# Designing various algorithms based on DAG-pathwidth

Jun Kawahara[1][0000−1111−2222−3333] and Shinya Izu[1][1111−2222−3333−4444]

[1] Princeton University, Princeton NJ 08544, USA
[2] Springer Heidelberg, Tiergartenstr. 17, 69121 Heidelberg, Germany
lncs@springer.com
http://www.springer.com/gp/computer-science/lncs
[3] ABC Institute, Rupert-Karls-University Heidelberg, Heidelberg, Germany
{abc,lncs}@uni-heidelberg.de

**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.
It is known that computing the DAG-pathwidth is NP-hard, and, to our knowledge, no algorithm has been known for finding small DAG-pathwidth. In this paper, we 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 at most $O(ld^t)$ or provides an evidence that the DAG-pathwidth of $H$ is greater than $t$.

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

## 1 Introduction

Tree decomposition is one of the approach to efficiently solve NP-hard problems. This operation is dividing the vertices of a undirected graph into subsets, treating each subset as a node, and transforming the graph into a tree-like structure. This enables algorithms designed for trees to general graphs, which allows efficient solutions even for NP-hard problems. Each node in a tree decomposition

is called bag, and the *width* of the tree decomposition is the maximum number of vertices in a single bag. A *treewidth* is the minimum width across all possible tree decompositions of a graph. A smaller treewidth indicates that the graph's structure is closer to a tree. When the treewidth is bounded by a constant, it is sometimes possible to solve NP-hard problems in polynomial time with respect to the number of vertices.

Similarly, *path decomposition* transforms a graph into a path-like structure by partitioning its vertices into subsets. Like treewidth, *pathwidth* is the minimum width across all possible path decompositions. A path decomposition also enables efficient algorithms for NP-hard problems when the pathwidth is bounded by a constant.

The pathwidth of an undirected graph was first proposed by Robertson et al. [21]. They also introduced the concept of treewidth [22]. 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 fixed-parameter tractable (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 [18] 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.

For directed graphs, several width parameters have been proposed. *Directed pathwidth* was introduced by Reed in 1997 [20], and *directed treewidth* by Johnson et al. in 2001 [15]. In 2012, Berwanger et al. [6] proposed width parameters specifically for directed acyclic graphs (DAGs). Directed pathwidth measures how close a directed graph is to a DAG. However, for DAGs, this parameter is always at most one, making it challenging to construct parameterized algorithms for NP-hard problems on DAGs. In response, Kasahara et al. [16] proposed *DAG-pathwidth* in 2023, which measures how close a directed graph is to a directed path. DAG-pathwidth adds a rule requiring that for any edge, its endpoints must appear in the same bag of the decomposition. This ensures that strongly connected components are contained in a single bag, allowing non-trivial widths for DAGs and facilitating effective FPT algorithm design.

In [16], they constructed a parameterized algorithm for the discord $k$-independent set problem using DAG-pathwidth and proved that computing the DAG-pathwidth is NP-hard. On the other hand, to the best of our knowledge, it has not been developed so far to construct parameterized algorithms for the other NP-hard problems, as well as to compute a DAG-path-decomposition of small width.

Our contributions are as follows:

1. Constructing parameterized algorithms using DAG-pathwidth for various NP-hard problems on DAGs.
2. Proposing approximation algorithms for DAG-path-decomposition and parameterized algorithms to construct decompositions with small width.
3. Extending DAG-pathwidth to DAG treewidth, a parameter measuring the similarity of a directed graph to a directed tree.

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 a $O(\log^{3/2} n)$-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.

## 2 Preliminaries

In this Section, we introduce definitions and notations for graphs and directed graphs.

### 2.1 DAG

This study deals with DAGs as input graphs. A DAG is defined as follows.

**Definition 1.** *A DAG (Directed Acyclic Graph) is a directed graph with no cycles.*

Next, we define predecessors and successors for each vertex of a DAG.

