The Fine Details of Fast Dynamic Programming over Tree Decompositions

Hans L. Bodlaender[1], Paul Bonsma[2], and Daniel Lokshtanov[3]

[1] Institute of Information and Computing Sciences, Utrecht University, PO Box 80.089, 3508 TB Utrecht, the Netherlands
[2] Faculty of EEMCS, University of Twente, the Netherlands, PO Box 217, 7500 AE Enschede, the Netherlands p.s.bonsma@ewi.utwente.nl (corresponding author)
[3] Department of Informatics, University of Bergen, PO Box 7803, 5020 Bergen, Norway

Abstract. We study implementation details for dynamic programming over tree decompositions. Firstly, a fact that is overlooked in many papers and books on this subject is that it is not clear how to test adjacency between two vertices in time bounded by a function of k, where k is the width of the given tree decomposition. This is necessary to obtain linear time dynamic programming algorithms. We address this by giving a simple $O(kn)$ time and space preprocessing procedure that enables adjacency testing in time $O(k)$, where n is the number of vertices of the graph.
Secondly, we show that a large class of NP-hard problems can be solved in time $O(q^{k+1}n)$, where q^{k+1} is the natural size of the dynamic programming tables. The key improvement is that we avoid a polynomial factor in k. This holds for all problems that can be formulated as a Min Weight Homomorphism problem: given a (large) graph G on n vertices and a (small) graph H on q vertices, with integer vertex and edge weights, is there a homomorphism from G to H with total (vertex and edge image) weight at most M? This result implies e.g. $O(2^k n)$ algorithms for Max Independent Set and Max Cut, and a $O(q^{k+1}n)$ algorithm for q-Colorability. The table building techniques we develop are also useful for many other problems.

1 Introduction

Dynamic programming over tree decompositions has become an important algorithmic technique, that is used as a central subroutine in many parameterized, exact and approximation algorithms for NP-hard problems. The key property that is often used is that for any constant k, many NP-hard graph problems can be solved in linear time, if a tree decomposition of the graph of width at most k is provided. Examples of early results of this kind are [3, 4, 7, 9, 19]. Good introductions can be found in [15, 6, 8]. Recent breakthrough results appear in [16, 10, 5].

In this paper, we study implementation details of dynamic programming over tree decompositions. Throughout, denote by n the number of vertices of the input graph, and by k the width of the given tree decomposition. (k is viewed as

a parameter, not a constant.) Firstly, a fact that has been overlooked in many papers and books on this subject, is that it is not clear how to test adjacency between two vertices in time bounded by any function of k. This is necessary to obtain linear time algorithms. Note that we cannot simply assume that an adjacency matrix is given: graphs of bounded treewidth are sparse (they have fewer than kn edges), and therefore an n^2 bit adjacency matrix cannot be constructed in linear time from a typical $O(kn \log n)$ bit input encoding, e.g. based on adjacency lists. (Note that for all cases where this method is relevant, k is much smaller than n.) We remark that if using quadratic space is allowed, then there is an easy linear time preprocessing procedure that enables constant time adjacency testing, using a lazy (i.e. uninitialized) adjacency matrix [1, Exercise 2.12]. However, there seems to be no straightforward way of obtaining linear time and space dynamic programming algorithm. In Section 3 we discuss these claims in more detail, and also present our first result: a simple $O(kn)$ time and space preprocessing procedure that, given a graph on n vertices and a tree decomposition on $O(n)$ nodes of width k, enables adjacency testing in time $O(k)$. This is not very deep, but also not obvious. Since it solves a gap in the existing literature, we feel it should be published.

There are more examples of seemingly trivial problems on sparse graphs where the fact that adjacency testing cannot be done in constant time is surprisingly problematic. For instance, it is well-known that a graph is a series parallel graph if and only if it can be reduced to a K_2 by iteratively suppressing vertices of degree two, and replacing multi-edges by single edges. Assuming that adjacency testing can be done in constant time, this characterization would easily yield a linear time algorithm for recognizing series parallel graphs. Nevertheless, without this assumption, a significantly more sophisticated algorithm is needed; see [18, Section 3.3].

Secondly, we consider the dependency of the complexity on the parameter k. For many NP-hard problems, $O(\text{poly}(k)c^k n)$ time algorithms are known, for some constant c ($\text{poly}(k)$ denotes a polynomial factor in k). For problems where solutions can be characterized using local properties, this has been known for a long time, see e.g. [17]. In recent breakthrough results, such a complexity has also been obtained for problems with global connectivity constraints [5, 10]. However, we study the simpler local problems here. For various of these problems, $O(\text{poly}(k)c^k n)$ algorithms are known, where c^k is the natural size of the dynamic programming tables. For various problems such as Max Independent Set, Max Cut and q-Colorability (problems without complex join operations), such a complexity is relatively easy to prove (see e.g. [6, 15]). For other problems, such as in particular Min Dominating Set, this is significantly harder, but a $(3^k k^2 n)$ complexity has been achieved using the fast subset convolution technique [16], improving on the previous $O(4^k n)$ algorithm [2].

Assuming the Strong Exponential Time Hypothesis [12], it has been shown in [14] that the constant c in the exponential factor c^k cannot be improved for the aforementioned problems. Therefore, we study the question whether the $\text{poly}(k)$ factor can be removed. For problems addressed by the fast subset convolution

technique, this seems impossible. However, for a large class of other problems this can be done: in Section 4 we consider the Minimum Weight Homomorphism (MWH) problem: given a graph G on n vertices, and a graph H on q vertices with integer vertex and edge weights, find a minimum weight homomorphism from G to H, or decide that none exists. This weight is the sum of the vertex and edge image weights. The graphs G and H may be directed and may have loops. MWH generalizes well-studied problems such as Max Independent Set, Max Cut and q-Colorability, and many others. We give a $O(q^{k+1}n)$ dynamic programming algorithm for MWH in Section 5.

