Designing various algorithms based on DAG-pathwidth

Shinya Izu[1] and Jun Kawahara[1][0000−0001−7208−044X]

Yoshida Honmachi, Sakyo-ku, Kyoto, 606–8501, Japan
{izu.shinya.54w@st,jkawahara@i}.kyoto-u.ac.jp

Abstract. DAG (Directed Acyclic Graph)-pathwidth is a parameter that measures how closely a directed graph is to a directed path. This parameter is useful for designing parameterized algorithms to solve NP-hard problems even on DAGs. In this paper, we first design parameterized algorithms with DAG-pathwidth for various NP-hard problems even on DAGs. Specifically, we design fixed-parameter tractable (FPT) algorithms for the Directed Dominating Set Problem and the Max Leaf Outbranching Problem. Given a DAG with n vertices and a DAG-path-decomposition of width w, both problems can be solved exactly in $O(2^w wn)$ time. Similarly, we propose parameterized algorithms for the Directed Steiner Tree Problem and the k-Disjoint Path Problem. Next, we show the existence of a polynomial-time approximation algorithm for DAG-pathwidth that achieves an $O(\log^{3/2} n)$ approximation ratio by demonstrating the equivalence between constructing DAG-path-decomposition and solving one-shot Black Pebbling game. We also design an algorithm that, given an integer t and DAG H with l roots and at most d outdegree, either computes a DAG-path-decomposition of H with width $O(ld^t)$ or provides a witness that the DAG-pathwidth of H is greater than t.

Keywords: Graph algorithm · Computational complexity · Directed acyclic graph · Pathwidth.

1 Introduction

Path and *tree decompositions* [22,23] are prominent techniques for efficiently solving NP-hard problems on graphs. These methods transform the input graph into a path-like or tree-like structure, enabling the extension of dynamic programming algorithms originally designed for paths or trees to more general graph classes. The effectiveness of these decompositions is characterized by their width: *pathwidth* for path decompositions and *treewidth* for tree decompositions. A smaller pathwidth/treewidth indicates that the graph's structure is closer to a path/tree. When the width is bounded by a constant, it is sometimes possible to solve NP-hard problems in polynomial time with respect to the number of vertices.

For directed graphs, *directed pathwidth* was introduced by Reed [21], and *directed treewidth* by Johnson et al. [16]. Berwanger et al. [6] proposed a width

parameter, called *DAG-width*, specifically for directed acyclic graphs (DAGs). DAG-width measures how close a directed graph is to a DAG. However, if the input graph is a DAG, this parameter is always at most one, making it difficult to construct parameterized algorithms for NP-hard problems on DAGs.

Kasahara et al. [17] proposed *DAG-pathwidth*, which measures how close a directed graph is to a directed path. DAG-path decompositions follow the same rules as standard path decompositions, with the added constraint that for any vertex, its first appearance in a bag must respect the topological order of the original input graph, and all vertices within a strongly connected component of the original graph must appear together in one bag. This constraint facilitates the design of efficient fixed-parameter tractable (FPT) algorithms especially on DAGs. While they introduced DAG-pathwidth to analyze the discord k-independent set problem, we believe that this concept has broader applicability and can be a valuable tool for analyzing various NP-hard problems on DAGs. Although they demonstrated that computing DAG-pathwidth is NP-hard, they did not provide a practical algorithm for its computation.

Based on the above, our contributions are as follows:

1. We construct parameterized algorithms using DAG-pathwidth for various NP-hard problems on DAGs.
2. We propose approximation algorithms for DAG-path-decomposition and parameterized algorithms to construct decompositions with small width.

In Section 3, we design FPT algorithms for the Directed Dominating Set Problem and the Max Leaf Outbranching Problem on DAGs in $O(2^w wn)$ time, given a DAG with n vertices and a DAG-path-decomposition of width w. We also design an FPT algorithm for the Directed Steiner Tree Problem, which runs in $O(2^w(k+w)n+n^2)$ time when the size of the terminal set is k. Additionally, we propose a parameterized algorithm for the k-Disjoint Path Problem, which runs in $O((k+1)^w(w^2+k)n+n^2)$ time. At the end of Section 3, we show the advantages of DAG-pathwidth compared to treewidth using the Directed Edge Dominating Set Problem. While the proposed algorithms are designed for DAGs, they can also be applied to general directed graphs by performing a strongly connected component. In Section 4, we show the existence of an $O(\log^{3/2} n)$-approximation algorithm for DAG-pathwidth on DAGs by demonstrating the equivalence between the one-shot Black Pebbling Problem and the problem of computing the DAG-pathwidth. Finally, in Section 5, we design an algorithm that, given an integer t, and a DAG with maximum outdegree d, number of roots l, provides a DAG-path-decomposition with width $O(l \cdot d^t)$. This algorithm is based on the one for undirected path decompositions [7], and both of these algorithms utilize the graph embedding of complete trees.

Related work The pathwidth of an undirected graph was first proposed by Robertson et al. [22]. They also introduced the concept of treewidth [23]. Arnborg et al. [2] later demonstrated that it is NP-complete to determine whether the treewidth and pathwidth of a graph is at most k. They also studied various FPT algorithms using these parameters [3]. It is unknown there exists a

polynomial-time algorithm computing a tree decomposition with a width at most any constant factor of treewidth k. Similarly, it is also unclear whether constant-factor approximations for pathwidth are achievable. However, Amir [1] proposed a polynomial-time algorithm that approximates treewidth within a factor of $O(\sqrt{\log k})$. Tuukka [19] also proposed a 2-approximation algorithm with a time complexity of $2^{O(k)}$. Similarly, for pathwidth, Carla et al. [13] proposed a polynomial-time algorithm with an approximation ratio of $O(k\sqrt{\log k})$, where k is the treewidth. In addition, Cattell et al. [7] presented a polynomial-time algorithm that computes a path decomposition with a width of $O(2^{pw})$, where pw is the pathwidth.

2 Preliminaries

A *directed acyclic graph* (DAG) is a directed graph with no cycles. For a DAG $G = (V, E)$ and a vertex $v \in V$, we define the *predecessors* as $\mathsf{pred}(v) = \{u \in V \mid (u, v) \in E\}$ and *successors* as $\mathsf{suc}(v) = \{w \in V \mid (v, w) \in E\}$ of v. The vertices with indegree 0 are called the *roots*. A *directed tree* is a DAG that has the unique root r, and for any vertex v, there is the unique directed path from r to v.

We define the *DAG-path-decomposition* and the *DAG-pathwidth* [17]. Let $G = (V, E)$ be a directed graph. A *DAG-path-decomposition* of G is a sequence $X = (X_1, X_2, \ldots, X_s)$ $(X_i \subseteq V)$ that satisfies the following three conditions:

(DPD1) $X_1 \cup X_2 \cup \cdots \cup X_s = V$.
(DPD2) For every directed edge $(u, v) \in E$, there exists i such that $u, v \in X_i$ and $v \notin X_{i-1}$.
(DPD3) For any i, j, k $(1 \leq i \leq j \leq k \leq s)$, $X_i \cap X_k \subseteq X_j$.

In (DPD2), we interpret $X_0 = \emptyset$. Therefore, a directed edge (u, v) satisfies (DPD2) if $u, v \in X_1$. The *width* of X is defined as $\max_i\{|X_i| - 1\}$. The *DAG-pathwidth* of G is the minimum width over all possible DAG-path-decompositions of G. Note that in [17], (DPD2) is changed into the condition that there exists i such that $u, v \in X_i$ and $u \notin X_{i-1}$. In this study, we adopt the above definition for the sake of algorithmic simplicity.

The difference between (undirected) path decomposition, directed path decomposition, and DAG-path-decomposition is the second rule. The second rule of path decomposition is "for any edge $(u, v) \in E$, there exists an i such that $u, v \in X_i$," while that of directed path decomposition is "for any directed edge $(u, v) \in E$, there exist i, j $(i \leq j)$ such that $u \in X_i$, $v \in X_j$."

The DAG-pathwidth is a parameter that indicates how closely the structure of a directed graph resembles a directed path. From (DPD3), any vertex v is contained in consecutive bags. Additionally, (DPD2) and (DPD3) imply that for any edge (u, v), u and v first appear together in a single bag or u appears in a bag without v, followed by a bag containing both u and v. Thus, a DAG-path-decomposition can be interpreted as an operation of adding vertices to bags according to a topological order of the graph.

To facilitate designing a dynamic programming-based algorithm, we define the *nice DAG-path-decomposition* [17], which is a DAG-path-decomposition $X = (X_1, X_2, \ldots, X_s)$ of a directed graph $H = (V, E)$ that satisfies the following rules:

1. $X_1 = X_s = \emptyset$.
2. For any i $(2 \leq i \leq s-1)$, one of the following holds:
   - (introduce) There exists a strongly connected component $S \subseteq V$ such that $S \cap X_i = \emptyset$ and $X_{i+1} = X_i \cup S$.
   - (forget) There exists a vertex $v \in V$ such that $X_{i+1} = X_i \setminus \{v\}$.

If H is a DAG, each strongly connected component S consists of a single vertex. Thus, the introduce operation can be redefined that there exists a vertex $v \in V$ such that $\{v\} \cap X_i = \emptyset$ and $X_{i+1} = X_i \cup \{v\}$. It means, for any vertex, there exists exactly one introduce bag and one forget bag due to (DPD1) and (DPD3).

The nice DAG-path-decomposition simplifies the design of dynamic programming algorithms, as each bag involves either introducing or forgetting a single vertex for DAGs. It is shown in [17] that a nice DAG-path-decomposition with the same width as a given DAG-path-decomposition can be constructed in polynomial time. Moreover, the number of bags is not more than $2|V[H]| + 1$. For DAGs, the number is exactly $2|V[H]| + 1$.

3 Algorithms for various NP-hard problems on DAGs based on DAG-pathwidth

For NP-hard problems on a DAG, it is sometimes possible to construct a parameterized algorithm using the treewidth of the underlying graph of the DAG. However, the lack of edge direction information in tree decompositions might complicate algorithm construction. Conversely, DAG-path decompositions, which preserve edge direction, generally facilitate simpler algorithm design compared to approaches using tree decompositions.

In this section, we design algorithms for various NP-hard problems on DAGs based on DAG-pathwidth. Section 3.1 demonstrates a typical technique based on DAG-path-decomposition by showing an algorithm for the Directed Dominating Set Problem. Section 3.2 compares standard tree decomposition with DAG-path-decomposition, focusing on the Directed Edge Dominating Set Problem. Finally, we propose algorithms for other NP-hard problems on DAGs in Section 3.3.

3.1 Directed Dominating Set Problem

Given a DAG $G = (V, E)$, the Directed Dominating Set Problem (DiDS problem) asks us to find the smallest subset $S \subseteq V$ such that, for every $v \in V$, either $v \in S$ or there exists some $u \in S$ such that $(u, v) \in E$.

Let us construct a DAG-path-decomposition-based algorithm. Suppose that we are given a nice DAG-path-decomposition $(X_1, \ldots, X_s)$. We first define the

function DS. The function DS partitions each bag X_i into two disjoint vertex sets $A_i, B_i \subseteq X_i$ ($A_i \cup B_i = X_i, A_i \cap B_i = \varnothing$). Then, among the directed dominating sets of the subgraph G_i of G induced by $X_1 \cup X_2 \cup \cdots \cup X_i$, it finds the smallest directed dominating set that includes all elements of A_i and none of B_i. By computing this for all combinations of A_i and B_i, we obtain the minimum dominating set for G_i. By considering all $i = 1, \ldots, s$, the minimum dominating set of the input graph G is obtained. The function DS is defined as follows:

$$\mathsf{DS}(i, A_i, B_i) = \min \left\{ |S_i| \;\middle|\; \begin{array}{l} S_i \subseteq X_1 \cup X_2 \cup \cdots \cup X_i, S_i \text{ is a DiDS of } G_i, \\ A_i \subseteq S_i, B_i \cap S_i = \emptyset \end{array} \right\}. \quad (1)$$

DS computes the size of the minimum DiDS of G_i. We provide the recurrence formula for DS. If X_i introduces $v \in V$,

$$\mathsf{DS}(i, A_i, B_i) = \begin{cases} \mathsf{DS}(i-1, A_i \backslash \{v\}, B_i) + 1 & (v \in A_i) \\ \mathsf{DS}(i-1, A_i, B_i \backslash \{v\}) & (v \in B_i \text{ and } \mathsf{pred}(v) \cap A_i \neq \emptyset) \\ \infty & (\text{otherwise}) \end{cases}.$$

Informally, $\mathsf{pred}(v) \cap A_i \neq \emptyset$ means that one of the predecessors of v is in the dominating set, and thus v is dominated. (DPD2) ensures that all the predecessors of v must appear in some of bags $X_1, \ldots, X_{i-1}$ and in X_i, which facilitates the design of the algorithm.

The case of forget and the correctness of the algorithm is shown in Appendix A.1.

Theorem 1 (proof in Appendix A.1). *Given a DAG G and its nice DAG-path-decomposition of width w, there exists an algorithm that solves the DiDS problem on G in $O(2^w wn)$ time.*

3.2 Advantages of DAG-path-decomposition Compared to Tree Decomposition

In this subsection, we compare an algorithm using tree decomposition for the Directed Edge Dominating Set Problem (DEDS problem) [5] with an algorithm using DAG-path-decomposition and demonstrate the characteristics of DAG-path-decomposition.

