

Polynomial-Time Algorithms for Path Cover on Trees and Graphs of Bounded Treewidth

Florent Foucaud[1], Atrayee Majumder[2], Tobias Mömke[2], and Aida Roshany-Tabrizi[2(✉)]

[1] CNRS, Clermont Auvergne INP, Mines Saint-Étienne, LIMOS, 63000 Clermont-Ferrand, France
[2] University of Augsburg, Augsburg, Germany
aida.roshanytabrizi@uni-a.de

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. Let t be the treewidth of a given graph. We then show that Path Cover can be solved in polynomial time on graphs of bounded treewidth, in XP time $n^{t^{O(t)}}$, using a dynamic programming scheme. Our algorithm gives an FPT $2^{O(t \log t)}n$ algorithm for Path Partition as a corollary. These results also apply to the variants where the paths are required to be induced (i.e. chordless) and/or edge-disjoint.

Keywords: Path Cover · Trees · Treewidth

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 from which efficient algorithms were obtained [6, Chapter 24]. Finding *disjoint paths* is also a problem of utmost importance, both in algorithmic and structural graph theory [25]. 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

F. Foucaud—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)
A. Majumder—Supported by DFG Grant 439637648 (Sachbeihilfe)
T. Mömke—Partially supported by DFG Grant 439522729 (Heisenberg-Grant) and DFG Grant 439637648 (Sachbeihilfe).

© The Author(s), under exclusive license to Springer Nature Switzerland AG 2025
D. Gaur and R. Mathew (Eds.): CALDAM 2025, LNCS 15536, pp. 147–159, 2025.
https://doi.org/10.1007/978-3-031-83438-7_13

generalizations, Path Cover and Path Partition, which are about covering the vertices of a graph using a minimum number of paths (unrestricted and pairwise vertex-disjoint, respectively). They are formally defined as follows.

Path Cover
Input: A graph G.
Problem: 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.
Problem: 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 at least 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 [8] 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 [1].

Unlike us, most work in the literature refers to Path Partition as "Path Cover"; indeed, the former is much more studied than the latter: see [7,15] for such works on Path Partition. In some cases, Path Partition is also called Hamiltonian Completion [12,13,18]. 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 [21] on path-type problems.

Several papers from the 1970 s studied Path Partition on trees [4,13,18]. However, they did not explicitly analyze the running times. A properly analyzed linear-time algorithm was given in 2002 [12]. Path Partition was also shown to be solvable in polynomial time on many other graph classes such as cographs [20], distance-hereditary graphs [15], cocomparability graphs [7] (which contain all interval graphs), or block graphs/cactii [23]. 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 [9].

Both Path Cover and Path Partition have numerous applications, in particular in program testing [22], circuit testing [2], or machine translation [19], 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 [5,24]. We refer to [5,10,22,24] for the few references about Path Cover that we are aware of.

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 Sect. 2) on Path Cover on trees, for which no efficient 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 [14] 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 Sect. 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. With a slight simplification, the same algorithm also solves Path Partition and runs in improved FPT running time $2^{O(t \log t)}n$.

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 [11,21] and references therein).

We finally conclude in Sect. 4.

2 Path Cover on Trees

We first study Path Cover on trees. In [14, 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 of a tree. 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 [14], 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 of a tree 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. [6, 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 $p = (x_1, x_2, \ldots, x_k)$ as a notation for paths where x_i are all distinct vertices of T connected by the path p. The vertices $x_2, \ldots, x_{k-1}$ are internal vertices; the vertices x_1 and x_k are endpoints of the path p.