Removing the poly(k) factor requires precise treatment of various dynamic programming details that can usually be ignored, such as bag and table ordering, fast table building, and enabling constant time adjacency checking. One of the few papers that also discusses some of these in detail is the paper by Alber and Niedermeier [2], presenting an $O(4^k n)$ algorithm for Min Dominating Set (see also [15]). To be precise, only the join operation requires time $O(4^k)$ in [2]. For the case of path decompositions, where no join operation is required, they give an $O(k3^k n)$ algorithm. To illustrate that our techniques can be applied to a wider variety of problems, in the full version of this paper we will show that the $O(k)$ factor can be removed in this result. More precisely, using our techniques an $O(3^k n)$ algorithm for Min Dominating Set can be constructed, when a path decomposition of width k is given. In [6], another algorithm without poly(k) factor is presented: a $O(2^k n)$ time algorithm for Max Independent Set is sketched, although various details are omitted. This inspired the current study.

We remark that from a purely theoretical asymptotic analysis viewpoint, removing poly(k) factors seems irrelevant. Indeed, abusing the O-notation, for any $\epsilon > 0$ one may for instance write $O(k^2 3^k n) \subseteq O((3+\epsilon)^k n)$. Nevertheless, we note that e.g. $k^2 3^k < 4^k$ only holds when $k \geq 22$. Since at this point, dynamic programming tables cannot be stored in a normal computer memory anymore, we conclude that the previous $O(4^k n)$ MDS algorithm by Alber and Niedermeier [2] is still the most efficient one in practice! (Compared to [16].) Similarly, the complexity improvements we present are important in practice. We start in Section 2 with basic definitions. Proofs omitted for space constraints are marked with a star, and can be found in the appendix.

2 Preliminaries

For basic graph theory notations, see [11]. By uv and (u, v) we denote undirected and directed edges, respectively. By $N(u)$, $N^+(u)$ and $N^-(u)$ we denote the (undirected) neighborhood, out-neighborhood and in-neighborhood of u, respectively.

Definition 1 *A* tree decomposition *of a (di)graph G is a 2-tuple (T, X) where T is a tree and $X = \{X_v : v \in V(T)\}$ is a set of subsets X_v of $V(G)$ such that the following properties hold:*

1. *For every $xy \in E(G)$ (resp. $(x, y) \in E(G)$), there is a $v \in V(T)$ with $\{x, y\} \subseteq X_v$.*

2. *For every $x \in V(G)$, the subgraph of T induced by $X^{-1}(x) = \{v \in V(T) : x \in X_v\}$ is non-empty and connected.*

To distinguish between vertices of G and T, the latter are called *nodes*. For $v \in V(T)$, the set X_v is also called the *bag* of v. The *width* of a tree decomposition is $\max_{v \in V(T)} |X_v| - 1$. The *treewidth* $\mathrm{tw}(G)$ of a graph G is the minimum width over all tree decompositions of G. A *rooted tree decomposition* $(T, X), r$ of G is obtained by additionally choosing a root $r \in V(T)$, which defines a child/parent relation between every pair of adjacent nodes, and ancestors/descendants in the usual way. A node without children is called a *leaf*.

Definition 2 *A rooted tree decomposition $(T, X), r$ of G is* nice *if every node $u \in V(T)$ is of one of the following types:*

Leaf: *u has no children.*
Forget: *u has one child v with $X_u \subset X_v$ and $|X_u| = |X_v| - 1$.*
Introduce: *u has one child v with $X_v \subset X_u$ and $|X_u| = |X_v| + 1$.*
Join: *u has two children v and w with $X_u = X_v = X_w$.*

The tree decomposition is called very nice *if in addition, for every leaf node u it holds that $|X_u| = 1$.*

If u is an introduce node with child v and $X_u \setminus X_v = \{x\}$, then we say x is *introduced* in u. The following fact is well-known and easy to prove: if G is a graph on n vertices with tree width k, then G has at most kn edges. For dynamic programming *(DP)* over tree decompositions, the following definitions are important. Let $(T, X), r$ be a rooted tree decomposition of G. For a node $u \in V(T)$, we denote $X(u) = \cup_v X_v$, where the union is taken over all descendants v of u, including u itself. The subgraph $G(u)$ is then defined as $G(u) = G[X(u)]$. DP algorithms rely on the following two key properties, which follow easily from Definition 1: firstly, $G(r) = G$. Secondly, for every $u \in V(T)$, the only vertices of $G(u)$ that (in G) may be incident with edges that are not in $G(u)$ are vertices in X_u.

Computation model and Assumptions We use the standard computation model (i.e. the RAM model, see e.g. [1]). The memory consists of an unbounded number of registers r_i, $i \in \mathbb{N}$, which can hold integer values. In constant time, we can read or write any r_i, execute an elementary program control instruction, or carry out a basic arithmetic operation. As basic arithmetic operations, we only require addition, subtraction, multiplication, and testing whether an integer is positive.

For convenience, we assume that for all graphs G we consider, $V(G) = \{1, \ldots, n\}$. The following standard (but usually implicit) assumption is important when discussing linear time (space) algorithms: if for an algorithmic result, no input encoding is specified, the result should hold for all "reasonable input encodings". For the case of a graph G with n vertices and m edges, and $\mathrm{tw}(G) = k$, this implies in particular that for designing linear time algorithms, we cannot assume that an adjacency matrix is given. This is because an n^2 bit adjacency

matrix cannot be constructed in linear time (which is $O(m) \subseteq O(kn)$) from adjacency lists, or edge lists. On the other hand, an adjacency list representation of G can be constructed in linear time from all other reasonable representations, so we may assume that an adjacency list encoding of G is given. Similarly, if a tree decomposition (T, X) of G is given, we should assume that the bags X_u are given as unordered lists of vertices of G, and that no additional information is available (such as a list of edges of $G[X_u]$).

3 Enabling Fast Adjacency Testing

We first discuss the useful and simple quadratic space solution using *lazy adjacency matrices*. Note that we may not assume that all registers are initialized to zero at the beginning of the computation, so using an n^2 bit adjacency matrix requires (non-linear) initialization time $\Theta(n^2)$. However this can be avoided using the following known trick (see e.g. [1, Exercise 2.12]): Store a list of edge objects in a consecutive memory block of size $O(m) \subseteq O(kn)$. Reserve n^2 registers for the adjacency matrix, called $a_{i,j}$ for $i, j \in V(G)$, but do not initialize these. Instead, for every edge $e = \{i, j\}$, initialize $a_{i,j}$ and $a_{j,i}$ with a pointer to the register containing the edge object e. Observe that this now allows constant time adjacency checking.

