

Dynamic Meta-Kernelization*

Christian Bertram[†] Deborah Haun[‡] Mads Vestergaard Jensen[†]
Tuukka Korhonen[†]

Abstract

Kernelization studies polynomial-time preprocessing algorithms. Over the last 20 years, the most celebrated positive results of the field have been linear kernels for classical NP-hard graph problems on sparse graph classes. In this paper, we lift these results to the dynamic setting.

As the canonical example, Alber, Fellows, and Niedermeier [J. ACM 2004] gave a linear kernel for dominating set on planar graphs. We provide the following dynamic version of their kernel: Our data structure is initialized with an n -vertex planar graph G in $\mathcal{O}(n \log n)$ amortized time, and, at initialization, outputs a planar graph K with $\text{OPT}(K) = \text{OPT}(G)$ and $|K| = \mathcal{O}(\text{OPT}(G))$, where $\text{OPT}(\cdot)$ denotes the size of a minimum dominating set. The graph G can be updated by insertions and deletions of edges and isolated vertices in $\mathcal{O}(\log n)$ amortized time per update, under the promise that it remains planar. After each update to G , the data structure outputs $\mathcal{O}(1)$ updates to K , maintaining $\text{OPT}(K) = \text{OPT}(G)$, $|K| = \mathcal{O}(\text{OPT}(G))$, and planarity of K .

Furthermore, we obtain similar dynamic kernelization algorithms for all problems satisfying certain conditions on (topological-)minor-free graph classes. Besides kernelization, this directly implies new dynamic constant-approximation algorithms and improvements to dynamic FPT algorithms for such problems.

Our main technical contribution is a dynamic data structure for maintaining an approximately optimal *protrusion decomposition* of a dynamic topological-minor-free graph. Protrusion decompositions were introduced by Bodlaender, Fomin, Lokshtanov, Penninkx, Saurabh, and Thilikos [J. ACM 2016], and have since developed into a part of the core toolbox in kernelization and parameterized algorithms.

*This work was supported by the VILLUM Foundation, Grant Number 54451, Basic Algorithms Research Copenhagen (BARC). T.K was supported by the European Union under Marie Skłodowska-Curie Actions (MSCA), project no. 101206430.

[†]University of Copenhagen, Denmark. {chbe@di.ku.dk, mvje@di.ku.dk, tuko@di.ku.dk}

[‡]Karlsruhe Institute of Technology, Germany. deborah.haun@student.kit.edu

Contents

1	Introduction	1
2	Overview	7
2.1	Dynamic protrusion decomposition	7
2.2	Dynamic kernelization	12
3	Preliminaries	13
3.1	(Hyper)graphs and trees	13
3.2	(Hyper)graph decompositions	15
3.3	Parameterized graph problems and kernelization	19
4	Existence of mergeable root children	21
4.1	Many torso vertices	21
4.2	Proof of Lemma 4.1	27
5	Dynamic local search	31
6	Balancing protrusions	37
6.1	Basic rotations	38
6.2	Balancing subroutines	41
6.3	Proofs for the protrusion-balancing data structure	43
7	Assembling the main data structure	48
8	Dynamic kernelization	53
8.1	The protrusion replacement automaton	53
8.2	The dynamic kernelization data structure	60
9	Conclusions	64
A	Missing proofs	71
A.1	Proofs missing from Section 3	71
A.2	Proofs missing from Section 8	72
A.3	Proofs missing from Section 9	72
B	Tree decomposition automata	73

1 Introduction

The field of kernelization studies polynomial-time preprocessing algorithms for NP-hard problems. In order to achieve worst-case performance guarantees in this setting, one must consider parameterized problems where the hardness of an instance is captured by a parameter. A *kernelization algorithm*, also called simply a *kernel*, for a parameterized problem takes an input (I, k) , runs in time $\text{poly}(|I|, k)$, and outputs an instance (K, k') of the problem so that (1) (K, k') is a yes-instance if and only if (I, k) is a yes-instance, and (2) $|K| + k' \leq f(k)$ for a computable function f [DF99].

It is a classic observation that a problem has a kernel if and only if it is fixed-parameter tractable [CCDF97]. This motivates the notions of a *polynomial kernel* and a *linear kernel*, which restrict the function f in the definition above to be a polynomial (resp. linear) function. Lower bounds for kernels are known under the assumption that $\text{coNP} \not\subseteq \text{NP/poly}$, showing that some natural problems, such as LONGEST PATH parameterized by the length, do not (likely) have polynomial kernels despite being fixed-parameter tractable [BDFH09, FS11]. Similarly, FEEDBACK VERTEX SET has a kernel with $\mathcal{O}(k^2)$ edges [Tho10], but not with $\mathcal{O}(k^{2-\varepsilon})$ edges for any $\varepsilon > 0$, unless $\text{coNP} \subseteq \text{NP/poly}$ [DvM14]. We refer the reader to [FLSZ19] for a recent textbook on kernelization.

In this paper, we design *dynamic* linear kernels for graph problems on sparse graph classes. This means designing data structures that support updating the input graph while simultaneously efficiently maintaining a kernel. To be concrete about our contribution before diving deeper into the literature, let us start by stating a representative example of a result that we obtain. It provides a dynamic linear kernel for DOMINATING SET (parameterized by the solution size) on planar graphs.

Theorem 1.1. *There is a data structure that is initialized with a planar graph G in $\mathcal{O}(|G| \log |G|)$ ¹ amortized time, supports updating G via insertions and deletions of edges and isolated vertices in amortized $\mathcal{O}(\log |G|)$ time per update under the promise that G remains planar, and throughout maintains a graph K so that*

1. K is planar,
2. $\text{OPT}(K) = \text{OPT}(G)$, where $\text{OPT}(\cdot)$ denotes the size of a minimum dominating set, and
3. $|K| \leq \mathcal{O}(\text{OPT}(G))$.

The data structure outputs K at the initialization, and after each update it outputs at most $\mathcal{O}(1)$ updates to K , which consist of insertions and deletions of edges and isolated vertices.

Theorem 1.1 follows from our meta-theorem Theorem 1.3, which provides similar results for any pair of a problem and a graph class that satisfy certain conditions.

The reader may notice that the kernel of Theorem 1.1 does not quite match the definition we provided a couple of paragraphs earlier, but is in fact slightly stronger, working for all values k of the parameter simultaneously. We also highlight the fact that the graph K is updated by (worst-case) $\mathcal{O}(1)$ updates per update to G , which is a non-trivial result under any polynomial running time. It enables efficient chaining of Theorem 1.1 with other dynamic data structures. Another observation is that the data structure of Theorem 1.1 also maintains a constant approximation to the minimum dominating set size due to the inequalities $\text{OPT}(G) \leq |K| \leq \mathcal{O}(\text{OPT}(G))$.

Let us then discuss the literature of linear kernels on sparse graph classes before stating our results in the full generality.

¹We denote by $|G| = |V(G)| + |E(G)|$ the total number of vertices and edges of a graph G .

Linear kernels on sparse graph classes. Among the most influential results in kernelization are linear kernels for NP-hard graph problems parameterized by solution size on sparse graph classes. The first such result was by Alber, Fellows, and Niedermeier [AFN04], who gave a linear kernel for DOMINATING SET on planar graphs with running time $\mathcal{O}(n^3)$. We note that even the existence of any kernel for DOMINATING SET on planar graphs is non-trivial and perhaps surprising, as on general graphs DOMINATING SET is W[2]-complete [DF95] and thus has no kernel under the standard assumption $\text{FPT} \neq \text{W}[2]$.

The result of Alber et al. led to a flurry of linear kernels on planar graphs, for example, for FEEDBACK VERTEX SET [BP08], CYCLE PACKING [BPT08], INDUCED MATCHING [MS07, KPSX11], FULL-DEGREE SPANNING TREE [GNW06], and CONNECTED DOMINATING SET [LMS11] (see also [CFKX07]). Guo and Niedermeier [GN07] gave a framework capturing some of the techniques for kernels on planar graphs, obtaining linear kernels also for CONNECTED VERTEX COVER, EDGE DOMINATING SET, TRIANGLE PACKING, and EFFICIENT DOMINATING SET. Fomin and Thilikos generalized the planar DOMINATING SET kernel to graphs of bounded Euler genus [FT04].

In 2009, all of the aforementioned results were subsumed by a general meta-theorem of Bodlaender, Fomin, Lokshtanov, Penninkx, Saurabh, and Thilikos [BFL⁺09, BFL⁺16]. The theorem states that all problems on graphs of bounded Euler genus that (1) are “quasi-coverable”, and (2) have “finite integer index” (FII) have a linear kernel. The appendix of [BFL⁺16] mentions 31 such problems.

The condition (1) is a technical statement that uses distances on a surface. Its modern equivalent [FLST20] formulation would be that all instances G of the problem admit a treewidth- η -modulator $X \subseteq V(G)$ of size $|X| \leq \mathcal{O}(\text{OPT}(G))$ for a constant η , i.e., a set X such that $\text{tw}(G \setminus X) \leq \eta$.² The condition (2), i.e. FII, means roughly speaking that the problem has a dynamic programming algorithm on graphs of bounded treewidth, such that the integer values on the dynamic programming table are well-behaved.

The main technique introduced by [BFL⁺16] is *protrusion replacement*. A c -protrusion in a graph G is a set $P \subseteq V(G)$ so that $\text{tw}(G[P]) \leq c$ and $|\partial P| \leq c$, where ∂P denotes the vertices in P that have a neighbor outside of P . The parameter c should be seen as a constant that depends on the problem and the graph class. Protrusion replacement means the operation of replacing a protrusion P by a small gadget, while maintaining that the optimum value of the problem stays the same, up to a shifting constant that can be computed while replacing the protrusion. Having FII implies that protrusions can be replaced by constant-size gadgets [BFL⁺16].

The algorithm of [BFL⁺16] works by replacing protrusions by constant-size gadgets until the graph has no more large enough protrusions to make progress. To argue that this results in a kernel, they use *protrusion decompositions*. A (k, c) -protrusion decomposition of a graph G is a pair (T, bag) , where T is a tree rooted at a node r and $\text{bag}: V(T) \rightarrow 2^{V(G)}$ is a function that satisfies

1. for all $uv \in E(G)$ there is $t \in V(T)$ with $u, v \in \text{bag}(t)$,
2. for all $v \in V(G)$, the set $\{t \in V(T) \mid v \in \text{bag}(t)\}$ is non-empty and connected in T ,
3. $|\text{bag}(r)| \leq k$ and the degree of r is $\leq k$, and
4. $|\text{bag}(t)| \leq c$ for all $t \in V(T) \setminus \{r\}$.

In other words, a (k, c) -protrusion decomposition is a tree decomposition (T, bag) , where the root-bag has size and degree $\leq k$ and other bags size $\leq c$. The subtrees rooted at the children of the root form c -protrusions, and thus, a graph that has a (k, c) -protrusion decomposition, but whose

²We denote the treewidth of a graph G by $\text{tw}(G)$. For the definition of treewidth we refer the reader to Section 3.2, and for a comprehensive introduction, see [FLSZ19, Chapter 14].

c -protrusions have size $\mathcal{O}(1)$, has size $\mathcal{O}(k)$. Bodlaender et al. showed that the condition (1) on graphs of bounded genus implies the existence of an $(\mathcal{O}(\text{OPT}(G)), \mathcal{O}(1))$ -protrusion decomposition, implying that replacing protrusions results in a linear kernel.

The machinery of protrusion replacement and protrusion decompositions was later used for even more general kernelization meta-theorems [KLP⁺16, FLST20], for other applications in kernelization [FLMS12, FLST12, FLM⁺15, FLM⁺16, DGvH⁺17, GH0⁺17, FLST18, KST18, GPR⁺21, JW25, LSS⁺25, GRZ25], and for applications in parameterized and approximation algorithms outside of kernelization [FLMS12, JPS⁺14, KLP⁺16, KPP15, CRST18, BST23, GST23].

The meta-theorem of Fomin, Lokshtanov, Saurabh, and Thilikos [FLST20] states that (1) all problems that are “CMSO₂-definable”, “linear-separable”, and “minor-bidimensional” admit linear kernels on minor-free graphs, and (2) all problems that are “CMSO₂-definable”, “linear-separable”, and “contraction-bidimensional” admit linear kernels on apex-minor-free graphs. The problems for which (1) applies include for example CYCLE PACKING and FEEDBACK VERTEX SET. The problems for which (2) applies include additionally (r) -DOMINATING SET, CONNECTED VERTEX COVER, and r -SCATTERED SET. The meta-theorem of Kim, Langer, Paul, Reidl, Rossmanith, Sau, and Sikdar [KLP⁺16] states that all problems that are “linearly treewidth-bounding” and have FII have linear kernels on topological-minor-free graphs. These include for example FEEDBACK VERTEX SET, CHORDAL VERTEX DELETION, and EDGE DOMINATING SET.

The algorithms of both [FLST20] and [KLP⁺16] follow the same idea as the algorithm of [BFL⁺16]: They use FII and employ protrusion replacement, and then argue via treewidth modulators that each G has an $(\mathcal{O}(\text{OPT}(G)), \mathcal{O}(1))$ -protrusion decomposition, certifying that protrusion replacement results in a linear kernel. These algorithms, as well as other purely protrusion-replacement based kernels, can be implemented in linear time by the linear-time protrusion-replacement routine of Fomin, Lokshtanov, Misra, Ramanujan, and Saurabh [FLM⁺15] (see also [FLMS12, FLSZ19]).

Our contribution. The main technical contribution of this paper is to provide a dynamic version of the protrusion replacement technique, resulting in dynamic versions of protrusion-based kernels. While there are some previous works on dynamic kernelization [IO14, CCHM15, AMV20, BHRT22, ACJ⁺24], there are no previous dynamic algorithms based on protrusions nor dynamic versions of the above discussed linear kernels for sparse graph classes.

We provide a dynamic algorithm that maintains a protrusion decomposition of a topological-minor-free graph, so that the first parameter of the decomposition is linear in the minimum size of a treewidth modulator. We use $\mathcal{O}_{\bar{p}}(\cdot)$ to denote the $\mathcal{O}$ -notation that ignores factors that depend on a tuple of parameters $\bar{p}$ and are computable given $\bar{p}$. We denote by $\text{tw-mod}_{\eta}(G)$ the size of a smallest set $X \subseteq V(G)$ so that $\text{tw}(G \setminus X) \leq \eta$. We state here a version of our main theorem omitting certain technical details. The version with the details is stated as Theorem 7.5.

Theorem 1.2. *There is a data structure that is initialized with a graph H , integer η , and an H -topological-minor-free graph G . It supports updating G via insertions and deletions of edges and isolated vertices, under the promise that G remains H -topological-minor-free. It maintains an $(\mathcal{O}_{H,\eta}(\text{tw-mod}_{\eta}(G)), \mathcal{O}_{H,\eta}(1))$ -protrusion decomposition $\mathcal{T}$ of G . The amortized running time of the initialization is $\mathcal{O}_{H,\eta}(|G| \log |G|)$, and the amortized update time is $\mathcal{O}_{H,\eta}(\log |G|)$.*

Furthermore, the data structure provides infrastructure for maintaining dynamic programming procedures on the subtrees of $\mathcal{T}$ rooted at the children of the root. Each update to G causes updates to only $\mathcal{O}_{H,\eta}(1)$ such subtrees, and changes the root-bag only by at most $\mathcal{O}_{H,\eta}(1)$.

The second paragraph of the statement of Theorem 1.2 is an informal explanation of the technical details that are needed for the application to dynamic kernelization.

The parameters of the protrusion decomposition of Theorem 1.2 are optimal by $\mathcal{O}_{H,\eta}(1)$ -factors, since having a (k, η) -protrusion decomposition implies that $\text{tw-mod}_{\eta}(G) \leq k$. Further-

more, topological-minor-free graph classes are the most general subgraph-closed graph classes where a linear relation between treewidth modulators and protrusion decompositions holds (see [Proposition 9.1](#)), so the restriction to topological-minor-free graphs is justified.

To the best of our knowledge, the initialization algorithm³ of [Theorem 1.2](#) is the first algorithm to explicitly compute a protrusion decomposition with approximately optimal parameters in near-linear time, without a given treewidth-modulator. Previously, an $\mathcal{O}_{H,\eta}(|G|^2)$ time algorithm was given by Kim, Serna, and Thilikos [[KST18](#)] (in the same setting of H -topological-minor-free graphs and $\mathcal{O}_{H,\eta}(1)$ -approximation). However, most of the previous kernels based on protrusion replacement do not use the decomposition explicitly, but instead employ iterative protrusion replacement, which can be implemented in linear time [[FLM⁺15](#)].

The data structure of [Theorem 1.2](#) can be seen as a generalization of the recent dynamic treewidth data structure of Korhonen [[Kor25a](#)]. Applying it to graphs of treewidth $\leq \eta$ (which exclude $K_{\eta+2}$ as a topological-minor and have $\text{tw-mod}_\eta(G) = 0$) recovers the result of [[Kor25a](#)], but with a significantly higher dependency on η in both the running time and the width. This implies that the factor $\log |G|$ in the amortized update time is optimal, as it is required (unconditionally) even for maintaining dynamic forests [[PD06](#)].

Dynamic meta-kernelization. We then present our kernelization meta-theorem, which provides dynamic kernelization algorithms similar to [Theorem 1.1](#) for a large class of problems. For that, we need a couple of definitions.

By CMSO₂ we mean the Counting Monadic Second-Order logic on graphs (see e.g. [[FLSZ19](#), Section 14.5]). We say that a graph class $\mathcal{G}$ is CMSO₂-definable if there is a CMSO₂-sentence Φ so that $G \models \Phi$ if and only if $G \in \mathcal{G}$. Most of the natural graph classes are CMSO₂-definable. A graph class $\mathcal{G}$ excludes a topological minor if there exists a graph H so that no graph in $\mathcal{G}$ contains H as a topological minor. Classes excluding a topological minor include classes excluding a minor, such as the planar graphs, but also the graphs of bounded degree.

We consider parameterized graph problems Π that are either minimization or maximization problems, and denote by $\text{OPT}_\Pi(G)$ the smallest (resp. largest) value k so that (G, k) is a yes-instance.⁴ A problem Π is *linearly treewidth-bounding* on a graph class $\mathcal{G}$ if there is a constant η so that for all $G \in \mathcal{G}$ we have $\text{tw-mod}_\eta(G) \leq \mathcal{O}(\text{OPT}_\Pi(G))$. For example, for DOMINATING SET on planar graphs we can take $\eta = 2$ [[FLSZ19](#)]. The statement of [Theorem 1.3](#) uses also the definition of FII, which we already introduced informally, and which will be defined in [Section 3.3](#).

Theorem 1.3. *Let $\mathcal{G}$ be a CMSO₂-definable graph class that excludes a topological minor, and Π a parameterized graph problem that is linearly treewidth-bounding on $\mathcal{G}$ and has FII.*

There is a data structure that is initialized with a graph $G \in \mathcal{G}$ in $\mathcal{O}(|G| \log |G|)$ time, supports updating G via insertions and deletions of edges and isolated vertices in amortized $\mathcal{O}(\log |G|)$ time per update under the promise that G remains in $\mathcal{G}$, and throughout maintains a graph K and a non-negative integer Δ so that

1. $K \in \mathcal{G}$,
2. $\text{OPT}_\Pi(K) + \Delta = \text{OPT}_\Pi(G)$, and
3. $|K| \leq \mathcal{O}(\text{OPT}_\Pi(G))$.

The data structure outputs (K, Δ) at the initialization, and after each update it outputs at most $\mathcal{O}(1)$ updates to (K, Δ) , which consist of insertions and deletions of edges and isolated vertices to/from K , and (unbounded) changes of Δ .

³The initialization algorithm simply inserts vertices and edges one-by-one into the data structure.

⁴For the purpose of stating [Theorem 1.3](#), let us assume that such k exists. A more precise form of [Theorem 1.3](#) is given as the pair of [Lemmas 8.16](#) and [8.17](#).

Now, for example [Theorem 1.1](#) follows by taking $\mathcal{G}$ to be the class of planar graphs, Π the DOMINATING SET problem, and using the known fact that DOMINATING SET is linearly treewidth-bounding on planar graphs [\[FLST20\]](#). More generally, we could replace $\mathcal{G}$ by any apex-minor-free CMSO₂-definable graph class [\[FLST20\]](#), for example, the graphs of Euler genus at most 10.

For most of the natural problems such as DOMINATING SET, we can get rid of the shifting-constant Δ by encoding it in K via gadgets. In particular, from the fact that K changes by at most $\mathcal{O}(1)$ updates per update to G , it follows that the value of Δ also changes by at most $\mathcal{O}(1)$, so we can within the same bounds maintain a set of Δ additional isolated vertices in K .

The algorithm of [Theorem 1.3](#) applies to all problems to which the meta-theorems of Fomin et al. [\[FLST20, Theorem 1.1\]](#) and Kim et al. [\[KLP⁺16, Theorem 1\]](#) apply. This also encompasses the results of Bodlaender et al. about linear kernels [\[BFL⁺16, Theorem 1.3\]](#). We give a list of concrete problems to which [Theorem 1.3](#) applies in [Section 9](#).

The amortized update time $\mathcal{O}(\log |G|)$ in [Theorem 1.3](#) is optimal for some problems captured by it. In particular, the result of Pătraşcu and Demaine [\[PD06\]](#) implies that the problem of maintaining whether a planar graph is a forest or contains a cycle requires $\mathcal{O}(\log |G|)$ (amortized and randomized) update time, unconditionally. This implies that $\mathcal{O}(\log |G|)$ update time is required for any problem Π on planar graphs whose optimum value is a constant c if and only if G is a forest, for example, CYCLE PACKING or FEEDBACK VERTEX SET.

The statement of [Theorem 1.3](#) is non-constructive, in particular, it is not clear how the problem Π would even be described. The proof is also inherently non-constructive because of the non-constructive nature of the definition of FII and the related protrusion-replacement machinery. However, it can be made constructive for large classes of concrete problems by using the techniques introduced by Garnero, Paul, Sau, and Thilikos [\[GPST15, GPST19\]](#).

Applications outside of kernelization. [Theorem 1.3](#) has some direct applications outside of kernelization. As the first example, we observe that it improves the update times of many dynamic parameterized algorithms from $2^{\mathcal{O}(\sqrt{k})} \log n$ to $2^{\mathcal{O}(\sqrt{k})} + \mathcal{O}(\log n)$. In particular, Korhonen [\[Kor25a\]](#) observed that his dynamic treewidth data structure gives a dynamic algorithm for maintaining whether an n -vertex planar graph contains a dominating set of size $\leq k$ in $2^{\mathcal{O}(\sqrt{k})} \log n$ amortized update time. Plugging in [Theorem 1.3](#) (in fact, [Theorem 1.1](#)), directly improves this to an amortized update time of $2^{\mathcal{O}(\sqrt{k})} + \mathcal{O}(\log n)$. Similar effect happens to a large class of parameterized problems on (topological-)minor-free graph classes.

Another direct application is to dynamic approximation algorithms. We observe that if a problem Π satisfies that $\text{OPT}_{\Pi}(G) \leq \mathcal{O}(|G|)$ for all G , then the data structure of [Theorem 1.3](#) directly maintains a constant-factor approximation of $\text{OPT}_{\Pi}(G)$, because of the inequalities $\text{OPT}_{\Pi}(G) \leq \mathcal{O}(|K|) + \Delta \leq \mathcal{O}(\text{OPT}_{\Pi}(G))$. To the best of our knowledge, this is the first dynamic constant-factor approximation algorithm for many problems captured by [Theorem 1.3](#), for example, for FEEDBACK VERTEX SET on planar graphs.

Previously, Korhonen, Nadara, Pilipczuk, and Sokołowski [\[KNPS24\]](#) gave dynamic $(1 + \varepsilon)$ -approximation algorithms for WEIGHTED INDEPENDENT SET on apex-minor-free graphs and WEIGHTED DOMINATING SET on bounded-degree minor-free graphs. They achieve $f(\varepsilon) \cdot n^{o(1)}$ amortized update time by implementing a dynamic version of the Baker’s scheme [\[Bak94\]](#). Dynamic approximation algorithms are well-studied for MATCHING and VERTEX COVER, see e.g. [\[Sol16, BHI18, BKS24\]](#).

Our techniques. Our main technical contribution is [Theorem 1.2](#), which implies [Theorem 1.3](#) via a dynamic implementation of the known protrusion replacement machinery. The basic blueprint behind the data structure of [Theorem 1.2](#) is inspired by the dynamic treewidth data structure of Korhonen [\[Kor25a\]](#). In particular, the subtrees of bounded treewidth are maintained

by the same routine as in [Kor25a], and the whole protrusion decomposition satisfies the same key invariant of “downwards well-linkedness” as the tree decomposition maintained by [Kor25a]. The difference is that now the root-bag of the decomposition can have large size, and in particular, its size should be maintained to be approximately the same as the minimum treewidth- η -modulator.

It is easy to incorporate edge insertions/deletions to this structure so that they increase the size of the root-bag by a constant in each operation. Therefore, the main challenges are to:

1. show that if the root-bag grows too large compared to the optimum treewidth- η -modulator, then it can be reduced by chopping off a small part to the bounded-treewidth subtrees, and
2. implement this chopping within $\mathcal{O}(\log n)$ amortized update time and $\mathcal{O}(1)$ changes to the root per update.

Both of the parts (1) and (2) are non-trivial. The part (1) yields an essentially new type of an algorithm for constructing a protrusion decomposition, which constructs a protrusion decomposition by iteratively making the root-bag smaller, unlike the previous algorithms, which proceed by starting with a treewidth- η -modulator and then making the root-bag a superset of it. The proof of (1) uses graph-theoretical techniques, requiring topological-minor-freeness and heavily relying on the “downwards well-linkedness” of the decomposition. The part (2) requires a careful dynamic implementation of a local search procedure for finding large sets of vertices with small neighborhoods.

Related work on dynamic kernelization. So far we omitted the discussion on previous dynamic kernelization algorithms, so let us review them here. Let us focus only on dynamic polynomial kernels, and omit the larger body of work on dynamic FPT algorithms (see e.g. [Bod93, DT13, DKT14, AMV20, CCD⁺21, OPR⁺23, KMN⁺23, MPZ24, Kor25a, MPS25]).

Iwata and Oka [IO14] gave dynamic kernels for VERTEX COVER and CLUSTER VERTEX DELETION. The kernel for VERTEX COVER has $\mathcal{O}(k^2)$ update time and size $\mathcal{O}(k^2)$, while the kernel for CLUSTER VERTEX DELETION has $\mathcal{O}(k^8 \log n)$ update time and size $\mathcal{O}(k^5)$.

Alman, Mnich, and Vassilevska Williams [AMV20] improved the update time of the dynamic vertex cover kernelization to $\mathcal{O}(k)$ worst-case and $\mathcal{O}(1)$ amortized. They also gave dynamic polynomial kernels for d -HITTING SET, EDGE DOMINATING SET, and POINT LINE COVER. The kernel for EDGE DOMINATING SET has update time $\mathcal{O}(1)$ and size $\mathcal{O}(k^2)$, while the kernels for d -HITTING SET and POINT LINE COVER have update times and sizes $\text{poly}(k)$.

An improved dynamic kernel for d -HITTING SET, along with a dynamic kernel for SET PACKING, was given by Bannach, Heinrich, Reischuk, and Tantau [BHRT22]. An, Cho, Jang, Jung, Lee, Oh, Shin, Shin, and Song [ACJ⁺24] designed dynamic kernels on unit disk graphs for VERTEX COVER, TRIANGLE HITTING SET, FEEDBACK VERTEX SET, and CYCLE PACKING.

Organization of the paper. The rest of the paper is organized as follows. We start by presenting informal sketches of the proofs of Theorem 1.2 and Theorem 1.3 in Section 2. We present the proofs in detail in Sections 3 to 8. In particular, we start by presenting definitions and preliminaries in Section 3. Then, the proof of the part (1) discussed above is presented in Section 4, and the main part of the proof of (2) in Section 5. Section 6 is dedicated to lifting the dynamic treewidth data structure of [Kor25a] to maintain the bounded-treewidth parts of the protrusion decomposition. Then, in Section 7 we combine the material of Sections 4 to 6 to finish the proof of Theorem 1.2. We prove Theorem 1.3 in Section 8. We conclude with additional remarks and discussion in Section 9.

2 Overview

We first sketch a proof of [Theorem 1.2](#), i.e., explain how we maintain an approximately optimal protrusion decomposition of a topological-minor-free graph. Afterwards, we describe how this data structure can be used for dynamic kernelization, sketching the proof of [Theorem 1.3](#).

2.1 Dynamic protrusion decomposition

We consider a dynamic H -topological-minor-free graph G and a parameter η . Our goal is to maintain, under insertions and deletions of edges and isolated vertices, a $(\mathcal{O}_{H,\eta}(\text{tw-mod}_\eta(G)), \mathcal{O}_{H,\eta}(1))$ -protrusion decomposition of G . This is a rooted tree decomposition (T, bag) , where the root-bag $r \in V(T)$ has size and degree $\mathcal{O}_{H,\eta}(\text{tw-mod}_\eta(G))$, and other bags have size $\mathcal{O}_{H,\eta}(1)$.

We start by describing in [Section 2.1.1](#) the definitions and ideas that are based on those of [\[Kor25a\]](#) (many of which in turn originate from [\[Kor25b\]](#), and further from [\[RS91\]](#)). The more novel parts of our algorithm are described in [Section 2.1.2](#).

2.1.1 The framework and basic operations

Downwards well-linked superbranch decompositions. The key idea of the dynamic treewidth algorithm of [\[Kor25a\]](#) is to maintain a structure called *downwards well-linked superbranch decomposition*. We do the same in this paper, although naturally with different constraints to reflect that the superbranch decomposition should correspond to a protrusion decomposition instead of a tree decomposition of bounded width.

A *superbranch decomposition* of a graph G is a pair $\mathcal{T} = (T, \mathcal{L})$, where T is a rooted tree, in which every non-leaf node has at least two children, and $\mathcal{L}$ is a bijection that maps every leaf of T to an edge of G . This is similar to the classic definition of a branch decomposition [\[RS91\]](#), but allowing nodes of degree higher than three. For a node $t \in V(T)$, we denote by $\mathcal{L}[t] \subseteq E(G)$ the set of edges of G that are associated by $\mathcal{L}$ with leaves in the subtree of T below t .

The *boundary* of an edge set $A \subseteq E(G)$, denoted by $\text{bd}(A) \subseteq V(G)$, is the set of vertices that are incident to edges in both A and $E(G) \setminus A$. We denote the size of the boundary by $\lambda(A) = |\text{bd}(A)|$. The function $\lambda: 2^{E(G)} \rightarrow \mathbb{Z}_{\geq 0}$ is *symmetric and submodular*, meaning that (1) $\lambda(A) = \lambda(E(G) \setminus A)$ and (2) $\lambda(A \cup B) + \lambda(A \cap B) \leq \lambda(A) + \lambda(B)$ for all $A, B \subseteq E(G)$ [\[RS91\]](#).

A set $A \subseteq E(G)$ of edges is *well-linked* if for every bipartition (A_1, A_2) of A it holds that $\lambda(A_1) \geq \lambda(A)$ or $\lambda(A_2) \geq \lambda(A)$ (this is called *robust* in [\[RS91\]](#)). The *well-linked number* $\text{wl}(A)$ of a set $A \subseteq E(G)$ is the maximum of $\lambda(A')$ over well-linked subsets $A' \subseteq A$. A superbranch decomposition is *downwards well-linked* if for every node $t \in V(T)$, the set $\mathcal{L}[t]$ is well-linked.

The intuitive reason why well-linkedness is a powerful notion in our context is the following three properties [\[Kor25b, Kor25a\]](#):

1. If $A \subseteq E(G)$ is a well-linked set and $B \subseteq E(G)$, then either $\lambda(B \cup A) \leq \lambda(B)$ or $\lambda(B \setminus A) \leq \lambda(B)$. In particular, well-linked sets are perfectly “uncrossable”.
2. Every set $A \subseteq E(G)$ can be partitioned into at most $2^{\lambda(A)}$ well-linked subsets.
3. For all $A \subseteq E(G)$, it holds that $\text{wl}(A) = \Theta(\text{tw}(G[A]))$.

There is also a fourth powerful property, called the “transitivity” of well-linkedness in [\[Kor25b, Kor25a\]](#), but which requires more complex definitions to state, so we omit it for now.

From superbranch to protrusion decompositions. There is a natural way to relate superbranch decompositions to tree decompositions, for which we use the following definition of an *adhesion*. For an edge $tp \in E(T)$ of a superbranch decomposition between a node t and

its parent p , the adhesion at tp is the set $\text{adh}(tp) = \text{bd}(\mathcal{L}[t])$. Now, if $(T, \mathcal{L})$ is a superbranch decomposition of a graph G , we construct a function $\text{bag}: V(T) \rightarrow 2^{V(G)}$ so that (T, bag) is a tree decomposition of G as follows. For a leaf-node ℓ with $\mathcal{L}(\ell) = uv$, we set $\text{bag}(\ell) = \{u, v\}$, and for a non-leaf-node t , we set $\text{bag}(t) = \bigcup_{s \in N(t)} \text{adh}(st)$, where $N(t)$ denotes the neighbors of t . (This does not quite work if G contains isolated vertices, but let us ignore that for now.) We observe that (T, bag) is an $(\mathcal{O}_{H,\eta}(\text{tw-mod}_\eta(G)), \mathcal{O}_{H,\eta}(1))$ -protrusion decomposition of G if

- a. every adhesion has size $\mathcal{O}_{H,\eta}(1)$,
- b. the root-node has degree $\mathcal{O}_{H,\eta}(\text{tw-mod}_\eta(G))$, and
- c. every non-root-node has degree $\mathcal{O}_{H,\eta}(1)$.

This follows from the fact that the size of $\text{bag}(t)$ is bounded by the product of the degree of t and the maximum adhesion size. Instead of maintaining the conditions of [Items a to c](#) directly, our goal is to maintain a superbranch decomposition $\mathcal{T} = (T, \mathcal{L})$ satisfying that

- 1. $\mathcal{T}$ is downwards well-linked,
- 2. for all non-root $t \in V(T)$, $\text{wl}(\mathcal{L}[t]) \leq \mathcal{O}_{H,\eta}(1)$,
- 3. the root-node has degree $\mathcal{O}_{H,\eta}(\text{tw-mod}_\eta(G))$,
- 4. every non-root-node has degree $\mathcal{O}_{H,\eta}(1)$, and
- 5. T has depth $\mathcal{O}_{H,\eta}(\log |G|)$.

