

Polynomial-time algorithms for PATH COVER and PATH PARTITION on trees and graphs of bounded treewidth^{*}

Florent Foucaud^{a,1,*}, Atrayee Majumder^{b,2}, Tobias Mömke^{c,3},
Aida Roshany-Tabrizi^c

^a CNRS, Clermont Auvergne INP, Mines Saint-Étienne, LIMOS, Clermont-Ferrand, 63000, France

^b Department of Mathematics, University of West Bohemia in Pilsen, Czech Republic

^c University of Augsburg, Germany

ARTICLE INFO

Section Editor: Prof. Seth Pettie
Handling Editor: Katie Harris

Keywords:

Path cover
Path partition
Trees
Treewidth

ABSTRACT

In the PATH COVER problem, one asks to cover the vertices of a graph using the smallest possible number of (not necessarily disjoint) paths. While the variant where the paths need to be pairwise vertex-disjoint, which we call PATH PARTITION, is extensively studied, surprisingly little is known about PATH COVER. We start filling this gap by designing a linear-time algorithm for PATH COVER on trees. We show that PATH COVER can be solved in polynomial time on graphs of bounded treewidth using a dynamic programming scheme. It runs in XP time $n^{t^{O(t)}}$ (where n is the number of vertices and t the treewidth of the input graph) or $\kappa^{t^{O(t)}}$ if there is an upper-bound κ on the solution size. A similar algorithm gives an FPT $2^{O(t \log t)} n$ algorithm for PATH PARTITION, which can be improved to (randomized) $2^{O(t)} n$ using the Cut&Count technique. These results also apply to the variants where the paths are required to be induced (i.e. chordless) and/or edge-disjoint.

1. Introduction

Path problems in graphs are fundamental problems in algorithmic graph theory, consider for example the problem of computing shortest paths in a graph, which has been one of the first studied graph problems for which efficient algorithms were obtained [2, Chapter 24]. Finding *disjoint paths* is also a problem of utmost importance, both in algorithmic and structural graph theory [3]. When it comes to *covering* the graph, the HAMILTONIAN PATH problem is another classic path-type problem studied both in combinatorics and computer science. We study its generalizations, PATH COVER and PATH PARTITION, which are about covering the vertices of a path using a minimum number of paths (unrestricted and pairwise vertex-disjoint, respectively). They are formally defined as follows.

^{*} A preliminary version of the paper appeared in the proceedings of the CALDAM 2025 conference [1].

^{*} Corresponding author.

E-mail address: florent.foucaud@uca.fr (F. Foucaud).

¹ Funded by French government IDEX-ISITE initiative 16-IDEX-0001 (CAP 20-25), International Research Center “Innovation Transportation and Production Systems” of the I-SITE CAP 20-25, and ANR project GRALMECO (ANR-21-CE48-0004).

² Supported by Academic Career in Pilsen Program, 2025 cycle.

³ Partially supported by DFG Grant 439522729 (Heisenberg-Grant).

PATH COVER**Input:** A graph G .**Task:** Compute a minimum-size *path cover*, that is, a set of paths of G such that every vertex of G belongs to at least one of the paths.

Its variant that requires the solution paths to be pairwise vertex-disjoint is very well-studied and defined as follows.

PATH PARTITION**Input:** A graph G .**Task:** Compute a minimum-size *path partition*, that is, a set of pairwise vertex-disjoint paths of G such that every vertex of G belongs to exactly one of the paths.

Our goal is to study the algorithmic complexity of PATH COVER and PATH PARTITION on trees and graphs of bounded treewidth. Treewidth is an important graph parameter and the associated tree decompositions enable to solve various problems efficiently when this parameter is bounded. We refer to the book [4] for more on the topic of algorithms for graphs of bounded treewidth.

Related work. As both problems generalize HAMILTONIAN PATH (which amounts to decide whether a graph can be covered by a single path), they are both NP-hard, and this holds even, for example, for 2-connected cubic bipartite planar graphs [5]. Note that HAMILTONIAN PATH is W[1]-hard for cliquewidth [6] (this is originally proved for HAMILTONIAN CYCLE, but a standard reduction from HAMILTONIAN CYCLE to HAMILTONIAN PATH yields the result: take any vertex, make a twin for it, and attach a pendant vertex to each twin). Hence, both PATH COVER and PATH PARTITION are W[1]-hard for cliquewidth, which generalizes treewidth (in the sense that every graph of bounded treewidth has bounded cliquewidth).

Most work in the literature refers to PATH PARTITION as “PATH COVER”, which is unfortunately misleading. PATH PARTITION is much more studied than PATH COVER, see [7,8] for examples of such works on PATH PARTITION. In some cases, PATH PARTITION was also called HAMILTONIAN COMPLETION [9–11]. To avoid any confusion between the two problems, we use the terminology of PATH COVER and PATH PARTITION as defined above, a choice taken from the survey [12] on path-type problems. Also see the PhD theses [13,14].

Several papers from the 1970s studied PATH PARTITION on trees [10,11,15]. However, they did not explicitly analyze the running times of their algorithms. A properly analyzed linear-time algorithm was given in 2002 [9]. PATH PARTITION was also shown to be solvable in polynomial time on many other graph classes such as cographs [16], distance-hereditary graphs [8], co-comparability graphs [7] (which contain all interval graphs), or block graphs/cactii [17]. We do not know of any explicit algorithm for PATH PARTITION on graphs of bounded treewidth, but algorithms exist for the closely related CYCLE PARTITION problem [18].

We refer to [14,19–22] for the few references about PATH COVER that we are aware of. Notably, a polynomial-time algorithm for PATH COVER on block graphs was given in the PhD thesis [14]. The variants where we need to cover the *edges* of the graph are also studied, see [23,24].

Both PATH COVER and PATH PARTITION have numerous applications, in particular in program testing [21], circuit testing [23], or machine translation [25], to name a few. Although PATH PARTITION is more widely studied, PATH COVER is also a natural problem, with specific applications in bio-informatics when restricted to directed acyclic graphs [19,22].

Variants of PATH COVER and PATH PARTITION are also studied, where the solution paths must be induced/chordless, shortest paths, bounded length paths, etc. We refer to [12] for a survey. Recently, the variant of PATH PARTITION with shortest (isometric) paths was studied with respect to treewidth, where it was shown to be XP but W[1]-hard for treewidth [26].

Our results. Although PATH PARTITION is well-studied on trees and other graph classes, surprisingly, this is not the case of PATH COVER. Note that despite the two problems having similar statements, they typically have very different optimal solutions. For example, on a star with k leaves, an optimal path partition has size $k - 1$, but an optimal path cover has size $\lceil k/2 \rceil$.

We first focus (in Section 2) on PATH COVER on trees, for which no linear-time algorithm has been given in the literature. For trees, the size of an optimal solution is given by the ceiling of half of the number of leaves. The proof of this fact was given by Harary and Schwenk in 1972 [24] for the problem of covering the *edges* of the tree. An analysis of their proof yields a quadratic-time algorithm. We show how PATH COVER can in fact be solved in linear time on trees, by giving an improved algorithm based on depth-first-search (DFS).

We then study graphs of bounded treewidth in Section 3. We design an explicit dynamic programming algorithm that runs in time $n^{t^{O(t)}}$ for graphs of treewidth at most t and order n . In fact, this algorithm runs in time $\kappa^{t^{O(t)}} n$, where κ is the maximum number of allowed paths in a solution (the “solution size” if we reformulate PATH COVER as a decision problem). With a slight simplification, the same algorithm also solves PATH PARTITION and runs in improved FPT running time $2^{O(t \log t)} n$. However, we show in Section 4 how to obtain a randomized algorithm for PATH PARTITION running in time $2^{O(t)} n$ using the Cut&Count technique.

It is not clear whether PATH COVER can be solved in FPT time as well or not; however, we give some indications of why that might not be the case.

Moreover, we argue that our algorithms also apply to the versions of both PATH COVER and PATH PARTITION where the paths in the solution are required to be *induced* (i.e. chordless) or pairwise edge-disjoint. These variants have been studied in the literature (see [12,27] and references therein).

We finally conclude in Section 5.

2. A linear-time algorithm for PATH COVER on trees

We first study PATH COVER on trees. In [24, Theorem 7], it is proved that the minimum number of paths needed to cover the *edges* of a tree is equal to $\lceil \ell/2 \rceil$, where ℓ is the number of leaves. This is an obvious lower bound, since for any leaf of a tree, there must be a solution path starting at that leaf. This also holds for covering the *vertices*. The argument of [24], based on pairing the leaves arbitrarily and switching the pairing to increase the number of covered vertices at each step, leads to a quadratic-time algorithm for PATH COVER on trees. We next present an algorithm for solving PATH COVER on trees with a runtime that is linear in the number of vertices.