Nevertheless, if we insist on using linear space, there is no obvious reason why known dynamic programming algorithms can be implemented in linear time. For instance, graphs of treewidth k may contain vertices for which the degree is not bounded by any function of k. Furthermore, it may be that in any low width nice tree decomposition, such high degree vertices are introduced many times. So if for every introduce node, one considers the entire neighbor list of the introduced vertex, this does not yield a linear time algorithm.

Let T be a tree with root r. For any set $S \subseteq V(T)$, define a node $v \in S$ to be a *top node* if the unique (r, v)-path contains no vertices from S other than v. We will need the following basic properties of top nodes.

Proposition 3 *Let T, r be a rooted tree, and $S \subseteq V(T)$ with $T[S]$ connected.*

(i) S contains exactly one top node; denote this node by $\top(S)$.
(ii) Let $u \in S$ with parent v. Then $u = \top(S)$ if and only if $v \notin S$.
(iii) For any $S' \subseteq V(T)$ with $T[S']$ connected and $S \cap S' \neq \emptyset$: $\top(S') \in S$ or $\top(S) \in S'$.

Proof: (i): Suppose to the contrary that S contains two top nodes u and v. Let P_u and P_v be the unique (r, u)-path and (r, v)-path in T, respectively. Let w be the last vertex on P_u that is also in P_v (i.e. the lowest common ancestor of u and v). Combining the subpath of P_u from u to w and the subpath of P_v from w to v yields a path P_{uv} from u to v with an internal vertex w, with $w \notin S$. But then P_{uv} is the unique path from u to v in T, so deleting w separates u from v, contradicting that $T[S]$ is connected.

(ii): Suppose to the contrary that the unique (r,u)-path contains another vertex w with $w \in S$. Then the unique path from u to w contains $v \notin S$, contradicting that S is connected.

(iii): Consider a top node u of $S \cap S'$. Then the parent v of u is not in $S \cap S'$. If $v \notin S$, then by (ii), $u = \top(S)$. Analogously, if $v \notin S'$, then $v = \top(S')$. □

Theorem 4 *Given a rooted tree decomposition of width k on $O(n)$ nodes, of a (di)graph G with $V(G) = \{1, \ldots, n\}$, there is an $O(kn)$ time preprocessing procedure that enables testing whether $x \in N(y)$ (resp. $x \in N^+(y)$ or $x \in N^-(y)$) in time $O(k)$, for all $x, y \in V(G)$.*

Proof: Let $(T, X), r$ be the given tree decomposition of width k, of a graph G on n vertices. For $x \in V(G)$, denote $\top(x) = \top(X^{-1}(x))$, i.e. the top node of all nodes that contain x in their bag. We now argue that in time $O(kn)$, $\top(x)$ can be computed for every $x \in V(G)$. For every node $u \in V(T)$: if u has a parent v, then mark all $x \in X_v$. Next, set $\top(x) = u$ for all unmarked $x \in X_u$. This is correct by Proposition 3(ii). Finally, reset the markings for all $x \in X_v$. For a single node, this procedure takes time $O(k)$. During this process, we can also compute for every node $u \in V(T)$ a list $L(u)$ of vertices x with $\top(x) = u$.

For $x \in V(G)$, define $N_{\text{top}}(x) = N(x) \cap X_{\top(x)}$. We argue that in time $O(kn)$, $N_{\text{top}}(x)$ can be computed for every $x \in V(G)$. For every node u, first mark all vertices in X_u. Next, for every vertex $x \in L(u)$, construct $N_{\text{top}}(x)$ by considering all neighbors and adding those that are marked to $N_{\text{top}}(x)$. Finally, reset the markings for all $x \in X_u$. For a single node u this procedure takes time $O(k) + O(\sum_{x \in L(u)} d(x))$. Since G has at most $O(kn)$ edges, This yields a complexity of $O(kn)$.

We argue that for any two vertices $x, y \in V(G)$, $xy \in E(G)$ if and only if $x \in N_{\text{top}}(y)$ or $y \in N_{\text{top}}(x)$. Suppose $xy \in E(G)$. By Definition 1, both $X^{-1}(x)$ and $X^{-1}(y)$ are connected, and $X^{-1}(x) \cap X^{-1}(y) \neq \emptyset$. So by Proposition 3(iii), $\top(x) \in X^{-1}(y)$ or $\top(y) \in X^{-1}(x)$ holds. Thus $y \in N_{\text{top}}(x)$ or $x \in N_{\text{top}}(y)$.

Finally, since every bag contains at most $k+1$ vertices, $|N_{\text{top}}(x)| \leq k+1$ holds for all $x \in V(G)$. So testing whether $x \in N_{\text{top}}(y)$ or $y \in N_{\text{top}}(x)$ can be done in time $O(k)$.

I simple modification of the above proof yields the statement for the case of digraphs. (The key statement then is that $(x, y) \in E(G)$ if and only if $x \in N^-_{\text{top}}(y)$ or $y \in N^+_{\text{top}}(x)$.) □

4 Dynamic programming rules for Min-Weight Homomorphism

Let G and H be digraphs, possibly with loops. A *homomorphism* from G to H is a function $f : V(G) \to V(H)$ such that for all $(u, v) \in E(G)$, $(f(u), f(v)) \in E(H)$.

