Theory Comput Syst (2017) 61:283–304
DOI 10.1007/s00224-017-9751-3

Space Saving by Dynamic Algebraization Based on Tree-Depth

Martin Fürer[1] · Huiwen Yu[2,3]

Published online: 9 March 2017

Abstract Dynamic programming is widely used for exact computations based on tree decompositions of graphs. However, the space complexity is usually exponential in the treewidth. We study the problem of designing efficient dynamic programming algorithms based on tree decompositions in polynomial space. We show how to use a tree decomposition and extend the algebraic techniques of Lokshtanov and Nederlof (In: 42nd ACM Symposium on Theory of Computing, pp. 321–330, 2010) such that a typical dynamic programming algorithm runs in time $O^*(2^h)$, where h is the tree-depth (Nešetřil et al., Eur. J. Comb. **27**(6):1022–1041, 2006) of a graph. In general, we assume that a tree decomposition of depth h is given. We apply our algorithm to the problem of counting perfect matchings on grids and show that it outperforms other polynomial-space solutions. We also apply the algorithm to other set covering and partitioning problems.

Keywords Dynamic programming · Tree-depth · Tree decomposition · Space-efficient algorithms · Exponential time algorithms · Zeta transform

Research supported in part by NSF Grant CCF-0964655 and CCF-1320814

Part of this work has been done while the first author was visiting Theoretical Computer Science, ETH Zürich, Switzerland.

✉ Martin Fürer
furer@cse.psu.edu

Huiwen Yu
yhw.huiwenyu@gmail.com

[1] Department of Computer Science and Engineering, The Pennsylvania State University, University Park, PA, USA

[2] Google Incorporated, 1600 Amphitheatre Pkwy, Mountain View, CA, USA

[3] Pennsylvania State University, University Park, PA, USA

1 Introduction

Exact solutions to NP-hard problems typically adopt a branch-and-bound, inclusion/exclusion or dynamic programming framework. While algorithms based on branch-and-bound or inclusion/exclusion techniques [26] have shown to be both, time and space efficient, one problem with dynamic programming is that for many NP-hard problems, it uses exponential space to store the computation table. As in practice programs usually run out of space before they run out of time [33], exponential-space algorithms are considered impractical. Lokshtanov and Nederlof [24] have recently shown that algebraic tools like the zeta transform and Möbius inversion [30, 31] can be used to obtain space efficient dynamic programming solutions under some circumstances. The idea is sometimes referred to as the coefficient extraction technique which also appears in [21, 22].

The principle of space saving is best illustrated with the better known Fourier transform. Assume we want to compute a sequence of polynomial additions and multiplications modulo $x^n - 1$. We can either use a linear amount of storage and do many complicated convolution operations throughout, or we can start and end with the Fourier transforms and do the simpler component-wise operations in between. Because we can handle one component after another, very little space is needed during the main computation. This principle works for the zeta transform and subset convolution [5] as well.

In this paper, we study the problem of designing polynomial-space dynamic programming algorithms based on tree decompositions. Lokshtanov et al. [23] have also studied polynomial-space algorithms based on tree decompositions. They employ a divide and conquer approach. For a general introduction of tree decomposition, see the survey [9]. It is well-known that dynamic programming has wide applications and produces prominent results on efficient computations defined on path decompositions or tree decompositions in general [6]. Tree decompositions are very useful on low degree graphs as they are known to have a relatively low pathwidth [15]. For example, it is known that any degree 3 graph of n vertices has a path decomposition of pathwidth $\frac{n}{6}$. As a consequence, the minimum dominating set problem can be solved in time $O^*(3^{n/6})$[1], which is the best known running time in this case [29]. However, the algorithm trades large space usage for fast running time.

To tackle the high space complexity issue, we extend the method of [24] in a novel way to problems based on tree decompositions. In contrast to [24], we do not have a fixed ground set, and we cannot do the transformations only at the beginning and the end of the computation. The underlying set changes continuously, therefore a direct application to tree decompositions does not lead to an efficient algorithm. We introduce the new concept of zeta transforms for dynamic sets. Guided by a tree decomposition, the underlying set (of vertices in a bag) gradually changes. We adapt the transform so that it always corresponds to the current set of vertices. Herewith, we significantly expand the applicability of the space saving method by algebraization.

[1] O^* notation hides the polynomial factors of the expression.

We broadly explore problems which fit into this framework. Especially, we analyze the problem of counting perfect matchings on grids which is an interesting problem in statistical physics [18]. There is no previous theoretical analysis on the performance of any algorithm for counting perfect matchings on grids of dimension at least 3. We analyze two other natural types of polynomial-space algorithms, branching algorithms and dynamic programming algorithms based on path decompositions of subgraphs [20]. We show that our algorithm outperforms these two approaches. Our method is particularly useful when the treewidth of the graph is large. For example, grids, k-nearest-neighbor graphs [25], and low degree graphs are important graphs in practice with large treewidth. In these cases, standard dynamic programming on tree decompositions requires exponential space.

The paper is organized as follows. In Section 2, we summarize the basics of tree decomposition and techniques to save space by algebraization [24]. We also relate tree decompositions to tree-depth. In Section 3, we present the framework of our algorithm. In Section 4, we study the problem of counting perfect matchings on grids and extend our algorithmic framework to other problems. We reach a conclusion in Section 5.

2 Preliminaries

2.1 Saving Space Using Algebraic Transformations

Lokshtanov and Nederlof [24] introduce algebraic techniques to solve three types of problems. The first technique is using discrete Fourier transforms (DFT) on problems of very large domains, e.g., for the subset sum problem. The second one is using Möbius and zeta transforms when recurrences used in dynamic programming can be formulated as subset convolutions, e.g., for the unweighted Steiner tree problem. The third one is to solve the minimization version of the second type of problems by combining the above transforms, e.g., for the traveling salesman problem. As a service to the reader of this paper, we explain the techniques used in the second type of problems.

Given a universe V and a ring $\mathcal{R}$, we consider functions from 2^V to $\mathcal{R}$. Denote the collection of such functions by $\mathcal{R}[2^V]$. A singleton $f_A[X]$ is an element of $\mathcal{R}[2^V]$ which is zero unless $X = A$. The operator $\oplus$ is the pointwise addition, and the operator $\odot$ is the pointwise multiplication. We first define some useful algebraic transforms.

The *zeta transform* of a function $f \in \mathcal{R}[2^V]$ is defined to be

$$\zeta f[Y] = \sum_{X \subseteq Y} f[X]. \tag{1}$$

The *Möbius transform/inversion* [30, 31] of f is defined by

$$\mu f[Y] = \sum_{X \subseteq Y} (-1)^{|Y \setminus X|} f[X]. \tag{2}$$

The Möbius transform is the inverse transform of the zeta transform, as the two transformations interact in the following way [30, 31]:

$$\mu(\zeta f)[X] = f[X] \text{ and } \zeta(\mu f)[X] = f[X]. \tag{3}$$

The high level idea of [24] is that a direct computation of $f[V]$ has a mirror image where the zeta transformed versions of all intermediate results show up. Not only can the whole computation be performed directly with the zeta transformed values, but the computation is actually more convenient. While the direct computation stores exponentially many intermediate results $\{f[X]\}_{X\subseteq V}$, the computation operating on the zeta transformed values is partitioned into exponentially many parallel strands of computation. They can be executed sequentially, one strand after the other, using only polynomial space. The final value $f[V]$ can be obtained by Möbius inversion Eq. 2 as $f[V] = \sum_{X\subseteq V}(-1)^{|V\setminus X|}(\zeta f)[X]$.

Problems which can be solved in this manner have a common nature. They have recurrences which can be formulated by subset convolutions. The *subset convolution* [5] is defined to be

$$(f *_{\mathcal{R}} g)[X] = \sum_{X'\subseteq X} f(X')\, g(X\setminus X'). \tag{4}$$

To apply the zeta transform to $f *_{\mathcal{R}} g$, we need the *union product* [5] which is defined as

$$(f *_u g)[X] = \sum_{X_1 \bigcup X_2 = X} f(X_1)\, g(X_2). \tag{5}$$

The relation between the union product and the zeta transform is as follows [5]:

$$\zeta(f *_u g)[X] = (\zeta f)\odot(\zeta g)[X]. \tag{6}$$

In [24], functions over $(\mathcal{R}[2^V]; \oplus, *_{\mathcal{R}})$ are modeled by arithmetic circuits. Such a circuit is a directed acyclic graph where every node is either a singleton (constant gate), a $\oplus$ gate, or a $*_{\mathcal{R}}$ gate. In order to take advantage of Eq. 6, the concept of a relaxation has been introduced.