Theorem 1. *PATH COVER can be solved in linear time on trees, and the optimal size of a solution for a tree with ℓ leaves is $\lceil \ell/2 \rceil$.*

Consider the input tree T to be rooted at an arbitrary internal vertex r of T . The intuition here is to cover the vertices of the tree by simulating the Depth-First-Search (DFS) algorithm with some modifications in the steps, thus the running time would become the same as the running time of DFS.

First, we recall the recursive DFS algorithm, see e.g. [2, Chapter 22.3]. In this algorithm, there are two timestamps assigned to each vertex: the first timestamp is given when we discover the vertex for the first time while traversing the tree, and the second timestamp is given when we finish traversing all the vertices of the sub-tree rooted at that particular vertex. We use these timestamps in our algorithm. We use the first timestamp to mark the vertex v as visited and the second timestamp to consider a valid solution of PATH COVER to cover the vertices rooted at v , including v . The algorithm starts at the root and marks the vertices as visited in DFS order until it reaches a leaf. At the leaf on the way back, it starts a path with endpoints at the leaf and continues toward the parent, and the paths are recorded. Now we need some definitions to specify the steps of our algorithm.

We use $p = (x_1, x_2, \dots, x_k)$ as a notation for paths where x_i are all distinct vertices of T connected by the path p . The vertices $x_2, \dots, x_{k-1}$ are internal vertices; the vertices x_1 and x_k are endpoints of the path p .

Definition 2. A path is *closed* if both of its endpoints are leaves, and a path is *open* if one of its endpoints is not a leaf. We assume a singleton path at a leaf of the tree to be an open path.

Definition 3. Let v be a vertex in the tree T and $\mathcal{P}$ be a set of paths in the subtree rooted at v . For instance $p_1 = (a_1, \dots, a_j)$ and $p_2 = (b_1, \dots, b_k)$ are two paths in $\mathcal{P} = \{p_1, p_2, \dots\}$ where a_1 and b_1 are leaves of T . We define the following operations:

- We use the notation $\mathcal{P} \cdot v$ to *concatenate* the paths in $\mathcal{P}$ with vertex v , that is, For each path $p \in \mathcal{P}$ if there exists an edge between v and the endpoint of $p \in \mathcal{P}$, then we add vertex v to the path p . As an example, if $a_j v, b_k v \in E(T)$, then $\mathcal{P} \cdot v = \{(a_1, \dots, a_j, v), (b_1, \dots, b_k, v), \dots\}$. If $\mathcal{P} = \{p\}$ for a single path p , we write $p \cdot v$ as shorthand for $\mathcal{P} \cdot v$.
- Two vertex-disjoint open paths $p_1 = (a_1, \dots, a_j)$ and $p_2 = (b_1, \dots, b_k)$ are *combined* at the vertex v if v is connected to one of the endpoints of each path. By combining p_1 and p_2 at vertex v , we obtain a new path, formally, $\text{comb}(p_1, p_2) = (a_1, \dots, a_j, v, b_k, \dots, b_1)$.

Let T_v be the subtree of T rooted at a given vertex v . Let $\{T_v^1, T_v^2, \dots\}$ be the set of connected components of $T_v \setminus \{v\}$. Two open paths coming from these connected components to v can only be combined at v if they are coming from two different components from $\{T_v^1, T_v^2, \dots\}$. Since, for the combining operation, the paths need to be vertex-disjoint, the open paths coming from the same component to v can not be combined at v . Two open paths are called *unrelated* if they come from two different components from $\{T_v^1, T_v^2, \dots\}$ at v .

We say that a path is *extended* at a vertex v if it does not terminate at v , but continues through v towards the root.

In the algorithm, for each vertex $v \in T$, we define three sets of paths:

- $\mathcal{P}_v^{\text{close}}$ is a set of paths that consists of *closed* paths combined at vertex v ,
- $\mathcal{P}_v^{\text{open}}$ is a set of paths that consists of *open* paths that will be extended further from vertex v ,
- $\mathcal{P}_v$ is the set of all paths that are extended as open paths from the children of v .

Among these sets, we need to store the set $\mathcal{P}_v^{\text{open}}$ corresponding to each node for further usage in the parent node of v .

The aim of our algorithm is to have all paths closed, and when a path is open it means we extend the path until it gets combined and closed. Note that we mark a vertex covered when all the vertices of the subtree rooted at that vertex are visited and covered. Now, we present Algorithm 1 to compute a path cover of T . The input to our algorithm is a tree T , a designated root vertex r , and a solution set S , which is initially $\emptyset$. After the execution of Algorithm 1, the solution set S provides an optimal path cover of T . We store a path $p = (a_1, a_2, \dots, a_j)$ in terms of its endpoints i.e. $p = (a_1, a_j)$.

Throughout this section, r denotes the fixed root of T , whereas v denotes the vertex processed in the current recursive call. The algorithm is initially called as $PC(T, r, S)$.

Algorithm 1 $PC(T, r, S)$.

```

1: Mark the current vertex  $v$  as visited
2: for Each child  $u$  of  $v$  do
3:   if  $u$  is unvisited then
4:     recursively call  $PC(T_u, u, S)$ 
5:   end if
6: end for
7: if  $v$  is a leaf then
8:    $\mathcal{P}_v^{open} \leftarrow \{(v, v)\}$     and     $\mathcal{P}_v^{close} \leftarrow \emptyset$ 
9: else
10:   $\mathcal{P}_v \leftarrow \bigcup_c \mathcal{P}_c^{open}$                                  $\triangleright$  taking a union over all the paths coming from all children  $c$  of  $v$ 
11:   $\mathcal{P}_v^{close} \leftarrow \emptyset$ 
12:  while  $|\mathcal{P}_v| > 2$  do
13:    Find two unrelated paths  $p_i$  and  $p_j$ 
14:     $\mathcal{P}_v^{close} \leftarrow \mathcal{P}_v^{close} \cup \text{comb}(p_i, p_j)$ 
15:     $\mathcal{P}_v \leftarrow \mathcal{P}_v \setminus \{p_i, p_j\}$ 
16:  end while
17:  if  $v = r$  and  $|\mathcal{P}_v| = 2$  then
18:     $\mathcal{P}_v^{close} \leftarrow \mathcal{P}_v^{close} \cup \text{comb}(p_1, p_2)$ 
19:  else if  $v = r$  and  $|\mathcal{P}_v| = 1$  then
20:     $\mathcal{P}_v^{close} \leftarrow \mathcal{P}_v^{close} \cup p_1 \cdot v$ 
21:  else
22:     $\mathcal{P}_v^{open} \leftarrow \mathcal{P}_v \cdot v$                                  $\triangleright$  when  $v \neq r$  and  $|\mathcal{P}_v| \leq 2$ 
23:  end if
24: end if
25: Mark  $v$  as covered
26:  $S \leftarrow S \cup \mathcal{P}_v^{close}$ 
27: return  $\mathcal{P}_v^{open}$ 

```

The algorithm begins its search from the root, marks each vertex *visited*, and recursively calls itself until it reaches a leaf (say v) [line 1–6]. The algorithm starts a singleton path (v, v) from the leaf and extends the path to its parent as an open path (line 8) and marks v as *covered*.

During the backtracking process, at each visited vertex v , the algorithm marks v as *covered* when all its children are visited and *covered*. For each vertex v , other than a leaf and the root of T , $\mathcal{P}_v$ contains all the open paths coming from its children. It is clear from line 22 that each child can forward at most two open paths to its parent. Therefore if $|\mathcal{P}_v| \geq 3$, we will always get two unrelated paths coming from two different children of v that can be combined to create a closed path. In the while loop in (12–16) we precisely do it. The algorithm constructs $\mathcal{P}_v^{close}$ by combining all possible pairs of unrelated paths at v and adding them to the final solution set S (line 26).

The steps the algorithm follows for the root r of T are the same as any other internal vertex, except all the open paths are closed at this point. If the number of open paths in $\mathcal{P}_r$ is even, all the paths are combined at r and closed (line 18). Note that r is an internal vertex of T . Therefore, the number of children of $r \geq 2$. Hence, it is always possible to close all pairwise unrelated paths at r . If the number of paths in $\mathcal{P}_r$ is odd, we make r as the endpoint of the only remaining open path, which cannot be combined with any other path (line 20). We are now ready to prove [Theorem 1](#).