Let $a : V(H) \to \mathbb{N}$ and $b : E(H) \to \mathbb{N}$ be weight functions. The *weight* of a homomorphism f from G to H is then

$$w(f) = \sum_{v \in V(G)} a(f(v)) + \sum_{(u,v) \in E(G)} b(f(u), f(v)).$$

The problem *Min-Weight Homomorphism (MWH)* is defined as follows: given digraphs G and H, with vertex and edge weights a and b, decide whether a homomorphism from G to H exists, and if so, compute on of minimum weight. Note that the analog problem where both G and H are undirected is a special case, since undirected edges can be replaced by a pair of oppositely directed edges. This problem generalizes many well-studied problems, such as:

- Max Independent Set (Min Vertex Cover): choose H to be an undirected graph on two vertices u and v, with an edge between them and a loop on u. The two edges and v have weight zero, u has weight one.
- q-Colorability: choose H to be a complete undirected graph on q vertices. The weights are irrelevant, since this is a decision problem.
- Max-Cut (Min Edge Bipartization): choose H to be an undirected graph on two vertices, with an edge between them and loops on both. The loops both receive weight one, and the other edge and both vertices receive weight zero.

For notational convenience, we modify H as follows: for every $u, v \in V(H)$ (including $u = v$), if $(u, v) \notin E(H)$ then add an edge (u, v) with weight ∞. So the original graph admits a homomorphism if and only if the new graph admits a homomorphism of finite weight. From now on, assume that $(u, v) \in E(H)$ for all u, v, possibly with infinite weight.

Let $(T, X), r$ be a rooted tree decomposition of G. For a node $u \in V(T)$, our DP computes values $\mathrm{val}_u(f)$ for every $f : X_u \to V(H)$, defined as follows:

$$\mathrm{val}_u(f) = \min\{w(h) \mid h : X(u) \to V(H) \text{ s.t. } h|_{X_u} = f\}.$$

So $\mathrm{val}(f)$ is the minimum weight of a homomorphism from $G(u)$ to H that coincides with f. Then since $G(r) = G$, the minimum weight of a homomorphism from G to H is computed by taking the minimum value of $\mathrm{val}_r(f)$ over all $f : X_r \to V(H)$. The values $\mathrm{val}_u(f)$ can be computed as follows, in case $(T, X), r$ is a nice tree decomposition:

Lemma 5 (*) *Let $(T, X), r$ be a nice tree decomposition, and let $u \in V(T)$.*

Leaf: *If u is a leaf node, then $val_u(f) = w(f)$.*
Forget: *If u is a forget node with child v, then $val_u(f) = \min\{val_v(h) \mid h : X_v \to V(H)$ s.t. $h|_{X_u} = f\}$.*
Introduce: *If u is an introduce node with child v and $X_u \setminus X_v = \{x\}$, then*[4]

$$val_u(f) = val_v(f|_{X_v}) + a(f(x)) +$$

[4] Note that summing over $y \in N^+(x) \cap X_u$ and $y \in N^-(x) \cap X_v$ is done to guarantee that a possible loop on x is only considered once.

$$\sum_{y \in N^+(x) \cap X_u} b(f(x), f(y)) + \sum_{y \in N^-(x) \cap X_v} b(f(y), f(x)).$$

Join: *If u is a join node with children v and x, then $val_u(f) = val_v(f) + val_x(f) - w(f)$.*

5 Implementation

We now show how the above rules can be implemented to yield a total DP complexity of $O(q^{k+1}n)$. First note that we use nice tree decompositions, and not *very* nice tree decompositions, i.e. we do not require for leaf nodes u that $|X_u| = 1$. This has the downside that the computation for leaf nodes becomes more complicated. However, the problem with very nice tree decompositions is that they may have more than $O(n)$ nodes (recall that we do not view k as a constant!): Consider the graph G_n with vertex set $\{x_1, \ldots, x_n\} \cup \{y_1, \ldots, y_n\}$, and edges $x_i x_j$ for all i, j, and $x_i y_j$ for all $i \neq j$. This graph has treewidth $n-1$: it has a tree decomposition (T, X) where T is a $K_{1,n}$, where the central node u has $X_u = \{x_1, \ldots, x_n\}$, and every y_i is contained in the bag for exactly one leaf. All bags have size n. It can be verified that any very nice tree decomposition of this graph, of width $k = \mathrm{tw}(G_n) = n - 1$, has $\Omega(kn)$ nodes. This makes it hard to prove the desired complexity, so we use nice tree decompositions instead, for which it is well-known that they have at most $4k$ nodes (see e.g. [13, Lemma 13.1.2]). Algorithmically, using the fact that for any $uv \in E(T)$, $X_u \setminus X_v$ and $X_v \setminus X_u$ can be computed in time $O(k)$ (see Section 3), one can prove the following:

Lemma 6 (*) *Let G be a graph on n vertices. Given a tree decomposition (T, X) of G of width k, on $O(n)$ nodes, in time $O(kn)$, a nice tree decomposition (T', X') of G of width k can be constructed, with at most $4n$ nodes.*

(We remark that in the end, this distinction is not so important; our computation method for leaf nodes can also be viewed as a way of analyzing very nice tree decompositions that have a specific form, which can always be guaranteed.) So now it suffices to prove that for any single node u, all values $\mathrm{val}_u(f)$ can be computed in time $O(q^{k+1})$. Since there may be q^{k+1} functions $f : X_u \to V(H)$, this requires that every value can be computed in (amortized) constant time. In the case of a join node (which is the most challenging to implement), the value $\mathrm{val}_v(f) + \mathrm{val}_w(f) - w(f|_{X_u})$ should be be computed in constant time. This provides two challenges: computing $w(f|_{X_u})$ in constant time for every f, and looking up the matching values $\mathrm{val}_v(f)$ and $\mathrm{val}_w(f)$ in constant time. The latter challenge is addressed by storing the values $\mathrm{val}_u(f)$ and $\mathrm{val}_v(f)$ for all f in two tables (for u and for v), which have to be ordered the same way. We first discuss details related to this ordering.

