Space-Efficient Data Structure for Posets with Applications

Tatsuya Yanagita
The University of Tokyo, Japan

Sankardeep Chakraborty
The University of Tokyo, Japan

Kunihiko Sadakane
The University of Tokyo, Japan

Srinivasa Rao Satti
Norwegian University of Science and Technology, Trondheim, Norway

Abstract

Space efficient data structures for partial ordered sets or posets are well-researched field. It is known that a poset with n elements can be represented in $n^2/4 + o(n^2)$ bits [30] and can also be represented in $(1+\epsilon)n\log n + 2nk + o(nk)$ bits [19] where k is width of the poset. In this paper, we make the latter data structure occupy $2n(k-1) + o(nk)$ bits by considering topological labeling on the elements of posets. Also considering the topological labeling, we propose a new data structure that calculates queries on transitive reduction graphs of posets faster though queries on transitive closure graphs are computed slower. Moreover, we propose an alternative data structure for topological labeled posets that calculates both of the queries faster though it uses $3nk - 2n + o(nk)$ bits of space. Additionally, we discuss the advantage of these data structures from the perspective of an application for BlockDAG, which is a more scalable version of Blockchain.

2012 ACM Subject Classification Mathematics of computing → Graph theory

Keywords and phrases Succinct Data Structures, Posets, Blockchain

Digital Object Identifier 10.4230/LIPIcs.SWAT.2022.33

1 Introduction

We consider space-efficient data structures for partially ordered sets (posets). Such data structures have been studied earlier. For arbitrary posets, Munro and Nicholson [30] proposed a data structure whose space complexity is $n^2/4 + \mathrm{o}(n^2)$ bits, and this matches the lower bound $n^2/4 + 3n/2 + \mathrm{O}(\log n)$ which is found by Kleitman and Rothschild [29]. When the poset has Dilworth's width [16] k[1], Daskalakis et al. [15] showed that posets can be represented in $\mathrm{O}(nk)$ words where n is the number of elements of the poset. Using their idea, Farzan and Fischer [19] developed a data structure for posets with $(1+\epsilon)n\log n + 2nk + o(nk)$ bits, for any positive constant $\epsilon \leq 1$. It is known that

$$\frac{n!}{k!}4^{n(k-1)}n^{-24k(k-1)} \leq N_k(n) \leq n!\,4^{n(k-1)}n^{-\frac{(k-1)(k-2)}{2}}k^{\frac{k(k-1)}{2}}$$

when $N_k(n)$ is the number of posets on n elements with width k [10]. Hence, the information theoretic lower bound is $n\log n + 2n(k-1) - \Theta(k^2\log n)$ bits. Thus the data structure of Farzan and Fischer is succinct if $k = \mathrm{o}(\sqrt{n})$ and $\epsilon = \mathrm{o}(1)$.

There are a lot of other related works. It is well-known as Birkhoff's representation theorem that posets have one-to-one correspondence with distributive lattices [8]. There is a data structure for distributive lattices [32] whose size is close to the lower bound given by

[1] Hereafter we denote Dilworth's width by just *width*.

Table 1 Comparison of data structures for posets. The data structure in Section 2.5 is the original version of Farzan and Fischer's data structure [19] and the one in Section 3.2 is a modified version of that. G_C is a transitive closure graph of a poset and G_R is a transitive reduction graph of it. Here $o(nk)$ denotes $n \cdot o(k) + o(n) \cdot k$, and t denotes the size of the output of a query.

	Section 2.5	Section 3.2	Section 3.3	Section 3.4
Labeling	General	Topological	Topological	Topological
Space [bits]	$(1+\epsilon)n \log n$ $+2n(k-1) + o(nk)$	$2n(k-1) + o(nk)$	$2nk + o(nk)$	$3nk - 2n + o(nk)$
$\mathrm{adj}_{G_C}(u, v)$	$O(1/\epsilon)$	$O(\log \log k)$	$O(k^2)$	$O(\log \log k)$
$\mathrm{succ}_{G_C}(v)$	$O(1/\epsilon + k + t)$	$O(k + t)$	$O(k^2 + t)$	$O(k + t)$
$\mathrm{pred}_{G_C}(v)$	$O(1/\epsilon + k + t)$	$O(k + t)$	$O(k^2 + t)$	$O(k + t)$
$\mathrm{adj}_{G_R}(u, v)$	$O(1/\epsilon + k)$	$O(k)$	$O(\log \log k)$	$O(\log \log k)$
$\mathrm{succ}_{G_R}(v)$	$O(1/\epsilon + k^2)$	$O(k^2)$	$O(k)$	$O(\log \log k + t)$
$\mathrm{pred}_{G_R}(v)$	$O(1/\epsilon + k^2)$	$O(k^2)$	$O(k)$	$O(k)$

Erne, Heitzig and Reinhold [18]. There are also succinct/compact data structures which can represent arbitrary binary relations [6, 7], interval graphs, chordal graphs and finally arbitrary graphs among many others [1, 2, 12, 13, 20, 33]. Recently, a new parameter called twin-width, which can be defined on posets, graphs and more generally matrices, was introduced by Bonnet et al. [9] and it is shown that the twin-width of posets is linear in their Dilworth's width [5]. A compact data structure for matrices with fixed twin-width was proposed [36].

1.1 Main Results

We denote by n the number of elements in a poset and its width by k. We first show that by considering not generally labeled posets but topologically labeled posets, the space complexity of Farzan and Fischer's data structure can be reduced to $2n(k-1) + o(nk)$ bits (Section 3.2). This is succinct for non-constant $k = o(\frac{n}{\log n})$ (see Section 3.1). Also, we propose two alternative data structures to store topologically labeled posets. Our first data structure can support queries on transitive reduction graphs (defined in Section 2.2) faster than their data structure though queries on transitive closure graphs are slower (Section 3.3). The other data structure can support the queries on both the transitive reduction and transitive closure graphs faster though the space increases to $3nk - 2n + o(nk)$ bits (Section 3.4).

Table 1 shows the comparison of time complexity of queries among the modified version of Farzan and Fischer's data structure in Section 3.2, the proposed data structure in Section 3.3 and another one in Section 3.4.

Our other contributions are to compress some of these data structures into less space (Section 3.5) and to make these data structures dynamic (Appendix B). The dynamic versions support the operation of adding an element to the poset.

1.2 Application

Blockchain is a technology for a public ledger of cryptocurrencies like Bitcoin, and was proposed by Satoshi Nakamoto in 2009 [34]. In Blockchain, blocks storing the data of transactions are linked in a chain, which has a scalability problem. In order to solve this problem, some protocols called *BlockDAG* are proposed in which the blocks are connected in a directed acyclic graph (DAG) instead of a chain. Typical BlockDAG protocols are PHANTOM and GHOSTDAG [37].