Definition 1. *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 2. *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 $p_1 = (a_1, \ldots, a_j)$ and $p_2 = (b_1, \ldots, b_k)$ are two paths in $\mathcal{P} = \{p_1, p_2, \ldots\}$ 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 $\mathcal{P} \cdot v = \{(a_1, \ldots, a_j, v), (b_1, \ldots, b_k, v), \ldots\}$. 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 $p_1 = (a_1, \ldots, a_j)$ and $p_2 = (b_1, \ldots, b_k)$ 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, $comb(p_1, p_2) = (a_1, \ldots, a_j, v, b_k, \ldots, b_1)$.*

Let T_v be the subtree of T rooted at a given vertex v. Let $\{T_v^1, T_v^2, \ldots\}$ 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 $\{T_v^1, T_v^2, \ldots\}$. 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 $\{T_v^1, T_v^2, \ldots\}$ at v.

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

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

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

```
1: Mark v as visited
2: for Each child u of v do
3:     if u is unvisited then
4:         recursively call PC(T, u, S)
5:     end if
6: end for
7: if v is a leaf then
8:     P_v^open ⟵ {(v, v)}        and        P_v^close ⟵ ∅
9: else
10:    P_v ⟵ ⋃_c P_c^open          ▷ union over all the paths coming from all children c of v
11:    P_v^close ⟵ ∅
12:    while |P_v| > 2 do
13:        Find two unrelated paths p_i and p_j
14:        P_v^close ⟵ P_v^close ⋃ comb(p_i, p_j)
15:        P_v ⟵ P_v \ {p_i, p_j}
16:    end while
17:    if v is the root of T and |P_v| = 2 then
18:        P_v^close ⟵ P_v^close ⋃ comb(p_1, p_2)
19:    else if v is the root of T and |P_v| = 1 then
20:        P_v^close ⟵ P_v^close ⋃ p_1 · v
21:    else
22:        P_v^open ⟵ P_v · v          ▷ when v is not the root of T and |P_v| ≤ 2
23:    end if
24: end if
25: Mark v as covered
26: S ⟵ S ⋃ P_v^close
27: return P_v^open
```

– $\mathcal{P}_v$ is the set of all paths extended as open paths from the children of v.

Among these sets, we need to store the set $\mathcal{P}_v^{open}$ 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 $p = (a_1, a_2, \ldots, a_j)$ in terms of its endpoints i.e. $p = (a_1, a_j)$.

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, $comb(p,q)$ 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. □

3 Path Cover on Graphs with 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 2. Path Cover *can be solved in time* $n^{t^{O(t)}}$*, where* n *is the number of vertices, and* t *is the treewidth of the input graph.*

See [3,8,16] for basic definitions of tree decompositions and treewidth. 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 3 ([16]). *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* $X_v = X_{u_1} = X_{u_2}$.

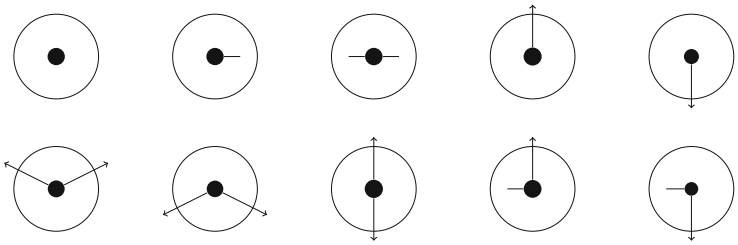


Fig. 1. Illustration of the different types of a vertex with respect to some node v and partial path. The set of types is $\mathcal{N} = \{\emptyset, \{-\}, \{-,-\}, \{\uparrow\}, \{\downarrow\}, \{\uparrow,\uparrow\}, \{\downarrow,\downarrow\}, \{\downarrow,\uparrow\}, \{\uparrow,-\}, \{\downarrow,-\}\}$.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

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 $2^{O(t)}n$ [17]. 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 $O(t \cdot n)$ [3].

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 [8].

Lemma 1. *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 $X_u \cap X_v$.*

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 $\varphi_p : \Pi_p \to \mathcal{N}$ 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.