Ordered bags and tables W.l.o.g. we assume throughout that $V(H) = Q = \{0, \ldots, q-1\}$. Let $u \in V(H)$ and $X_u = \{x_1, \ldots, x_p\}$. Functions $f : X_u \to Q$

will be represented by a vector $(c_1, \ldots, c_p)$ here $c_i = f(x_i)$. This requires a complete order on the bag vertices. We assume that the vertices of X_u are ordered $(x_1, \ldots, x_p)$ such that for all $i < j$, $x_i < x_j$ holds. (Recall that $V(G) = \{1, \ldots, n\}$.) Ordered bags are denoted as tuples instead of sets. The values $\mathrm{val}_u(f)$ are then stored in a table (array) T_u of length q^p, according to the order given by $\mathrm{index}(f) = \sum_{i=1}^{p} q^{i-1} c_i$. So $T_u[\mathrm{index}(f)] = \mathrm{val}_u(f)$. (Note that $\mathrm{index}(f)$ is a bijection from all possible functions f to $\{0, \ldots, q^p - 1\}$.) If this ordering method is used, we say that T_u *is a table representing all functions* $f : X_u \to Q$*, ordered according to* $(x_1, \ldots, x_p)$. For a join node u with children v and w, we now have for every i that $T_u[i] = T_v[i] + T_w[i] - w(f|_{X_u})$ (where $i = \mathrm{index}(f)$), so we can easily find matching values in the tables of v and w. The desired order on the vertices of each bag can be guaranteed using a simple preprocessing step: Using any $O(k \log k)$ time sorting algorithm, this can be done with in time $O(nk \log k)$ for all nodes.

However, this introduces a small problem for forget and introduce nodes: e.g. for an introduce node u with child v, it is convenient to assume that $X_u = (x_1, \ldots, x_{p+1})$ and $X_v = (x_1, \ldots, x_p)$, i.e. the newly introduced vertex is the last one. This means that the tables may need to be reordered. Since this reordering corresponds to 'swapping only one coordinate', this can be done in time $O(q^p)$, as shown in the next lemma.

Lemma 7 *Let* $X = \{x_1, \ldots, x_p\}$*, and* $Q = \{0, \ldots, q-1\}$*. Let* T *and* T' *be tables representing all functions* $f : X \to Q$*, ordered according to* $(x_1, \ldots, x_p)$ *and* $(x_1, \ldots, x_{i-1}, x_{i+1}, \ldots, x_p, x_i)$*, respectively. Then* T *and* T' *can be computed from each other in time* $O(q^p)$*.*

Proof: For $f : X \to Q$ with $f(x_i) = c_i$ for all i, recall that $\mathrm{index}(f) = \sum_{j=1}^{p} q^{j-1} c_j$. Define $\mathrm{index}'(f) = \sum_{j=1}^{i-1} q^{j-1} c_j + \sum_{j=i+1}^{p} q^{j-2} c_j + q^{p-1} c_i$. This way, for every f it holds that $T[\mathrm{index}(f)] = T'[\mathrm{index}'(f)]$.

Define $x(f) = \sum_{j=1}^{i-1} q^{j-1} c_j$, $y(f) = \sum_{j=1}^{p-i} q^{j-1} c_{j+i}$ and $z(f) = c_i$. Then we can alternatively write $\mathrm{index}(f) = x(f) + q^i y(f) + q^{i-1} z(f)$ and $\mathrm{index}'(f) = x(f) + q^{i-1} y(f) + q^{p-1} z(f)$. In addition, observe that for every combination of values $x \in \{0, \ldots, q^{i-1} - 1\}$, $y \in \{0, \ldots, q^{p-i} - 1\}$ and $z \in \{0, \ldots, q-1\}$ there is a function f with $x(f) = x$, $y(f) = y$ and $z(f) = z$. Therefore, the tables T and T' can be computed from each other by looping over all possible combinations of values x, y and z, and using the equality $T[x + q^i y + q^{i-1} z] = T'[x + q^{i-1} y + q^{p-1} z]$. The values q^i, q^{i-1} and q^{p-1} can be computed beforehand, which ensures that the computation for a single combination of x, y and z requires only a constant number of elementary arithmetic operations (addition and multiplication). Since there are exactly q^p combinations of x, y and z, this proves the statement. □

Table computation We now show for every type of node u how the table T_u can be computed efficiently.

Lemma 8 *Let* u *be a forget node with child* v*, for which the table* T_v *is known. Let* $p = |X_v|$*. Then in time* $O(q^p)$*, the table* T_u *can be computed.*

Proof: First suppose that $X_u = (x_1, \ldots, x_{p-1})$ and $X_v = (x_1, \ldots, x_p)$, i.e. the last vertex in the (ordered) bag X_v is forgotten. Then we compute the values $T_u[i]$ as follows. First, initialize $T_u[i] = \infty$ for all $i \in \{0, \ldots, q^{p-1} - 1\}$. Next, for all combinations of $i \in \{0, \ldots, q-1\}$ and $j \in \{0, \ldots, q^{p-1} - 1\}$, reassign $T_u[j] := T_v[j + iq^{p-1}]$, if the latter value is smaller than the current value of $T_u[j]$. Because of the way the tables are ordered, and because $X_v \setminus X_u = \{x_p\}$, this correctly computes $T_u[\text{index}(f)] = \min_h T_v[\text{index}(h)]$ over all $h : X_v \to Q$ with $h|_{X_u} = f$. This computation takes constant time for one entry of T_v (using the precomputed value q^{p-1}), so time $O(q^p)$ in total.

In the case that $X_v = (x_1, \ldots, x_p)$ and $X_u = (x_1, \ldots, x_{i-1}, x_{i+1}, \ldots, x_p)$ for $i < p$, we can first translate the table T_v into a table T' ordered according to $(x_1, \ldots, x_{i-1}, x_{i+1}, \ldots, x_p, x_i)$, in time $O(q^p)$ (Lemma 7), and then apply the above procedure. □

Next, we show how to efficiently compute the table T_u for an introduce node u. Recall that if u is an introduce node with child v and $X_u \setminus X_v = \{x\}$, then $\text{val}_u(f)$ can be computed by adding to $\text{val}_v(f|_{X_v})$ a *correction term* of $a(f(x)) + \sum_{y \in N^+(x) \cap X_u} b(f(x), f(y)) + \sum_{y \in N^-(x) \cap X_v} b(f(y), f(x))$. We will show how to compute this correction term in (amortized) constant time. This involves looping over all $y \in X_u$, and deciding whether $y \in N^+(x)$ or $y \in N^-(x)$, respectively. This needs to be done in constant time, instead of time $O(k)$, as given by the method from Section 3. To this end, we use an initial preprocessing step that computes a *local adjacency matrix* A^u for every node u. To define A^u, we use the bag order introduced above again. Let $X_u = (x_1, \ldots, x_p)$. Then for $i, j \in \{1, \ldots, p\}$, $A^u_{i,j} = 1$ if and only if $(x_i, x_j) \in E(G)$, and $A^u_{i,j} = 0$ otherwise.