Let $G = (V, E)$ be a DAG where $V = \{1, 2, \ldots, n\}, E \subseteq V \times V$. The DAG class of BlockDAG satisfies the conditions below:

1. $\forall (u, v) \in E, v < u$;
2. Only one sink (i.e., a vertex with no outgoing edges); and
3. If $(u, v_1), (u, v_2) \in E$, then there is no path from v_1 to v_2.

In Condition 1, $<$ is the standard total order on V, i.e. $1 < 2 < \ldots < n$. We call DAGs that satisfy Condition 1 as *topologically labeled DAGs*. Conditions 1 and 2 imply that the node labeled 1 is the sink and the DAG is connected. DAGs which satisfy Condition 3 are called transitive reduction graphs [3]. If we remove the sink from G, then the DAG class has one-to-one correspondence to posets labeled by a topological order.

For a Blockchain, its corresponding graph is a chain (path), which has width 1. For a BlockDAG, it is expected that the graph has small width k. It is therefore worth giving a space-efficient representation for graphs with small width.

2 Preliminaries

2.1 Chain Decomposition

A *partially ordered set* or a *poset* is a set with a binary relation which is reflexive, antisymmetric and transitive. We denote a poset by $\mathcal{P} = (V, \preceq)$ where $V = \{1, 2, \ldots, n\}$ and $\preceq$ is the relation. For any $a, b, c \in V$, the following holds:

- reflexivity: $a \preceq a$,
- antisymmetry: if $a \preceq b$ and $b \preceq a$, then $a = b$,
- transitivity: if $a \preceq b$ and $b \preceq c$, then $a \preceq c$.

When $u \preceq v$ and $u \neq v$, we denote it by $u \prec v$. We say that a poset is *topologically labeled* if $u \preceq v \Rightarrow u \leq v$ for any $u, v \in V$ where $\leq$ is the standard total order on V. In 1950, Dilworth showed the duality between chains and antichains of posets [16].

▶ **Definition 1** (Antichain). *$A \subseteq V$ is an antichain of $\mathcal{P}$ if any two elements in A are unordered on $\mathcal{P}$.*

▶ **Definition 2** (Chain). *$C \subseteq V$ is a chain of $\mathcal{P}$ if C is totally ordered on $\mathcal{P}$.*

▶ **Definition 3** (Chain Decomposition). *A set of disjoint sets $\{C_p\}_{p=0}^{k'-1}$ is a chain decomposition of $\mathcal{P}$ if C_p is a chain of $\mathcal{P}$ for all $p \in \{0, 1, \ldots, k'-1\}$ and $\bigcup_{p=0}^{k'-1} C_p = V$.*

▶ **Theorem 4** (Dilworth's Theorem [16]). *The maximum size of an antichain is equal to the minimum number of chains in any chain decomposition.*

The maximum size of antichain is called *Dilworth's width* or simply *width* and we denote it by k. Fulkerson found that Dilworth's theorem is equivalent to König's theorem and the minimal chain decomposition can be obtained by solving a maximum matching on a bipartite graph [23]. It can be solved in $O(n^{2.5})$ time by using Hopcroft and Karp's algorithm [27]. The time complexity to calculate the minimal chain decomposition can be reduced to $O(kn^2)$ [14, 15].

2.2 Transitive Closure Graphs vs. Transitive Reduction Graphs

Transitive closure and transitive reduction are well-researched topics since 1970s. Given a poset $\mathcal{P}$, we define its transitive closure graph and transitive reduction graph as follows.

▶ **Definition 5** (Transitive Closure Graph). *A transitive closure graph of $\mathcal{P}$ is a DAG $G_C = (V, E_C)$ where $E_C = \{(u, v) \in V \times V; v \prec u\}$.*

▶ **Definition 6** (Transitive Edge). *A transitive edge of a DAG $G = (V, E)$ is an edge $(u, v) \in E$ such that there exists a path from u to v other than the path going through the edge (u, v).*

▶ **Definition 7** (Transitive Reduction Graph). *A transitive reduction graph of $\mathcal{P}$ is a DAG $G_R = (V, E_R)$ where E_R is a set such that all the transitive edges are removed from E_C.*

Aho, Garey and Ullman generalized the transitive reduction to binary relations [3]. If the binary relation is antisymmetric, the transitive reduction graph is unique. Thus, the transitive reduction graph of a poset is unique. They also proved that both directions of the conversions between a transitive closure graph and a transitive reduction graph have algorithms of same time complexity.

The conversions between a transitive reduction graph and a transitive closure graph can be obtained in $O(n^{2.37286})$ time by using boolean matrix multiplication algorithm [4, 21]. They can also be calculated in $O(nm_R + m_C)$ time [25] where $m_R = |E_R|$ and $m_C = |E_C|$, and this time complexity is $O(kn^2)$ since $m_R \leq kn$ and $m_C \leq n(n-1)/2$.

Definition 7 is essentially equivalent to Condition 3 of the definition of BlockDAG in Section 1.2. The transitive reduction graphs are essentially same as Hasse diagrams of posets. A poset can be represented either by its transitive closure graph or its transitive reduction graphs.

2.3 Queries on Posets

In this section, we introduce the queries that are supported by our data structures for posets, or transitive closure/reduction graphs. We consider the following six queries:

- $\mathrm{adj}_{G_C}(u, v) \cdots$ return 1 if $(u, v) \in E_C$ and otherwise 0;
- $\mathrm{succ}_{G_C}(v) \cdots$ return the set $\{u \in V; (v, u) \in E_C\}$;
- $\mathrm{pred}_{G_C}(v) \cdots$ return the set $\{u \in V; (u, v) \in E_C\}$;
- $\mathrm{adj}_{G_R}(u, v) \cdots$ return 1 if $(u, v) \in E_R$ and otherwise 0;
- $\mathrm{succ}_{G_R}(v) \cdots$ return the set $\{u \in V; (v, u) \in E_R\}$;
- $\mathrm{pred}_{G_R}(v) \cdots$ return the set $\{u \in V; (u, v) \in E_R\}$.

In particular, $\mathrm{adj}_{G_R}(u, v) = 1 \Rightarrow \mathrm{adj}_{G_C}(u, v) = 1$ holds for all $u, v \in V$ and $\mathrm{succ}_{G_R}(v) \subseteq \mathrm{succ}_{G_C}(v)$ and $\mathrm{pred}_{G_R}(v) \subseteq \mathrm{pred}_{G_C}(v)$ also hold for any $v \in V$.

We also define some terminologies and utility functions. Given a chain decomposition of a poset, *node index* is the label of a node and *chain index* of a node is the pair (p, i) such that the node is the i-th node of chain p. We often specify a node not only by its node index but also by its chain index. Function node_index(p, i) converts the chain index (p, i) into its node index. On the other hand, function chain_index(v) converts the node index v into its chain index. The *lower bound* in chain q of a node (p, i) is the node (q, j) such that j is the maximum value which satisfies $(q, j) \prec (p, i)$ and we denote it by $lb_q(p, i)$. To the contrary, the *upper bound* in chain q of a node (p, i) is the node (q, j) such that j is the minimum value which satisfies $(p, i) \prec (q, j)$ and we denote it by $ub_q(p, i)$. When v is the node index of (p, i), we define $lb_q(v) = lb_q(p, i)$ and $ub_q(v) = ub_q(p, i)$.

