of the time complexity of a single step from $\mathrm{O}(k^2 + \#SUFF)$ to $\mathrm{O}(k^2)$. Note that it makes the time complexity of a single step independent of the length

Algorithm 2: Enumerate Canonical Words

```
Input: w := a_1 a_1 ... a_1;
Output: succ(w);
for i := 1 to n do
    Update Oracle V_i;
repeat
    i := last index such that w_i ≠ a_k;
    repeat
        w_i := succ(w_i);
    until V_i(w_i) = w_i ;
    for j := i + 1 to n do
        w_j := D_min(w_{j-1}); // Generate suffix
    for j := i + 1 to n do
        Update Oracle V_j;
    OUTPUT w ;
until w = a_k a_k ... a_k ;
```

of the word. However, implementing this solution we can not forget that each non-singleton block needs two columns of the oracle.

Let us recall the example presented on Figure 7. Observe that the compressed version of working suffix is ce^4, while the resulting word is abe^2cb^3. This way we avoid the problem of reference copying and filling the suffix with a constant repeating value (letter b in the example).

If we set k as a constant enlarging only n, the time complexity of the single step of X-successor generation is $\mathrm{O}(\#SUFF)$, or $\mathrm{O}(1)$ in the compressed version, and therefore is optimal. Nevertheless, it would be very interesting to investigate the case when k is close to n. This case needs another kind of optimisation and new algorithms.

5 Summary and Future Work

In the paper we have discussed an approach to encode posets by strings. We have used concurrent alphabets and a well known notion of Hasse diagram, which might be significantly smaller than the graph of a poset. We have

Algorithm 3: Update Oracle V_i

```
1  if i = 1 then
2      foreach a ∈ Σ do V_1(a) := a;
3  else if i > 2 and w_{i-2} = w_{i-1} then
4      V_i := V_{i-1};
5  else
6      foreach a ∈ Σ do
7          if aDw_{i-1} then
8              V_i(a) := a;
9          else
10             V_i(a) := max(w_{i-1}, V_{i-1}(a));
```

shown that every poset can be represented by a pair consisting of a concurrent alphabet and a word over this alphabet. However, it is very interesting how to choose the best pair. The first criterion is the size of the concurrent alphabet (the one from the proof of Lemma 1 is taken in a very inefficient way). The second important property is preservation of N-freeness by achieving the $P4$-free dependence relation graph.

In the third section we gave an efficient online algorithm that decompress a concurrent word into a Hasse diagram. It is worth to note that the concurrent word given as an input for our algorithm does not have to be in a normal form and may be very long, as we do not have to store neither entire word nor entire diagram (only a small piece of size $\mathrm{O}(k^2)$). Moreover, utilising additional data in Algorithm 1 we are able to implement an efficient algorithm for concatenation of Hasse diagrams (over the same concurrent alphabet). The study of similar constructions for star operation would be very interesting and shall lead to an efficient algebra of posets. Such a tool would be very useful for modelling systems based on partial orders.

Section four is devoted to an algorithm which enumerates all nonequivalent strings (in the sense of dependence relation). The main idea is to construct an algorithm that is optimal (for constant size k of the alphabet) with respect to performed changes. We also present an idea of using well known compressed string representation (RLE), which results in obtaining the constant time complexity of a single step. The case of k close to n needs further work and new algorithms. Other possible directions for further

research is making use of run length encoding and considering only traces restricted to a fixed Parikh vector.

References

[1] G. Brightwell and P. Winkler. Counting linear extensions is #P-complete. In *ACM Symposium on Theory of Computing (STOC)*, pages 175–181, 1991. doi:10.1145/103418.103441.

[2] O. Cogis and M. Habib. Nombre de sauts et graphes série-parallèles. *RAIRO- Informatique théorique*, 13(1):3–18, 1979.

[3] C.J. Colbourn and W.R. Pulleyblank. Minimizing setups in ordered sets of fixed width. *Order*, 1(3):225–229, 1985. doi:10.1007/BF00383598.

[4] V. Diekert. *Combinatorics on Traces*, volume 454 of *Lecture Notes in Computer Science*. Springer, 1990. doi:10.1007/3-540-53031-2.

[5] V. Diekert and Y. Métivier. *Partial commutation and traces*, pages 457–533. Springer-Verlag, New York, USA, 1997. doi:10.1007/978-3-642-59126-6_8.

[6] V. Diekert and G. Rozenberg (editors). *The Book of Traces*. World Scientific, Singapore, 1995.

[7] D.E. Knuth. *The Art of Computer Programming, Volume 3*. Addison-Wesley, Reading, 1973.

[8] D.E. Knuth. *The Art of Computer Programming: Volume 4, Fascicle 3. Generating All Combinations and Partitions*. Addison-Wesley, 2005.

[9] D. Kuske. Infinite series-parallel posets: Logic and languages. *Lecture Notes in Computer Science*, 1853:648–662, 2000. doi:10.1007/3-540-45022-X_55.

[10] A.B. Kwiatkowska and M.M. Sysło. On page number of N-free posets. *Electronic Notes in Discrete Mathematics*, 24:243 – 249, 2006. Fifth Cracow Conference on Graph Theory USTRON '06. doi:10.1016/j.endm.2006.06.030.

[11] A. Mazurkiewicz. Concurrent program schemes and their interpretations. DAIMI Report PB-78, Aarhus University, 1977.

[12] Ł. Mikulski. Projection representation of Mazurkiewicz traces. *Fundamenta Informaticae*, 85:399–408, 2008.

[13] Ł. Mikulski, M. Piatkowski and S. Smyczynski. Algorithmics of posets generated by words over partially commutative alphabets. In *Proceedings of the Prague Stringology Conference, Prague, Czech Republic, 2011*, pages 209–219, 2011.

[14] K. Sayood. *Introduction to Data Compression*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1996.

[15] M.M. Sysło. Minimizing the jump number for partially ordered sets: A graph-theoretic approach. *Order*, 1:7–19, 1984. doi:10.1007/BF00396269.

[16] K. Takamizawa, T. Nishizeki and N. Saito. Linear-time computability of combinatorial problems on series-parallel graphs. *Journal of the ACM*, 29(3):623–641, 1982. doi:10.1145/322326.322328.

[17] J. Valdes. *Parsing Flowcharts and Series-Parallel Graphs*. Ph.D. dissertation, Stanford University, Stanford, 1978.

[18] J. Valdes, R.E. Tarjan and E.L. Lawler. The recognition of series parallel digraphs. In *ACM Symposium on Theory of Computing (STOC)*, pages 1–12, 1979. doi:10.1145/800135.804393.