**Definition 2.** *For a DAG $G = (V, E)$ and a vertex $v \in V$, the predecessors* $\mathsf{pred}(v)$ *and successors* $\mathsf{suc}(v)$ *of $v$ are defined as follows:*

$$\mathsf{pred}(v) = \left\{ u \in V \,\middle|\, (u, v) \in E \right\},$$
$$\mathsf{suc}(v) = \left\{ w \in V \,\middle|\, (v, w) \in E \right\}.$$

The roots and leaves of a DAG are defined as follows.

**Definition 3.** *For a DAG $G = (V, E)$, a vertex $v \in V$ is called a root if its in-degree is 0. Similarly, $u \in V$ is called a leaf if its out-degree is 0.*

In general, a DAG can have multiple roots and leaves. We further define a specific type of DAG, the directed tree, as follows:

**Definition 4.** *A DAG $G = (V, E)$ is called a directed tree if it satisfies all the following conditions:*

1. *$G$ has exactly one root $r$.*
2. *The underlying undirected graph of $G$, ignoring the direction of edges, is a tree.*
3. *For any vertex $v \in V$, there exists exactly one directed path from $r$ to $v$.*

For a directed tree, the graph obtained by reversing the direction of all edges is called an anti-directed tree. In an anti-directed tree, a vertex with out-degree 0 is referred to as the root, and a vertex with in-degree 0 is referred to as a leaf.

### 2.2 Various Path Decompositions and Pathwidth

To facilitate understanding of DAG-pathwidth, we first introduce the definitions of (undirected) path decomposition and pathwidth, followed by directed path decomposition and directed pathwidth.

**Definition 5 (Path Decomposition).** *[[21]] Let $G = (V, E)$ be an undirected graph. A path decomposition of $G$ is a sequence $X = (X_1, X_2, \ldots, X_s)$ of subsets $X_i \subseteq V$ $(i = 1, 2, \ldots, s)$ that satisfies the following three conditions:*

1. *$X_1 \cup X_2 \cup \cdots \cup X_s = V$.*
2. *For any edge $(u, v) \in E$, there exists an $i$ $(\geq 1)$ such that $u, v \in X_i$.*
3. *For any integers $i, j, k$ $(1 \leq i \leq j \leq k \leq s)$, $X_i \cap X_k \subseteq X_j$, that is, for any vertex $v \in V$, $v$ induces exactly one non-empty path in $X$.*

**Definition 6 (Pathwidth).** *For a path decomposition $X = (X_1, X_2, \ldots, X_s)$ of an undirected graph $G$, the width of $X$ is defined as $\max_i\{|X_i| - 1\}$. The pathwidth of $G$ is the minimum width over all possible path decompositions of $G$.*

The pathwidth is a parameter that represents how closely the graph is to a path. The smaller the pathwidth, the closer the graph structure is to a path. Computing the pathwidth of a general undirected graph is NP-hard [[2]].

Next, we define directed path decomposition and directed pathwidth.

**Definition 7 (Directed Path Decomposition).** *[[20]] Let $G = (V, E)$ be a directed graph. A directed path decomposition of $G$ is a sequence $X = (X_1, X_2, \ldots, X_s)$ of subsets $X_i \subseteq V$ $(i = 1, 2, \ldots, s)$ that satisfies the following three conditions:*

1. *$X_1 \cup X_2 \cup \cdots \cup X_s = V$.*

2. *For any directed edge* $(u, v) \in E$*, there exist* $i, j$ $(i \leq j)$ *such that* $u \in X_i$*,* $v \in X_j$*.*
3. *For any* $i, j, k$ $(1 \leq i \leq j \leq k \leq s)$*,* $X_i \cap X_k \subseteq X_j$*, that is, for any vertex* $v \in V$*,* $v$ *induces exactly one non-empty path in* $X$*.*

**Definition 8 (Directed Pathwidth).** *For a directed path decomposition* $X = (X_1, X_2, \ldots, X_s)$ *of a directed graph* $G$*, the width of* $X$ *is defined as* $\max_i\{|X_i| - 1\}$*. The directed pathwidth of* $G$ *is the minimum width over all possible directed path decompositions of* $G$*.*

Directed pathwidth represents how closely the structure of a graph is to a DAG. A smaller directed pathwidth indicates a structure closer to a DAG. Computing the directed pathwidth of a general directed graph is also NP-hard.