2.4 Succinct Data Structures

In this paper, we discuss data structures in the word-RAM model [22], which supports reading, writing, arithmetic operations and bitwise operations on a word of $w = \Omega(\log N)$ bits in constant time, where N is the size of data. We make use of some of the basic succinct data structures such as bitvector, string and permutation.

A bitvector data structure stores a sequence N bits, $B[1 \dots N]$ and supports random access and rank/select queries described below:

- $B[i] \cdots$ return i-th element;
- $\mathrm{rank}_c(B, i) \cdots$ return the number of $j \in \{1, 2, \dots, \min\{i, n\}\}$ such that $B[j] = c$;
- $\mathrm{select}_c(B, i) \cdots$ return $\inf\{j \in \mathbb{N}; \mathrm{rank}_c(B, j) \geq i\}$.

Note that $\mathrm{select}_c(B, i)$ returns ∞ when i is greater than the number of c's in B. All of these queries can be supported in constant time using a data structure whose space complexity is $N + \mathrm{o}(N)$ bits [28]. It can also be compressed into $\log \binom{N}{M} + \mathrm{o}(N)$ bits [11, 35] when M is the number of 1s in the bitvector.

String $S[1 \dots N]$ is a generalisation of the bitvector into a sequence of characters from the alphabet $\{0, 1, \dots, L-1\}$ instead of bits in $\{0, 1\}$. A succinct string data structure supports random access and rank/select queries for each $c \in \{0, 1, \dots, L-1\}$. Random access and rank queries can be calculated in $\mathrm{O}(\log \log L)$ time and select query can be computed in $\mathrm{O}(1)$ time, and the data structure can be stored in $N \log L + \mathrm{o}(N \log L)$ bits of space [24]. Moreover, the wavelet tree [26] data structure can be used to support all three operations in $O(\log L)$ time.

A permutation data structure represents a bijection $\pi : \{1, 2, \dots, N\} \to \{1, 2, \dots, N\}$ and supports the queries $\pi(i)$ and $\pi^{-1}(i)$ for all $i = 1, 2, \dots, N$. The permutation data structure of Munro et al. [31] supports $\pi(i)$ in constant time and $\pi^{-1}(i)$ in $\mathrm{O}(1/\epsilon)$ time using $(1+\epsilon)N \log N + N + \mathrm{o}(N)$ bits, for any parameter $0 < \epsilon \leq 1$.

2.5 Farzan and Fischer's Data Structure [19]

Let $\mathcal{P}$ be a poset on n elements with Dilworth's width k, and let $\{C_p\}_{p \in \Sigma}$ be one of the minimal chain decompositions of $\mathcal{P}$ where $\Sigma = \{0, 1, \dots, k-1\}$. Farzan and Fischer's data structure consists of a permutation π, a bitvector B and bitvectors D_{pq} for each $p, q \in \Sigma$ such that $p \neq q$. The permutation π is the bijection of $V \to V$ and it uses $(1+\epsilon)n \log n + n + \mathrm{o}(n)$ bits of space for any $\epsilon \in (0, 1]$. The length of the bitvector B is n and that of D_{pq} is $|C_p| + |C_q|$ for each $p, q \in \Sigma$, $p \neq q$. We store auxiliary structures to support rank and select queries on each of the bit vectors (B and D_{pq}, for each $p, q \in \Sigma$, $p \neq q$). Thus the space complexity of the whole data structure is

$$(1+\epsilon)n \log n + n + \mathrm{o}(n) + \left(n + \sum_{p,q \in \Sigma,\, p \neq q} (|C_p| + |C_q|) \right) (1 + \mathrm{o}(1)) = (1+\epsilon)n \log n + 2nk + \mathrm{o}(nk)$$

bits.

Permutation π and bitvector B represent the correspondence between node indices and chain indices. We construct π so that $\pi^{-1}(v) = i + \sum_{q<p} |C_q|$ for each $v \in V$ where (p, i) is the chain index of v. In other words, π reorders the nodes from the lexicographical order of chain indices to the order of node indices. Also, we construct B in such a way that $B[j] = 1$ if and only if there exists $p \in \Sigma$ which satisfies $j = \sum_{q \leq p} |C_q|$; B essentially represents the length of the individual chains in the decomposition (note that one can store B in a compressed form, but this would not change the overall space complexity). Then, node_index(v) and chain_index(p, i) are implemented as Algorithms 1 and 2 respectively. The time complexity of node_index(v) is $\mathrm{O}(1)$ and that of chain_index(v) is $\mathrm{O}(1/\epsilon)$.

For each $p, q \in \Sigma, p \neq q$, D_{pq} is constructed as $D_{pq} = 0^{\delta^0_{pq}} 1\, 0^{\delta^1_{pq}} 1 \dots 0^{\delta^{|C_p|-1}_{pq}} 1\, 0^{\delta^{|C_p|}_{pq}}$ where δ^i_{pq} is an increase of the number of edges on G_C from node $(p, i+1)$ to nodes in chain q as compared with node (p, i). We consider that there is no outgoing edge from node $(p, 0)$ and any node can be reached from node $(p, |C_p|+1)$. In particular, $\sum_{i<i'} \delta^i_{pq} = |\{j \in$

Algorithm 1 convert chain index to node index.

```
1: procedure node_index(p, i)
2:     a := i + select_1(B, p)
3:     return π(a)
```

Algorithm 2 convert node index to chain index.

```
1: procedure chain_index(v)
2:     a := π^{-1}(v)
3:     p := rank_1(B, a − 1)
4:     return (p, a − select_1(B, p))
```

$J; (q, j) \prec (p, i')\}|$ for all $i' \in \{0, 1, \ldots, |C_p|+1\}$ where $J = \{1, 2, \ldots, |C_q|\}$. By using D_{pq}, we can obtain the lower bound and the upper bound in chain q of node (p, i) in constant time with Algorithm 3 and 4. An edge from node (p, i) to node (q, j) is a transitive edge when $j <$ lower_bound(p, i, q) because there exists a path which goes over $lb_q(p, i)$. Node (p, i) does not have a path to node (q, j) when $j >$ lower_bound(p, i, q) since if there is such a path, $lb_q(p, i) \prec (q, j) \prec (p, i)$ and it contradicts the definition of lower bound.