A *relaxation* of a function $f \in \mathcal{R}[2^V]$ is a sequence of functions

$$\{f^i : f^i \in \mathcal{R}[2^V], 0 \le i \le |V|\},$$

such that $\forall i, X \subseteq V$,

$$f^i[X] = \begin{cases} f[X] & \text{if } i = |X|,\\ 0 & \text{if } i < |X|,\\ \text{an arbitrary value} & \text{if } i > |X|. \end{cases}$$

Given any circuit C over $(\mathcal{R}[2^V]; \oplus, *_{\mathcal{R}})$ which outputs f, every gate in C computing an output a from its inputs b, c is replaced by small circuits computing a relaxation $\{a^i\}_{i=1}^{|V|}$ of a from relaxations $\{b^i\}_{i=1}^{|V|}$ and $\{c^i\}_{i=1}^{|V|}$ of b and c respectively. For a $\oplus$ gate, replace $a = b \oplus c$ by $a^i = b^i \oplus c^i$, for $0 \le i \le |V|$. For a $*_{\mathcal{R}}$ gate, replace $a = b *_{\mathcal{R}} c$ by the circuits for $a^i = \sum_{j=0}^{i} b^j *_u c^{i-j}$, for $0 \le i \le |V|$. This new circuit C_1 over $(\mathcal{R}[2^V]; \oplus, *_u)$ is of size $O(|C|\cdot|V|)$. An output $f[X]$ shows up as $f^{|X|}[X]$. The next step is to replace every $*_u$ gate by a gate $\odot$ and every constant gate a by ζa. It turns C_1 to a circuit C_2 over $(\mathcal{R}[2^V]; \oplus, \odot)$, such that for every gate of C_1 with output a, the corresponding gate in C_2 outputs ζa. Since additions and multiplications in C_2 are pointwise, C_2 can be viewed as $2^{|V|}$ disjoint circuits C^Y over $(\mathcal{R}[2^V]; +, \cdot)$ for every subset $Y \subseteq V$. The circuit C^Y outputs $(\zeta f)[Y]$. It is easy to see that the construction of every C^Y takes polynomial time.

As all problems of interest in this paper work on the integer domain $\mathbb{Z}$, we consider $\mathcal{R} = \mathbb{Z}$ and replace $*_{\mathcal{R}}$ by $*$ for simplicity. Assuming $0 \le f[V] < m$ for some integer m, we can view the computation as being done in the finite ring $\mathbb{Z}_m$. Additions and multiplications can be implemented efficiently in $\mathbb{Z}_m$ (e.g., using the fast algorithm of [16] for multiplication).

Theorem 1 (Theorem 5.1 [24]) *Let C be a circuit over $(\mathbb{Z}[2^V]; \oplus, *)$ which outputs f. Let all constants in C be singletons and let $f[V] < m$ for some integer m. Then $f[V]$ can be computed in time $O^*(2^{|V|})$ and space $O(|V||C|\log m)$.*

2.2 Tree Decomposition

For any graph $G = (V, E)$, a *tree decomposition* of G is a tree $\mathcal{T} = (V_{\mathcal{T}}, E_{\mathcal{T}})$ such that every node x in $V_{\mathcal{T}}$ is associated with a set B_x (called the bag of x) of vertices in G, and $\mathcal{T}$ has the following additional properties:

1. Every vertex is in the bag of some node, i.e., $\bigcup_{x \in V_{\mathcal{T}}} B_x = V$.
2. For any edge $e = \{u, v\} \in E$, there exists a node x such that $u, v \in B_x$.
3. For any nodes x, y, and any node z belonging to the path connecting x and y in $\mathcal{T}$, $B_x \cap B_y \subseteq B_z$.

The *width* of a tree decomposition $\mathcal{T}$ is $\max_{x \in V_{\mathcal{T}}} |B_x| - 1$. The *treewidth* of a graph G is the minimum width over all tree decompositions of G. In the following, we reserve the letter k for treewidth. Constructing a tree decomposition with minimum treewidth is an NP-hard problem. If the treewidth of a graph is bounded by a constant, a linear time algorithm for finding the minimum treewidth is known [8]. An $O(\log n)$-approximation algorithm for the treewidth is given in [11]. The result has been further improved to $O(\log k)$ in [1, 2, 12] and $O(\sqrt{\log k})$ in [14]. There also is a series of works studying constant factor approximations of treewidth k with running time exponential in k, see [10] and references therein.

To simplify the presentation of dynamic programming based on tree decompositions, an arbitrary tree decomposition is usually transformed into a *nice* tree decomposition. We employ a version of nice tree decomposition with explicit nodes to introduce edges. A similar type of nodes has been used by Cygan et al. [13] to introduce one edge at a time. Nice tree decompositions are tree decompositions with the following additional properties. A node in a nice tree decomposition has at most 2 children. Let c be the only child of x or let c_1, c_2 be the two children of x. Any node x in a nice tree decomposition is of one of the following five types:

1. A *leaf* node, a leaf of $\mathcal{T}$.
2. An *introduce vertex* node (introduce vertex v), where $B_x = B_c \cup \{v\}$.
3. A *forget* node (forget vertex v), where $B_x = B_c \setminus \{v\}$.
4. A *join* node, where x has two children and $B_x = B_{c_1} = B_{c_2}$.

5. An *introduce edges* node (introduce edge set E_v), where $B_x = B_c$, $E_v = E \cap \{\{u, v\} : u \in B_x\}$ for the vertex $v \in B_x$ that will be forgotten in the parent node of x. Node x is labeled with the edge set E_v.

We further require the root node to be a forget node with empty bag.

We don't require the leaves to have empty bags in order to keep the tree size linear in n. Every edge of G belongs to the label of exactly one introduce edges node. Notice that an introduce edges node is not a type of node in the standard definition of a nice tree decomposition, where an edge belongs to every node which contains its incident vertices. For any tree decomposition, a nice tree decomposition (according to the standard definition) with the same width can easily be constructed in polynomial time [19].

Our definition of nice tree decomposition also differs from the definition in [13] in two ways. We allow more than one edge to be introduced in one node, and leaves are not required to have empty bags. Without omitting these restrictions, one cannot guarantee minimum width and $O(n)$ tree size. On the other hand, our restriction of the edge set in a node to be a star allows efficient handling of these edges for many problems including the counting of perfect matchings, whereas for arbitrary edge sets, this would be a more complicated problem.

With our definition, we can easily modify a tree decomposition into a nice one of small size.

Proposition 1 *Every tree decomposition can be modified in polynomial time into a nice tree decomposition with the same width and size $O(n)$.*

Proof With our type of introduce edges node, still every edge can be introduced. If $\{u, v\}$ is an edge, then there is a bag B_x containing u and v. Assume v is forgotten in a lower ancestor y of x than u. Then the edge $\{u, v\}$ can be introduced in the child c of y.

We can require every subtree of the decomposition tree rooted at a child of a join node to contain at least one forget node. If this property is not fulfilled, we just delete such a subtree, as it is useless. As there are n forget nodes, we now have less than n join nodes. With our definition, the number of introduce edges nodes is also at most n. We insert such a node only as a child of a forget node.

To limit the number of introduce vertex nodes, we push them down the tree as far as we can without increasing the width of the tree. This procedure involves duplication whenever we push down past a join node. If an introduce vertex node is the parent of a leaf, then its introduced vertex is added to the leaf, and the introduce vertex node is removed. Now every introduce vertex node is the parent of a forget node, and there are at most n of them. □

One should note that the tree modification in this proof, also does not increase the tree-depth, which is defined next.

2.3 Tree-Depth

Our algorithms will crucially depend on the maximal number h of forget nodes along a path from the root to a leaf in a decomposition tree $\mathcal{T}$ of a graph G with an empty bag in the root node. We may always assume that $\mathcal{T}$ is a nice tree decomposition, as the simple transformation of any decomposition tree into a nice decomposition tree does not change the parameter h. It is easy to see that h can also be characterized as the maximum size of the union of all bags along any path from the root to a leaf in $\mathcal{T}$.

Let $h_m(G)$ be the minimum value of h for all tree decompositions of G. We show that $h_m(G)$ is equal to a well studied parameter, the *tree-depth* of a graph [27].

In the following definition, we measure the height of a tree by the maximal number of edges from the root to a leaf. (Some researchers use the number of nodes instead).

Definition 1 (tree-depth [27]) Given a rooted tree T with vertex set V, the closure of T, $clos(T)$ is the graph G with the same vertex set V, and the edge set consisting of all pairs $\{x, y\}$ such that one is a proper ancestor of the other. The tree-depth of T is the height of T plus 1. The tree-depth $\mathrm{td}(G)$ of a graph G, is the minimum height of trees T such that $G \subseteq clos(T)$.

Proposition 1 *For any connected graph G, $h_m(G) = td(G)$.*

Proof Given any tree decomposition of G, we first transform it to a nice tree decomposition $\mathcal{T}$. Then we contract $\mathcal{T}$ by deleting all nodes except the children of forget nodes. Let T_f be this contracted tree such that for every forget node in $\mathcal{T}$ which forgets a vertex x in G, the corresponding vertex in T_f is x. We have $G \subseteq clos(T_f)$. Therefore, $\mathrm{td}(G) \le h$, where h is the maximum number of forget nodes along any path from the root to a leaf in $\mathcal{T}$.