The conditions of [Items 1 and 2](#) imply the condition of [Item a](#) by the definition of well-linked number and downwards well-linkedness. Thus, any protrusion decomposition satisfying [Items 1 to 4](#) also satisfies [Items a to c](#), and thus corresponds to a $(\mathcal{O}_{H,\eta}(\text{tw-mod}_\eta(G)), \mathcal{O}_{H,\eta}(1))$ -protrusion decomposition. The logarithmic-depth requirement of [Item 5](#) is for efficient dynamic maintenance of the superbranch decomposition.

Basic maintenance of the superbranch decomposition. The idea of our data structure is that we first implement the procedures for inserting and deleting edges in an “easy” way that increases the degree of the root by $\mathcal{O}_{H,\eta}(1)$ per operation but maintains the other invariants. Then, the hard part of our data structure is a procedure that decreases the degree of the root whenever it is too large compared to $\text{tw-mod}_\eta(G)$. This procedure will be run after every update to keep the degree of the root controlled.

We postpone the hard part to [Section 2.1.2](#), and start here with the “easy” procedure for inserting and deleting edges. At this point, we need to reveal the technical detail that the superbranch decomposition is not actually a superbranch decomposition of G , but of the hypergraph $\mathcal{H}(G)$ that has vertex set $V(\mathcal{H}(G)) = V(G)$, and in addition to the normal edges $\{u, v\}$ for all $uv \in E(G)$, contains singleton edges $\{v\}$ for all $v \in V(G)$. Our definitions extend naturally to hypergraphs, and later we will also use hypergraphs with edges of size more than 2.

We use the tree-rotation techniques from [\[Kor25a\]](#) for maintaining the subtrees below the root. A key subroutine that we implement with those techniques is a procedure that “rotates up” a specified set of leaves of the decomposition. In particular, given a set A of hyperedges with $|A| = \mathcal{O}(1)$, the procedure uses tree rotations to transform $(T, \mathcal{L})$ so that

- 1. the leaves corresponding to A become children of the root,
- 2. the degree of the root increases by $\mathcal{O}_{H,\eta}(1)$, and
- 3. all other invariants are maintained.

This subroutine runs in $\mathcal{O}_{H,\eta}(\log |G|)$ amortized time.

Now, to insert an edge between vertices u and v , we use the subroutine with $A = \{\{u\}, \{v\}\}$, and observe that after $\{u\}$ and $\{v\}$ are children of the root, inserting the hyperedge $\{u, v\}$ while maintaining the invariants (but increasing the root-degree by 1) is trivial, by simply adding a leaf corresponding to it as another child of the root. Similarly, to delete an edge uv , we use the procedure with $A = \{\{u, v\}, \{u\}, \{v\}\}$, after which the deletion becomes similarly straightforward. In both cases, the degree of the root increases by $\mathcal{O}_{H,\eta}(1)$ because of the rotating-up subroutine.

The reason why it is essential that also $\{u\}$ and $\{v\}$ are children of the root while inserting/deleting $\{u, v\}$ is that this ensures that the boundaries $\text{bd}(\mathcal{L}[t])$ stay unchanged in the decomposition. Otherwise, the boundaries $\text{bd}(\mathcal{L}[t])$ could abruptly change globally throughout the decomposition, potentially ruining downwards well-linkedness.

2.1.2 Controlling the root degree

Let us then turn to the hard part of our data structure, namely, the procedure for controlling the degree of the root. We will decrease the degree of the root by finding a set of at least 2 but at most $\mathcal{O}_{H,\eta}(1)$ subtrees rooted at children of the root, and combining them into one subtree rooted at a new child of the root. This combination procedure will be done in a straightforward manner, in particular, by simply adding a new child and moving the subtrees to be rooted under that child instead of the root.

Denote the children of the root corresponding to such subtrees by $C = \{c_1, c_2, \dots, c_h\}$, and denote by $\mathcal{L}[C] = \bigcup_{c_i \in C} \mathcal{L}[c_i]$ the set of hyperedges corresponding to leaves below C . We need that

1. $|C| \geq 2$ so that the degree of the root actually decreases,
2. $|C| \leq \mathcal{O}_{H,\eta}(1)$ so that the degree of the new node is bounded,
3. $\mathcal{L}[C]$ is well-linked so that the decomposition stays downwards well-linked, and
4. $\text{wl}(\mathcal{L}[C]) \leq \mathcal{O}_{H,\eta}(1)$ so that the well-linked number of the subtrees stays bounded.

Furthermore, the bounds hidden by $\mathcal{O}_{H,\eta}(\cdot)$ above should not depend on the current parameters of the decomposition, but only on the original parameters H and η , in order to not make the parameters of the decomposition gradually worse throughout updates. Decreasing the degree by combining such a set of children C maintains all other invariants of the decomposition except may increase the depth by one because of the new subtree. However, the new subtree can be balanced in amortized $\mathcal{O}_{H,\eta}(\log |G|)$ time with the techniques from [Kor25a].

It follows that now, the two main challenges are to show that

1. whenever the degree of the root is too large compared to $\text{tw-mod}_\eta(G)$, such a set of children C exists, and
2. in that case, we can also find C efficiently.

For both of these two challenges, the key definition we need is that of the *torso hypergraph*. Let t be a node of a superbranch decomposition. The torso of t , denoted by $\text{torso}(t)$, is the hypergraph with the vertex set $V(\text{torso}(t)) = \bigcup_{s \in N(t)} \text{adh}(st)$, and having the hyperedge $\text{adh}(st)$ for every $s \in N(t)$. Note that the same hyperedge can occur multiple times if $\text{adh}(s_1t) = \text{adh}(s_2t)$ for $s_1 \neq s_2$, and we indeed treat $\text{torso}(t)$ as a “multi”-hypergraph, thinking about the hyperedge set of $\text{torso}(t)$ as consisting of labels e_s for all $s \in N(t)$, together with a mapping that maps each label e_s to the set $\text{adh}(st)$.

Let r be the root node and $A \subseteq E(\text{torso}(r))$ a subset of hyperedges of its torso, which naturally corresponds to a set of children C_A of the root. We denote by $A \triangleright \mathcal{T} = \mathcal{L}[C_A] \subseteq E(\mathcal{H}(G))$ the

set of hyperedges corresponding to the leaves under C_A . The “transitivity of well-linkedness” from [Kor25b, Kor25a] tells that $A \triangleright \mathcal{T}$ is well-linked in $\mathcal{H}(G)$ if and only if A is well-linked in $\text{torso}(r)$. Now, the task of finding the required set of children C translates into finding a set $A \subseteq E(\text{torso}(r))$ so that

1. $2 \leq |A| \leq \mathcal{O}_{H,\eta}(1)$,
2. A is well-linked in $\text{torso}(r)$, and
3. $\text{wl}(A \triangleright \mathcal{T}) \leq \mathcal{O}_{H,\eta}(1)$.

We can in fact simplify these three conditions even further. The combination of Items 2 and 3 implies that $\lambda(A)$ should be bounded by $\mathcal{O}_{H,\eta}(1)$. By using the fact that any set $A \subseteq E(\text{torso}(r))$ can be partitioned into at most $2^{\lambda(A)}$ well-linked sets, we can relax the condition that A is well-linked into a condition that A is large enough compared to $2^{\lambda(A)}$ – then a postprocessing routine can find a well-linked subset of A . We end up with the conditions

1. $2^{\lambda(A)} < |A| \leq \mathcal{O}_{H,\eta}(1)$ and
2. $\text{wl}(A \triangleright \mathcal{T}) \leq \mathcal{O}_{H,\eta}(1)$.

Existence of a mergeable subset. Let us then get to the proof that such a set $A \subseteq E(\text{torso}(r))$ exists if the root degree is too large. Note that the root degree equals $|E(\text{torso}(r))|$.

The first ingredient is a proof that $|E(\text{torso}(r))| \leq \mathcal{O}_{H,\eta}(|V(\text{torso}(r))|)$. This uses the H -topological-minor-freeness of G and the downwards well-linkedness, in particular, if $\text{torso}(r)$ would be too dense, we could realize this density as a topological minor via downwards well-linkedness. Another case is that $\text{torso}(r)$ contains a lot of hyperedges with exactly the same vertex set, but in that case the desired conclusion is easy to achieve by just selecting a subset of them. Therefore, we can assume that $V(\text{torso}(r))$ is large compared to $\text{tw-mod}_\eta(G)$.

Now, we consider an imaginary “optimal” protrusion decomposition of G . By [KLP⁺16], G indeed has an $(\mathcal{O}_{H,\eta}(\text{tw-mod}_\eta(G)), \mathcal{O}_{H,\eta}(1))$ -protrusion decomposition $\mathcal{T}^* = (T^*, \text{bag}^*)$, rooted at a node r^* . If $V(\text{torso}(r))$ is larger than $\Omega_{H,\eta}(|\text{bag}^*(r^*)| + \Delta(r^*)) = \Omega_{H,\eta}(\text{tw-mod}_\eta(G))$ ⁵, then there exists a child c^* of r^* , so that the subtree of $\mathcal{T}^*$ rooted at c^* contains at least δ vertices from $V(\text{torso}(r))$, and has treewidth ω , where δ and ω are parameters in $\mathcal{O}_{H,\eta}(1)$ with $\omega \ll \delta$. By choosing an appropriate subtree under c^* , we indeed obtain a set $B \subseteq E(\mathcal{H}(G))$ so that

1. $\delta \leq |V(B) \cap V(\text{torso}(r))| \leq 3 \cdot \delta$,⁶
2. $\lambda(B) \leq \omega$, and
3. $\text{wl}(B) \leq \omega$,

where δ is set to be roughly $2^{\Theta(\omega)}$.

The set B would be perfect for us if it would be “uncrossed” with the sets $\mathcal{L}[c]$ for the children c of the root r of our superbranch decomposition, that is, if either $\mathcal{L}[c] \subseteq B$ or $\mathcal{L}[c] \cap B = \emptyset$ would hold for all c . This would allow us to construct the desired set $A \subseteq E(\text{torso}(r))$ by taking $e_c \in A$ whenever $\mathcal{L}[c] \subseteq B$.

Now the natural idea is to use the downwards well-linkedness to uncross the set B with the sets $\mathcal{L}[c]$. It implies that either $\lambda(B \setminus \mathcal{L}[c]) \leq \lambda(B)$ or $\lambda(B \cup \mathcal{L}[c]) \leq \lambda(B)$, so one can always either include or exclude $\mathcal{L}[c]$ without increasing $\lambda(B)$. Furthermore, we show that doing this for

⁵Where $\Delta(r^*)$ denotes the degree of r^* .

⁶Where $V(B) \subseteq V(G)$ denotes all vertices incident to B .

all children c does not drastically affect the quantity $|V(B) \cap V(\text{torso}(r))|$, as long as it is large enough compared to $\lambda(B)$.

However, this uncrossing idea has one issue that makes the proof much more complicated: The quantity $\text{wl}(B)$ can increase if we set $B := B \cup \mathcal{L}[c]$. On the surface this seems to not be a big issue since $\text{wl}(\mathcal{L}[c])$ is anyway bounded by $\mathcal{O}_{H,\eta}(1)$, but here we run into the issue that the “new parameters” of the decomposition should not depend on the “old parameters” of the decomposition, or otherwise they would stack up throughout many operations. In particular, this uncrossing would work fine if we could assume that $\text{wl}(\mathcal{L}[c])$ is small enough compared to the other parameters of the decomposition.

We fix this issue by ensuring that this type of uncrossing can indeed only happen if $\text{wl}(\mathcal{L}[c])$ is really small. In particular, we observe that $\text{wl}(\mathcal{L}[c])$ can be larger than $\mathcal{O}(\eta)$ only for at most $\text{tw-mod}_\eta(G)$ many subtrees of our decomposition, as the modulator must hit all such subtrees. By tuning the existence proof of protrusion decompositions from [KLP⁺16], we can ensure that the set $\text{bd}(\mathcal{L}[c])$ is contained in $\text{bag}^*(r^*)$ for all such subtrees, and therefore by choosing B according to this tuned decomposition (T^*, bag^*) , $V(B)$ can intersect such $\text{bd}(\mathcal{L}[c])$ only in $\text{bd}(B)$. With this, we can ensure that the uncrossing of type $B := B \cup \mathcal{L}[c]$ never happens for such children c .

Thus, skipping many tedious technical details here, the uncrossing idea works out in the end to prove the existence of the desired set $A \subseteq E(\text{torso}(r))$.

Finding a mergeable subset. The next challenge is to find a set $A \subseteq E(\text{torso}(r))$ with (1) $2^{\lambda(A)} < |A| \leq \mathcal{O}_{H,\eta}(1)$ and (2) $\text{wl}(A \triangleright \mathcal{T}) \leq \mathcal{O}_{H,\eta}(1)$ whenever such a set exists, in $\mathcal{O}_{H,\eta}(\log |G|)$ time. We start by observing that the proof above gives us some slack for approximation, in particular, it is enough to be able to either return such a set, or to conclude that there is no such set with (1) replaced by (1') $2^{2 \cdot \lambda(A)} < |A| \leq \mathcal{O}_{H,\eta}(1)$ and (2) replaced by a smaller bound on the well-linked number. In particular, instead of bound the well-linked number of $A \triangleright \mathcal{T}$, we bound $\lambda(A)$ and the *internal treewidth* of $A \triangleright \mathcal{T}$, that is $\text{tw}(G[V(A \triangleright \mathcal{T}) \setminus \text{bd}(A)])$. This is approximately equivalent to bounding the well-linked number.

Let us start by discussing how to even check whether a given set A satisfies these properties. In our data structure, we maintain $\text{torso}(r)$ explicitly, so (1) is easy enough to check in $\mathcal{O}_{H,\eta}(1)$ time. However, checking (2) is not easy, as the internal treewidth involves information not captured by $\text{torso}(r)$. At this point, we note that thanks to the techniques from [Kor25a], we are able to maintain dynamic programming procedures on the subtrees of $\mathcal{T}$ below the root. In particular, since such subtrees correspond to tree decompositions of bounded width, we can use powerful bounded-treewidth machinery to capture information about them. This is used for the application of our data structure for kernelization, but we also use it here to figure out the internal treewidth of $A \triangleright \mathcal{T}$. In particular, we maintain a modified version of the Bodlaender-Kloks dynamic programming procedure for computing treewidth [BK96] on the subtrees, and when given A , combine the information from the subtrees corresponding to A to compute the internal treewidth of $A \triangleright \mathcal{T}$.

Now we know how to efficiently check if a given set A is suitable, but still, there are too many candidates to be brute-forced. Let us sketch an $\mathcal{O}_{H,\eta}(|E(\text{torso}(r))|)$ time algorithm. A set $B \subseteq E(\text{torso}(r))$ is *internally connected* if all vertices in $V(B)$ can reach each other by paths through hyperedges in B but not containing vertices from $\text{bd}(B)$ as internal vertices. Internally connected sets are useful because for parameters k and s and a hyperedge e , there are at most s^k internally connected sets B with $|B| \leq s$, $\lambda(B) \leq k$, and $e \in B$, and they can be listed via a local-search procedure in $s^{\mathcal{O}(k)}$ time.

Now, a suitable set A can be uniquely partitioned into internally connected sets $A_1, \dots, A_h$, so that $\text{bd}(A_i) \subseteq \text{bd}(A)$. By possibly shrinking $|A|$ by a factor of $2^{\lambda(A)}$, which is fine by the slack for approximation, we can assume that $\text{bd}(A_i) = \text{bd}(A)$ for all A_i . We also observe that the internal treewidth of $A \triangleright \mathcal{T}$ is equal to the maximum internal treewidth of $A_i \triangleright \mathcal{T}$. Now, this type of A

is easy enough to find in $\mathcal{O}_{H,\eta}(|E(\text{torso}(r))|)$ time by first enumerating all internally connected sets with specified size and boundary size, checking their internal treewidth, and then grouping them by their boundary. Furthermore, by using the fact that each hyperedge participates in only $\mathcal{O}_{H,\eta}(1)$ such internally connected sets, we can design a dynamic implementation of this algorithm, that works in $\mathcal{O}_{H,\eta}(\log |E(\text{torso}(r))|)$ time per update to $\text{torso}(r)$.

2.2 Dynamic kernelization

The application of our data structure to kernelization follows the high-level idea of protrusion replacement, which was pioneered by Bodlaender et al. [BFL⁺16], and afterwards used in dozens of kernelization algorithms. However, our implementation of protrusion replacement is non-standard in that we explicitly compute a protrusion decomposition and replace each of its protrusions in one shot. The typical protrusion replacement implementations work by repeatedly replacing constant-size protrusions, without actually computing a protrusion decomposition. Nevertheless, this approach of explicitly computing a protrusion decomposition has been used before by Kim, Serna, and Thilikos [KST18], who needed it in the context of kernelization for counting problems.

Let $\mathcal{G}$ be a CMSO₂-definable graph class that excludes a topological minor H , and Π a problem that has FII and is linearly treewidth-bounding on $\mathcal{G}$. For example, we can let $\mathcal{G}$ be the planar graphs and Π the DOMINATING SET problem. Because Π is linearly treewidth-bounding in $\mathcal{G}$, there exists η so that $\text{tw-mod}_\eta(G) \leq \mathcal{O}(\text{OPT}_\Pi(G))$ for all $G \in \mathcal{G}$. Now, it suffices to give a kernelization algorithm parameterized by the parameter $\text{tw-mod}_\eta(G)$ instead of $\text{OPT}_\Pi(G)$. For this, we apply the data structure of Theorem 1.2, initialized with the parameters H and η .

Let $(T, \mathcal{L})$ be the superbranch decomposition maintained by the data structure, corresponding to a $(\mathcal{O}_{H,\eta}(\text{tw-mod}_\eta(G)), \mathcal{O}_{H,\eta}(1))$ -protrusion decomposition (T, bag) . For kernelization, it is in fact more instructive to think about $(T, \mathcal{L})$ than about (T, bag) . For each child c of the root r of T , we consider the *boundaried graph* G_c , having vertex set $V(G_c) = V(\mathcal{L}[c])$, i.e., the union of the hyperedges in $\mathcal{L}[c]$, and edge set $E(G_c)$ being the edges of G in $\mathcal{L}[c]$, and boundary $\text{bd}(\mathcal{L}[c])$.

Now, the idea is to replace the boundaried graph G_c by a *representative* of bounded size. In particular, we replace G_c by a boundaried graph R_c with the same boundary as G_c , but which has bounded size (bounded by a function of the size of the boundary, the problem Π , and the class $\mathcal{G}$). If we denote by G' the graph obtained after this replacement, we want that $\text{OPT}_\Pi(G') = \text{OPT}_\Pi(G) - \Delta_c$, for a non-negative *shifting constant* Δ_c that can be computed from G_c . We also want that $G' \in \mathcal{G}$ if $G \in \mathcal{G}$.

The known properties of FII and CMSO₂ imply that such a representative R_c indeed exists (see e.g. [BFL⁺16]), and can be computed by dynamic programming on the tree decomposition of G_c , together with the shifting constant Δ_c . This dynamic programming can indeed be maintained on the bounded-treewidth subtrees of $(T, \mathcal{L})$ by the dynamic treewidth data structure, so with our data structure we can maintain the representatives R_c and shifting constants Δ_c for all children c of the root.

Now, the kernel is obtained simply as the union of the boundaried graphs R_c (which may overlap at the boundary vertices). As these graphs have constant size, and there are at most $\mathcal{O}_{H,\eta}(\text{tw-mod}_\eta(G))$ of them, the size of the kernel is $\mathcal{O}_{H,\eta}(\text{tw-mod}_\eta(G))$. The final shifting constant Δ is obtained as the sum of the individual shifting constants Δ_c .

This kernel can be maintained with logarithmic update time and with a constant number of changes to it thanks to the properties guaranteed by the data structure of Theorem 1.2. In particular, it guarantees that one update to the graph causes updates only in a constant number of the subtrees rooted at the children c of the root, so the representatives R_c and shifting constants Δ_c need to be updated only for them. Both of them are re-computed in amortized logarithmic update time thanks to the dynamic treewidth data structure we maintain on the subtrees.

3 Preliminaries

In this section, we introduce the definitions and prove some preliminary results. We start with some basic notations. For two integers a and b , we denote by $[a, b]$ the set of all integers i with $a \leq i \leq b$, and by $[a]$ the set $[1, a]$. For a function $f: X \rightarrow Y$ and a set $Z \subseteq X$, we denote by $f|_Z: Z \rightarrow Y$ the restriction of f to Z . For a set S , we denote by $\binom{S}{2}$ the set of all subsets of S of size exactly two. The $\mathcal{O}_{\bar{p}}(\cdot)$ -notation, for a tuple of parameters $\bar{p}$, hides factors that depend on $\bar{p}$ and are computable given it.

3.1 (Hyper)graphs and trees

Graphs. In this paper, all graphs are finite, undirected, and, unless otherwise stated, simple. For a graph G , we denote by $V(G)$ the set of vertices and by $E(G) \subseteq \binom{V(G)}{2}$ the set of edges. We also denote an edge $e = \{u, v\}$ by uv . For a set $A \subseteq E(G)$ of edges, we denote by $V(A) = \bigcup_{uv \in A} \{u, v\}$ the union of their endpoints, and use $V(\{e\}) =: V(e)$. The *size* of a graph G is $|G| = |V(G)| + |E(G)|$.

A *boundaried graph* (G, B, Λ) is a graph G together with a set $B \subseteq V(G)$, called the *boundary* of G , consisting of distinguished *boundary vertices*, and an injective labeling $\Lambda: B \rightarrow \mathbb{Z}_{\geq 1}$. Slightly abusing the notation, we sometimes refer to G as the *boundaried graph* and then denote by $\text{bd}(G)$ the set of boundary vertices. The *label set* of a boundaried graph with boundary B is denoted by $\Lambda(G) = \{\Lambda(v) : v \in B\}$. A graph G is said to be *t-boundaried* if $\Lambda(G) \subseteq [t]$. We denote by $\mathcal{F}$ the set of all boundaried graphs, by $\mathcal{F}_I$ the class of all boundaried graphs with label set I (for $I \subseteq \mathbb{Z}_{\geq 1}$), and by $\mathcal{F}_{\subseteq I}$ the set $\bigcup_{I' \subseteq I} \mathcal{F}_{I'}$.

Minors and minor-free graphs. For a graph G and an edge $e = uv \in E(G)$, we denote by G/e the graph obtained from G by *contracting* the edge e , that is, by identifying u and v and, if necessary, removing loops and multiple edges. A graph H is a *contraction* of G if it can be obtained from G via zero or more edge contractions. A graph H is a *minor* of G if it is a contraction of a subgraph of G . If each contracted edge has at least one endpoint with degree at most two, H is called a *topological minor*. A graph G is called *H-(topological-)minor-free* if H is not a (topological) minor of G . A graph class $\mathcal{G}$ is called *H-(topological-)minor-free* if every graph $G \in \mathcal{G}$ is *H-(topological-)minor-free*, and just *(topological-)minor-free* if such H exists. In these cases can also say that $G/\mathcal{G}$ *excludes* H as a (topological) minor.

A particularly interesting subclass of minor-free graphs is the class of apex-minor-free graphs. A graph G is called an *apex graph* if there exists a vertex $v \in V(G)$ such that $G - v$ is planar. A graph class $\mathcal{G}$ is called *apex-minor-free* if there exists an apex graph H such that $\mathcal{G}$ is *H-minor-free*. For example, planar graphs are apex-minor-free, because they exclude K_5 , which is an apex graph, because K_4 is a planar graph. More generally, graphs of bounded Euler genus are apex-minor-free (see e.g. [Epp00]).

Hypergraphs. Following the definitions of [Kor25a], a hypergraph G consists of a set of vertices $V(G)$, a set of hyperedges $E(G)$, and a mapping $V: E(G) \rightarrow 2^{V(G)}$ that associates each hyperedge with a set of vertices. For a set A of hyperedges, we denote by $V(A) = \bigcup_{e \in A} V(e)$ the union of their vertex sets. The *size* of a hypergraph G is $|G| = |V(G)| + \sum_{e \in E(G)} (|V(e)| + 1)$. The *rank* of a hyperedge e is $|V(e)|$, and the *rank* $\text{rk}(G)$ of a hypergraph G is the maximum rank over all its hyperedges. We allow distinct hyperedges $e_1, e_2 \in E(G)$ with $V(e_1) = V(e_2)$, and call the *multiplicity* of an hyperedge e the number of hyperedges $e' \in E(G)$ with $V(e') = V(e)$. This counts e itself, so the multiplicity of a hyperedge is always at least one. The *multiplicity* of a hypergraph is then the maximum multiplicity over all its hyperedges. For a set $E \subseteq E(G)$, we also say the *multiplicity* of E is the maximum multiplicity of all hyperedges in E .

The *primal graph* $\mathcal{P}(G)$ of a hypergraph G is the graph with vertex set $V(\mathcal{P}(G)) = V(G)$ that has an edge between two vertices $u, v \in V(\mathcal{P}(G))$ if and only if there exists a hyperedge $e \in E(G)$ with both u and v in $V(e)$. For a graph G , the *support hypergraph* $\mathcal{H}(G)$ is the hypergraph with vertex set $V(\mathcal{H}(G)) = V(G)$ and hyperedge set $E(\mathcal{H}(G)) = V(G) \cup E(G)$ such that for every vertex $v \in V(G)$, $V(v) = \{v\}$, and for every edge $uv \in E(G)$, $V(uv) = \{u, v\}$. Note that $\mathcal{P}(\mathcal{H}(G)) = G$, $\mathcal{H}(G)$ has rank at most two and multiplicity one, and $|\mathcal{H}(G)| \leq \mathcal{O}(|G|)$.

We call a (hyper)graph *empty* if it contains no vertices and no (hyper)edges. Given a vertex $v \in V(G)$ of a (hyper)graph G , we denote by $N(v)$ the set of neighbors of v , that is, all vertices $u \neq v$ for which there exists a (hyper)edge $e \in E(G)$ with $u, v \in V(e)$. A vertex v is *isolated* if $N(v) = \emptyset$. Denote by $N[v] = N(v) \cup \{v\}$ the closed neighborhood of v , and by $N^{\text{inc}}(v)$ the set of (hyper)edges that are incident to v . For a set $A \subseteq V(G)$ of vertices or $A \subseteq E(G)$ of (hyper)edges of a (hyper)graph G , we denote by $G[A]$ the sub(hyper)graph of G induced by A . More precisely, when $A \subseteq E(G)$, we define $G[A]$ as the graph with vertex set $V(A)$ and (hyper)edge set A . We use $G \setminus A = G[V(G) \setminus A]$ or $G \setminus A = G[E(G) \setminus A]$, respectively. Similarly, for a set A of vertices not in $V(G)$ or a set A of (hyper)edges not in $E(G)$ but with endpoints in $V(G)$, we denote by $G \cup A$ the (hyper)graph obtained by adding A to $V(G)$ or $E(G)$. For two (hyper)graphs G_1, G_2 , we denote by $G_1 \cup G_2$ the (hyper)graph $G_1 \cup G_2 = (V(G_1) \cup V(G_2), E(G_1) \cup E(G_2))$.

For a set $A \subseteq E(G)$ of (hyper)edges of a (hyper)graph G , we denote its complement by $\bar{A} = E(G) \setminus A$. Then, we denote by $\text{bd}(A) = V(A) \cap V(\bar{A})$ the set of *boundary vertices*, by $\lambda(A) = |\text{bd}(A)|$ the number of boundary vertices, and by $\text{int}(A) = V(A) \setminus \text{bd}(A) = V(A) \setminus V(\bar{A})$ the set of *internal vertices* of A . As proven by, for example, [RS91], the function $\lambda: 2^{E(G)} \rightarrow \mathbb{Z}_{\geq 0}$ is a *symmetric submodular function*, meaning that

- $\lambda(A \cup B) + \lambda(A \cap B) \leq \lambda(A) + \lambda(B)$ for all $A, B \in E(G)$ (submodularity) and
- $\lambda(A) = \lambda(\bar{A})$ for all $A \subseteq E(G)$ (symmetry).

Following the definition of [Kor25b], a non-empty set $A \subseteq E(G)$ of edges in a hypergraph G is called *internally connected* if there is no *bipartition* (B_1, B_2) of A , i.e. $B_1 \cup B_2 = A$ and $B_1 \cap B_2 = \emptyset$, into two non-empty sets B_1 and B_2 with $\text{bd}(B_i) \subseteq \text{bd}(A)$ for both $i \in [2]$. A non-empty set A of hyperedges is called *internally disconnected* if it is not internally connected. An *internal component* of a set of hyperedges $A' \subseteq E(G)$ is then defined as an inclusion-maximal internally connected subset $A \subseteq A'$. Alternatively, one can think of internal connectivity as follows: a set A is internally connected if either it consists of a single hyperedge, or $\mathcal{P}(G)[\text{int}(A)]$ is non-empty, connected, and for every $e \in A$ it holds that $V(e) \setminus \text{bd}(A) \neq \emptyset$.

Trees. A *tree* is an acyclic connected graph. To better distinguish from graphs, we sometimes call the vertices of a tree *nodes*. A *rooted tree* is a tree T , together with a *root* $r \in V(T)$. For the following definitions, let T be a tree rooted at r , and $t \in V(T)$ a node.

The *root path* of t is the unique path between t and r . If $t \neq r$, the *parent* of t is the unique node adjacent to t on the root path of t . A node is a *child* of t if it is adjacent to t and not on the root path of t . We denote by $\text{chd}(t)$ the set of children of t , and by $\Delta(t) = |\text{chd}(t)|$ the number of children of t .

For a set $X \subseteq V(T)$, we denote by $\Delta(X) = \max_{t' \in X} \Delta(t')$ the maximum number of children of any node in X . We let $\Delta(T) = \Delta(V(T))$, and say that T is *binary* if $\Delta(T) \leq 2$.

The *depth* of t is the number of edges on the root path of t . In particular, the depth of the root is always zero. The *depth* of T , denoted by $\text{depth}(T)$, is the maximum depth over all nodes of T .

A *leaf* is a node with no children, and an *internal node* is a node that is not a leaf. We denote by $\text{L}(T)$ the set of leaves of T and by $V_{\text{int}}(T) = V(T) \setminus \text{L}(T)$ the set of all internal nodes of T . We denote by $\text{cl}(t) = \text{chd}(t) \cap \text{L}(T)$ the set of *leaf-children* of t , and for a node $\ell \in \text{cl}(t)$ we refer to ℓ as a *leaf-child* of t .

A node t is an *ancestor* of a node s , and s is a *descendant* of t , if t is on the root path of s . In particular, every node is both an ancestor and a descendant of itself. The set of ancestors of a node t is denoted by $\text{anc}(t)$, and the set of descendants be $\text{desc}(t)$. For a set $X \subseteq V(T)$, we denote by $\text{anc}(X) = \bigcup_{t \in X} \text{anc}(t)$ the set of all ancestors of vertices in X . A *prefix* of a rooted tree is a set $P \subseteq V(T)$ with $P = \text{anc}(P)$, i.e., a connected set of nodes that contains the root.

3.2 (Hyper)graph decompositions

We consider different types of decompositions of (hyper)graphs.

Tree decompositions. A *tree decomposition* of a graph G is a pair $\mathcal{T} = (T, \text{bag})$, where T is a tree and $\text{bag}: V(T) \rightarrow 2^{V(G)}$ is a function mapping each node x of T to a *bag* of vertices such that:

- $V(G) = \bigcup_{x \in V(T)} \text{bag}(x)$.
- For every edge $uv \in E(G)$, there exists a node $x \in V(T)$ such that $\text{bag}(x)$ contains both u and v .
- For every vertex $v \in V(G)$, the set $\{x \in V(T) \mid v \in \text{bag}(x)\}$ induces a connected subtree of T .

The *width* of a tree decomposition $\mathcal{T}$ is the maximum size of a bag minus one. The *treewidth* $\text{tw}(G)$ of a graph G is the minimum width of a tree decomposition of G . For a hypergraph G , the *treewidth* $\text{tw}(G)$ of G is defined as the treewidth of the primal graph of G , i.e., $\text{tw}(G) = \text{tw}(\mathcal{P}(G))$. For a set A of (hyper)edges of a (hyper)graph G , we define the *treewidth* $\text{tw}(A) = \text{tw}(G[A])$, and the *internal treewidth* $\text{itw}(A) = \text{tw}(G[\text{int}(A)])$. Similarly, for a boundaried graph G , we define the *internal treewidth* to be $\text{itw}(G) = \text{tw}(G \setminus \text{bd}(G))$.

For a graph G and an integer η , a set $X \subseteq V(G)$ is called a *treewidth- η -modulator* if $\text{tw}(G \setminus X) \leq \eta$. We denote the size of a smallest treewidth- η -modulator of G by $\text{tw-mod}_\eta(G)$.

A *rooted tree decomposition* is a tree decomposition $\mathcal{T} = (T, \text{bag})$, where the tree T is a rooted tree. For a rooted tree decomposition $\mathcal{T} = (T, \text{bag})$, given a node t and its parent p , we call the set $\text{adh}(tp) = \text{bag}(t) \cap \text{bag}(p)$ the *adhesion* of the edge tp . An *annotated tree decomposition* of a graph G is a triple $(T, \text{bag}, \text{edges})$ so that

- T is a binary tree,
- $\text{bag}: V(T) \rightarrow 2^{V(G)}$ is a function so that (T, bag) is a tree decomposition of G , and
- $\text{edges}: V(T) \rightarrow 2^{E(G)}$ is a function that maps each node $t \in V(T)$ to the set $\text{edges}(t)$ that contains the edges $uv \in E(G)$, for which t is the unique smallest-depth node with $u, v \in \text{bag}(t)$.