We define DAG-path-decomposition and DAG-pathwidth as follows:

**Definition 9 (DAG-path-decomposition [16]).** *Let* $G = (V, E)$ *be a directed graph. A DAG-path-decomposition of* $G$ *is a sequence* $X = (X_1, X_2, \ldots, X_s)$ *of subsets* $X_i \subseteq V$ $(i = 1, 2, \ldots, s)$ *that satisfies the following three conditions:*

1. $X_1 \cup X_2 \cup \cdots \cup X_s = V$*.*
2. *For every directed edge* $(u, v) \in E$*, one of the following holds:*
   - $u, v \in X_1$*.*
   - *There exists* $i$ $(i \geq 2)$ *such that* $u, v \in X_i$ *and* $v \notin X_{i-1}$ *.*
3. *For any* $i, j, k$ $(1 \leq i \leq j \leq k \leq s)$*,* $X_i \cap X_k \subseteq X_j$*. That is, for any vertex* $v \in V$*,* $v$ *induces exactly one non-empty directed path in* $X$*.*

Note that in [16], 2 is defined as follows. In this study, we adopt the above definition for the sake of algorithmic simplicity:

1. There exists $i$ $(i \geq 2)$ such that $u, v \in X_i$ and $u \notin X_{i-1}$ .

Each subset $X_i$ is called a *bag*.

**Definition 10 (DAG-pathwidth).** *Given a DAG-path-decomposition* $X = (X_1, X_2, \ldots, X_s)$ *of a directed graph* $G$*, the width of* $X$ *is defined as* $\max_i\{|X_i| - 1\}$*. The DAG-pathwidth of* $G$ *is the minimum width among all possible DAG-path-decompositions of* $G$*.*

The DAG-pathwidth is a parameter that indicates how closely the structure of a directed graph resembles a directed path. A smaller DAG-pathwidth implies a closer resemblance to a directed path. From Rule 3 of the DAG-path-decomposition, any vertex $v$ is contained in a connected sequence of bags. Additionally, Rules 2 and 3 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.

Computing the DAG-pathwidth for general directed graphs is NP-hard [16]. However, it can be shown that graphs with DAG-pathwidth 1 are exactly the *caterpillar-shaped* directed graphs, defined as follows:

**Definition 11 (Caterpillar-Shaped Graph).** *A directed graph $G$ is said to be caterpillar-shaped if it is a directed tree such that removing vertices with in-degree 1 and out-degree 0 leaves a single directed path.*

**Lemma 1.** *For a connected directed graph $G$ with $n$ vertices ($n > 2$), the DAG-pathwidth of $G$ is 1 if and only if $G$ is caterpillar-shaped.*

The proof of **Lemma 1** is provided in the appendix.

To facilitate dynamic programming, we define a *nice DAG-path-decomposition* as follows:

**Definition 12 (Nice DAG-path-decomposition [16]).** *A DAG-path-decomposition $X = (X_1, X_2, \ldots, X_s)$ of a directed graph $G = (V, E)$ is called a nice DAG-path-decomposition if it 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 $X_{i+1} = X_i \setminus \{v\}$.*

For a DAG $G$, each strongly connected component $S$ consists of a single vertex. Thus, the *introduce* operation can be redefined as follows:

1. (introduce) There exists a vertex $v \in V$ such that $\{v\} \cap X_i = \emptyset$ and $X_{i+1} = X_i \cup \{v\}$.