For any tree T such that $G \subseteq clos(T)$, we construct a corresponding tree decomposition $\mathcal{T}$ of G such that, $\mathcal{T}$ is initialized to be T and every bag associated with the vertex x of T contains the vertex itself. For every vertex $x \in T$, we also put all ancestors of x in T into the bag associated with x. It is easy to verify that this is a valid tree decomposition of G. Therefore, the tree-depth of T, $\mathrm{td}(T) \ge h_m(G)$. □

There is a close connection between treewidth k and tree-depth $\mathrm{td}(G)$ of a graph G. First of all, $\mathrm{td}(G) = h_m$ immediately implies $\mathrm{td}(G) \ge k + 1$. Moreover, $\mathrm{td}(G) \le (k + 1) \log |V|$ [27]. Furthermore, for any graph $G = (V, E)$ with a given tree decomposition of width k, one can find in polynomial time a tree decomposition of width $3k + 2$ and height $O(\log |V|)$ [7]. Thus, this tree has a tree-depth $O(k \log |V|)$.

3 Algorithmic Framework

3.1 The Principal of Dynamic Algebraization

In the traditional static setting, we have a circuit computing convolutions and (pointwise) additions of functions defined on all subsets of a fixed base set of size n. Using relaxations to break the functions into levels and applying a zeta transform to all functions, we obtain a mirror image of the original computation. The new zeta transformed computation is actually easier than the original computation, because multiplications are pointwise.

What is new in our dynamic setting is that we don't have a fixed base set. Instead of working with the full set of vertices, the algorithm works with a small subset defined by the bag of a node in the tree decomposition. This set changes dynamically, as the algorithm works its way bottom-up through the tree decomposition. Besides the convolution and addition operations, we also have operations, where the underlying set changes by one element. The function on the subsets of a new set is defined by case distinctions and linear combinations of some functions defined on subsets of the preceding old set. We will show that with these dynamic sets, we still can do the whole computation in the zeta transformed image, even as the set changes.

3.2 Dynamic Algebraization for Bounded Treewidth

We explain the algorithmic framework using the problem of counting perfect matchings based on a tree decomposition as an example to help understand the recurrences. The result can easily be applied to other problems. A *perfect matching* in a graph $G = (V, E)$ is a collection of $|V|/2$ edges such that every vertex in G belongs to exactly one of these edges.

Consider a connected graph G and a nice tree decomposition $\mathcal{T}$ of treewidth k on G. Let $\mathcal{T}_x$ be the subtree rooted at x. Let T_x be the set of vertices contained in bags associated with nodes in $\mathcal{T}_x$ which are not in B_x. For any $X \subseteq B_x$, let Y_X be the union of X and T_x. For every node x, we define a function $f_x \in \mathbb{Z}[2^{B_x}]$ as follows. For any $X \subseteq B_x$, let $f_x[X]$ be the number of perfect matchings in the subgraph whose vertex set is Y_X and whose edges are the edges within Y_X that have already been introduced in $\mathcal{T}_x$. We will see that the recurrence for computing f_x at a join node x can be formulated as a subset convolution, while at other types of tree nodes it is a linear combination or case distinction.

We go three times through the operations done by a dynamic programming algorithm working bottom-up on the tree decomposition $\mathcal{T}$ in order to compute the number of perfect matchings. First, we explain how to efficiently evaluate all f_x on a nice tree decomposition by dynamic programming in exponential space. X always denotes an arbitrary subset of B_x. Second, we show how to compute relaxations f_x^i of all the functions f_x. Finally, in the third round, we show how to compute $(\zeta f_x^i)[X]$ for all x, i, and X. It is important that these entries are computed at a node x from the entries at the children of x. This third round is the space efficient computation that is actually done by the algorithm. It does not work by applying the zeta transform to f_x^i, but operates entirely within the transformed functions. The previous rounds are just to motivate the algorithm and show its correctness.

Now we start the first round, showing how to compute $f_x[X]$ for the perfect matching problem for all nodes x and all subsets X of B_x.

1. x is a leaf node. $f_x[\emptyset] = 1$, as there is one empty perfect matching for the empty set of vertices. For all $X \neq \emptyset$, $f_x[X] = 0$, as there is no perfect matching without edges. For other problems, $f_x[X]$ can be anything that is easily computable.

2. x is an introduce vertex node. If the introduced vertex v is not in X, then $f_x[X] = f_c[X]$ by definition. If $v \in X$, then $f_x[X] = 0$, as v has no incident edges. For other problems, $f_x[X]$ may equal to $f_c[X \setminus \{v\}]$, which implies a similar recurrence.
3. x is a forget node. $f_x[X] = f_c[X \cup \{v\}]$ by definition.
4. x is a join node with two children. By assumption, the computation of f_x at a join node can be formulated as a subset convolution. We have

$$f_x[X] = \sum_{X' \subseteq X} f_{c_1}[X'] f_{c_2}[X \setminus X'] = (f_{c_1} * f_{c_2})[X]. \tag{7}$$

   For the problem of counting perfect matchings, it is easy to verify that $f_x[X]$ can be computed using Eq. 7. This is so, because every vertex of X is either matched in the left or the right subtree. Every matching in the left subtree involving X' can be combined with every matching in the right subtree involving $X \setminus X'$ to result in a matching involving X.
5. x is an introduce edges node introducing the edge set $E_v = \{\{u_j, v\} : j = 1, \ldots, q\}$. We have to consider matchings avoiding all edges of E_v, and matchings using one edge of E_v. If $v \notin X$, then $f_x[X] = f_c[X]$. If $v \in X$ and $E_v = \{\{u_j, v\} : j = 1, \ldots, q\}$ with $u_1, \ldots, u_p \in X$, but $u_{p+1}, \ldots, u_q \notin X$ then

$$f_x[X] = f_c[X] + \sum_{j=1}^{p} f_c[X \setminus \{u_j, v\}].$$

   Here, $f_c[X]$ counts the matchings avoiding E_v, and $f_c[X \setminus \{u_j, v\}]$ counts the matchings using the edge $\{u_j, v\}$ of E_v.

   For other problems, the recurrence can be different. Since the goal of the analysis of this case is to explain why we need to modify the construction of an introduce edges node, we consider only the recurrence for the counting perfect matchings problem.

Here, we should notice some significant differences between our dynamic framework and the static framework of Theorem 1. In case 2, we observe that there is no addition, but a case distinction, and in cases 3 and 5, the addition operation is not a strictly pointwise operation. This is because in a tree decomposition, the bags of different nodes are not the same.

In case 2, the domain of f_x is twice the size of the domain f_c of the child c of x. Depending on whether the new vertex v is in X or not, the value is copied or set to 0.

In case 3, the domain of f_x is half the size of the domain f_c of the child c of x. The value at X is copied from $X \cup \{v\}$, while the value of f_c at X is not used. An immediate implication is that ζf_x is obtained as a simple linear combination of the values of f_c on the two halves of its domain, as we will see below.

Having defined the functions f_x for all x, we show now how to obtain their relaxations f_x^i for $0 \le i \le |B_i|$. Computing all relaxations f_x^i of all functions f_x produces all the results obtainable by computing f_x for all x.

1. For a leaf node x, we can choose $f_x^i[X] = f_x[X]$ for $i \geq |X|$, and $f_x^i[X] = 0$ for $i < |X|$. Thus for the perfect matching problem, $f_x^i[X] = f_x[X]$ for all X and i, in other words, $f_x^i[\emptyset] = 1$ for all i, and $f_x^i[X] = 0$ for all $X \neq \emptyset$.
2. For an introduce vertex node x introducing the vertex v, $f_x^i[X] = f_c^i[X]$ if $v \notin X$ and $f_x^i[X] = 0$ if $v \in X$.
3. For a forget node x forgetting the vertex v, $f_x^i[X] = f_c^{i+1}[X \cup \{v\}]$.
4. For a join node x, we want to compute $\tilde{f}_x^i[X] = \sum_{j=0}^{i}(f_{c_1}^j * f_{c_2}^{i-j})[X]$. We replace the subset convolution by a union product to obtain $f_x^i[X] = \sum_{j=0}^{i}(f_{c_1}^j *_u f_{c_2}^{i-j})[X]$. It does not matter that we compute f_x^i instead of $\tilde{f}_x^i$, because these two functions agree on all arguments X of size i. The sequences of functions f_x^i $(i = 0, 1, \ldots, |B_x|)$ and $\tilde{f}_x^i$ $(i = 0, 1, \ldots, |B_x|)$ are different relaxations of the same function f_x.
5. For an introduce edges node x, introducing the edges $\{\{u_j, v\} : j = 1, \ldots, q\}$, we have $f_x^i[X] = f_c^i[X] + \sum_{j=1}^{p} f_c^{i-2}[X \setminus \{u_j, v\}]$ for $i \geq 2$, where $\{u_j, v\} \subseteq X$ for $j = 1, \ldots, p$ and $\{u_j, v\} \nsubseteq X$ for $j = p+1, \ldots, q$. For $i \leq 1$ we have $f_x^i[X] = f_c^i[X]$.