For a directed graph $G = (V, E)$, a subset $S \subseteq E$ is a *Directed Edge Dominating Set (DEDS)* of G if, for any $(v, w) \in E$, either $(v, w) \in S$ holds or there exists some $u \in V$ such that $(u, v) \in S$. The *minimum DEDS (mDEDS)* is the DEDS S with the smallest $|S|$ among all DEDS. The DEDS problem is the problem of determining the size of the mDEDS of G.

The paper [14] proved that the DEDS problem remains NP-hard even for planar bounded degree DAGs. The paper [5] established the following result:

Proposition 1 ([5]). *If the treewidth of the underlying graph of a DAG G is at most tw, then there exists an FPT algorithm that solves the DEDS problem on G in $4^{2tw^2} 8^{2tw} n^{O(1)}$ time.*

In a standard tree decomposition, for an edge $(u, w) \in E$, w might appears in a bag before u, which complicates the design of a tree dcomposition-based algorithm, as evidenced by the intricate structure of the algorithm in [5]. In contrast, DAG-path-decomposition includes information on edge directions, enabling the sequential investigation of edge dominance relationships from the roots of the DAG. Together with the technique of converting the DAG into a line graph, described below, DAG-path-decomposition reduces the number of variables required for algorithmic design, leading to a simplified and intuitive algorithm.

In this subsection, we prove the following theorem:

Theorem 2 (proof in Appendix A.4). *Given a DAG G and its nice DAG-path-decomposition of width w, there exists an algorithm that solves the DEDS problem on G in $O(2^{w^2} w^2 n^2)$ time.*

Note that since we can regard a DAG-path-decomposition as a tree decomposition, the DAG-pathwidth of a graph is at least the treewidth of the graph. Therefore, we cannot directly compare $O(2^{w^2} w^2 n^2)$ and $4^{2tw^2} 8^{2tw} n^{O(1)}$.

Before constructing the above algorithm, we define the *line graph* of a directed graph. For a directed graph $G = (V, E)$, its *line graph* $L(G) = (V_L, E_L)$ is defined as follows: $V_L = \{e \mid e \in E\}$ and $E_L = \{(e_1, e_2) \mid e_1 = (u, v) \in E, e_2 = (v, w) \in E\}$. That is, the line graph of a directed graph is obtained by replacing the vertices and edges of the original graph while preserving edge directions.

A DAG-pathwidth-based algorithm reformulates the mDEDS problem as the problem of finding the minimum DiDS of the line graph and solves it using the DAG-path-decomposition of the line graph. We first prove the following two lemmas, which immediately lead to **Theorem 2**.

Lemma 1 (proof in Appendix A.2). *Given a DAG G with n vertices and its nice DAG-path-decomposition of width w, a nice DAG-path-decomposition of $L(G)$ with width at most w^2 can be constructed in $O(w^2 n)$ time.*

Lemma 2 (proof in Appendix A.3). *For a DAG G, the mDEDS of G and the minimum DiDS of $L(G)$ are equal.*

Note that the standard treewidth of the line graph of a graph G cannot be bounded by tw^2, where tw is the treewidth of G. Instead, it can be bounded only by $(tw + 1)\Delta(G) - 1$, where $\Delta(G)$ is the maximum degree of G [15].

3.3 Design of Various Parameterized Algorithms Using DAG-path-decomposition

Beyond the DEDS problem, DAG-path-decomposition enables the construction of simple parameterized algorithms for NP-hard problems on DAGs. In this section, we design parameterized algorithms using DAG-path-decomposition for the following four problems, which are NP-hard even on DAGs [11]. Proofs of the theorems are provided in the appendix.

First, we present an FPT algorithm for the Max Leaf Outbranching Problem (MaxLOB). Given a directed graph $G = (V, E)$ and a root $r \in V$ of G, the MaxLOB problem finds the maximum number of leaves in a directed spanning tree rooted at r.

Theorem 3 (proof in Appendix A.5). *Given a DAG G and its nice DAG-path-decomposition of width w, there exists an algorithm that solves the MaxLOB problem on G in $O(2^w wn)$ time.*

We also design a parameterized algorithm for the Disjoint Path Problem, defined as follows: Given a directed graph $G = (V, E)$ and k vertex pairs $(s_1, t_1), (s_2, t_2), \ldots, (s_k, t_k)$, let $\mathcal{P} = (P_1, P_2, \ldots, P_k)$ be a set of vertex-disjoint paths such that P_i is a path from s_i to t_i. The goal of the Disjoint Path Problem is to find the minimum sum of path lengths, $\sum_{i=1}^{k} |P_i|$.

Theorem 4 (proof in Appendix A.6). *Given a DAG G and its nice DAG-path-decomposition of width w, there exists an algorithm that solves the Disjoint Path Problem on G in $O((k+1)^w(w^2 + wk)n + n^2)$ time.*

Finally, we present an FPT algorithm for the Directed Steiner Problem (DST problem). Given a weighted directed graph $G = (V, E)$, a root $r \in V$ of G, and a set of terminals $R = \{t_1, t_2, \ldots, t_k\} \subseteq V$, the DST problem finds the minimum-weight directed tree rooted at r that spans all $t_i \in R$.

Theorem 5 (proof in Appendix A.7). *Given a DAG G and its nice DAG-path-decomposition of width w, there exists an FPT algorithm that solves the DST problem on G in $O(2^w(k + w)n + n^2)$ time.*

4 $O(\log^{3/2} n)$-approximation algorithm for computing DAG-pathwidth via one-shot black pebbling

The problem of determining the DAG-pathwidth for a general DAG is NP-hard [17]. In this section, we show the existence of a polynomial-time algorithm that provides a DAG-path-decomposition with an approximation ratio of $O(\log^{3/2} n)$ for a given DAG G with n vertices, via *one-shot Black Pebbling game (one-shot BP)* [24]. The *approximation ratio* of an algorithm $\mathcal{A}$ is defined as $\sup_I f(S_{\mathcal{A},I})/f(S_{\mathrm{opt},I})$, where $S_{\mathcal{A},I}$ and $S_{\mathrm{opt},I}$ are the outputs of $\mathcal{A}$ and the optimal algorithm for an input I, and $f(S)$ is the objective function value.

In one-shot BP, given a DAG $G = (V, E)$, a sequence of vertex sets (called a strategy) $P = (P_0, P_1, \ldots, P_t)$ ($P_i \subseteq V$) is constructed to satisfy the following rules:

(BP1) Initially, no pebble is placed on the vertices of G.
(BP2) *Pebble*: If no pebble is placed on $v \in V$ and all predecessors of v have pebbles, the player may place a pebble on v.
(BP3) *Unpebble*: The player may remove a pebble placed on v at any time.
(BP4) Each vertex of G must have a pebble placed on it at least once.

(BP5) Each vertex of G is pebbled only once.

(BP2) represents the operation of placing a pebble on a vertex, and (BP3) represents the operation of removing a pebble from a vertex. (BP1)–(BP4) form the *black pebbling* game, while the addition of (BP5) to the black pebbling yields one-shot BP. Note that any root of the DAG, which have no predecessors, can be pebbled at any time. Furthermore, the *space* of P is defined as $\max_i\{|P_i|\}$. The *pebbling number* of G is the minimum space over all possible strategies of G. Computing the pebbling number is NP-hard [24]. Per et al. [4] have shown the existence of the following approximation algorithm for one-shot BP.

Proposition 2 ([4]). *Given a DAG G with n vertices, there exists an algorithm that outputs a strategy for one-shot BP whose pebbling number is $O(\log^{3/2} n)$ times the minimum value.*

Using the above algorithm [4], we prove the following theorem.

Theorem 6. *Given a DAG G with n vertices, if the DAG-pathwidth of G is pw, there exists a polynomial-time algorithm that provides a nice DAG-path-decomposition with width $O(pw \cdot \log^{3/2} n)$.*

To prove Theorem 6, it is sufficient to show the following lemma. The proof of this lemma is in the appendix.

Lemma 3 (proof in Appendix B.1). *On a DAG, the construction of a strategy for one-shot BP and that of a nice DAG-path-decomposition are equivalent.*

5 Algorithm to find DAG-path-decomposition with width $O(ld^t)$

In this section, we propose an algorithm that, given a DAG H with maximum outdegree d and number of roots l, along with a non-negative integer t, either outputs a DAG-path-decomposition with width $O(ld^t)$ or provides a witness that the DAG-pathwidth of H is greater than t. For undirected path decompositions, Cattell et al. [7] showed that given an undirected graph H with n vertices and a non-negative integer t, there exists an $O(n)$ time algorithm that either provides a witness that the pathwidth of H is greater than t or returns a path decomposition with width $O(2^t)$.

Following their algorithm, we construct an algorithm that, for a DAG H with maximum outdegree d and number of roots l, either provides a witness that the DAG-pathwidth of H is greater than t or computes a DAG-path-decomposition with width $O(ld^t)$. This algorithm utilizes a *homeomorphic embedding*. A homeomorphic embedding of a directed graph $G_1 = (V_1, E_1)$ into another directed graph $G_2 = (V_2, E_2)$ is a mapping $f : V_1 \to V_2$ satisfying the following conditions: (i) f is an injective function, and (ii) there exists a bijective mapping g from E_1 to a set of vertex-disjoint paths in G_2 such that for any edge $e = (u, v) \in E_1$, the path $g(e)$ starts at $f(u)$ and ends at $f(v)$. Here, two paths are allowed to

share only their endpoints. We refer to homeomorphic embeddings simply as embeddings. In this section, we establish the following theorem for general DAGs. (Note that all the proofs of lemmas and theorems in this section are shown in the appendix.)

Theorem 7 (proof in Appendix C.7). *Let H be a DAG with maximum outdegree d and number of roots l, and let t be a non-negative integer. Then, exactly one of the following holds:*

(a) The DAG-pathwidth of H is at most $ld^{t+3} - 1$.
(b) H can be partitioned into two vertex sets X, Y such that $X \cup Y = V[H]$ and $X \cap Y = \varnothing$. Let A and B be the subgraphs of H induced by X and Y, respectively. In H, there exist only edges directed from A to B, and the DAG-pathwidth of A is greater than t.

The following corollary, which is the main result in this section, is obtained from Theorem 7.

Corollary 1 (proof in Appendix C.8) *Given a DAG H with l roots and maximum outdegree d, and an integer t, there exists an $O(n^2)$ time algorithm that either provides a witness that the DAG-pathwidth of H is greater than t, or provides a DAG-path-decomposition of width $O(ld^t)$.*

To prove Theorem 7, we first define a complete directed tree. For an integer d (≥ 1), a *complete d-ary directed tree* is a directed tree that the outdegree of every non-leaf vertex is exactly d and all paths from the root to leaves have the same length. The length plus one is called the *height* of the tree. We denote a complete d-ary directed tree of height h by $T_{h,d}$ ($h, d > 1$). The DAG-pathwidth of a complete d-ary directed tree is given as follows:

Lemma 4 (proof in Appendix C.2). *The DAG-pathwidth of $T_{h,d}$ is $h - 1$.*

We construct a parameterized algorithm that satisfies the statement of Theorem 7 based on the algorithm [7]. Given an input DAG $H = (V, E)$, we modify it to have a single root. Concretely, we create the DAG H' by adding a complete d-ary directed tree of height $\lceil \log_d l \rceil$ to H and by connecting its leaves to each root of H. We also define $M_{t,d,l}$ as a complete d-ary directed tree of height $\lceil \log_d l \rceil + t + 2$ (see Figure 1 in Appendix C.1).

The algorithm searches for an embedding of $M_{t,d,l}$ into H'. If such an embedding is found, it implies that the DAG-pathwidth of H' is at least that of $M_{t,d,l}$. The vertices of $M_{t,d,l}$ are called *tokens*. The algorithm places tokens onto vertices of H' preserving the tree structure. Once no further placement is possible, next placement is done after moving some tokens to other vertices. If all tokens of $M_{t,d,l}$ are used in the embedding, it indicates that an embedding from $M_{t,d,l}$ to H' has been found. When a token T is placed on a vertex of H', T is said to be *tokened*, and when it is not placed on any vertex, it is said to be *untokened*. Throughout the algorithm, each vertex of H' can be placed a token at most once.

We give a *label* to each token (i.e., each vertex of $M_{t,d,l}$) according to the following recursive rule:

Algorithm 1 GrowTokenTree

while there is a vertex $u \in H'$ with token T and a blue successor v of u whose all predecessors are placed token, and token T has an *untokened* child $T \cdot b$ **do**
  Place the token $T \cdot b$ on v.
end while
Output $\{v \in V[H'] \mid v \text{ is placed a token}\}$.

Algorithm 2 FindEmbedding

Place the root token λ on the root of H'.
$i \leftarrow 1$
Call GrowTokenTree and set the output to X_i.
while $|X_i| < |V[M_{t,d,l}]|$ and H' has at least one blue vertex **do**
  if there is a vertex $v \in H'$ with token T such that v has no blue successor and T has at most one *tokened* child **then**
    Remove T from H'.
    if T had one *tokened* child $T \cdot b$ **then**
      Replece all tokens $T \cdot b \cdot S$ with $T \cdot S$ on H'.
    end if
  else
    Output X_i as a witness of $pw > t$ and halt.
  end if
  $i \leftarrow i + 1$
  Call GrowTokenTree and set the output to X_i.
end while
if $|X_i| = |V[M_{t,d,l}]|$ **then**
  Output X_i as a witness of $pw > t$.
end if
if H' has at least one blue vertex **then**
  Output $(X_1, \ldots, X_i)$ as a DAG-path-decomposition with width $ld^{t+3} - 1$.
end if

1. The root token is labeled with the empty string λ.
2. If a parent token has label $m = b_1 b_2 \ldots b_{h-1}$, its child tokens are labeled $m \cdot 1, m \cdot 2, \ldots, m \cdot d$.