Algorithm 3 return j if (q, j) is the largest node reachable from (p, i).

```
1: procedure lower_bound(p, i, q)
2:     if p = q then return i − 1
3:     else return select_1(D_pq, i) − i
```

Then, we can implement the algorithms for the six queries mentioned in Section 2.3 as follows. First, $\mathrm{adj}_{G_C}(u, v) = 1$ if and only if $lb_q(u) \preceq v$ for such q that $v \in C_q$. Hence, all we have to do in order to compute $\mathrm{adj}_{G_C}(u, v)$ is to check $j \leq$ lower_bound(p, i, q) when (p, i) is the chain index of u and (q, j) is that of v. The bottleneck of the computation is the conversions to chain indices and the time complexity of $\mathrm{adj}_{G_C}(u, v)$ is $\mathrm{O}(1/\epsilon)$. Also, $\mathrm{succ}_{G_C}(v)$ (respectivly, $\mathrm{pred}_{G_C}(v)$) can be calculated by collecting all the nodes less (respectively, greater) than or equal to $lb_p(v)$ (respectively, $ub_p(v)$) for all $p \in \Sigma$. The time complexities of $\mathrm{succ}_{G_C}(v)$ and $\mathrm{pred}_{G_C}(v)$ are both $\mathrm{O}(1/\epsilon + k + t)$ where t is the size of the output.

To compute $\mathrm{adj}_{G_R}(u, v)$, we need to check that there is no path from u to v other than the direct edge (u, v). If there exists $p \in \Sigma$ such that $ub_p(v) \preceq lb_p(u)$, there exists a non-direct path which goes through $ub_p(v)$ since $v \prec ub_p(v) \preceq lb_p(u) \prec u$. Thus, $\mathrm{adj}_{G_R}(u, v) = 1$ if and only if $\mathrm{adj}_{G_C}(u, v) = 1$ and $lb_p(u) \prec ub_p(v)$ for all $p \in \Sigma$. Checking $lb_p(u) \prec ub_p(v)$ for all $p \in \Sigma$ costs $\mathrm{O}(1/\epsilon+k)$ time and therefore the time complexity of $\mathrm{adj}_{G_R}(u, v)$ is also $\mathrm{O}(1/\epsilon+k)$. It can be easily observed that $\mathrm{succ}_{G_R}(v) = \{u \in L(v);\ \mathrm{adj}_{G_R}(v, u) = 1\}$ and $\mathrm{pred}_{G_R}(v) = \{u \in U(v);\ \mathrm{adj}_{G_R}(u, v) = 1\}$ where $L(v) = \{lb_p(v);\ p \in \Sigma\}$ and $U(v) = \{ub_p(v);\ p \in \Sigma\}$. Hence, the time complexities of $\mathrm{succ}_{G_R}(v)$ and $\mathrm{pred}_{G_R}(v)$ are both $\mathrm{O}(1/\epsilon + k^2)$.

3 Improved Data Structures

3.1 Lower Bound on Space

Let $Z_k(n)$ be the number of topologically labeled posets and $X_k(n)$ be the number of unlabeled posets. As written in Section 1, the number of generally labeled posets $N_k(n)$ satisfies

$$N_k(n) \geq \frac{n!}{k!} 4^{n(k-1)} n^{-24k(k-1)}.$$

Algorithm 4 return j if (q, j) is the smallest node reachable to (p, i).

```
1: procedure upper_bound(p, i, q)
2:     if p = q then return i + 1
3:     else return select_0(D_qp, i) − i + 1
```

Each unlabeled posets has at most $n!$ distinct labeling, therefore $n! \cdot X_k(n) \geq N_k(n)$ holds. It is clear that $Z_k(n) \geq X_k(n)$ and the information-theoretic lower bound on the number of topologically labeled posets is

$$\log Z_k(n) \geq 2n(k-1) - \Theta(k^2 \log n)$$

bits. Thus, it is possible to compress posets into space which is linear in n with fixed k when the posets are topologically labeled.

3.2 Modified Farzan and Fischer's Data Structure

In this section, we adapt Farzan and Fischer's data structure in Section 2.5 for topologically labeled posets and make its space close to the lower bound in Section 3.1. In order to do so, we replace the permutation π and the bitvector B with a string S. Eventually, this data structure consists of the string S and the bitvectors D_{pq} $(p, q \in \Sigma,\ p \neq q)$.

The length of S is n, its alphabet is Σ and $S[v] = p$ if $v \in C_p$. String S plays the role of storing information of the correspondence between node indices and chain indices, as well as π and B in Section 2.5 do. By using S, node_index(p, i) and chain_index(v) can be calculated by Algorithms 5 and 6, respectively. Since select$_p$ takes $\mathrm{O}(1)$ time and rank$_p$ takes $\mathrm{O}(\log \log k)$ time, node_index(p, i) can be supported in $\mathrm{O}(1)$ time and chain_index(v) in $\mathrm{O}(\log \log k)$ time.

Algorithm 5 convert chain index to node index.

```
1: procedure node_index(p, i)
2:     return select_p(S, i)
```

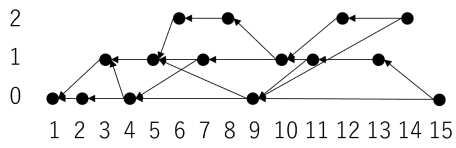
Algorithm 6 convert node index to chain index.

```
1: procedure chain_index(v)
2:     p := S[v]
3:     return (p, rank_p(S, v))
```

Then, $\mathrm{adj}_{G_C}(u, v)$ can be computed in $\mathrm{O}(\log \log k)$ time, $\mathrm{succ}_{G_C}(v)$ and $\mathrm{pred}_{G_C}(v)$ in $O(k + t)$ time where t is the size of output, $\mathrm{adj}_{G_R}(u, v)$ in $\mathrm{O}(k)$ time, and $\mathrm{succ}_{G_R}(v)$ and $\mathrm{pred}_{G_R}(v)$ in $\mathrm{O}(k^2)$ time. Also, the total space of the data structure is

$$\left(n \log k + \sum_{p,q \in \Sigma,\ p \neq q} (|C_p| + |C_q|) \right) (1 + \mathrm{o}(1)) = 2n(k-1) + \mathrm{o}(nk)$$

bits. For non-constant $k = \mathrm{o}(\frac{n}{\log n})$, this data structure is succinct since $\Theta(k^2 \log n) = \mathrm{o}(nk)$.



$S = 001012120112120$

$I_{00} = 11110$ $I_{10} = 110001$ $I_{20} = 0000$
$I_{01} = 10110$ $I_{11} = 111110$ $I_{21} = 0100$
$I_{02} = 00010$ $I_{12} = 010100$ $I_{22} = 1010$

$O_{00} = 01111$ $O_{10} = 101010$ $O_{20} = 0001$
$O_{01} = 00111$ $O_{11} = 011111$ $O_{21} = 1010$
$O_{02} = 00000$ $O_{12} = 000100$ $O_{22} = 0101$

Figure 1 Example of a poset and its encoding.

3.3 Proposed Data Structure

The data structure in Section 3.2 can perform queries on G_C fast, whereas it is slow to do queries on G_R. However, when we have to run some BFS/DFS-based algorithm, such as algorithms to solve a shortest path problem on G_R or a longest path problem, it is required to do such queries on G_R faster. For this reason, we propose a new data structure which can support queries on G_R faster.

