Algorithmics of Posets Generated by Words over Partially Commutative Alphabets*

Łukasz Mikulski, Marcin Piątkowski, and Sebastian Smyczyński

Faculty of Mathematics and Computer Science
Nicolaus Copernicus University
Chopina 12/18, 87-100 Toruń, Poland
frodo@mat.umk.pl, martinp@mat.umk.pl, smyczek@mat.umk.pl

Abstract. It is natural to try 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 will investigate this relation in detail and propose an efficient on-line algorithm that decompresses a string to Hasse diagram. Further we propose a definition of the canonical representatives of classes of equivalent words. The advantage of this representation lies in the fact that we can enumerate the classes of equivalent words in a lexicographical order. We will give an algorithm which enumerates all distinct classes of words over partially commutative alphabets by their lexicographically minimal representatives.

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

Introduction

Many practical problems related to partially ordered sets have a very high time of computation. The examples of very hard tasks are #P-complete problem of counting number of posets linearisations [1] or NP-complete problem of computing the minimal number of jumps [10]. From less complex problems we can provide a problem of computing transitive reduction of a poset graph which has cubic time complexity.

One of the main reasons for such a situation is the dependence of the complexity exclusively on the number of elements of a poset. We show a stringologic approach to the posets that uses words over partially commutative alphabets and allows us to exploit the inner structure of a given poset. As a result, we achieve algorithms with complexity dependent not only on the number of elements but also on the size of the concurrent alphabet.

In the first section, we give some basic notions related to the formal languages theory, partial orders and the concurrent systems modeling. In Section 2 we will look more closely at the connections between words over semi-commutative alphabets, their dependency graphs and Hasse diagrams and graphs of partial orders. In the following section we will deal with decoding Hasse diagrams from strings and give an $O(nk^2)$ complexity algorithm. Here and subsequently n denotes the size of the poset and k – the size of the (possibly significantly smaller) alphabet.

The studies on the properties of words over partially commutative alphabets require an efficient tool for enumeration of distinct classes of equivalent words (in the sense of the independency relation). In the fourth section we deal with this practical

* The research partially supported by Ministry of Science and Higher Education of Poland, grant N N206 258035.

Łukasz Mikulski, Marcin Piątkowski, Sebastian Smyczyński: Algorithmics of Posets Generated by Words over Partially Commutative Alphabets, pp. 209–219.
Proceedings of PSC 2011, Jan Holub and Jan Žďárek (Eds.), ISBN 978-80-01-04870-2 © Czech Technical University in Prague, Czech Republic

problem. The solution is motivated by a well known SEPA algorithm [4,5] and has similar complexity of a single step. Basically we will identify classes of equivalent words with their lexicographically smallest representatives. Those representatives are called *canonical*. Further we will show how to compute the considered representatives of all classes in the lexicographical order. The single step of this computation is based on the function which for a given class returns the next class in an order implied by the lexicographical order of their canonical representatives.

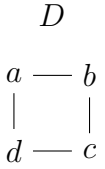
1 Basic notions

We use some basic notions of formal languages theory. By Σ we denote an arbitrary finite set, called *alphabet*. Elements of the alphabet are called *letters*. *Words* are sequences over the alphabet Σ. The sets of all finite words will be denoted by Σ^*.

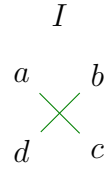
The *concurrent alphabet* is a pair (Σ,D), where Σ is a common alphabet (finite set) and $D \subseteq \Sigma \times \Sigma$ is an arbitrary reflexive and symmetric relation, called *dependency relation*. With dependency we associate, as another relation, an *independency relation* $I = \Sigma \times \Sigma \setminus D$. Having the concurrent alphabet we define a relation that identifies similar words. We say that word $\sigma \in \Sigma^*$ is in relation $\equiv_D$ with word $\tau \in \Sigma^*$ if and only if there exists a finite sequence of commutation of subsequent, independent letters that leads from σ to τ. Relation $\equiv_D \subseteq \Sigma^* \times \Sigma^*$ is a congruence relation (whenever it will not be confusing, relation symbol D will be omitted).

After dividing set Σ^* by the relation $\equiv$ we get a quotient monoid. The elements of $\Sigma^*/_{\equiv}$ are often called *traces* (see [3,8,9]). This way, every word σ is related to a trace $\alpha = [\sigma]$, containing this word.