Finally, in our third round, we explain how to efficiently evaluate all ζf_x^i on a nice tree decomposition by dynamic programming in polynomial space. Step 5, the handling of an introduce edges node, is tentative. We will modify it afterwards. For the root x, $(\zeta f_x^0)[\emptyset] = f_x^0[\emptyset] = f_x[\emptyset]$, which is the value of interest, i.e., the number of perfect matchings in the whole graph for our example.

As in the construction of Theorem 1, after replacing every f_x by a relaxation $\{f_x^i\}_{0 \leq i \leq |B_x|}$ of f_x, we apply zeta transforms. Here we consider the zeta transforms of f_x^i, for all x and $0 \leq i \leq |B_x|$.

1. For a leaf node x, $f_x^i[\emptyset] = 1$ and $f_x^i[X] = 0$ for $X \neq \emptyset$ for the perfect matching problem. This immediately implies $(\zeta f_x^i)[X] = 1$ for all i and all X.
2. For an introduce vertex node x introducing the vertex v, we have $f_x^i[X] = f_c^i[X]$ if $v \notin X$ and $f_x^i[X] = 0$ if $v \in X$. By definition of the zeta transform, if $v \in X$, we have

$$(\zeta f_x^i)[X] = \sum_{X' \subseteq X} f_x^i[X'] = \sum_{v \notin X' \subseteq X} f_x^i[X'] = \sum_{X' \subseteq X \setminus \{v\}} f_x^i[X'].$$

Therefore, we obtain

$$(\zeta f_x^i)[X] = \begin{cases} (\zeta f_c^i)[X] & v \notin X \\ (\zeta f_c^i)[X \setminus \{v\}] & v \in X \end{cases} \tag{8}$$

for the perfect matching problem. Thus,

$$(\zeta f_x^i)[X] = (\zeta f_c^i)[X \setminus \{v\}] \text{ for all } X.$$

3. For a forget node x, we have $f_x^i[X] = f_c^{i+1}[X \cup \{v\}]$. Therefore, we obtain

$$\begin{aligned}(\zeta f_x^i)[X] &= \sum_{X' \subseteq X} f_x^i[X'] = \sum_{X' \subseteq X} f_c^{i+1}[X' \cup \{v\}] \\ &= (\zeta f_c^{i+1})[X \cup \{v\}] - (\zeta f_c^{i+1})[X]. \end{aligned} \tag{9}$$

4. In a join node x, we have $f_x^i[X] = \sum_{j=0}^{i}(f_{c_1}^j *_u f_{c_2}^{i-j})[X]$. Now we use the big advantage of the zeta transform. Union products transform into pointwise multiplication. We obtain

$$(\zeta f_x^i)[X] = \sum_{j=0}^{i}(\zeta f_{c_1}^j)[X] \cdot (\zeta f_{c_2}^{i-j})[X], \text{ for } 0 \leq i \leq k+1. \tag{10}$$

5. For an introduce edges node x, introducing the edges $\{\{u_j, v\} : j = 1, \dots, q\}$, we have $f_x^i[X] = f_c^i[X] + \sum_{j=1}^{p} f_c^{i-2}[X \setminus \{u_j, v\}]$ for $i \geq 2$, where $\{u_j, v\} \subseteq X$ for $j = 1, \dots, p$ and $\{u_j, v\} \not\subseteq X$ for $j = p+1, \dots, q$ and $f_x^i[X] = f_c^i[X]$ for $i \leq 1$.

   Therefore, we obtain

$$\begin{aligned}(\zeta f_x^i)[X] &= \sum_{X' \subseteq X} f_x^i[X'] \\ &= \sum_{X' \subseteq X} f_c^i[X'] + \sum_{j=1}^{p} \sum_{\{u_j,v\} \subseteq X' \subseteq X} f_c^{i-2}[X' \setminus \{u_j, v\}] \\ &= \sum_{X' \subseteq X} f_c^i[X'] + \sum_{j=1}^{p} \sum_{X' \subseteq X \setminus \{u_j,v\}} f_c^{i-2}[X'] \\ &= (\zeta f_c^i)[X] + \sum_{j=1}^{p}(\zeta f_c^{i-2})[X \setminus \{u_j, v\}]\end{aligned}$$

   for $i \geq 2$, and $(\zeta f_x^i)[X] = (\zeta f_c^i)[X]$ for $i \leq 1$. Note that the sum over j depends on X.

In cases 3 and 5, we see that the value of $(\zeta f_x)[X]$ depends on the values of ζf_c on more than one subset. We can visualize the computation along a path from a leaf to the root as a computation tree. This computation tree branches on introduce edges nodes and forget nodes. Suppose along any path from the root to a leaf in $\mathcal{T}$, the maximum number of introduce edges nodes is m' and the maximum number of forget nodes is h. To avoid exponentially large storage for keeping partial results in this computation tree, we compute along every path from a leaf to the root in this tree. This leads to an increase of the running time by an exponential factor in m' and k, but the computation is in polynomial space (explained in detail later). To reduce the running time, we eliminate the branching introduced by introduce edges nodes. On the other hand, the branching introduced by forget nodes seems inevitable.

We describe now a new way to handle the introduction of edges. It still allows to introduce a whole star of edges. For any introduce edges node x which introduces an edge set $E_v = \{\{u_j, v\} : j = 1, \dots, q\}$ and has a child c in the original nice tree decomposition $\mathcal{T}$, we add an new child c' of x, such that $B_{c'} = B_x$ and introduce the edge set E_v in c'. The node c' is an auxiliary leaf. We assume the evaluation of ζf_c takes only polynomial time in general. For the counting perfect matchings problem, $f_{c'}[X] = 1$ only when $X = \emptyset$ or $X = \{u_j, v\}$ for some edge $\{u_j, v\} \in E_v$, otherwise it is equal to 0. We can define $f_{c'}^i[\emptyset] = 1$ for all i, $f_{c'}^2[X] = 1$ for $X = \emptyset$

or $X = \{u_j, v\}$ for some j, and $f^i_{c'}[X] = 0$ for all other cases. Then $(\zeta f^0_{c'})[X] = 1$ for all X, $(\zeta f^2_{c'})[X]$ is equal to one more than the number of edges $\{u_j, v\}$ which are subsets of X, and $(\zeta f^i_{c'})[X] = 0$ for all $i \notin \{0, 2\}$ and all X.

We call x a *modified introduce edges* node and c' an *auxiliary leaf*. As the computation at x is the same as the computation at a join node, we do not talk about the computation at modified introduce edges nodes separately. We call the new tree decomposition a modified nice tree decomposition.

In Theorem 1, the transformed circuit C_2 can be viewed as $2^{|V|}$ disjoint circuits. In the case of a tree decomposition, the computation branches on forget nodes in addition to the branching at join nodes. Therefore, we cannot take C_2 as $O(2^k)$ disjoint circuits.

Let h be the tree-depth of $\mathcal{T}$. Let h' be the number of forget nodes along any path $\mathcal{P}$ from the root to a leaf of $\mathcal{T}$. Then $h' \le h$. Corresponding to this path $\mathcal{P}$, there is a computation tree with $2^{h'}$ leaves branching at each forget node on the path $\mathcal{P}$.

The time spent on this path $\mathcal{P}$ is $O^*(2^h)$, as the computation traverses the tree corresponding to $\mathcal{P}$ in a depth-first search order. The tree $\mathcal{T}$ has at most V leaves, and therefore at most V paths from the root to a leaf. Hence, the total time is still $O^*(2^h)$ and the space is polynomial.

We call our algorithm based on a modified nice tree decomposition **Algorithm 1**. To summarize, we present the algorithm for the problem of counting perfect matchings based on a modified nice tree decomposition $\mathcal{T}$ in Algorithm 1.

Algorithm 1 Counting perfect matchings on a modified nice tree decomposition

```
    **Input**: a modified nice tree decomposition $\mathcal{T}$ with root $r$.
    **return** $(\zeta f)(r, \emptyset, 0)$.
    **procedure** $(\zeta f)x, X, i$. // $(\zeta f)(x, X, i)$ represents $(\zeta f^i_x)[X]$.
        // Assume $X$ is a subset of $B_x$ and $0 \le i \le |X|$.
        **if** $x$ is a leaf: **return** 1.
        **if** $x$ is the auxiliary leaf for $E_v = \{\{u_j, v\} : j = 1, \dots, q\}$
            **return** 1 when $v \notin X$,
            **return** $1 + |\{u_1, \dots, u_q\} \cap X|$ when $v \in X$.
        **end if**
        **if** $x$ is an introduce vertex node : **return** $(\zeta f)(c, X \setminus \{v\}, i)$.
        **if** $x$ is a forget node: **return** $(\zeta f)(c, X \cup \{v\}, i+1) - (\zeta f)(c, X, i+1)$.
        **if** $x$ is a join node: **return** $\sum_{j=0}^{i} (\zeta f)(c_1, X, j) \cdot (\zeta f)(c_2, X, i-j)$.
    **end procedure**
```

Theorem 2 *Let $G = (V, E)$ be any graph, and let $\mathcal{T} = (V_\mathcal{T}, E_\mathcal{T})$ be a modified nice tree decomposition of G with depth h, width k, and size $|V_\mathcal{T}| = O(|V|)$. Let $\{f_x : x \in V_\mathcal{T}\}$ be a collection of functions with $f_x \in \mathbb{Z}[2^{B_x}]$. Assume, the function f_x at x is defined by the functions at its children as follows.*

If x is a leaf: $f_x \in \mathbb{Z}[2^{B_x}]$ *is an arbitrary function with ζf_x computable in linear time from B_x.*

If x is an auxiliary leaf for some edge set E_x: $f_x \in \mathbb{Z}[2^{B_x}]$ *is an arbitrary function with ζf_x computable in linear time from (B_x, E_x).*

If x is an introduce vertex v node with child c:

For some constants d_1, d_2, either

$$f_x[X] = \begin{cases} d_1 f_c[X] & \text{if } v \notin X \\ 0 & \text{if } v \in X \end{cases} \quad \text{or} \quad f_x[X] = \begin{cases} 0 & \text{if } v \notin X \\ d_2 f_c[X \setminus \{v\}] & \text{if } v \in X. \end{cases}$$

If x is a forget node forgetting vertex v with child c: *For some constants d_1', d_2',* $f_x[X] = d_1' f_c[X] + d_2' f_c[X \cup \{v\}]$.