Note that G and (T, bag) define the annotated tree decomposition $(T, \text{bag}, \text{edges})$ uniquely, and conversely, $(T, \text{bag}, \text{edges})$ defines G . Given a set $A \subseteq V(T)$, the restriction of $(T, \text{bag}, \text{edges})$ to A , denoted by $(T, \text{bag}, \text{edges}) \upharpoonright_A$, is the tuple $(T[A], \text{bag} \upharpoonright_A, \text{edges} \upharpoonright_A)$, where $\text{bag} \upharpoonright_A$ and $\text{edges} \upharpoonright_A$ are the restrictions of bag and edges to A , respectively.

Let G be a graph together with an annotated tree decomposition $\mathcal{T} = (T, \text{bag}, \text{edges})$ of width $t-1$. For a node $x \in V(T)$, we denote by T_x the subtree of T rooted at x , $V_x = \bigcup_{y \in \text{desc}(x)} \text{bag}(y)$, $E_x = \bigcup_{y \in \text{desc}(x)} \text{edges}(y)$, and $G_x = (V_x, E_x)$. Then, the triple $(G_x, \text{bag}(x), \Lambda)$ is a t -boundaried graph for any injective labeling $\Lambda: \text{bag}(x) \rightarrow [t]$. Given a boundaried graph (G, B, Λ) , a *boundaried tree decomposition* of (G, B, Λ) is a rooted tree decomposition, where B is contained in the root bag.

A rooted tree decomposition $\mathcal{T} = (T, \text{bag})$ of a graph G is *nice* if every node $t \in V(T)$ is one of the following types:

- **Leaf:** t is a leaf of T ,
- **Forget:** t has exactly one child s and there is a vertex $v \in V(G)$ (called the *forgotten vertex*) such that $\text{bag}(t) = \text{bag}(s) \setminus \{v\}$,
- **Introduce:** t has exactly one child s and there is a vertex $v \in V(G)$ (called the *introduced vertex*) such that $\text{bag}(t) = \text{bag}(s) \cup \{v\}$, or
- **Join:** t has exactly two children s_1, s_2 and $\text{bag}(t) = \text{bag}(s_1) = \text{bag}(s_2)$.

Protrusion decompositions. For integers k and c , a (k, c) -*protrusion decomposition* is a rooted tree decomposition $\mathcal{T} = (T, \text{bag})$ such that

- $|\text{bag}(r)| \leq k$,
- $\Delta(r) \leq k$, and
- $|\text{bag}(t)| \leq c$ for every non-root node $t \in V(T) \setminus \{r\}$.

For each root child $t \in \text{chd}(r)$, the tree decomposition $(T_t, \text{bag} \upharpoonright_{V(T_t)})$ is called a *protrusion*.

Well-linkedness. Let G be a hypergraph. A set $A \subseteq E(G)$ of hyperedges is called *well-linked* if for every bipartition (A_1, A_2) of A , it holds that $\lambda(A_1) \geq \lambda(A)$ or $\lambda(A_2) \geq \lambda(A)$. Equivalently, a set $A \subseteq E(G)$ is well-linked if for any two sets $B_1, B_2 \subseteq \text{bd}(A)$ of the same size, possibly overlapping, there are $|B_1| = |B_2|$ vertex-disjoint paths in $\mathcal{P}(G[A])$ between B_1 and B_2 . The *well-linked number* $\text{wl}(G)$ of a hypergraph G (or of a set $E \subseteq E(G)$) is the largest integer k so that there is a well-linked set $A \subseteq E(G)$ (or $A \subseteq E$) with $\lambda(A) = k$.

Lemma 3.1 ([RS95]). *For every graph G , $\text{wl}(\mathcal{H}(G)) \leq 3 \cdot (\text{tw}(G) + 1)$.*

Superbranch decompositions. A *superbranch decomposition* of a hypergraph G is a pair $\mathcal{T} = (T, \mathcal{L})$, where T is a tree, whose every internal node has degree at least three and $\mathcal{L}: L(T) \rightarrow E(G)$ is a bijection from the leaves of T to the edges of G . A *rooted superbranch decomposition* is a superbranch decomposition $\mathcal{T} = (T, \mathcal{L})$, where T is a rooted tree. For a node $t \in V(T)$ of a rooted superbranch decomposition $\mathcal{T} = (T, \mathcal{L})$ of a hypergraph G , we denote by $\mathcal{L}[t] \subseteq E(G)$ the set of hyperedges that are mapped to leaves in $L(T)$. For each edge tp , where p is the parent of t , we call the set $\text{adh}(tp) := \text{bd}(\mathcal{L}[t])$ the *adhesion* at tp . The maximum size of an adhesion of $\mathcal{T}$ is denoted by $\text{adhsize}(\mathcal{T})$. We say that $\mathcal{T}$ has *adhesion size* c if $\text{adhsize}(\mathcal{T}) \leq c$. A rooted superbranch decomposition of a hypergraph G is *downwards well-linked* if for every node $t \in V(T)$ the set $\mathcal{L}[t] \subseteq E(G)$ is well-linked in G . As $\text{adh}(tp) = \text{bd}(\mathcal{L}[t])$ for each node t with parent p , we have $\text{adhsize}(\mathcal{T}) \leq \text{wl}(G)$.

For a rooted superbranch decomposition $\mathcal{T} = (T, \mathcal{L})$ or a rooted tree decomposition $\mathcal{T} = (T, \text{bag})$, and an internal node $t \in V_{\text{int}}(T)$, we define the *torso* of t to be the labeled hypergraph $\text{torso}(t)$ with

- $E(\text{torso}(t)) = \{e_s \mid st \in E(T)\}$,
- $V(e_s) = \text{adh}(st)$ for each $e_s \in E(\text{torso}(t))$, and
- $V(\text{torso}(t)) = \bigcup_{e_s \in E(\text{torso}(t))} V(e_s)$.

We say that the hyperedge $e_s \in E(\text{torso}(t))$ *corresponds* to the respective node s . Consider a set $A \subseteq E(\text{torso}(t))$ that does not contain the hyperedge e_p corresponding to the edge $tp \in E(T)$, where p is the parent of t . Then, A corresponds to a set of children $\mathcal{C}_A = \{c \in \text{chd}(t) \mid e_c \in A\}$ of t , which in turn corresponds to a set $\bigcup_{c \in \mathcal{C}_A} \mathcal{L}[c]$ of hyperedges of G . We denote this set by $A \triangleright \mathcal{T}$. As shown by [Kor25a], each well-linked set in the torso of a node of a superbranch decomposition of a hypergraph G corresponds to a well-linked set in G .

Lemma 3.2. *Let G be a hypergraph and $\mathcal{T} = (T, \mathcal{L})$ a rooted superbranch decomposition of G with root r . Let also $t \in V_{\text{int}}(T)$ be a node, so that $\mathcal{L}[c]$ is well-linked for every child c of t . Let $C \subseteq E(\text{torso}(t))$ be the set of hyperedges corresponding to the children of t . Then, a set $A \subseteq C$ is well-linked in $\text{torso}(t)$ if and only if $A \triangleright \mathcal{T}$ is well-linked in G .*

Proof. This is a straightforward generalization of [Kor25a, Lemma 4.3]. For the sake of completeness, the full proof can be found in Section A. $\square$

We observe that there is a natural connection between superbranch and tree decompositions. Let G be a hypergraph and let $\mathcal{T} = (T, \mathcal{L})$ be a superbranch decomposition of $\mathcal{H}(G)$ rooted at $r \in V(T)$. Let $\tilde{\mathcal{T}} = (\tilde{T}, \text{bag})$ be a rooted tree decomposition of G with root $r \in V(\tilde{T})$ (with a slight abuse of notation, we denote some nodes of T and $\tilde{T}$ with the same symbol, to simplify things later). We say that $\tilde{\mathcal{T}}$ *corresponds* to $\mathcal{T}$ if

- $\text{bag}(r) = V(\text{torso}(r))$,
- $\text{chd}_{\mathcal{T}}(r) = \text{chd}_{\tilde{\mathcal{T}}}(r)$, and
- for each $t \in \text{chd}(r)$, we have $V(\mathcal{L}[t]) = \bigcup_{t' \in \text{desc}_{\tilde{\mathcal{T}}}(t)} \text{bag}(t')$.

A protrusion decomposition $\mathcal{T} = (T, \text{bag})$ of a graph G with root r is called *normal* if for every hyperedge $e \in E(\mathcal{H}(G))$ there is a node $t \in V(T) \setminus \{r\}$ with $V(e) \subseteq \text{bag}(t)$. We now show that every protrusion decomposition that corresponds to a superbranch decomposition is normal.

Lemma 3.3. *Let G be a graph and let $\tilde{\mathcal{T}} = (\tilde{T}, \text{bag})$ be a protrusion decomposition of G corresponding to a superbranch decomposition $\mathcal{T} = (T, \mathcal{L})$ of $\mathcal{H}(G)$. Then $\tilde{\mathcal{T}}$ is normal.*

Proof. Let r be the root of T and $\tilde{r}$ be the root of $\tilde{T}$. Let $e \in E(\mathcal{H}(G))$. We wish to show that $V(e) \subseteq \text{bag}(\tilde{t})$ for some $\tilde{t} \in V(\tilde{T}) \setminus \{\tilde{r}\}$. Since $\tilde{\mathcal{T}}$ is a tree decomposition, we are done unless $V(e) \subseteq \text{bag}(\tilde{r})$, so assume this. Since r is not a leaf, we have $e \in \mathcal{L}[t]$ for some $t \in \text{chd}(r)$. In particular, $V(e) \subseteq V(\mathcal{L}[t])$. Since $\tilde{\mathcal{T}}$ corresponds to $\mathcal{T}$, we have $V(\mathcal{L}[t]) = \bigcup_{\tilde{t}' \in \text{desc}(\tilde{t})} \text{bag}(\tilde{t}')$ for some $\tilde{t} \in \text{chd}(\tilde{r})$. So $V(e) \subseteq \text{bag}(\tilde{t}')$ for some $\tilde{t}' \in \text{desc}(\tilde{t})$. For each $v \in V(e)$, the subgraph of $\tilde{T}$ induced by the vertices with bags containing v is connected, and contains both $\tilde{r}$ and a node in $\text{desc}(\tilde{t})$, so in particular, it contains $\tilde{t}$. So $V(e) \subseteq \text{bag}(\tilde{t})$. $\square$

Representation of objects. Graphs are represented by the adjacency list format, with each edge uv appearing in both the list of u and the list of v , and having a pointer to its other appearance. Note that we can insert a vertex in $\mathcal{O}(1)$ time, insert an edge in $\mathcal{O}(1)$ time (given that we know it does not already exist), delete an edge in $\mathcal{O}(1)$ time given a pointer to it, and delete an isolated vertex in $\mathcal{O}(1)$ time. A hypergraph G is represented as a bipartite graph with bipartition $(V(G), E(G))$, and there is an edge between $v \in V(G)$ and $e \in E(G)$ if and only if $v \in V(e)$. A tree is represented as a graph, and a rooted tree also contains a global pointer to the root, and each non-root node t stores a pointer to the edge tp where p is the parent of t .

A superbranch decomposition $\mathcal{T} = (T, \mathcal{L})$ of a hypergraph G is represented as follows:

- a representation of T ,

- for each $\ell \in L(T)$ a pointer from ℓ to $\mathcal{L}(\ell)$,
- for each $e \in E(G)$ a pointer from e to $\mathcal{L}^{-1}(e)$,
- for each node $t \in V(T)$, the number of $|L[t]|$ of leaf descendants of it,
- for each internal node $t \in V_{\text{int}}$, a representation of $\text{torso}(t)$, where additionally each hyperedge $e_s \in E(\text{torso}(t))$ corresponding to an edge $st \in E(T)$ stores a pointer to st , and st stores a pointer to e_s .

An annotated tree decomposition $\mathcal{T} = (T, \text{bag}, \text{edges})$ rooted at r is represented as follows:

- a representation of T ,
- for each $t \in V(T) \setminus \{r\}$, the set $\text{bag}(t)$ and $\text{edges}(t)$ are each stored as a linked list to which t contains pointers to, and
- for the root r , the set $\text{edges}(r)$ and $\text{bag}(r)$ are each stored as a balanced binary search tree containing all edges $e \in \text{edges}(r)$ and vertices $v \in \text{bag}(r)$, respectively, which r contains pointers to.

We now describe an algorithm that, given a set of hyperedges $C \subseteq E(G)$, computes the vertex set $V(C)$ and the boundary $\text{bd}(C)$.

Lemma 3.4. *Let G be a hypergraph of rank r whose representation is stored. There is an algorithm that given a set $C \subseteq E(G)$, in time $\mathcal{O}(|C| \cdot r)$ returns the sets $V(C)$ and $\text{bd}(C)$.*

Proof. We start by computing a linked-list representation of $V(C)$. For each hyperedge $e \in C$, we iterate over all vertices $v \in V(e)$ and add v to $V(C)$, if it has not already been added. The running time is at most $\mathcal{O}(|C| \cdot r)$.

To compute $\text{bd}(C)$, we do the following. In the bipartite representation of G , we mark all vertices corresponding to edges $e \in C$. Then, for every vertex $v \in V(C)$, we check if v has an unmarked neighbor and if so, include v in $\text{bd}(C)$. Every edge incident to a marked vertex is touched at most once, and therefore the running time is at most $\mathcal{O}(|C| \cdot r)$. $\square$

Basic (hyper)graph operations. For representing manipulations of (hyper)graphs, we introduce a set of operations, called *basic (hyper)graph operations*, defined as follows.

- **AddVertex:** Given a new vertex $v \notin V(G)$, add v to $V(G)$ and return a pointer to it, in $\mathcal{O}(1)$ time.
- **DeleteVertex:** Given a pointer to an isolated vertex $v \in V(G)$, remove v from $V(G)$ in $\mathcal{O}(1)$ time.
- **AddEdge/AddHyperedge:** For a new (hyper)edge $e \notin E(G)$ with $V(e) \subseteq V(G)$, given pointers to each vertex $v \in V(e)$, add e to $E(G)$ and return a pointer to it, in $\mathcal{O}(|V(e)|)$ time.
- **DeleteEdge/DeleteHyperedge:** Given a pointer to a (hyper)edge $e \in E(G)$, remove e from $E(G)$, in $\mathcal{O}(|V(e)|)$ time.

If G and G' are the (hyper)graphs before and after one basic (hyper)graph operation o , we say that o transforms G into G' . A *sequence of operations* is a finite sequence $\mathcal{C} = c_1 c_2 \dots c_k$, where each c_i specifies a basic (hyper)graph operation together with the information required to perform it: For adding a vertex, the vertex v is stored, for deleting a vertex, the pointer to vertex v is stored, for adding a (hyper)edge, the edge e is stored, and for deleting a (hyper)edge,

the list of pointers to each vertex $v \in V(e)$ is stored. Applying $\mathcal{C}$ to a (hyper)graph G produces a sequence of (hyper)graphs $G = G^{(0)}, G^{(1)}, \dots, G^{(k)}$. The *length* of $\mathcal{C}$ is $|\mathcal{C}| = k$. The *size* of a sequence $\|\mathcal{C}\|$ is the total information stored: A vertex operation has size 1, and a (hyper)edge operation has size $|V(e)| + 1$. This also means that, given a sequence $\mathcal{C}$ of operations, it can be performed in $\mathcal{O}(\|\mathcal{C}\|)$ time. Note that $|\mathcal{C}| \leq \|\mathcal{C}\|$.

Tree decomposition automata. We will now informally define tree decomposition automata for annotated tree decompositions. For more formal definitions, see [Section B](#). The idea of a tree decomposition automaton is to assign a state to each node of an annotated tree decomposition, in a way where the state of a node depends only on the immediate properties of the node and its children, as well as the states of its children. In particular, one should think of a tree decomposition automaton as describing a bottom-up dynamic programming algorithm on tree decompositions.

More concretely, a *tree decomposition automaton* of width ℓ is a tuple (Q, F, ι, δ) , where Q is a set of *states* (containing the *null state* $\perp$), $F \subseteq Q$ is a set of *accepting states*, ι is an *initial mapping* assigning a state $\iota(G)$ to each bounded graph G with $|V(G)| \leq \ell + 1$, and δ is a *transition mapping* computing the state of a node x given $\text{bag}(x)$, $\text{bag}(y)$ for every $y \in \text{chd}(x)$, $\text{edges}(x)$, and the states of the children of x . The state of a leaf $l \in L(T)$ is given by $\iota(G_l)$. Let $\mathcal{T} = (T, \text{bag}, \text{edges})$ be an annotated tree decomposition of width at most ℓ . The *run* $\rho_{\mathcal{A}}^{\mathcal{T}}: V(T) \rightarrow Q$ of $\mathcal{A}$ on $\mathcal{T}$ is the map from nodes to their states. The initial mapping ι and the transition mapping δ are to be computable, whereas the run $\rho_{\mathcal{A}}^{\mathcal{T}}$ merely recalls states already computed using ι and δ . Note that we can compute the full run by using the initial mapping ι on each leaf of $\mathcal{T}$ and then using the transition mapping δ in a bottom-up fashion. We say that $\mathcal{A}$ has *evaluation time* τ if the functions ι and δ can be evaluated in time τ , and it can be decided whether a state $q \in Q$ is in F in time τ .

3.3 Parameterized graph problems and kernelization

In this section, we introduce all the necessary definitions around parameterized graph problems and kernelization, mostly following [\[FLSZ19\]](#). A *parameterized graph problem* is a problem Π with instances (G, k) , where G is a graph and $k \in \mathbb{Z}$ an integer such that one of the following holds:

- For all G, k with $k < 0$: $(G, k) \in \Pi$.
- For all G, k with $k < 0$: $(G, k) \notin \Pi$.

Naturally, for maximization problems, we will always consider the first case, and for minimization problems, the second case. We restrict ourselves to optimization problems that are parameterized by the solution size. For this, we denote by $\text{OPT}_{\Pi}(G)$ the size of an optimal solution for Π on G , i.e.,

$$\text{OPT}_{\Pi}(G) = \begin{cases} \min\{k \mid (G, k) \in \Pi\} & \text{if such a } k \text{ exists} \\ +\infty & \text{otherwise} \end{cases}$$

if Π is a minimization problem, and

$$\text{OPT}_{\Pi}(G) = \begin{cases} \max\{k \mid (G, k) \in \Pi\} & \text{if such a } k \text{ exists} \\ -\infty & \text{otherwise} \end{cases}$$

if Π is a maximization problem.

A parameterized problem is *linearly treewidth-bounding* on a class of graphs $\mathcal{G}$ if there are constants η and c so that for all $G \in \mathcal{G}$ it holds that $\text{tw}(G) \leq \mathcal{O}(\text{tw-mod}_{\eta}(G))$.

FPT and kernelization. A parameterized graph problem Π is called *fixed-parameter tractable* if there exists an algorithm $\mathcal{A}$ (called *fixed-parameter algorithm*) that, given a graph G and an integer k , correctly decides if $(G, k) \in \Pi$ in time $f(k) \cdot n^{\mathcal{O}(1)}$, where f is a computable function. The complexity class of all fixed-parameter tractable problems is called *FPT*. The most fundamental way of finding fixed-parameter algorithms is via kernelization algorithms.

A *kernelization algorithm* for a parameterized graph problem Π is an algorithm $\mathcal{A}$ that, given a graph G , outputs in polynomial time a graph K together with a non-positive integer Δ such that for every integer k , $(G, k) \in \Pi$ if and only if $(K, k + \Delta) \in \Pi$. Moreover, there exists a computable function $g: \mathbb{N} \rightarrow \mathbb{N}$ such that $|K| \leq g(k)$. The output (K, Δ) of the algorithm $\mathcal{A}$ is called the *kernel*. If $g(k) \leq k^{\mathcal{O}(1)}$ or $g(k) \leq \mathcal{O}(k)$, K (or (K, Δ)) is called a *polynomial* or *linear kernel*, respectively.

(Counting) monadic second order logic. The syntax of *monadic second order logic* (MSO_2) of graphs consists of variables for vertices, edges, sets of vertices, and sets of edges, Boolean connectives $\wedge, \vee, \neg, \rightarrow, \leftrightarrow$, and quantifiers $\exists, \forall$ that can be applied to all four types of variables. Furthermore, the following atomic formulas are included:

- $x \doteq y$, where x and y are variables of the same type;
- $x \in X$, where x is a vertex (edge) variable and X a vertex (edge) set variable;
- $\text{adj}(x, y)$, where x and y are vertex variables, and the interpretation is that x and y are adjacent;
- $\text{inc}(x, y)$, where x is a vertex variable and y an edge variable, and the interpretation is that x and y are incident.

Additionally, MSO_2 can be extended to *counting monadic second order logic* (CMSO_2) by adding the atomic formulas $\text{card}_{q,r}$ for $0 \leq q \leq r$, $r \geq 2$, which are interpreted as $\text{card}_{q,r}(S) = \top$ if and only if $|S| \equiv q \pmod{r}$. A graph class $\mathcal{G}$ is called *CMSO_2 -definable* if there is a CMSO_2 -formula ϕ such that $G \in \mathcal{G}$ if and only if $G \models \phi$. For a detailed introduction to CMSO_2 , we refer to [Cou90, Cou92, Cou97].

Finite integer index. In our dynamic kernelization algorithm, we want to replace boundaried subgraphs by “equivalent” graphs. For this, the notion of finite integer index plays a central role. Let $X = (G_X, B_X, \Lambda_X)$, $Y = (G_Y, B_Y, \Lambda_Y)$ be two boundaried graphs. We define $X \oplus_u Y$ as the (not boundaried) graph obtained by taking the disjoint union of G_X and G_Y and then identifying boundary vertices of G_X and G_Y that share the same label. If X and Y have the same label set, we further define $X \oplus_b Y = (X \oplus_u Y, B, \Lambda)$ to be the boundaried graph, where the boundary B contains exactly the vertices that have been identified in the gluing process, and the labeling Λ is the corresponding labeling from X (or equivalently, Y). We observe that $\oplus_u$ is commutative and $\oplus_b$ is commutative and associative. If $G = X \oplus_u Y$ and Y' is another boundaried graph, we say that $G' = X \oplus_u Y'$ is the graph obtained from G by *replacing* Y with Y' .

For a parameterized graph problem Π and two boundaried graphs $G_1, G_2 \in \mathcal{F}$, we say $G_1 \equiv_\Pi G_2$ if and only if $\Lambda(G_1) = \Lambda(G_2)$ and there exists a *transposition constant* $\Delta := \Delta_\Pi(G_1, G_2) \in \mathbb{Z}$ such that

$$\forall (F, k) \in \mathcal{F} \times \mathbb{Z} : (F \oplus_u G_1, k) \in \Pi \Leftrightarrow (F \oplus_u G_2, k + \Delta) \in \Pi.$$

We remark that $\equiv_\Pi$ is indeed an equivalence relation (see [FLSZ19, Exercise 16.2]).

The goal of protrusion replacement is to replace a large protrusion with an equivalent small graph. To do this efficiently, we need to bound the number of equivalence classes of $\equiv_\Pi$. Unfortunately, this is not possible, as it is easy to see that two boundaried graphs with different

label sets belong to different equivalence classes. However, for some problems Π , the number of equivalence classes might be bounded if we restrict $\equiv_\Pi$ to bounded graphs with the same label set $I \subseteq \mathbb{Z}_{\geq 1}$. A parameterized problem Π has *finite integer index* (*FII*) if for every finite $I \subseteq \mathbb{Z}_{\geq 1}$, the number of equivalence classes of $\equiv_\Pi$ that are subsets of $\mathcal{F}_I$ is finite.

4 Existence of mergeable root children

When the degree of the root of the superbranch decomposition grows too large, we need to reduce it without creating another high-degree node and without violating the downwards well-linkedness. That is, we need to find a set of root-children, identified by their corresponding hyperedges in the torso, that we can merge into a downwards well-linked subtree with small treewidth and adhesion size. The goal of this section is to prove [Lemma 4.1](#), which asserts the existence of such a set of root-children. Then, in the subsequent [Section 5](#), we will describe how this set can be found efficiently.

Lemma 4.1. *Let H be a graph and η, α integers. There are integers $\omega := \omega(H, \eta)$ and $\delta := \delta(H, \eta, \alpha)$ such that for every k the following holds: Let G be an H -topological-minor-free graph that has a treewidth- η -modulator of size k . Let $\mathcal{T} = (T, \mathcal{L})$ be a downwards well-linked superbranch decomposition of $\mathcal{H}(G)$ rooted at r and with adhesion size α . If $\Delta(r) \geq \delta \cdot k$, then there exists a subset of hyperedges $B \subseteq E(\text{torso}(r))$ such that*

1. $\lambda(B) \leq \omega$,
2. $\text{itw}(B \triangleright \mathcal{T}) \leq \omega$,
3. $2^{\omega+2} \leq |B| \leq \mathcal{O}_{H, \eta}(\alpha)$, and
4. for every internal component B' of B , we have $\text{bd}(B') = \text{bd}(B)$.

Furthermore, the functions $\omega(H, \eta)$ and $\delta(H, \eta, \alpha)$ are computable.

The rest of this section consists of the proof of [Lemma 4.1](#), divided into two main parts. These parts correspond to the two reasons for the degree $\Delta(r)$ to be high: Either $\text{torso}(r)$ has many vertices, or there are hyperedges with high multiplicity in $\text{torso}(r)$. We start with the first case in [Section 4.1](#), and continue with the second in [Section 4.2](#).

4.1 Many torso vertices

This section deals with the case of [Lemma 4.1](#) where there are many vertices in $\text{torso}(r)$. Specifically, this section is dedicated to the proof of the following lemma.

Lemma 4.2. *Let H be a graph and η, α integers. There are integers $\omega := \omega(H, \eta)$ and $\delta_V := \delta_V(H, \eta, \alpha)$ such that for every k the following holds: Let G be an H -topological-minor-free graph with a treewidth- η -modulator X of size k . Let $\mathcal{T} = (T, \mathcal{L})$ be a downwards well-linked superbranch decomposition of $\mathcal{H}(G)$ with adhesion size α and root r . If $|V(\text{torso}(r))| \geq \delta_V \cdot k$, then there exists a subset of hyperedges $B \subseteq E(\text{torso}(r))$ such that*

1. $\lambda(B) \leq \omega$,
2. $\text{itw}(B \triangleright \mathcal{T}) \leq \omega$,
3. $2^{2\omega+2} \leq |B|$, and
4. $|V(B)| \leq \mathcal{O}_{H, \eta}(\alpha)$.

Furthermore, the functions $\omega(H, \eta)$ and $\delta_V(H, \eta, \alpha)$ are computable.

We split the proof of [Lemma 4.2](#) into two parts. First, in [Section 4.1.1](#) we show that we can find a subset $C \subseteq E(\mathcal{H}(G))$ of hyperedges of the support hypergraph of G , instead of a subset of $E(\text{torso}(r))$, but with similar properties as the desired set B from [Lemma 4.1](#) (see [Lemma 4.3](#)). Then, in [Section 4.1.2](#) we show that we can “uncross” this set C , i.e., for every child c of the root, we either include or exclude the entire set $\mathcal{L}[c]$ from C , roughly preserving our conditions on the boundary size and internal treewidth of C . Finally, the set B is obtained by taking exactly those torso hyperedges that correspond to root children c , where $\mathcal{L}[c]$ is included in C .

4.1.1 Before uncrossing

The goal of this section is to prove the following [Lemma 4.3](#) saying that there exists a set $C \subseteq \mathcal{H}(G)$ with small boundary and internal treewidth, which we will uncross later in [Section 4.1.2](#) to obtain the set $B \subseteq E(\text{torso}(r))$ for [Lemma 4.2](#). The last requirement ([Item 4](#)) is crucial to ensure that the treewidth does not grow too much during the uncrossing.

Lemma 4.3. *Let H be a graph and η, α integers. There are integers $\omega := \omega(H, \eta)$ and $\delta_V := \delta_V(H, \eta, \alpha)$ such that for every $\sigma \geq 3\omega + 1$ and every k the following holds: Let G be an H -topological-minor-free graph that has a treewidth- η -modulator of size k . Let also $\mathcal{T} = (T, \mathcal{L})$ be a downwards well-linked superbranch decomposition of $\mathcal{H}(G)$ rooted at r and with adhesion size α . If $|V(\text{torso}(r))| \geq \delta_V \cdot (\sigma + 1) \cdot k$, then there exists a set of hyperedges $C \subseteq E(\mathcal{H}(G))$, such that*

1. $\sigma \leq |\text{int}(C) \cap V(\text{torso}(r))| \leq 3 \cdot \sigma$,
2. $\lambda(C) \leq \omega + 1$,
3. $\text{itw}(C) \leq \omega$, and
4. for every root-child $s \in \text{chd}(r)$ with $\text{itw}(\mathcal{L}[s]) > \eta$, we have $\text{int}(C) \cap \text{bd}(\mathcal{L}[s]) = \emptyset$.

Furthermore, ω is computable when given H and η , and δ is computable when given H, η , and α .

We will find the set C within a subtree of an optimal protrusion decomposition, the existence of which is given by the following two lemmas. First, Fomin, Lokshtanov, Saurabh, and Thilikos [[FLST20](#)] showed that every H -minor-free graph with a treewidth- η -modulator of size k has a $(\mathcal{O}_{H,\eta}(k), \mathcal{O}_{H,\eta}(1))$ -protrusion decomposition. This result has been extended to H -topological-minor-free graphs by Kim, Langer, Paul, Reidl, Rossmanith, Sau, and Sikdar [[KLP⁺16](#)], even though it was only explicitly stated and proven later by Kim, Serna, and Thilikos [[KST18](#)] (the proof appears only in the full version of [[KST18](#)]).

Lemma 4.4 ([[FLST20](#), [KLP⁺16](#), [KST18](#)]). *Let H be a graph and η an integer. For every k , every H -topological-minor-free graph G with a treewidth- η -modulator X of size k has an $(\mathcal{O}_{H,\eta}(k), \mathcal{O}_{H,\eta}(1))$ -protrusion decomposition, whose root bag contains X .*

We remark that in the case of topological-minor-free graphs it is nowhere stated explicitly that the treewidth- η -modulator X is contained in the root bag of the protrusion decomposition. However, this follows directly from the algorithm [[KLP⁺16](#), Algorithm 1] (see also [Lemma 2](#) in the full version of [[KST18](#)]).

We only want to merge root-children whose subtrees induce a small internal treewidth. To that end, we have one additional requirement on our “optimal” protrusion decomposition, in which we find the set C for [Lemma 4.3](#), namely, we want the root bag to contain the boundaries of all high-treewidth root-children of our current superbranch decomposition. The existence of such a protrusion decomposition is given by the following lemma.

Lemma 4.5. *Let H be a graph and η, α integers. For every k the following holds: Let G be an H -topological-minor-free graph with treewidth- η -modulator X of size k . Let also $\mathcal{T} = (T, \mathcal{L})$ be a downwards well-linked superbranch decomposition of $\mathcal{H}(G)$ with root r and adhesion size α . Let $Y = \{c \in \text{chd}(r) \mid \text{itw}(\mathcal{L}[c]) > \eta\}$ and $X' = X \cup \bigcup_{c \in Y} \text{bd}(\mathcal{L}[c])$. Then, there exists an $(\mathcal{O}_{H,\eta,\alpha}(k), \mathcal{O}_{H,\eta}(1))$ -protrusion decomposition of G whose root bag contains X' .*

Proof. First, we remark that $X' \supseteq X$ is again a treewidth- η -modulator of G . In the following, we first bound the size of X' , and then apply Lemma 4.4 to obtain a protrusion decomposition of G .

Let $s \in Y$ be a root-child with $\text{itw}(\mathcal{L}[s]) > \eta$. Since X is a treewidth- η -modulator, $G \setminus X$ has treewidth at most η . Thus, the subgraph of G induced by $\text{int}(\mathcal{L}[c])$ is not completely contained in $G \setminus X$, so $X \cap \text{int}(\mathcal{L}[c]) \neq \emptyset$. Moreover, for two distinct root-children $s_1, s_2 \in Y$, we have that $\text{int}(\mathcal{L}[s_1]) \cap \text{int}(\mathcal{L}[s_2]) = \emptyset$, and in particular, the vertices from $\text{int}(\mathcal{L}[s_1])$ and $\text{int}(\mathcal{L}[s_2])$ that are contained in X are disjoint. Since X has size k , it follows that there are at most k such root-children s with $\text{itw}(\mathcal{L}[s]) > \eta$, and thus $|Y| \leq k$ and $|X'| \leq k + k \cdot \alpha$.

Applying Lemma 4.4, G has an $(\mathcal{O}_{H,\eta}(|X'|), \mathcal{O}_{H,\eta}(1))$ -protrusion decomposition, that is, an $(\mathcal{O}_{H,\eta,\alpha}(k), \mathcal{O}_{H,\eta}(1))$ -protrusion decomposition, whose root bag contains X' . $\square$

Now we have everything that we need to prove Lemma 4.3, which we re-state here before the proof.

Lemma 4.3. *Let H be a graph and η, α integers. There are integers $\omega := \omega(H, \eta)$ and $\delta_V := \delta_V(H, \eta, \alpha)$ such that for every $\sigma \geq 3\omega + 1$ and every k the following holds: Let G be an H -topological-minor-free graph that has a treewidth- η -modulator of size k . Let also $\mathcal{T} = (T, \mathcal{L})$ be a downwards well-linked superbranch decomposition of $\mathcal{H}(G)$ rooted at r and with adhesion size α . If $|V(\text{torso}(r))| \geq \delta_V \cdot (\sigma + 1) \cdot k$, then there exists a set of hyperedges $C \subseteq E(\mathcal{H}(G))$, such that*

1. $\sigma \leq |\text{int}(C) \cap V(\text{torso}(r))| \leq 3 \cdot \sigma$,
2. $\lambda(C) \leq \omega + 1$,
3. $\text{itw}(C) \leq \omega$, and
4. for every root-child $s \in \text{chd}(r)$ with $\text{itw}(\mathcal{L}[s]) > \eta$, we have $\text{int}(C) \cap \text{bd}(\mathcal{L}[s]) = \emptyset$.

Furthermore, ω is computable when given H and η , and δ is computable when given H , η , and α .