Example 1. To the alphabet $\Sigma = \{a, b, c, d\}$ we add a dependency relation



or, equivalent, an independency relation

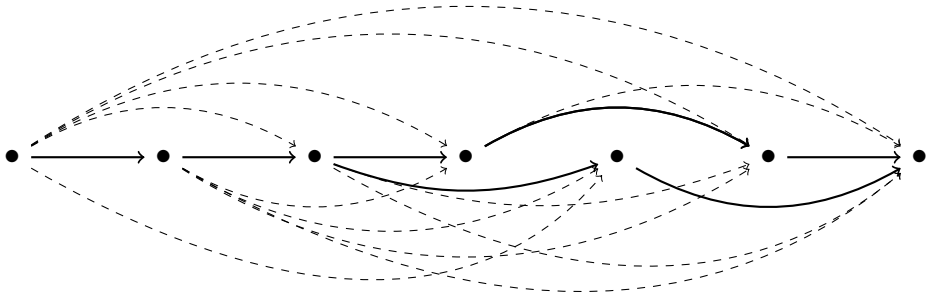


Words *abbaacd* and *abbcaad* are equivalent.

The *partial order* in a set X is a relation $\leq \subseteq X \times X$, such that $\leq$ is reflexive, antisymmetric and transitive. Pair $(X,\leq)$ is called *partially ordered set*, or shortly *poset*. With every poset we can associate its directed graph (digraph in short) $G = (X, E)$. The vertices are the elements of the poset. There is an edge between two vertices $x, y \in X$ if and only if $x < y$ (ie. $x \leq y$ but $x \neq y$). Such a graph is always acyclic. We can also define a Hasse diagram of poset $(X,\leq)$ by the transitive reduction of graph G.

Definition 2. *Let $G = (X, E)$ be an acyclic graph. The* Hasse diagram *of graph G i acyclic graph $H = (X, E' \subseteq E)$, such that an edge $(x, y) \in E'$ if and only if $(x, y) \in E$ and if there is $z \in X$ such that there are both path from x to z and from z to y then $x = z$ or $y = z$.*

Example 3. The graph of a poset. The dashed edges are not contained in Hasse diagram.



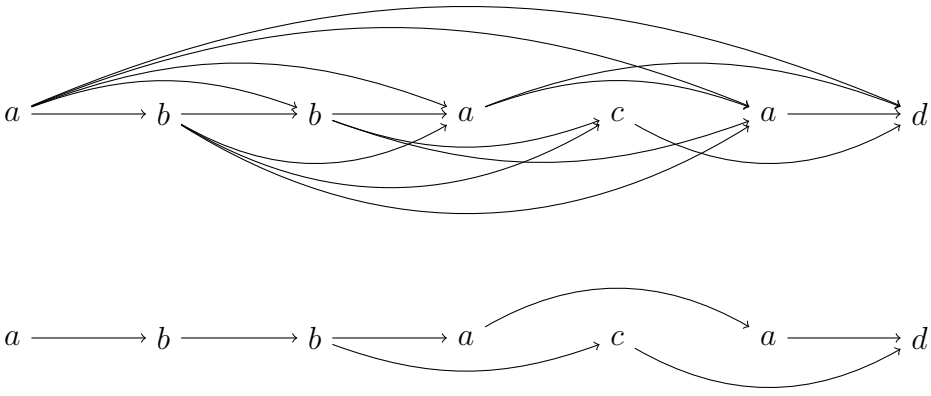
2 From partially commutative words to posets

With every word w over partially commutative alphabet (Σ, D) we can associate a poset. The preorder of this poset is induced by the dependency graph of a word w. A letter w_j is greater than a letter w_i if and only if $i < j$ and $w_i D w_j$. It is worth noting that two words are equivalent if and only if their dependency graphs are the same (isomorphic and respecting labelling).

Reflexive transitive closure of dependency graph of a word is basically a graph of a poset associated with the word. We can represent it by the graph of its transitive reduction, called Hasse diagram.

Example 4. A concurrent alphabet (Σ, D), dependency graph and Hasse diagram of word *abbacad* over that alphabet.

$$\Sigma = \{ a, b, c, d\} \qquad D = \begin{array}{ccc} a & - & b \\ | & & | \\ d & - & c \end{array}$$



Lemma 5. *Every poset $(P, \leq)$ can be generated by a word over concurrent alphabet.*

Proof. For given 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 abitrary linearisation of poset $(P, \leq)$ corresponds in 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.