If x is a join node with children c_1 and c_2: $f_x[X] = f_{c_1}[X] * f_{c_2}[X]$.

Assume, $0 \le f_r[\emptyset] < m$ for some integer m, where r is the root. Then, $f_r[\emptyset]$ can be computed in time $O^*(2^h)$ and space $O(|V| + hk \log m)$.

Proof As in the proof of Theorem 1, we have a circuit C defined by the decomposition tree with the algebraic operations $\oplus$ and $*$. In addition, due to the dynamic nature of the underlying set, we have modifications of the functions best described by array operations. For insertion nodes, when the array size doubles, either half can be padded with zeros, for forget nodes, when the array is divided in half, the array can be a linear combination of the previous halves.

The size of the circuit C is the number of nodes of the decomposition tree $\mathcal{T}$, which is $O(|V|)$. Again, to obtain union products instead of subset convolutions, we use relaxations. As all sets are of size at most $k+1$, the relaxations have only the levels $0, \ldots, k+1$. Thus, we obtain a circuit C_1 of size $O(k|V|)$ operating on the levels of the relaxations. The circuit C_1 can easily be found in polynomial time. The circuit C_1 has $O(k^2|V|)$ edges, but there is no need to represent C_1 explicitly.

The circuit C_1 is transformed into a circuit C_2 operating on the zeta transformed functions. We now describe the circuit C_2 recursively according to the types of nodes involved. For all types of nodes, except for join nodes, the relaxations play no major role, because no convolutions are involved. Therefore, to simplify the description, we pretend to move directly from f_x to ζf_x. In reality, the described transformation is not applied to f_x but to every level f_x^i of the relaxation of f_x.

If x is an ordinary or auxiliary leaf, then we have assumed ζf_x is linear time computable.

If x is an introduce vertex node, then the theorem allows two options. If

$$f_x[X] = \begin{cases} d_1 f_c[X] & \text{if } v \notin X \\ 0 & \text{if } v \in X, \end{cases}$$

then

$$(\zeta f_x)[X] = d_1 (\zeta f_c)[X \setminus \{v\}].$$

If

$$f_x[X] = \begin{cases} 0 & \text{if } v \notin X \\ d_2 f_c[X \setminus \{v\}] & \text{if } v \in X. \end{cases}$$

then

$$(\zeta f_x)[X] = \begin{cases} 0 & \text{if } v \notin X \\ d_2 (\zeta f_c)[X \setminus \{v\}] & \text{if } v \in X. \end{cases}$$

If x is a forget node, then $f_x[X] = d'_1 f_c[X] + d'_2 f_c[X \cup \{v\}]$ implies

$$(\zeta f_x)[X] = (d'_1 - d'_2)(\zeta f_c)[X] + d'_2(\zeta f_c)[X \cup \{v\}].$$

If x is a join node, then the operation is problem independent. Thus, the argument is exactly the same as in the special case of counting perfect matchings Eqs. 7 and 10.

Finally, at the end of the computation a Möbius transform (inverse of zeta transform) is required. But this is trivial if the root node r has an empty bag. We have $f_r[\emptyset] = (\zeta f_r)[\emptyset]$, and $\emptyset$ is the only element in the domain of f_r.

The claim about the running time follows from the fact that only in case of a forget node or a join node is $(\zeta f_x)[X]$ computed from two different previous values, i.e., we have a twofold branching in these nodes. In order to show a bound on the running time, we map the computation tree $\mathcal{T}'$ with branchings at forget nodes and join nodes in the natural way into the original tree $\mathcal{T}$ with branchings only at the join nodes. In this way up to 2^h nodes of the computation tree $\mathcal{T}'$ are mapped into one node of $\mathcal{T}$. During the computation, $\mathcal{T}'$ is traversed in depth-first order in time $O(2^h|V|)$, because a single traversal of $\mathcal{T}$ takes $O(|V|)$ steps. In the node of the computation tree corresponding to (x, X), all levels of the relaxation $f_x^0[X], \ldots, f_x^{|X|}[X]$ of f_x are handled together using space $O(k)$ and time $O(k^2)$ if the small convolutions Eq. 10 are executed in a trivial manner.

Naturally, the implicit depth-first search evaluation, which proceeds as in Algorithm 1, recomputes certain values. That's why, it can save space at the cost of increasing the exponent in the time from the treewidth k to the tree-depth h. Instead of storing an array of length 2^{k+1}, we compute up to 2^h times an entry of this array. (Here, an entry means a vector $(f_x^0[X], \ldots, f_x^{|X|}[X])$.)

The term $|V|$ in the space bound is obtained by allowing to store the whole decomposition tree $\mathcal{T}$. It is not necessary to store the bags explicitly, because they are determined by the introduce and forget nodes. During the computation, h tree nodes are active. In each such node x, one component X of all $k+2$ levels of the relaxation $f_x^0, \ldots, f_x^{k+1}$ is stored. It is sufficient to store all numbers modulo m. □

Corollary 1 *Let $G = (V, E)$ be any graph, and let $\mathcal{T} = (V_\mathcal{T}, E_\mathcal{T})$ be a modified nice tree decomposition of G with depth h and width k. Then the number of perfect matchings in G can be computed in time $O^*(2^h)$ and space $O(|V|hk \log k)$.*

Proof There are $O(k^{|V|})$ perfect matchings in G, as every one of the $O(|V|)$ auxiliary leaves has at most $k+1$ choices to select or not to select a matched edge. Thus, $m = O(|V| \log k)$ is sufficient. □

4 Counting Perfect Matchings

The problem of counting perfect matchings is ♯P-complete. It has long been known that in a bipartite graph of size $2n$, counting perfect matchings takes $O^*(2^n)$ time using the inclusion and exclusion principle. A recent breakthrough [3] shows that the same running time is achievable for general graphs. For low degree graphs, improved results based on dynamic programming on path decompositions of a sufficiently large subgraph are known [4].

Counting perfect matchings on grids is an interesting problem in statistical physics [18]. The more general problem is the Monomer-Dimer problem [18], which essentially asks to compute the number of matchings of a specific size. We model the Monomer-Dimer problem as the computation of the matching polynomial. For grids in dimension 2, the pure Dimer (perfect matching) problem is polynomial-time tractable and an explicit expression of the solution is known [32]. We consider the problem of counting perfect matchings in a cube or hypercube in Section 4.1. Results on counting perfect matchings in more general grids, computing the matching polynomial and applications to other set covering and partitioning problems are presented in Section 4.2.

4.1 Counting Perfect Matchings on Cube/hypercube

We consider the case of counting perfect matchings on grids of dimension d, where $d \geq 3$ and the length of the grid is n in each dimension. We denote this grid by $G_d(n)$. To apply Algorithm 1, we first construct a balanced tree decomposition on $G_d(n)$ with the help of balanced separators. The balanced tree decomposition can easily be transformed into a modified nice tree decomposition. The results also hold for any subgraph of $G_d(n)$ of size $\Omega(n^d)$.