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 $\varphi_p : \Pi_p \to \mathcal{N}$ 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 $f(p) \in \{1,\ldots,n\}$ 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 $f : P \to \{1,\ldots,n\}$, 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.

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 $f : P \to \{1, \ldots, n\}$ 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 $[v, P, f]$ *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 $opt[v, P, f]$ 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 $opt[v, P, f] = \infty$.

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-sates; 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 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. (We omit the case of forget nodes and join nodes due to space constraints.)

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

Introduce Node. Let v be an introduce node with child node u such that $X_v = X_u \dot\cup \{x\}$ 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 $opt[v, P, f] = \infty$. Indeed, by Lemma 1, 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 $P_{\overline{x}}$ 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 $P_{\overline{x}}$ 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:

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

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 $opt[u, P', f']$, where $[u, P', f'] \in C$ is such that $opt[u, P', f']$ 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 $opt[v, P, f] \leq |S| \leq |S'| + k = \min_{[u,P',f'] \in C}\{opt[u, P', f']\} + k$.

Conversely, note that an optimal partial solution S of size $opt[v, P, f]$ 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 $\min_{[u,P',f'] \in C}\{opt[u, P', f']\} \leq |S'| \leq opt[v, P, f] - k$ and thus, $opt[v, P, f] \geq \min_{[u,P',f'] \in C}\{opt[u, P', f']\} + k$.

Proof (Proof of Theorem 2). The algorithm goes through the tree 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 follows from the above discussion and the correctness of the inductive formulas. The optimal value of a solution is found 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 n using the function f. There are at most $n^{(10t)!}$ possible functions f, thus, the total number of partial solutions inside each node is at most $n^{t^{O(t)}}$, 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 k^2 time (if there are at most k 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 k^3 time. In each case, we have $k = n^{t^{O(t)}}$ and thus this is still $n^{t^{O(t)}}$. We have at most $4n$ nodes in the tree decomposition; hence, the algorithm solves Path Cover in time $n^{t^{O(t)}}$. □

The Case of Path Partition. In the Path Partition version of the problem, we are dealing with vertex-disjoint paths. Therefore, the number of paths that pass through each bag of the tree decomposition is at most the size of the bag. Hence, a DP-state simply needs a collection of partial paths that forms a partition of the bag into ordered sets, and thus, there are at most $t^{O(t)} = 2^{O(t \cdot \log t)}$ 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 $2^{O(t \cdot \log t)} n$.

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, it suffices to check that the partial paths in every considered DP-state are edge-disjoint; otherwise the algorithm is the same.

4 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 $n^{t^{O(t)}}$ XP-time dynamic programming scheme for graphs of treewidth t. The same algorithm modified for Path Partition gives a $2^{O(t \log t)} n$ 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.

It would be nice to improve the running times of our algorithms. Probably, one can design a $2^{O(t)} n$ algorithm for Path Partition using the ideas of [9], where they solved Cycle Partition in this running time. But can one get an $n^{O(t)}$ algorithm for Path Cover? Can Path Cover even be solved in FPT time for parameter treewidth? 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. In case of a negative answer, this would show a striking contrast between the algorithmic complexities of Path Cover and Path Partition.