Proof. (Proof of [Theorem 1](#)) First, we have to prove that the size of S is $\lceil \frac{\ell}{2} \rceil$ where the number of leaves of T is ℓ . We start a path in the leaf of T and keep it as an open path as long as it is not combined with another path and closed. When a path is closed, both of its endpoints become leaves of T . Additionally, in the root, all the paths get closed except for possibly one path. Hence, for an even number of leaves, the size of S is $\frac{\ell}{2}$ and for an odd number of leaves, the size of S is $\frac{\ell-1}{2} + 1$, which proves the first part of the theorem. Now, we show that the running time of [Algorithm 1](#) is $O(n)$ where n is the number of vertices of T . [Algorithm 1](#) closely resembles the DFS algorithm, with the addition of some constant time operations to store the paths covering the vertices. Two operations, $\text{comb}(p, q)$ and $\mathcal{P}_v \cdot v$ take $O(1)$ time as we store the paths by their endpoints, and after each of these operations, only the endpoints are changed while the new paths are created. Finding a pair of unrelated paths can also be done in $O(1)$ time, as the children of a particular vertex can be ordered from left to right, and the paths coming from the children can also be ordered accordingly. For each path p , we just need to check at most two consecutive paths in this ordering to find a path q which is unrelated to p . Therefore, in each of the iterations of the while loop (line 12–16) we do the operations in $O(1)$ time. All the iterations of the while loop together traverse each of the edges between v and its children $O(1)$ time. These edges are only traversed in v 's recursive call. The recursive $PC(T, v, S)$ call is made only once for each vertex. Therefore, the recursive calls altogether make the running time of the algorithm $O(n + m)$, where m is the number of edges of T (since each edge of T is traversed $O(1)$ times), which is same as the running time of DFS. As m is $O(n)$ for T , the total running time becomes $O(n)$, which proves the second part of the theorem. $\square$

3. PATH COVER on graphs of bounded treewidth

In this section, we present an algorithm that solves PATH COVER on general graphs in XP time when parameterized by treewidth. The algorithm is a classic dynamic programming scheme over a tree decomposition.

Theorem 4. *PATH COVER can be solved in time $\kappa^{O(t)} \cdot n$ on graphs of treewidth t and order n , where κ is the maximum allowed solution size.*

Since we can always assume that $\kappa \leq n$, we have the following corollary.

Corollary 5. *PATH COVER can be solved in time $n^{O(t)}$, where n is the number of vertices, and t is the treewidth of the input graph.*

3.1. Preliminaries

Here, we give the definitions of *tree decomposition*, *treewidth*, and a well-structured form of it, a *nice tree decomposition* from [28].

Definition 6. Let G be an undirected graph. A tree decomposition of $G = (V, E)$ is a pair $(\mathcal{T}, X)$, where $\mathcal{T}$ is a tree and $X : V(\mathcal{T}) \rightarrow 2^V$ satisfies the following conditions:

- for each $e \in E$ there exists a node $v \in V(\mathcal{T})$ such that both endpoints of e are inside X_v , a bag associated with v .
- for each $u \in V$ the set $\{v \in V(\mathcal{T}) : u \in X_v\}$ is connected in $\mathcal{T}$.

The *width* of $(\mathcal{T}, X)$ is $\max_{v \in V(\mathcal{T})} |X_v| - 1$. The *treewidth* of a graph G , denoted by t , is the minimum width of all possible tree decompositions of G .

We distinguish between vertices of G and vertices of the tree decomposition by referring to the latter as *nodes*. Each node v is associated with a *bag* X_v : a subset of vertices of graph G . We use the classic *nice tree decompositions*:

Definition 7 ([28]). A tree decomposition $\mathcal{T}$ of graph G is *nice* if:

- $\mathcal{T}$ is rooted at node r with $X_r = \emptyset$.
- Each leaf node v of $\mathcal{T}$ has an empty bag $X_v = \emptyset$.
- Each non-leaf node is of one of the types below:
 1. An **Introduce node** v has exactly one child u such that $X_v = X_u \dot{\cup} \{x\}$ for some vertex x in V .
 2. A **Forget node** v has exactly one child u such that $X_v = X_u \setminus \{x\}$ for some vertex x in V .
 3. A **Join node** v has two children u_1, u_2 such that $X_v = X_{u_1} = X_{u_2}$.

Given a graph G on n vertices and an integer t , there is an algorithm that either outputs a tree decomposition of G of width at most $2t + 1$, or determines that the treewidth of G is larger than t , in time $2^{O(t)} n$ [29]. Given a tree decomposition of G of width t and $O(n)$ nodes, a nice tree decomposition of width t with at most $4n$ nodes can be constructed in time $O(t \cdot n)$ [30].

Let $G = (V, E)$ be a graph and $(\mathcal{T}, X)$ be a nice tree decomposition of G with width at most t . Let $\mathcal{T}_v$ be the subtree of $\mathcal{T}$ rooted at node v and V_v be the union of all bags in this subtree including X_v . We define G_v as the subgraph of G induced by V_v . We will use the following lemma from [4].

Lemma 8. *Let $(\mathcal{T}, X)$ be a tree decomposition of a graph G and let uv be an edge of $\mathcal{T}$. The forest $\mathcal{T} - uv$ obtained from $\mathcal{T}$ by deleting edge uv consists of two connected components $\mathcal{T}_u$ and $\mathcal{T}_v$. Then (V_u, V_v) is a separation of G with separator $X_u \cap X_v$.*

3.2. The dynamic programming scheme

We implement our dynamic programming scheme for PATH COVER in a bottom-up manner, starting at the leaf nodes of $\mathcal{T}$. At each node v , we deal with a potential solution for G that covers the vertices of X_v . We define a path as a sequence of vertices where any vertex in a path p is incident to at most two vertices from p , giving at most two neighbours in the subgraph induced by p . Each neighbour of a vertex in p can either belong to $G_v - X_v$ (and thus have been forgotten in the bag of some descendant node of v), which we represent with “ $\downarrow$ ”, or belong to $G - G_v$ (it will appear in the bag of some ancestor node of v), which we represent by “ $\uparrow$ ”, or belong to X_v , which we represent by “ $-$ ”. Therefore, we characterize each vertex by the types of its neighbour(s) inside a potential solution path. To this end, let $\mathcal{N}$ be the set of multisets of size at most two whose elements are taken from the set $\{-, \uparrow, \downarrow\}$.

At each node v , we define a *partial path* p (representing the intersection of a path of G with the bag X_v) by (Π_p, φ_p) where Π_p is an ordered subset of X_v and φ_p is a function $\varphi_p : \Pi_p \rightarrow \mathcal{N}$ which describes, for each vertex of p , the types of its neighbour(s) inside the path of G represented by p . These paths are called “partial” since they consist of several paths, and might have neighbour(s) in the path that lie outside the node. Each vertex x of a partial path p in X_v has the possibility of being in either of the types illustrated in Fig. 1 with respect to the path p' of G represented by p .

Definition 9. The set $\mathcal{N}$ of types of a vertex with respect to a partial path p of a node v are as follows:

1. $p = p'$ is a one-vertex path (and thus x has no neighbours in p'); x is assigned the type $\emptyset$.
2. x is an endpoint of p' (with a unique neighbour in p'), which is inside the bag; it is assigned the type $\{-$.

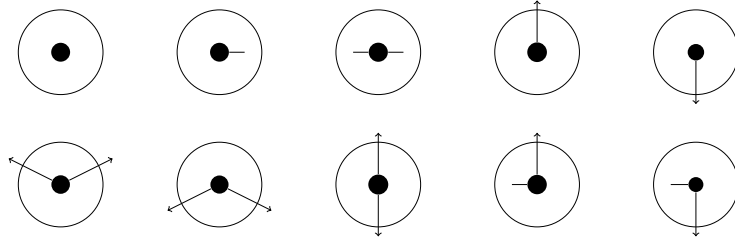


Fig. 1. Illustration of the different types of a vertex with respect to some node v and partial path. The set of types is $\mathcal{N} = \{\emptyset, \{-\}, \{-, -\}, \{\uparrow\}, \{\downarrow\}, \{\uparrow, \uparrow\}, \{\downarrow, \downarrow\}, \{\downarrow, \uparrow\}, \{\uparrow, -\}, \{\downarrow, -\}\}$. “ $-$ ” shows a vertex with a neighbour inside the bag X_v , “ $\uparrow$ ” shows a vertex with a neighbour in $G - G_v$, and “ $\downarrow$ ” shows a vertex with a neighbour in $G_v - X_v$.