Proof. Let H be a graph, η, α, k integers, and let G be an H -topological-minor-free graph that has a treewidth- η -modulator X of size k . Let $\mathcal{T} = (T, \mathcal{L})$ be a downwards well-linked superbranch decomposition of $\mathcal{H}(G)$ with adhesion size α that is rooted at r . Let Y and X' be defined as in Lemma 4.5, i.e., $Y = \{s \in \text{chd}(r) \mid \text{itw}(\mathcal{L}[s]) > \eta\}$ and $X' = X \cup \bigcup_{s \in Y} \text{bd}(\mathcal{L}[s])$.

Then, by Lemma 4.5, there exist constants $c_1 \leq \mathcal{O}_{H,\eta,\alpha}(1)$ and $c_2 \leq \mathcal{O}_{H,\eta}(1)$ such that G has a $(c_1 \cdot k, c_2)$ -protrusion decomposition $\mathcal{T}^* = (T^*, \text{bag}^*)$ rooted at r^* with $X' \subseteq \text{bag}^*(r^*)$. We set $\delta_V = \delta_V(H, \eta, \alpha) := 2c_1$ and $\omega = \omega(H, \eta) := c_2$.

Given an integer $\sigma \geq 3\omega + 1$, we show that if $|V(\text{torso}(r))| \geq \delta_V \cdot (\sigma + 1) \cdot k$, then there exists a set $C \subseteq E(\mathcal{H}(G))$ that satisfies Items 1 to 4. First, we observe that since the bag of the root r^* of the “optimal” protrusion decomposition $\mathcal{T}^*$ has size at most $c_1 \cdot k$, there are at most $c_1 \cdot k$ vertices from $V(\text{torso}(r))$ in $\text{bag}^*(r^*)$. By the pigeonhole principle, and since r^* has at most $c_1 \cdot k$ children, there is a root-child $s \in \text{chd}(r^*)$ such that $\text{int}(E(\mathcal{H}(G_s)))$ contains at least

$$\frac{|V(\text{torso}(r))| - c_1 \cdot k}{c_1 \cdot k} \geq \frac{\delta_V \cdot (\sigma + 1) \cdot k - c_1 \cdot k}{c_1 \cdot k} = \frac{2 \cdot c_1 \cdot (\sigma + 1) \cdot k - c_1 \cdot k}{c_1 \cdot k} = 2\sigma + 1 \geq \sigma + \omega + 1$$

vertices from $V(\text{torso}(r))$.

We can assume that every node $t \in V(T^*) \setminus \{r^*\}$ has at most two children. This can be achieved by replacing every node $t \in V(T^*) \setminus \{r^*\}$ with more than two children, say $s_1, s_2, \dots, s_q$, by a path $t_1, t_2, \dots, t_q$, where $\text{bag}(t_i) = \text{bag}(t)$, and then replacing the edge $s_i t$ by $s_i t_i$ for $i = 1, \dots, q$. Let $t \in V(T^*)$ such that $|\text{int}(E(\mathcal{H}(G_t))) \cap V(\text{torso}(r))| \geq \sigma + \omega + 1$ but for all (one or two) children $s \in \text{chd}(t)$ we have $|\text{int}(E(\mathcal{H}(G_s))) \cap V(\text{torso}(r))| \leq \sigma + \omega$. We set $C := E(\mathcal{H}(G_t)) \setminus \{e \mid V(e) = \{v\} \subseteq \text{bag}(t)\} \subseteq E(\mathcal{H}(G))$. Since removing the singleton edges e with $V(e) = \{v\} \in \text{bag}(t)$ from $E(\mathcal{H}(G_t))$ cannot increase the number of internal vertices and decreases it by at most $|\text{bag}(t)| \leq \omega + 1$, we have $\sigma \leq |\text{int}(E(\mathcal{H}(G_t))) \cap V(\text{torso}(r))| - \omega - 1 \leq |\text{int}(C) \cap V(\text{torso}(r))| \leq |\text{int}(E(\mathcal{H}(G_t))) \cap V(\text{torso}(r))| \leq 2(\sigma + \omega) + \omega + 1 \leq 3\sigma$, so [Item 1](#) is satisfied.

Note that any boundary vertex $v \in \text{bd}(C)$ is also in $\text{bag}(t)$, so $\lambda(C) \leq \omega + 1$, satisfying [Item 2](#). Furthermore, we have $\text{itw}(C) \leq \text{tw}(\mathcal{H}(G_t)) \leq \omega$, satisfying [Item 3](#). Now, consider a vertex $x \in X'$ with $x \in V(C)$. Then, there is a node t' in the subtree of T^* rooted at t such that $x \in \text{bag}(t')$. Since $x \in \text{bag}(r^*)$, it follows that $x \in \text{bag}(t)$. Thus, the singleton edge $e_x \in E(\mathcal{H}(G))$ with $V(e_x) = \{x\}$ is not in C . Since $x \in V(C)$, it follows that $x \in \text{bd}(C)$. Hence, $V(C) \cap X' \subseteq \text{bd}(C) \cap X'$, so $\text{int}(C) \cap X' = \emptyset$. Since for every root-child s with $\text{itw}(\mathcal{L}[s]) \geq \eta$, we have $\text{bd}(\mathcal{L}[s]) \subseteq X'$, it holds that $\text{int}(C) \cap \text{bd}(\mathcal{L}[s]) \subseteq \text{int}(C) \cap X' = \emptyset$, finally satisfying [Item 4](#). $\square$

4.1.2 Uncrossing

The second part of the proof of [Lemma 4.2](#) is to uncross the set C from [Lemma 4.3](#), i.e., for every root-child s , either include or exclude $\mathcal{L}[s]$ completely from C . During the uncrossing, we need to make sure that the treewidth and the boundary stay bounded, and the set does not become too small. The following lemma tells us that always one of the two options, either including or excluding a well-linked set $A \subseteq E(G)$ from a set $B \subseteq E(G)$, does not increase the size of the boundary.

Lemma 4.6. *Let G be a hypergraph, $A \subseteq E(G)$ a well-linked set and $B \subseteq E(G)$. Then, either $\lambda(B \cup A) \leq \lambda(B)$ or $\lambda(B \setminus A) \leq \lambda(B)$.*

Proof. Assume that $\lambda(B \cup A) > \lambda(B)$. We show that $\lambda(B \setminus A) \leq \lambda(B)$. First, by the submodularity of λ we have $\lambda(B \cap A) < \lambda(A)$. Since A is well-linked and $(B \cap A, \overline{B} \cap A)$ is a bipartition of A , it follows that $\lambda(\overline{B} \cap A) \geq \lambda(A)$. Then, again due to submodularity, $\lambda(\overline{B} \cup A) \leq \lambda(\overline{B})$, where $\lambda(\overline{B} \cup A) = \lambda(B \setminus A)$ and $\lambda(\overline{B}) = \lambda(B)$ both due to the symmetry of λ . Thus, $\lambda(B \setminus A) \leq \lambda(B)$. $\square$

When excluding something from C , the internal treewidth $\text{itw}(C)$ clearly does not increase. When adding something to C , $\text{itw}(C)$ could increase. Nevertheless, we observe the internal treewidth increases by at most the smaller boundary size of the two sets.

Lemma 4.7. *Let G be a hypergraph, $A, B \subseteq E(G)$. Then $\text{itw}(A \cup B) \leq \max(\text{itw}(A), \text{itw}(B)) + \min(\lambda(A), \lambda(B))$.*

Proof. It follows from the symmetry and submodularity of λ that $\lambda(A \setminus B) + \lambda(B \setminus A) \leq \lambda(A) + \lambda(B)$. Therefore, either $\lambda(A \setminus B) \leq \lambda(A)$ or $\lambda(B \setminus A) \leq \lambda(B)$. Thus, by either replacing A by $A \setminus B$ or B by $B \setminus A$, we can assume without loss of generality that A and B are disjoint.

When A and B are disjoint, the inequality

$$\text{itw}(A \cup B) \leq \max(\text{itw}(A), \text{itw}(B)) + |V(A) \cap V(B)| \leq \max(\text{itw}(A), \text{itw}(B)) + \min(\lambda(A), \lambda(B))$$

clearly holds. $\square$

Lastly, the following lemma tells us that the set C cannot cross internal components of a set $\mathcal{L}[s]$ arbitrarily.

Lemma 4.8. *Let G be a hypergraph, $A \subseteq E(G)$ an internally connected set, and $B \subseteq E(G)$. If $\text{bd}(B) \cap \text{int}(A) = \emptyset$ and $A \cap B \neq \emptyset$, then $A \subseteq B$.*

Proof. The case that $|A| = 1$ is trivial, so assume $|A| > 1$. Assume for the sake of contradiction that there exists an edge e such that $e \in A$ and $e \notin B$. First, we show that then $\text{int}(A) \cap \text{int}(B) = \emptyset$.

Suppose there exists a vertex $u \in \text{int}(A) \cap \text{int}(B)$. Since A is internally connected and $|A| > 1$, there exists a vertex $v \in V(e) \setminus \text{bd}(A)$. Then, $v \in \text{int}(A)$ and we remark that $v \notin \text{int}(B)$, since $e \notin B$. As A is internally connected, there is a path between u and v in $\mathcal{P}(G)[\text{int}(A)]$. Since $u \in \text{int}(B)$ and $v \notin \text{int}(B)$ there is a vertex z on the path such that $z \in \text{bd}(B)$. This is a contradiction to $\text{bd}(B) \cap \text{int}(A) = \emptyset$.

Thus, we have $\text{int}(A) \cap \text{int}(B) = \emptyset$. Let $e' \in A \cap B$ and $v \in V(e')$ with $v' \in \text{int}(A)$. Note that $v' \notin \text{bd}(B)$, as otherwise $\text{bd}(B) \cap \text{int}(A) \neq \emptyset$. But then $v' \in \text{int}(B)$, a contradiction. $\square$

Now we are ready to finish the proof of [Lemma 4.2](#), which we re-state here.

Lemma 4.2. *Let H be a graph and η, α integers. There are integers $\omega := \omega(H, \eta)$ and $\delta_V := \delta_V(H, \eta, \alpha)$ such that for every k the following holds: Let G be an H -topological-minor-free graph with a treewidth- η -modulator X of size k . Let $\mathcal{T} = (T, \mathcal{L})$ be a downwards well-linked superbranch decomposition of $\mathcal{H}(G)$ with adhesion size α and root r . If $|V(\text{torso}(r))| \geq \delta_V \cdot k$, then there exists a subset of hyperedges $B \subseteq E(\text{torso}(r))$ such that*

1. $\lambda(B) \leq \omega$,
2. $\text{itw}(B \triangleright \mathcal{T}) \leq \omega$,
3. $2^{2\omega+2} \leq |B|$, and
4. $|V(B)| \leq \mathcal{O}_{H, \eta}(\alpha)$.

Furthermore, the functions $\omega(H, \eta)$ and $\delta_V(H, \eta, \alpha)$ are computable.

Proof. Let $\omega' = \omega'(H, \eta)$ be the constant $\omega(H, \eta)$ of [Lemma 4.3](#). Here, we set $\omega = \omega(H, \eta) := \max(\eta + \omega' + 1, 2 \cdot \omega' + (\omega' + 1)^2 + 1)$. Furthermore, we set $\sigma := \alpha \cdot (\omega' + 1) + \alpha \cdot 2^{2\omega+2} \geq 3\omega' + 1$. By [Lemma 4.3](#), there exists $\delta_V = \delta_V(H, \eta, \alpha, \sigma)$, so that for every k , the following holds: Let G be an H -topological-minor-free graph with a treewidth- η -modulator of size k , and $\mathcal{T} = (T, \mathcal{L})$ be a downwards well-linked superbranch decomposition of $\mathcal{H}(G)$ with $\text{adhsz}(\mathcal{T}) = \alpha$ and root r , where $|V(\text{torso}(r))| \geq \delta_V \cdot k$. Then, there exists a set $C \subseteq E(\mathcal{H}(G))$ satisfying [Items 1 to 4](#), i.e.,

1. $\sigma \leq |\text{int}(C) \cap V(\text{torso}(r))| \leq 3 \cdot \sigma$,
2. $\lambda(C) \leq \omega' + 1$,
3. $\text{itw}(C) \leq \omega'$, and
4. for every root-child $s \in \text{chd}(r)$ with $\text{itw}(\mathcal{L}[s]) > \eta$, we have $\text{int}(C) \cap \text{bd}(\mathcal{L}[s]) = \emptyset$.

For each child s of r that does not satisfy $\mathcal{L}[s] \subseteq C$ or $\mathcal{L}[s] \cap C = \emptyset$, we will modify C by either including or excluding $\mathcal{L}[s]$ completely from C . Throughout this uncrossing process, we change the internal treewidth $\text{itw}(C)$ and $|\text{int}(C) \cap V(\text{torso}(r))|$ slightly so that we end up with

1. $\sigma - \alpha \cdot (\omega' + 1) = \alpha \cdot 2^{2\omega+2} \leq |\text{int}(C) \cap V(\text{torso}(r))| \leq 3 \cdot \sigma + \alpha \cdot (\omega' + 1) + \omega' + 1 \leq \mathcal{O}_{H, \eta}(\alpha)$,
2. $\lambda(C) \leq \omega' + 1 \leq \omega$,
3. $\text{itw}(C) \leq \max(\eta + \omega' + 1, 2 \cdot \omega' + (\omega' + 1)^2 + 1) \leq \omega$, and
4. for each root-child $s \in \text{chd}(r)$ either $\mathcal{L}[s] \subseteq C$ or $\mathcal{L}[s] \cap C = \emptyset$.

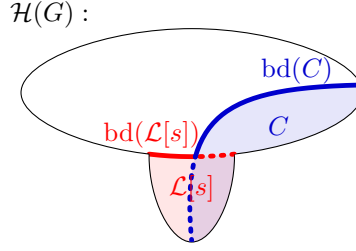


Figure 1: Step 2 of the uncrossing: We want to uncross $\mathcal{L}[s]$, where s is a root-child with $\text{int}(\mathcal{L}[s]) \cap \text{bd}(C) \neq \emptyset$. The boundary of $C \setminus \mathcal{L}[s]$ is the boundary of C (blue) without the part that lies in $\mathcal{L}[s]$ (blue, dashed), that is, $\text{bd}(C) \cap \text{int}(\mathcal{L}[s])$, combined with the boundary of $\mathcal{L}[s]$ that lies in C (red, dashed), that is $\text{int}(C) \cap \text{bd}(\mathcal{L}[s])$.

Then, since each child s of r satisfies $|V(\mathcal{L}[s]) \cap V(\text{torso}(r))| = \lambda(\mathcal{L}[s]) \leq \text{adhsz}(\mathcal{T}) = \alpha$, it follows from [Item 1](#) that C must contain the set $\mathcal{L}[s]$ for at least $2^{2\omega+2}$ children s of r . Then, our desired set B will be the set of hyperedges corresponding to exactly those children, i.e., we choose B so that $B \triangleright \mathcal{T} = C$. Note that $|V(B)| = |V(C) \cap V(\text{torso}(r))| \leq |\text{int}(C) \cap V(\text{torso}(r))| + \lambda(C) \leq \mathcal{O}_{H,\eta}(\alpha)$.

We then describe how to do the uncrossing while achieving the requirements. We say that a child s of r is *crossing* if neither $\mathcal{L}[s] \subseteq C$ nor $\mathcal{L}[s] \cap C = \emptyset$ holds. By *uncrossing* s , we mean setting either $C := C \cup \mathcal{L}[s]$ or $C := C \setminus \mathcal{L}[s]$. Because the sets $\mathcal{L}[s]$ are pairwise disjoint, uncrossing s does not affect whether other children s' are crossing or not. Therefore, we can analyze the process of uncrossing all crossing children s by uncrossing them one-by-one, in an arbitrary order. This uncrossing is done in three steps.

Step 1. We begin with the crossing children s with $\text{itw}(\mathcal{L}[s]) > \eta$. For each such child, we exclude $\mathcal{L}[s]$ from C , that is, we set $C := C \setminus \mathcal{L}[s]$. Clearly $\text{int}(C)$ does not grow, and we still have $\text{itw}(C) \leq \omega'$. Recall that $V(\mathcal{L}[s]) \cap V(\text{torso}(r)) = \text{bd}(\mathcal{L}[s])$ and $\text{bd}(\mathcal{L}[s]) \cap \text{int}(C) = \emptyset$ by [Item 4](#). Therefore, $\mathcal{L}[s]$ contains no hyperedge incident to a vertex in $\text{int}(C) \cap V(\text{torso}(r))$, and thus, $|\text{int}(C) \cap V(\text{torso}(r))|$ is preserved. Lastly, we will show that $\text{bd}(C \setminus \mathcal{L}[s]) \subseteq \text{bd}(C)$, implying that $\lambda(C \setminus \mathcal{L}[s]) \leq \lambda(C)$. For this, consider a vertex $v \in \text{bd}(C \setminus \mathcal{L}[s])$ and suppose $v \notin \text{bd}(C)$. Then $v \in \text{int}(C) \cap \text{bd}(\mathcal{L}[s])$, which contradicts [Item 4](#). Thus, $\lambda(C)$ is preserved.

Step 2. We continue with the children s of r with $\text{int}(\mathcal{L}[s]) \cap \text{bd}(C) \neq \emptyset$ (see [Figure 1](#)). Since for two different root children $s_1, s_2 \in \text{chd}(r)$ the sets $\text{int}(\mathcal{L}[s_1])$ and $\text{int}(\mathcal{L}[s_2])$ are disjoint, we have at most $\lambda(C)$ such children. By [Lemma 4.6](#) we can either include or exclude $\mathcal{L}[s]$ from C without increasing the boundary. We show that, when applying [Lemma 4.6](#) to $\mathcal{L}[s]$ for one fixed child, $\text{itw}(C)$ changes by at most $\lambda(C)$, and $|\text{int}(C) \cap V(\text{torso}(r))|$ changes by at most α . Since there are at most $\lambda(C) \leq \omega' + 1$ such children, we have $\sigma - \alpha \cdot (\omega' + 1) \leq |\text{int}(C) \cap V(\text{torso}(r))| \leq 3 \cdot \sigma + \alpha \cdot (\omega' + 1)$ and $\text{itw}(C) \leq \omega' + (\omega' + 1)^2$ after this step.

First, assume that we are in the first case of [Lemma 4.6](#), i.e., $\lambda(C \cup \mathcal{L}[s]) \leq \lambda(C)$, and thus, we include $\mathcal{L}[s]$, i.e., we set $C := C \cup \mathcal{L}[s]$. By [Lemma 4.7](#), we have $\text{itw}(C \cup \mathcal{L}[s]) \leq \max\{\text{itw}(\mathcal{L}[s]), \text{itw}(C)\} + \lambda(C)$. Furthermore, $|\text{int}(C) \cap V(\text{torso}(r))|$ is clearly not reduced, but it could increase by at most $|V(\mathcal{L}[s]) \cap V(\text{torso}(r))| \leq \alpha$.

Now, assume that we exclude $\mathcal{L}[s]$, i.e., we set $C := C \setminus \mathcal{L}[s]$. In this case, clearly we have $\text{itw}(C \setminus \mathcal{L}[s]) \leq \text{itw}(C)$, but $|\text{int}(C) \cap V(\text{torso}(r))|$ might decrease. The decrease is again bounded by $|V(\mathcal{L}[s]) \cap V(\text{torso}(r))| \leq \alpha$.

Step 3. For this last step, we are left with the crossing children s with $\text{int}(\mathcal{L}[s]) \cap \text{bd}(C) = \emptyset$ and $\text{itw}(\mathcal{L}[s]) \leq \eta$ (see [Figure 2](#)). For a fixed crossing child s , let $A_1, \dots, A_\ell$ denote the internal

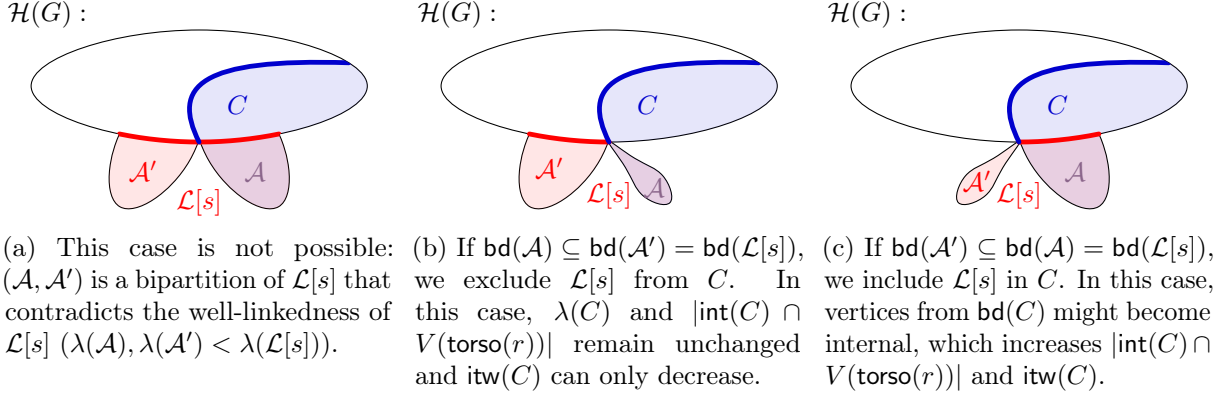


Figure 2: Step 3 of the uncrossing: We want to uncross $\mathcal{L}[s]$, where s is a root-child with $\text{int}(\mathcal{L}[s]) \cap \text{bd}(C) = \emptyset$. $\mathcal{A}$ is the union of the internal components of $\mathcal{L}[s]$ that intersect C (which implies $\mathcal{A} \subseteq C$ by Lemma 4.8), $\mathcal{A}'$ is the union of the remaining components. Figure 2a visualizes, why we always have $\text{bd}(\mathcal{A}) \subseteq \text{bd}(\mathcal{A}') = \text{bd}(\mathcal{L}[s])$ or $\text{bd}(\mathcal{A}') \subseteq \text{bd}(\mathcal{A}) = \text{bd}(\mathcal{L}[s])$. In the first case, where $\text{bd}(\mathcal{A}) \subseteq \text{bd}(\mathcal{A}')$ as visualized in Figure 2b, we exclude $\mathcal{L}[s]$ from C , while in the latter case, where $\text{bd}(\mathcal{A}') \subseteq \text{bd}(\mathcal{A})$ as visualized in Figure 2c, we include $\mathcal{L}[s]$ in C .

components of $\mathcal{L}[s]$. Let $\mathcal{A} = \bigcup_{i: A_i \cap C \neq \emptyset} A_i$ be the union of the internal components intersecting C and $\mathcal{A}' = \mathcal{L}[s] \setminus \mathcal{A}$ the union of the remaining internal components.

By Lemma 4.8 we have $A_i \subseteq C$ for every A_i in $\mathcal{A}$, and thus $\mathcal{A} \subseteq C$. This implies that $\mathcal{A}'$ is non-empty, as otherwise $\mathcal{L}[s] \subseteq C$. By the well-linkedness of $\mathcal{L}[s]$, we either have $\text{bd}(\mathcal{A}') \subseteq \text{bd}(\mathcal{A}) = \text{bd}(\mathcal{L}[s])$ or $\text{bd}(\mathcal{A}) \subseteq \text{bd}(\mathcal{A}') = \text{bd}(\mathcal{L}[s])$. Otherwise, $(\mathcal{A}, \mathcal{A}')$ would be a bipartition of $\mathcal{L}[s]$ with both $\text{bd}(\mathcal{A}), \text{bd}(\mathcal{A}') \subsetneq \text{bd}(\mathcal{L}[s])$, and thus $\lambda(\mathcal{A}), \lambda(\mathcal{A}') < \lambda(\mathcal{L}[s])$ (see Figure 2a).

If $\text{bd}(\mathcal{A}) \subseteq \text{bd}(\mathcal{A}')$ (see Figure 2b), we set $C := C \setminus \mathcal{L}[s] = C \setminus \mathcal{A}$. In this case, every vertex $v \in \text{bd}(\mathcal{A})$ is incident to a hyperedge in $\mathcal{A} \subseteq C$ and to a hyperedge in $\mathcal{A}'$, where $\mathcal{A}' \cap C = \emptyset$, so $v \in \text{bd}(C)$, and hence we have $\text{bd}(\mathcal{A}) \subseteq \text{bd}(C)$. Therefore, excluding $\mathcal{A}$ cannot change the boundary of C . Furthermore, $|\text{int}(C) \cap V(\text{torso}(r))|$ and $\text{itw}(C)$ are clearly preserved.

If $\text{bd}(\mathcal{A}') \subseteq \text{bd}(\mathcal{A})$ (see Figure 2c), we set $C := C \cup \mathcal{L}[s] = C \cup \mathcal{A}'$. With similar arguments as before, we have $\text{bd}(\mathcal{A}') \subseteq \text{bd}(C)$, so including $\mathcal{L}[s]$ cannot add new vertices to the boundary. However, some boundary vertices can become internal, which can increase both $|\text{int}(C) \cap V(\text{torso}(r))|$ and $\text{itw}(C)$. Specifically, let $D \subseteq \text{bd}(C)$ be the boundary vertices that become internal when we add $\mathcal{A}'$ to C , and note that $D \subseteq \text{bd}(\mathcal{A}') \subseteq \text{bd}(\mathcal{L}[s])$. We observe that $|\text{int}(C) \cap V(\text{torso}(r))|$ increases by exactly $|D|$. Furthermore, we observe that

$$\text{itw}(C \cup \mathcal{L}[s]) \leq \max(\text{itw}(C), \text{itw}(\mathcal{L}[s])) + |D| \leq \max(\text{itw}(C), \eta) + |D|.$$

To analyze the total increase of $\text{itw}(C)$ and $|\text{int}(C) \cap V(\text{torso}(r))|$ throughout consecutive uncrossings in step 3, we recall that $\lambda(C)$ never increases, and $|D|$ is bounded by the decrease of $\lambda(C)$. Therefore, the sum of $|D|$ over all uncrossings is at most the initial value of $\lambda(C)$, i.e., at most $\omega' + 1$. It follows that, after all uncrossings in step 3, we have $\lambda(C) \leq \omega' + 1$, $\text{itw}(C) \leq \max(\eta + \omega' + 1, 2 \cdot \omega' + (\omega' + 1)^2 + 1)$, and $\sigma - \alpha \cdot (\omega' + 1) \leq |\text{int}(C) \cap V(\text{torso}(r))| \leq 3 \cdot \sigma + \alpha \cdot (\omega' + 1) + \omega' + 1$, as promised. $\square$

4.2 Proof of Lemma 4.1

In this section, we deal with the case where there are not many vertices in $\text{torso}(r)$, but $\Delta(r)$ is still high, which will then complete the proof of Lemma 4.1. As we show in the following Lemma 4.12, few vertices but many edges in $\text{torso}(r)$ implies that there is an edge with high multiplicity. Later, we argue that we can pick such a high-multiplicity edge in a way so that the

set B for Lemma 4.1 with the required properties can be formed by taking only copies of this edge.

We first recall that H -topological-minor-free graphs have bounded average degree.

Lemma 4.9 ([Mad67]). *If G is a H -topological-minor-free graph, then $|E(G)| \leq \mathcal{O}_H(|V(G)|)$.*

It is an easy consequence of Lemma 4.9 that H -topological-minor-free graphs have a linear number of cliques.

Lemma 4.10. *If G is a H -topological-minor-free graph, then G contains at most $\mathcal{O}_H(|V(G)|)$ cliques.*

Proof. By Lemma 4.9, let $c_H \leq \mathcal{O}_H(1)$ be a constant so that H -topological-minor-free graphs have at most $c_H \cdot |V(G)|$ edges. We prove that H -topological-minor-free graphs have at most $2^{2 \cdot c_H} \cdot |V(G)|$ cliques by induction on $|V(G)|$. This obviously holds when $|V(G)| = 0$. When $|V(G)| \geq 1$, there exists a vertex $v \in V(G)$ of degree $\leq 2 \cdot c_H$. There are at most $2^{2 \cdot c_H}$ cliques containing v . By induction, $G \setminus \{v\}$ contains at most $2^{2 \cdot c_H} \cdot (|V(G)| - 1)$ cliques, so in total there are at most $2^{2 \cdot c_H} \cdot |V(G)|$ cliques. $\square$

We need the following lemma to construct topological minors based on a torso of a node of a superbranch decomposition.

Lemma 4.11. *Let G be a graph, $\mathcal{T} = (T, \mathcal{L})$ be a downwards well-linked superbranch decomposition of $\mathcal{H}(G)$, and $t \in V(T)$ a non-root node. For every two vertices $u, v \in \text{bd}(\mathcal{L}[t])$, there exists a path between u and v whose all edges are in $\mathcal{L}[t]$ and intermediate vertices in $\text{int}(\mathcal{L}[t])$.*

Proof. Let $B_1 = \text{bd}(\mathcal{L}[t]) \setminus \{u\}$ and $B_2 = \text{bd}(\mathcal{L}[t]) \setminus \{v\}$. Due to the well-linkedness of $\mathcal{L}[t]$, there are $|B_1| = |B_2|$ vertex-disjoint paths between B_1 and B_2 using only edges in $\mathcal{L}[t]$. The vertices in $B_1 \cap B_2 = \text{bd}(\mathcal{L}[t]) \setminus \{u, v\}$ are forced to connect to themselves by one-vertex paths, so u must connect to v avoiding vertices in $B_1 \cap B_2$. $\square$

We call such a path from Lemma 4.11 a t -outside path. We then apply Lemmas 4.9 to 4.11 to bound the number of hyperedges in multiplicity-one subsets of $E(\text{torso}(r))$ by the number of their vertices.

Lemma 4.12. *Let H be a graph, G an H -topological-minor-free graph, and $\mathcal{T} = (T, \mathcal{L})$ a downwards well-linked superbranch decomposition of $\mathcal{H}(G)$ with root r . Let also $E \subseteq E(\text{torso}(r))$ be a set of hyperedges with multiplicity one. It holds that $|E| \leq \mathcal{O}_H(|V(E)|)$.*

Proof. We say that a subset $R \subseteq E$ is *routable* if there exists an injective function $g: R \rightarrow \binom{V(E)}{2}$ that maps each hyperedge $e \in R$ to a unique pair of distinct vertices $g(e) = \{u_e, v_e\} \in \binom{V(E)}{2}$.

Claim 4.13. *Any routable set R has size at most $|R| \leq \mathcal{O}_H(|V(E)|)$.*

Proof of the claim. Denote $R = \{e_1, \dots, e_p\}$, and let $S = \{s_1, \dots, s_p\} \subseteq \text{chd}(r)$ be the corresponding set of children of r in T . Let $g: R \rightarrow \binom{V(E)}{2}$ be the injective function, and denote $g(e_i) = \{u_i, v_i\}$. By Lemma 4.11, for each $i \in [p]$ there exists an s_i -outside path P_i in G between u_i and v_i . For different $i \neq j$, the paths P_i and P_j can only intersect in their endpoints. We obtain a graph G' by taking the vertex set $V(E)$, and adding an edge between u_i and v_i for each $i \in [p]$. We observe that G' is a topological minor of G , and therefore H -topological-minor-free. By Lemma 4.9, G' has at most $\mathcal{O}_H(|V(E)|)$ edges, and therefore $|R| \leq \mathcal{O}_H(|V(E)|)$. $\triangleleft$

It remains to show that if $|E|$ would be too large compared to $|V(E)|$, then we could construct a large routable set.

Claim 4.14. *There exists a routable set R of size $|R| \geq |E| - \mathcal{O}_H(|V(E)|)$*

Proof of the claim. Let $R \subseteq E$ be a maximum-size routable set and $g: R \rightarrow \binom{V(E)}{2}$ the corresponding function. As in Claim 4.13, we define a graph G' with the vertex set $V(E)$ and the edge set consisting of the pairs of vertices $\{g(e) \mid e \in R\}$. As before, we observe that G' is a topological minor of G , and therefore G' is H -topological-minor-free. By Lemma 4.10, G' has at most $\mathcal{O}_H(|V(E)|)$ cliques.

For each hyperedge $e \in E \setminus R$ and for each pair $u, v \in V(e)$ of distinct vertices, there must exist $e' \in R$ with $g(e') = \{u, v\}$, as otherwise we could increase $|R|$ by adding e to it. Therefore, for each $e \in E \setminus R$ the set $V(e)$ is a clique in G' . Because E has multiplicity one, the cliques are distinct, and therefore $|E \setminus R| \leq \mathcal{O}_H(|V(E)|)$. $\square$

By combining the conclusions of Claims 4.13 and 4.14, we deduce $|E| \leq \mathcal{O}_H(|V(E)|)$. $\square$

Lemma 4.12 essentially tells us that few vertices and many edges in $\text{torso}(r)$ imply high multiplicity. Next, we show that if there is an edge with high multiplicity, then there is also a low-rank edge with high multiplicity. Then, we can later find the set B within the copies of this edge.

Lemma 4.15. *Let H be a graph on h vertices and let G be an H -topological-minor-free graph together with a downwards well-linked superbranch decomposition $\mathcal{T} = (T, \mathcal{L})$ of $\mathcal{H}(G)$ rooted at r . For every hyperedge $e \in E(\text{torso}(r))$ with rank $|V(e)| \geq h$, the multiplicity of e is less than $\binom{h}{2}$.*

Proof. Let $e \in E(\text{torso}(r))$ be a hyperedge with $|V(e)| \geq h$. Suppose that there are $\ell = \binom{h}{2}$ children $s_1, \dots, s_\ell \in \text{chd}(r)$ with $\text{bd}(\mathcal{L}[s_1]) = \dots = \text{bd}(\mathcal{L}[s_\ell]) = V(e)$. Let $S := \{s_1, \dots, s_\ell\}$. We show that G contains the complete graph K_h as a topological minor, which is a contradiction to the H -topological-minor-freeness.

To that end, we fix a subset $V_h \subseteq V(e)$ of size exactly h together with an arbitrary bijective mapping $g: S \rightarrow \binom{V_h}{2}$. For $1 \leq i \leq \ell$, let $g(s_i) = \{u_i, v_i\} \subseteq V_h \subseteq \text{bd}(\mathcal{L}[s_i])$. We remark that by Lemma 4.11, there is an s_i -outside path between u_i and v_i for every $1 \leq i \leq \ell$. Then, we contract these s_i -outside paths until there is only one edge left of each and obtain the complete graph K_h . Since the s_i -outside paths are vertex-disjoint, except for their endpoints, this K_h is a topological minor of G , which is a contradiction. $\square$

We are now ready to put together an almost-final version of Lemma 4.1 as the following Lemma 4.16. It lacks only one condition compared to Lemma 4.1, which is then immediately added in the following final proof of Lemma 4.1.