References

1. Akiyama, T., Nishizeki, T., Saito, N.: NP-completeness of the Hamiltonian cycle problem for bipartite graphs. J. Inform. Process. **3**(2), 73–76 (11980/81)
2. Andreatta, G., Mason, F.: Path covering problems and testing of printed circuits. Discret. Appl. Math. **62**(1–3), 5–13 (1995)
3. Bodlaender, H.L., Bonsma, P., Lokshtanov, D.: The fine details of fast dynamic programming over tree decompositions. In: Gutin, G., Szeider, S. (eds.) IPEC 2013. LNCS, vol. 8246, pp. 41–53. Springer, Cham (2013). https://doi.org/10.1007/978-3-319-03898-8_5
4. Boesch, F., Chen, S., McHugh, J.: On covering the points of a graph with point disjoint paths. In: Graphs and Combinatorics, pp. 201–212. Springer (1974)
5. Cáceres, M., Cairo, M., Mumey, B., Rizzi, R., Tomescu, A.I.: 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, pp. 359–376 (2022)
6. Cormen, T.H., Leiserson, C.E., Rivest, R.L., Stein, C.: Introduction to Algorithms. The MIT Press, 3rd edition (2009)
7. Corneil, D.G., Dalton, B., Habib, M.: LDFS-based certifying algorithm for the minimum path cover problem on cocomparability graphs. SIAM J. Comput. **42**(3), 792–807 (2013)
8. Cygan, M., et al.: Parameterized Algorithms. Springer, Cham (2015)
9. Cygan, M., et al.: Solving connectivity problems parameterized by treewidth in single exponential time. ACM Trans. Algorithms **18**(2), 17:1–17:31 (2022)
10. Cáceres, M., et al.: Safety in multi-assembly via paths appearing in all path covers of a dag. IEEE/ACM Trans. Comput. Biol. Bioinf. **19**(6), 3673–3684 (2022)
11. Fernau, H., Foucaud, F., Mann, K., Padariya, U., Rao, K.N.R.: Parameterizing path partitions. In: Mavronicolas, M. (ed.) CIAC 2023. LNCS, vol. 13898, pp. 187–201. Springer, Cham (2023). https://doi.org/10.1007/978-3-031-30448-4_14
12. Franzblau, D.S., Raychaudhuri, A.: Optimal hamiltonian completions and path covers for trees, and a reduction to maximum flow. ANZIAM J. **44**(2), 193–204 (2002)
13. Goodman, S., Hedetniemi, S.: On the hamiltonian completion problem. In: Bari, R.A., Harary, F. (eds.) Graphs and Combinatorics, pp. 262–272. Springer, Berlin Heidelberg (1974)
14. Harary, F., Schwenk, A.: Evolution of the path number of a graph: covering and packing in graphs, II. Graph Theory and Computing, pp. 39–45 (1972)
15. Hung, R., Chang, M.: Finding a minimum path cover of a distance-hereditary graph in polynomial time. Discret. Appl. Math. **155**(17), 2242–2256 (2007)
16. Kloks, T.: Treewidth: computations and approximations. Springer (1994)
17. Korhonen, T.: A single-exponential time 2-approximation algorithm for treewidth. In: 62nd IEEE Annual Symposium on Foundations of Computer Science, FOCS 2021, pp. 184–192 (2021)
18. Kundu, S.: A linear algorithm for the hamiltonian completion number of a tree. Inf. Process. Lett. **5**(2), 55–57 (1976)
19. Lin, G., Cai, Z., Lin, D.: Vertex covering by paths on trees with its applications in machine translation. Inf. Process. Lett. **97**(2), 73–81 (2006)
20. Lin, R., Olariu, S., Pruesse, G.: An optimal path cover algorithm for cographs. Comput. Math. Appl. **30**(8), 75–83 (1995)

21. Manuel, P.: Revisiting path-type covering and partitioning problems. arXiv preprint arXiv:1807.10613 (2018)
22. Ntafos, S., Hakimi, S.: On path cover problems in digraphs and applications to program testing. IEEE Trans. Softw. Eng. **SE-5**(5), 520–529 (1979)
23. Pan, J., Chang, G.J.: Path partition for graphs with special blocks. Discret. Appl. Math. **145**(3), 429–436 (2005)
24. Rizzi, R., Tomescu, A.I., Mäkinen, V.: On the complexity of minimum path cover with subpath constraints for multi-assembly. BMC Bioinform. **15**(S-9), S5 (2014)
25. Robertson, N., Seymour, P.D.: Graph minors. XIII. The disjoint paths problem. J. Combinatorial Theor. Ser. B **63**(1), 65–110 (1995)