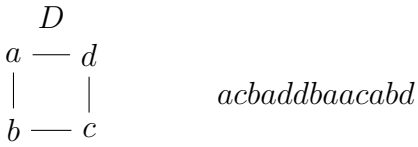
Further optimisation, possible only for Hasse diagrams which are minimal series-parallel graphs [14], lead us to another data structure which can be used to solve many problems in a simpler way (for instance #P-complete problem of counting number of linear expansions [1] is linear for such posets).

Definition 6. *Minimal Series-Parallel digraph (MSP) is a graph consisting of a single vertex and no edges or is constructed from two MSP – $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$ – by the following operations:*

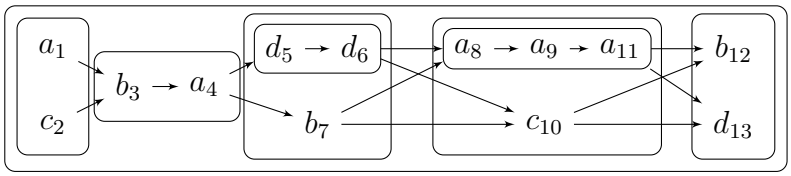
- *Parallel composition: $G_P = (V_1 \cup V_2, E_1 \cup E_2)$;*
- *Serial composition: $G_P = (V_1 \cup V_2, E_1 \cup E_2 \cup T_1 \times S_2)$;*

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.

Example 7. The dependent alphabet D, the word w and its Hasse diagram divided to series-parallel blocks.



acbaddbaacabd

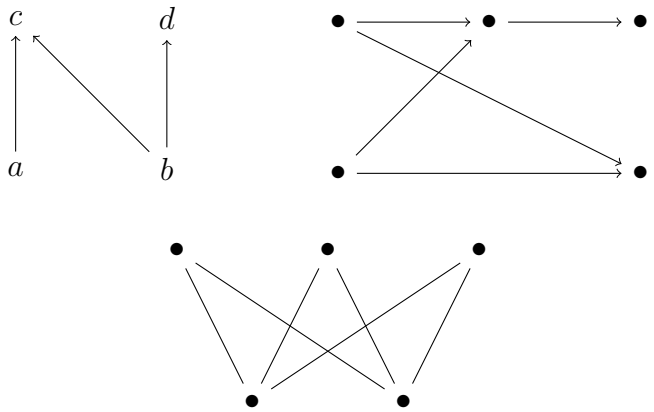


The properties of series-parallel graphs are deeply studied (see for instance [2,11,14]). A very useful determinant for sequential parallel graphs is their N-freeness [13].

Definition 8. *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).[7]*

Definition 9. *N-free posets are posets whose Hasse diagrams do not contain an induced subgraph isomorphic with Hasse diagram of N-poset.*
In the case of undirected graphs, analogue is P4-free graph (a graph that does not contain induced path of length 3).

Example 10. N-poset, simple N-free poset and P4-free graph



In general, this type of graphs, also in context of partial orders, is deeply studied (see [6,12,14] and the references therein). However, observations worth mentioning are the following:

Lemma 11. *If a dependency graph D of an alphabet Σ is $P4-free$ then Hasse diagram of every partially commutative word $w \in \Sigma^*$ is $N-free$.*

The above discution gives another motivation for studies over efficient algorithms that constructs Hasse diagram.

3 Construction of Hasse diagram

This section is devoted to the problem of constructing the Hasse diagram (see Definition 2) for a given word over a concurrent alphabet (which is a transitive reduction of some poset). 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 summarize it in the following facts:

Lemma 12. *Let $w \in \Sigma^*$ be a word and $H(w) = (V, E_H)$ be a Hasse diagram of w. If there exists the edge connecting vertices labeled $w_i = a$ and $w_j = b$ then letters a and b do not appear in word w between indexes i and j.*

Proof. Let us denote the dependence digraph $G = (V, E)$ of a word w over concuurent alphabet (Σ, D). The existance of an edge between w_i and w_j in graph H implies that there is also an edge in graph G, hence letters a and b are dependent (formally aDb). Let us suppose that there exists a letter $w_k = c$ (for $i < k < j$) that is dependent both with a and b. Then by the Definition 2 there is a path in graph G between vertices w_i and w_j of length longer than one, so there is no edge between w_i and w_j in graph $H(w)$, which provides to contradiction and completes the proof. □