Proposition 9 (*) *Let (T, X) be a tree decomposition on $O(n)$ nodes of width k, of a graph G on n vertices, with ordered bags. In time $O(k^2 n)$, local adjacency matrices can be computed for every node.*

Furthermore, for H we store a vertex weight vector and edge weight matrix in the memory, to ensure that for any $x, y \in V(H)$, the values $a(x)$ and $b(x, y)$ can be retrieved in constant time. (Recall that $V(H) = \{0, \ldots, q-1\}$). This introduces at most a negligible term $O(q^2)$ to the complexity.

Lemma 10 *Let u be an introduce node with child v, for which the table T_v is known. Let $p = |X_u|$. Then in time $O(q^p)$, the table T_u can be computed.*

Proof: We assume that $X_u = (x_1, \ldots, x_p)$ and $X_v = (x_1, \ldots, x_{p-1})$, so x_p is the vertex that is introduced to the bag. This assumption is justified after using an $O(q^{p-1})$ preprocessing step, similar to the case of forget nodes (see above).

For $\alpha \in Q$ and $j \in \{0, \ldots, p-1\}$, define $X^j_u = \{x_1, \ldots, x_j, x_p\}$ (in particular $X^0_u = \{x_p\}$) and

$$\text{cor}^\alpha_j(f) = a(\alpha) + \sum_{y \in N^+(x) \cap X^j_u} b(\alpha, f(y)) + \sum_{y \in N^-(x) \cap X^j_u \setminus \{x\}} b(f(y), \alpha),$$

for any function $f : X_u^j \to Q$ with $f(x_p) = \alpha$. Define C_j^α to be a table containing these values, ordered according to $(x_1, \ldots, x_j)$. (More precisely: $C_j^\alpha[\text{index}(f)] = \text{cor}_j^\alpha(f)$, where $\text{index}(f) = \sum_{i=1}^{j} q^{i-1} f(x_i)$. The table C_j^α has length q^j.)

The table T_u can then be computed using the equality $T_u[i + q^{p-1}\alpha] = T_v[i] + C_{p-1}^\alpha[i]$, for all $i \in \{0, \ldots, q^{p-1} - 1\}$ and $\alpha \in \{0, \ldots, q-1\}$. So it now suffices to show how, for every $\alpha \in Q$, the table C_{p-1}^α can be computed in time $O(q^{p-1})$. These tables can be computed as follows: C_0^α has a single entry, with value $a(\alpha)$ if there is no loop on x_p, and value $a(\alpha) + b(\alpha, \alpha)$ otherwise. For every $j \geq 1$, C_j^α can be computed from C_{j-1}^α as follows. Loop over all values $\beta := f(x_j) \in \{0, \ldots, q-1\}$. For every such β, the next segment of C_j^α (of length q^{j-1}) consists of a copy of the table C_{j-1}^α, with a term $\text{val}^- + \text{val}^+$ added to each entry, where

- $\text{val}^- = b(\beta, \alpha)$ if $x_j \in N^-(x_p)$, and $\text{val}^- = 0$ otherwise, and
- $\text{val}^+ = b(\alpha, \beta)$ if $x_j \in N^+(x_p)$, and $\text{val}^+ = 0$ otherwise.

We use the entries $A_{j,p}^u$ and $A_{p,j}^u$ of the local adjacency matrix A^u to decide in constant time whether $x_j \in N^-(x_p)$ or $x_j \in N^+(x_p)$. Recall that using a precomputed edge weight matrix, the value $b(\beta, \alpha)$ can be retrieved in constant time. Since every entry computation now takes constant time, C_j^α can be computed from C_{j-1}^α this way in time $O(q^j)$, for every $j \in \{0, \ldots, p-1\}$. In total, this gives a complexity of $q + q^2 + \ldots + q^{p-1} = \frac{q^p - 1}{q-1} - 1 \in O(q^{p-1})$. □

Lemma 11 *Let u be a leaf, with $p = |X_u|$. Then in time $O(q^p)$, the table T_u can be computed.*

Proof: Let $X_u = (x_1, \ldots, x_p)$. Essentially, computing T_u is done by turning the nice tree decomposition into a very nice tree decomposition: the leaf u is replaced by a path $u_p, u_{p-1}, \ldots, u_1$ (rooted at u_p), with $X_{u_i} = (x_1, \ldots, x_i)$. In particular, $X_{u_p} = X_u$.

The table for node u_1 can be trivially computed in time $O(q)$. For $i \geq 2$, u_i is an introduce node, and by Lemma 10, the table T_{u_i} can be computed in time $O(q^i)$. Since $q + q^2 + \ldots + q^p = \frac{q^{p+1} - 1}{q-1} - 1 \in O(q^p)$, this shows that computing the table $T_{u_p} = T_u$ can be done in time $O(q^p)$. □

Lemma 12 *Let u be a join node, with $p = |X_u|$, and children v and w for which the tables T_v and T_w are known. Then in time $O(q^p)$, the table T_u can be computed.*

Proof: Recall that $\text{val}_u(f) = \text{val}_v(f) + \text{val}_w(f) - w(f)$ holds for every $f : X_u \to Q$. Because the tables for T_v and T_w are ordered the same way, the only remaining challenge lies in computing the correction terms $w(f)$. Essentially, this is done by introducing a new leaf child ℓ for u, with $X_\ell = X_u$ (ordered the same way as X_u). Then we can write $\text{val}_u(f) = \text{val}_v(f) + \text{val}_w(f) - \text{val}_\ell(f)$ for every f. In other words, $T_u[i] = T_v[i] + T_w[i] - T_\ell[i]$ for every index i. By Lemma 11, the table T_ℓ can be computed in time $O(q^p)$, which concludes the proof. □

Now we can prove our main result.