Our data structure consists of a string S and bitvectors I_{pq}, O_{pq} $(p, q \in \Sigma)$. The string S is same as the one in Section 3.2. The lengths of I_{pq} and O_{pq} are both equal to $|C_p|$. Therefore, the total space of our data structure is

$$\left(n \log k + 2k \cdot \sum_{p=0}^{k-1} |C_p|\right)(1 + \mathrm{o}(1)) = 2nk + \mathrm{o}(nk)$$

bits. $I_{pq}[i] = 1$ when node (p, i) has an incoming edge from some node of chain q and otherwise $I_{pq}[i] = 0$. Similarly, $O_{pq}[i] = 1$ when node (p, i) has an outgoing edge to some node of chain q and otherwise $O_{pq}[i] = 0$. In Figure 1, the left graph is an example of a transitive reduction graph of a poset with $n = 15$ and $k = 3$ and its encoding is shown in the right side.

If there exists an edge $(u, v) \in E_R$ such that $u \in C_p$ and $v \in C_q$, there is no edge $(u', v') \in E_R \setminus \{(u, v)\}$ which satisfies $u' \in C_p, v' \in C_q$ and $u \preceq u', v \succeq v'$. This is because when there exists such an edge (u', v'), then $u \preceq u' \prec v' \preceq v$ and (u, v) becomes a transitive edge since $(u, v) \neq (u', v')$. Therefore, the incoming edge corresponding to the i-th 1 of I_{pq} is the same as the outgoing edge corresponding to the i-th 1 of O_{qp}.

Let $E_R(K) = \{(u, v) \in E_R; u \in C_i, v \in C_j, i, j \in K\}$ for each $K \subseteq \Sigma$. We define functions called nearest_dst(p, i, q) and nearest_src(p, i, q) which return the second element of chain index of the nearest node in chain q reachable from (respectively, reachable to) the node (p, i) through the edges in $E_R(\{p, q\})$. If there is no such node, then it returns 0 (respectively, ∞). These can be calculated by Algorithms 7 and 8 in constant time.

Algorithm 7 return the nearest destination in chain q from the node (p, i).

```
1: procedure nearest_dst(p, i, q)
2:     return select_1(I_qp, rank_1(O_pq, i))
```

Algorithm 8 return the nearest source in chain q to the node (p, i).

```
1: procedure nearest_src(p, i, q)
2:     return select_1(O_qp, rank_1(I_pq, i − 1) + 1)
```

Other utility functions we defined are lower_bounds(v) and upper_bounds(v). These functions return an array of length k which stores the node indices of $lb_p(v)$ (respectively, $ub_p(v)$) for each $p \in \Sigma$. If there is no such node, then it stores 0 (respectively, ∞). The algorithms of lower_bounds(v) and upper_bounds(v) are shown in Algorithm 9 and 10. These algorithms use the data structures known as priority queue. priority_queue$_{\leq}$ and priority_queue$_{\geq}$ support push(p, i) to push a new node (p, i), pop() to pop out the node of the largest (respectively, smallest) node index, top() to access the node of the largest (respectively, smallest) node index and update(p, i) to update the node $(p, *)$ in the priority queue to (p, i) if the node index of (p, i) is larger (respectively, smaller) than that of $(p, *)$. Incidentally, we regard the node index of $(*, 0)$ as 0 and that of $(*, \infty)$ as ∞. Remind that the conversion from chain index to node index can be done in constant time, therefore comparing nodes by node index does not affect the time complexities. In this paper, we use Relaxed Heap [17] which supports push(p, i), top(p, i), and update(p, i) in $O(1)$ time and pop(p, i) in $O(\log N)$ time where N is the number of elements in the priority queue.

Algorithm 9 return the largest nodes reachable from v in each chain.

```
1:  procedure lower_bounds(v)
2:      array R[0...k-1] := {0,...,0}
3:      priority_queue_≤ Q := {(0,0),(1,0),...,(k-1,0)}
4:      (p_0, i_0) := chain_index(v)
5:      Q.update(p_0, i_0)
6:      while Q ≠ ∅ do
7:          (p, i) := Q.top()
8:          Q.pop()
9:          if i = 0 then break
10:         R[p] ← node_index(p, i)
11:         for q = 0, 1, ..., k-1 do
12:             if R[q] ≠ 0 then continue
13:             j := nearest_dst(p, i, q)
14:             Q.update(q, j)
15:     R[p_0] ← node_index(p_0, i_0 - 1)
16:     return R
```

In Algorithms 9 and 10, the while loop iterates at most k times because priority queue Q has k elements before the while loop and the elements are popped one by one in each iteration. The bottleneck of lower_bounds(v) and upper_bounds(v) is the for loop. The statements in it run at most k^2 times and the time complexity of the whole algorithm is $O(k^2)$.

The correctness of Algorithm 9 can be shown as follows. Trivially, each element of R is updated in line 10 at most once before reaching line 15. Let $K \subseteq \Sigma$ be a set of the indices such that the elements of R corresponding to them have been already updated and $\bar{K} = \Sigma \setminus K$. In the while loop, assume that $R[p']$ correctly stores the largest node v' in chain p' such that $v' \preceq v$ when $p' \in K$. Let u be the node index of the node (p, i) declared in line 7. If $i = 0$, it means that the nodes in C_p cannot be reached from v and the nodes in $\bigcup_{p' \in \bar{K}} C_{p'}$ cannot be either because $Q = \{(p', 0);\ p' \in \bar{K}\}$ holds since the node indices of all the nodes in Q are not larger than the node index of $(p, 0)$. Otherwise, it is easy to observe that node u is the largest destination reachable from v through the edges in $E_R(K \cup \{p\})$. If there exists a node $w \in \bigcup_{p' \in \bar{K} \setminus \{p\}} C_{p'}$ such that $u \prec w \prec v$, then for some $p' \in \bar{K} \setminus \{p\}$, there

Algorithm 10 return the smallest nodes reachable to v in each chain.

```
1:  procedure upper_bounds(v)
2:      array R[0...k-1] := {∞, ..., ∞}
3:      priority_queue_≥ Q := {(0,∞), (1,∞), ..., (k-1,∞)}
4:      (p_0, i_0) := chain_index(v)
5:      Q.update(p_0, i_0)
6:      while Q ≠ ∅ do
7:          (p, i) := Q.top()
8:          Q.pop()
9:          if i = ∞ then break
10:         R[p] ← node_index(p, i)
11:         for q = 0, 1, ..., k-1 do
12:             if R[q] ≠ ∞ then continue
13:             j := nearest_src(p, i, q)
14:             Q.update(q, j)
15:     R[p_0] ← node_index(p_0, i_0 + 1)
16:     return R
```

exists at least one node $w' \in C_{p'}$ such that $u \prec w'$ and w' can be reached from v through the edges in $E_R(K \cup \{p'\})$, and there also exists a node $w'' \in C_{p'}$ which satisfies $w'' \in Q$ and $w' \preceq w''$. However, $u < w''$ if $u \prec w''$ and it contradicts with the features of priority queue Q. Therefore, node u is also the largest destination reachable from v through any edges in E_R. Hence, even if we update the value of $R[p]$ into u and K into $K \cup \{u\}$, it does not go against the assumption, and recursively, it can be confirmed that $R[p] \preceq v$ holds for each $p \in \Sigma$ just before line 15. Finally, $R[p_0]$ is modified in line 15 not to be equal to v itself, and then we obtain the correct answer. The correctness of Algorithm 10 can be proved in the same way.