The nice DAG-path-decomposition simplifies the design of dynamic programming algorithms, as each bag involves either introducing or forgetting a single vertex. It is shown in [4] 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 satisfies the following:

**Proposition 1.** *Let $X = (X_1, X_2, \ldots, X_s)$ be any DAG-path-decomposition of a directed graph $G$. If $X_i \neq X_{i+1}$ for all $i$, then $s \leq 2|V[G]| + 1$.*

In the following, we say that the bag $X_i$ is introduce when the bag introduces a strongly connected component $S$. Similarly, we say that the bag $X_i$ is forget when the bag forgets a vertex $v$. For any vertex, there exists exactly one introduce bag and one forget bag due to Rules 1 and 3.

### 2.3 Black Pebbling Game

To compare with DAG-path-decomposition, we define the Black Pebbling game as follows:

**Definition 13 (Black Pebbling Game).** *[[17]] The Black Pebbling game is a game in which, 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:*

**Rule 1** $P_0 = \emptyset$

**Rule 2** *pebble: If no pebble is placed on $v \in V$ and all predecessors of $v$ have pebbles, a pebble may be placed on $v$. That is, if $v \notin P_{i-1}$ and for every $(u, v) \in E$, $u \in P_{i-1}$, then $P_i = P_{i-1} \cup \{v\}$.*

**Rule 3** *unpebble: A pebble placed on $v$ can be removed at any time. That is, if $v \in P_{i-1}$, then $P_i = P_{i-1} \setminus \{v\}$.*

**Rule 4** *Each vertex must have a pebble placed on it at least once.*

Pebble (Rule 2) represents the operation of placing a pebble on a vertex, and an unpebble (Rule 3) represents the operation of removing a pebble from a vertex. Note that roots of the graph, which have no predecessors, can be pebbled at any time. Furthermore, the pebbling number is defined as follows:

**Definition 14 (Pebbling Number).** *For a strategy $P = (P_0, P_1, \ldots, P_\tau)$ of a DAG $G$, the* space *and* time *are defined as follows:*

1. $\mathsf{space}(P) = \max_i\{|P_i|\}$
2. $\mathsf{time}(P) = \tau$

*The pebbling number of $G$ ($\mathsf{Peb}(G)$) is the minimum space over all strategies of $G$.*

For general DAGs, the problem of computing the pebbling number is PSPACE-complete [[12]]. The Black Pebbling game is used in blockchain technology known as Proof of Space [[8]]. Proof of Space is a method to prove the amount of free disk space held, where the input DAG corresponds to the free disk space. Moreover, the pebbling number corresponds to the amount of memory used simultaneously. A larger pebbling number indicates greater memory or data usage, making the proof more challenging and thus indicating higher security of the proof.

Additionally, the one-shot Black Pebbling (one-shot BP) is defined by adding the following rule to the Black Pebbling game:

**Rule 5** Each vertex of the DAG $G$ is pebbled only once.

The pebbling number for one-shot BP can also be defined similarly. For general DAGs, the problem of computing the pebbling number for one-shot BP is NP-hard [[23]]. In Section 4.3, we demonstrate that one-shot BP is equivalent to the problem of constructing a nice DAG-path-decomposition on DAGs.

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

soon.

## 4 $O(\log^{3/2} n)$-approximation algorithm for DAG-pathwidth

soon.

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

In this chapter, 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 at most $O(ld^t)$ or provides evidence that the DAG-pathwidth of $H$ is greater than $t$. This algorithm is constructed based on the path decomposition algorithm for undirected graphs [7].

**Proposition 2.** *[7] Given an undirected graph $H$ with $n$ vertices and a non-negative integer $t$, there exists an $O(n)$ time algorithm that either provides evidence that the pathwidth of $H$ is greater than $t$ or returns a path decomposition with width at most $O(2^t)$.*

Following this algorithm [7], we construct an algorithm that, for a DAG $H$ with maximum outdegree $d$ and number of roots $l$, either provides evidence that the DAG-pathwidth of $H$ is greater than $t$ or returns a DAG-path-decomposition with width at most $O(ld^t)$. This algorithm utilizes the following homeomorphic embedding.