Theorem 13 *Let G be a digraph on n vertices, for which a tree decomposition (T, X) on $O(n)$ nodes is given, of width k. Let H be a digraph with $|V(H)| = q \geq 2$, and nonnegative integer vertex and edge weights. Then in time $O(nq^{k+1})$, it can be decided whether a homomorphism $f : V(G) \to V(H)$ exists, and if so, one of minimum weight can be computed.*

Proof: The algorithm can be summarized as follows. First, in time $O(kn)$ we transform the given tree decomposition into a nice tree decomposition of width k, on $O(n)$ nodes (Lemma 6). Next, we order the bags for every node, in time $O(k \log k\ n)$. Then we use the $O(kn)$ time preprocessing procedure from Theorem 4 to ensure that (directed) adjacency testing can be done in time $O(k)$. Finally, we use this to compute local adjacency matrices for every node, as defined above, in total time $O(k^2 n)$ (Proposition 9). This preprocessing phase has a total complexity of $O(k^2 n)$. For H, we precompute a vertex weight vector, edge weight matrix and power vector, to ensure that for every $x, y \in V(H)$, the values $a(x)$ and $b(x, y)$ can be retrieved in constant time, and for every $i \in \{0, \ldots, k\}$, the value q^i can be retrieved in constant time. This adds only a negligible term to the total complexity. Define $b(x, y) = \infty$ if $(x, y) \notin E(H)$.

At this point, the tree decomposition is in the desired form, and auxiliary data structures have been built, so that Lemmas 8–12 can be applied. These lemmas show that in time $O(q^p)$, the tables for a node u with $|X_u| = p$ can be computed. Since $p \leq k+1$, and there are at most $O(n)$ nodes, this yields a total complexity of $O(q^{k+1}n)$, to compute the table T_r for the root node $r \in V(T)$. Recall that by definition of the tables, and because $G(r) = G$, there exists a homomorphism f from G to H of weight at most m if and only if the table T_r contains a value of at most m. The total complexity of this algorithm becomes $O(k^2 n) + O(q^{k+1}n) \subseteq O(q^{k+1}n)$. Constructing a minimum weight homomorphism f can subsequently be done in a standard way: mark a minimum entry in the table for T_r, and use this to mark the corresponding entries in all other nodes, in a top down way (for forget nodes, see the proof of Lemma 8). Then for every vertex $x \in V(G)$, $f(x)$ can be computed by considering the marked entry in $\top(x)$. This can be implemented such that the complexity does not increase. □

6 Discussion

The fast DP table building techniques we use here can be used for many other DP problems. For instance, in the full version of this paper we will use them to show that Min Dominating Set can be solved in time $O(3^k n)$, if a *path* decomposition of width k is given. (For join nodes, no $O(3^k)$ implementation seems possible.) This is somewhat surprising since values in introduce nodes tables are the minimum of multiple values in the child table. These can nevertheless be computed in constant time on average, by computing a table of correction *pointers*, similar to the table of correction terms in Lemma 10.

References

1. Alfred V. Aho, John E. Hopcroft, and Jeffrey D. Ullman. The design and analysis of computer algorithms. *Addison-Wesley, Reading*, 4:1–2, 1974.
2. Jochen Alber and Rolf Niedermeier. Improved tree decomposition based algorithms for domination-like problems. In *LATIN 2002: Theoretical Informatics*, pages 613–627. Springer, 2002.
3. Stefan Arnborg and Andrzej Proskurowski. Linear time algorithms for NP-hard problems restricted to partial k-trees. *Discrete Applied Mathematics*, 23(1):11–24, 1989.
4. Hans L. Bodlaender. Dynamic programming on graphs with bounded treewidth. In *Proceedings 15th International Colloquium on Automata, Languages and Programming*, volume 317 of *LNCS*, pages 105–118. Springer, 1988.
5. Hans L. Bodlaender, Marek Cygan, Stefan Kratsch, and Jesper Nederlof. Solving weighted and counting variants of connectivity problems parameterized by treewidth deterministically in single exponential time. *arXiv preprint arXiv:1211.1505*, 2012. To appear in Proceedings ICALP 2013.
6. Hans L. Bodlaender and Arie M.C.A. Koster. Combinatorial optimization on graphs of bounded treewidth. *The Computer Journal*, 51(3):255–269, 2008.
7. Richard B. Borie. *Recursively Constructed Graph Families*. PhD thesis, School of Information and Computer Science, Georgia Institute of Technology, 1988.
8. Richard B. Borie, R. Gary Parker, and Craig A. Tovey. Solving problems on recursively constructed graphs. *ACM Computing Surveys*, 41(4), 2008.
9. Bruno Courcelle. The monadic second-order logic of graphs. I. recognizable sets of finite graphs. *Information and computation*, 85(1):12–75, 1990.
10. Marek Cygan, Jesper Nederlof, Marcin Pilipczuk, Johan M.M. van Rooij, and Jakub O. Wojtaszczyk. Solving connectivity problems parameterized by treewidth in single exponential time. In *Foundations of Computer Science (FOCS), 2011 IEEE 52nd Annual Symposium on*, pages 150–159. IEEE, 2011.
11. Reinhard Diestel. *Graph theory*. Springer-Verlag, 2010. 4th edition.
12. Russell Impagliazzo and Ramamohan Paturi. On the complexity of k-sat. *Journal of Computer and System Sciences*, 62(2):367–375, 2001.
13. Ton Kloks. *Treewidth: computations and approximations*, volume 842. Springer-Verlag New York Incorporated, 1994.
14. Daniel Lokshtanov, Dániel Marx, and Saket Saurabh. Known algorithms on graphs of bounded treewidth are probably optimal. In *Proceedings of the Twenty-Second Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 777–789. SIAM, 2011.
15. Rolf Niedermeier. *Invitation to fixed-parameter algorithms*, volume 31 of *Oxford Lecture Series in Mathematics and its Applications*. Oxford University Press, Oxford, 2006.
16. Johan M.M. van Rooij, Hans L. Bodlaender, and Peter Rossmanith. Dynamic programming on tree decompositions using generalised fast subset convolution. In *Algorithms-ESA 2009*, pages 566–577. Springer, 2009.
17. Jan Arne Telle and Andrzej Proskurowski. Algorithms for vertex partitioning problems on partial k-trees. *SIAM Journal on Discrete Mathematics*, 10(4):529–550, 1997.
18. Jacobo Valdes, Robert E. Tarjan, and Eugene L. Lawler. The recognition of series parallel digraphs. In *Proceedings of the eleventh annual ACM symposium on Theory of computing*, pages 1–12. ACM, 1979.