Using these utility functions, we can perform the six queries shown in Section 2.3. Each algorithm of the queries are given in Algorithms 11, 12, 13, 14, 15 and 16. $\mathrm{adj}_{G_C}(u, v)$ can be calculated in $\mathrm{O}(k^2)$, $\mathrm{succ}_{G_C}(v)$ and $\mathrm{pred}_{G_C}(v)$ in $\mathrm{O}(k^2 + t)$ where t is the size of output, $\mathrm{adj}_{G_R}(u, v)$ in $\mathrm{O}(1)$ and $\mathrm{succ}_{G_R}(v)$ and $\mathrm{pred}_{G_R}(v)$ in $\mathrm{O}(k)$.

Note that I_{pp} and O_{pp} $(p \in \Sigma)$ is not necessary to reconstruct the poset because $I_{pp}[i] = 1$ if and only if $lb_q(p, i+1) \prec ub_q(p, i)$ for all $q \in \Sigma$ and $O_{pp}[i] = 1$ if and only if $lb_q(p, i) \prec ub_q(p, i-1)$ for all $q \in \Sigma$. Remind that lower_bounds(v) and upper_bounds(v) can be obtained without using I_{pp} and O_{pp} $(p \in \Sigma)$ since the arguments never be $p = q$ in line 13 of Algorithm 9 and 10. The total space of I_{pp} and O_{pp} $(p \in \Sigma)$ is

$$\left(2 \cdot \sum_{p=0}^{k-1} |C_p|\right)(1 + \mathrm{o}(1)) = 2n + \mathrm{o}(n)$$

bits. Thus, when we do not store them, the space of the data structure becomes $2n(k-1) + \mathrm{o}(nk)$ bits and this is close to the information theoretical lower bound in Section 3.1. However, the space with I_{pp} and O_{pp} $(p \in \Sigma)$ is also asymptotically the same because

$$2nk + \mathrm{o}(nk) = 2n(k-1) + 2n + \mathrm{o}(nk) = 2n(k-1) + \mathrm{o}(nk).$$

Therefore, we store them for the sake of fast response to the queries.

3.4 Faster Index

We also propose another data structure which is efficient to compute all six queries at the extra expense of space. The idea of this data structure is based on the modified version of Farzan and Fischer's data structure in Section 3.2. This data structure consists of three components: string S, bitvectors D_{pq} $(p, q \in \Sigma,\ p \neq q)$ and bitvectors T_v $(v \in V)$.

The string S and the bitvectors D_{pq} is same as the one in Section 3.3. The bitvectors T_v has length k for each $v \in V$ and represents whether node v has a non-transitive edge to each chain or not. If one of the edges on G_C from node v to the node in chain p is not a transitive edge, then $T_v[p] = 1$.

The string S requires $n \log k + \mathrm{o}(n \log k)$ bits, the bitvector D_{pq} does $|C_p|+|C_q|+o(|C_p| + |C_q|)$ bits for each $p, q \in \Sigma,\ p \neq q$ and the bitvector T_v does $k + \mathrm{o}(k)$ bits for each $v \in V$. Thus, the total space of the data structure is

$$\left(n \log k + \sum_{p,q \in \Sigma,\, p \neq q} (|C_p|+|C_q|) + \sum_{v \in V} k \right) (1 + \mathrm{o}(1)) = 3nk - 2n + \mathrm{o}(nk)$$

bits.

On this data structure, node_index(p, i) and chain_index(v) can be done by Algorithm 5 and 6. Also, $\mathrm{adj}_{G_C}(u, v)$, $\mathrm{succ}_{G_C}(v)$ and $\mathrm{pred}_{G_C}(v)$ can be computed in the same way as mentioned in Section 3.2.

$\mathrm{adj}_{G_R}(u, v)$, $\mathrm{succ}_{G_R}(v)$ and $\mathrm{pred}_{G_R}(v)$ can be calculated by the following ways. The edge from (p, i) to $lb_q(p, i)$ is a non-transitive edge when $T_v[q] = 1$ where v is the node index of (p, i). Thus, $\mathrm{adj}_{G_R}(u, v)$ returns 1 if and only if lower_bound$(p, i, q) = j$ and $T_u[q] = 1$ where (p, i) is a chain index of u and (q, j) is that of v. The bottleneck of $\mathrm{adj}_{G_R}(u, v)$ is the conversions from node indices to chain indices and $\mathrm{adj}_{G_R}(u, v)$ can be obtained in $\mathrm{O}(\log \log k)$ time. $\mathrm{succ}_{G_R}(v)$ can be computed by the process that collects the node index of $lb_q(p, i)$ for all $q \in \Sigma$ such that $T_v[q] = 1$ where (p, i) is a chain index of v. When we only iterate q which satisfies the condition $T_v[q] = 1$ by using select query on T_v, the time complexity is $\mathrm{O}(\log \log k + t)$ where t is the size of output. $\mathrm{pred}_{G_R}(v)$ can be computed in $\mathrm{O}(k)$ time by the process that checks whether node $ub_q(p, i)$ is adjacent to node v on G_R for each $q \in \Sigma$ where (p, i) is a chain index of v.

3.5 Higher Order Compression

First, we consider the data structure in Section 3.3. Let m_{pq} be the number of 1s in bitvector I_{pq} for each $p, q \in \Sigma$. It is obvious that m_{pq} is also the number of 1s in bitvector O_{qp} for each $p, q \in \Sigma$ and $\sum_{p,q \in \Sigma} m_{pq} = m_R$. When we use the compressed bitvectors for each I_{pq} and O_{pq}, the space complexity of the whole data structure becomes

$$n \log k + 2 \cdot \sum_{p,q \in \Sigma} \log \binom{|C_p|}{m_{pq}} + \mathrm{o}(nk) \leq 2 \log \binom{nk}{m_R} + \mathrm{o}(nk)$$

bits.

The data structure in Section 3.4 can also be compressed. Let m'_v be the number of 1s in bitvectors T_v for each $v \in V$. Then, $\sum_{v \in V} m'_v = m_R$. The space complexity of the data structure becomes

$$n \log k + \sum_{p,q \in \Sigma,\, p \neq q} (|C_p|+|C_q|) + \sum_{v \in V} \log \binom{k}{m'_v} + \mathrm{o}(nk) \leq \log \binom{nk}{m_R} + 2n(k-1) + \mathrm{o}(nk)$$

bits if we use compressed bitvectors for each T_v.