Note that since λ is the empty string, the child tokens of the root are labeled $1, 2, \ldots, m$. We use the symbol λ to represent the root token.

Each vertex of H' (not $M_{t,d,l}$) has a *color*, red or blue. Initially, all vertices of H' are assumed to be blue. When a token is placed on a vertex v of H', the color of v changes to red, and it remains red even if the token is removed. Tokens can only be placed on blue vertices, which means that each vertex of H' can have a token at most once. Since we promise that each vertex of H' can be placed a token at most once, the color of any vertex does not change from red to blue.

The algorithm consists of the main routine FindEmbedding, presented in Algorithm 2, and the subroutine GrowTokenTree, presented in Algorithm 1, which is called from FindEmbedding. GrowTokenTree greedily places tokens of $M_{t,d,l}$ onto

vertices of H' while preserving the condition of embeddings. A token can only be placed on a vertex whose all predecessors already have tokens. This process continues until no more tokens can be placed, at which point GrowTokenTree outputs the set of vertices in H' on which tokens are placed (see Figure 2 in Appendix C.1).

FindEmbedding attempts to output a sequence of vertex sets $(X_1, X_2, \dots)$ that form a DAG-path-decomposition. Initially, the algorithm places the token λ on the single root of H'. The algorithm executes GrowTokenTree and assigns the output (set of vertices of H') to X_1. Subsequently, for each $i = 1, 2, \dots$, the algorithm repeats the following procedure. Suppose that $(X_1, X_2, \dots, X_i)$ has been constructed. If X_i includes all tokens of $M_{t,d,l}$, it represents an embedding from $M_{t,d,l}$ to H', which indicates that the DAG-pathwidth of H' is at least that of $M_{t,d,l}$ (shown in Lemma 7). Therefore, FindEmbedding returns X_i as a witness of $pw > t$ (in Line 17 in Algorithm 2). Otherwise, if all vertices of H' are red, we can show that the sequence $(X_1, X_2, \dots, X_i)$ forms a DAG-path-decomposition of H' and its DAG-pathwidth is at most $ld^{t+3} - 1$ (in Lemma 6 and in Line 15). Otherwise (i.e., if not all tokens of $M_{t,d,l}$ are included in X_i and at least one vertex in H' remains blue), there may be potential for further execution of GrowTokenTree by modifying the token placement. If the algorithm succeeds the modification, it calls GrowTokenTree and assigns the output to X_{i+1}. If not, the algorithm returns X_i as a witness of $pw > t$ (in Lemma 8 and Line 11).

The idea of GrowTokenTree and FindEmbedding is similar to [7], but the main difference is the modification of the token placement. In [7], there always exists a removable token T until the algorithm terminates. In contrast, in this algorithm, there may be cases where such a token does not exist. This is because embedding of directed graphs is more difficult than that of undirected graphs, as it requires considering the directions of the edges.

We sketch the detail of the modification. The formal description is shown in the proof of Lemma 8. We consider removing the token T placed on a vertex $v \in V[H']$ that satisfies the following two conditions:

(a) All successor vertices of v are red.
(b) T has at most one *tokened* child token in $M_{t,d,l}$.

Condition (a) corresponds to a forget operation in the DAG-path-decomposition, and condition (b) ensures that the embedding remains valid. A token T can be removed from v only if both conditions are met. In this process, removing T might disconnect the *tokened* token set in $M_{t,d,l}$. To maintain connectivity, all tokens in the directed tree rooted at the *tokened* child $T \cdot b$ are relocated to the corresponding positions in the directed tree rooted at T. This replacement must proceed from the tokens closest to the root to those farther away to ensure that the replacement target tokens are always *untokened*. This operation generates tokens that switch from *tokened* to *untokened*, allowing GrowTokenTree to be executed again, with its output denoted as X_{i+1} (see Figure 3 in Appendix C.1). If no token satisfies both conditions (a) and (b), the algorithm outputs the last output of GrowTokenTree. This output represents a witness that the DAG-pathwidth of H' is at least that of $M_{t,d,l}$.

We first show the following lemma:

Lemma 5 (proof in Appendix C.3). *The subgraph G' induced by the tokened token set in $M_{t,d,l}$ is connected, and an embedding from G' to H' exists.*

The algorithm FindEmbedding outputs the following: (1) in Line 20, $(X_1, \ldots, X_i)$ as a DAG-path-decomposition with width $ld^{t+3} - 1$; (2) in Line 17, X_i as a witness of $pw > t$; or (3) in Line 11, X_i as a witness of $pw > t$. The following lemmas demonstrate that the algorithm works correctly in each of these cases.

Lemma 6 (proof in Appendix C.4). *If* FindEmbedding *outputs $(X_1, \ldots, X_i)$ under condition (1), then there exists a DAG-path-decomposition of H with width at most $ld^{t+3} - 1$.*

Lemma 7 (proof in Appendix C.5). *If* FindEmbedding *outputs X_i under condition (2), then H can be splitted into two disjoint subgraphs A and B such that $V[A] \cup V[B] = V[H]$ and $V[A] \cap V[B] = \varnothing$. Moreover, in H, only edges from A to B exist, and the DAG-pathwidth of A is greater than t but at most $ld^{t+3} - 1$.*

Lemma 8 (proof in Appendix C.6). *If* FindEmbedding *outputs X_i under condition (3), then the DAG-pathwidth of H is greater than t.*

Lemmas 6, 7, and 8 imply Theorem 7. The proof is in appendix C.7.

6 Conclusion

In this study, we designed dynamic programming algorithms for solving various NP-hard problems on DAGs, using DAG-pathwidth as a parameter. Additionally, we demonstrated the existence of an $O(\log^{3/2} n)$-approximation algorithm for computing DAG-pathwidth, as well as a parameterized algorithm for constructing a DAG-path-decomposition with width $O(ld^k)$. The former is demonstrated by showing the equivalence between constructing DAG-path-decomposition and solving one-shot Black Pebbling game, while the latter leverages DAG embeddings. Notably, the latter algorithm is independent of the number of vertices in input graph and can also serve as an algorithm for estimating the one-shot Black Pebbling number.

A key challenge for future work is to further reduce the width $O(ld^k)$ of the parameterized algorithm. In particular, since the maximum outdegree d could grow up to the number of vertices, we aim to explore methods to bound d by a constant.

Acknowledgements

This work was supported by JSPS KAKENHI Grant Numbers JP24H00690 and JP24H00697.

References

1. Amir, E.: Approximation algorithms for treewidth. Algorithmica **56**, 448–479 (2010)
2. Arnborg, S., Corneil, D., Proskurowski, A.: Complexity of finding embeddings in a k-tree. SIAM Journal on Algebraic Discrete Methods **8, no.2**, 277–284 (1987)
3. Arnborg, S., Proskurowski, A.: Linear time algorithms for np-hard problems restricted to partial k-trees. Discrete Applied Mathematics **23, no.1**, 11–24 (1989)
4. Austrin, P., Pitassi, T., Wu, Y.: Inapproximability of treewidth, one-shot pebbling, and related layout problems. International Workshop on Approximation Algorithms for Combinatorial Optimization **65**, 13–24 (2012)
5. Belmonte, R., Hanaka, T., Katsikarelis, I., Kim, E.J., Lampis, M.: New results on directed edge dominating set. Episciences, open access overlay journals **vol. 25:1** (2023)
6. Berwanger, D., Dawar, A., P. Hunter, S.K., Obdržálek, J.: The dag-width of directed graphs. Journal of Combinatorial Theory, Series B **102, no.4**, 900–923 (2012)
7. Cattell, K., Dinneen, M.J., Fellows, M.R.: A simple linear-time algorithm for finding path-decompositions of small width. Information processing letters **57**, 197–203 (1996)
8. Dziembowski, S., Faust, S., Kolmogorov, V., K.Pietrzak: Proofs of space. Advances in Cryptology - CRYPTO 2015 pp. 585–605 (2015)
9. Even, S., Itai, A., Shamir, A.: On the complexity of timetable and multicommodity flow problems. SIAM Journal on Computing **5**, 691–703 (1976)
10. Fomin, F.V., Grandoni, F., Kratsch, D., Lokshtanov, D., Saurabh, S.: Computing optimal steiner trees in polynomial space. Algorithmica **65**, 584–604 (2013)
11. Ganian, R., Hliněný, P., Kneis, J., Langer, A., Obdržálek, J., Rossmanith, P.: Digraph width measures in parameterized algorithmics. Discrete applied mathematics **168**, 88–107 (2014)
12. Gilbert, J.R., Lengauer, T., Tarjan, R.E.: The pebbling problem is complete in polynomial space. Proceedings of the eleventh annual ACM symposium on Theory of computing pp. 237–248 (1979)
13. Groenland, C., Joret, G., Nadara, W., Walczak, B.: Approximating pathwidth for graphs of small treewidth. ACM transactions on algorithms **19**, 1–19 (2023)
14. Hanaka, T., Nishimura, N., Ono, H.: On directed covering and domination problems. Discrete Applied Mathematics **259**, 76–99 (2019)
15. Harvey, D.J., Wood, D.R.: The treewidth of line graphs. Journal of Combinatorial Theory, Series B **132**, 157–179 (2018)
16. Johnson, T., Robertson, N., Seymour, P., Thomas, R.: Directed tree-width. Journal of Combinatorial Theory, Series B **82, no.1**, 138–154 (2001)
17. Kasahara, S., Kawahara, J., Minato, S., Mori, J.: Dag-pathwidth: Graph algorithmic analyses of dag-type blockchain networks. IEICE Transactions on Information and Systems **E106-D, No.3** (2023)
18. Kirousis, L., Papadimitriou, C.: Searching and pebbling. Theoretical Computer Science **47**, 205–218 (1986)
19. Korhonen, T.: A single-exponential time 2-approximation algorithm for treewidth. SIAM Journal on Computing pp. FOCS21–174 (2023)
20. Ravi, R., Agrawal, A., Klein, P.: Ordering problems approximated: single-processor scheduling and interval graph completion. Automata, Languages and Programming: 18th International Colloquium Madrid, Spain, July 8–12, 1991 Proceedings 18 pp. 751–762 (1991)

21. Reed, B.: Tree width and tangles: a new connectivity measure and some applications. Surveys in Combinatorics pp. 87–162 (1997)
22. Robertson, N., Seymour, P.: Graph minors. i. excluding a forest. Journal of Combinatorial Theory, Series B **35, no.1**, 39–61 (1983)
23. Robertson, N., Seymour, P.: Graph minors. iii. planar tree-width. Journal of Combinatorial Theory, Series B **36, no.1**, 49–64 (1984)
24. Sethi, R.: Complete register allocation problems. SIAM Journal on Computing **4**, 226–248 (1975)

A Proofs of Section 3

A.1 Algorithm to solve DiDS problem and the proof of Theorem 1

To construct an algorithm satisfying **Theorem 1**, we first define the function DS. The function DS partitions each bag X_i into two disjoint vertex sets $A_i, B_i \subseteq X_i$ ($A_i \cup B_i = X_i, A_i \cap B_i = \varnothing$). Then, among the directed dominating sets of the subgraph G_i induced by $X_1 \cup X_2 \cup \cdots \cup X_i$, it finds the smallest directed dominating set that includes all elements of A_i and none of B_i. By computing this for all combinations of A_i and B_i, we obtain the minimum dominating set for G_i. By considering all i, the minimum dominating set of the input graph G. The function DS is defined as follows:

$$\mathsf{DS}(i, A_i, B_i) = \min \left\{ |S_i| \;\middle|\; \begin{array}{l} S_i \subseteq X_1 \cup X_2 \cup \cdots \cup X_i, \\ S_i \text{ is a DiDS of } G_i, \\ A_i \subseteq S_i, B_i \cap S_i = \varnothing \end{array} \right\}. \tag{2}$$

DS computes the size of the minimum DiDS of G_i. Below, we provide the recurrence formula for DS, dividing cases based on whether X_i introduces or forgets a vertex.

- When X_i introduces $v \in V$:

$$\mathsf{DS}(i, A_i, B_i) = \begin{cases} \mathsf{DS}(i-1, A_i \backslash \{v\}, B_i) + 1 & (v \in A_i) \\ \mathsf{DS}(i-1, A_i, B_i \backslash \{v\}) & (v \in B_i \text{ and } \mathsf{pred}(v) \cap A_i \neq \varnothing) \\ \infty & (\text{otherwise}) \end{cases}.$$

- When X_i forgets $v \in V$:

$$\mathsf{DS}(i, A_i, B_i) = \min\{\mathsf{DS}(i-1, A_i \cup \{v\}, B_i), \mathsf{DS}(i-1, A_i, B_i \cup \{v\})\}.$$

Using DS, we define the algorithm $\mathsf{Compute}(P)$, which outputs the size of the minimum DiDS of G when given a nice DAG-path-decomposition P of G.