19. Thomas V. Wimer. *Linear Algorithms on k-Terminal Graphs.* PhD thesis, Dept. of Computer Science, Clemson University, 1987.

A Omitted proofs

Proof of Lemma 5: If u is a leaf node, then $G(u) = G[X_u]$, so $\mathrm{val}_u(f) = w(f)$ follows immediately from the definitions.

Suppose u is a forget node with child v, so $X(u) = X(v)$. Consider the function $g : X(u) \to V(H)$ with $g|_{X_u} = f$ that minimizes $w(g)$, so $\mathrm{val}_u(f) = w(g)$. Let $h = g|_{X_v}$. Then $h|_{X_u} = f$ and $\mathrm{val}_v(h) \leq w(g)$. This yields

$$\mathrm{val}_u(f) = w(g) \geq \mathrm{val}_v(h) \geq \min\{\mathrm{val}_v(h') \mid h' : X_v \to V(H) \text{ s.t. } h'|_{X_u} = f\}.$$

Similarly, choose $h : X_v \to V(H)$ to be the function with $h|_{X_u} = f$ that minimizes $\mathrm{val}_v(h)$. Let $g : X(v) \to V(H)$ with $g|_{X_v} = h$ and $w(g) = \mathrm{val}_v(h)$. Then $g|_{X_u} = f$, so

$$\min\{\mathrm{val}_v(h') \mid h' : X_v \to V(H) \text{ s.t. } h'|_{X_u} = f\} = \mathrm{val}_v(h) = w(g) \geq \mathrm{val}_u(f).$$

This proves the equality.

Suppose u is an introduce node with child v, and $X_u \setminus X_v = \{x\}$. Let $g : X(u) \to V(H)$ be the function with $g|_{X_u} = f$ that minimizes $w(g)$, so $\mathrm{val}_u(f) = w(g)$. By Property (ii) of Definition 1, $X(v) = X(u) \setminus \{x\}$ (x cannot be contained again in bags for descendants of v). Furthermore, all edges $(x, y) \in E(G)$ or $(y, x) \in E(G)$ with $y \in X(v)$ satisfy $y \in X_v$: otherwise, Property (ii) of Definition 1 shows that y is only contained in bags for descendants of v, whereas x is not contained in any such bag, which contradicts Property (i) of Definition 1. We conclude that the only edges in $G(u)$ that are not in $G(v)$ are of the form (x, y) or (y, x) with $y \in X_u$. This shows that

$$\mathrm{val}_u(f) = w(g) = w(g|_{X(v)}) + a(f(x)) +$$

$$\sum_{y \in N^+(x) \cap X_u} b(f(x), f(y)) + \sum_{y \in N^-(x) \cap X_v} b(f(y), f(x)).$$

(Note that we have to be careful not to count a possible loop (x, x) twice, since then $x \in N^+(x)$ and $x \in N^-(x)$.) Since $w(g|_{X(v)}) = \mathrm{val}_v(h)$ with $h = f|_{X_v}$, this proves the statement.

Let u be a join node with children v and x. By Property (ii) of Definition 1, $X(v) \cap X(x) = X_u$. In addition, similar to before, Property (i) shows that the edge set of $G(u)$ is the union of the edge sets of $G(v)$ and $G(x)$. So if $g : X(u) \to V(H)$ is the function with $g|_{X_u} = f$ that minimizes $w(g)$, then

$$\mathrm{val}_u(f) = w(g) = w(g|_{X(v)}) + w(g|_{X(x)}) - w(g|_{X_u}) = \mathrm{val}_v(f) + \mathrm{val}_x(f) - w(f).$$

□

Proof sketch of Lemma 6: Choose an arbitrary root $r \in V(T)$ and root the tree in time $O(n)$ (i.e. construct lists of children for each node). For every edge $uv \in E(T)$, compute $X_u \setminus X_v$ and $X_v \setminus X_u$ in time $O(k)$ (See the proof of Theorem 4). If both are empty (i.e. $X_u = X_v$) then contract uv (implemented properly, this can be done in constant time). Otherwise, subdivide uv with $|X_u \setminus X_v| + |X_v \setminus X_u| - 1$ new nodes, with appropriate bags. Next, for every node with $t \geq 2$ children, introduce $2t-2$ child nodes with the same bag, and connect these appropriately. It can be shown that this way a nice tree decomposition with at most $4n$ nodes can be constructed. Every new node introduction can be done in time $O(k)$, which yields a total complexity of $O(kn)$. We leave the implementation details as an exercise. □

Proof of Proposition 9: First compute $\top(x)$ and $N_{\text{top}}(x)$ for every $x \in V(G)$ in time $O(kn)$ (see Section 3). Observe that for a node u with $X_u = (x_1, \ldots, x_p)$ and a node x_i with $\top(x_i) = u$, row i and column i of the local adjacency matrix $A^u_{i,j}$ can be initialized in time $O(k)$, using the list $N_{\text{top}}(x)$. For the root node r, every vertex in X_r satisfies this property, so A^r can be computed in time $O(k^2)$. For children v of a join node u, $A^v = A^u$ holds, so this matrix can simply be copied in time $O(k^2)$. For children v of an introduce node u, A^v is obtained from A^u by deleting the appropriate row and column, so A^v can be constructed from A^u in time $O(k^2)$. For children v of a forget node u, say with $x = X_v \setminus X_u$, A^v can be constructed from A^u by adding a new row and column for x. Since $\top(x) = v$ (Proposition 3(ii)), this row and column can be constructed in time $O(k)$ using $N_{\text{top}}(x)$. □