In the case when there are many vertices in $\text{torso}(r)$, the proof of Lemma 4.16 follows by combining Lemma 4.2 with Lemmas 4.12 and 4.15. Otherwise, we can directly use the results of Lemmas 4.12 and 4.15 to obtain the desired result in the form of a low-rank high-multiplicity hyperedge of the torso.

Lemma 4.16. *Let H be a graph and η, α integers. There are integers $\omega := \omega(H, \eta)$ and $\delta := \delta(H, \eta, \alpha)$ such that for every k the following holds: Let G be an H -topological-minor-free graph that has a treewidth- η -modulator of size k . Let $\mathcal{T} = (T, \mathcal{L})$ be a downwards well-linked superbranch decomposition of $\mathcal{H}(G)$ rooted at r and with adhesion size α . If $\Delta(r) \geq \delta \cdot k$, then there exists a subset of hyperedges $B \subseteq E(\text{torso}(r))$ such that*

1. $\lambda(B) \leq \omega$,
2. $\text{itw}(B \triangleright \mathcal{T}) \leq \omega$, and
3. $2^{2\omega+2} \leq |B| \leq \mathcal{O}_{H, \eta}(\alpha)$.

Furthermore, ω is computable when given H and η , and δ is computable when given H , η , and α .

Proof. Let H be a graph and η, α, k integers. Let $\omega = \omega(H, \eta)$ be the maximum of the integer $\omega(H, \eta)$ of Lemma 4.2 and $\eta + |V(H)|$. Let also δ_V be the integer $\delta_V(H, \eta, \alpha)$ from Lemma 4.2. Furthermore, let $c_H \leq \mathcal{O}_H(1)$ be the constant from Lemma 4.12 so that for every $E \subseteq E(\text{torso}(r))$ with multiplicity one it holds that $|E| \leq c_H \cdot |V(E)|$.

We set $\delta := c_H \cdot \delta_V \cdot \left(\binom{|V(H)|}{2} + 2^{2\omega+2} \right) + 1$. Let G be an H -topological-minor-free graph that has a treewidth- η -modulator of size k . Let $\mathcal{T} = (T, \mathcal{L})$ be a downwards well-linked superbranch decomposition of $\mathcal{H}(G)$ with adhesion size α and root r , where $\Delta(r) = |E(\text{torso}(r))| \geq \delta \cdot k = c_H \cdot \delta_V \cdot \left(\binom{|V(H)|}{2} \right) \cdot k + c_H \cdot \delta_V \cdot 2^{2\omega+2} \cdot k + k$. We distinguish the two cases that $|V(\text{torso}(r))| \geq \delta_V \cdot k$ or $|V(\text{torso}(r))| < \delta_V \cdot k$.

Many torso vertices. In the first case, by Lemma 4.2, there exists a set $B' \subseteq E(\text{torso}(r))$ of hyperedges such that $\lambda(B') \leq \omega$, $\text{itw}(B' \triangleright \mathcal{T}) \leq \omega$, $|B'| \geq 2^{2\omega+2}$, and $|V(B')| \leq \mathcal{O}_{H,\eta}(\alpha)$. If $|B'| \leq |V(B')| \cdot c_H \cdot \max\left\{\binom{|V(H)|}{2}, 2^{2\omega+2}\right\} \leq \mathcal{O}_{H,\eta}(\alpha)$, we can take B' as B and are done. Otherwise, by the pigeonhole principle, there is a hyperedge $e \in B'$ with multiplicity at least $\max\left\{\binom{|V(H)|}{2}, 2^{2\omega+2}\right\}$ in B' . By Lemma 4.15, $|V(e)| < |V(H)|$. We choose $B \subseteq \{e' \in B' \mid V(e') = V(e)\}$ arbitrarily such that $|B| = 2^{2\omega+2}$. Then, since $B' \subseteq B$, we have $\text{itw}(B' \triangleright \mathcal{T}) \leq \text{itw}(B \triangleright \mathcal{T}) \leq \omega$, and $\lambda(B') \leq |V(B')| = |V(e)| < |V(H)| \leq \omega$.

Few torso vertices. From now on, assume that $|V(\text{torso}(r))| < \delta_V \cdot k$. Since for every $E \in E(\text{torso}(r))$ with multiplicity one, we have $|E| \leq c_H \cdot |V(E)| \leq c_H \cdot |V(\text{torso}(r))| < c_H \cdot \delta_V \cdot k$, it follows that there are fewer than $c_H \cdot \delta_V \cdot k$ hyperedges with pairwise different vertex sets in $E(\text{torso}(r))$. Thus, by Lemma 4.15, there are fewer than $\binom{|V(H)|}{2} \cdot c_H \cdot \delta_V \cdot k$ edges of rank at least $|V(H)|$ and hence more than $c_H \cdot \delta_V \cdot 2^{2\omega+2} \cdot k + k$ edges of rank smaller than $|V(H)|$ in $E(\text{torso}(r))$. Next, we argue that of all these low-rank edges, there are more than $c_H \cdot \delta_V \cdot 2^{2\omega+2} \cdot k$ low-rank edges $e \in E(\text{torso}(r))$ with $\text{itw}(\{e\} \triangleright \mathcal{T}) \leq \eta$.

For this, let X be a treewidth- η -modulator of G of size k , i.e., with $\text{tw}(G - X) \leq \eta$. Therefore, for every hyperedge $e \in E(\text{torso}(r))$, in order to have $\text{itw}(\{e\} \triangleright \mathcal{T}) > \eta$, we need $\text{int}(\{e\} \triangleright \mathcal{T}) \cap X \neq \emptyset$. Because for two distinct hyperedges $e_1, e_2 \in E(\text{torso}(r))$, we have $\text{int}(\{e_1\} \triangleright \mathcal{T}) \cap \text{int}(\{e_2\} \triangleright \mathcal{T}) = \emptyset$, there are at most $|X| = k$ hyperedges $e \in E(\text{torso}(r))$ with $\text{int}(\{e\} \triangleright \mathcal{T}) \cap X \neq \emptyset$ and thus with $\text{itw}(\{e\} \triangleright \mathcal{T}) > \eta$.

It follows that there are more than $c_H \cdot \delta_V \cdot 2^{2\omega+2} \cdot k$ hyperedges $e \in E(\text{torso}(r))$ with rank $|V(e)| < |V(H)|$ and $\text{itw}(\{e\} \triangleright \mathcal{T}) \leq \eta$. Since there are fewer than $c_H \cdot \delta_V \cdot k$ hyperedges with pairwise different vertex sets in $E(\text{torso}(r))$, by the pigeonhole principle, there is a set $B \subseteq E(\text{torso}(r))$ of size $|B| = 2^{2\omega+2}$ such that for all hyperedges $e \in B$, we have $V(e) = V(B)$ with $|V(B)| < |V(H)|$ and $\text{itw}(\{e\} \triangleright \mathcal{T}) \leq \eta$. It follows that $\text{bd}(B) \subseteq V(B)$, so $\lambda(B) < |V(H)| \leq \omega$ and $\text{itw}(B \triangleright \mathcal{T}) \leq \max_{e \in B} \{\text{itw}(\{e\} \triangleright \mathcal{T})\} + \lambda(B) < \eta + |V(H)| \leq \omega$. $\square$

The only remaining condition to prove Lemma 4.1 is Item 4, i.e., that for every internal component B' of the desired set B , we have $\text{bd}(B') = \text{bd}(B)$. We ensure this condition by taking a subset of the set B from Lemma 4.16 consisting of internal components with the same boundary. Let us now re-state Lemma 4.1 and finish its proof.

Lemma 4.1. *Let H be a graph and η, α integers. There are integers $\omega := \omega(H, \eta)$ and $\delta := \delta(H, \eta, \alpha)$ such that for every k the following holds: Let G be an H -topological-minor-free graph that has a treewidth- η -modulator of size k . Let $\mathcal{T} = (T, \mathcal{L})$ be a downwards well-linked superbranch decomposition of $\mathcal{H}(G)$ rooted at r and with adhesion size α . If $\Delta(r) \geq \delta \cdot k$, then there exists a subset of hyperedges $B \subseteq E(\text{torso}(r))$ such that*

1. $\lambda(B) \leq \omega$,

2. $\text{itw}(B \triangleright \mathcal{T}) \leq \omega$,
3. $2^{\omega+2} \leq |B| \leq \mathcal{O}_{H,\eta}(\alpha)$, and
4. for every internal component B' of B , we have $\text{bd}(B') = \text{bd}(B)$.

Furthermore, the functions $\omega(H, \eta)$ and $\delta(H, \eta, \alpha)$ are computable.

Proof. Let H be a graph and η, α, k integers. Let δ, ω be the integers from Lemma 4.16 such that the following holds: Let G be an H -topological-minor-free graph that has a treewidth- η -modulator of size k . Let $\mathcal{T} = (T, \mathcal{L})$ be a downwards well-linked superbranch decomposition of $\mathcal{H}(G)$ rooted at r and with adhesion size α . By Lemma 4.16, there exists a set $C \subseteq E(\text{torso}(r))$ with $\lambda(C), \text{itw}(C \triangleright \mathcal{T}) \leq \omega$ and $2^{2\omega+2} \leq |C| \leq \mathcal{O}_{H,\eta}(\alpha)$. We consider the internal components of C . First, we observe that if C' is an internal component of C , then $\text{bd}(C') \subseteq \text{bd}(C)$.

Now, we group the internal components of C by their boundary. Since each such boundary is a subset of $\text{bd}(C)$, there are at most $2^{\lambda(C)} \leq 2^\omega$ such groups. Thus, by the pigeonhole principle, there exists at least one group such that the union over all internal components in this group contains at least $|C|/2^\omega \geq 2^{\omega+2}$ hyperedges. Let B be the union of all internal components in this group. Clearly, B satisfies Items 3 and 4. It is also easy to see that $\text{itw}(B \triangleright \mathcal{T}) \leq \text{itw}(C \triangleright \mathcal{T}) \leq \omega$, since $B \subseteq C$. Furthermore, since each internal component of B has the same boundary, and this boundary is a subset of $\text{bd}(C)$, $\lambda(B) \leq \lambda(C) \leq \omega$. $\square$

5 Dynamic local search

In the previous section, we showed that if the degree of the root of our superbranch decomposition becomes too high, then there exists a set of root-children, identified by their corresponding hyperedges in the torso of the root, that we can merge. In detail, we showed that there exists a set $B \subseteq E(\text{torso}(r))$ of bounded size such that $B \triangleright \mathcal{T}$ has small boundary and internal treewidth. Additionally, we can assume that for every internal component B' of B we have $\text{bd}(B') = \text{bd}(B)$, which makes this set B traceable via a local search – a property that we will exploit in this section.

Here, we construct a data structure that keeps track of all small internally connected sets in the torso of the root, grouped by their respective boundary, so that we can easily find the desired set B within such a group. An important ingredient to this data structure is the following local search algorithm that efficiently finds all small internally connected sets that intersect a given set I of hyperedges. In our data structure, we will then employ this local search algorithm to locally recompute internally connected sets after every update to the torso of the root.

The local search algorithm is based on a similar algorithm given by Korhonen [Kor25b, Lemma 7.3], the main difference being that they did not require an upper bound on the size of the sets A but instead on the multiplicity of the hypergraph G . For clarity, we again describe the algorithm here.

Lemma 5.1 (Based on [Kor25b, Lemma 7.3]). *Let G be a hypergraph of rank r , whose representation is already stored. There is an algorithm that, given three integers $p, s, k \geq 0$, and two sets $I \subseteq E(G)$ and $X \subseteq V(I)$ of size $1 \leq |I| \leq p$ and $|X| \leq s$, in time $pr \cdot s^{\mathcal{O}(k)}$ lists all sets $A \subseteq E(G)$ so that*

- $I \subseteq A$,
- $X \subseteq \text{bd}(A)$,
- $|A| \leq p$,

- $|V(A)| \leq s$,
- $\lambda(A) \leq k$, and
- every internal component of A intersects I .

The number of such sets is at most $s^{\mathcal{O}(k)}$.

Proof. We describe a recursive algorithm. In each recursive step, we first compute in time $\mathcal{O}(|I| \cdot r) = \mathcal{O}(p \cdot r)$ the set $V(I)$ and $\text{bd}(I)$ using [Lemma 3.4](#). If $|I| > p$, $|V(I)| > s$, or $|X| > k$, we can immediately return the empty list. Now, we can assume that $|V(I)| \leq s$ and $|I| \leq p$. If X intersects $\text{int}(I)$, we can again return the empty list. If $X = \text{bd}(I)$, the only set A that satisfies the requirements is I itself, so we can output I .

Otherwise, we can assume that $X \subsetneq \text{bd}(I)$. Then, we choose an arbitrary $v \in \text{bd}(I) \setminus X$. For any set that satisfies the requirements, we either have $v \in \text{bd}(A)$ or $v \in \text{int}(A)$. We now make two recursive calls, one enumerating the sets A of the first type and the other enumerating the sets A of the latter type. For the first type, it suffices to simply add v to X . For the second case, we need to add all hyperedges in $N^{\text{inc}}(v)$ to I . If $|N^{\text{inc}}(v)| > p$, the recursive call with $I \cup N^{\text{inc}}(v)$ would immediately return, so we can skip the recursion in this case. Otherwise, we recurse on $I \cup N^{\text{inc}}(v)$.

We observe that each recursive call takes time $\mathcal{O}(p \cdot r)$. Moreover, we show that the recursion tree has size at most $s^{\mathcal{O}(k)}$. For this, we observe that in a recursive call of the first type, $|X|$ increases, and in a recursive call of the second type, $|\text{int}(I)|$ increases. As $|X|$ is bounded by k and $|\text{int}(I)|$ by s , we find that the size of the recursion tree is at most $\binom{s+k}{k} \leq s^{\mathcal{O}(k)}$. It follows that the total running time of our algorithm is $pr \cdot s^{\mathcal{O}(k)}$ and that there are at most $s^{\mathcal{O}(k)}$ sets in the list. $\square$

Now, we are ready to construct our local search data structure, which we will later use to keep track of the small internally connected sets with small internal treewidth and boundary size in the torso of the root of our superbranch decomposition. To abstract away from the internal treewidth requirement, we introduce the concept of an oracle $\mathcal{O}$ that, given a set S of hyperedges, depends on S itself and on the boundary $\text{bd}(S)$ and returns true $\top$ or false $\perp$ – later corresponding to whether $S \triangleright \mathcal{T}$ has small internal treewidth or not. As we are only interested in small sets of size at most s_2 , we restrict the oracle to be s_2 -bounded, i.e., the oracle only needs to decide correctly whether the internal treewidth of $S \triangleright \mathcal{T}$ is small if $|S| \leq s_2$.

Then, given a hypergraph G , this data structure supports insertions and deletions of vertices and hyperedges to G , and the **Query** operation allows us to find a set C of bounded size and bounded boundary such that every internal component of C has the same boundary as C and satisfies the oracle $\mathcal{O}$. Applied to $\text{torso}(r)$ with an internal treewidth oracle (realized by using the internal treewidth automaton from [Lemma 7.1](#)) this data structure returns (if possible) a set $C \subseteq E(\text{torso}(r))$ of hyperedges, corresponding to mergeable subtrees of our superbranch decomposition. Whenever the degree $\Delta(r)$ is too high, the existence of such a set C is guaranteed by [Lemma 4.1](#), so in this case the data structure will always return such a set C and we can decrease $\Delta(r)$ by merging the corresponding subtrees.

Recall that during this paper we are working with *labeled* hyperedges, i.e., each insertion inserts a hyperedge with a new label. In particular, we do not require the hypergraph G to have multiplicity one, i.e., there might be hyperedges with the same vertex set but different labels in G . Then, most importantly, for a set $S \subseteq E(G)$ of labeled hyperedges, the result of the oracle $\mathcal{O}(S)$ really depends on the labels of the hyperedges, and not only their vertex sets.

Lemma 5.2. *Let G be a dynamic hypergraph of rank r and let s_1 , s_2 , and k be integers with $s_1 \leq s_2$. Let $\mathcal{O}$ be an s_2 -bounded oracle that, given a set $S \subseteq E(G)$ of labeled hyperedges of size*

$|S| \leq s_2$, depends on S and $\text{bd}(S)$ and returns $\top$ or $\perp$ in time $\mathcal{O}_{r,s_2}(\log |E(G)|)$. There is a data structure that maintains G and supports the following operations:

- **Init(G):** Given an empty hypergraph G , initialize the data structure. Runs in time $\mathcal{O}(1)$.
- **AddVertex(v):** Given a vertex $v \notin V(G)$, add v to G . Runs in time $\mathcal{O}(1)$.
- **DeleteVertex(v):** Given an isolated vertex $v \in V(G)$, remove v from G . Runs in time $\mathcal{O}(1)$.
- **AddHyperedge(e):** Given a hyperedge $e \notin E(G)$ of rank at most r with a new label, add e to G . Runs in time $\mathcal{O}_{r,s_2,k}(\log |E(G)|)$.
- **DeleteHyperedge(e):** Given a hyperedge $e \in E(G)$, remove e from G . Runs in time $\mathcal{O}_{r,s_2,k}(\log |E(G)|)$.
- **Query:** Returns a set $C \subseteq E(G)$ such that:
 - $s_1/2 \leq |C| \leq s_2$,
 - $\lambda(C) \leq k$,
 - for every internal component C' of C , we have $\text{bd}(C') = \text{bd}(C)$, and
 - for every internal component C' of C , we have $\mathcal{O}(C') = \top$,

or reports that no such set of size $s_1 \leq |C| \leq s_2$ exists. Runs in time $\mathcal{O}_{r,s_2,k}(1)$.

Proof. We define a *chip* of a hypergraph G to be an internally connected set $Z \subseteq E(G)$ with $|Z| \leq s_2$, $\lambda(Z) \leq k$, and $\mathcal{O}(Z) = \top$. Note that due to the bound on the rank, it also holds that $|V(Z)| \leq r \cdot s_2$. For a set $\mathcal{Z}$ of chips, we denote by $\text{vol}(\mathcal{Z}) := \sum_{Z \in \mathcal{Z}} |Z|$ the total size over all chips in $\mathcal{Z}$. For a set $B \subseteq V(G)$ of size $|B| \leq k$, we denote by $\text{chips}(B)$ the set of all chips with boundary B and by $\text{vol}(B) := \text{vol}(\text{chips}(B))$. We maintain the following three data structures.

- For every $B \subseteq V(G)$ of size $|B| \leq k$ with $\text{vol}(B) > 0$, a balanced binary search tree $\mathcal{S}_B$ containing all chips $Z \in \text{chips}(B)$.
- A max-heap containing all sets $B \subseteq V(G)$ of size $|B| \leq k$ with $\text{vol}(B) > 0$ ranked by $\text{vol}(B)$ and pointing to the corresponding search tree $\mathcal{S}_B$.
- A balanced binary search tree $\mathcal{S}^*$ that contains each set B of size $|B| \leq k$ with $\text{vol}(B) > 0$ together with a pointer pointing to the element of the max-heap that corresponds to B .

Before describing how we maintain these three data structures, we first show that the local search algorithm from [Lemma 5.1](#) can be used to find all chips of G containing a specific set of hyperedges. This will later be used to locally recompute chips after every update.

Claim 5.3. *There is an algorithm that, given a set $I \subseteq E(G)$ with $1 \leq |I| \leq s_2$ and a set $X \subseteq I$, enumerates all chips Z with $I \subseteq Z$ and $X \subseteq \text{bd}(Z)$ in time $\mathcal{O}_{r,s_2,k}(\log |E(G)|)$. Moreover, the number of such chips is $\mathcal{O}_{s_2,k}(1)$.*

Proof of the claim. Given two sets $I \subseteq E(G)$ and $X \subseteq I$, in order to find all chips Z with $I \subseteq Z$ and $X \subseteq \text{bd}(Z)$, we call the algorithm from [Lemma 5.1](#) with I , X , and the required bounds from the definition of a chip, and then filter out the sets that are no chips. That is, for every set A enumerated by the algorithm, we check whether it is internally connected and satisfies the oracle. For a set A , checking whether it is internally connected can be done in $(r + |A|)^{\mathcal{O}(1)} = \mathcal{O}_{r,s_2}(1)$ time, and checking whether $\mathcal{O}(A) = \top$ can be done in time $\mathcal{O}_{r,s_2}(\log |E(G)|)$. Since the algorithm from [Lemma 5.1](#) takes time $s_2 \cdot r \cdot (r \cdot s_2)^{\mathcal{O}(k)} = (r \cdot s_2)^{\mathcal{O}(k)}$ and returns at most $s_2^{\mathcal{O}(k)}$ sets A , the total running time is $\mathcal{O}_{r,s_2,k}(\log |E(G)|)$. $\triangleleft$

To bound the running times for updates to our three data structures, we now use [Claim 5.3](#) to show that there are at most $\mathcal{O}_{r,s_2,k}(|E(G)|)$ chips in G .

Claim 5.4. *There are at most $\mathcal{O}_{s_2,k}(|E(G)|)$ chips in G .*

Proof of the claim. If we call the algorithm from [Claim 5.3](#) with $I = \{e\}$ for some hyperedge $e \in E(G)$ and $X = \emptyset$, we obtain all chips $Z \subseteq E(G)$ with $e \in A$. So, repeating this call to the algorithm for every hyperedge $e \in E(G)$ gives us all chips. Since every call returns at most $\mathcal{O}_{s_2,k}(1)$ sets, there are at most $\mathcal{O}_{s_2,k}(|E(G)|)$ chips in G . $\triangleleft$

It follows that each balanced binary search tree $\mathcal{S}_B$ contains at most $\mathcal{O}_{s_2,k}(|E(G)|)$ chips, each of size at most s_2 , so inserting and deleting takes time at most $\mathcal{O}_{s_2,k}(\log |E(G)|)$. It also holds that there are at most $\mathcal{O}_{s_2,k}(|E(G)|)$ sets $B \subseteq V(G)$ with $\text{vol}(B) > 0$, so with the help of $\mathcal{S}^*$, we can find the max-heap entry corresponding to a given set $B \subseteq V(G)$ in $\mathcal{O}_{s_2,k}(\log |E(G)|)$ time, and thus inserting or deleting B , or updating the rank $\text{vol}(B)$ in the max-heap can also be done in $\mathcal{O}_{s_2,k}(\log |E(G)|)$ time.

In the following, we first describe how we maintain these three data structures, i.e., how we implement the operations `Init`, `AddVertex`, `DeleteVertex`, `AddHyperedge`, and `DeleteHyperedge`, before we describe how queries are answered. For the `AddHyperedge` and `DeleteHyperedge` operations, let G denote the hypergraph before and G' the hypergraph after the insertion/deletion of the hyperedge. For a set $Z \in E(G) \cap E(G')$, we denote by $\text{bd}_G(Z)$ and $\text{bd}_{G'}(Z)$, $\lambda_G(Z)$ and $\lambda_{G'}(Z)$, and $\text{int}_G(Z)$ and $\text{int}_{G'}(Z)$ the boundary, boundary size and set of internal vertices of Z in G or G' , respectively. To avoid duplicates, we assume that during the `AddHyperedge` and `DeleteHyperedge` operations, every time a chip Z is inserted into its binary search tree $\mathcal{S}_{\text{bd}_{G'}(Z)}$, we first check whether it is already there, and if so, skip its insertion.

Without explicitly stating it later in the descriptions of the operations, we assume that whenever we add the first chip to its balanced binary search tree or remove the last one, the search tree is created or removed, and the max-heap and the balanced binary search tree $\mathcal{S}^*$ are updated accordingly. That is, whenever we want to add a new chip Z to a balanced binary search tree $\mathcal{S}_B$ but $\mathcal{S}_B$ does not yet exist, we create $\mathcal{S}_B$ as a new balanced binary search tree with only one entry Z . We also compute $\text{vol}(B) = |Z|$, and insert the set B together with a pointer to $\mathcal{S}_B$ and with rank $\text{vol}(B)$ into the max-heap. Lastly, we add B together with a pointer to the corresponding max-heap entry to $\mathcal{S}^*$. As argued before, these operations can be done in $\mathcal{O}_{r,s_2,k}(\log |E(G)|)$ time. Similarly, when we remove a chip Z from a balanced binary search tree $\mathcal{S}_B$, and afterward have $\text{vol}(B) = 0$, we can delete the entire search tree $\mathcal{S}_B$ together with the corresponding entries in the max-heap and the balanced binary search tree $\mathcal{S}^*$. For this, we look up the pointer to the corresponding max-heap entry in $\mathcal{S}^*$. Then, the max-heap entry points to the search tree $\mathcal{S}_B$, which we delete. Next, we delete the entry from the max-heap and restore the max-heap property. Lastly, we remove the entry from the $\mathcal{S}^*$. This can all be done in $\mathcal{O}_{r,s_2,k}(\log |E(G)|)$ time.

Init: Given a hypergraph G without vertices and hyperedges, the data structure can clearly be initialized in $\mathcal{O}(1)$ time.

AddVertex(v) / DeleteVertex(v): We remark that inserting or deleting an isolated vertex does neither change anything about the existing chips nor creates any new chips. The only thing that needs to be done is updating the representation of G itself, which clearly can be done in $\mathcal{O}(1)$ time.

AddHyperedge(e): Before we update our three data structures, let us first consider the representation of G . Recall that a hypergraph G is stored as a bipartite graph, where the two parts of the bipartition are $V(G)$ and $E(G)$ and there is an edge between $v \in V(G)$ and $e' \in E(G)$ if $v \in V(e')$. Since the new hyperedge e has rank at most r , we need to add one vertex and at most r edges to the representation of G , which can be done in $\mathcal{O}(r)$ time.

Now, we describe how the three data structures are updated. The only thing that can happen to an existing chip Z when adding a hyperedge to G is that some internal vertices can become boundary vertices, which increases the boundary size and might destroy the internal connectivity. We say a chip Z of G is *affected* if $\text{bd}_G(Z) \neq \text{bd}_{G'}(Z)$ and remark that an affected chip is not necessarily a chip of G' . However, every *unaffected* chip, i.e., every chip of G that is not affected, is also a chip of G' . For this, note that $\text{bd}_G(Z) = \text{bd}_{G'}(Z)$ also implies that $\text{int}_G(Z) = \text{int}_{G'}(Z)$. Hence, every unaffected chip Z is internally connected in G' , $\lambda_{G'}(Z) = \lambda_G(Z) \leq k$, and $\mathcal{O}(Z) = \top$ in G' , so Z is a chip of G' . Therefore, we only need to take care of the affected chips in the following.

For every affected chip Z , we need to remove Z from $\mathcal{S}_{\text{bd}_G(Z)}$, and if Z is a chip of G' , insert it into $\mathcal{S}_{\text{bd}_{G'}(Z)}$. To find out, which chips are affected, we first argue that if $\text{int}_G(Z) \cap V(e) = \emptyset$ for a chip Z of G , then $\text{bd}_G(Z) = \text{bd}_{G'}(Z)$, so Z is unaffected.

Claim 5.5. *Let $Z \subseteq E(G)$ be a set with $\text{int}_G(Z) \cap V(e) = \emptyset$. Then, $\text{bd}_G(Z) = \text{bd}_{G'}(Z)$.*

Proof of the claim. First, we observe that $\text{bd}_G(Z) \subseteq \text{bd}_{G'}(Z)$ as adding a hyperedge cannot decrease the boundary. To show that $\text{bd}_G(Z) \supseteq \text{bd}_{G'}(Z)$, let $v \in \text{bd}_{G'}(Z)$. Then, $v \in V(Z)$, so either $v \in \text{bd}_G(Z)$ or $v \in \text{int}_G(Z)$. In the first case, we are done. In the second case, there is no hyperedge $e' \in E(G) \setminus Z$ with $v \in V(e')$ and we know that $v \notin V(e)$ since $\text{int}_G(Z) \cap V(e) = \emptyset$. Thus, there is also no hyperedge $e' \in E(G') \setminus Z$, which contradicts $v \in \text{bd}_{G'}(Z)$. Hence, $\text{bd}_G(Z) = \text{bd}_{G'}(Z)$. $\triangleleft$

It follows that every affected chip Z has a vertex $v \in \text{int}_G(Z) \cap V(e)$. Now, we can determine the affected chips as follows. Note that for every vertex $v \in V(e)$ with $|N_G^{\text{inc}}(v)| > s_2$ and every chip Z , we have $N_G^{\text{inc}}(v) \not\subseteq Z$, so $v \notin \text{int}_G(Z)$. Thus, we only need to consider vertices $v \in V(e)$ with $|N_G^{\text{inc}}(v)| \leq s_2$. For a fixed such vertex v , we can determine all chips Z with $N_G^{\text{inc}}(v) \subseteq Z$ and thus with $v \in \text{int}_G(Z) \cap V(e)$, by calling the algorithm from Claim 5.3 with $I = N_G^{\text{inc}}(v)$ and $X = \emptyset$ on the hypergraph G , which takes time $\mathcal{O}_{r,s_2,k}(\log |E(G)|)$ and returns at most $\mathcal{O}_{s_2,k}(1)$ chips. Then, for every such chip Z we compute the boundary $\text{bd}_G(Z)$ in time $\mathcal{O}(r \cdot s_2)$, remove Z from the corresponding balanced binary search tree $\mathcal{S}_{\text{bd}_G(Z)}$, and update the rank $\text{vol}(\text{bd}_G(Z))$ in the max-heap in time $\mathcal{O}_{r,s_2,k}(\log |E(G)|)$.

Now, the balanced binary search trees contain only those chips of G' that are also chips of G with $\text{int}_G(Z) \cap V(e) = \emptyset$, and each one of them is in the correct balanced binary search tree with respect to their boundary in G' . Next, we reinsert the chips Z of G' with $\text{int}_G(Z) \cap V(e) \neq \emptyset$, which were deleted in the previous step. To do this, we again apply the algorithm from Claim 5.3, now on G' , with $I = N_{G'}^{\text{inc}}(v) \setminus \{e\}$ and $X = \{v\}$ for every vertex $v \in V(e)$ with $|N^{\text{inc}}(v) \setminus \{e\}| \leq s_2$. For each enumerated chip Z of G' , we compute $\text{bd}_{G'}(Z)$, insert Z into $\mathcal{S}_{\text{bd}_{G'}(Z)}$, and update the rank $\text{vol}(\text{bd}_{G'}(Z))$ in time $\mathcal{O}_{r,s_2,k}(\log |E(G)|)$.

Afterwards, every chip Z of G' with $Z \subseteq E(G)$ is in the correct balanced binary search tree $\mathcal{S}_{G'}(Z)$ since either Z was an unaffected chip of G (more precisely, $\text{int}_G(Z) \cap V(e) = \emptyset$) or we (re)inserted Z in the previous step. Thus, the only chips Z of G' that are missing are the ones with $Z \not\subseteq E(G)$, i.e., with $e \in Z$. We can find all these chips using the algorithm from Claim 5.3 with $I = \{e\}$ and $X = \emptyset$ in time $\mathcal{O}_{r,s_2,k}(\log |E(G)|)$ and there are at most $\mathcal{O}_{s_2,k}(1)$ of them. For each one of them, we compute $\text{bd}_{G'}(Z)$, insert Z into $\mathcal{S}_{\text{bd}_{G'}(Z)}$, and update the rank $\text{vol}(\text{bd}_{G'}(Z))$ in time $\mathcal{O}_{r,s_2,k}(\log |E(G)|)$.

DeleteHyperedge(e): Again, we start with updating the representation of G , which is a bipartite graph H with parts $V(G)$ and $E(G)$. For the hyperedge $e \in E(G)$, we need to delete every edge that is incident to e in H . This can be done in time $\mathcal{O}(r)$ by iterating over the adjacencies of e and using the pointers to find the second appearance of the respective edge. Lastly, deleting the vertex $e \in E(G)$ from H can be done in $\mathcal{O}(1)$ time.

We continue with updating our data structures for the chips of G . When deleting e , every chip Z with $e \in Z$ needs to be removed. By calling the algorithm from [Claim 5.3](#) with $I = \{e\}$ and $X = \emptyset$, we find all chips of G that contain e in time $\mathcal{O}_{r,s_2,k}(\log |E(G)|)$, and there are at most $\mathcal{O}_{s_2,k}(1)$ of them. We remove every such Z from $\mathcal{S}_{\text{bd}_G(Z)}$ and update $\text{vol}(\text{bd}_G(Z))$ in time $\mathcal{O}_{r,s_2,k}(\log |E(G)|)$.

For sets $Z \subseteq E(G)$ with $e \notin Z$, the boundary might decrease when deleting e . For one, this means that some of the existing chips need to be regrouped. Secondly, there might be chips Z in G' that are not chips of G because $\text{bd}_G(Z) > k$ or Z is not internally connected in G . In the following, we show that this boundary decrease is the only thing that can happen to sets Z with $e \notin Z$, before we take care of these sets.