**Definition 15 (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:*

1. *$f$ is an injective function.*
2. *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.

**Theorem 1.** *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 $t$.*

To prove **Theorem 1**, we first define the notion of a complete directed tree.

**Definition 16.** *A directed tree $T = (V, E)$ is called a complete $d$-ary directed tree if it satisfies the following conditions:*

1. *Every non-leaf vertex has exactly $d$ children.*

*2. All root-to-leaf paths have the same length.*

If $T$ is a complete $d$-ary directed tree, we define the height of $T$ as the length of any root-to-leaf path plus 1. If $T$ consists of only the root, then its height is defined as 1.

**Lemma 2.** *Let $T_{h,d}$ be a complete $d$-ary directed tree of height $h$ $(h, d > 1)$. Then, the DAG-pathwidth of $T_{h,d}$ is $h - 1$.*

*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 Rule 2 of DAG-path-decomposition, 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 Rule 3 of DAG-path-decomposition, $r$ 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 $h, d \geq 2$.

To prove **Theorem 1**, we construct the parameterized algorithm. Given an input DAG $H = (V, E)$, we modify it to have a single root by adding a complete $d$-ary directed tree of height $\lceil \log_d l \rceil$ and connecting its leaves to each root of $H$. Let $H'$ be the resulting DAG. We also define $M_{t,d,l}$ as a complete $d$-ary directed tree of height $\lceil \log_d l \rceil + t + 2$.

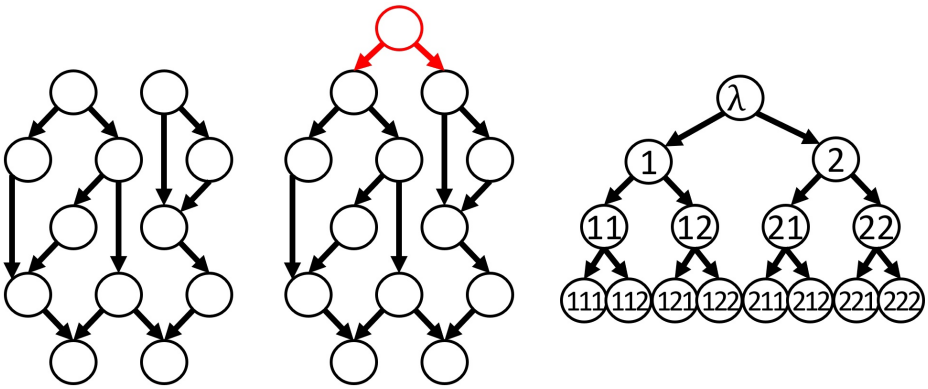


**Fig. 1.** given $H$ with $d = 2, l = 2$ (left) and $t = 1$, the construction of $H'$ (center) and $M_{t,d,l}$ (right). $H'$ is obtained by adding a complete directed $d$-ary tree (red part) connected to each root of $H$. Moreover, the labeling of $M_{t,d,l}$ is constructed recursively by appending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

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 define recursive token labeling as follows:

1. The root token is labeled with the empty string $\lambda$.
2. If a parent token has label $m = \lambda b_1 b_2 \dots b_{h-1}$, its children are labeled $m \cdot 1, m \cdot 2, \dots, m \cdot d$ from lef child to right child.

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, meaning that each vertex of $H'$ can have a token at most once.

GrowTokenTree and FindEmbedding are presented in Algorithms 1 and 2, respectively. GrowTokenTree greedily places tokens of $M_{t,d,l}$ onto vertices of $H'$ while preserving the tree structure. A token can only be placed on a vertex whose predecessors already have tokens. This process continues until no more tokens can be placed, at which point the algorithm outputs the set of vertices in $H'$ placed tokens.