3. x is an internal vertex of p' with two neighbours in p' , both inside the bag; it is assigned the type $\{-, -\}$
4. x is an endpoint of p' with a unique neighbour in p' , in an ancestor bag; it is assigned the type $\{\uparrow\}$.
5. x is an endpoint of p' with a unique neighbour in p' , in a descendant bag; it is assigned the type $\{\downarrow\}$.
6. x is an internal vertex of p' with two neighbours in p' , both in ancestor bags; it is assigned the type $\{\uparrow, \uparrow\}$.
7. x is an internal vertex of p' with two neighbours in p' in descendant bags; it is assigned the type $\{\downarrow, \downarrow\}$.
8. x is an internal vertex of p' with a neighbour in p' in a descendant bag and a neighbour in p' in an ancestor bag; it is assigned the type $\{\downarrow, \uparrow\}$.
9. x is an internal vertex of p' with a neighbour in p' in an ancestor bag and a neighbour in p' inside the bag itself; it is assigned the type $\{\uparrow, -\}$.
10. x is an internal vertex of p' with a neighbour in p' in a descendant bag and a neighbour in p' inside the bag itself; it is assigned the type $\{\downarrow, -\}$.

We say that a partial path for node v is *consistent* if it corresponds to the intersection with X_v of some path of G .

Now, we extend the notion of a partial path p of a node v to that of a partial path p of a subgraph G_v in a natural way as follows. A partial path p of G_v is an ordered subset Π_p of V_v and a function $\varphi_p : \Pi_p \rightarrow \mathcal{N}$ which describes, for each vertex of a path, the types of its neighbour(s) inside the path, where “ $\uparrow$ ” now refers to a neighbour in $V(G) \setminus V_v$, “ $-$ ” refers to a neighbour in V_v , and there is no vertex of p whose type contains “ $\downarrow$ ”.

We say that a partial path p of v *agrees* with a partial path p' of G_v if the following conditions hold:

- $\Pi_p \subseteq \Pi_{p'}$;
- the order of the vertices in $\Pi_p \cap \Pi_{p'}$ is the same in Π_p and $\Pi_{p'}$;
- all vertices in $\Pi_{p'} \setminus \Pi_p$, have a type in $\{-\}, \{-, -\}, \emptyset$;
- $\varphi_p(x) = \varphi_{p'}(x)$, for each vertex x with types $\emptyset, \{-\}, \{-, -\}, \{\uparrow\}$, and $\{\uparrow, -\}$;
- If $\varphi_p(x) = \{\downarrow\}$ then $\varphi_{p'}(x) = \{-\}$;
- If $\varphi_p(x) = \{\uparrow, \uparrow\}$ then $\varphi_{p'}(x) = \{\uparrow, \uparrow\}$;
- If $\varphi_p(x) = \{\downarrow, \downarrow\}$ then $\varphi_{p'}(x) = \{-, -\}$;
- If $\varphi_p(x) = \{\uparrow, \downarrow\}$ then $\varphi_{p'}(x) = \{\uparrow, -\}$;
- If $\varphi_p(x) = \{\downarrow, -\}$ then $\varphi_{p'}(x) = \{-, -\}$.

For a node v , we define a *partial solution* S of G_v as a collection P of partial paths of G_v whose vertices cover all the vertices in V_v . We require that every partial path is consistent, that is, it may correspond to the intersection of a path of G with G_v . We also require that any partial path that has no vertex whose type contains “ $\uparrow$ ”, is unique (i.e. appears only once in S). However, partial paths with vertices whose type contains “ $\uparrow$ ” may appear multiple times (at most n times). Indeed, they might correspond to different paths in a path cover of G , that all happen to have the same intersection with G_v . For notational convenience, a partial solution S is stored as a set P of partial paths, and a function f where, for every partial path p in S , we let $f(p) \in \{1, \dots, n\}$ be the number of times the partial path p appears in the partial solution S .

Let P be a set of partial paths of a node v together with a function $f : P \rightarrow \{1, \dots, n\}$, and $S = (P', f')$ be a partial solution of G_v . Let M be the multiset obtained from P by adding, for every partial path p of P , $f(p)$ copies of p to M . Similarly, let M' be the multiset obtained from P' obtained from the subset of partial paths of P' that intersect X_v , by adding to M' , $f'(p)$ copies of each such path p of P' . We say that P *corresponds to* S if there is a bijection ψ from the M to M' , such that p agrees with $\psi(p)$ for every partial path p of P .

We are now ready to describe the dynamic programming scheme.

Dynamic Programming. We define a DP-state $[v, P, f]$ as follows:

- v is a node in a nice tree decomposition $(\mathcal{T}, X)$;

- P is a set of partial paths that covers the vertices of X_v ;
- f is a function $f : P \rightarrow \{1, \dots, n\}$ that maps each partial path p in P to a value that shows how many times p is used.

We say that a DP-state $[v, P, f]$ is *valid* if every partial path in P is consistent, and (P, f) corresponds to a partial solution S of G_v .

For a valid DP-state, we define $\text{opt}[v, P, f]$ as the minimum number of partial paths in a partial solution of G_v corresponding to (P, f) . If there exists no such partial solution, then the DP-state is invalid, and $\text{opt}[v, P, f] = \infty$.

Now, we explain how to compute the value of a DP-state from the values of the children's DP-states. We need to check the compatibility condition between DP-states of a node and the children's DP-states; this is specific for each node type. Therefore, we consider each case individually.

Let v be an introduce node with a child node u . Two valid DP-states $[v, P, f]$ and $[u, P', f']$ are *compatible* if there exist partial solutions S_v of G_v and S_u of G_u corresponding to (P, f) and (P', f') respectively, such that the intersection of S_v with G_u is S_u . Let v be a forget node with a child u . Two valid DP-states $[v, P, f]$ and $[u, P', f']$ are *compatible* if there exist partial solutions S_v of G_v and S_u of G_u corresponding to (P, f) and (P', f') respectively, such that the intersection of S_u with G_v is S_v .

Let v be a join node with two children u_1 and u_2 . Three valid DP-states $[v, P, f]$, $[u_1, P', f']$, and $[u_2, P'', f'']$ are *compatible* if there exist partial solutions S_u, S_{u_1}, S_{u_2} of G_v, G_{u_1} and G_{u_2} corresponding to (P, f) , (P', f') and (P'', f'') respectively, such that the intersection of S_v with G_{u_1} is S_{u_1} and the intersection of S_v with G_{u_2} is S_{u_2} .

The algorithm proceeds as follows. It starts at the leaf nodes and approaches the root. At each node, the algorithm computes a value for each possible DP-state using the values of the children's compatible DP-states, and chooses the minimum possibility. We next describe the computations done at each node, depending on their type.

Leaf node. For each leaf node v , $X_v = \emptyset$, so it is trivial that $\text{opt}[v, \emptyset, f] = 0$.

Introduce node. Let v be an introduce node with child node u such that $X_v = X_u \cup \{x\}$ for some $x \in V(G)$. Let $[v, P, f]$ be a valid DP-state for node v .

Note that if there is a partial path $p \in P$ and x with one of the values $\varphi_p(x) = \{\downarrow\}$, $\varphi_p(x) = \{\uparrow, \downarrow\}$, $\varphi_p(x) = \{-, \downarrow\}$, and $\varphi_p(x) = \{\downarrow, \downarrow\}$, then the DP-state is not valid and $\text{opt}[v, P, f] = \infty$. Indeed, by Lemma 8, there is no edge between x and a vertex in $V_v \setminus X_v$.

Otherwise, we will look for all DP-states $[u, P', f']$ for u compatible with $[v, P, f]$. To check whether $[u, P', f']$ and $[v, P, f]$ are compatible, we only need to check the type of x in every partial path p (and the type of its neighbour(s) in p), since the remaining state in X_u must be the same as X_v .

Let P_x be the set of partial paths of P that contains at least one vertex other than x . We must find a bijection between the partial paths of P_x and those of P' (taking into account their multiplicities), in the following way. For every partial path p of P not containing x at all, we require that p also belongs to P' , with $f(p) = f'(p)$. Moreover, for every partial path p of P containing x and at least one other vertex of X_v , there must be a partial path p' of P' such that p' agrees with p after removing x .