Claim 5.6. *Let $Z \subseteq E(G)$ be a set with $e \notin Z$. Then, $\lambda_G(Z) - r \leq \lambda_{G'}(Z) \leq \lambda_G(Z)$ and $\text{int}_{G'}(Z) \supseteq \text{int}_G(Z)$. Furthermore, if there is no vertex $v \in \text{bd}_G(Z) \cap V(e)$ with $N_G^{\text{inc}}(v) \setminus \{e\} \subseteq Z$, then $\text{bd}_G(Z) = \text{bd}_{G'}(Z)$ and $\text{int}_{G'}(Z) \supseteq \text{int}_G(Z)$.*

Proof of the claim. First, note that deleting e cannot increase the boundary of a set Z , i.e., $\text{bd}_{G'}(Z) \subseteq \text{bd}_G(Z)$, so $\lambda_{G'}(Z) \leq \lambda_G(Z)$ and $\text{int}_{G'}(Z) \supseteq \text{int}_G(Z)$. Moreover, we observe that for every vertex $v \in \text{bd}_G(Z) \setminus \text{bd}_{G'}(Z)$, we have $v \in V(e)$ and $N_G^{\text{inc}}(v) \setminus \{e\} \subseteq Z$. Since $|V(e)| \leq r$, it follows that $\lambda_{G'}(Z) \geq \lambda_G(Z) - r$. Moreover, if there is no such vertex $v \in \text{bd}_G(Z) \cap V(e)$ with $N_G^{\text{inc}}(v) \setminus \{e\} \subseteq Z$, then $\text{bd}_G(Z) = \text{bd}_{G'}(Z)$. $\triangleleft$

By [Claim 5.6](#), for a set $Z \subseteq E(G)$ with $e \notin Z$, the only thing that can happen when deleting e is that the boundary might decrease and that Z might become internally connected, and this happens only if there is a vertex $v \in V(e) \cap \text{bd}_G(Z)$ with $N_G^{\text{inc}}(v) \setminus \{e\} \subseteq Z$. We find all chips Z of G' with a vertex $v \in V(e) \cap \text{bd}_G(Z)$ and $N_G^{\text{inc}}(v) \setminus \{e\} \subseteq Z$ as follows. If $|N_G^{\text{inc}}(v) \setminus \{e\}| > s_2$, there is no chip Z with $N_G^{\text{inc}}(v) \setminus \{e\} \subseteq Z$. For all other vertices $v \in V(e)$, we call the algorithm from [Claim 5.3](#) on G with $I = N_G^{\text{inc}}(v) \setminus \{e\}$, $X = \{v\}$, and a relaxed bound $\lambda(Z) \leq k + r$ on the boundary size, which takes time $\mathcal{O}_{r,s_2,k}(\log |E(G)|)$ and returns at most $\mathcal{O}_{s_2,k}(1)$ chips. We remark that the algorithm from [Claim 5.3](#) can clearly work with this relaxed bound on λ . We check whether $\text{bd}_{G'}(Z) \leq k$ in time $\mathcal{O}_{s_2,r,k}(1)$ and if so, add Z to $\mathcal{S}_{\text{bd}_{G'}(Z)}$, and update the rank $\text{vol}(\text{bd}_{G'}(Z))$ in the max-heap in time $\mathcal{O}_{r,s_2,k}(\log |E(G)|)$. Before inserting Z into $\mathcal{S}_{\text{bd}_{G'}(Z)}$, we also check whether Z is in $\mathcal{S}_{\text{bd}_G(Z)}$, and if so, remove it from there and update the rank $\text{vol}(\text{bd}_G(Z))$ in the max-heap.

Query: First, we find the set $B \subseteq V(G)$ of size $|B| \leq k$ with the maximum value of $\text{vol}(B)$ in the max-heap in time $\mathcal{O}(1)$, which also points to the corresponding balanced binary search tree $\mathcal{S}_B$. If $\text{vol}(B) < s_1$, we can report that there is no suitable set C of size $s_1 \leq |C| \leq s_2$. Otherwise, we iterate over (at most) the first s_2 sets in the balanced binary search tree $\mathcal{S}_B$. If we find a chip $Z \in \text{chips}(B)$ with $|Z| \geq s_1/2$, we set $C := Z$ and return. Clearly, this set C satisfies all the requirements as Z is a chip, so Z is internally connected, $s_1/2 \leq |Z| \leq s_2$, $\lambda(Z) \leq k$, and $\mathcal{O}(Z) = \top$.

If we do not find such a large set Z , then the first s_2 sets (or less, if $\mathcal{S}_B$ contains less than s_2 sets) in $\mathcal{S}_B$ all have size smaller than $s_1/2$. In this case, we can find an arbitrary subset $\mathcal{Z}$ of these first (at most) s_2 chips such that $s_1/2 \leq \text{vol}(\mathcal{Z}) < s_1 \leq s_2$ and we set $C = \bigcup_{Z \in \mathcal{Z}} Z$. We now show that all chip pieces of $\text{chips}(B)$ are disjoint.

Claim 5.7. *Let G be a hypergraph and $A, B \subseteq E(G)$ be two internally connected sets with $\text{bd}(A) = \text{bd}(B)$. If $A \cap B \neq \emptyset$, then $A = B$.*

Proof of the claim. Let $e \in A \cap B$ and suppose $A \neq B$. First, we argue that there is a vertex $v \in V(e) \cap \text{int}(A) \cap \text{int}(B)$. For this, observe that $V(e) \cap \text{int}(A) = V(e) \setminus \text{bd}(A) = V(e) \setminus \text{bd}(B) = V(e) \cap \text{int}(B)$, so every vertex in $V(e) \cap \text{int}(A)$ is also in $V(e) \cap \text{int}(B)$. Moreover, there is a vertex $v \in V(e) \cap \text{int}(A)$, since otherwise $V(e) \subseteq \text{bd}(A) = \text{bd}(B)$, which implies by the definition of internal connectivity $|A| = |B| = 1$, so $A = B = \{e\}$ contradicting $A \neq B$. So, let $v \in V(e) \cap \text{int}(A) \cap \text{int}(B)$.

Without loss of generality, we assume that $A \setminus B \neq \emptyset$ and consider a hyperedge $e' \in A \setminus B$. Note that $V(e') \subseteq V(B)$ is clearly not possible since $V(e') \cap \text{int}(B) = \emptyset$ and $V(e') \not\subseteq \text{bd}(B)$ by the definition of internal connectivity. So, let $v' \in V(e') \setminus V(B)$ and note that then $v' \notin \text{bd}(B) = \text{bd}(A)$, so $v' \in \text{int}(A)$. Since $v, v' \in \text{int}(A)$ and A is internally connected, there is a path between v and v' in $\mathcal{P}(G)[\text{int}(A)]$. This is a contradiction since $\text{int}(A) \cap \text{bd}(B) = \emptyset$, so this is a path from a vertex $v' \notin V(B)$ to a vertex $v \in \text{int}(B)$ that does not use vertices from $\text{bd}(B)$. $\triangleleft$

Note that since $\text{vol}(B) \geq s_1$ there is a chip $Z \in \text{chips}(B) \setminus \mathcal{Z}$, and since by Claim 5.7 the chips in $\text{chips}(B)$ are disjoint, we have $C \cap Z = \emptyset$. Thus, $\text{bd}(C) = \text{bd}(Z) = B$, so $\lambda(C) = |B| \leq k$, and every internal component C' of C is a chip $C' \in \text{chips}(B)$, so $\text{bd}(C') = B = \text{bd}(C)$ and $\text{O}(C') = \top$.

Since we only need to look at (at most) the first s_2 sets in $\mathcal{S}_B$, determine their size, and possibly take the union of these sets, the Query operation can be done in time $\mathcal{O}_{s_2, r, k}(1)$. $\square$

6 Balancing protrusions

Recall that our goal is to build a dynamic protrusion decomposition. This can be seen as having two main challenges, (1) maintaining a root with small bag and low-degree, and (2) maintaining protrusions with small treewidth. In the previous sections, we built a data structure, that will enable us to handle the first challenge, and in this section we will build a data structure that will enable us to handle the second challenge. In Section 7, we will combine the two parts to maintain the full protrusion decomposition.

The techniques of this section are based on the dynamic treewidth data structure of [Kor25a]. We essentially want to use a copy of this data structure for each of our protrusions. Unfortunately, we cannot do so in a completely black-box manner, because we will sometimes need to split and merge protrusions. Nevertheless, our data structure will largely reuse subroutines from [Kor25a]. What we obtain is captured in Lemma 6.1, which provides a dynamic protrusion decomposition data structure, but leaves the maintenance of the degree of the root to the user of the data structure.

Lemma 6.1. *There is a data structure that, initialized with an integer $c \geq 3$, maintains a graph G , the support hypergraph $\mathcal{H}(G)$ of G , a downwards well-linked superbranch decomposition $\mathcal{T}$ of $\mathcal{H}(G)$ with adhesion size at most c , as well as an annotated normal $(\Delta(r) \cdot c, \mathcal{O}_c(1))$ -protrusion decomposition $\tilde{\mathcal{T}}$ corresponding to $\mathcal{T}$ under the following operations:*

- **Init(c):** *Given an integer $c \geq 3$, initialize the data structure with c and an empty graph G .*
- **AddVertex(v):** *Given a new vertex $v \notin V(G)$, add v to G .*
- **DeleteVertex(v):** *Given an isolated vertex $v \in V(G)$, remove v from G .*
- **AddEdge(e):** *Given a new edge $e \in \binom{V(G)}{2} \setminus E(G)$, add e to G .*
- **DeleteEdge(e):** *Given an edge $e \in E(G)$, remove e from G .*

- **Merge(A):** Given a proper non-empty subset $A \subsetneq \text{chd}(r)$ of root-children with $2 \leq |A| \leq 2^{2c} + 1$ and $|\overline{A}| \geq 2$, corresponding to a set $B_A \subseteq E(\text{torso}(r))$ such that $B_A \triangleright \mathcal{T}$ is well-linked in $\mathcal{H}(G)$ and $\text{wl}(B_A \triangleright \mathcal{T}) \leq c$, add a new root-child node r' and change the parent of each node in A to be r' instead of r , leaving T otherwise unchanged.

The **Init** operation runs in $\mathcal{O}(1)$ time, the **Merge** operation runs in $\mathcal{O}_c(1)$ amortized time, and the remaining operations each run in $\mathcal{O}_c(\log |G|)$ amortized time. For each operation, the changes to $\text{torso}(r)$ can be described as a sequence $\mathcal{C}$ of hypergraph operations of size $\mathcal{O}_c(1)$, which is returned. Changes to $\text{edges}(r)$ can be described as a list of insertions and deletions of size at most $\mathcal{O}_c(1)$, which is returned.

Moreover, if upon initialization the data structure is provided a tree decomposition automaton $\mathcal{A}$ with evaluation time τ , then a run of $\mathcal{A}$ on each protrusion is maintained, incurring an additional τ factor on the running times. If the automaton state at the root-child node corresponding to an hyperedge $e \in E(\text{torso}(r))$ is modified by an operation, the corresponding sequence $\mathcal{C}$ will contain the operation **DeleteHyperedge**(e).

The remainder of this section consists of proving this lemma. In [Section 6.1](#), we introduce a standardized set of subroutines through which we will modify our superbranch decomposition, based on [\[Kor25a\]](#). In [Section 6.2](#), we generalize tree-balancing subroutines from [\[Kor25a\]](#) to be suitable in our setting, and then in [Section 6.3](#), we provide the full data structure of [Lemma 6.1](#).

6.1 Basic rotations

In this subsection, we describe four subroutines for local transformation of superbranch decompositions, which we call *basic rotations*. Each basic rotation modifies only a local part of the tree and updates the whole representation, including torsos accordingly. Our basic rotations correspond to those of [\[Kor25a\]](#), although we must be a bit more careful here, to avoid spending time linear in the root size. In the following, let G be a hypergraph and $\mathcal{T} = (T, \mathcal{L})$ a superbranch decomposition of G .

Contracting. Let $pt \in E(T)$ be an edge of T so that p is the parent of t and $t \in V_{\text{int}}(T)$. *Contracting* the edge pt in $\mathcal{T}$, simply means contracting pt in T while preserving the leaf mapping $\mathcal{L}$. In the following lemma, we give an efficient algorithm for contracting an edge of a superbranch decomposition.

Lemma 6.2. *A representation of $\mathcal{T}$ can be turned into a representation of $\mathcal{T}$ with pt contracted in time $\mathcal{O}(\text{adhsz}(\mathcal{T}) \cdot |\text{torso}(t)|)$. Torso changes can be described as a sequence of hypergraph operations of size at most the running time, and these sequences are returned.*

Proof. Let $\mathcal{T}' = (T', \mathcal{L}')$ denote the new superbranch decomposition. We start by transforming T into T' , so that p corresponds to the new contracted node. This is done by deleting t and its incident edges from T , and then inserting an edge ps for each $s \in \text{chd}(t)$. This takes $\mathcal{O}(|\text{torso}(t)|)$ time. To transform $\text{torso}_{\mathcal{T}}(p)$ to $\text{torso}_{\mathcal{T}'}(p)$, we start by deleting e_t corresponding to $\text{adh}_{\mathcal{T}}(pt)$ in time $\mathcal{O}(|V(e_t)|) = \mathcal{O}(|\text{torso}(t)|)$. Then, for each hyperedge $e \in E(\text{torso}(t)) \setminus \{e_t\}$, we insert e into the bipartite representation of $\text{torso}(p)$:

- add a vertex corresponding to e in $\mathcal{O}(1)$ time, and
- for each $v \in V(e)$ check whether $v \in V(\text{torso}_{\mathcal{T}'}(p))$. If so, add an edge between the existing vertex for v and the new vertex for e . Otherwise, create a new vertex for v and add the edge.

Note that $v \in V(\text{torso}_{\mathcal{T}}(p))$ if and only if $v \in \text{adh}_{\mathcal{T}}(pt)$, so we can check this and retrieve a pointer to the vertex in $\mathcal{O}(\text{adhsize}(\mathcal{T}))$ time. As there are $\mathcal{O}(|\text{torso}(t)|)$ edges to insert, the total running time is $\mathcal{O}(\text{adhsize}(\mathcal{T}) \cdot |\text{torso}(t)|)$.

During this transformation, we can also construct a sequence $\mathcal{C}$ of hypergraph operations that describes the change of $\text{torso}(p)$. The sequence consists of

- $\text{DeleteEdge}(e_t)$,
- for each $e \in E(\text{torso}(t) \setminus \{e_t\})$: $\text{AddVertex}(v)$ for each new $v \in V(e)$ and $\text{AddEdge}(e)$

The size of $\mathcal{C}$ is upper bounded by $\text{adhsize}(\mathcal{T}) + |\text{torso}(t)| = \mathcal{O}(\text{adhsize}(\mathcal{T}) \cdot |\text{torso}(t)|)$ $\square$

Splitting. Let $t \in V_{\text{int}}(T)$ be an internal node and $C \subseteq E(\text{torso}(t))$ a set of hyperedges with $|C| \geq 2$ and $|\overline{C}| \geq 2$. We assume that C does not contain a hyperedge corresponding to the parent of t (if it exists). *Splitting* the subset C from t in $\mathcal{T}$, means replacing t with two new adjacent nodes, t' and t_C , so that t' is adjacent to every node s with $e_s \in \overline{C}$ and t_C is adjacent to every node s with $e_s \in C$.

Lemma 6.3. *A representation of $\mathcal{T}$ can be turned into a representation with C splitting from t , in time $\mathcal{O}(\text{adhsize}(\mathcal{T}) \cdot |C|)$. Torso changes can be described as a sequence of hypergraph operations of size at most the running time, and these sequences are returned.*

Proof. Let $\mathcal{T}' = (T', \mathcal{L}')$ denote the new superbranch decomposition. We first transform T into T' . Rename t into t' , add a new node t_C , and insert the edge $t't_C$. For every child s of t with $e_s \in C$, remove the edge ts and instead connect t_C to s . This step takes $\mathcal{O}(|C|)$ time.

Next, we update the adhesions. For each s with $e_s \in C$, we set $\text{adh}_{\mathcal{T}'}(t_C c) = \text{adh}_{\mathcal{T}}(tc)$. For the edge $t't_C$ we set $\text{adh}_{\mathcal{T}'}(t't_c) = \text{bd}(C)$, where $\text{bd}(C)$ can be computed in time $\mathcal{O}(|C| \cdot \text{adhsize}(\mathcal{T}))$ by Lemma 3.4.

To obtain $\text{torso}(t')$, remove all hyperedges of C from $\text{torso}(t)$ and remove every vertex v that becomes isolated after removing C , and finally add the new hyperedge e_{t_C} with $V(e_{t_C}) = \text{bd}(C)$. To obtain $\text{torso}(t_C)$, copy the part of $\text{torso}(t)$ corresponding to edges in C and add e_{t_C} . Both steps take time $\mathcal{O}(|C| \cdot \text{adhsize}(\mathcal{T}))$.

Performing these transformations, we can simultaneously construct a sequence $\mathcal{C}$ describing the change of $\text{torso}(t)$ to $\text{torso}(t')$. The sequence consists of

- $\text{DeleteEdge}(e)$ for each $e \in C$,
- $\text{AddEdge}(e_{t_C})$, and
- $\text{DeleteVertex}(v)$ for each vertex v that becomes isolated after deleting C .

The size of $\mathcal{C}$ is upper bounded by $\mathcal{O}(\text{adhsize}(\mathcal{T}) \cdot |C|)$. Similarly, the changes of $\text{torso}(t_C)$ can be described by a sequence $\mathcal{C}'$ consisting of

- $\text{AddVertex}(v)$ for each $v \in V(C)$,
- $\text{AddEdge}(e)$ for each $e \in C$,
- $\text{AddEdge}(e_{t_C})$.

The size of $\mathcal{C}'$ is upper bounded by $\mathcal{O}(\text{adhsize}(\mathcal{T}) \cdot |C|)$. $\square$

Inserting a leaf. We use this leaf insert operation to insert a new hyperedge e to G . This operation may also introduce new vertices, where all vertices in $V(e) \setminus V(G)$ are added to G together with e . To insert the hyperedge into the superbranch decomposition, we proceed as follows.

Let $t \in V_{\text{int}}(T)$ be an internal node of T , and let $\text{cl}(t) = \text{chd}(t) \cap L(T)$ denote the set of leaf-children of t . Now, for a hyperedge $e \notin E(G)$ with $V(e) \cap V(G) \subseteq V(\mathcal{L}(\text{cl}(t)))$, inserting e as a child of t in $\mathcal{T}$ means adding a leaf-node ℓ_e as a child of t in T , and setting $\mathcal{L}(\ell_e) = e$.

To obtain an algorithm independent of $|\text{torso}(t)|$, we take as an additional input a set $X \subseteq \text{cl}(t)$ with $V(e) \cap V(G) \subseteq V(\mathcal{L}(X))$. Since only edges incident to X are affected by the insertion, $\text{torso}(t)$ can be updated by considering X rather than the entire torso.

Lemma 6.4. *Given a new hyperedge e and $X \subseteq \text{cl}(t)$ with $V(e) \cap V(G) \subseteq V(\mathcal{L}(X))$, a representation of $\mathcal{T}$ can be turned into a representation of $\mathcal{T}$ with e inserted as a child of t in time $\mathcal{O}(|X| \cdot \text{rk}(G) + 1) \cdot |V(e)| + |\text{anc}(t)|$. Torso changes can be described as a sequence of hypergraph operations of size at most the running time, and these sequences are returned.*

Proof. Let $\mathcal{T}'$ denote the new tree decomposition and let e' denote the hyperedge with $V(e') = V(e) \cap V(G)$. Since $V(e') \subseteq V(\mathcal{L}(X))$, any edge uv of T' not incident to a child in X satisfies $\text{adh}_{\mathcal{T}'}(vt) = \text{adh}_{\mathcal{T}}(vt)$. Hence, only adhesions between t and the children in X should be updated. For $x \in X$ and $v \in V(\mathcal{L}(x))$, note that $v \notin \text{adh}_{\mathcal{T}'}(xt)$ implies $v \notin V(\text{torso}(t))$. So, to update $\text{adh}_{\mathcal{T}'}(xt)$ and the corresponding hyperedge $e_x \in E(\text{torso}(t))$, whenever $v \in V(e')$ and $v \notin \text{adh}_{\mathcal{T}}(xt)$ we insert v into $\text{adh}_{\mathcal{T}'}(xt)$, add v to $V(\text{torso}(t))$, and add v to e_x in the bipartite representation of $\text{torso}(t)$. This takes $\mathcal{O}(|X| \cdot \text{rk}(G) \cdot |V(e)|)$ time.

Next, we add a new leaf w with edge tw , set $\text{adh}_{\mathcal{T}'}(tw) = V(e')$, and insert the hyperedge e' into $\text{torso}(t)$. Since $V(e') \subseteq V(\mathcal{L}(X))$, all vertices of $V(e')$ can be identified in the bipartite representation of $\text{torso}(t)$ by scanning the hyperedges corresponding to X , so the insertion runs in time $\mathcal{O}(|X| \cdot \text{rk}(G) \cdot |V(e)|)$. Finally, we need to increase the stored number of descendant leaves for the ancestors of t in time $\mathcal{O}(|\text{anc}(t)|)$.

During this transformation, we can also construct a sequence $\mathcal{C}$ of hypergraph operations that describes the change of $\text{torso}(t)$. The sequence consists of

- **AddVertex**(v) for each new $v \in V(e') \setminus V(\text{torso}_{\mathcal{T}}(t))$,
- **DeleteEdge**(e_x) for each $e_x \in E(\text{torso}_{\mathcal{T}}(t))$ with $x \in X$,
- **AddEdge**(e_x) for each $e_x \in E(\text{torso}_{\mathcal{T}'}(t))$ with $x \in X$, and
- **AddEdge**(e').

The size of $\mathcal{C}$ is upper bounded by $|V(e)| + 2 \cdot |X| \cdot (\text{rk}(G) + 1) + (|V(e)| + 1)$. □

Deleting a leaf. Let $t \in V_{\text{int}}(T)$ be an internal node of T with at least 3 children. Now, for a leaf $\ell \in \text{cl}(t)$ with $V(\mathcal{L}(\ell)) \cap V(G \setminus \mathcal{L}(\ell)) \subseteq V(\mathcal{L}(\text{cl}(t) \setminus \{\ell\}))$, deleting ℓ from $\mathcal{T}$ means deleting ℓ and the edge $t\ell$ from T .

Lemma 6.5. *A representation of $\mathcal{T}$ can be turned into a representation of $\mathcal{T}$ with ℓ deleted in time $\mathcal{O}(\text{adhsize}(\mathcal{T})^2 + |\text{anc}(t)|)$. Torso changes can be described as a sequence of hypergraph operations of size at most the running time, and these sequences are returned.*

Proof. Let $\mathcal{T}'$ denote the new tree decomposition. Since $V(\mathcal{L}(\ell)) \cap V(G \setminus \mathcal{L}(\ell)) \subseteq V(\mathcal{L}(\text{cl}(t) \setminus \{\ell\}))$, any edge uv of T' not incident to a child of t satisfies $\text{adh}_{\mathcal{T}'}(vt) = \text{adh}_{\mathcal{T}}(vt)$. Hence, only adhesions between t and the children should be updated.

For a child s , a vertex $v \in \text{adh}_{\mathcal{T}}(ts)$ may need to be removed only if $v \in V(e_\ell)$. This is the case if v is incident only to e_s and e_ℓ in $\text{torso}_{\mathcal{T}}(t)$. This also means that each $v \in V(e_\ell)$ can affect

at most one adhesion, which can be updated in $\mathcal{O}(\text{adhsz}(\mathcal{T}))$ time. As $|V(e_\ell)| \leq \text{adhsz}(\mathcal{T})$ this takes $\mathcal{O}(\text{adhsz}(\mathcal{T})^2)$ time. Finally, we remove ℓ from T , and then we need to decrease the stored number of descendant leaves for the ancestors of t in time $\mathcal{O}(|\text{anc}(t)|)$.

During this transformation, we can also construct a sequence $\mathcal{C}$ of hypergraph operations that describes the change of $\text{torso}(t)$. The sequence consists of

- $\text{DeleteEdge}(e_\ell)$,
- for each hyperedge that loses a vertex of $V(e_\ell)$, update it by performing DeleteEdge followed by AddEdge ,
- $\text{DeleteVertex}(v)$ for each $x \in V(e_\ell)$ that becomes isolated after deleting e_ℓ .

The size of $\mathcal{C}$ is upper bounded by $\mathcal{O}(\text{adhsz}(\mathcal{T})^2)$. $\square$

Representing basic rotations. We will denote a basic rotation by s , where s encodes the type of rotation (contraction, split, leaf insertion, or leaf deletion) together with the information required to perform it: for a contraction of an edge st , the pair (s, t) is stored; for a split of a node t with edge set C , the pair (t, C) is stored; for inserting a leaf at t with respect to a set X , the pair (t, X) is stored; and for deleting a leaf ℓ , the node ℓ is stored.

If $\mathcal{T}$ and $\mathcal{T}'$ are the superbranch decompositions before and after applying a basic rotation s , we say that s *transforms* $\mathcal{T}$ into $\mathcal{T}'$. The size of a rotation, denoted $\|s\|$, is defined so that applying s takes time $\mathcal{O}(\|s\|)$. Specifically, a contraction has size $\text{adhsz}(\mathcal{T}) \cdot |\text{torso}(t)|$, a split has size $\text{adhsz}(\mathcal{T}) \cdot |C|$, a leaf insertion has size $(|X| \cdot \text{adhsz}(\mathcal{T}) + 1) \cdot |V(e)| + |\text{anc}(t)|$, and a leaf deletion has size $\text{adhsz}(\mathcal{T})^2 + |\text{anc}(t)|$.

A *sequence of basic rotations* is a finite sequence $\mathcal{S} = s_1 s_2 \cdots s_k$, where each s_i is a basic rotation as described above. Applying $\mathcal{S}$ to a superbranch decomposition $\mathcal{T}$ *transforms* $\mathcal{T}$ into a new decomposition $\mathcal{T}'$ by applying s_1 to s_k one sequentially. The *size* of the sequence is $\|\mathcal{S}\| = \sum_i (\|s_i\| + 1)$, so that applying $\mathcal{S}$ takes time $\mathcal{O}(\|\mathcal{S}\|)$. We define $V_{\mathcal{T}}(\mathcal{S})$ as the set of nodes of T involved in the rotations: all internal nodes involved in a contraction or a split, and all nodes deleted or inserted together with their parents. Analogously, $V_{\mathcal{T}'}(\mathcal{S})$ denotes the corresponding set in the transformed decomposition. The *trace* of $\mathcal{S}$ with respect to $\mathcal{T}$ is $\text{trace}_{\mathcal{T}}(\mathcal{S}) = \text{anc}(V_{\mathcal{T}}(\mathcal{S}))$, the set of ancestors of nodes involved in the sequence. We define $\text{trace}_{\mathcal{T}'}(\mathcal{S})$ analogously. Finally, we let $\|\mathcal{S}\|_{\mathcal{T}} = \|\mathcal{S}\| + |\text{trace}_{\mathcal{T}}(\mathcal{S})|$. Observe that $|\text{trace}_{\mathcal{T}'}(\mathcal{S})| \leq \|\mathcal{S}\|_{\mathcal{T}}$.

6.2 Balancing subroutines

In this section, we introduce balancing subroutines that allow us to maintain the invariants of the superbranch decomposition under dynamic updates. To this end, we define the notions of c -semigood and c -good. The property of being c -good captures exactly what we wish to maintain: bounded degree and logarithmic depth. However, we sometimes revert to maintaining the weaker condition of c -semigood and afterwards restoring the c -good property by applying a dedicated subroutine. The definitions and subroutines of this subsection are taken from [Kor25a] with slight adaptations to our setting.

c -good and c -semigood. Let $\mathcal{T} = (T, \mathcal{L})$ be a rooted superbranch decomposition of a hypergraph G . For a node $x \in V(T)$, we say that $\mathcal{T}$ is *c -semigood at x* , if

- for every $v \in V(T_x)$, the set $\mathcal{L}[v]$ is well-linked in G (i.e., the subtree is downwards well-linked),
- $\Delta(T_x) \leq 2^{2c} + 1$,

- $\text{wl}(\mathcal{L}[x]) \leq c$.

For a node $x \in V(T)$, we say that x is d -unbalanced for an integer $d \geq 1$ if there exists a $s \in \text{desc}(t)$, so that $\text{depth}(s) \geq \text{depth}(t) + d$ and $|\mathcal{L}[s]| \geq (2/3) \cdot |\mathcal{L}[x]|$. A node $x \in V(T)$ is d -balanced if it is not d -unbalanced, i.e., no such descendant exists. For a node $x \in V(T)$, we say that $\mathcal{T}$ is c -good at x , if

- every $v \in V(T_x)$ is 2^{2c+1} -balanced,
- $\mathcal{T}$ is c -semigood at x .

We say that $\mathcal{T}$ is c -good (respectively, c -semigood) if, for every root child t , $\mathcal{T}$ is c -good (respectively, c -semigood) at t .

Being c -good at a node implies that the corresponding part of the decomposition has logarithmic height, by the following lemma.

Lemma 6.6 ([Kor25a, Lemma 6.2]). *Let G be a hypergraph and $\mathcal{T} = (T, \mathcal{L})$ a rooted superbranch decomposition of G . For a node $x \in V(T)$, if $\mathcal{T}$ is c -good at x then $\text{depth}(T_x) \leq 2^{\mathcal{O}(c)} \log |T_x|$*

Potential functions. The analysis of the balancing subroutines is based on potential functions. As in [Kor25a], we define the potential of a single internal node $t \in V_{\text{int}}(T)$ as

$$\Phi_{\mathcal{T}}(t) = (\Delta(t) - 1) \log_2 |\mathcal{L}[t]|.$$

Note that all internal nodes then have potential at least 1. We then define the potential of $\mathcal{T}$ as

$$\Phi(\mathcal{T}) = \sum_{t \in V_{\text{int}}(T)} \Phi_{\mathcal{T}}(t).$$

Balancing subroutines. We now give the balancing subroutines, described in the two lemmas below. The first shows how to restore the c -good property at a node.

Lemma 6.7 (Generalization of [Kor25a, Lemma 7.1]). *Let G be a hypergraph, $\mathcal{T} = (T, \mathcal{L})$ a superbranch decomposition of G rooted at r , and $c \geq 1$ an integer. Suppose also that t is a root child for which $\mathcal{T}$ is c -semigood at t . There is an algorithm that, given c , t , and a prefix $R \subseteq V(T_t)$ of T_t containing all 2^{2c+1} -unbalanced nodes of T_t , transforms $\mathcal{T}$ into a superbranch decomposition $\mathcal{T}' = (T', \mathcal{L}')$ of G rooted at r' via a sequence $\mathcal{S}$ of basic rotations, which is returned, such that,*

- $\Phi(\mathcal{T}') \leq \Phi(\mathcal{T})$,
- $V_{\mathcal{T}}(\mathcal{S}) \subseteq \text{desc}(t)$,
- $\mathcal{T}'$ is c -good at t' ,

where t' is the child of r' with $V_{\mathcal{T}'}(\mathcal{S}) \subseteq \text{desc}(t')$ if $|\mathcal{S}| > 0$, and $t' = t$ otherwise. The running time of the algorithm is $2^{\mathcal{O}(c)}(|R| + \Phi(\mathcal{T}) - \Phi(\mathcal{T}'))$, which is also an upper bound for $\|\mathcal{S}\|_{\mathcal{T}}$.

Proof. The lemma from [Kor25a] only works on the root of tree decompositions, not on subtrees of tree decompositions. In [Kor25a] tree decompositions have a degree-one “dummy root”, making the unique node at depth 1 (what we think of as the root) an internal node. This means that their situation actually is very similar to ours, where our root child t acts as the root a tree decomposition of the protrusion $\mathcal{L}[t]$. The main difference is that our “root” t can have a boundary/adhesion to nodes outside $\mathcal{L}[t]$. The proof carries through, essentially without modification, so we will not repeat it here. Instead, we mention the following changes to be made to the proof of [Kor25a] (which will not make sense without reading [Kor25a]). Instead

of assuming that $\mathcal{T}$ is c -good and that $\text{wl}(G) \leq c$, we should assume this for the subtree we wish to balance, and R should be a prefix of this subtree, containing all the unbalanced nodes of the subtree. (This is what we do in our statement.) Their proof uses their Lemma 5.6 and 5.4, where $\text{wl}(G)$ should be replaced by $\text{wl}(\mathcal{L}[t])$. Finally, note that our definition of semigood includes bounded well-linked number. $\square$

The next balancing subroutine allows us to move a hyperedge close to the root. This will be useful when inserting and deleting hyperedges in the data structure.

Lemma 6.8 (Generalization of [Kor25a, Lemma 8.1]). *Let G be a hypergraph, and $\mathcal{T} = (T, \mathcal{L})$ a superbranch decomposition of G rooted at r and $c \geq 3$ an integer. Suppose also that t is a root child such that $\mathcal{T}$ is c -good at t . There is an algorithm that, given c , t , and a hyperedge $e \in \mathcal{L}[t]$ with $|V(e)| \leq 2$, transforms $\mathcal{T}$ into a superbranch decomposition $\mathcal{T}' = (T', \mathcal{L}')$ of G rooted at r' via a sequence $\mathcal{S}$ of basic rotations, which is returned, such that,*

- $\Phi(\mathcal{T}') \leq \Phi(\mathcal{T}) + 2^{\mathcal{O}(c)} \log |G|$,
- $V_{\mathcal{T}}(\mathcal{S}) \subseteq \text{desc}(t)$,
- $\mathcal{L}'^{-1}(e) \in \text{chd}(t')$,
- $\mathcal{T}'$ is c -semigood at t' ,

where t' is the child of r' with $\mathcal{L}'^{-1}(e) \in \text{desc}(t')$. The running time of the algorithm is $2^{\mathcal{O}(c)} \log |G|$, which is also an upper bound for $\|\mathcal{S}\|_{\mathcal{T}}$.

Proof. This lemma follows from [Kor25a] in a simliarily straightforward manner as Lemma 6.7. We again mention the trivial changes to be made to their proof. Instead of assuming that $\mathcal{T}$ is c -good and that $\text{wl}(G) \leq c$, we should assume this for the subtree rooted at the node t we wish to rotate to, and depth should be measured against this node instead of the unique node at depth 1. (This is what we do in our statement.) Their Lemma 8.1 uses Lemma 8.2, 5.6, and 5.4, where $\text{wl}(G)$ should be replaced by $\text{wl}(\mathcal{L}[t])$. $\square$

6.3 Proofs for the protrusion-balancing data structure

In this subsection we prove the correctness and time bounds of the balancing data structure of Lemma 6.1. First, we give a subroutine for rotating leaves upwards until they become root children.

Lemma 6.9. *Let $\mathcal{T} = (T, \mathcal{L})$ be c -good superbranch decomposition rooted at r of $\mathcal{H}(G)$, for an integer $c \geq 3$. There is an algorithm that, given c and a set of hyperedges $X \subseteq E(\mathcal{H}(G))$ with $|X| \leq 3$, transforms $\mathcal{T}$ into a c -good superbranch decomposition $\mathcal{T}' = (T', \mathcal{L}')$ rooted at r' via a sequence $\mathcal{S}$ of basic rotations, so that*

- $\mathcal{L}'^{-1}(X) \subseteq \text{chd}(r')$ and
- $\Phi(\mathcal{T}') \leq \Phi(\mathcal{T}) + \mathcal{O}_c(\log |G|)$.