**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 token $T \cdot b$ on $v$
**end while**
**return** $\{v \in V[H'] \mid v$ is placed a token$\}$

**Algorithm 2** FindEmbedding

place root token $\lambda$ on root of $H'$
$i \leftarrow 1$
$X_i \leftarrow$ call GrowTokenTree
**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**
    **return** $X_i$
  **end if**
  $i \leftarrow i + 1$
  $X_i \leftarrow$ call GrowTokenTree
**end while**

The algorithm FindEmbedding (Algorithm 2) attempts to output a sequence of vertex sets $(X_1, X_2, \dots)$ that form a DAG-path-decomposition. Initially, a token $\lambda$ is placed on the single root of $H'$. Then, setting $i = 1$, GrowTokenTree is executed, and the output is assigned to $X_1$. Subsequently, for each $i$, the following process is repeated. Assuming that $(X_1, X_2, \dots, X_i)$ has been constructed, if $X_i$ simultaneously uses all tokens of $M_{t,d,l}$, then it represents an embedding from $M_{t,d,l}$ to $H'$, which indicates the DAG-pathwidth of $H'$ is at least that of $M_{t,d,l}$. Moreover, if all vertices of $H'$ have turned red, each vertex of $H'$ has had exactly one token placed on it. In this case, the sequence $(X_1, X_2, \dots, X_i)$ forms a DAG-path-decomposition of $H'$, with proof provided later.

In any other case—namely, if not all tokens of $M_{t,d,l}$ are simultaneously used and at least one vertex in $H'$ remains blue—there may be potential for further execution of GrowTokenTree by modifying the token placement. Therefore, we

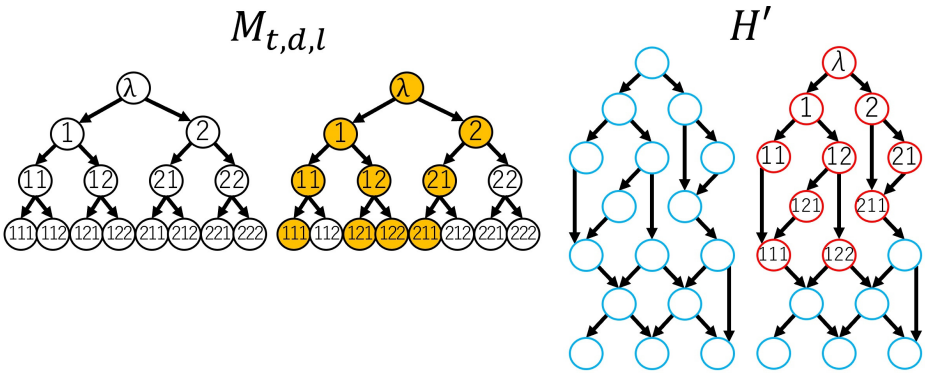


**Fig. 2.** Illustration of the operation of GrowTokenTree. The orange tokens in $M_{t,d,l}$ indicate *tokened* tokens. Starting from each left state and ending with each right state. GrowTokenTree places tokens of $M_{t,d,l}$ onto vertices of $H'$ while preserving the tree structure

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 $X_{i+1}$. If no token satisfies both conditions (a) and (b), the algorithm returns the last output of GrowTokenTree. This output represents the evidence that the DAG-pathwidth of $H'$ is at least that of $M_{t,d,l}$, as will be demonstrated below.