1. First Step: Set $\mathsf{DS}(0, \varnothing, \varnothing) = 0$.
2. Execution Step: For each X_i ($i = 1, 2, ..., s$) in P, compute $\mathsf{DS}(i, A_i, B_i)$ for all combinations of A_i and B_i.

3. Final Step: If $i = s$, output $\mathsf{DS}(s, \varnothing, \varnothing)$.

To demonstrate **Theorem 1**, it is sufficient to prove the following two Lemmas.

Lemma 9. Compute *returns the size of the minimum DiDS of* G.

Proof. It suffices to show that for each i, $\mathsf{DS}(i, A_i, B_i)$ satisfies the definition (2) of DS. We prove this by induction on i.

Base Case ($i = 0$): Clearly, definition (2) holds.

Inductive Step ($i = k \to i = k + 1$): Assume that $\mathsf{DS}(i, A_i, B_i)$ satisfies definition (2) for $i = k$. We consider cases where X_{k+1} introduces or forgets $v \in V$.

- Case 1: X_{k+1} introduces v
  If $v \in A_{k+1}$, then by (DPD2), v does not dominate any vertex in G_{k+1}. Thus, $\mathsf{DS}(k+1, A_{k+1}, B_{k+1})$ is equal to $\mathsf{DS}(k, A_{k+1} \backslash \{v\}, B_{k+1}) + 1$. By induction, this satisfies definition (2).
  If $v \in B_{k+1}$ and $\mathsf{pred}(v) \cap A_{k+1} \neq \varnothing$, then some vertex $u \in A_{k+1}$ dominates v. Thus, $\mathsf{DS}(k+1, A_{k+1}, B_{k+1}) = \mathsf{DS}(k, A_{k+1}, B_{k+1} \backslash \{v\})$, which also satisfies definition (2) by induction.
  Otherwise, if $v \in B_{k+1}$ and $\mathsf{pred}(v) \cap A_{k+1} = \varnothing$, no vertex in A_{k+1} dominates v. Hence, no valid minimum DiDS exists, and we set $\mathsf{DS}(k+1, A_{k+1}, B_{k+1}) = \infty$.
- Case 2: X_{k+1} forgets v
  Since $G_{k+1} = G_k$, the minimum DiDS of G_{k+1} is equal to the minimum DiDS of G_k. Thus, it is given by

  $$\min\{\mathsf{DS}(k, A_{k+1} \cup \{v\}, B_{k+1}), \mathsf{DS}(k, A_{k+1}, B_{k+1} \cup \{v\})\},$$

  which satisfies definition (2).

Thus, by induction, **Lemma 9** is proven.

Lemma 10. *Given a DAG G with n vertices and a DAG-path-decomposition P of width w,* Compute(P) *runs in $O(2^w wn)$ time.*

Proof. Since $|X_i| \leq w + 1$, the number of combinations of A, B is at most 2^{w+1}. The number of bags is $2|V| + 1$. Checking $\mathsf{pred}(v) \cap A_i \neq \varnothing$ takes $O(w)$ time. Thus, the overall complexity is $O(2^w wn)$.

A.2 Lemma 1

Proof. Suppose that a nice DAG-path-decomposition $X = (X_1, X_2, \ldots, X_s)$ of a DAG G with width w is given. For each $i = 1, 2, \ldots, s$, define $X_i^L = \{(u, v) \in E[G] \mid v \in X_i\}$. Then, $X^L = (X_1^L, X_2^L, \ldots, X_s^L)$ forms a DAG-path-decomposition of $L(G)$ with width at most w^2. We prove this below.

First, X^L satisfies (DPD1) for $L(G)$. Since the DAG-path-decomposition X of G ensures that each vertex appears in some bag by (DPD1), each edge of G must also be contained in some bag X_i^L.

Furthermore, X^L satisfies (DPD3) for $L(G)$. In X, (DPD3) ensures that the subgraph of X induced by the bags containing any given vertex is a non-empty path. Similarly, in X^L, the subgraph induced by the bags containing any given edge of G is also a non-empty path.

Additionally, X^L satisfies (DPD2) for $L(G)$. Consider a bag X_k in X that introduces a vertex v. By (DPD2) and (DPD3), for any vertex $u \in \mathsf{pred}(v)$, it holds that $u, v \in X_k$ and $v \notin X_{k-1}, u \in X_{k-1}$. Let e_v be any edge with head v, and let e_u be any predecessor of e_v in $L(G)$. Since e_u has head u, the edge (e_u, e_v) in $L(G)$ satisfies $e_u, e_v \in X_i^L$ and $e_v \notin X_{i-1}^L$. Extending this argument to all v, we conclude that X^L satisfies (DPD2) for $L(G)$.

Thus, X^L is a DAG-path-decomposition of $L(G)$. Letting the width of X^L be w^L, we note that the maximum in-degree δ_{in} of G is at most w. Therefore, $w^L \leq \delta_{in} w \leq w^2$ holds.

Moreover, the complexity of constructing X^L is $O(w^2 n)$. This is because the number of bags in X is $2|V[G]|+1$, and for each bag X_i, we only need to consider the predecessors of at most w vertices it contains. Thus, the number of edges added to X_i^L is at most $\delta_{in} w$. Consequently, the overall construction of X^L takes $O(\delta_{in} wn) \leq O(w^2 n)$ time.

A.3 Lemma 2

Proof. First, we show that if S is a DEDS of DAG G, then S is also a DiDS of $L(G)$. If $S \in E[G]$ is a DEDS, for any edge $(v, w) \in E[G]$, either $(v, w) \in S$ holds or there exists some edge $(u, v) \in S$. In $L(G)$, for any vertex $(v, w) \in V[L(G)]$, either $(v, w) \in S$ holds or there exists some vertex $(u, v) \in S$. Thus, S is a DiDS of $L(G)$. Similarly, if $S \in V[L(G)]$ is a DiDS of $L(G)$, then S is a DEDS of G. Hence, the DEDS of G and the DiDS of $L(G)$ correspond one-to-one. Therefore, the mDEDS of G and the minimum DiDS of $L(G)$ are equal.

A.4 Theorem 2

Proof. By **Lemma 2**, it suffices to compute the minimum DiDS of $L(G)$. By **Lemma 1**, given a nice DAG-path-decomposition X of G with width w, we can construct a DAG-path-decomposition X^L of $L(G)$ with width at most w^2 and convert to nice DAG-path-decomposition X_{nice}^L in polynomial time. Therefore, we can compute the minimum DiDS of $L(G)$ using the algorithm described in **Theorem 1**. Noting that X_{nice}^L has width $O(w^2)$ and $O(n^2)$ bags, the time complexity of this computation is $O(2^{w^2} w^2 n^2)$.

A.5 Algorithm to solve MaxLOB problem and the proof of Theorem 3

The input graph is assumed to be a DAG with a single root r. In this case, there must exist a directed spanning tree with the maximum number of leaves in G. It is sufficient to prove the following lemma.

Lemma 11. *For a connected DAG G, having exactly one root r is both necessary and sufficient for G to have a directed spanning tree.*

Proof. If there are two or more vertices with in-degree 0 in G, it is evident that no directed spanning tree can exist. Conversely, if there exists exactly one vertex r with in-degree 0, we arrange the vertices of G in a topological order starting with r, denoted as $r, v_1, v_2, \ldots, v_{n-1}$. Since G is a connected DAG, each vertex v_i $(i = 1, 2, \ldots, n-1)$ other than r must have at least one parent in the set $\{r, v_1, v_2, \ldots, v_{i-1}\}$. For each vertex v_i, there exists a path P_i from r to v_i. When considering the union of all such paths, if the result is a directed tree, then it is a directed spanning tree. If there is a cycle, we can remove it by replacing part of the paths with cycles. This can be done for all cycles, transforming the union of all paths into a directed tree, which will be a directed spanning tree.

If G has two or more vertices with in-degree 0, it is clear that no directed spanning tree can exist. Conversely, if there is exactly one vertex r with in-degree 0, we can arrange the vertices of G in a topological order starting from r, denoted as $r, v_1, v_2, \ldots, v_{n-1}$. Since G is a connected DAG, every vertex v_i $(i = 1, 2, \ldots, n-1)$ other than r must have at least one parent in the set $\{r, v_1, v_2, \ldots, v_{i-1}\}$. Therefore, for each vertex v_i, there exists a path P_i from r to v_i. Considering the union of all such paths, if the result forms a directed tree, it is a directed spanning tree. If a cycle appears, there exist two paths P_i and P_j $(i \neq j)$ and each containing an internal path P'_i and P'_j (P'_i and P'_j are vertex-disjoint paths with the same starting and ending points), respectively. In this case, by replacing P'_i in P_i with P'_j, the cycle formed by P'_i and P'_j can be removed. By performing this operation for all cycles, the union of all paths P_i can be transformed into a directed tree. This directed tree is a directed spanning tree.

To construct an algorithm satisfying **Theorem 3**, we define the function LOB. Let the nice DAG-path-decomposition of DAG $G = (V, E)$ be $P = (X_1, X_2, \ldots, X_s)$. For each i $(i = 1, 2, \ldots, s)$, let vertex sets $A_i, B_i \subseteq V$ satisfy $A_i \cup B_i = X_i$ and $A_i \cap B_i = \varnothing$. Let G_i be the subgraph of G induced by the vertex set $X_1 \cup X_2 \cup \cdots \cup X_i$. This algorithm finds all directed spanning trees of G_i in which A_i forms the leaf set and B_i does not form the leaf set. By calculating this for all combinations of A_i and B_i, the maximum number of leaves directed spanning tree for G_i is obtained. By performing this calculation for all i, the maximum number of leaves directed spanning tree for the input graph G is obtained. LOB is defined as follows, where the leaf set of a directed tree T is denoted as $\mathsf{Leaf}(T)$:

$$\mathsf{LOB}(i, A_i, B_i) = \max \left\{ |\mathsf{Leaf}(T_i)| \;\middle|\; \begin{array}{l} T_i = (V[T_i], E[T_i]) \text{ is a directed spanning} \\ \text{tree of } G_i \text{ rooted at } r, \\ V[T_i] = V[G_i], E[T_i] \subseteq E[G_i], \\ A_i \subseteq \mathsf{Leaf}(T_i), B_i \subseteq V[T_i] \backslash \mathsf{Leaf}(T_i) \end{array} \right\}. \tag{3}$$

The function LOB computes the MaxLOB for G_i. Below, we provide the recurrence relation for LOB, dividing the cases based on whether X_i introduces or forgets a vertex.

- When X_i introduces $v \in V$:

$$\mathsf{LOB}(i, A_i, B_i) = \begin{cases} \mathsf{LOB}(i-1, A_i \backslash \{v\}, B_i) + 1 & (v \in A_i \text{ and } \mathsf{pred}(v) \cap B_i \neq \varnothing) \\ \mathsf{LOB}(i-1, A_i, B_i \backslash \{v\}) & (v \in B_i \text{ and } \mathsf{pred}(v) \cap B_i \neq \varnothing) \\ -\infty & (otherwise) \end{cases}.$$

- When X_i forgets $v \in V$:

$$\mathsf{LOB}(i, A_i, B_i) = \max\{\mathsf{LOB}(i-1, A_i \cup \{v\}, B_i), \mathsf{LOB}(i-1, A_i, B_i \cup \{v\})\}.$$

Using LOB, we define the algorithm $\mathsf{Compute}(P)$ to output the solution of MaxLOB for G when given a nice DAG-path-decomposition P of G.

1. First Step: If $V_r = \{r\}$, output 1 as the solution. Otherwise, set $\mathsf{LOB}(1, \{r\}, \varnothing) = -\infty,\ \mathsf{LOB}(1, \varnothing, \{r\}) = 0$.
2. Execution Step: For each X_i $(i = 1, 2, ..., s)$ in P, compute $\mathsf{LOB}(i, A_i, B_i)$ for all combinations of A_i and B_i.
3. Final Step: If $i = s$, output $\mathsf{LOB}(s, \varnothing, \varnothing)$.

To demonstrate **Theorem 3**, it is sufficient to prove the following two Lemmas.

Lemma 12. $\mathsf{Compute}$ *outputs the solution of MaxLOB for* G.

Proof. From the First Step of $\mathsf{Compute}$, if G consists only of root r, it outputs 1. It is the solution to MaxLOB. Next, consider the case where G contains vertices other than r. The proof of **Lemma 12** follows by showing that $\mathsf{LOB}(i, A_i, B_i)$ satisfies the definition (3) of LOB for each i. This is done by induction on i. For convenience, we refer to non-leaf vertices of a tree as stems.
When $i = 1$, the First Step of $\mathsf{Compute}$ gives $\mathsf{LOB}(1, \{r\}, \varnothing) = -\infty$ and $\mathsf{LOB}(1, \varnothing, \{r\}) = 0$. Since r cannot be a leaf, it clearly satisfies the definition (3).
For $i = k$, assume that $\mathsf{LOB}(i, A_i, B_i)$ satisfies definition (3). We consider the cases where X_{k+1} introduces or forgets $v \in V$.