Let C be the collection of all DP-states $[u, P', f']$ that are compatible with DP-state $[v, P, f]$ and let k be the number of partial paths of P (accounting for their multiplicity via function f) containing only x . We have:

$$\text{opt}[v, P, f] = \min_{[u, P', f'] \in C} \{\text{opt}[u, P', f']\} + k$$

To justify the validity of this formula, note that (by induction hypothesis) our process constructs only valid partial solutions. Indeed, consider a partial solution S' corresponding to (P', f') of size $\text{opt}[u, P', f']$, where $[u, P', f'] \in C$ is such that $\text{opt}[u, P', f']$ is minimum. We obtain a partial solution S of size $|S'| + k$ corresponding to (P, f) by extending S' . To do so, include vertex x into the partial paths of P' that correspond to partial paths of P containing x , as explained in the compatibility check. Moreover, add as many singleton partial paths x as needed. This shows that $\text{opt}[v, P, f] \leq |S| \leq |S'| + k = \min_{[u, P', f'] \in C} \{\text{opt}[u, P', f']\} + k$.

Conversely, note that an optimal partial solution S of size $\text{opt}[v, P, f]$ corresponding to (P, f) can be transformed into another one, S' , by deleting x from it, with $|S'| = |S| - k$ and S' corresponds to some (P', f') where $[u, P', f']$ is compatible with $[v, P, f]$. This shows that $\min_{[u, P', f'] \in C} \{\text{opt}[u, P', f']\} \leq |S'| \leq \text{opt}[v, P, f] - k$ and thus, $\text{opt}[v, P, f] \geq \min_{[u, P', f'] \in C} \{\text{opt}[u, P', f']\} + k$.

Forget node. Let v be a forget node with child u such that $X_v = X_u \setminus \{x\}$ for some $x \in V$. Let $[v, P, f]$ be a valid DP-state of X_v .

Again, we will look for all DP-states $[u, P', f']$ compatible with $[v, P, f]$. To do so, similarly as in an introduce node, we need to check the type of x (and its neighbours in the bag) for every partial path p of $[u, P', f']$. We observe that in $[u, P', f']$, x cannot have a type containing “ $\uparrow$ ” since by Lemma 8, x cannot be adjacent to a vertex of $V(G) \setminus V_v$, so this type of DP-states can be disregarded. For every partial path p in P , we need to find a partial path p' of P' such that p' agrees with p , that is, either $p = p'$, or p' is obtained from p by removing x . Moreover, the multiplicities of the partial paths of P corresponding to those of P' have to match. Moreover, the only partial paths of P' not corresponding to any partial paths of P are those containing only x ; every other partial path of P' must correspond to one of P .

Let C be the collection of all DP-states $[u, P', f']$ that are compatible with $[v, P, f]$. We have:

$$\text{opt}[v, P, f] = \min_{[u, P', f'] \in C} \{\text{opt}[u, P', f']\}$$

As for the introduce node, the upper bound follows because one can construct a partial solution of G_v corresponding to $[v, P, f]$ using the partial solutions for G_u in C . The lower bound follows because one can, conversely, create a partial solution for G_u corresponding to some DP-state of C by using one for G_v corresponding to $[v, P, f]$.

Join node. Let v be a join node with children u_1 and u_2 such that $X_v = X_{u_1} = X_{u_2}$. Let $[v, P, f]$ be a valid DP-state for v .

As before, we will check the compatibility of $[v, P, f]$ with (valid) DP-states $[u_1, P', f']$ and $[u_2, P'', f'']$. Similarly as before, the DP-states $[v, P, f]$, $[u_1, P', f']$, and $[u_2, P'', f'']$ are compatible if there is a bijection ψ_1 between the partial paths of P and those of P' (again, accounting for their multiplicities, that is, we consider $f(p)$ copies of each partial path p), and a similar bijection ψ_2 between the partial paths of P and those of P'' . These bijections must be so that for any copy p' of a partial path p of P , $\psi_1(p')$ and $\psi_2(p')$ have the same vertices and the same orderings: $\Pi_p = \Pi_{\psi_1(p')} = \Pi_{\psi_2(p')}$. Moreover, for any vertex x in a copy p' of a partial path $p \in P$, if $\varphi_p(x)$ contains “↓”, then together $\varphi_{\psi_1(p')}(x)$ and $\varphi_{\psi_2(p')}(x)$ must contain the same number of “↓” as $\varphi_p(x)$. If $\varphi_p(x)$ contains “↑” or “−”, then each of $\varphi_{\psi_1(p')}(x)$ and $\varphi_{\psi_2(p')}(x)$ must contain the same number of “↑” and “−” as $\varphi_p(x)$. Finally, if $\varphi_p(x) = \emptyset$, also $\varphi_{\psi_1(p')}(x) = \varphi_{\psi_2(p')}(x) = \emptyset$.

Let k be the number of partial paths of P that intersect X_v , and let C be the collection of all pairs of DP-states $([u_1, P', f'], [u_2, P'', f''])$ compatible with $[v, P, f]$. We have:

$$\text{opt}[v, P, f] = \min_{([u_1, P', f'], [u_2, P'', f'']) \in C} \{ \text{opt}[u_1, P', f'] + \text{opt}[u_2, P'', f''] - k \}$$

To see the validity of the formula, as before, we argue that given two DP-states $[u_1, P', f']$, $[u_2, P'', f'']$ for u_1 and u_2 that are compatible with $[v, P, f]$, we can produce a partial solution that corresponds to (P, f) by merging the partial solutions corresponding to (P', f') and (P'', f'') . By Lemma 8, since X_v is a separator of G and the paths of P, P' and P'' agree on X_v , we indeed obtain a partial solution of G_v . This shows the upper bound of the formula.

For the lower bound, conversely, we argue that given any optimal partial solution S corresponding to (P, f) , we can find two DP-states $[u_1, P', f']$, $[u_2, P'', f'']$ for u_1 and u_2 compatible with $[v, P, f]$ such that S_1 (the partial solution equal to S restricted to G_{u_1}) corresponds to $[u_1, P', f']$, and S_2 (the partial solution equal to S restricted to G_{u_2}) corresponds to $[u_2, P'', f'']$. Thus, the size of S must be at least $|S_1| + |S_2| - k$. Hence, we have $\text{opt}[v, P, f] \geq \text{opt}[u_1, P', f'] + \text{opt}[u_2, P'', f''] - k \geq \min_{([u_1, P', f'], [u_2, P'', f'']) \in C} \{ \text{opt}[u_1, P', f'] + \text{opt}[u_2, P'', f''] - k \}$.

We are now ready to prove Theorem 4.

Proof. (Proof of Theorem 4) Recall that given the graph G of order n with treewidth t , one can compute a nice tree decomposition of width at most $2t + 1$ with at most $4n$ nodes in time $2^{O(t)}n$ [29,30].

Our algorithm goes through this tree decomposition in a bottom-up fashion and computes, for each possible DP-state of the current node, the optimal value for this state using the children’s optimal values. The correctness of this algorithm is given by the above discussion and the correctness of the inductive formulas. The optimal value of a solution is then obtained at the root. To obtain the actual path cover, one may use a standard backtracking procedure to build it inductively.

The running time is dominated by the generation, at each node, of all possible DP-states. By the preliminary discussions, we have $|X_v| \leq 2t + 2$ for each node v of the tree decomposition, if G is of treewidth t . Let us count the number of possible DP-states $[v, P, f]$ for a node v . P is a collection of partial paths that covers the vertices of X_v . Each partial path in P is an ordered subset of X_v , where each vertex has 10 possible types inside the partial path. Therefore, there are at most $(10t)! = t^{O(t)}$ possible partial paths. To each of them, we associate a number between 1 and κ using the function f . There are at most $\kappa^{(10t)!}$ possible functions f , thus, the total number of partial solutions inside each node is at most $\kappa^{t^{O(t)}}$, and we can compute them in this time. We can also check the validity, compatibility, etc of the DP-states in time that is polynomial in the size of a DP-state. In a forget/introduce node, we process all pairs of DP-states coming from the parent and child node, which takes p^2 time (if there are at most p possible DP-states per node), and in a join node, we go through all triples of DP-states coming from the parent and the two children nodes, which takes p^3 time. In each case, we have $p = \kappa^{t^{O(t)}}$ and thus p^3 is still $\kappa^{t^{O(t)}}$. We have at most $4n$ nodes in the tree decomposition; hence, the algorithm solves PATH COVER in time $\kappa^{t^{O(t)}}n$. $\square$

3.3. The case of induced paths and edge-disjoint paths

The case where the solution paths are required to be induced can be handled easily, indeed, a solution path is induced if and only if its intersection with every bag is induced as well. Thus, it suffices to only consider the DP-states where the partial paths have no unwanted chord inside the bag. Otherwise, the algorithm remains the same.