To prove **Theorem 1**, we first demonstrate the following lemma:

**Lemma 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.*

*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

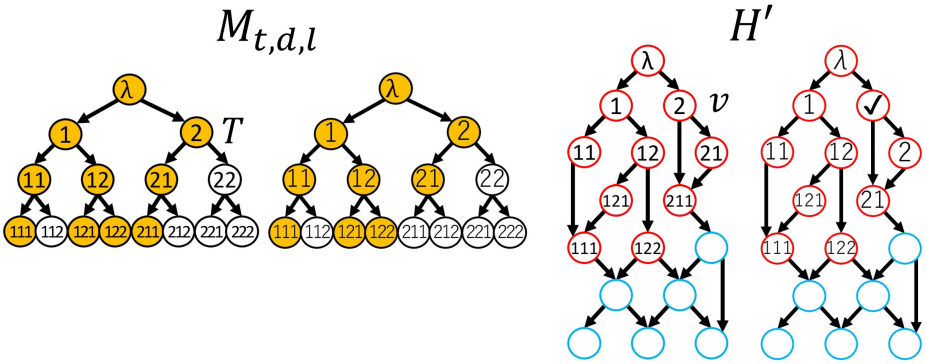


**Fig. 3.** Illustration of token replacement in FindEmbedding. A vertex $v$ satisfying the condition in line 5 of the pseudocode (Algorithm 2) and the token $T$ placed at $v$ are selected ($M_{t,d,l}$ left, $H'$ left), and $T$ is removed from $v$. Subsequently, $T$'s all *tokened* descendants $T \cdot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

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 $H'$.

The algorithm FindEmbedding terminates when one of the following conditions is met: (1) in line 4, $H'$ no longer contains any blue vertices; (2) in line 4, $|X_i| = |V[M_{t,d,l}]|$ holds; or (3) the process in line 11 is executed. The following lemmas demonstrate that the algorithm functions correctly in each of these cases.

**Lemma 4.** *If* FindEmbedding *terminates under condition (1), then there exists a DAG-path-decomposition of $H$ with width at most $ld^{t+3}-1$.*

*Proof.* Suppose that in line 4 of FindEmbedding, all vertices of $H'$ have turned red at step $i=s$, leading to termination. 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 rule 1 of the DAG-path-decomposition. 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 rule 3.

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 rule 2. 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 $ld^{t+3}-1$.

**Lemma 5.** *If* FindEmbedding *terminates under condition (2), then $H$ can be splited 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 $ld^{t+3}-1$.*

*Proof.* Suppose that in line 4 of FindEmbedding, at step $i=s$, 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 $A'$, rule 1 of the DAG-path-decomposition is clearly satisfied. Additionally, rules 2 and 3 are satisfied by the argument similar to **Lemma 4**. 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 rule 2 of the DAG-path-decomposition 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 3 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 4**.

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 3** 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 $V[A']\backslash V[A]$

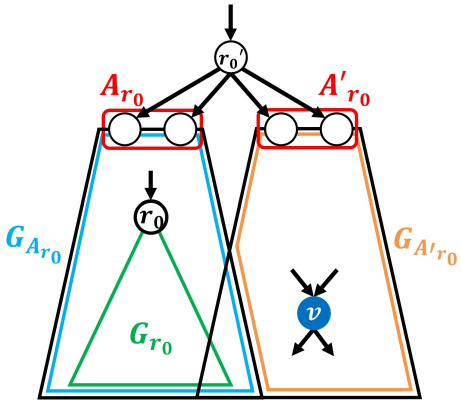


**Fig. 4.** Illustration of the proof of **Lemma 6**. The recent common descendant of $r_0$ and $v$ is denoted as $r_0'$. The proof demonstrates that a contradiction arises if the blue vertex $v$ is not included in $G_{r_0}$, which consists of $r_0$ and its descendants.