4 Conclusions

On the DAGs of BlockDAG, the growth of the width is much less likely than the depth, which is the longest path of the DAG. Thus, the space to store them can be almost linear in the number of elements with the data structures we consider in Section 3. This is one of the advantages that the other existing data structures do not have. Also, we provide the trade-off among time of the queries on G_C, time on G_R and space.

There remain some open problems such as:

- Can adding operation in the dynamic data structure be faster?
- Can the time of construction of these data structures be reduced by calculating the chain decomposition approximately?

It would be also interesting to consider other queries.

References

1 H. Acan, S. Chakraborty, S. Jo, K. Nakashima, K. Sadakane, and S. R. Satti. Succinct representations of intersection graphs on a circle. In *31st Data Compression Conference, DCC 2021, Snowbird, UT, USA, March 23-26, 2021*, pages 123–132. IEEE, 2021.

2 H. Acan, S. Chakraborty, S. Jo, and S. R. Satti. Succinct data structures for families of interval graphs. In *Algorithms and Data Structures - 16th International Symposium, WADS 2019, Edmonton, AB, Canada, August 5-7, 2019, Proceedings*, volume 11646 of *Lecture Notes in Computer Science*, pages 1–13. Springer, 2019.

3 A. V. Aho, M. R. Garey, and J. D. Ullman. The transitive reduction of a directed graph. *SIAM Journal on Computing*, 1(2):131–137, 1972.

4 J. Alman and V. V. Williams. A refined laser method and faster matrix multiplication. In *Proceedings of the 2021 ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 522–539. ACM-SIAM, 2021.

5 J. Balabán and P. Hliněný. Twin-width is linear in the poset width. In *16th International Symposium on Parameterized and Exact Computation (IPEC 2021)*, volume 214 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 6:1–6:13, Dagstuhl, Germany, 2021. Schloss Dagstuhl – Leibniz-Zentrum für Informatik.

6 J. Barbay, F. Claude, and G. Navarro. Compact binary relation representations with rich functionality. *Information and Computation*, 232:19–37, 2013.

7 J. Barbay, M. He, J. I. Munro, and S. R. Satti. Succinct indexes for strings, binary relations and multilabeled trees. *ACM Transactions on Algorithms*, 7(4), 2011.

8 Garrett Birkhoff. On the structure of abstract algebras. *Mathematical Proceedings of the Cambridge Philosophical Society*, 31(4):433–454, 1935.

9 É. Bonnet, E. J. Kim, S. Thomassé, and R. Watrigant. Twin-width I: Tractable FO model checking. *Journal of the ACM*, 69(1), 2021.

10 G. Brightwell and S. Goodall. The number of partial orders of fixed width. *Order*, 13(4):315–337, 1996.

11 A. Brodnik and J. I. Munro. Membership in constant time and almost-minimum space. *SIAM Journal on Computing*, 28(5):1627–1640, 1999.

12 S. Chakraborty, S. Jo, K. Sadakane, and S. R. Satti. Succinct data structures for series-parallel, block-cactus and 3-leaf power graphs. In *Combinatorial Optimization and Applications - 15th International Conference, COCOA 2021, Tianjin, China, December 17-19, 2021, Proceedings*, volume 13135 of *Lecture Notes in Computer Science*, pages 416–430. Springer, 2021.

13 S. Chakraborty, S. Jo, K. Sadakane, and S. R. Satti. Succinct data structures for small clique-width graphs. In *31st Data Compression Conference, DCC 2021, Snowbird, UT, USA, March 23-26, 2021*, pages 133–142. IEEE, 2021.

14 Y. Chen and Y. Chen. On the DAG decomposition. *British Journal of Mathematics & Computer Science*, 10:1–27, 2015.

15 C. Daskalakis, R. M. Karp, E. Mossel, S. J. Riesenfeld, and E. Verbin. Sorting and selection in posets. *SIAM Journal on Computing*, 40(3):597–622, 2011.

16 R. P. Dilworth. A decomposition theorem for partially ordered sets. *Annals of Mathematics*, 51(1):161–166, 1950.

17 J. R. Driscoll, H. N. Gabow, R. Shrairman, and R. E. Tarjan. Relaxed heaps: An alternative to Fibonacci heaps with applications to parallel computation. *Communications of the ACM*, 31(11):1343–1354, 1988.

18 M. Erné, J. Heitzig, and J. Reinhold. On the number of distributive lattices. *The Electronic Journal of Combinatorics*, 9, 2002.

19 A. Farzan and J. Fischer. Compact representation of posets. In *Proceedings of the 22nd International Conference on Algorithms and Computation (ISAAC)*, pages 302–311, Berlin, Heidelberg, 2011. Springer-Verlag.

20 A. Farzan and J. I. Munro. Succinct encoding of arbitrary graphs. *Theoretical Computer Science*, 513:38–52, 2013.

21 M. J. Fischer and A. R. Meyer. Boolean matrix multiplication and transitive closure. In *12th Annual Symposium on Switching and Automata Theory (swat 1971)*, pages 129–131, 1971.

22 M. L. Fredman and D. E. Willard. Blasting through the information theoretic barrier with fusion trees. In *Proceedings of the Twenty-Second Annual ACM Symposium on Theory of Computing (STOC)*, pages 1–7, New York, NY, USA, 1990.

23 D. R. Fulkerson. Note on Dilworth's decomposition theorem for partially ordered sets. *Proceedings of the American Mathematical Society*, 7(4), 1956.

24 A. Golynski, J. I. Munro, and S. R. Satti. Rank/select operations on large alphabets: A tool for text indexing. In *Proceedings of the Seventeenth Annual ACM-SIAM Symposium on Discrete Algorithm (SODA)*, pages 368–373, USA, 2006. Society for Industrial and Applied Mathematics.

25 A. Goralčíková and V. Koubek. A reduct-and-closure algorithm for graphs. In *Mathematical Foundations of Computer Science 1979*, pages 301–307, Berlin, Heidelberg, 1979. Springer Berlin Heidelberg.

26 R. Grossi, A. Gupta, and J. S. Vitter. High-order entropy-compressed text indexes. In *Proceedings of the Fourteenth Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 841–850, USA, 2003. Society for Industrial and Applied Mathematics.

27 J. E. Hopcroft and R. M. Karp. An $n^{5/2}$ algorithm for maximum matchings in bipartite graphs. *SIAM Journal on Computing*, 2(4):225–231, 1973.

28 G. J. Jacobson. *Succinct Static Data Structures*. PhD thesis, Carnegie Mellon University, USA, 1988. AAI8918056.

29 D. J. Kleitman and B. L. Rothschild. Asymptotic enumeration of partial orders on a finite set. *Transactions of the American Mathematical Society*, 205:205–220, 1975.

30 J. I. Munro and P. K. Nicholson. Succinct posets. *Algorithmica*, 76(2):445–473, 2016.

31 J. I. Munro, R. Raman, V. Raman, and S. R. Satti. Succinct representations of permutations and functions. *Theoretical Computer Science*, 438:74–88, 2012.