The running time is $\mathcal{O}_c(\log |G| + \max\{\Phi(\mathcal{T}) - \Phi(\mathcal{T}_3), 0\})$, which is also an upper bound for $\|\mathcal{S}\|_{\mathcal{T}}$. Furthermore, $\text{torso}_{\mathcal{T}'}(r')$ can be obtained from $\text{torso}_{\mathcal{T}}(r)$ via a sequence $\mathcal{C}$ of basic hypergraph operations of size $\mathcal{O}_c(1)$. If the subtree of $\mathcal{T}$ rooted at the node corresponding to an edge $e \in E(\text{torso}_{\mathcal{T}}(r))$ is modified by the algorithm, then $\mathcal{C}$ is guaranteed to contain the operation $\text{DeleteHyperedge}(e)$. The algorithm returns $\mathcal{S}$ and $\mathcal{C}$.

Proof. We first describe an algorithm that satisfies all the claimed properties but takes a single hyperedge e as input instead of a set X . The full algorithm for a set X is then obtained by applying it once for each $e \in X$. To ensure that all hyperedges in X end up as children of the root, the algorithm will also guarantee that for every $x \in \text{chd}(r) \setminus \{t\}$, where t is the root child with $e \in \mathcal{L}[t]$, the node x remains a child of the new root r' and the subtree rooted at x is left unchanged. This way, once a hyperedge is moved to the root, it is not moved away again in later calls.

The algorithm is divided into three steps, each producing a new superbranch decomposition $\mathcal{T}_i$, with $\mathcal{T}_3 = (T_3, \mathcal{L}_3)$ being the final superbranch decomposition that is c -good and satisfies $\mathcal{L}_3^{-1}(e) \in \text{chd}(r_3)$ where r_3 is the root.

Step 1. In this step, we will rotate e close to the root. First identify the root child $t \in \text{chd}(r)$ with $e \in \mathcal{L}[t]$ in time $\mathcal{O}(\text{depth}(\mathcal{T})) = \mathcal{O}_c(\log |G|)$. By applying [Lemma 6.8](#) to $\mathcal{T}$ with c , t and e , we obtain in time $\mathcal{O}_c(\log |G|)$ a superbranch decomposition $\mathcal{T}_1 = (T_1, \mathcal{L}_1)$ rooted at r_1 via a sequence $\mathcal{S}_1$ of basic rotations, so that,

- $\Phi(\mathcal{T}_1) \leq \Phi(\mathcal{T}) + \mathcal{O}_c(\log |G|)$,
- $V_{\mathcal{T}}(\mathcal{S}_1) \subseteq \text{desc}(t)$,
- $\|\mathcal{S}_1\|_{\mathcal{T}} \leq \mathcal{O}_c(\log |G|)$,
- $\mathcal{T}_1$ is c -semigood at t_1 ,
- $\mathcal{L}_1^{-1}(e) \in \text{chd}(t_1)$,

where t_1 is the child of r_1 with $V_{\mathcal{T}}(\mathcal{S}) \subseteq \text{desc}(t_1)$. Because $V_{\mathcal{T}}(\mathcal{S}_1) \subseteq \text{desc}(t)$, the only difference between $\mathcal{T}$ and $\mathcal{T}_1$ is the subtree at t_1 , in the sense that replacing the subtree $(T_1)_{t_1}$ with T_t gives a superbranch decomposition identical to $\mathcal{T}$. This means that for each $s \in \text{chd}(r_1) \setminus \{t_1\}$, the subtree rooted at s is not touched by the algorithm, so $\mathcal{T}_1$ is still c -good at s . This also means that $\text{torso}_{\mathcal{T}}(r) = \text{torso}_{\mathcal{T}_1}(r_1)$.

Step 2. Since $\mathcal{L}_1^{-1}(e) \in \text{chd}(t_1)$, we can make $\mathcal{L}_1^{-1}(e)$ a root child by contracting the edge rt_1 using the basic rotation of [Lemma 6.2](#) in $\mathcal{O}(\text{adhsz}(\mathcal{T}) \cdot |\text{torso}(t_1)|) = \mathcal{O}_c(1)$ time. Let $\mathcal{T}_2 = (T_2, \mathcal{L}_2)$ be the resulting superbranch decomposition rooted at r_2 (the contracted node). By the lemma, we further have that $\text{torso}_{\mathcal{T}_2}(r_2)$ can be obtained from $\text{torso}_{\mathcal{T}_1}(r_1)$ by a sequence $\mathcal{C}$ of size $\mathcal{O}_c(1)$. As $\Delta(r_2)$ increases by at most $\mathcal{O}_c(1)$, the potential increases by at most $\mathcal{O}_c(\log |G|)$, i.e. $\Phi(\mathcal{T}_2) \leq \Phi(\mathcal{T}_1) + \mathcal{O}_c(\log |G|)$. Let $\mathcal{S}_{1,2}$ be the concatenation of $\mathcal{S}$ and this contraction basic rotation, satisfying $\|\mathcal{S}_{1,2}\|_{\mathcal{T}} \leq \|\mathcal{S}_1\|_{\mathcal{T}} + \mathcal{O}_c(1) \leq \mathcal{O}_c(\log |G|)$.

Step 3. Finally we will transform $\mathcal{T}_2$ (which is c -semigood) into a c -good superbranch decomposition $\mathcal{T}_3$. Let $R \subseteq \text{chd}(r_2) \setminus \{\mathcal{L}_2^{-1}(e)\}$ be the set of children of r_2 corresponding to the children of t_1 in $\mathcal{T}_1$, excluding the child associated with e . Since $\mathcal{T}_1$ is c -semigood at t_1 , it follows that for each $s \in R$, the superbranch decomposition $\mathcal{T}_2$ is c -semigood at s . Note that for every other child of r_2 , the subtree is still c -good at this node. Thus, to obtain a c -good superbranch decomposition, it suffices to consider the children in R . For a node $s \in R$, define $P_s = \text{trace}_{\mathcal{T}_2}(\mathcal{S}_{1,2}) \cap V((T_2)_s)$. This is a prefix of $(T_2)_s$ that contains all of its 2^{2c+1} -unbalanced nodes, and $|P_s| \leq \|\mathcal{S}_{1,2}\|_{\mathcal{T}} \leq \mathcal{O}_c(\log |G|)$. Applying [Lemma 6.7](#) with s and P_s for each $s \in R$, transforms $\mathcal{T}_2$ into a c -good superbranch decomposition $\mathcal{T}_3$ rooted at r_3 via a sequence $\mathcal{S}_3$ of basic rotations, so that,

- $\Phi(\mathcal{T}_3) \leq \Phi(\mathcal{T}_2)$ and

- $V_{\mathcal{T}_2}(\mathcal{S}_3) \subseteq \bigcup_{s \in R} \text{desc}(s)$.

The latter implies that $\mathcal{L}_3^{-1}(e)$ is still a child of the root. The running time is $\mathcal{O}_c(\log |G| + \Phi(\mathcal{T}_2) - \Phi(\mathcal{T}_3))$, which also bounds $\|\mathcal{S}_3\|_{\mathcal{T}_2}$. Since $r_2 \notin V_{\mathcal{T}_2}(\mathcal{S}_2)$, we have $\text{torso}_{\mathcal{T}_2}(r_2) = \text{torso}_{\mathcal{T}_3}(r_3)$. Therefore we can obtain $\text{torso}_{\mathcal{T}_3}(r_3)$ from $\text{torso}_{\mathcal{T}}(r)$ via the sequence $\mathcal{C}$ of size $\mathcal{O}_c(1)$. Combining inequalities from the previous steps, $\Phi(\mathcal{T}_3) \leq \Phi(\mathcal{T}) + \mathcal{O}_c(\log |G|)$. The total running time is $\mathcal{O}_c(\log |G|) + (\Phi(\mathcal{T}_2) - \Phi(\mathcal{T}_3))$, and since $\Phi(\mathcal{T}_2) \leq \Phi(\mathcal{T}) + \mathcal{O}_c(\log |G|)$ we get the claimed running time. Let $\mathcal{S}$ be the entire sequence of basic rotations, i.e., the concatenation of $\mathcal{S}_{1,2}$ and $\mathcal{S}_3$. Then the size $\|\mathcal{S}\|_{\mathcal{T}} \leq \|\mathcal{S}_{1,2}\|_{\mathcal{T}} + \|\mathcal{S}_3\|_{\mathcal{T}_2}$ is upper bounded by the running time. Observe that $V_{\mathcal{T}}(\mathcal{S}) \subseteq \text{desc}(t) \cup \{r\}$. Since the only basic rotation involving r is the contraction, every $x \in \text{chd}(r) \setminus \{t\}$ remains a child of the new root r' and the subtree rooted at x is left unchanged. This also implies that for every hyperedge e in $E(\text{torso}_{\mathcal{T}}(r))$, if $\mathcal{C}$ does not delete e , then the subtree of $\mathcal{T}$ rooted at the node corresponding to e does not change during the transformation. Finally, we return $\mathcal{C}$ and $\mathcal{S}$. $\square$

We now prove the first part of the protrusion balancing data structure, maintaining a superbranch decomposition.

Proof of Lemma 6.1. We first show how to maintain the superbranch decomposition under all operations, before moving to the protrusion decomposition.

Superbranch decomposition. We will maintain a c -good superbranch decomposition $\mathcal{T}$ of $\mathcal{H}(G)$ and analyze the amortized running time using the potential function Φ defined in Section 6.2. Any change $\Phi(\mathcal{T}') - \Phi(\mathcal{T})$ is charged as $\mathcal{O}_c(\Phi(\mathcal{T}') - \Phi(\mathcal{T}))$ in the amortized time. Note that, since $\mathcal{T}$ is downwards well-linked and every node $t \in V(\mathcal{T}) \setminus \{r\}$ satisfies $\text{wl}(\mathcal{L}[t]) \leq c$ in G , it follows that $\text{adhsz}(\mathcal{T}) \leq c$.

In all operations other than `Init`, we will update $\mathcal{T}$ by a sequence $\mathcal{S}$ of basic rotations, which we save for use later in the proof. The sizes $\|\mathcal{S}\|_{\mathcal{T}}$ of these sequences will have the same amortized upper bound as the running times.

Furthermore, each operation will guarantee that if the subtree rooted at a node $t \in \text{chd}(r)$ corresponding to a hyperedge $e \in E(\text{torso}(r))$ is modified, then $\mathcal{C}$ will contain the operation `DeleteHyperedge`(e). This will be clear from our operation implementations, when considering the guarantee from Lemma 6.9.

We note that the algorithm of Lemma 6.9 runs in $\mathcal{O}_c(\log |G|)$ amortized time. Specifically, it transform $\mathcal{T}$ into a new superbranch decomposition $\mathcal{T}'$ such that $\Phi(\mathcal{T}') \leq \Phi(\mathcal{T}) + \mathcal{O}_c(\log |G|)$ in time $\mathcal{O}_c(\log |G| + \max\{\Phi(\mathcal{T}) - \Phi(\mathcal{T}'), 0\})$. By accounting for the change in potential, we get this amortized running time.

AddVertex(v). First, we transform G into a graph G' by adding v to the vertex set. We transform $\mathcal{T}$ into a superbranch decomposition $\mathcal{T}'$ of $\mathcal{H}(G')$ rooted at r' by inserting e_v as a child of r using the rotation of Lemma 6.4 with $X = \emptyset$ in $\mathcal{O}(|V(e)|) = \mathcal{O}(1)$ time. This corresponds to a sequence $\mathcal{S}$ with $\|\mathcal{S}\|_{\mathcal{T}} \leq \mathcal{O}(1)$. The torso of r' can be obtained from $\text{torso}(r)$ by inserting a hyperedge e with $V(e) = \emptyset$ corresponding to the empty adhesion of e_v . This can be done through a sequence $\mathcal{C}$ of basic hypergraph operations of size $\mathcal{O}(1)$. Since v is an isolated vertex, the resulting decomposition $\mathcal{T}'$ remains downwards well-linked, with unchanged well-linked number. Then, since every node of $\mathcal{T}'$ other than the root has the same number of children as in $\mathcal{T}$, it follows that $\mathcal{T}'$ is c -good.

DeleteVertex(v). First, we transform G into a graph G' by deleting v from the vertex set. We apply Lemma 6.9 to e_v which transforms $\mathcal{T}$ into a c -good superbranch decomposition $\mathcal{T}_1$ of $\mathcal{H}(G)$ rooted at r_1 with $\mathcal{L}^{-1}(e_v) \in \text{chd}(r_1)$ in $\mathcal{O}_c(\log |G|)$ amortized time, through a sequence $\mathcal{S}_1$ of

basic rotations of size bounded by $\mathcal{O}_c(\log |G|)$. The lemma also gives a sequence of hypergraph operations $\mathcal{C}_1$ of size $\mathcal{O}_c(1)$ transforming $\text{torso}(r)$ into $\text{torso}(r_1)$. Next, we transform $\mathcal{T}_1$ into a superbranch decomposition $\mathcal{T}_2$ of $\mathcal{H}(G')$ rooted at r_2 by deleting e_v as a child of r_1 using the rotation of [Lemma 6.5](#) in $\mathcal{O}(\text{adhsz}(\mathcal{T}_1)^2) = \mathcal{O}_c(1)$ time, through a (single-rotation) sequence $\mathcal{S}_2$ of size $\|\mathcal{S}_2\| \leq \mathcal{O}_c(1)$. Concatenating the sequences $\mathcal{S}_1$ and $\mathcal{S}_2$, we get a final sequence $\mathcal{S}$ of size $\mathcal{O}_c(\log |G|)$. The torso of r_2 can be obtained from $\text{torso}(r_1)$ by removing the hyperedge corresponding to the empty adhesion of e_v , through a sequence $\mathcal{C}_2$ of basic hypergraph operations of size $\mathcal{O}(1)$. Concatenating $\mathcal{C}_1$ and $\mathcal{C}_2$ we get a final sequence of size $\mathcal{O}_c(1)$ transforming $\text{torso}(r)$ into $\text{torso}(r_2)$. Since v is an isolated vertex, the resulting decomposition $\mathcal{T}_2$ remains downwards well-linked, with unchanged well-linked number. Then, since every node of $\mathcal{T}_2$ other than the root has the same number of children as in $\mathcal{T}$, it follows that $\mathcal{T}'$ is c -good.

AddEdge(uv). First, we transform G into a graph G' by adding uv to the edge set. We apply [Lemma 6.9](#) to both hyperedges e_u and e_v , which transforms $\mathcal{T}$ into a c -good superbranch decomposition $\mathcal{T}_1$ of $\mathcal{H}(G)$ rooted at r_1 with $\mathcal{L}^{-1}\{e_u, e_v\} \subseteq \text{chd}(r)$ in $\mathcal{O}_c(\log |G|)$ amortized time. Let $\mathcal{S}_1$ and $\mathcal{C}_1$ be the returned sequences of basic rotations and hypergraph operations, respectively. The amortized size of $\|\mathcal{S}_1\|_{\mathcal{T}}$ is $\mathcal{O}_c(\log |G|)$ and $\|\mathcal{C}_1\| \leq \mathcal{O}_c(1)$.

Then, we transform $\mathcal{T}_1$ into a superbranch decomposition $\mathcal{T}_2$ of $\mathcal{H}(G')$ rooted at r_2 by inserting e_{uv} as a child of r_1 using the rotation of [Lemma 6.4](#) with $X = \mathcal{L}_1^{-1}\{e_u, e_v\}$ in $\mathcal{O}(\text{rk}(\mathcal{H}(G)) \cdot |V(e)|) = \mathcal{O}_c(1)$ time. This corresponds to a sequence $\mathcal{S}_2$ with $\|\mathcal{S}_2\|_{\mathcal{T}_1} \leq \mathcal{O}_c(1)$. By the lemma, we also have that $\text{torso}(r_2)$ can be obtained from $\text{torso}(r_1)$ via a sequence $\mathcal{C}_2$ of basic hypergraph operations of size $\mathcal{O}_c(1)$. We can thus let $\mathcal{S}$ be the concatenation of $\mathcal{S}_1$ and $\mathcal{S}_2$, and $\mathcal{C}$ be the concatenation of $\mathcal{C}_1$ and $\mathcal{C}_2$.

Note that $\mathcal{T}_2$ is trivially c -good at e_{uv} . Furthermore, $\mathcal{T}_2$ is c -good at every other $t \in \text{chd}(r_2) \setminus \{e_{uv}\}$, as the insertion basic rotation leaves the subtree rooted at t , as well as the boundary of every subset of $\mathcal{L}_2[t]$ unchanged, since e_u and e_v were already root children.

DeleteEdge(uv). First, we transform G into a graph G' by deleting uv from the edge set. We apply [Lemma 6.9](#) to both hyperedges e_u, e_v, e_{uv} , which transforms $\mathcal{T}$ into a c -good superbranch decomposition $\mathcal{T}_1$ rooted at r_1 with $\mathcal{L}^{-1}\{e_u, e_v, e_{uv}\} \subseteq \text{chd}(r)$ in $\mathcal{O}_c(\log |G|)$ amortized time. Let $\mathcal{S}_1$ and $\mathcal{C}_1$ be the returned sequences of basic rotations and hypergraph operations, respectively. The amortized size of $\|\mathcal{S}_1\|_{\mathcal{T}}$ is $\mathcal{O}_c(\log |G|)$ and $\|\mathcal{C}_1\| \leq \mathcal{O}_c(1)$.

Then, we transform $\mathcal{T}_1$ into a superbranch decomposition of $\mathcal{T}_2$ of $\mathcal{H}(G')$ rooted at r_2 by deleting e_{uv} as a child of r using the rotation of [Lemma 6.5](#) in $\mathcal{O}(\text{adhsz}(\mathcal{T}_1)^2) = \mathcal{O}_c(1)$ time. This corresponds to a sequence $\mathcal{S}_2$ with $\|\mathcal{S}_2\|_{\mathcal{T}_1} \leq \mathcal{O}_c(1)$. By the lemma, we have that $\text{torso}(r_2)$ can be obtained from $\text{torso}(r_1)$ via a sequence $\mathcal{C}_2$ of basic hypergraph operations of size $\mathcal{O}_c(1)$. We can thus let $\mathcal{S}$ be the concatenation of $\mathcal{S}_1$ and $\mathcal{S}_2$, and $\mathcal{C}$ be the concatenation of $\mathcal{C}_1$ and $\mathcal{C}_2$.

By the same argument as for **AddEdge(uv)**, we get that $\mathcal{T}_2$ is c -good.

Merge(A). By performing a split rotation using [Lemma 6.3](#) with B_A , we transform $\mathcal{T}$ into a superbranch decomposition $\mathcal{T}_1 = (\mathcal{T}_1, \mathcal{L}_1)$ rooted at r_1 , where the children of A become children of a new node t that is a child of r_1 . This corresponds to a sequence $\mathcal{S}_1$ with $\|\mathcal{S}_1\|_{\mathcal{T}} \leq \mathcal{O}_c(1)$. By the lemma, we have that $\text{torso}(r_1)$ can be obtained from $\text{torso}(r)$ via a sequence $\mathcal{C}_1$ of basic hypergraph operations of size $\mathcal{O}_c(1)$. Since $\mathcal{T}$ is downwards well-linked and $B_A \triangleright \mathcal{T}$ is well-linked in G , it follows that $\mathcal{T}_1$ is downwards well-linked. Because $|A| \leq 2^{2c} + 1$, we have $\Delta((\mathcal{T}_1)_t) \leq 2^{2c} + 1$. Together with $\text{wl}(B_A \triangleright \mathcal{T}) \leq c$, this shows that $\mathcal{T}_1$ is c -semigood. Note that a split rotation cannot increase the potential, i.e. $\Phi(\mathcal{T}_1) \leq \Phi(\mathcal{T})$.

Observe that the only node in $V(\mathcal{T}_1) \setminus \{r_1\}$ that can be 2^{2c+1} -unbalanced is t . By applying [Lemma 6.7](#) with t and prefix $\{t\}$, we transform $\mathcal{T}_1$ into a c -good superbranch decomposition $\mathcal{T}_2$ via a sequence $\mathcal{S}_2$ such that $r_1 \notin V_{\mathcal{T}_1}(\mathcal{S}_2)$ and $\Phi(\mathcal{T}_2) \leq \Phi(\mathcal{T}_1)$. Thus, the root torso is unchanged.

Furthermore, the amortized running time of this algorithm is $\mathcal{O}_c(1)$, which is also an upper bound for $\|\mathcal{S}_2\|_{\mathcal{T}_1}$. We let $\mathcal{S}$ be the concatenation of $\mathcal{S}_1$ and $\mathcal{S}_2$, and $\mathcal{C} = \mathcal{C}_1$. By the above, $\Phi(\mathcal{T}_2) \leq \Phi(\mathcal{T})$, and the total amortized running time is $\mathcal{O}_c(1)$, which is also an upper bound on $\mathcal{S}$.

Protrusion decomposition. Next, we show how to maintain a corresponding protrusion decomposition. We will construct $\tilde{\mathcal{T}}$ from $\mathcal{T}$ in time $\mathcal{O}_c(\|\mathcal{S}\|_{\mathcal{T}})$, giving the claimed amortized time bounds. As a first step, we let $\tilde{T} = T$ and $\mathbf{bag}(t) = V(\text{torso}(t))$ for all $t \in V(\tilde{T})$. Note that this already is a tree decomposition corresponding to $\mathcal{T}$ with bag size $|\mathbf{bag}(t)| \leq \mathcal{O}_c(1)$ for every non-root node $t \in V(\tilde{T}) \setminus \{r\}$, and $|\mathbf{bag}(r)| \leq \Delta(r)c$ for the root.

We now make the protrusions of $\tilde{\mathcal{T}}$ binary. For each non-root node $t \in V(\tilde{T}) \setminus \{r\}$ with $d > 2$ children $c_1, \dots, c_d$ and some parent p , we remove t and its incident edges from $\tilde{T}$, and add new nodes $t_1, \dots, t_{d-1}$, together with edges $t_i t_{i+1}$ for $i \in \{1, \dots, d-2\}$, edges $t_i c_i$ for $i \in \{1, \dots, d-1\}$, and edges pt_1 and $t_{d-1}c_d$. The new nodes are given bags $\mathbf{bag}(t_i) = \mathbf{bag}(t)$. We say that the $t_i \in V(\tilde{T})$ correspond to $t \in V(T)$. We observe that $(\tilde{T}, \mathbf{bag})$ can be maintained locally for all non-root nodes, in the sense that if $\mathcal{T}$ is transformed to $\mathcal{T}'$ by a sequence of basic rotations $\mathcal{S}$, then the only parts of $\tilde{\mathcal{T}}$ that need to be recomputed are the nodes corresponding to the nodes in $\text{trace}_{\mathcal{T}'}(\mathcal{S})$. This can be done in time $\mathcal{O}_c(|\text{trace}_{\mathcal{T}'}(\mathcal{S})|) \leq \mathcal{O}_c(\|\mathcal{S}\|_{\mathcal{T}})$. For the root r , we cannot recompute the entire $\mathbf{bag}(r)$ within the claimed time bounds. Instead, we can use the sequence $\mathcal{C}$ of basic hypergraph operations computed for each operation above. For every hypergraph operation $\text{DeleteVertex}(v)$ or $\text{AddVertex}(v)$ in $\mathcal{C}$, we update the balanced binary search tree representing $\mathbf{bag}(r)$, by deleting or inserting the corresponding vertex v . The running time of this update is $\mathcal{O}_c(\log |G|)$.

We know that $\tilde{\mathcal{T}}$ is normal by Lemma 3.3. For $\tilde{\mathcal{T}}$ to be an annotated normal $(\Delta(r)c, \mathcal{O}_c(1))$ -protrusion decomposition corresponding to $\mathcal{T}$, the only thing left is to maintain the $\mathbf{edges}$ function. Let us consider the process of making $\tilde{T}$ binary, i.e., when we split t into nodes $t_1, \dots, t_{d-1}$. Since $\mathbf{bag}(t_i) = \mathbf{bag}(t)$ for all $i \in \{1, d-1\}$, we have that $\mathbf{edges}(t_i) = \emptyset$ for all $i \in \{2, d-1\}$ and $\mathbf{edges}_{\tilde{T}}(t_1) = \mathbf{edges}_T(t)$. Furthermore, note that each time we split a single high-degree node into a chain, this does not change the value of $\mathbf{edges}$ on the untouched nodes. Thus, to compute the $\mathbf{edges}$ function, it suffices to compute it for nodes in T .

For this, we will maintain the function $\mathbf{EL}: V(T) \rightarrow 2^{E(G)}$ where, for each node $t \in V(T)$, $\mathbf{EL}(t)$ is the set of all edges $uv \in E(G)$ with $u, v \in V(\text{torso}(t))$ and $e_{uv} \in \mathcal{L}[t]$. The function is represented as follows: for each non-root node $t \in V(T) \setminus \{r\}$, the set $\mathbf{EL}(t)$ is stored as a linked list; and for the root r , the set $\mathbf{EL}(r)$ is stored as a balanced binary search tree $\mathcal{B}$ containing all edges $e \in \mathbf{EL}(r)$. Observe that for any internal node $t \in V_{\text{int}}(T)$ with children $c_1, \dots, c_d$, we have

$$\mathbf{EL}(t) = \bigcup_i \{e \in \mathbf{EL}(c_i) \mid V(e) \subseteq \text{adh}(tc_i)\}. \quad (6.10)$$

The only parts of $\tilde{\mathcal{T}}$ where $\mathbf{EL}$ must be recomputed are the nodes in $\text{trace}_{\mathcal{T}'}(\mathcal{S})$, since no torso outside the trace changes during our operations. We start by updating $\mathbf{EL}(t)$ for each non-root node $t \in \text{trace}_{\mathcal{T}'}(\mathcal{S}) \setminus \{r\}$ in a bottom-up manner. Since $|\mathbf{EL}(t)| \leq \binom{|\text{torso}(t)|}{2} \leq \mathcal{O}_c(1)$, and given $\mathbf{EL}(c)$ for all children c of t , we can recompute $\mathbf{EL}(t)$ in time $\mathcal{O}_c(1)$ using Equation (6.10). This gives a total running time of $\mathcal{O}_c(|\text{trace}_{\mathcal{T}'}(\mathcal{S})|) \leq \mathcal{O}_c(\|\mathcal{S}\|_{\mathcal{T}})$.

We now update $\mathbf{EL}(r)$ represented by the binary balanced search tree $\mathcal{B}$. For any $e \in E(\text{torso}(r))$ that also appears in the torso after applying the sequence $\mathcal{S}$ (i.e. where the sequence $\mathcal{C}$ does not contain $\text{DeleteEdge}(e)$), we have ensured above, that the corresponding subtree is unchanged. Since the adhesion is the same, it contributes the same edges to $\mathbf{EL}_{\mathcal{T}'}(r)$ in Equation (6.10). Hence, only the children inserted or deleted by the sequence $\mathcal{C}$ of basic hypergraph operations must be considered. As $\|\mathcal{C}\| \leq \mathcal{O}_c(1)$, there are at most $\mathcal{O}_c(1)$ insertions or deletions, and we can maintain the balanced binary search tree $\mathcal{B}$ in time $\mathcal{O}_c(\log |G|)$.

Now, we compute $\text{edges}(t)$ for each $t \neq r$ as the set of edges in $\text{EL}(t)$ which are not also in $\text{EL}(p)$, where p is the parent of t . If p is not the root, this takes $\mathcal{O}_c(1)$ time, and otherwise, we have to query $\mathcal{B}$ resulting in a running time of $\mathcal{O}_c(\log |G|)$. Since at most $\mathcal{O}_c(1)$ nodes in $\text{trace}_{\mathcal{T}}(\mathcal{S})$ have the root as a parent, the total running time for this step is $\mathcal{O}_c(\log |G| + \|\mathcal{S}\|_{\mathcal{T}})$, which remains within the claimed bounds. Note that $\text{edges}(r) = \text{EL}(r)$, so the balanced binary search tree $\mathcal{B}$ serves as the representation of $\text{edges}(r)$. Each time we insert or delete an edge in $\mathcal{B}$, we can append it to a list, which is returned. This list has size $\mathcal{O}_c(1)$.

Run of an automaton. Finally, we prove that we can maintain a run of an automaton. Recall from earlier in this proof, that when $\mathcal{T}$ is modified by a sequence of basic rotations $\mathcal{S}$, we rebuild a prefix of $\tilde{\mathcal{T}}$ of size at most $\mathcal{O}_c(\|\mathcal{S}\|_{\mathcal{T}})$. This can be done in a bottom-up fashion, so that when we build a non-root node t , we can also compute its automaton state $\rho_{\mathcal{A}}^{\tilde{\mathcal{T}}}(t)$ incurring an additional factor τ on the running times.

If the automaton state at the node $t \in \text{chd}(r)$ corresponding to an edge $e \in E(\text{torso}(r))$ is modified by an operation, this happens because the subtree of the superbranch decomposition rooted at t was modified. If so, the sequence of hypergraph operations $\mathcal{C}$ will contain the hypergraph operation $\text{DeleteHyperedge}(e)$, as guaranteed above. $\square$

7 Assembling the main data structure

In this section, we finally assemble our main data structure that maintains a protrusion decomposition of a dynamic topological-minor-free graph G , thus proving [Theorem 1.2](#). The data structure from [Lemma 6.1](#) already performs all the required operations for the insertion and deletion of edges and vertices and maintains a superbranch decomposition and a corresponding protrusion decomposition, where the subtrees below the root satisfy all our requirements. However, this data structure allows the root degree, and thus the first parameter of the corresponding protrusion decomposition, to be arbitrarily large, and instead provides a **Merge** operation, which allows the user to decrease the root degree themselves – if they know a suitable set of “mergeable” root-children.

In [Section 4](#), for topological-minor-free graphs with a small treewidth- η -modulator, we proved the existence of such a set of “mergeable” root-children, identified by their corresponding hyperedges in the torso of the root, if the degree of the root r of the superbranch decomposition is high enough. That is, we showed that there exists a set $B \subseteq E(\text{torso}(r))$ of bounded size such that $B \triangleright \mathcal{T}$ has small boundary and internal treewidth and all internal components of B have the same boundary as B . Then, in [Section 5](#), we constructed a data structure that, when applied to $\text{torso}(r)$ and provided with an oracle that decides whether the internal treewidth of such a set $B \triangleright \mathcal{T}$ is “small”, returns such a set B (if it exists).

Essentially, there are two major things left to be done in this section. For one, we need to construct such an internal treewidth oracle. Secondly, the **Merge** operation from [Lemma 6.1](#) is restricted to well-linked sets of root-children to keep the superbranch decomposition downwards well-linked. Thus, we need to find a well-linked subset of this “mergeable” set B .

We start with the internal treewidth oracle. To obtain this, we use a tree decomposition automaton for computing internal treewidth, stated in [Lemma 7.1](#). The proof of this lemma is given in [Section B](#). It is based on the observation that the Bodlaender-Kloks dynamic programming for treewidth [\[BK96\]](#) can be seen as an tree decomposition automaton (used also e.g. by [\[KMN⁺23\]](#)), hence the name $\mathcal{IBK}$.

Lemma 7.1. *For every pair of integers $k \leq \ell$, there is a tree decomposition automaton $\mathcal{IBK}_{k,\ell}$ of width ℓ with the following property: For any boundaried graph G and its annotated boundaried tree decomposition $\mathcal{T} = (T, \text{bag}, \text{edges})$ of width at most ℓ , $\mathcal{IBK}_{k,\ell}$ accepts $\mathcal{T}$ if and only if the*

internal treewidth $\text{itw}(G)$ of G is at most k . The state space of $\mathcal{IBK}_{k,\ell}$ is of size $\mathcal{O}_{k,\ell}(1)$ and can be computed in time $\mathcal{O}_{k,\ell}(1)$. The evaluation time of $\mathcal{IBK}_{k,\ell}$ is $\mathcal{O}_{k,\ell}(1)$ as well.

Using [Lemma 7.1](#) we can construct the internal treewidth oracle.

Lemma 7.2. *Let G be a graph, ω an integer, and $\mathcal{T}' = (T', \mathcal{L}')$ a downwards well-linked super-branch decomposition of $\mathcal{H}(G)$ with adhesion size α and root r . Let further $\mathcal{T} = (T, \text{bag}, \text{edges})$ be an annotated normal (k, γ) -protrusion decomposition with root r that corresponds to $\mathcal{T}'$, and $\rho_{\mathcal{A}}^{\mathcal{T}}$ the run of the automaton $\mathcal{A} := \mathcal{IBK}_{\omega, (\gamma+1) \cdot \sigma}$ from [Lemma 7.1](#) on $\mathcal{T}$.*

Suppose that the representations of $\mathcal{T}'$ and $\mathcal{T}$ are stored, and for each $t \in \text{chd}(r)$ we can query the state $\rho_{\mathcal{A}}^{\mathcal{T}}(t)$ in $\mathcal{O}_{\omega, \gamma, \sigma}(1)$ time. Then, given a set $S \subseteq E(\text{torso}(r))$ of size $|S| \leq \sigma$, we can in time $\mathcal{O}_{\alpha, \gamma, \omega, \sigma}(\log \Delta(r))$ decide whether $\text{itw}(S \triangleright \mathcal{T}') \leq \omega$.

Proof. Let $S = \{e_1, \dots, e_q\} \subseteq E(\text{torso}(r))$ be a set of hyperedges and for $1 \leq i \leq q$, let $t_i \in \text{chd}(r)$ be the root-child that corresponds to e_i , i.e., $\{e_i\} \triangleright \mathcal{T}' = \mathcal{L}'[t_i]$, and thus $V(e_i) = \text{bd}(\mathcal{L}'[t_i])$. Since $\mathcal{T}$ corresponds to $\mathcal{T}'$, for each $t \in \text{chd}(r)$, we have $V(\mathcal{L}'[t]) = \bigcup_{t' \in \text{desc}(t)} \text{bag}(t')$. Let G_S be the boundaried graph obtained from $G[\bigcup_{t_i} V(\mathcal{L}'[t_i])]$ by setting $\text{bd}(S)$ as the boundary. We observe that $\text{itw}(G_S) = \text{itw}(S \triangleright \mathcal{T}')$, so our goal is to determine whether $\text{itw}(G_S) \leq \omega$. We do this by computing the root-state of $\mathcal{A}$ on a boundaried tree decomposition of G_S .