Lemma 13. *Let $w \in \Sigma^*$ be a word and $H(w)$ be a Hasse diagram of w. For each vertex there are no more than $k = |\Sigma|$ outgoing edges and no more than k ingoing edges.*

Proof. Let us denote the dependence digraph $G = (V, E)$ of a word w over concuurent alphabet (Σ, D). Let us suppose that there is a vertex w_i which has $k + 1$ outgoing edges. There are k letters in alphabet Σ, so two of these outgoing edges lead to two distinct vertices w_j and w_k $(i < j < k)$ labelled with the same letter. Withaout loss of genrality we can assume that $w_i = a$ and $w_j = w_k = b$. From the Lemma 12 there is no edge in graph $H(w)$ between vertices w_i and w_k, which proves that there are at most k outgoing edges. Similar reasonig allows us to prove second part of thesis and limit the number of ingoing edges. □

Lemma 14. *Let $w \in \Sigma^*$ be a word and $H(w) = (V, E)$ be a Hasse diagram of w. Ingoing edges of the given vertex v are fully determined by the last occurrences of the letters dependent with $v = w_i$. More formally, $(w_j, w_i = v) \in E$ if and only if $j < i$ and there is no vertex $w_k D w_i$ such that $j < k < i$ and there is a path from w_j to w_k.*

Proof. Let $(w_j = a, w_i)$ be an edge in $H(w)$. Lemma 12 implies that w_j must be the last occurrence of letter a in word w that preceedes w_i. Second part of the thesis follows direclty from the Definition 2 (see proof of the Lemma 12). □

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 new vertex to Hasse diagram, with all of its ingoing edges. Our structure consists of a list of dependency, a set of visibility (both of size at most k) and a pointer to 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. Set V_a contains all letters b whose last occurrences are visible from the last vertex labeled with a. In other words, there exists a path from $v_i = b$ to $v_j = a$ where v_i and v_j represent the last occurrences of those letters 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 labeled with the letter a in hitherto diagram. We will also use a temporary set V.

Before we start generating a Hasse diagram we set all pointers to *null* and all sets to be empty. The lists of dependences 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 on-line manner, updating the proposed structure after each step and creating a new vertex and edges. During adding the new vertex labeled with the 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 (is not already visible from new vertex). If we succeed, we add a new edge from the vertex pointed by L_b to the newly created vertex. Addition of a new edge implies that there is also a path from every source of 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 dynamically changing set V.

After adding new edges, 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 so it can not be a source at all. 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.

Algorithm 1: Hasse diagram

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

The correctness of the algorithm presented above bases 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. It gives us k^2 elements. The k sets V can be implemented using $k \log k$ memory, we also need k pointers L. Summing up, the most significant part of this data structure is a set of list and the memory complexity is $O(k^2)$.

The presented algorithm is on-line, which gives a linear factor in time complexity. Let us analyze 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. The operation in the second loop (line 10) can be done in constant time. Furthermore, the operation in last loop (line 12) has logarithmic time complexity if we make use of priority queue but can be implemented in constant time. Summarizing, 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.

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. For a given ordered concurrent alphabet $(\Sigma = \{a_1 < a_2 < \cdots < a_k\}, D)$ 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 *canonical representative*. Basically, from all the representatives we choose the lexicographically minimal one as a canonical representative. All words that are canonical representatives of a trace are called *canonical words*. Obviously, all the words of the length not greater then one are canonical. The natural problem that arises, is to enumerate all nonequivalent words (in fact canonical representatives of traces) of length n for a given concurrent alphabet. In this section we deal with this problem.