- Case 1: X_{k+1} introduces $v \in V$
  If $v \in A_{k+1}$, by (DPD2), $\mathsf{pred}(v) \subseteq (A_{k+1} \cup B_{k+1})$. Since B_{k+1} represents the stems of the directed spanning tree, v can only be considered a leaf if there exists a stem vertex $u \in \mathsf{pred}(v) \cap B_{k+1}$. By assumption, in G_k, we have $\mathsf{LOB}(k, A_{k+1} \setminus \{v\}, B_{k+1})$ as valid.
  If $v \in B_{k+1}$ and there exists an $u \in \mathsf{pred}(v) \cap A_{k+1}$, we can include v in the leaf set. The definition of LOB holds in this case as well.
  Otherwise, if $v \in B_{k+1}$ and $\mathsf{pred}(v) \cap A_{k+1} = \varnothing$, no valid directed spanning tree exists with v as a leaf, so $\mathsf{LOB}(k+1, A_{k+1}, B_{k+1}) = -\infty$.
- Case 2: When X_{k+1} forgets $v \in V$
  Since $G_{k+1} = G_k$, the MaxLOB of G_{k+1} equals the MaxLOB of G_k. Thus, we have the recurrence:

  $$\mathsf{LOB}(k{+}1, A_{k+1}, B_{k+1}) = \max\left\{\mathsf{LOB}(k, A_{k+1} \cup \{v\}, B_{k+1}), \mathsf{LOB}(k, A_{k+1}, B_{k+1} \cup \{v\})\right\},$$

  which satisfies the definition of LOB.

Thus, by induction, **Lemma 12** is proven.

Lemma 13. *Let G be a DAG with n vertices. Given the DAG-path-decomposition P of G with width w, the algorithm* $\mathsf{Compute}(P)$ *outputs the result in $O(2^w wn)$ time.*

Proof. The First Step and Final Step can each be computed in $O(1)$ time. We now analyze the time complexity for the Execution Step. For each X_i, note that $|X_i| \leq w + 1$, so the number of combinations of A and B is at most 2^{w+1}. Additionally, we have $0 \leq i \leq 2|V| + 1$. Furthermore, since the computation of $\mathsf{pred}(v)$ takes $O(w)$ time, the time complexity of computing $\mathsf{LOB}(i, A_i, B_i)$ is $O(w)$ if X_i introduces a vertex and $O(1)$ if X_i forgets a vertex. Therefore, the overall time complexity of $\mathsf{Compute}$ is $O(w2^w n)$.

A.6 Algorithm to Solve the Disjoint Path Problem and Proof of Theorem 4

To construct an algorithm satisfying **Theorem 4**, we define the function Cal. Let $X = (X_1, X_2, \ldots, X_s)$ be a nice DAG-path-decomposition of a DAG $G = (V, E)$. For each i ($i = 1, 2, \ldots, s$), let the vertex sets $A_i^1, A_i^2, \ldots, A_i^k, B_i \subseteq V$ satisfy $A_i^1 \cup A_i^2 \cup \cdots \cup A_i^k \cup B_i = X_i$, and assume that any two of $A_i^1, A_i^2, \ldots, A_i^k, B_i$ have no common elements. Define G_i as the subgraph of G induced by the vertex set $X_1 \cup X_2 \cup \cdots \cup X_i$, and set $\mathscr{A}_i = (A_i^1, A_i^2, \ldots, A_i^k)$.

The function Cal determines whether each A_i^m ($m = 1, 2, \ldots, k$) can be part of a vertex-disjoint path starting from s_m for each G_i. Among such combinations of vertex-disjoint paths, it finds the one with the minimum total path length. By computing this for all possible combinations of $A_i^1, A_i^2, \ldots, A_i^k, B_i$, we obtain the minimum total length of vertex-disjoint paths in G_i. Performing this calculation for all i yields the minimum total length of vertex-disjoint paths in the input graph G. The function Cal is defined as follows:

$$\mathsf{Cal}(i, \mathscr{A}_i, B_i) = \min \sum_{m=1}^{k} (|P_i^m| - 1). \tag{4}$$

Here, the vertex sets P_i^m $(i \leq m \leq k)$ satisfy $P_i^m \subseteq X_1 \cup X_2 \cup \cdots \cup X_i$, forming a vertex-disjoint path starting from s_m. Moreover, for $m' \neq m$, we require that $A_i^m \subseteq P_i^m$, $A_i^{m'} \cap P_i^m = \varnothing$, and $B_i \cap P_i^m = \varnothing$. Thus, Cal calculates the minimum total length of k vertex-disjoint paths starting from s_m in G_i.

Next, we provide the recurrence formulas for computing Cal. The computation is divided into cases based on whether each X_i introduces or forgets a vertex. Let $S = \{s_1, s_2, \ldots, s_k\}$ and $T = \{t_1, t_2, \ldots, t_k\}$ be the sets of start and end vertices, respectively.

- When X_i introduces $v \in S$ $(v = s_m)$:

$$\mathsf{Cal}(i, \mathscr{A}_i, B_i) = \begin{cases} 0 & (A_i^m = \{v\}) \\ \infty & (\text{otherwise}) \end{cases}. \tag{5}$$

- When X_i introduces $v \in T$ $(v = t_m)$:

$$\mathsf{Cal}(i, \mathscr{A}_i, B_i) = \begin{cases} \mathsf{Cal}(i-1, \mathscr{A}_i^m, B_i) + 1 & (v \in A_i^m \text{ and there exists } w \in \mathsf{pred}(v) \cap A_i^m \\ & \text{such that } \mathsf{suc}(w) \cap A_i^m = \{v\}) \\ \infty & (\text{otherwise}) \end{cases}. \tag{6}$$

- When X_i introduces $v \in V \backslash (S \cup T)$:

$$\mathsf{Cal}(i, \mathscr{A}_i, B_i) = \begin{cases} \mathsf{Cal}(i-1, \mathscr{A}_i^m, B_i) + 1 & (v \in A_i^m \text{ and there exists } w \in \mathsf{pred}(v) \cap A_i^m \\ & \text{such that } \mathsf{suc}(w) \cap A_i^m = \{v\}) \\ \mathsf{Cal}(i-1, \mathscr{A}_i^m, B_i \backslash \{v\}) & (v \in B_i) \\ \infty & (\text{otherwise}) \end{cases}. \tag{7}$$

- When X_i forgets $v \in V$:

$$\mathsf{Cal}(i, \mathscr{A}_i, B_i) = \min\{\min_{1 \leq m \leq k} \{\mathsf{Cal}(i-1, \overline{\mathscr{A}}_i^m, B_i)\}, \mathsf{Cal}(i-1, \mathscr{A}_i^m, B_i \cup \{v\})\}. \tag{8}$$

For any $v \in V$, define $\mathscr{A}_i^m$ and $\overline{\mathscr{A}}_i^m$ as follows:

$$\mathscr{A}_i^m = (A_i^1, A_i^2, \ldots, A_i^m \backslash \{v\}, \ldots, A_i^k).$$
$$\overline{\mathscr{A}}_i^m = (A_i^1, A_i^2, \ldots, A_i^m \cup \{v\}, \ldots, A_i^k).$$

Using Cal, we present the algorithm $\mathsf{Compute}(P)$ that outputs a solution to the Disjoint Path Problem for a given nice DAG-path-decomposition P of DAG G.

1. Preprocessing: If the input graph consists of a single vertex $s_1 = t_1$, output 0. Otherwise, remove all incoming edges to each vertex $t \in T$. Let G be the resulting graph.
2. First Step: Set $\mathsf{Cal}(0, (\varnothing, \varnothing, \ldots, \varnothing), \varnothing) = 0$.
3. Execution Step: For each X_i $(i = 1, 2, \ldots, s)$ in P, compute $\mathsf{Cal}(i, \mathscr{A}_i, B_i)$ for all combinations of $\mathscr{A}_i, B_i$.
4. Final Step: If $i = s$, output $\mathsf{Cal}(s, (\varnothing, \varnothing, \ldots, \varnothing), \varnothing)$.

To demonstrate **Theorem 4**, it is sufficient to prove the following two Lemmas.

Lemma 14. $\mathsf{Compute}$ *outputs a solution to the Disjoint Path Problem of* G.

Proof. If the input graph G consists of a single vertex $s_1 = t_1$, then $\mathsf{Compute}$ outputs 0 due to the preprocessing step. This is clearly a valid disjoint path of G. Now, suppose that G consists of more than one vertex. When a bag X_i introduces a vertex $v \in V$, v is included in exactly one of $A_i^1, A_i^2, \ldots, A_i^k, B_i$. By considering this property from 1 to i, we can establish that each path P_i^m remains a vertex-disjoint path. Additionally, let X_{i_s} be the bag introducing a starting vertex $s_m \in S$. Any vertex introduced at $X_{i'}$ $(i' < i_s)$ does not belong to the path $P_{i_s}^m$ because if a vertex u is introduced at $X_{i'}$ and $A_{i'}^m = \{u\}$, then $\mathsf{pred}(u) \cap A_{i'}^m = \varnothing$. Consequently, due to condition 7, Cal outputs ∞. Similarly, for an endpoint $t_m \in S$, let X_{i_t} be the bag introducing it. Any vertex introduced at $X_{i'}$ $(i_t < i')$ does not belong to the path $P_{i'}^m$ because if a vertex w is introduced at X_j $(i_t < j)$ and $w \in \mathsf{suc}(t_m)$ in the original graph before preprocessing, then by the rule of DAG-path-decomposition, if $w \in A_j^m$, it must be that $t_m \in A_j^m$. If $\mathsf{pred}(w) \cap (A_j^m \setminus \{t_m\}) = \varnothing$, then by preprocessing, $\mathsf{pred}(w) \cap A_j^m = \varnothing$, making Cal return ∞ due to condition 7. Otherwise, if there exists a vertex $p \in \mathsf{pred}(w) \cap A_j^m$ $(p \neq t_m)$, then there exists a vertex $q \in A_j^m$ $(p \in \mathsf{pred}(q) \cap (A_j^m \setminus \{w\})$ and $\mathsf{suc}(p) \cap (A_j^m \setminus \{w\}) = \{q\})$. Since preprocessing removes edge (t_m, w), we find that $\{q, w\} \subseteq \mathsf{suc}(p) \cap A_j^m$, making Cal return ∞. Using the same argument, any vertex w' introduced after X_j and included in A_l^m $(j < l)$ will result in Cal returning ∞. Thus, vertices introduced after X_{i_t} do not get added to the path $P_{i_t}^m$.

With the above observations, it suffices to prove that for each i $(i_s \leq i \leq i_t)$, there exists some m $(1 \leq m \leq k)$ such that $\mathsf{Cal}(i, \mathscr{A}_i, B_i)$ satisfies the definition (4). We prove this by induction on i.

Base Case: When $i = i_s$, equation 5 ensures that if $A_{i_s}^m = \{s_m\}$, then $\mathsf{Cal}(i_s, \mathscr{A}_{i_s}, B_{i_s}) = 0$, indicating that the path $P_{i_s}^m$ consists only of s_m with length 0, which clearly satisfies definition (4). If $A_{i_s}^m \neq \{s_m\}$, then Cal outputs ∞, implying that no path starting at s_m is constructed, which also satisfies definition (4).

Inductive Step: Assume that for some i ($i_s \leq i < i_t$), $\mathsf{Cal}(i, \mathscr{A}_i, B_i)$ satisfies definition (4). We consider two cases based on whether X_{i+1} introduces or forgets a vertex.

- Case 1: X_{i+1} introduces $v \in V$
  If $v \in A^m_{i+1}$, let u be the last introduced vertex in A^m_{i+1} before v. By (DPD2), an edge (u, v) must exist, or an edge (u', v) exists for some $u' \in A^m_{i+1}$. If (u, v) exists, equation 7 ensures that $\mathsf{Cal}(i+1, \mathscr{A}_{i+1}, B_{i+1}) = \mathsf{Cal}(i, \mathscr{A}^m_{i+1}, B_{i+1}) + 1$. By assumption, $\mathsf{Cal}(i, \mathscr{A}^m_{i+1}, B_{i+1})$ represents the minimal total path length when m-th path extends from s_m to u. Since adding v to the path increases length by 1, $\mathsf{Cal}(i+1, \mathscr{A}_{i+1}, B_{i+1})$ maintains the minimum path sum and satisfies definition (4).
  If $v \in B_{i+1}$, since $v \notin A^m_{i+1}$, the total minimum path length remains $\mathsf{Cal}(i, \mathscr{A}^m_{i+1}, B_{i+1} \setminus \{v\})$, which satisfies definition (4).
- Case 2: X_{i+1} forgets $v \in V$
  Since $G_{i+1} = G_i$, the total path length remains unchanged. Thus, the two cases—whether v is part of some path or in B_i—are considered. Taking the minimum of both values from the previous step, equation 8 correctly determines the minimum path sum, satisfying definition (4).

By induction, **Lemma 14** is proven.

Lemma 15. *Let the number of vertices in the DAG G be n. Given a DAG-path-decomposition P of G with width w, the function* $\mathsf{Compute}(P)$ *computes the result in $O((k+1)^w(w^2+k)n+n^2)$ time, where k is the size of the terminal set.*

Proof. In preprocessing, it can be determined in $O(1)$ time whether G consists of a single vertex. Also, removing all edges entering each $t \in T$ takes $O(n^2)$ time. The First Step and Final Step can each be computed in $O(1)$ time. Below, we analyze the time complexity of the Execution Step. For each X_i, notice that $|X_i| \leq w+1$. The number of combinations of $A^1_i, A^2_i, \ldots, A^k_i, B_i$ is at most $(k+1)^{w+1}$. Furthermore, we know that $0 \leq i \leq 2|V|+1$. Additionally, since the calculation of $\mathsf{pred}(v)$ and $\mathsf{suc}(v)$ takes $O(w)$ time, the time complexity of $\mathsf{Cal}(i, \mathscr{A}_i, B_i)$ is $O(w^2)$ when X_i is an "introduce" operation, and $O(k)$ when it is a "forget" operation. Therefore, the total time complexity of $\mathsf{Compute}$ is $O((k+1)^w(w^2+k)n+n^2)$.