For the edge-disjoint variants, similarly, it suffices to check for every considered DP-state, whether its partial paths are edge-disjoint. Otherwise, the algorithm remains the same.

4. PATH PARTITION on graphs of bounded treewidth

In PATH PARTITION, since the solution paths are vertex-disjoint, the number of paths that go through each bag of the tree decomposition is at most the size of the bag. Hence, a DP-state from the algorithm in Section 3 simply needs a collection of partial paths that forms a partition of the bag into ordered sets, and thus, there are at most $t^{O(t)} = 2^{O(t \log t)}$ possible DP-states for every node, since there are at most $(t + 1)! = t^{O(t)}$ orderings and $t^{O(t)}$ possible partitions of a bag. Thus, we obtain an FPT algorithm for PATH PARTITION in time $2^{O(t \log t)}n$.

We next use the Cut&Count method described in [31] to solve PATH PARTITION in time $2^{O(t)}n$. In [31], they show that this method uses a randomization technique based on the Isolation Lemma [32], which leads to a one-sided error Monte-Carlo algorithm. A one-sided error Monte-Carlo algorithm is a randomized algorithm where a probabilistic bias towards a specific outcome is provided. To be specific, our algorithm is a false-biased one-sided error Monte-Carlo algorithm. Therefore, if the solution to our problem does not

exist, then the algorithm always returns “NO” accurately, whereas it does not always return “YES” accurately; it returns “YES” with a reasonably good probability.

We now define the notions required for the isolation lemma. Let $S = \{S_1, S_2, \dots, S_k\}$ be a family of subsets of a universe $U = \{x_1, x_2, \dots, x_n\}$ and $w : U \rightarrow \mathbb{Z}$ be a weight function.

Definition 10. The weight function w *isolates* the subset family S if there exists a unique set $S_i \in S$ such that $\sum_{x_j \in S_i} w(x_j)$ is minimum.

The following *Isolation Lemma* by Mulmuley, Vazirani and Vazirani [32] is the key ingredient of the Cut&Count technique. This probabilistic lemma was originally introduced to give a parallel algorithm for finding a maximum matching in a general graph. The lemma works for any arbitrary set system and finds applications in parallel computation and randomized reductions.

Lemma 11 (Isolation Lemma [32, Lemma 1]). *If the weights $w(x_i)$ of the elements $x_i \in U$ are chosen uniformly and independently at random from $\{1, 2, \dots, N\}$ then*

$$\Pr[w \text{ isolates } S] \geq 1 - \frac{n}{N}.$$

We use the Cut&Count technique following [31] for our problem. The technique has two parts: the Cut part and the Count part. We use the notion of *markers* (described in [31, Section 3.2]). Let S be the solution set of tuples of the form (P, M) for PATH PARTITION for the given graph $G = (V, E)$, such that $P \subseteq E$ is a path partition of G , and $M \subseteq V$ is a set of marked vertices (markers) with $|M| = k$.

In this method, we consider the set of paths in the solution as a subset of the edges, and in the next part we explain in detail how to check whether a subset of edges corresponds to a path partition. Note that in our case, we do not need to store the sequence of the vertices in a path, as we are looking for the subset of edges which reflects the sequence of the vertices of the path constructed by that particular set of edges. Additionally, in the above setting, it is required that each path in P contains at least one marked vertex from M . This ensures that the number of paths in P is at most k .

The Cut part. We assign weights to the edges and vertices of G (defining the universe $U = E \cup V$) uniformly and independently at random from $\{1, 2, \dots, N\}$, such that $N = 2(|E| + |V|)$. Therefore, the weight function w is defined as $w : E \cup V \rightarrow \{1, 2, \dots, 2(|E| + |V|)\}$.

For a particular integer weight ω , where $0 \leq \omega \leq 2(|E| + |V|)^2$, we define $\mathcal{R}_\omega$, S_ω , and C_ω as follows:

1. Let $\mathcal{R}_\omega$ be the set of tuples of the form (P, M) , where $P \subseteq E$ is a path partition of G , and $M \subseteq V$ is a set of marked vertices with $|M| = k$. $\mathcal{R}_\omega$ denotes the family of possible candidate solutions corresponding to the weight ω , i.e. $w(P \cup M) = \omega$. To check whether P is a path partition of G , we follow the argument below. Let $G[P]$ be the subgraph of G with vertex set V and edge set P . Let c'_p and c''_p be the numbers of singleton and non-singleton components in $G[P]$, respectively. Since P is a collection of vertex-disjoint paths and singleton vertices, $\deg_{G[P]}(v) \leq 2$ for any vertex $v \in V$. The number of degree 0 vertices in $G[P]$ is c'_p , the number of degree 1 vertices in $G[P]$ is $2 \cdot c''_p$, and the remaining vertices, i.e. $(n - 2 \cdot c''_p - c'_p)$ vertices, are degree 2 vertices (assuming n is the number of vertices in $G[P]$). To get the value of c'_p and c''_p we can run a BFS or DFS on $G[P]$.
2. Let S_ω be the solution set of tuples $(P, M) \in \mathcal{R}_\omega$ such that each component in $G[P]$ contains at least one marked vertex from M (if the component is a singleton vertex then that vertex must be a marked vertex from M).
3. Let C_ω be the set of tuples of the form $((P, M), (V(P_1), V(P_2)))$, where $(P, M) \in \mathcal{R}_\omega$ and $(V(P_1), V(P_2))$ is a *consistent cut* of $G[P]$ and $M \subseteq V(P_1)$ i.e. all the marked vertices M are on one side of the cut. A cut is called a consistent cut if for any two disjoint vertices $u \in V(P_1)$ and $v \in V(P_2)$, the edge $uv \notin E(G)$.

Proposition 12. *G has a path partition of size at most k if and only if there exists a weight value ω for which S_ω is non-empty.*

Proof. According to the definition of S_ω , if S_ω is non-empty that means there exists at least one tuple (P, M) such that P is a path partition, and each path in P contains at least one marked vertex from M . Since $|M| = k$, therefore, the number of paths in PATH PARTITION is at most k .

Suppose G has a PATH PARTITION P of size at most k , then from each path in P , we mark one vertex and add it to the set M . Since the paths in P are vertex-disjoint, the size of M is at most k . Let ω be the sum of the weights on the edges of the path P and weights on the marked vertices. Thus, we can define a non-empty S_ω by P and M and corresponding ω . $\square$

The Count Part. We now describe the Count part of the Cut&Count method.

Lemma 13. *Given $G, \omega, \mathcal{R}_\omega, S_\omega$, and C_ω , for each weight value ω , we have $|S_\omega| \equiv |C_\omega| \pmod{2}$.*

Proof. For a tuple $(P, M) \in \mathcal{R}_\omega$, let c_p be the number of connected components of $G[P]$ that do not contain any vertex $v \in M$. Given a tuple $((P, M), (V(P_1), V(P_2))) \in C_\omega$, all the marked vertices $v \in M$ are in $V(P_1)$. Therefore, the components containing no vertices of M are either part of $V(P_1)$ or $V(P_2)$. Hence, for a fixed tuple $(P, M) \in \mathcal{R}_\omega$, there exist 2^{c_p} tuples in C_ω , by varying the consistent cuts $V(P_1)$ and $V(P_2)$. Therefore, $|C_\omega| = \sum_{(P, M) \in \mathcal{R}_\omega} 2^{c_p}$. Since S_ω is the solution set of tuples $(P, M) \in \mathcal{R}_\omega$ such that each connected component of $G[P]$ contains at least one marked vertex from M , a specific tuple $(P, M) \in S_\omega$ corresponds to those tuples of C_ω where $c_p = 0$, or equivalently $V(P_2) = \emptyset$. Therefore, $|S_\omega| \equiv |C_\omega| \pmod{2}$. $\square$

The type of tree decomposition we use is the same as described in [31]. We call this type of tree decomposition an *advanced nice tree decomposition* where along with the properties of a nice tree decomposition, an additional type of node is used. These nodes are called *introduce edge nodes*.

Definition 14. An *introduce edge node* x has exactly one child y such that $X_x = X_y$. The node is labeled with an edge $uv \in E$ with $u, v \in X_x = X_y$.