The proposed algorithm is motivated by the well known SEPA algorithm. We identify and modify only the *working suffix* – the suffix of the given canonical word which makes it different from its successor in the lexicographic order. We begin enumeration with lexicographically minimal word $w = a_1 a_1 \cdots a_1$. Then, we consecutively modify

the current word to its successor in lexicographic order, skipping all noncanonical ones. To perform this enumeration effectively we will use the following facts:

Lemma 15. *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. Let us suppose that word w is not canonical. Then there is a lexicographical smaller and equivalent with respect to the relation $\equiv_D$ word w'. From the equivalence of words w and w' we conclude that also words wv and $w'v$ are equivalent. The word $w'v$ is lexicographically smaller then word wv. Therefore the word wv cannot be canonical. Simlar argumentation shows that v is also a canonical word. □

Lemma 16. *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. Suppose that $w_i \geq w_{i+1}$. If they are equal then by the definition they are dependent which contradicts the assumption of their independence, hence $w_i > w_{i+1}$. We can assume that $w = uw_iw_{i+1}v$ which is equivalent with respect to the relation $\equiv_D$ to the word $uw_{i+1}w_iv$ that is lexicographically smaller than w. That is in conflict with the assumption of cononicality of the word w and completes the proof. □

Lemma 17. *If there exists a substring $w_iw_{i+1}\cdots w_{j-1}w_j$ of canonical word w such that letter w_j is independent with all letters $w_i, w_{i+1}, \ldots, w_{j-1}$ then w_j is the maximal amongst these letters. More precisely,*

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

Proof. Let us denote the word $w = uw_i\cdots w_jv$ and suppose that one of the letters $w_k \in \{w_i, w_{i+1}, \ldots, w_{j-1}\}$ is smaller then w_j. Then, from the independence of each of these letters with w_j we have an equivalent with respect to the relation $\equiv_D$ to w word $w' = uw_i\cdots w_{k-1}w_jw_k\cdots w_{j-1}$. Obviously the word w' is lexicographically smaller then w, hence w cannot be a canonical word. □

Lemma 18. *If there exists a substring $w_iw_{i+1}\cdots w_{j-1}w_j$ of canonical word w such that letter w_j is independent with all letters $w_{i+1}, \ldots, w_{j-1}, w_j$ then w_i is the minimal amongst these letters. More precisely,*

$$\forall_{l\in\{i+1,\ldots,j-1,j\}} w_i < w_l.$$

Proof. Proof of the lemma is similar to the proof of Lemma 17. □

Definition 19. *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 20. *Let $w_1 \in \Sigma$ be an arbitrary but fixed letter and $w = w_1w_2\cdots w_n$ 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 $w_2\cdots w_n$ is the lexicographically smallest word from $\mathbf{C}_{w_2}^{n-1}$. Moreover, the sequence of letters $w_1, w_2, \ldots, w_n$ is nonencreasing and every two consecutive letters from this sequence are dependent.*

Proof. We give the proof by induction on the length n.
Let $w \in \mathbf{C}^2_{w_1}$. Then w is of the form w_1w_2, where w_2 is dependent with w_1 or strictly greater then w_1. Therefore, the smallest element of $\mathbf{C}^2_{w_1}$ is the word w_1w_2, where w_2 is the smallest letter dependent with w_1 (maybe w_1 itself). Other parts of the thesis are clearly satisfied.

Let us suppose that the thesis holds for all letters and lengths n smaller than k. We prove the case of letter w_1 and length k. Let us suppose, that word $w = w_1w_2 \cdots w_k$ is the lexicographically smallest word from $\mathbf{C}^k_{w_1}$. Then the letter w_2 is (similary to the case of length 2) dependent to w_1 and not greater then w_1. Moreover, from Lemma 15 the word $w_2 \cdots w_k$ is canonical. If it would not be the smallest word from the set $\mathbf{C}^{k-1}_{w_2}$, we could change it to the word of such property achieving better candidate for minimum, and the proof is complete. □

The foregoing lemmas provide us enough information of the structure of the canonical words to design the algorithm transforming given canonical word w into its 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 $w_1w_2 \cdots w_{i-1}a$ is canonical. It is implied by Lemma 15.
3. Generating the rest of the working suffix to obtain the minimal canonical word that starts from w_i.

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

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

For every letter a such that $V_i(a) = a$, the string $w_1w_2 \cdots w_{i-1}a$ is canonical.

For the efficient generation of the working suffix in step three we will 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 index l in the working suffix that $w_l = w_{l+1}$, then the rest of the suffix is constant (all letters equal to w_l) and computation of missing oracle values are trivial ($V_{l+2} = V_{l+3} = \cdots = V_n = V_{l+1}$).