By modifying the above algorithm, we can construct algorithms for solving related problems, such as the edge-disjoint path problem.

A.7 Algorithm to Solve the DST Problem and the Proof of Theorem 5

To construct an algorithm satisfying **Theorem 5**, we define the function ST. Let the nice DAG-path-decomposition of the DAG $G = (V, E)$ be $P = (X_1, X_2, \ldots, X_s)$. For some i ($i = 1, 2, \ldots, s$), let $A_i, B_i \subseteq V$ satisfy $A_i \cup B_i = X_i$ and $A_i \cap B_i = \varnothing$. Let G_i be the subgraph of G induced by the vertex set $X_1 \cup X_2 \cup \cdots \cup X_i$.

The function ST computes the directed Steiner tree that includes all vertices in $R \cap V[G_i]$ for each G_i. The Steiner tree contains all vertices in A_i, excludes all vertices in B_i, and has the minimum size. By computing this for all combinations of A_i and B_i, we obtain the minimum directed Steiner tree for G_i. By repeating this computation for all i, we eventually obtain the minimum directed Steiner tree for the entire input graph G. Let the weight of edge e be denoted by $d(e)$. We define the function ST as follows:

$$\mathsf{ST}(i; A_i, B_i) = \min \left\{ \sum_{(u,v) \in E[G_{T_i}]} d(u,v) \;\middle|\; \begin{array}{l} T_i \subseteq X_1 \cup X_2 \cup \cdots \cup X_i \\ G_{T_i} \text{ is a directed tree with root } r \\ \text{ on } G_i \\ V[G_{T_i}] = T_i, E[G_{T_i}] \subseteq E[G_i] \\ A_i \subseteq T_i, B_i \cap T_i = \varnothing \\ \forall t \in R \cap G_i, t \in T_i \end{array} \right\}. \tag{9}$$

For each i, if a directed tree G_{T_i} exists that satisfies the above conditions, we call it the optimal directed Steiner tree and denote it by $G^{opt}_{T(i,A_i,B_i)}$. The tree $G^{opt}_{T(s,\varnothing,\varnothing)}$ is the minimum directed Steiner tree (minimum-DST).

The following provides the computation formula for ST. We compute it by distinguishing between the cases when X_i introduces or forgets a vertex. Note that since the input graph is a DAG, any strongly connected component introduced in a nice DAG-path-decomposition consists of only a single vertex.

- When X_i introduces a vertex $v \in V$

$$\mathsf{ST}(i; A_i, B_i) = \begin{cases} \mathsf{ST}(i-1; A_i \setminus \{v\}, B_i) + \min\limits_{w \in \mathsf{pred}(v) \cap A_i} d(w,v) & \text{(if } v \in A_i \text{ and} \\ & \mathsf{pred}(v) \cap A_i \neq \varnothing) \\ \mathsf{ST}(i-1; A_i, B_i \setminus \{v\}) & \text{(if } v \in B_i \text{ and} \\ & v \notin R \cup \{r\}) \\ \infty & \text{(otherwise)} \end{cases}. \tag{10}$$

- When X_i forgets a vertex $v \in V$

$$\mathsf{ST}(i; A_i, B_i) = \min \{\mathsf{ST}(i-1; A_i \cup \{v\}, B_i), \mathsf{ST}(i-1; A_i, B_i \cup \{v\})\}. \tag{11}$$

When given a nice DAG-path-decomposition P of DAG G, root r, and terminal set R, the following algorithm $\mathsf{Compute}(P)$ outputs the total weight of the minimum-DST of G that contains all vertices in R:

1. Preprocessing: Let V_r be the set of vertices reachable from the root r in G. For each bag X_i of P, remove any vertex $v \in V \setminus V_r$ from X_i. The resulting sequence of vertex sets is converted back into a nice DAG-path-decomposition, which we denote as $P' = (X_1, X_2, \ldots, X_{s'})$ for convenience.

2. First Step: Set $\mathsf{ST}(1; \{r\}, \varnothing) = 0$.
3. Execution Step: For each X_i ($i = 2, 3, \ldots, s'$), compute $\mathsf{ST}(i; A_i, B_i)$ for all combinations of A_i and B_i.
4. Final Step: If $i = s'$, output $\mathsf{ST}(s'; \varnothing, \varnothing)$.

To demonstrate **Theorem 5**, it is sufficient to prove the following two lemmas.

Lemma 16. *The function* Compute *outputs the total weight of the minimum-DST of G, with root r and containing all vertices in R.*

Proof. First, we show that the solution remains unchanged after preprocessing. Since V_r is the set of vertices reachable from the root r, any $v \in V \backslash V_r$ will not be included in the desired minimum-DST. Therefore, even if we consider the graph $G' = G[V \backslash V_r]$, the solution does not change. Moreover, P' is a nice DAG-path-decomposition of G', and using P' as input to ST will yield the same result. Additionally, the width of P' does not exceed that of P.

Next, we show that after preprocessing, Compute outputs the minimum-DST of G that contains all vertices in R. This can be shown by demonstrating that equations (10) and (11) correspond to definition (9) for each i, which we prove by mathematical induction on i.

When $i = 1$, clearly $\mathsf{ST}(1; \{r\}, \varnothing) = 0$ satisfies definition (9).

Assume that for $i = k$ ($1 \leq k < s'$), equations (10) and (11) satisfy definition (9). We will now show that for $i = k+1$, equations (10) and (11) hold definition (9) as well.

– Case 1: X_{k+1} introduces v
  We consider the case where $v \in A_{k+1}$ and $v \in B_{k+1}$ separately. When $v \in A_{k+1}$, if $\mathsf{pred}(v) \cap A_{k+1} = \varnothing$, then there are no predecessors of v in A_i. In this case, by (DPD2), since no predecessors of v are introduced after X_{k+1}, the directed tree containing v has v as one of its roots. Since this tree cannot have r as its only root, we set $\mathsf{ST}(k+1; A_{k+1}, B_{k+1}) = \infty$ to indicate that the directed tree $G^{opt}_{T(k, A_{k+1}, B_{k+1})}$ represented by equation (9) does not exist. On the other hand, if $\mathsf{pred}(v) \cap A_{k+1} \neq \varnothing$, then there exists at least one predecessor $w \in A_{k+1} \backslash \{v\}$ of v. If a directed tree $G^{opt}_{T(k, A_{k+1} \backslash \{v\}, B_{k+1})}$ exists, then $w \in G^{opt}_{T(k, A_{k+1} \backslash \{v\}, B_{k+1})}$. By the preprocessing operation, w is reachable from r. Therefore, v is also reachable from r through w, and $G^{opt}_{T(k, A_{k+1} \backslash \{v\}, B_{k+1}) \cup \{v\}}$ is a directed tree with r as its root. The minimal total weight at that point is equal to $\sum_{(u,v) \in E[G^{opt}_{T(k, A_{k+1} \backslash \{v\}, B_{k+1})}]} d(u, v) + \min_{w \in \mathsf{pred}(v) \cap A_i} d(w, v)$. By assumption, $\mathsf{ST}(k; A_k, B_k) = \sum_{(u,v) \in E[G^{opt}_{T(k, A_k, B_k)}]} d(u, v)$, and since $A_{k+1} \backslash \{v\} = A_k$ and $B_{k+1} = B_k$, we conclude that $\mathsf{ST}(k+1; A_{k+1}, B_{k+1})$ is represented by equation (9).
  Next, consider the case where $v \in B_{k+1}$. When $v \in R \cup \{r\}$, $\mathsf{ST}(k+1; A_{k+1}, B_{k+1}) = \infty$, indicating that no directed tree $G^{opt}_{T(k, A_{k+1}, B_{k+1})}$ exists as represented by equation (9). If $v \notin R \cup \{r\}$, then by equation (9),

v does not belong to the directed tree that is the solution. In this case, we have $G^{opt}_{T(k+1,A_{k+1},B_{k+1}\setminus\{v\})} = G^{opt}_{T(k,A_k,B_k)}$. By assumption, $\mathsf{ST}(k; A_k, B_k) = \sum_{(u,v)\in E[G^{opt}_{T(k,A_k,B_k)}]} d(u,v)$, and since $A_{k+1} = A_k$ and $B_{k+1}\setminus\{v\} = B_k$, we conclude that $\mathsf{ST}(k+1; A_{k+1}, B_{k+1})$ is represented by equation (9).

– Case 2: X_{k+1} forgets v
From $G_{k+1} = G_k$, the directed tree $G^{opt}_{T(k+1,A_{k+1},B_{k+1})}$ is equal to either $G_{T_A} = G^{opt}_{T(k,A_k,B_k)}$ when $v \in A_k$, or $G_{T_B} = G^{opt}_{T(k,A_k,B_k)}$ when $v \in B_k$, whichever has the smaller total weight. By assumption, if $v \in A_k$, then $\mathsf{ST}(k; A_k, B_k) = \sum_{(u,v)\in E[G^{opt}_{T_A}]} d(u,v)$, and if $v \in B_k$, then $\mathsf{ST}(k; A_k, B_k) = \sum_{(u,v)\in E[G^{opt}_{T_B}]} d(u,v)$. Furthermore, if $v \in A_k$, then $A_{k+1} \cup \{v\} = A_k$ and $B_{k+1} = B_k$; if $v \in B_k$, then $A_{k+1} = A_k$ and $B_{k+1} \cup \{v\} = B_k$. Therefore, we conclude that $\mathsf{ST}(k+1; A_{k+1}, B_{k+1})$ is represented by equation (9).

Thus, for $i = k+1$, both equations (10) and (11) represent equation (9). By mathematical induction, **Lemma 16** is proven.

Lemma 17. *Let the number of vertices in the DAG $G = (V, E)$ be n. Given a DAG-path-decomposition P of G with width w, root $r \in V$, and terminal set $R \subseteq V$, where $k = |R|$, the function* $\mathsf{Compute}(P, r, R)$ *computes the optimal solution in $O(2^w(k+w)n + n^2)$ time.*

Proof. In preprocessing, the calculation of the set of vertices reachable from r takes $O(n^2)$ time. The First Step and Final Step can each be computed in $O(1)$ time. Now, let's analyze the time complexity of the Execution Step. For each X_i, notice that $|X_i| \leq w+1$, so the number of combinations of A_i and B_i is at most 2^{w+1}. Also, $0 \leq i \leq 2|V|+1$. Additionally, when introducing a vertex $v \in V$ in X_i, the computation of $\mathsf{pred}(v) \cap A_i \neq \varnothing$ and min, as well as the check for $v \in R \cup \{r\}$, each takes $O(w)$, $O(w)$, and $O(k)$ time, respectively. Therefore, if X_i introduces a vertex, the time complexity of calculating $\mathsf{ST}(i; A_i, B_i)$ is $O(k+w)$, and if X_i forgets a vertex, the time complexity is $O(1)$. Hence, the time complexity of the Execution Step is $O(2^w(k+w)n)$. Thus, the overall time complexity of $\mathsf{Compute}$ is $O(2^w(k+w)n + n^2)$.

Furthermore, by a simple extension of the above algorithm, we can efficiently solve the vertex-weighted directed Steiner tree problem.

Theorem 8. *Given a vertex-weighted DAG G with terminal size $k = |R|$ and a nice DAG-path-decomposition of G with width w, there exists an FPT algorithm that solves the vertex-weighted DST problem for G in $O(2^w(k+w)n + n^2)$ time.*

B Proof of Section 4

B.1 Lemma 3

Proof. First, we show that if X is a nice DAG-path-decomposition of a DAG G, then X is also a strategy of one-shot BP. Let $X = (X_1, X_2, \ldots, X_s)$ be

a nice DAG-path-decomposition of G. Since $X_1 = \varnothing$ in a nice DAG-path-decomposition, it satisfies (BP1). By (DPD1) and (DPD3), every vertex is introduced exactly once, satisfying (BP4) and (BP5). Furthermore, by the rule of introduce and (DPD2), if $v \in V$ is introduced at X_i, then for any $(u, v) \in E$, it holds that $u \in X_{i-1}$. This satisfies (BP2). Additionally, since each vertex is forgotten exactly once, the forget and unpebble operations are clearly equivalent, thus satisfying (BP3). Therefore, the introduce and forget operations in X correspond to the pebble and unpebble operations, respectively, proving that X is a one-shot BP of G.