These nodes are present for every edge $uv \in E(G)$. Let $\mathcal{T}$ be an advanced nice tree decomposition, and x be an introduce edge node for an edge $uv \in E(G)$ with a child bag y . Then $u, v \in X_x$ and $X_x = X_y$, indicating that the edge uv is introduced in the bag of x . In a nice tree decomposition, when a vertex is introduced in an introduce node, all the edges that are incident to that vertex are also considered in the computation when we process the introduce node. However, for an advanced nice tree decomposition, there are separate bags (introduce vertex nodes and introduce edge nodes) that introduce a vertex and the edges incident to it separately. Therefore, when we process an introduce vertex node, the vertex that has been introduced in the corresponding bag remains isolated, and in later stages, with the help of introduce edge nodes, all the edges incident to that vertex are processed eventually, one at a time. The other node types in an advanced nice tree decomposition are the same as in Definition 7. It is noted in [31] that, given a tree decomposition, an advanced nice tree decomposition of the same width can be computed in polynomial time.

Lemma 15. Given a graph $G = (V, E)$, an advanced nice tree decomposition $\mathcal{T}$ of G , a weight function $w \rightarrow \{1, 2, \dots, N\}$, and an integer k , there exists an algorithm that computes $|C_\omega| \pmod{2}$ for all $0 \leq \omega \leq (k + |V|)N$ in $|V|^{O(1)}N^225^t$ time.

Proof. Let X_x be the set of vertices in bag x . Let G_x be a subgraph of G induced by vertex set V_x , where V_x is the union of all the bags in the subtree of $\mathcal{T}$ rooted at a bag x (including the vertices present in x), and the edge set E_x be the set of edges of G with both endpoints in V_x . The partial solutions of the problem are defined below. For each bag $x \in \mathcal{T}$, every integer $1 \leq i \leq k$, and for all weight values $0 \leq \omega \leq (k + |V|)N$, we define $\mathcal{A}_x(i, \omega, s)$ as a collection of tuples $((P, M), (V(P_1), V(P_2)))$ with the following properties:

1. $P \subseteq E_x$, and $M \subseteq V(P_1) \setminus X_x$.
2. $(V(P_1), V(P_2))$ be the consistent cut of $G[P]$, where $G[P]$ is a subgraph of G with vertex set V_x and edge set P .
3. For every $v \in V_x$, $\deg_{G[P]}(v) \leq 2$.
4. P is a path partition of $G[P]$ (follow the definition of $\mathcal{R}_\omega$ for details).
5. $w(X) + w(M) = \omega$, and $|M| = i$.
6. $s : X_x \rightarrow \{0_1, 0_2, 1_1, 1_2, 2\}$ is a function defined for each vertex in x and that assumes values based on their degree in $G[P]$. The value $s(v)$ of a vertex $v \in x$ is of the form α_β , where α denotes the degree of v in $G[P]$ with values either 0 (singleton set, not included in a path) or 1 (endpoint of a path) or 2 (internal vertex of a path) and β denotes which side of the consistent cut v is present with values either 1 (when $v \in V(P_1)$) or 2 (when $v \in V(P_2)$).

Note that when a vertex is forgotten in the forget node, then it can be made a marked vertex by properly adjusting the value of i . Moreover, if $\deg_{G[P]}(v) = 2$ then v has been processed and is no longer required in the future since v is an internal vertex of a path. Therefore, it is not necessary to store which side of the cut v is in such a case. Since we consider singleton paths in our calculation, it is important to store which side of the cut a vertex v is in when $\deg_{G[P]}(v) = 0$.

Let r be the root node of $\mathcal{T}$. Since $|C_\omega| \pmod{2} = \mathcal{A}_r(i, \omega, \emptyset) \pmod{2}$, to compute the value of $|C_\omega| \pmod{2}$ for all ω it is enough to compute $\mathcal{A}_r(i, \omega, \emptyset)$ for all ω .

Introduce vertex node. Let v be the vertex introduced in the introduce vertex bag. Since none of the edges incident to v are processed at this stage, v remains a singleton path.

$$\mathcal{A}_x(i, \omega, s[v \rightarrow 0_j]) = \mathcal{A}_y(i, \omega, s) \quad \text{for } j \in \{0, 1\}.$$

Introduce edge node. We refer to [31, Theorem A.14] for this part. The only thing that is different for our problem is the states for the degree 0 vertices (either 0_1 or 0_2). Following the definition of $\text{subs}(\alpha)$ in [31] we can write $\text{subs}(0_1) = \text{subs}(0_2) = \emptyset$. The rest of the analysis is the same as for the introduce edge node in [31, Theorem A.14].

Forget Node. Let v be the vertex that has been forgotten in the forget node. Then v can be a singleton vertex, or an internal vertex of a path, or an endpoint of a path. Hence,

$$\mathcal{A}_x(i, \omega, s) = \sum_{\substack{\alpha \in \{0_1, 0_2, \\ 1_1, 1_2, 2\}}} \mathcal{A}_y(i, \omega, s[v \rightarrow \alpha]).$$

Join node. We again refer to [31, Theorem A.14] for this part. Similar to the introduce edge node, the two states of a degree 0 vertex are handled separately.

A standard dynamic programming to compute all the values of $\mathcal{A}_x(i, \omega, s)$ takes time $|V|^{O(1)}N^225^t$, where the number $25 = 5^2$ comes from the calculation of the join node. (Note that one could use fast subset convolution like in [31] to reduce the constant value 25, which we do not do here.) $\square$

Theorem 16. There exists a Monte-Carlo algorithm that, given a graph G and a tree decomposition of width t of G , solves PATH PARTITION in $25^t n^{O(1)}$ time for graphs of order n . The algorithm does not give any false positives and gives false negatives with probability at most $1/2$.

Proof. Combining Algorithm 1 (by setting $U = E \cup V$), Corollary 3.1 of [31], and Lemma 15, we get the algorithm to solve PATH PARTITION. The correctness of the algorithm and the running time follow from Lemmas 13 and 15, respectively. $\square$

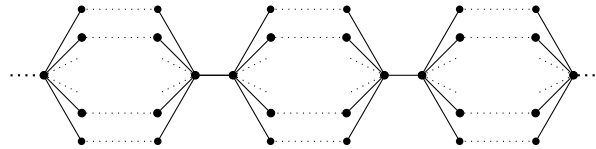


Fig. 2. A family of graphs of arbitrarily large order n and treewidth 2, with $\Theta(\sqrt{n})$ cut-vertices, and 2-connected components each of order $\Theta(\sqrt{n})$. An optimal solution to PATH COVER consists of $\Theta(\sqrt{n})$ solution paths, each of length $\Theta(\sqrt{n})$, all going from the left to the right side of the graph. Thus, there is an unbounded number of solution paths going through every bag of any tree decomposition of bounded width.

5. Conclusion

We have re-initiated the study of PATH COVER, which surprisingly is not extensively studied. We settled its complexity for trees by giving a linear-time algorithm on this class, and we gave an explicit $n^{O(t)}$ XP-time dynamic programming scheme for graphs of treewidth t , or $\kappa^{t^{O(t)}}$ if we have an upper bound on the solution size κ . An algorithm for PATH PARTITION using the Cut&Count technique gives a $2^{O(t)}n$ FPT running time. These running times also hold for the variants of PATH COVER and PATH PARTITION where the solution paths are required to be induced and/or edge-disjoint.

We note that in a recent paper, the first author, F. Foucaud, with two other colleagues, used the linear-time algorithm for PATH COVER on trees to obtain a linear-time algorithm for PATH COVER on outerplanar graphs [33]. It would be interesting to design polynomial-time algorithms for PATH COVER on other graph classes.

It would also be nice to improve the running times of our algorithm for PATH COVER. Can one get an improved $n^{O(t)}$ algorithm? Can it actually be solved in FPT time for parameter treewidth alone? It is possible that this is not the case, as the number of solution paths going through one bag of the tree decomposition can be arbitrarily large (see Fig. 2). In case of a negative answer, this would show a striking contrast between the algorithmic complexities of PATH COVER and PATH PARTITION.

CRedit authorship contribution statement

Florent Foucaud: Writing – review & editing, Writing – original draft, Methodology, Investigation, Funding acquisition, Conceptualization; **Atrayee Majumder:** Writing – review & editing, Writing – original draft, Investigation, Conceptualization; **Tobias Mömke:** Supervision, Methodology, Investigation, Funding acquisition, Conceptualization; **Aida Roshany-Tabrizi:** Writing – review & editing, Writing – original draft, Methodology, Investigation, Conceptualization.