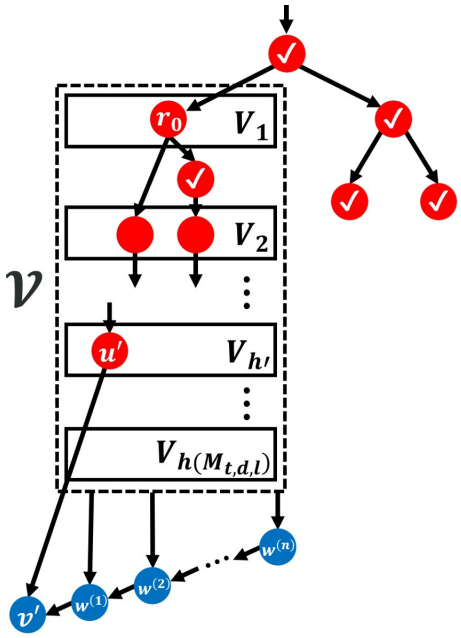


**Fig. 5.** Illustration of the proof of **Lemma 7**. The blue vertices represent blue, the red check-marked vertices indicate token-removed vertices, and the other red vertices indicate vertices with tokens placed on them. The proof demonstrates that a contradiction arises if there are no vertices in $V_{h(M_{t,d,l})}$.

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 2**, 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 $ld^{t+3} - 1$.

Before proving **Lemma 7**, we show the following. The outline of the proof is illustrated in Figure 4.

**Lemma 6.** *At any time $i = k$, let $r_0$ be the vertex of $H'$ on which the root token $\lambda$ 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 $\lambda$ 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 $v \in V[G_{A'_{r_0}}]$.

Consider the first moment $i = l$ ($l \leq k$) when a token $\lambda$ 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 7** is proven.

**Lemma 7.** *If* FindEmbedding *terminates at (3), then the DAG-pathwidth of $H$ is greater than $t$.*

*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 $\lambda$, 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 $\lambda$ 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 6**, 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 6**, 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 $M_{t,d,l}$.

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 row 5 in FindEmbedding in (3), for each token $m_i \in P$ (where $\lambda$ 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 $t$.

**Theorem 1** is shown below.

*Proof.* (**Theorem 1**)
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 4**, **Theorem 1**(a) holds. If it terminates in case (2), by **Lemma 5**, **Theorem 1**(b) holds. If it terminates in case (3), by **Lemma 7** and setting $A = H$, **Theorem 1**(b) holds. Therefore, **Theorem 1** is proven.

**Corollary 1** *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 evidence that the DAG-pathwidth of $H$ is greater than $t$, or provides a DAG-path-decomposition of width at most $O(ld^t)$.*

*Proof.* By **Theorem 1**, FindEmbedding either outputs an evidence that the DAG-pathwidth of $H$ is greater than $t$, or outputs a DAG-path-decomposition of width at most $O(ld^t)$. We now show that FindEmbedding terminates in polynomial time. In GrowTokenTree, the condition check of line 1 takes at most $O(dn)$ 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 at most $O(n)$ times. Therefore, the while loop in line 4 is also repeated at most $O(n)$ 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.

## 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 at most $O(ld^k)$. 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 $d$ may grow up to the vertex count $n$, we aim to explore methods to bound $d$ by a constant.

## Acknowledgements

soon.

## 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. CoRR **abs/1902.04919** (2019)
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., Hliněnỳ, 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. Johnson, T., Robertson, N., Seymour, P., Thomas, R.: Directed tree-width. Journal of Combinatorial Theory, Series B **82, no.1**, 138–154 (2001)
16. 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)
17. Kirousis, L., Papadimitriou, C.: Searching and pebbling. Theoretical Computer Science **47**, 205–218 (1986)
18. Korhonen, T.: A single-exponential time 2-approximation algorithm for treewidth. SIAM Journal on Computing pp. FOCS21–174 (2023)
19. 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)
20. Reed, B.: Tree width and tangles: a new connectivity measure and some applications. Surveys in Combinatorics pp. 87–162 (1997)
21. Robertson, N., Seymour, P.: Graph minors. i. excluding a forest. Journal of Combinatorial Theory, Series B **35, no.1**, 39–61 (1983)
22. Robertson, N., Seymour, P.: Graph minors. iii. planar tree-width. Journal of Combinatorial Theory, Series B **36, no.1**, 49–64 (1984)
23. Sethi, R.: Complete register allocation problems. SIAM Journal on Computing **4**, 226–248 (1975)