Next, we show that if P is a one-shot BP of G, then P is a nice DAG-path-decomposition of G. Let $P = (P_1, P_2, \ldots, P_t)$ be a strategy of one-shot BP of G. Since $P_1 = \varnothing$, it satisfies the initial condition of a nice DAG-path-decomposition. Furthermore, each vertex $v \in V$ is pebbled and unpebbled exactly once in P, satisfying (DPD1). Additionally, if a vertex v is pebbled and unpebbled at P_i and P_{k+1} $(1 \leq i \leq k \leq t-1)$, then by(BP5), no vertex is pebbled more than once. Hence, for any P_j $(i \leq j \leq k)$, it holds that $v \in P_j$. Applying this argument to all vertices in V, we conclude that (DPD3) is satisfied. Moreover, by the pebbling rule, if $v \notin P_{i-1}$ and $u \in P_{i-1}$ for any $u \in \mathsf{pred}(v)$, then we can construct $P_i = P_{i-1} \cup \{v\}$. This implies that $u, v \in P_i$ and $v \notin P_{i-1}$, thereby satisfying (DPD2). It means the pebbling operation corresponds to the introduce operation. Furthermore, the unpebble operation clearly satisfies the forget condition of a nice DAG-path-decomposition. Thus, P is a nice DAG-path-decomposition of G.

C Figures and Proofs of Section 5

C.1 Figure 1, 2, 3, 4, 5

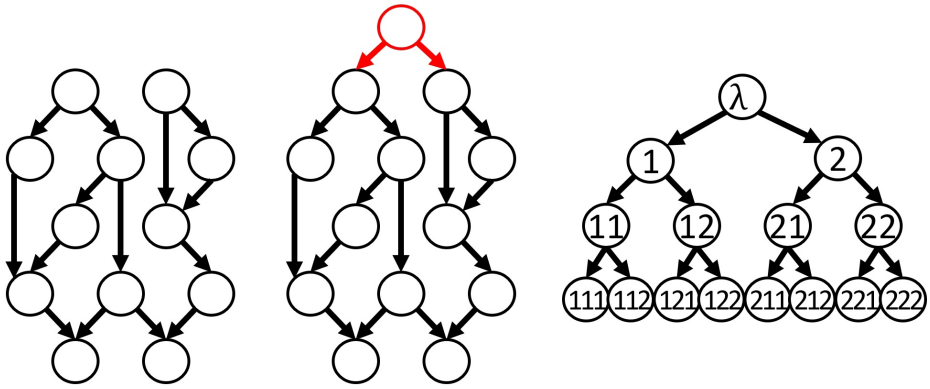

Fig. 1. given H with $d = 2, l = 2$ (left) and $t = 1$, the construction of H' (center) and $M_{t,d,l}$ (right). H' is obtained by adding a complete directed d-ary tree (red part) connected to each root of H. Moreover, the labeling of $M_{t,d,l}$ is constructed recursively by appending one character to the right of the parent's label.

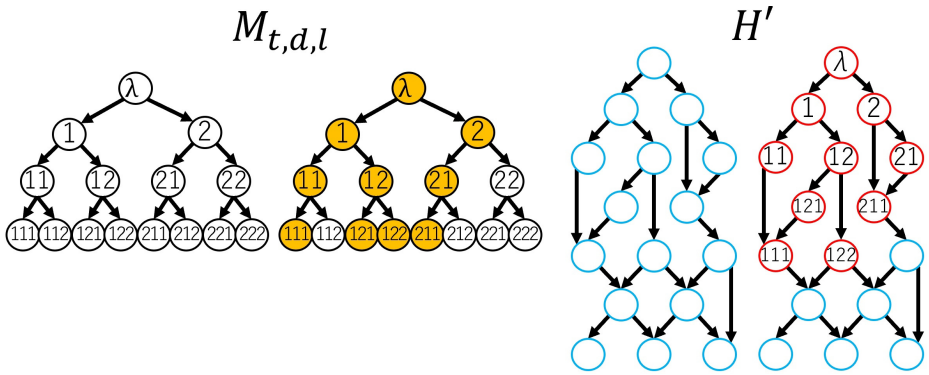

Fig. 2. Illustration of the operation of GrowTokenTree. The orange tokens in $M_{t,d,l}$ indicate *tokened* tokens. Starting from each left state and ending with each right state. GrowTokenTree places tokens of $M_{t,d,l}$ onto vertices of H' while preserving the tree structure

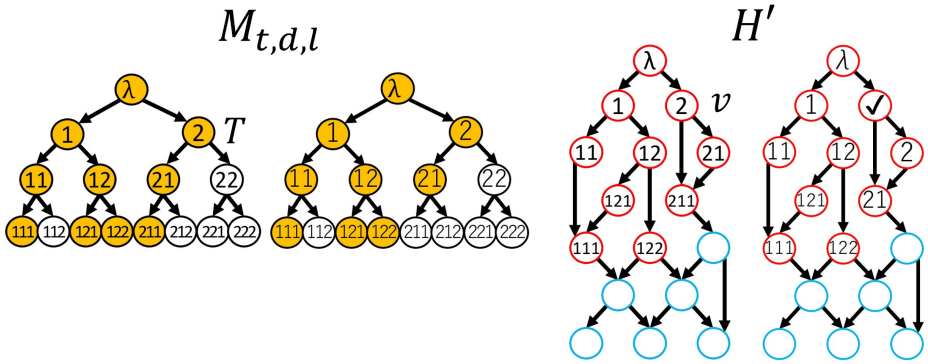

Fig. 3. Illustration of token replacement in FindEmbedding. A vertex v satisfying the condition in Line 5 of the pseudocode (Algorithm 2) and the token T placed at v are selected ($M_{t,d,l}$ left, H' left), and T is removed from v. Subsequently, T's all *tokened* descendants $T \cdot b \cdot S$ ($1 \leq b \leq d$ and S is a string of arbitrary length) are replaced with $T \cdot S$ on H'.

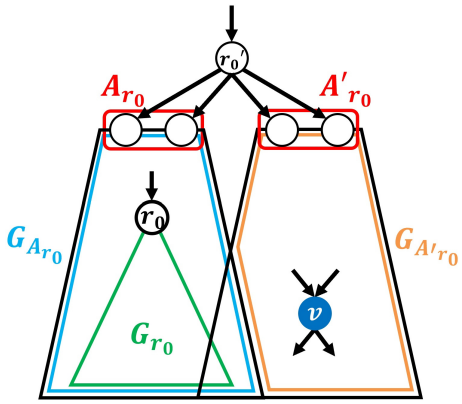

Fig. 4. Illustration of the proof of **Lemma 18**. The recent common descendant of r_0 and v is denoted as r_0'. The proof demonstrates that a contradiction arises if the blue vertex v is not included in G_{r_0}, which consists of r_0 and its descendants.

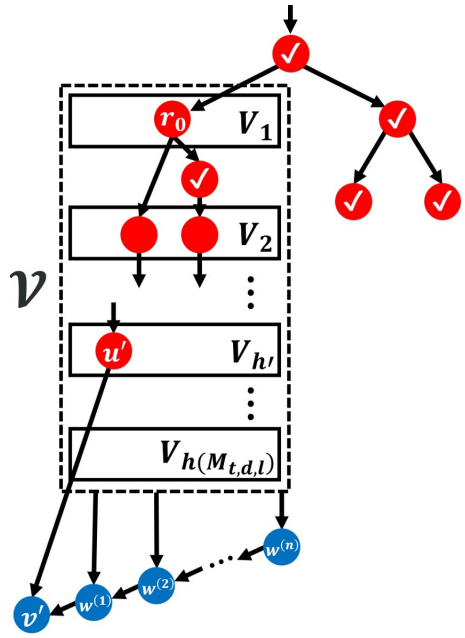

Fig. 5. Illustration of the proof of **Lemma 8**. The blue vertices represent blue, the red check-marked vertices indicate token-removed vertices, and the other red vertices indicate vertices with tokens placed on them. The proof demonstrates that a contradiction arises if there are no vertices in $V_{h(M_{t,d,l})}$.

C.2 Lemma 4

Proof. We prove this by mathematical induction on h.

For $h = 2$, the DAG-pathwidth of $T_{2,d}$ is clearly 1, so the lemma holds.

Next, assuming that the lemma holds for some $h > 1$, there exists a DAG-path-decomposition X_h of $T_{h,d}$ with DAG-pathwidth $h-1$. Here, note that $T_{h+1,d}$ is a graph obtained by connecting a single root r to the roots of d copies of $T_{h,d}$. We can construct a DAG-path-decomposition of $T_{h+1,d}$ with width h as follows. First, for the d DAG-path-decompositions X_h, connect the starting and ending bags of each decomposition sequentially to form a single long sequence of bags. Next, add r to each bag in this sequence. Finally, prepend a bag containing only r at the beginning of the sequence. The resulting sequence satisfies the three rules of a DAG-path-decomposition, making it a valid DAG-path-decomposition of $T_{h+1,d}$ with width h.

Furthermore, no DAG-path-decomposition of $T_{h+1,d}$ with width less than h exists. The reasons are shown below. $T_{h+1,d}$ contains $T_{h,d}$ as a substructure, so the DAG-pathwidth of $T_{h+1,d}$ cannot be smaller than $h-1$. Now, suppose there exists a DAG-path-decomposition of $T_{h+1,d}$ with width $h-1$. Since the d copies of $T_{h,d}$ are mutually unreachable, each can independently form a DAG-path-decomposition, and the width does not need to exceed that of a parallel decomposition of the d copies of $T_{h,d}$. Let $T'_{h,d}$ be the first decomposed tree among d $T_{h,d}$. By (DPD2), any DAG-path-decomposition of $T_{h+1,d}$ must include the root r in its first bag. However, the bag in the DAG-path-decomposition of $T'_{h,d}$ that has the maximum width of $h-1$ (denoted as X') does not contain r. By (DPD3), r does not reappear in any later bag, implying that the remaining $d-1$ copies of $T_{h,d}$ do not connect to r. This requires $d=1$, which contradicts the assumption that $d>1$.

Even if all vertices connected to r were included in a bag before X', some vertices from outside $T'_{h,d}$ must necessarily be included in X', which concludes the width greater than $h-1$. It leads to a contradiction.

Thus, no DAG-path-decomposition of $T_{h+1,d}$ with width $h-1$ exists, and the optimal DAG-path-decomposition of $T_{h+1,d}$ has width h. Therefore, the lemma holds for $h+1$, and by induction, it holds for any $h, d \geq 2$.

C.3 Lemma 5

Proof. First, we show that G' is always connected. Token placement and replacement occur only in Line 2 of GrowTokenTree or Lines 6 and 8 of FindEmbedding. The former explicitly ensures token placement maintains connectivity, and the latter replaces all tokens in the directed tree rooted at $T \cdot b$ with corresponding tokens in the directed tree rooted at T immediately after removing T. This guarantees that connectivity is preserved after processing Lines 6 and 8. Thus, G' remains connected at all times.

Next, we show that an embedding from G' to H' exists. Since GrowTokenTree places a token T on $u \in V[H']$ and its child v receives token $T \cdot b$, the embedding condition is clearly satisfied. It suffices to verify that the embedding condition holds throughout Lines 6 to 9 of FindEmbedding. Assuming that the embedding condition is satisfied at Line 5, if a token T satisfying Line 5 exists and has exactly one *tokened* child $T \cdot b$, Line 8 is executed. In Line 8, S represents an arbitrary-length string consisting of characters from 1 to d, and the operation replaces all tokens in the directed tree rooted at $T \cdot b$ with corresponding tokens in the directed tree rooted at T while maintaining tree-structural integrity. Since the replacement occurs from root-proximal tokens to distal ones, the target tokens are always *untokened*. As G' remains connected and T has only $T \cdot b$ as its *tokened* child, removing T and replacing all tokens in the directed tree rooted at $T \cdot b$ with the corresponding tokens in the directed tree rooted at T preserves the embedding condition. If T has no *tokened* children, only Line 6 is executed, and Line 8 is skipped, trivially maintaining the embedding condition. Thus, the embedding condition remains intact throughout the sequence of operations, proving the existence of an embedding from G' to H'.

C.4 Lemma 6

Proof. Suppose that Line 20 of FindEmbedding is executed and all vertices of H' have turned red at step $i = s$. The sequence of vertex sets output by the algorithm, $X_{H'} = (X_1, X_2, \ldots, X_s)$, constitutes a DAG-path-decomposition of H'. Since every vertex $v \in H'$ is red, it must be contained in at least one vertex set X_i, satisfying (DPD1). Moreover, each vertex v changes color from blue to red exactly once, and tokens are removed from red vertices without being placed again, ensuring that the vertex sets X_i form a connected path, satisfying (DPD3).

For any edge $(u, v) \in E[H']$, suppose that v changes from blue to red at $i = i_v$ ($i_v \leq s$). By the condition in Line 5 of FindEmbedding, a token placed on u is not removed before step i_v. Additionally, by the condition in Line 1 of GrowTokenTree, all predecessor vertices of v have tokens, implying that u must also be included in X_i. Since v is not in X_{i_v-1}, we conclude that $u, v \in X_{i_v}$ and $v \notin X_{i_v-1}$, satisfying (DPD2). Therefore, $X_{H'}$ is a DAG-path-decomposition of H'.

Noting that $\lceil \log_d l \rceil < \log_d l + 1$, the width of $X_{H'}$ is at most $|V[M_{t,d,l}]| = d^{\lceil \log_d l \rceil + t + 2} - 1 < ld^{t+3} - 1$. Since $X_H = (X_1 \cap V[H], X_2 \cap V[H], \ldots, X_s \cap V[H])$ satisfies the three rules of the DAG-path-decomposition for H, it follows that X_H is a DAG-path-decomposition of H with width at most $ld^{t+3} - 1$. Thus, we obtain a DAG-path-decomposition of H with width at most $ld^{t+3} - 1$.