Declaration of competing interest

The authors declare the following financial interests/personal relationships which may be considered as potential competing interests:

Florent Foucaud reports financial support was provided by French National Research Agency and is a member of the editorial board of the journal since January 2026. Tobias Moemke reports financial support was provided by German Research Foundation. If there are other authors, they declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

References

- [1] F. Foucaud, A. Majumder, T. Mömke, A. Roshany-Tabrizi, Polynomial-time algorithms for path cover on trees and graphs of bounded treewidth, in: D.R. Gaur, R. Mathew (Eds.), Algorithms and Discrete Applied Mathematics - 11th International Conference, CALDAM 2025, Coimbatore, India, February 13–15, 2025, Proceedings, Volume 15536 of Lecture Notes in Computer Science, Springer, 2025, pp. 147–159. https://doi.org/10.1007/978-3-031-83438-7_13
- [2] T.H. Cormen, C.E. Leiserson, R.L. Rivest, C. Stein, Introduction to Algorithms, The MIT Press, 3rd edition, 2009.
- [3] N. Robertson, P.D. Seymour, Graph minors. XIII. The disjoint paths problem, J. Comb. Theory B 63 (1) (1995) 65–110.
- [4] M. Cygan, F.V. Fomin, L. Kowalik, D. Lokshtanov, D. Marx, M. Pilipczuk, M. Pilipczuk, S. Saurabh, Parameterized Algorithms, Springer Cham, 2015.
- [5] T. Akiyama, T. Nishizeki, N. Saito, NP-completeness of the Hamiltonian cycle problem for bipartite graphs, J. Inf. Process. 3 (2) (1980/81) 73–76.
- [6] F.V. Fomin, P.A. Golovach, D. Lokshtanov, S. Saurabh, Intractability of clique-width parameterizations, SIAM J. Comput. 39 (5) (2010) 1941–1956. <https://doi.org/10.1137/080742270>
- [7] D.G. Corneil, B. Dalton, M. Habib, LDFS-based certifying algorithm for the minimum path cover problem on cocomparability graphs, SIAM J. Comput. 42 (3) (2013) 792–807.
- [8] R. Hung, M. Chang, Finding a minimum path cover of a distance-hereditary graph in polynomial time, Discret. Appl. Math. 155 (17) (2007) 2242–2256.
- [9] D.S. Franzblau, A. Raychaudhuri, Optimal Hamiltonian completions and path covers for trees, and a reduction to maximum flow, ANZIAM J. 44 (2) (2002) 193–204. <https://doi.org/10.1017/S1446181100013894>
- [10] S. Goodman, S. Hedetniemi, On the hamiltonian completion problem, in: R.A. Bari, F. Harary (Eds.), Graphs and Combinatorics, Springer Berlin Heidelberg, 1974, pp. 262–272.
- [11] S. Kundu, A linear algorithm for the hamiltonian completion number of a tree, Inf. Process. Lett. 5 (2) (1976) 55–57. [https://doi.org/10.1016/0020-0190\(76\)90080-6](https://doi.org/10.1016/0020-0190(76)90080-6)
- [12] P. Manuel, Revisiting path-type covering and partitioning problems, (2018). arXiv:1807.10613
- [13] B.A. Baker, The k-Fixed-Endpoint Path Partition Problem, PhD thesis, Lehigh University, Pennsylvania, USA, 2013. Available online at <http://preserve.lehigh.edu/etd/1420/>.
- [14] S.F. Florkowski, Extensions of the Hamiltonian Cycle/Path Problems in Special Families of Graphs, Ph.D. thesis, Lehigh University, 2017. Available online at <https://preserve.lehigh.edu/lehigh-scholarship/graduate-publications-theses-dissertations/theses-dissertations/extensions>.

- [15] F.T. Boesch, S. Chen, J.A.M. McHugh, On covering the points of a graph with point disjoint paths, in: *Graphs and Combinatorics*, Springer, 1974, pp. 201–212.
- [16] R. Lin, S. Olariu, G. Pruesse, An optimal path cover algorithm for cographs, *Comput. Math. Appl.* 30 (8) (1995) 75–83. [https://doi.org/10.1016/0898-1221\(95\)00139-P](https://doi.org/10.1016/0898-1221(95)00139-P)
- [17] J. Pan, G.J. Chang, Path partition for graphs with special blocks, *Discret. Appl. Math.* 145 (3) (2005) 429–436.
- [18] M. Cygan, J. Nederlof, M. Pilipczuk, M. Pilipczuk, J.M.M. van Rooij, J.O. Wojtaszczyk, Solving connectivity problems parameterized by treewidth in single exponential time, *ACM Trans. Algorithms* 18 (2) (2022) 17:1–17:31.
- [19] M. Cáceres, M. Cairo, B. Mumey, R. Rizzi, A.I. Tomescu, Sparsifying, shrinking and splicing for minimum path cover in parameterized linear time, in: *Proceedings of the 2022 ACM-SIAM Symposium on Discrete Algorithms, SODA 2022, Virtual Conference / Alexandria, VA, USA, January 9–12, 2022*, SIAM, 2022, pp. 359–376.
- [20] M. Cáceres, B. Mumey, E. Husić, R. Rizzi, M. Cairo, K. Sahlin, A.I. Tomescu, Safety in multi-assembly via paths appearing in all path covers of a DAG, *IEEE/ACM Trans. Comput. Biol. Bioinform.* 19 (6) (2022) 3673–3684. <https://doi.org/10.1109/TCBB.2021.3131203>
- [21] S.C. Ntafos, S.L. Hakimi, On path cover problems in digraphs and applications to program testing, *IEEE Trans. Softw. Eng.* SE-5 (5) (1979) 520–529. <https://doi.org/10.1109/TSE.1979.234213>
- [22] R. Rizzi, A.I. Tomescu, V. Mäkinen, On the complexity of minimum path cover with subpath constraints for multi-assembly, *BMC Bioinform.* 15 (S-9) (2014) S5.
- [23] G. Andreatta, F. Mason, Path covering problems and testing of printed circuits, *Discrete Appl. Math.* 62 (1-3) (1995) 5–13.
- [24] F. Harary, A. Schwenk, Evolution of the path number of a graph: covering and packing in graphs, II, in: *Graph Theory and Computing*, 1972, pp. 39–45.
- [25] G. Lin, Z. Cai, D. Lin, Vertex covering by paths on trees with its applications in machine translation, *Inf. Process. Lett.* 97 (2) (2006) 73–81. <https://doi.org/10.1016/j.ipl.2005.09.006>
- [26] D. Chakraborty, O. Defrain, F. Foucaud, M. Mari, P. Tale, Parameterized complexity of isometric path partition: treewidth and diameter, in: J. Goedgebeur, P. Rzázewski (Eds.), *52nd International Workshop on Graph-Theoretic Concepts in Computer Science, WG 2026, Kortrijk, Belgium, June 2–4, 2026*, Volume 376 of LIPIcs, Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2026, pp. 11:1–11:18. <https://doi.org/10.4230/LIPICS.WG.2026.11>
- [27] H. Fernau, F. Foucaud, K. Mann, U. Padariya, K.N. Rajath Rao, Parameterizing path partitions, *Theor. Comput. Sci.* 1028 (2025) 115029. <https://doi.org/10.1016/J.TCS.2024.115029>
- [28] T. Kloks, *Treewidth: Computations and Approximations*, Springer, 1994.
- [29] T. Korhonen, A single-exponential time 2-approximation algorithm for treewidth, in: *62nd IEEE Annual Symposium on Foundations of Computer Science, FOCS 2021, Denver, CO, USA, February 7–10, 2022*, IEEE, 2021, pp. 184–192.
- [30] H.L. Bodlaender, P. Bonsma, D. Lokshtanov, The fine details of fast dynamic programming over tree decompositions, in: *Parameterized and Exact Computation: 8th International Symposium, IPEC 2013, Sophia Antipolis, France, September 4–6, 2013, Revised Selected Papers 8*, Springer, 2013, pp. 41–53.
- [31] M. Cygan, J. Nederlof, M. Pilipczuk, M. Pilipczuk, J.M.M. Van Rooij, J.O. Wojtaszczyk, Solving connectivity problems parameterized by treewidth in single exponential time, *ACM Trans. Algorithms* 18 (2) (2022). <https://doi.org/10.1145/3506707>
- [32] K. Mulmuley, U.V. Vazirani, V.V. Vazirani, Matching is as easy as matrix inversion, in: *Proceedings of the Nineteenth Annual ACM Symposium on Theory of Computing*, 1987, pp. 345–354.
- [33] M. Dutta, F. Foucaud, S.C. Nandy, Algorithms and bounds for path covers of tree-structured graphs, in: *Proceedings of the 32nd International Computing and Combinatorics Conference (COCOON 2026)*, Volume 16835 of *Lecture Notes in Computer Science*, 2026, pp. 231–244.