Tree Decomposition Using Balanced Vertex Separators We first explain how to construct a balanced tree decomposition using vertex separators of general graphs. An α-balanced vertex separator of a graph/subgraph G is a set of vertices $S \subseteq G$, such that after removing S, G is separated into two disjoint parts A and B with no edge between A and B, and $|A|, |B| \leq \alpha|G|$, where α is a constant in (0, 1). Suppose we have an oracle to find an α-balanced vertex separator of a graph. We begin with creating the root of a tree decomposition $\mathcal{T}$ and associating the vertex separator S of the whole graph with the root. Consider a subtree $\mathcal{T}_x$ in $\mathcal{T}$ with the root x associated with a bag B_x. Denote the vertices belonging to nodes in $\mathcal{T}_x$ by V_x. Initially, $V_x = V$ and x is the root of $\mathcal{T}$. Suppose we have a vertex separator S_x which partitions V_x into two disjoint parts V'_{c_1} and V'_{c_2}. We create two children c_1, c_2 of x, such that the set of vertices belonging to $\mathcal{T}_{c_i}$ is $V_{c_i} = S_x \cup V'_{c_i}$. Denote the set of vertices belonging to nodes in the path from x to the root of $\mathcal{T}$ by U_x, we define the bag B_{c_i} to be $S_{c_i} \cup (V_{c_i} \cap U_x)$, for $i = 1, 2$. It is easy to verify that this is a valid tree decomposition. Since V_x decreases by a factor of at least $1 - \alpha$ in each partition, the height of the tree is at most $\log_{\frac{1}{1-\alpha}} n$. To transform this decomposition into a modified nice tree decomposition, we only need to add a series of introduce vertex nodes, forget nodes and modified introduce edges nodes between two originally adjacent nodes. We refer to the algorithm of this paragraph as the **Tree Decomposition Algorithm**.

We observe that after the transformation, the number of forget nodes from c_i to x is the size of the balanced vertex separator of V_{c_i}, i.e. $|S_{c_i}|$. Therefore, the number of forget nodes from the root to a leaf is the size of the union of the balanced vertex separators used to construct this path in the tree decomposition.

A grid graph $G_d(n)$ has a nice symmetric structure. Denote the d dimensions by $x_1, x_2, ..., x_d$ and consider an arbitrary subgrid G'_d of $G_d(n)$ with length n'_i in dimension x_i. The hyperplane in G'_d which is perpendicular to x_i and cuts G'_d into roughly two halves can be used as a $1/2$-balanced vertex separator. We always cut a dimension with a longest length. If $n'_i = n'_{i+1}$, we choose to first cut the dimension x_i, then x_{i+1}. We illustrate the construction of the 2-dimensional case in the following example.

Example 1 (*Balanced tree decomposition on $G_2(n)$*) The left picture is a partitioning of a 2-dimensional grid. We always bipartition the longer side of the grid/subgrid. The right picture is the corresponding balanced tree decomposition of this grid. The letter P with the same indices on both sides represent the same set of nodes. Each P_i represents a balanced vertex separator. We denote the left/top half of P_i by P_{i1}, and the right/bottom part by P_{i2} (see Fig. 1). The treewidth of this decomposition is $\frac{3}{2}n$.

To run the Tree Decomposition Algorithm on $G_d(n)$, we cut dimensions $x_1, x_2, ..., x_d$ consecutively with separators of size $\frac{1}{2^{i-1}}n^{d-1}$, for $i = 1, 2..., d$. Then we proceed with subgrids of length $n/2$ in every dimension. It is easy to see that the width of this tree decomposition is $\frac{3}{2}n^{d-1}$. The tree-depth h of this tree decomposition is at most $\sum_{j=0}^{\infty}\sum_{i=0}^{d-1}\frac{1}{2^i}\cdot(\frac{1}{2^j}n)^{d-1}$, which is $\frac{2^d-1}{2^{d-1}-1}n^{d-1}$.

Lemma 1 *The treewidth of the tree decomposition $\mathcal{T}$ on $G_d(n)$ obtained by the Tree Decomposition Algorithm is $k = \frac{3}{2}n^{d-1}$. The tree-depth of $\mathcal{T}$ is at most $h = \frac{2^d-1}{2^{d-1}-1}n^{d-1}$.*

Together with Corollary 1, we obtain the result for our main example.

Theorem 3 *The problem of counting perfect matchings on grids of dimension d and uniform length n can be solved in time $O^*(2^{\frac{2^d-1}{2^{d-1}-1}n^{d-1}})$ and in polynomial space.*

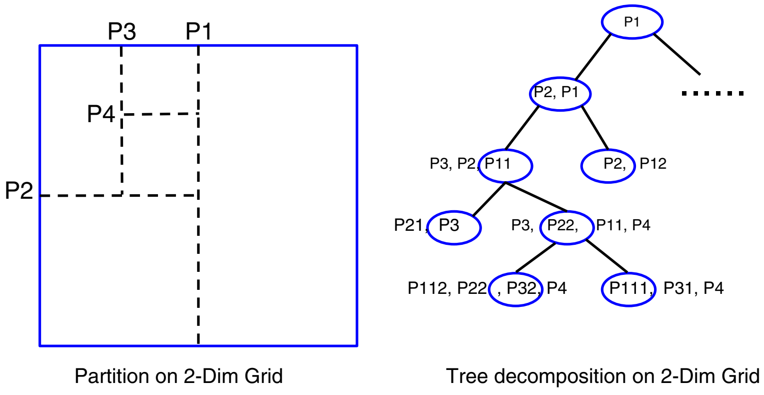

Fig. 1 An illustrative figure for balanced tree decomposition on $G_2(n)$

To the best of our knowledge, there is no rigorous time complexity analysis of the counting perfect matchings problem specifically for grids in the literature. But the exponential space algorithm of [28] (see next paragraph) runs fast on grids. To demonstrate the efficiency of Algorithm 1, we compare it to this and two other natural algorithms.

1. Dynamic Programming Based on Path Decomposition A path decomposition is a special tree decomposition where the underlying tree is a path. A path decomposition with width $2n^{d-1}$ is obtained by putting all vertices with x_1 coordinate equal to j or $j+1$ into the bag of node j, for $j = 0, 1, ..., n-1$. A path decomposition with a smaller pathwidth of n^{d-1} can be obtained as follows. Construct n nodes $p_1, p_2, ..., p_n$ where the bag of p_{j+1} contains all vertices with the x_1 coordinate equal to j, for $j = 0, 1, ..., n-1$. For any p_j, p_{j+1}, start from p_j, add a sequence of nodes by alternating between adding a vertex with $x_1 = j+1$ and deleting its neighbor with $x_1 = j$. The number of nodes increases by a factor of n^{d-1} compared to the first path decomposition. We run the standard dynamic programming on the second path decomposition. This algorithm runs in time $O^*(2^{n^{d-1}})$, however the space complexity is $O^*(2^{n^{d-1}})$. It is of no surprise that it has a better running time than Algorithm 1 due to an extra space usage. We remark that van Rooij et al. [28] give a dynamic programming algorithm for the problem of counting perfect matchings on any tree decomposition of treewidth k with running time $O^*(2^k)$ and space exponential to k.

2. Dynamic Programming Based on Path Decomposition on a Subgrid One way to obtain a polynomial space dynamic programming algorithm is to construct a low pathwidth decomposition on a sufficiently large subgraph. One can then run dynamic programming on this path decomposition and do an exhaustive enumeration on the remaining graph in a similar way as in [4]. To extract from $G_d(n)$ a subgrid of pathwidth $O(\log n)$ (notice that this is the maximum pathwidth for a polynomial space dynamic programming algorithm), we can delete a portion of vertices from $G_d(n)$ to turn a "cube"-shaped grid into a long "stripe" with an $O(\log n)$ cross-section area. It is sufficient to remove $O(n^d/(\log n)^{1/(d-1)})$ vertices. This leads to a polynomial-space algorithm with running time $2^{O(n^d/(\log n)^{1/(d-1)})}$, which is worse than Algorithm 1.

3. Branching Algorithm A naive branching algorithm starting from any vertex in the grid could have time complexity $2^{O(n^d)}$ in the worst case. We analyze a branching algorithm with a careful selection of the starting point. The branching algorithm works by first finding a balanced separator S and partitioning the graph into $A \cup S \cup B$. The algorithm enumerates every subset $X \subseteq S$. A vertex in X either matches a vertex in A or to a vertex in B while vertices in $S \setminus X$ are matched within S. Then the algorithm recurses on A and B. Let $T_d(n)$ be the running time of this branching algorithm on $G_d(n)$. We use the same balanced separator as in the Tree Decomposition Algorithm. We have an asymptotically tight upper bound on the running time of, $T_d(n) \leq 2T_d(\frac{n-|S|}{2}) \sum_{X \subseteq S} 2^{|X|} T_{d-1}(|S \setminus X|)$. We can use any polynomial space algorithm to count perfect matchings on $S \setminus X$. For example using Algorithm 1, since the separator is of size $O(n^{d-1})$, we have $T_{d-1}(|S \setminus X|) = 2^{O(n^{d-2})}$. Therefore, $T_d(n) \leq 2T_d(\frac{n}{2}) \cdot 2^{o(n^{d-1})} \sum_{i=0}^{|S|} \binom{|S|}{i} 2^i = 2T_d(\frac{n}{2}) \cdot 2^{o(n^{d-1})} 3^{|S|}$. We cannot get anything better than $T_d(n) = O^*(3^h)$, i.e. $O^*(3^{\frac{2^d-1}{2^{d-1}-1} n^{d-1}})$, which is worse than Algorithm 1. We remark that this branching algorithm can be viewed as a divide and conquer algorithm on a balanced tree decomposition, which is similar to the algorithm of [23].