32 J. I. Munro and C. Sinnamon. Time and space efficient representations of distributive lattices. In *Proceedings of the 2018 Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 550–567. ACM-SIAM, 2018.

33 J. I. Munro and K. Wu. Succinct data structures for chordal graphs. In *29th International Symposium on Algorithms and Computation, ISAAC 2018, December 16-19, 2018, Jiaoxi, Yilan, Taiwan*, volume 123 of *LIPIcs*, pages 67:1–67:12. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2018.

34 S. Nakamoto. Bitcoin: A peer-to-peer electronic cash system. *Cryptography Mailing list at https://metzdowd.com*, 2009. URL: `https://bitcoin.org/bitcoin.pdf`.

35 R. Pagh. Low redundancy in static dictionaries with constant query time. *SIAM Journal on Computing*, 31(2):353–363, 2001.

36 M. Pilipczuk, M. Sokołowski, and A. Zych-Pawlewicz. Compact representation for matrices of bounded twin-width. In *39th International Symposium on Theoretical Aspects of Computer Science (STACS 2022)*, volume 219 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 52:1–52:14, Dagstuhl, Germany, 2022. Schloss Dagstuhl – Leibniz-Zentrum für Informatik.

37 Y. Sompolinsky, S. Wyborski, and A. Zohar. Phantom ghostdag: A scalable generalization of Nakamoto consensus: September 2, 2021. In *Proceedings of the 3rd ACM Conference on Advances in Financial Technologies*, pages 57–70, New York, NY, USA, 2021.

A Pseudo Codes

Algorithm 11 whether $(u, v) \in E_C$.

```
1: procedure adj_{G_C}(u, v)
2:     R := lower_bounds(v)
3:     (p, ignore) := chain_index(u)
4:     if u ≤ R[p] then return 1
5:     else return 0
```

Algorithm 12 return out-neighbors of node v on G_C.

```
1: procedure succ_{G_C}(v)
2:     T := ∅
3:     R := lower_bounds(v)
4:     for p = 0, 1, ..., k − 1 do
5:         c := rank_p(S, R[p])
6:         for i = 1, 2, ..., c do
7:             u := node_index(p, i)
8:             T ← T ∪ {u}
9:     return T
```

Algorithm 13 return in-neighbors of node v on G_C.

```
1: procedure pred_{G_C}(v)
2:     T := ∅
3:     R := upper_bounds(v)
4:     for p = 0, 1, ..., k − 1 do
5:         c := rank_p(S, R[p])
6:         d := rank_p(S, n)
7:         for i = c, c + 1, ..., d do
8:             u := node_index(p, i)
9:             T ← T ∪ {u}
10:    return T
```

Algorithm 14 whether $(u, v) \in E_R$.

```
1: procedure adj_{G_R}(u, v)
2:     (p, i) := chain_index(u)
3:     (q, j) := chain_index(v)
4:     if u ≠ v and O_pq[i] = I_qp[j] = 1 and rank_1(O_pq, i) = rank_1(I_qp, j) then
5:         return 1
6:     else
7:         return 0
```

Algorithm 15 return out-neighbors of node v on G_R.

```
1: procedure succ_{G_R}(v)
2:     T := ∅
3:     (p, i) := chain_index(v)
4:     for q = 0, 1, ..., k − 1 do
5:         if O_pq[i] = 1 then
6:             u := node_index(q, nearest_dst(p, i, q))
7:             T ← T ∪ {u}
8:     return T
```

B Dynamic Data Structures

In BlockDAG application, the required operation to modify posets is only adding to the transitive reduction graph a node with some non-transitive edges outgoing from the new node. There is an algorithm called *peeling* introduced by Daskalakis et al. [15]. Given a poset and its chain decomposition of size $k' \leq 2k$ as inputs, this algorithm reduces the size of the chain decomposition to k one by one in each iteration and returns the minimal chain decomposition. What we have to do in order to realize the adding operation is that add the new node to a new chain, run the iteration of peeling algorithm once to obtain the minimal chain decomposition and modify the data structure.

According to their paper, lookup tables of $lb_p(v)$ and $ub_p(v)$ for each $v \in V$ and $p \in \Sigma$ is required to call peeling algorithm. They named the tables *chainmerge*. Chainmerge can be constructed in $\mathrm{O}(nk)$ time for the data structure in Sections 3.2 and 3.4 and $\mathrm{O}(nk^2)$ time for the data structure in Section 3.3. One iteration of peeling algorithm takes $\mathrm{O}(nk)$ time. Even if we create new strings and bitvectors of the whole data structure, it requires at most $\mathrm{O}(nk)$ time when the minimal chain decomposition is given. Therefore, the time complexity of modifying the data structure is absorbed into that of the iteration of peeling algorithm.

Algorithm 16 return in-neighbors of node v on G_R.

```
1: procedure pred_{G_R}(v)
2:     T := ∅
3:     (p, i) := chain_index(v)
4:     for q = 0, 1, ..., k − 1 do
5:         if I_pq[i] = 1 then
6:             u := node_index(q, nearest_src(p, i, q))
7:             T ← T ∪ {u}
8:     return T
```

Algorithm 17 add to the data structure D a node with edges $(n+1, u)$ for each $u \in U$.

```
1:  procedure add(D, U)
2:      C ≔ {C_p}_{p=0}^{k-1}
3:      C_k ≔ {n + 1}
4:      C ← C ∪ C_k
5:      array LB[1 ... n + 1][0 ... k], UB[1 ... n + 1][0 ... k]
6:      for v = 1, 2, ..., n do
7:          for p = 0, 1, ..., k − 1 do
8:              LB[v][p] ← lb_p(v)
9:              UB[v][p] ← ub_p(v)
10:     for p = 0, 1, ... k − 1 do
11:         LB[n + 1][p] ← max({LB[u][p]; u ∈ U} ∪ (U ∩ C_p))
12:         UB[n + 1][p] ← (p, ∞)
13:     for v = 1, 2, ... n do
14:         LB[v][k] ← (k, 0)
15:         (p, i) ≔ chain_index(v)
16:         if (p, i) ≺ LB[n + 1][p] then UB[v][k] ← (k, 1)
17:         else UB[v][k] ← (k, ∞)
18:     LB[n + 1][k] ← (k, 0)
19:     UB[n + 1][k] ← (k, ∞)
20:     run a peeling iteration on LB, UB and C and get a minimal chain decomposition C′
21:     construct a new data structure D′ by D and C′
22:     replace D by D′
```

On the data structure of Section 3.2 or Section 3.4, the time complexity of adding nodes is $O(nk)$ per one node. Consider that adding n elements to an empty poset one by one with this adding algorithm. Then, it costs $O(n^2k)$ time. This matches the time complexity to construct the data structure of the poset with n elements, which is $O(n^2k)$ time and is mainly spent by the chain decomposition and conversions between G_R and G_C when its minimal chain decomposition is not given. Algorithm 17 shows the detail of the adding operation. We construct the tables of chainmerge LB and UB from line 5 to line 19.