Algorithm 2: Enumerate Canonical Words

```
1  w := a_1 a_1 ··· a_1;
2  OUTPUT w;
3  for i := 1 to n do
4      Update Oracle V_i;
5  repeat
6      i := last index such that w_i ≠ a_k;
7      repeat
8          w_i := succ(w_i);
9      until V_i(w_i) = w_i;
10     for j := i + 1 to n do
11         w_j := D_min(w_{j-1}); // Generate suffix
12     for j := i + 1 to n do
13         Update Oracle V_j;
14     OUTPUT w ;
15 until w = a_k a_k ··· a_k;
```

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));
```

The observations mentioned above lead us to the Algorithms 2 and 3. Let us discuss the memory and time complexity. The used memory is obviously $O(nk)$, 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 $O(k^2 + \#SUFF)$ complexity of the last loop.

If we set k as a constant enlarging only n, the time complexity of the single step of successor generation is $O(\#SUFF)$ and is therefore optimal. Nevertheless, it would be very interesting to investigate the case when k is close to n. Probably this case needs another kind of optimization and new algorithms.

5 Summary and future work

In the paper we have discussed an approach to encode posets by strings. We used concurrent alphabets and well known notion of Hasse diagram, which is significantly smaller than the graph of a poset. We have 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 is taken in a very inefficient way). The second important property is preservation of N-freeness by achieving the P4-free dependency relation graph.

In the third section we gave an efficient algorithm that enables us to decompress a word into a Hasse diagram. Extending those results, we would like to equip Hasse diagrams (using additional data structures) with efficient concatenation and star operations.

Section four is devoted to the algorithm which enumerates all nonequivalent strings (in the sense of dependency relation). The main idea is motivated by SEPA algorithm. However, the innovative idea of using oracle allows us to construct an algorithm that is optimal (for constant size k of the alphabet) with respect to performed changes. The case of k close to n needs further work and new algorithms.

We believe that our results will have many theoretical and practical applications. For example, the extending of the results related to Hasse diagram may be very useful in verification of models, where partial orders or concurrent words play a key role.

References

1. G. Brightwell and P. Winkler: *Counting linear extensions is #P-complete*, in STOC: ACM Symposium on Theory of Computing (STOC), 1991.
2. O. Cogis and M. Habib: *Nombre de sauts et graphes série-parallèles.* ITA, 13(1) 1979.
3. V. Diekert and G. Rozenberg, eds., *The Book of Traces*, World Scientific, Singapore, 1995.
4. D. E. Knuth: *The Art of Computer Programming, Volume 3*, Addison-Wesley, Reading, 1973.
5. D. E. Knuth: *The Art of Computer Programming: Volume 4, Fascicle 3. Generating All Combinations and Partitions*, Addison-Wesley, 2005.
6. D. Kuske: *Infinite series-parallel posets: Logic and languages.* Lecture Notes in Computer Science, 1853 2000, pp. 648–662.
7. A. B. Kwiatkowska and M. M. Sysło: *On page number of n-free posets.* Electronic Notes in Discrete Mathematics, 24 2006, pp. 243 – 249, Fifth Cracow Conference on Graph Theory USTRON '06.
8. A. Mazurkiewicz: *Concurrent program schemes and their interpretations*, daimi report pb-78, Aarhus University, 1977.
9. Ł. Mikulski: *Projection representation of Mazurkiewicz traces.* Fundamenta Informaticae, 85 2008, pp. 399–408.
10. W. R. Pulleyblank: *On minimizing setups in precedence constrained scheduling*, Tech. Rep. 81185-OR, Univ. Bonn, Inst. f. Ökonometrie und OR, Bonn, 1981.
11. M. M. Sysło: *Minimizing the jump number for partially ordered sets: A graph-theoretic approach.* Order, 1 1984, pp. 7–19.
12. K. Takamizawa, T. Nishizeki, and N. Saito: *Linear-time computability of combinatorial problems on series-parallel graphs.* Journal of the ACM, 29(3) 1982, pp. 623–641.
13. J. Valdes: *Parsing Flowcharts and Series-Parallel Graphs*, Ph.D. dissertation, Stanford University, Stanford, 1978.
14. J. Valdes, R. E. Tarjan, and E. L. Lawler: *The recognition of series parallel digraphs*, in Eleventh Annual ACM Symposium on Theory of Computing (STOC '79), New York, Apr. 1979, ACM, pp. 1–12.