4.2 Extensions

Counting Perfect Matchings on General Grids Consider more general grids of dimension d with the ith dimension of length n_i, $1 \leq i \leq d$. Assume $n_1 \geq n_2 \geq \cdots \geq n_d$. Let $\mathcal{V}$ be the volume, i.e., $\mathcal{V} = n_1 \cdots n_d$. We use the Tree Decomposition Algorithm to construct a balanced tree decomposition $\mathcal{T}$ of a general grid and obtain an upper bound on the tree-depth h of $\mathcal{T}$. We always choose a middle hyperplane partitioning the longest dimension as a separator. We define the volume $\mathcal{V}$ of a piece as the number of the vertices not yet included in a previous separator. Likewise, the lengths n_i are measured without including any hyperplanes that have previously served as separators, thus maintaining $\mathcal{V} = n_1 \cdots n_d$. Hence, after one cut, n_1 and the volume $\mathcal{V}$ decrease by at least a factor of 2. When n_1 is even, then it is replaced by $n_1/2$ in one branch and $n_1/2 - 1$ in the other branch. After cutting, the dimensions are reordered to maintain the order $n_1 \geq n_2 \geq \cdots \geq n_d$.

Lemma 2 *Let $1 \leq n_1 \leq \cdots \leq n_d$. Apply the Tree Decomposition Algorithm to the $n_1 \times \cdots \times n_d$ grid with dimension d and volume $\mathcal{V} = n_1 \cdots n_d$ to produce a balanced tree decomposition $\mathcal{T}$. Then the tree-depth of $\mathcal{T}$ is at most $(2(1+\sqrt{2}) + \log_2 \frac{n_1}{n_2})\frac{\mathcal{V}}{n_1}$ (where $n_2 = 1$ for d=1).*

Proof First, we assume $n_1 \leq 2n_2$. By induction on the volume $\mathcal{V}$, we first prove the following modified statement.

Claim: The tree-depth of $\mathcal{T}$ is at most $\frac{cV}{\sqrt{n_1 n_2}}$, for a constant c to be determined later.

Basis: For the 1 vertex grid with $n_1 = \ldots = n_d = 1$ the claim is true for every $c \geq 1$.

Induction step: The Tree Decomposition Algorithm partitions the grid along the first dimension into two sub-grids of roughly half the size. If the two subgrids are of different size, we focus on the larger of the two. Let $n'_1 \leq n'_2 \leq \cdots \leq n'_d$ be the lengths of this sub-grid. $n'_1, \ldots, n'_d$ is obtained from $n_2, \ldots, n_d$ by inserting $n'_i = \lceil \frac{n_1-1}{2} \rceil \leq \frac{n_1}{2}$. We can always choose $i \geq 2$. By the inductive hypothesis, the sub-grid has tree-depth at most $c\frac{V'}{\sqrt{n'_1 n'_2}}$, which we upper bound as follows.

$$\frac{cV'}{\sqrt{n'_1 n'_2}} = \frac{\frac{n'_i}{n_1} cV}{\sqrt{n_2 n'_2}} \leq \frac{\frac{n'_i}{n_1} cV}{\sqrt{n_2 n'_i}} = \frac{\sqrt{\frac{n'_i}{n_1}} cV}{\sqrt{n_2 n_1}} \leq \frac{c}{\sqrt{2}} \frac{V}{\sqrt{n_1 n_2}}.$$

Adding the first separator of size $\frac{V}{n_1}$, we obtain

$$\frac{cV'}{\sqrt{n_1'n_2'}} + \frac{V}{n_1} \leq \left(\frac{c}{\sqrt{2}} + 1\right)\frac{V}{\sqrt{n_1 n_2}}.$$

This upper bound on the tree-depth of $\mathcal{T}$ is at most $\frac{cV}{\sqrt{n_1 n_2}}$ for $c \geq 2 + \sqrt{2}$. This proves the claim.

As we have assumed that $n_1 \leq 2n_2$, we have $\frac{cV}{\sqrt{n_1 n_2}} \leq \frac{\sqrt{2}cV}{n_1}$ implying an upper bound $2(1+\sqrt{2})\frac{\mathcal{V}}{n_1}$ on the tree-depth of $\mathcal{T}$, and proving the lemma for the special case of $n_1 \leq 2n_2$.

The general case follows easily, because we encounter at most $\log_2 \frac{n_1}{n_2}$ separators of size $\frac{\mathcal{V}}{n_1} = n_2 \cdots n_d$ before $n_1 \leq 2n_2$. □

Based on Lemma 2, we give time complexity results for the algorithms discussed in Section 4.1 applied to arbitrary grids. First, h is the only parameter to the running time of Algorithm 1 and the branching algorithm. Algorithm 1 runs in time $O^*(2^{\frac{2(1+\sqrt{2})\mathcal{V}}{n_1}})$ and the branching algorithm runs in time $O^*(3^{\frac{2(1+\sqrt{2})\mathcal{V}}{n_1}})$. The dynamic programming algorithm based on path decomposition on a subgrid has a running time $2^{O\left(\frac{\mathcal{V}}{(\log n_1)^{1/(d-1)}}\right)}$. These three algorithms have polynomial space complexity. For constant d, Algorithm 1 has the best time complexity. For the dynamic programming algorithm based on path decomposition, it runs in time $O^*(2^{\frac{\mathcal{V}}{n_1}})$ but in exponential space.

The result can easily be generalized to the k-nearest-neighbor (kNN) graphs and their subgraphs in d-dimensional space, as it is known that there exists a vertex separator of size $O(k^{1/d}n^{1-1/d})$ which splits the kNN graph into two disjoint parts with size at most $\frac{d+1}{d+2}n$ [25]. More generally, we know that a nontrivial result can be obtained by Algorithm 1 if there exists a balanced separator of the graph G with the following property. Let $s(n')$ be the size of a balanced separator S on any subgraph G' of G of size $n' \leq n$. S partitions the subgraph into two disjoint parts G_1', G_2', such that $S \cup G_i'$ is of size at most cn', for some constant $c \in (0, 1)$, $i = 1, 2$. If there exists a constant $\gamma < 1$, such that for every $n' \leq n$, $s(cn') \leq \gamma s(n')$, then the tree-depth of the resulting tree decomposition is at most $s(n) + \gamma s(n) + \gamma^2 s(n) + \cdots \leq \frac{s(n)}{1-\gamma}$. In this case, the tree decomposition of treewidth k constructed by the Tree Decomposition Algorithm has the tree-depth $h = \Theta(k)$. For $k = \Omega(\log n)$, Algorithm 1 has a similar running time as the standard dynamic programming algorithm but with much better space complexity.

Computing the Matching Polynomial The matching polynomial of a graph G is defined to be $m[G, \lambda] = \sum_{i=0}^{|G|/2} m^i[G]\lambda^i$, where $m^i[G]$ is the number of matchings of size i in graph G. We put the coefficients of $m[G, \lambda]$ into a vector $\mathbf{m}[G]$. The problem is essentially to compute the coefficient vector $\mathbf{m}[G]$.

For every node x in a tree decomposition, let vector $\mathbf{m}_x[X]$ be the coefficient vector of the matching polynomial defined on Y_X. Notice that every entry of $\mathbf{m}_x[X]$

is at most $|E|^{|V|/2}$ and all constants are singletons (and therefore easily computable). $\mathbf{m}_x^0[X] = 1$ and $\mathbf{m}_x^i[X] = 0$ for $i > |X|/2$. The case of x being a forget node follows exactly from Algorithm 1. For any type of tree node x,

- x is a leaf node. $\mathbf{m}_x^i[\emptyset] = 1$ if $i = 0$, or 0 otherwise.
- x is an introduce vertex node. If $v \in X$, $\mathbf{m}_x^i[X] = \mathbf{m}_c^i[X \setminus \{v\}]$. Hence $(\zeta\mathbf{m}_x^i)[X] = 2(\zeta\mathbf{m}_c^i)[X \setminus \{v\}]$ if $v \in X$, or $(\zeta\mathbf{m}_x^i)[X] = (\zeta\mathbf{m}_c^i)[X]$ otherwise.
- x is an auxiliary leaf of a modified introduce edges node. $\mathbf{m}_x^i[X] = 1$ only when $u, v \in X$ and $i = 1$, or $i = 0$. Otherwise it is 0.
- x is a join node. $\mathbf{m}_x^i[X] = \sum_{X' \subseteq X} \sum_{j=0}^{i} \mathbf{m}_{c_1}^j[X']\mathbf{m}_{c_2}^{i-j}[X \setminus X']$.

Counting l-packings Given a universe U of elements and a collection of subsets $\mathcal{S}$ on U, an l-packing is a collection of l disjoint sets. The l-packings problem can be solved in a similar way as computing the matching polynomial. Packing problems can be viewed as matching problems on hypergraphs. Tree decomposition on graphs can be generalized to tree decomposition on hypergraph, where we require every hyperedge to be assigned to a specific bag [17]. A hyperedge is introduced after all vertices covered by this edge are introduced.