C.5 Lemma 7

Proof. Suppose that Line 17 of FindEmbedding is executed at step $i = s$ (i.e., the condition $|X_s| = |V[M_{t,d,l}]|$ holds), and let A' be the subgraph of H' induced by the vertex set $X_1 \cup X_2 \cup \cdots \cup X_s$. Then, the sequence of vertex sets output by the algorithm, $X_{A'} = (X_1, X_2, \ldots, X_s)$, forms a DAG-path-decomposition of A'. By definition of A', (DPD1) is clearly satisfied. Additionally,(DPD2) and (DPD3) are satisfied by the argument similar to **Lemma 6**. Thus, $X_{A'}$ is a DAG-path-decomposition of A'.

Let A be the subgraph of H induced by $V[A'] \cap V[H]$ and B be the subgraph of H induced by $V[H] \backslash V[A]$. Since A and B are disjoint and (DPD2) ensures that only edges from A to B exist. Defining $X_A = (X_1 \cap V[H], X_2 \cap V[H], \ldots, X_s \cap V[H])$, this decomposition satisfies the three rules of the DAG-path-decomposition of A. Therefore, X_A forms a DAG-path-decomposition of A with width at most $ld^{t+3} - 1$ by the argument similar to **Lemma 6**.

Next, let A'_M be the subgraph of H' induced by the end bag X_s in $X_{A'}$. Since at step $i = s$, all tokens in $M_{t,d,l}$ have been used in the embedding, **Lemma 5** implies that A'_M represents an embedding of $M_{t,d,l}$ into H'. Since $M_{t,d,l}$ has height $\lceil \log_d l \rceil + t + 2$, and A' contains A'_M, the subgraph induced by $V[A'] \backslash V[A]$ forms a complete directed d-ary tree of height at most $\lceil \log_d l \rceil$. Thus, A contains a complete directed d-ary tree T_A of height at least $(\lceil \log_d l \rceil + t + 2) - (\lceil \log_d l \rceil) = t + 2$, meaning that an embedding from T_A to A exists.

By **Lemma 4**, the DAG-pathwidth of T_A is $t+1$, implying that the DAG-pathwidth of A is at least $t+1$. Since H contains A as a subgraph, the DAG-pathwidth of H must be greater than t. Consequently, the DAG-pathwidth of A is greater than t but at most $ld^{t+3}-1$.

C.6 Lemma 8

Before proving Lemma 8, we show the following.

Lemma 18. *At any time $i=k$, let r_0 be the vertex of H' on which the root token λ is placed. Then, all blue vertices in H' must have r_0 as an ancestor.*

Proof. We prove this by contradiction. Suppose at time $i=k$, there exists a blue vertex $v \in V[H']$ that does not have r_0 as an ancestor. Since H' has a single root, there exists a common ancestor of r_0 and v. Let r'_0 be such a most recent common ancestor that does not have another common ancestor of r_0 and v as its descendant. If multiple such vertices exist, choose one of them as r'_0. Notice that $v \neq r'_0$ because at time $i=k$, the root token λ is placed at r_0, implying that its ancestor r'_0 must have had its token removed. Assuming $r'_0 = v$ contradicts the fact that v is blue.

Define A_{r_0} as the set of children of r'_0 whose descendants include r_0, and let A'_{r_0} be the set of other children of r'_0. Let $G_{A_{r_0}}$ be the subgraph induced by the descendants of the vertices in A_{r_0} and $G_{A'_{r_0}}$ be the subgraph induced by the descendants of the vertices in A'_{r_0} that are not in $G_{A_{r_0}}$. Additionally, let G_{r_0} be the subgraph induced by the descendants of r_0. By assumption, $v \notin V[G_{r_0}]$, meaning $v \in V[G_{A_{r_0}}] \setminus V[G_{r_0}]$ or $v \in V[G_{A'_{r_0}}]$ (see Figure 4 in Appendix C.1).

Consider the first moment $i=l$ ($l \leq k$) when a token λ is placed on a vertex in $G_{A_{r_0}}$. At this moment, all vertices in $G_{A'_{r_0}}$ must have had their tokens removed. We justify this below. Any vertex in H' is either (a) blue, (b) red with a token, or (c) red and a token is already removed. Since v is blue at $i=l$, Line 5 of FindEmbedding requires that its parent must be (a) or (b). If the parent is (a), we continue checking its parent, which must also be (a) or (b). If all ancestors of v are (a), at least one blue vertex exists in A'_{r_0} at $i=l$, contradicting the conditions of Line 5. If at least one ancestor is (b), then, when an token $T_{r'_0}$ placed on r'_0 has been removed, $T_{r'_0}$ must have had at least two *tokened* children, one in $G_{A'_{r_0}}$ and another elsewhere. Since $T_{r'_0}$ has been removed from r'_0 at $i=l$, this contradicts Line 5 of FindEmbedding which requires that a removed token has at most one *tokened* child. Therefore, $v \notin V[G_{A'_{r_0}}]$ must hold. Given that $v \notin V[G_{A_{r_0}}]$ by assumption, it must hold that $v \in V[G_{A_{r_0}}] \setminus V[G_{r_0}]$. However, in this case, a common ancestor of v and r_0 must be included in A_{r_0}, which contradicts the fact that there exists no common ancestor of v and r_0 among the descendants of r'_0. Thus, the claim of **Lemma 8** is proven.

Now, we prove Lemma 8.

Proof. Suppose that Line 11 of FindEmbedding is executed at $i=k$. First, we prove that at $i=k$, there exists at least one root-to-leaf path $P = (\lambda \cdot m_1 \cdot$

$m_2 \cdot \ldots m_{\lceil \log_d l \rceil + t + 1}$) in the *tokened* token set on $M_{t,d,l}$. Therefore, we show a contradiction by assuming that such a path does not exist. Let P' be the longest path in the *tokened* token set rooted at λ, and let the height of $M_{t,d,l}$ be $h(M_{t,d,l}) = \lceil \log_d l \rceil + t + 2$. By assumption, $|P'| \leq h(M_{t,d,l}) - 1$. Let T' be the terminal token of P', placed at vertex $u' \in V[H']$. From Line 1 of GrowTokenTree, at least one of the following must hold:

1. There exists a vertex $v' \in \mathsf{suc}(u')$ such that one of its parents $w^{(1)}$ is blue.
2. T' has no *untokened* children.

Note that we do not need to consider the case where u' has no children, because in this case, we can remove the token placed on u', preventing the state described in (3) from occurring. If condition 2 holds, then T' is either a leaf or all of T''s children are *tokened*. However, neither of these conditions can hold, because by assumption, the endpoint T' of P' is not a leaf, and if T' has a *tokened* child, it would contradict the fact that P' is the longest path. Therefore, condition 1 must hold. Let r_0 be the vertex where the root token λ is placed. Let G_{r_0} be the subgraph induced by the descendants of r_0 in H', and let V_h be the set of vertices in G_{r_0} where tokens with height h ($1 \leq h \leq h(M_{t,d,l})$) on the $M_{t,d,l}$ graph are placed . Also, let $\mathcal{V} = \bigcup_{1 \leq h \leq h(M_{t,d,l})} V_h$. By assumption, $u' \in V_{h'}$ ($1 \leq h' \leq h(M_{t,d,l}) - 1$). Furthermore, by **Lemma 18**, both the blue vertices $v', w^{(1)}$ are reachable from r_0, so they are contained in G_{r_0}. Additionally, considering the placement of tokens in GrowTokenTree, we observe that the children of a blue vertex cannot be red, meaning that $v', w^{(1)}$ and their descendants are not in $\mathcal{V}$. Now, for $w^{(1)}$, one of the following must always hold:

1. All of $w^{(1)}$'s parents are in $\mathcal{V}$.
2. There exists at least one blue parent of $w^{(1)}$ that is not in $\mathcal{V}$.

Note that there are no red parents of $w^{(1)}$ that are not in $\mathcal{V}$. This is because, by a similar argument to **Lemma 18**, each of $w^{(1)}$'s parents must either be (a) blue or (b) red with a token. Thus, if any parent satisfies (a), condition 2 holds; otherwise, condition 1 holds. If condition 1 holds, a token should be placed on $w^{(1)}$. This is because each of $w^{(1)}$'s parents must be included in one of $V_1, V_2, \ldots, V_{h'}$, and each parent has exactly d children, the maximum outdegree in H'. Therefore, each parent must have at least one *untokened* child, and this token can be placed on $w^{(1)}$. Thus, the condition 1 contradicts the assumption that $w^{(1)}$ is blue. Therefore, condition 2 must hold.

If condition 2 holds for $w^{(1)}$, let $w^{(2)}$ be one of blue parents of $w^{(1)}$ satisfying condition 2. By similar reasoning, a blue parent $w^{(3)}$ of $w^{(2)}$ satisfying condition 2 must exist. Repeating this argument, since the number of vertices in H' is finite, there must eventually be a blue vertex $w^{(n)}$ that satisfies condition 1. Therefore, a contradiction arises, and there must be at least one path from the root to a leaf in the *tokened* token set on $M_{t,d,l}$ (see Figure 5 in Appendix C.1).

Next, we will show that the DAG-pathwidth of H is greater than t using the path P. Since there are no tokens satisfying the condition of Line 5 in

FindEmbedding in (3), for each token $m_i \in P$ (where λ is denoted by m_0), the vertex $v_i \in V[H']$ where the token m_i is placed must satisfy at least one of the following:

1. There exists a vertex $w_i \in \mathsf{suc}(v_i)$ such that w_i and all its descendants are not in $\mathcal{V}$, and w_i is blue.
2. The token placed on v_i has two or more *tokened* children.

Next, for each v_i, we will show that there exists a vertex u_i such that in any DAG-path-decomposition of H', each u_i (for $0 \leq i \leq h(M_{t,d,l}) - 1$) must be included in some bag. First, consider the case where v_i satisfies condition 1. In this case, v_i cannot be forgotten in the DAG-path-decomposition of H' until $m_{h(M_{t,d,l})-1}$ is placed at $v_{h(M_{t,d,l})-1}$. This is because w_i and all of its descendants are not included in $\mathcal{V}$, and by taking note of the operation of GrowTokenTree, a token is placed on w_i only after the token $m_{h(M_{t,d,l})-1}$ is placed on $v_{h(M_{t,d,l})-1}$. Since v_i, which has w_i as a predecessor, cannot remove m_i before that, v_i cannot be forgotten before the introduction of $v_{h(M_{t,d,l})-1}$ in the DAG-path-decomposition of H'. At this point, we set $u_i = v_i$.

Next, consider the case where v_i does not satisfy condition 1 but satisfies condition 2. m_i has at least one *tokened* child m'_{i+1} other than m_{i+1}. Since the vertex on which m'_{i+1} is placed also satisfies at least one of conditions 1 or 2, by repeating the above argument, and noting that any *tokened* leaf of $M_{t,d,l}$ must satisfy condition 1, there exists at least one descendant vertex of v_i which is placed a descendant token of m'_{i+1} and satisfies condition 1. Let such a vertex be u_i.

For the same reason as above, u_i cannot be forgotten in the DAG-path-decomposition of H' until $m_{h(M_{t,d,l})-1}$ is placed on $v_{h(M_{t,d,l})-1}$. Thus, for each v_i, we can construct a corresponding u_i.

In any DAG-path-decomposition X' of H', since each u_i is included in the bag where $v_{h(M_{t,d,l})-1}$ is introduced, the width of X' is at least $h(M_{t,d,l}) - 1$. Let P_H be the sequence of tokens placed at vertices in H from the sequence P. Since $|P \backslash P_H| \leq \lceil \log_d l \rceil$, we have $|P_H| = |P| - |P \backslash P_H| \geq (\lceil \log_d l \rceil + t + 2) - \lceil \log_d l \rceil = t + 2$. Thus, by the same reasoning, the width of any DAG-path-decomposition of H is greater than t.

C.7 Theorem 7

Proof. By inputting the DAG H into FindEmbedding, the algorithm will always terminate in one of the cases (1), (2), or (3). If it terminates in case (1), by **Lemma 6**, **Theorem 7**(a) holds. If it terminates in case (2), by **Lemma 7**, **Theorem 7**(b) holds. If it terminates in case (3), by **Lemma 8** and setting $A = H$, **Theorem 7**(b) holds. Therefore, **Theorem 7** is proven.

C.8 Corollary 1

Proof. By **Theorem 7**, FindEmbedding either outputs a witness that the DAG-pathwidth of H is greater than t, or outputs a DAG-path-decomposition of

width $O(ld^t)$. We now show that FindEmbedding terminates in polynomial time. In GrowTokenTree, the condition check of Line 1 takes $O(dn)$ time, and the while loop is repeated at most $|V[M_{t,d,l}]| = O(d^t)$ times. Also, due to the removal of T in Line 6 of FindEmbedding, the vertices where T was placed remain red, so this operation is performed $O(n)$ times. Therefore, the while loop in Line 4 is also repeated $O(n)$ times. Furthermore, the while loop in Line 10 clearly repeats at most d^2 times. Other steps are processed in $O(1)$ time. Thus, the algorithm terminates in $O(n^2)$ time if d and l are bounded by constants.

D Eligibility for the student award

The first author was a student until March 2025, and this research was conducted during the first author's student tenure. The first author performed the majority of this research.