# 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 design a polynomial-time approximation algorithm for DAG-pathwidth that achieves an $O(\log^2 n)$ approximation ratio. We further improve this ratio to $O(\log^{3/2} n)$ 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 also design an algorithm that, given an integer $t$ and DAG $G$ with $l$ roots and at most $d$ outdegree, either computes a DAG-path-decomposition of $G$ with width at most $O(ld^t)$ or provides an evidence that the DAG-pathwidth of $G$ 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 [**?**], 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 design an approximation algorithms computing $O(\log^2 n)$ ratio for DAG-pathwidth. We also 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 [**?**], and both of these algorithms utilize the graph embedding of complete trees.

## Preliminaries

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

### 1.1 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.

### 1.2 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 [**?**], 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 [**?**]. 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 [**?**] 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.

### 1.3 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.

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

here.

## 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., 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. 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)