Counting Dominating Sets, Counting Set Covers. The set cover problem is given a universe U of elements and a collection of sets $\mathcal{S}$ on U, find a subcollection of sets from $\mathcal{S}$ which covers the entire universe U. The dominating set problem is defined on a graph $G = (V, E)$. Let $U = V$, $\mathcal{S} = \{N[v]\}_{v \in V}$, where $N[v]$ is the union of the set of neighbors of v with $\{v\}$ itself. The dominating set problem is to find a subset of vertices S of V such that $\bigcup_{v \in S} N[v]$ covers V.

The set cover problem can be viewed as a covering problem on a hypergraph, where one selects a collection of hyperedges which cover all vertices. The dominating set problem is then a special case of the set cover problem. If $\mathcal{S}$ is closed under subsets, a set cover can be viewed as a disjoint cover. We only consider the counting set covers problem. For any subset $X \subseteq B_x$, we define $f_x[X]$ to be the number of set covers of Y_X. We have $f_x[X] \leq |U|^{|\mathcal{S}|}$, and all constants are singletons. We omit the recurrence for forget nodes as we can directly apply recurrence Eq. 9 in Algorithm 1. For any node x, $f_x[\emptyset] = 1$.

- x is a leaf node. $f_x[\emptyset] = 1$.
- x is an introduce vertex node. If $v \in X$, $f_x[X] = 0$. If $v \notin X$, $f_x[X] = f_c[X]$.
- x is an auxiliary leaf of a modified introduce hyperedge node introducing hyperedge e. $f_x[X] = 1$ when $X \subseteq e$, and $f_x[X] = 0$ otherwise.
- x is a join node. $f_x[X] = \sum_{X' \subseteq X} f_{c_1}[X']f_{c_2}[X - X']$.

Finally, we point out that our framework has its limitations. First, it cannot be applied to problems where the computation on a join node cannot be formalized as a convolution. The maximum independent set problem is an example. Also it is not known if there is a way to adopt the framework to the Hamiltonian path problem, the counting l-path problems, and the unweighted Steiner tree problem. It seems that for theses problems we need a large storage space to record intermediate results. It is interesting to find more problems which fit in our framework.

5 Conclusion

We study the problem of designing efficient dynamic programming algorithms based on tree decompositions in polynomial space. We show how to construct a modified nice tree decomposition $\mathcal{T}$ and extend the algebraic techniques in [24] to dynamic sets such that we can run the dynamic programming algorithm in time $O^*(2^h)$ and in polynomial space, with h being the tree-depth of a graph [27]. We apply our algorithm to many problems. It is interesting to find more natural graphs with non-trivial modified nice tree decompositions, and to find more problems which fit in our framework.

References

1. Amir, E.: Efficient Approximation for Triangulation of Minimum Treewidth. In: Proceedings of the 17th Conference on Uncertainty in Artificial Intelligence, pp. 7–15 (2001)
2. Amir, E.: Approximation algorithms for treewidth. Algorithmica **56**(4), 448–479 (2010). doi:10.1007/s00453-008-9180-4
3. Björklund, A.: Counting Perfect Matchings as Fast as Ryser. In: 23Rd Annual ACM-SIAM Symposium on Discrete Algorithms, pp. 914–921 (2012)
4. Björklund, A., Husfeldt, T.: Exact algorithms for exact satisfiability and number of perfect matchings. Algorithmica **52**(2), 226–249 (2008)
5. Björklund, A., Husfeldt, T., Kaski, P., Koivisto, M.: Fourier Meets Möbius: Fast Subset Convolution. In: 39Th Annual ACM Symposium on Theory of Computing, pp. 67–74 (2007)
6. Bodlaender, H.L.: Dynamic Programming on Graphs with Bounded Treewidth. In: 15Th International Colloquium on Automata, Languages and Programming, pp. 105–118 (1988)
7. Bodlaender, H.L.: NC-Algorithms for Graphs with Small Treewidth. In: 14Th International Workshop on Graph-Theoretic Concepts in Computer Science, pp. 1–10 (1989)
8. Bodlaender, H.L.: A Linear Time Algorithm for Finding Tree-Decompositions of Small Treewidth. In: 25Th Annual ACM Symposium on Theory of Computing, pp. 226–234 (1993)
9. Bodlaender, H.L.: Discovering Treewidth. In: 31St Conference on Current Trends in Theory and Practice of Computer Science, pp. 1–16 (2005)
10. Bodlaender, H.L., Drange, P.G., Dregi, M.S., Fomin, F.V., Lokshtanov, D., Pilipczuk, M.: An $O(C^k N)$ 5-Approximation Algorithm for Treewidth. In: 54Th Annual IEEE Symposium on Foundations of Computer Science, pp. 499–508 (2013)
11. Bodlaender, H.L., Gilbert, J.R., Kloks, T., Hafsteinsson, H.: Approximating Treewidth, Pathwidth, and Minimum Elimination Tree Height. In: 17Th International Workshop on Graph-Theoretic Concepts in Computer Science, pp. 1–12 (1992)
12. Bouchitté, V., Kratsch, D., Müller, H., Todinca, I.: On treewidth approximations. Discrete Appl. Math. **136**(2-3), 183–196 (2004)
13. Cygan, M., Nederlof, J., Pilipczuk, M., Pilipczuk, M., van Rooij, J.M.M., Wojtaszczyk, J.O.: Solving connectivity problems parameterized by treewidth in single exponential time. In: 52nd Annual IEEE Symposium on Foundations of Computer Science, pp. 150–159 (2011)
14. Feige, U., Hajiaghayi, M., Lee, J.: Improved approximation algorithms for minimum weight vertex separators. SIAM J. Comput. **38**(2), 629–657 (2008). doi:10.1137/05064299X
15. Fomin, F.V., Gaspers, S., Saurabh, S., Stepanov, A.A.: On two techniques of combining branching and treewidth. Algorithmica **54**(2), 181–207 (2009)
16. Fürer, M.: Faster integer multiplication. SIAM J. Comput. **39**(3), 979–1005 (2009)

17. Gottlob, G., Leone, N., Scarcello, F.: Hypertree Decompositions: a Survey. In: Mathematical Foundations of Computer Science, pp. 37–57 (2001)
18. Kenyon, C., Randall, D., Sinclair, A.: Approximating the number of monomer-dimer coverings of a lattice. J. Stat. Phys. **83**(3), 637–659 (1996). doi:10.1007/BF02183743
19. Kloks, T.: Treewidth, Computations and Approximations, vol. 842. Springer (1994)
20. Kneis, J., Mölle, D., Richter, S., Rossmanith, P.: A bound on the pathwidth of sparse graphs with applications to exact algorithms. SIAM J. Discret. Math. **23**(1), 407–427 (2009)
21. Koutis, I.: Faster Algebraic Algorithms for Path and Packing Problems. In: 35Th International Colloquium on Automata, Languages and Programming, pp. 575–586 (2008)
22. Koutis, I., Williams, R.: Limits and Applications of Group Algebras for Parameterized Problems. In: 36Th International Colloquium on Automata, Languages and Programming, pp. 653–664 (2009)
23. Lokshtanov, D., Mnich, M., Saurabh, S.: Planar K-Path in Subexponential Time and Polynomial Space. In: 37Th International Workshop on Graph-Theoretic Concepts in Computer Science, pp. 262–270 (2011)
24. Lokshtanov, D., Nederlof, J.: Saving Space by Algebraization. In: 42nd ACM Symposium on Theory of Computing, pp. 321–330 (2010)
25. Miller, G.L., Teng, S.H., Thurston, W., Vavasis, S.A.: Separators for sphere-packings and nearest neighbor graphs. J. ACM **44**(1), 1–29 (1997)
26. Nederlof, J.: Fast polynomial-space algorithms using inclusion-exclusion. Algorithmica **65**(4), 868–884 (2013)
27. Nešetřil, J., de Mendez, P.O.: Tree-depth, subgraph coloring and homomorphism bounds. Eur. J. Comb. **27**(6), 1022–1041 (2006)
28. van Rooij, J.M.M., Bodlaender, H.L., Rossmanith, P.: Dynamic Programming on Tree Decompositions Using Generalised Fast Subset Convolution. In: 17Th Annual European Symposium on Algorithms, pp. 566–577 (2009)
29. van Rooij, J.M.M., Nederlof, J., van Dijk, T.C.: Inclusion/Exclusion Meets Measure and Conquer. In: 17Th Annual European Symposium on Algorithms, pp. 554–565 (2009)
30. Rota, G.C.: On the foundations of combinatorial theory. i. theory of möbius functions. Zeitschrift fü,r Wahrscheinlichkeitstheorie und Verwandte Gebiete **2**(4), 340–368 (1964)
31. Stanley, R., Rota, G.: Enumerative Combinatorics, vol. 1. Cambridge University Press (2000)
32. Temperley, H., Fisher, M.: Dimer problem in statistical mechanics - an exact result. Philos. Mag. **6**, 1061–1063 (1961)
33. Woeginger, G.J.: Space and Time Complexity of Exact Algorithms: Some Open Problems (Invited Talk). In: 1St International Workshop on Parameterized and Exact Computation, pp. 281–290 (2004)