Without time constraints, this could be done simply by combining the subtrees $\mathcal{T}_i$ of $\mathcal{T}$ rooted at the nodes t_i to form an annotated boundaried tree decomposition of G_S and running $\mathcal{A}$ on it. However, building this entire tree decomposition would exceed the runtime constraints. But as we already know the states of $\mathcal{A}$ on the roots of the subtrees $\mathcal{T}_i$, it is sufficient to construct a prefix $\mathcal{T}_S$ of this tree decomposition that contains the root t_i from each $\mathcal{T}_i$ but nothing below. To get the same result when running the automaton $\mathcal{A}$ only on this prefix, we initialize the nodes t_i with their state $\rho_{\mathcal{A}}^{\mathcal{T}}(t_i)$, which we can query in $\mathcal{O}_{\omega, \gamma, \sigma}(1)$ time for each t_i .

In detail, to construct $\mathcal{T}_S = (T_S, \text{bag}_S, \text{edges}_S)$, we first take the nodes t_i for $1 \leq i \leq q$ and add them to $V(T_S)$. Then, we insert q additional nodes $s_1, \dots, s_{q-1}$ and r_S , where r_S will be the root of $\mathcal{T}_S$. For $i = 2, \dots, q-1$, we add the edges $s_i s_{i-1}$ and $s_i t_{i+1}$ to T_S . Additionally, we add the edges $s_1 t_1$, $s_1 t_2$, and $r_S s_{q-1}$ to $\mathcal{T}_S$. We define the bag_S function as follows in a bottom-up way:

- First, we set $\text{bag}_S(t_i) := \text{bag}(t_i)$ for every $1 \leq i \leq q$.
- Then, we set $\text{bag}_S(s_1) := V(e_1) \cup V(e_2)$.
- Next, we recursively set $\text{bag}_S(s_i) := \text{bag}_S(s_{i-1}) \cup V(e_{i+1})$ for $2 \leq i \leq q-1$.
- Lastly, we set $\text{bag}_S(r_S) := \text{bd}(S \triangleright \mathcal{T}') = \text{bd}(S) \subseteq \text{bag}_S(s_{q-1})$.

For the last step, by [Lemma 3.4](#), $\text{bd}(S)$ can be computed in time $\mathcal{O}(|S| \cdot \alpha) = \mathcal{O}_{\alpha, \sigma}(1)$. We observe that the width of $\mathcal{T}_S$ is at most $(\gamma + 1) \cdot |S| - 1 \leq \gamma \cdot \sigma$, so the automaton $\mathcal{A}$ of width $(\gamma + 1) \cdot \sigma$ can indeed be applied to $\mathcal{T}_S$.

Recall that the edges_S function is uniquely determined by G_S and the pair (T_S, bag_S) . Since $\text{bag}_S(t_i) = \text{bag}(t_i)$, and the adhesion of t_i to its parent p of T_S satisfy $\text{adh}_S(t_i p) = \text{adh}(t_i r) = V(e_i)$, it follows that $\text{edges}_S(t_i) = \text{edges}(t_i)$. All other bags of $\mathcal{T}_S$ except $\text{bag}_S(s_{q-1})$ and $\text{bag}_S(r_S)$ are subsets of their parents, so they have empty edges_S . For $\text{bag}_S(s_{q-1})$, note that all its vertices are in $\text{bag}(r)$, implying that $\text{edges}_S(s_{q-1}) \subseteq \text{edges}(r)$. Hence, we can compute $\text{edges}_S(s_{q-1})$ by finding all edges uv with $u, v \in \text{bag}_S(s_{q-1}) \setminus \text{bag}_S(r_S)$ and $uv \in \text{edges}(r)$. For this, we query the balanced binary search tree representing $\text{edges}(r)$ for each pair $u, v \in \text{bag}_S(s_{q-1}) \setminus \text{bag}_S(r_S)$ in $(|S| + \alpha)^{\mathcal{O}(1)} \cdot \mathcal{O}(\log(\Delta(r) \cdot \alpha)) = \mathcal{O}_{\alpha, \sigma}(\log \Delta(r))$ time. In the same manner, we can compute $\text{edges}_S(r_S)$ within the same time bound.

We observe that $\mathcal{T}_S$ can indeed be extended to an annotated boundaried tree decomposition $\mathcal{T}_S^*$ of G_S by attaching the subtrees $\mathcal{T}_i$ to t_i for each $1 \leq i \leq q$. Note that for each t_i , we

have $\rho_{\mathcal{A}}^{\mathcal{T}}(t_i) = \rho_{\mathcal{A}}^{\mathcal{T}_S^*}(t_i)$. Therefore, to determine the state of $\mathcal{A}$ on the node r_S of $\mathcal{T}_S^*$, it suffices to initialize the states of t_i to be $\rho_{\mathcal{A}}^{\mathcal{T}}(t_i)$, and compute the states of the rest of the nodes of $\mathcal{T}_S$ in a bottom-up manner with the transitions of $\mathcal{A}$. As $\mathcal{T}_S$ has $\mathcal{O}(\sigma)$ nodes and the evaluation time of $\mathcal{A}$ is $\mathcal{O}_{\omega, \gamma, \sigma}(1)$, this runs in $\mathcal{O}_{\omega, \gamma, \sigma}(1)$ time. Whether $\text{itw}(G_S) \leq \gamma$ can then be determined by whether the state of r_S is accepting. $\square$

From now on, we can assume that we have an oracle that decides for every small set $B \subseteq E(\text{torso}(r))$ whether the internal treewidth of $B \triangleright \mathcal{T}$ is small. Thus, we can give this oracle to the data structure from [Lemma 5.2](#), which will then help us find a “mergeable” set B of hyperedges in $\text{torso}(r)$ whenever needed.

Secondly, we need to show that we can always find a well-linked subset of this set B . To do this, [Lemma 7.4](#) provides us with an algorithm that partitions a set B of hyperedges of a hypergraph G into well-linked subsets. The algorithm is from [\[Kor25a, Lemma 5.3\]](#), although they have a runtime dependency on the size $|G|$ of the hypergraph G instead of only the size $|B|$ of B in their paper. The reason for this is that they use a version of the following algorithm, also with a runtime dependency on $|G|$ instead of $|B|$, as a subprocedure (see [\[Kor25a, Lemma 3.1\]](#)). However, as shown in [\[Kor25b, Lemma 7.1\]](#), this can be made to run in $2^{\mathcal{O}(\lambda(B))} \cdot \text{rk}(G)^2 \cdot |B|$ time.

Lemma 7.3 ([\[Kor25b\]](#)). *Let G be a hypergraph of rank r whose representation is already stored. There is an algorithm that, given a set $B \subseteq E(G)$, in time $2^{\mathcal{O}(\lambda(B))} \cdot r^2 \cdot |B|$ either*

- *correctly concludes that B is well-linked or*
- *returns a bipartition (B_1, B_2) of B so that $\lambda(B_i) < \lambda(B)$ for both $i \in [2]$.*

Now, if we plug this improved version into the algorithm from [\[Kor25a, Lemma 5.3\]](#), we obtain an algorithm that in time $2^{\mathcal{O}(\lambda(B))} \cdot \text{rk}(G)^2 \cdot |B|$ partitions a set B of hyperedges into well-linked sets.

Lemma 7.4 (Based on [\[Kor25a, Lemma 5.3\]](#)). *Let G be a hypergraph of rank r whose representation is already stored. There is an algorithm that, given a set $B \subseteq E(G)$, in time $2^{\mathcal{O}(\lambda(B))} \cdot r^2 \cdot |B|$ returns a partition $\mathfrak{B}$ of B into at most $|\mathfrak{B}| \leq 2^{\lambda(B)}$ sets, so that each $X \in \mathfrak{B}$ is well-linked in G .*

Proof. We maintain a partition $\mathfrak{B}$ of B , initially setting $\mathfrak{B} = \{B\}$. For each part $X \in \mathfrak{B}$, we repeatedly apply the algorithm from [Lemma 7.3](#) to test whether X is well-linked, and if not, replace X by the two sets X_1, X_2 returned by the algorithm, where (X_1, X_2) is a bipartition of X with $\lambda(X_i) < \lambda(X)$ for both $i \in [2]$.

Throughout this process, we always have $\sum_{X \in \mathfrak{B}} 2^{\lambda(X)} \leq 2^{\lambda(B)}$, while the size of $\mathfrak{B}$ increases strictly in each step. Therefore, it must terminate within $2^{\lambda(B)}$ iterations, with $|\mathfrak{B}| \leq 2^{\lambda(B)}$. As the algorithm from [Lemma 7.3](#) runs in time $2^{\mathcal{O}(\lambda(X))} \cdot r^2 \cdot |X|$, and $\lambda(X) \leq \lambda(B)$ for each part $X \in \mathfrak{B}$, the total running time is at most $2^{\mathcal{O}(\lambda(B))} \cdot r^2 \cdot |B|$. $\square$

Finally, we are ready to prove [Theorem 1.2](#). For this, we only need to combine our data structures from [Section 5](#) and [Section 6](#) together with the results from this section and [Section 4](#) and adjust them with the right parameters. We prove the following stronger and more technical version of [Theorem 1.2](#).

Theorem 7.5. *There is a data structure, that is initialized with a graph H , an integer η , and an empty graph G , and supports updating G under the assumption that it remains H -topological-minor-free. The data structure maintains G , $\mathcal{H}(G)$, and an annotated normal $(\mathcal{O}_{H, \eta}(\text{tw-mod}_{\eta}(G)), \mathcal{O}_{H, \eta}(1))$ -protrusion decomposition $\mathcal{T} = (T, \text{bag}, \text{edges})$ of G rooted at node r together with the hypergraph $\text{torso}(r)$. The data structure supports the following operations.*

- **Init**(H, η): We initialize the data structure with H and η , and an empty graph G . Afterwards, $\text{torso}(r)$ is an empty hypergraph. Runs in $\mathcal{O}_{H,\eta}(1)$ time.
- **AddVertex**(v): Given a new vertex $v \notin V(G)$, add v to G .
- **DeleteVertex**(v): Given an isolated vertex $v \in V(G)$, remove v from G .
- **AddEdge**(e): Given a new edge $e \in \binom{V(G)}{2} \setminus E(G)$, add e to G .
- **DeleteEdge**(e): Given an edge $e \in E(G)$, remove e from G .

Each operation except for **Init** runs in $\mathcal{O}_{H,\eta}(\log |G|)$ amortized time. For each operation, the changes to $\text{torso}(r)$ can be described as a sequence $\mathcal{C}$ of hypergraph operation of size $\mathcal{O}_{H,\eta}(1)$, which is returned. Changes to $\text{edges}(r)$ can be described as a list of insertions and deletions of size at most $\mathcal{O}_{H,\eta}(1)$, which is returned as well.

Moreover, if upon initialization the data structure is provided a tree decomposition automaton $\mathcal{A}$ with evaluation time τ , then a run of $\mathcal{A}$ on each protrusion is maintained, incurring an additional factor τ on the running times. If the automaton state at the node corresponding to an edge $e \in E(\text{torso}(r))$ is modified by an operation, the corresponding sequence $\mathcal{C}$ will contain the operation **DeleteHyperedge**(e).

Proof. We begin by outlining the structure of the proof. We use the data structure from [Lemma 6.1](#) with $c = \mathcal{O}_{\eta,H}(1)$ to maintain the graph G , the support hypergraph $\mathcal{H}(G)$, a downwards well-linked superbranch decomposition $\mathcal{T}'$ of $\mathcal{H}(G)$ rooted at a node r' , and an annotated normal $(\Delta(r') \cdot \mathcal{O}_{\eta,H}(1), \mathcal{O}_{\eta,H}(1))$ -protrusion decomposition $\mathcal{T}$ corresponding to $\mathcal{T}'$ under the required operations. We now also want to keep the degree of the root r' of the superbranch decomposition $\mathcal{T}'$ bounded, that is, $\Delta(r') \leq \mathcal{O}_{H,\eta}(\text{tw-mod}_\eta(G))$, implying that $\mathcal{T}$ is a $(\mathcal{O}_{H,\eta}(\text{tw-mod}_\eta(G)), \mathcal{O}_{H,\eta}(1))$ -protrusion decomposition as required. To that end, we let the data structure from [Lemma 5.2](#) maintain a representation of the hypergraph $\text{torso}(r')$ – the same $\text{torso}(r')$ that is also maintained as part of $\mathcal{T}'$ by the data structure from [Lemma 6.1](#). This will help us find a set of “mergeable” root-children (see below for a formal definition of “mergeable”), on which we can apply the **Merge** operation from [Lemma 6.1](#) to decrease the root degree whenever possible.

Let $\omega = \omega(H, \eta)$ be the integer from [Lemma 4.1](#), and $b \leq \mathcal{O}_{H,\eta}(1)$ the factor hidden by the $\mathcal{O}_{H,\eta}$ -notation in the upper bound $|B| \leq \mathcal{O}_{H,\eta}(\alpha)$ of [Lemma 4.1](#). We use the data structure of [Lemma 6.1](#) with $c = \alpha := \max(6\omega + 3, b)$. It maintains a downwards well-linked superbranch decomposition $\mathcal{T}' = (T', \mathcal{L}')$ rooted at r' with adhesion size α , as well as an annotated normal $(\Delta(r') \cdot \alpha, \mathcal{O}_\alpha(1))$ -protrusion decomposition $\mathcal{T} = (T, \text{bag}, \text{edges})$ that corresponds to $\mathcal{T}'$. Note that the adhesion size bound implies that the rank of the hypergraph $\text{torso}(r')$ is at most α .

Let $\ell = \mathcal{O}_\alpha(1)$ be the upper bound for the width of the protrusions of $\mathcal{T}$ maintained by [Lemma 6.1](#). At the initialization of [Lemma 6.1](#), we provide it also the automaton $\mathcal{IBK}_{\omega,\ell,\alpha^2}$ from [Lemma 7.1](#) and the automaton $\mathcal{A}$ that was given to this lemma. Note that [Lemma 6.1](#) can be generalized from maintaining just one automaton to maintaining multiple by considering the automaton formed as the cartesian product of them. Maintaining these automata causes an extra $\mathcal{O}_{H,\eta}(1) + \tau$ factor in the running time.

Let $\delta = \delta(H, \eta, \alpha)$ be the integer from [Lemma 4.1](#) such that the following holds: If $\Delta(r') \geq \delta \cdot \text{tw-mod}_\eta(G)$, then there exists a set $B \subseteq E(\text{torso}(r'))$ with

- $\lambda(B) \leq \omega$,
- $\text{itw}(B \triangleright \mathcal{T}) \leq \omega$,
- $2^{\omega+2} \leq |B| \leq b \cdot \alpha \leq \alpha^2$, and

- for every internal component B' of B , we have $\text{bd}(B') = \text{bd}(B)$.

We call a set $B \subseteq \text{torso}(r')$ of hyperedges *semi-mergeable* if $\lambda(B) \leq \omega$, $\text{itw}(B \triangleright \mathcal{T}') \leq \omega$, and for every internal component B' of B it holds that $\text{bd}(B') = \text{bd}(B)$. These properties are closely related to what we actually want to be able to apply the **Merge** operation, as we will show in [Claim 7.6](#). Specifically, we call a set $B \subseteq E(\text{torso}(r'))$ *mergeable* if it is well-linked and $\text{wl}(B \triangleright \mathcal{T}) \leq \alpha$. Furthermore, a set $C \subseteq \text{chd}(r')$ of root-children corresponding to such a set B is also called *(semi-)mergeable*. We now show that it is indeed a small step from being semi-mergeable to being mergeable.

Claim 7.6. *There exists an algorithm that, given a semi-mergeable set B of size $|B| \geq 2^{\omega+1}$, finds a mergeable subset $B' \subseteq B$ of size $|B'| \geq 2$ in time $2^{\mathcal{O}(\lambda(B))} \cdot \alpha^2 \cdot |B|$.*

Proof of the claim. By [Lemma 7.4](#), there exists an algorithm that in time $2^{\mathcal{O}(\lambda(B))} \cdot \alpha^2 \cdot |B|$ finds a partition $\mathfrak{B}$ of B into at most $|\mathfrak{B}| \leq 2^{\lambda(B)} \leq 2^\omega$ sets so that each $X \in \mathfrak{B}$ is well-linked in $\text{torso}(r')$. By the pigeonhole principle, there is a well-linked set $B' \in \mathfrak{B}$ of size $|B'| \geq |B|/2^\omega \geq 2$. By [Lemma 3.2](#), $B' \triangleright \mathcal{T}'$ is well-linked in $\mathcal{H}(G)$ as well. Moreover, by [Lemma 3.1](#) we have $\text{wl}(B' \triangleright \mathcal{T}') \leq 3 \cdot (\text{tw}(B' \triangleright \mathcal{T}') + 1) \leq 3(\text{tw}(B \triangleright \mathcal{T}') + 1) \leq 3 \cdot (\text{itw}(B \triangleright \mathcal{T}') + \lambda(B \triangleright \mathcal{T}') + 1) \leq 6\omega + 3 \leq \alpha$, where $\text{tw}(B' \triangleright \mathcal{T}') \leq \text{tw}(B \triangleright \mathcal{T}')$ holds since $B' \triangleright \mathcal{T}' \subseteq B \triangleright \mathcal{T}$. $\triangleleft$

Our goal is to keep $\Delta(r') < \delta \cdot \text{tw-mod}_\eta(G)$ at all times (without knowing the exact value of $\text{tw-mod}_\eta(G)$). To that end, whenever possible, we want to find a semi-mergeable set of hyperedges in $\text{torso}(r')$, turn it into a mergeable set using [Claim 7.6](#), and then apply the **Merge** operation from [Lemma 6.1](#) on the corresponding set of root-children. To find such a set of semi-mergeable hyperedges in $\text{torso}(r')$, we use the data structure from [Lemma 5.2](#) with the parameters $s_1 := 2^{\omega+2}$, $s_2 := \alpha^2$, and $k := \omega$ and let it maintain a representation of $\text{torso}(r')$.

To this end, we provide the data structure with a α^2 -bounded oracle $\mathcal{O}$ so that for every set $S \subseteq E(\text{torso}(r'))$ of size $|S| \leq \alpha^2$ the oracle is satisfied if and only if $\text{itw}(S \triangleright \mathcal{T}') \leq \omega$. This oracle is realized by the algorithm from [Lemma 7.2](#) together with the maintained automaton $\mathcal{IBK}_{\omega, \ell \cdot \alpha^2}$. Given a set $S \subseteq E(\text{torso}(r'))$ of size $|S| \leq \alpha^2$, the algorithm from [Lemma 7.2](#) decides whether $\text{itw}(S \triangleright \mathcal{T}') \leq \omega$ in time $\mathcal{O}_{\alpha, \omega}(\log \Delta(r')) \leq \mathcal{O}_{H, \eta}(\log |G|)$. Now, the **Query** operation of [Lemma 5.2](#) returns a semi-mergeable set $B \subseteq E(\text{torso}(r'))$ of size $2^{\omega+1} \leq |B| \leq \alpha^2$ or concludes that no semi-mergeable set of size $2^{\omega+2} \leq |B| \leq \alpha^2$ exists. Furthermore, the operations of [Lemma 5.2](#) run in time $\mathcal{O}_\alpha(\log |E(\text{torso}(r'))|) \leq \mathcal{O}_{H, \eta}(\log |G|)$.

We are now ready to describe the operations of our data structure. For the initialization, we compute the integers ω and α given H and η . Then, we initialize the data structure from [Lemma 6.1](#) with the integer α in $\mathcal{O}(1)$ time. Note that $\text{torso}(r')$ is an empty hypergraph at the beginning since G is an empty graph. Thus, we can also initialize the data structure from [Lemma 5.2](#), which will keep a representation of $\text{torso}(r')$, in $\mathcal{O}(1)$ time.

For an update operation (insert/delete, vertex/edge), we apply the respective operation from [Lemma 6.1](#) to update our representations of G , $\mathcal{H}(G)$, $\mathcal{T}$, and $\mathcal{T}'$ in $\mathcal{O}_\alpha(\log |G|)$ amortized time. This returns a sequence of operations of size at most $\mathcal{O}_\alpha(1)$, which describes how the hypergraph $\text{torso}(r')$ before the operation can be turned into the hypergraph $\text{torso}(r')$ after the operation. For every basic hypergraph operation in this sequence, we apply the respective operation from [Lemma 5.2](#) to update the representation of $\text{torso}(r')$, each in at most $\mathcal{O}_{H, \eta}(\log |G|)$ time, so in total this takes $\mathcal{O}_{H, \eta}(\log |G|)$ amortized time. At the end, we return forward this sequence of operations, and also the changes to the set $\text{edges}(r)$.

To restore our invariant $\Delta(r') < \delta \cdot \text{tw-mod}_\eta(G)$ we do the following. After every update to G , we repeatedly apply the **Query** operation from [Lemma 5.2](#) to find a semi-mergeable set $B \subseteq E(\text{torso}(r'))$ of size $2^{\omega+1} \leq |B| \leq \alpha^2$. Then, we use [Claim 7.6](#) to translate B into a mergeable set B' of size $2 \leq |B'| \leq \alpha^2$, and finally apply the **Merge** operation on the corresponding set of root-children to decrease $\Delta(r')$. Let ζ denote the maximum difference between the root degree

$\Delta(r')$ before and after the current operation and observe that $\zeta \leq \mathcal{O}_\alpha(1)$. We repeat these steps at most $\zeta + 2\delta$ times, or until the **Query** operation returns that no semi-mergeable set B of size $2^{\omega+2} \leq |B| \leq \alpha^2$ exists.

By Lemma 4.1, if no such B exists, we know that $\Delta(r') < \delta \cdot \text{tw-mod}_\eta(G)$. Furthermore, we need to argue that $\zeta + 2\delta$ applications of the **Merge** operation are enough to keep $\Delta(r') < \delta \cdot \text{tw-mod}_\eta(G)$, even if the data structure from Lemma 5.2 could find more semi-mergeable sets. For this, we first argue that the value of $\text{tw-mod}_\eta(G)$ changes by at most two during each update.

Claim 7.7. *Let G be a graph, $v \in V(G)$ a vertex, and $e \in E(G)$ an edge. For every integer η , we have*

- $\text{tw-mod}_\eta(G \setminus \{v\}) \leq \text{tw-mod}_\eta(G) \leq \text{tw-mod}_\eta(G \setminus \{v\}) + 1$ and
- $\text{tw-mod}_\eta(G \setminus \{e\}) \leq \text{tw-mod}_\eta(G) \leq \text{tw-mod}_\eta(G \setminus \{e\}) + 2$.

Proof of the claim. Recall that for a graph G , $\text{tw-mod}_\eta(G)$ denotes the size of a smallest treewidth- η -modulator of G , i.e., the size of a smallest set $X \subseteq V(G)$ with $\text{tw}(G \setminus X) \leq \eta$. It is clear from the definition that deleting a vertex or an edge cannot increase the size of the smallest treewidth modulator, so $\text{tw-mod}_\eta(G \setminus \{v\}) \leq \text{tw-mod}_\eta(G)$ and $\text{tw-mod}_\eta(G \setminus \{e\}) \leq \text{tw-mod}_\eta(G)$. Furthermore, if X is a treewidth- η -modulator of $G \setminus \{v\}$ for a vertex $v \in V(G)$ (or of $G \setminus \{e\}$ for an edge $e = uv \in E(G)$), then $X \cup \{v\}$ (or $X \cup \{u, v\}$) is a treewidth- η -modulator of G , and thus $\text{tw-mod}_\eta(G) \leq \text{tw-mod}_\eta(G \setminus \{v\}) + 1$ (and $\text{tw-mod}_\eta(G) \leq \text{tw-mod}_\eta(G \setminus \{e\}) + 2$). $\triangleleft$

It follows that the target value $\delta \cdot \text{tw-mod}_\eta(G)$ for the degree $\Delta(r')$ changes by at most 2δ during each update. Thus, after one update and before applying the **Merge** operation, we have $\Delta(r') < \delta \cdot (\text{tw-mod}_\eta(G) + 2) + \zeta$, so we need to decrease $\Delta(r')$ by at most $\zeta + 2\delta$. As we only apply the **Merge** operation to sets B' with $|B'| \geq 2$, each application decreases $\Delta(r')$ by at least one. Thus, $\zeta + 2\delta \leq \mathcal{O}_{H,\eta}(1)$ applications are enough to restore the invariant $\Delta(r') < \delta \cdot \text{tw-mod}_\eta(G)$.

As argued before, the updates themselves can be performed in $\mathcal{O}_\alpha(\log |G|)$ amortized time, so the total amortized running time of each update operation is $\mathcal{O}_{H,\eta}(\log |G|)$. Then, the **Query** operation from Lemma 5.2 runs in $\mathcal{O}_{H,\eta}(\log |G|)$ time. Lastly, each of the $\mathcal{O}_{H,\eta}(1)$ applications of **Merge**($C_{B'}$), where $C_{B'}$ is the set of root-children corresponding to the mergeable set B' , runs in $\mathcal{O}_\alpha(1)$ amortized time. Thus, the total amortized running time of each update operation is $\mathcal{O}_{H,\eta}(\log |G|)$. $\square$

8 Dynamic kernelization

In this section, we finally show how our data structure can be used to maintain kernels for various problems on sparse graphs. Specifically, in Section 8.1, we give a tree decomposition automaton for the protrusion replacement, based on the techniques introduced by [BFL⁺16]. Then, in Section 8, we combine this automaton with the data structure from Theorem 1.2 to obtain a dynamic kernelization algorithm for all treewidth-bounding problems with FII on CMSO₂-definable topological-minor-free graph classes, thus proving Theorem 1.3.

We note to the reader that in this section we will have constants that depend on a non-computable way from other constants, typically from the problem Π , the graph class $\mathcal{G}$, and the treewidth-modulator constant η . In this case, we do not use the $\mathcal{O}$ -notation with subscript, as it implies a computable dependence, but instead treat Π , $\mathcal{G}$, and η as true constants in the sense that numbers depending on them can be hidden by $\mathcal{O}(1)$.

8.1 The protrusion replacement automaton

Before constructing the protrusion replacement automaton, we first need to define the graphs by which the protrusions will be replaced. Specifically, for every boundaried graph, we define a

smaller “equivalent” graph. For this, recall the definition of $\equiv_\Pi$, that is, for two boundaried graphs $G_1, G_2 \in \mathcal{F}$, we have $G_1 \equiv_\Pi G_2$ if and only if they have the same label set, i.e., $\mathbf{L}(G_1) = \mathbf{L}(G_2)$, and there exists a transposition constant $\Delta = \Delta_\Pi(G_1, G_2) \in \mathbb{Z}$ such that

$$\forall (F, k) \in \mathcal{F} \times \mathbb{Z} : (F \oplus_u G_1, k) \in \Pi \Leftrightarrow (F \oplus_u G_2, k + \Delta) \in \Pi. \quad (8.1)$$

It is not clear if such a value $\Delta_\Pi(G_1, G_2)$ is uniquely defined, and it in fact is not unique in all cases. However, we will fix a unique definition for $\Delta_\Pi(G_1, G_2)$ as follows.

We say a boundaried graph G is *monotone* if either $(F \oplus_u G, k) \in \Pi$ for every $(F, k) \in \mathcal{F} \times \mathbb{Z}$ (*positive monotone*) or $(F \oplus_u G, k) \notin \Pi$ for every $(F, k) \in \mathcal{F} \times \mathbb{Z}$ (*negative monotone*). We observe that the set of all positive monotone graphs and the set of all negative monotone graphs with the same label set form their own equivalence classes under $\equiv_\Pi$, respectively. For two boundaried graphs G_1, G_2 with the same label set that are both either positive or negative monotone, we set $\Delta_\Pi(G_1, G_2) := 0$. For all non-monotone graphs G_1, G_2 , we now show that the transposition constant $\Delta_\Pi(G_1, G_2)$ is uniquely determined by Equation (8.1).

Lemma 8.2. *Let Π be an parameterized graph problem and let $G_1, G_2 \in \mathcal{F}$ be two non-monotone boundaried graphs with $G_1 \equiv_\Pi G_2$. Then, there is exactly one constant $\Delta = \Delta_\Pi(G_1, G_2)$ such that for every $F \in \mathcal{F}$ and every $k \in \mathbb{Z}$ it holds that $(F \oplus_u G_1, k) \in \Pi \Leftrightarrow (F \oplus_u G_2, k + \Delta) \in \Pi$.*

Proof. Without loss of generality (by flipping the $\in \Pi$ relation if needed), we assume that Π is a parameterized graph problem with $(G, k) \notin \Pi$ for every graph G and $k < 0$. Let us fix a boundaried graph $F \in \mathcal{F}$ such that there is an integer k with $(F \oplus_u G_1, k) \in \Pi$. This graph F exists since G_1 is non-monotone. Note that $G_1 \equiv_\Pi G_2$ implies that there is an integer k' such that $(F \oplus_u G_2, k') \in \Pi$. For $i \in [2]$, let k_i^* denote the smallest integer such that $(F \oplus_u G_i, k_i^*) \in \Pi$. Since $(G, k) \notin \Pi$ for $k < 0$, these integer k_i^* exist and are non-negative. We now show that $\Delta := k_2^* - k_1^*$ is the only integer that satisfies Equation (8.1).

Suppose for the sake of contradiction, there is an integer $\Delta' \neq \Delta$ that satisfies Equation (8.1). That is, for every $k \in \mathbb{Z}$, we have $(F \oplus_u G_1, k) \in \Pi \Leftrightarrow (F \oplus_u G_2, k + \Delta') \in \Pi$, or equivalently, for every $k \in \mathbb{Z}$, we have $(F \oplus_u G_1, k - \Delta') \in \Pi \Leftrightarrow (F \oplus_u G_2, k) \in \Pi$. First, suppose $\Delta' < \Delta$. Since $(F \oplus_u G_1, k_1^*) \in \Pi$, it follows that $(F \oplus_u G_2, k_1^* + \Delta') \in \Pi$, where $k_1^* + \Delta' < k_1^* + \Delta = k_2^*$, which contradicts k_2^* being the smallest integer k with $(F \oplus_u G_2, k) \in \Pi$. Now, suppose $\Delta' > \Delta$. Then, $(F \oplus_u G_2, k_2^*) \in \Pi$ implies that $(F \oplus_u G_1, k_2^* - \Delta') \in \Pi$, which is again a contradiction to the minimality of k_1^* since $k_2^* - \Delta' < k_2^* - \Delta = k_1^*$.

Thus, there is no $\Delta' \neq \Delta$ that satisfies Equation (8.1). Since $G_1 \equiv_\Pi G_2$, there exists an integer that satisfies Equation (8.1), so this must be Δ . $\square$

We further observe that for any two boundaried graphs $G_1, G_2 \in \mathcal{F}$ with $G_1 \equiv_\Pi G_2$, we have $\Delta_\Pi(G_1, G_2) = -\Delta_\Pi(G_2, G_1)$. Moreover, we show that the function Δ_Π satisfies the following *transitivity* condition.

Lemma 8.3. *Let $G_1, G_2, G_3 \in \mathcal{F}$ be three boundaried graphs that are equivalent under $\equiv_\Pi$. Then, $\Delta_\Pi(G_1, G_3) = \Delta_\Pi(G_1, G_2) + \Delta_\Pi(G_2, G_3)$.*

Proof. If G_1, G_2 , and G_3 are monotone, this lemma trivially holds, so assume that they are non-monotone. By Equation (8.1), for every boundaried graph $F \in \mathcal{F}$ and every integer $k \in \mathbb{Z}$, we have the following:

$$(F \oplus_u G_1, k) \in \Pi \Leftrightarrow (F \oplus_u G_3, k + \Delta_\Pi(G_1, G_3)) \in \Pi,$$

and further

$$\begin{aligned} (F \oplus_u G_1, k) \in \Pi &\Leftrightarrow (F \oplus_u G_2, k + \Delta_\Pi(G_1, G_2)) \in \Pi \\ &\Leftrightarrow (F \oplus_u G_3, k + \Delta_\Pi(G_1, G_2) + \Delta_\Pi(G_2, G_3)) \in \Pi \end{aligned}$$

Due to the uniqueness of Δ_Π , as shown in [Lemma 8.2](#), it follows that $\Delta_\Pi(G_1, G_3) = \Delta_\Pi(G_1, G_2) + \Delta_\Pi(G_2, G_3)$. $\square$

For a graph class $\mathcal{G}$ and two boundaried graphs $G_1, G_2 \in \mathcal{F}$, we further define $G_1 \equiv_{\mathcal{G}} G_2$ if and only if $\mathbf{\Lambda}(G_1) = \mathbf{\Lambda}(G_2)$ and for every $F \in \mathcal{F}$, $F \oplus_u G_1 \in \mathcal{G}$ if and only if $F \oplus_u G_2 \in \mathcal{G}$. It is easy to see that $\equiv_{\mathcal{G}}$ is an equivalence relation. For CMSO₂-definable graph classes, it was proven in [\[BFL⁺16\]](#) (based on [\[Cou90, ALS91, BPT92, AF93, DF99\]](#)) that it has a finite number of equivalence classes.

Lemma 8.4 ([\[BFL⁺16, Lemma 3.2\]](#)). *Let $\mathcal{G}$ be a CMSO₂-definable graph class and let $I \subseteq \mathbb{Z}_{\geq 1}$ be a finite set. Then, the number of equivalence classes of $\equiv_{\mathcal{G}}$ that are subsets of $\mathcal{F}_I$ is finite.*

For a parameterized graph problem Π , a graph class $\mathcal{G}$, and two boundaried graphs $G_1, G_2 \in \mathcal{F}$, we say $G_1 \equiv_{\Pi, \mathcal{G}} G_2$ if and only if $G_1 \equiv_\Pi G_2$ and $G_1 \equiv_{\mathcal{G}} G_2$. We observe that $\equiv_{\Pi, \mathcal{G}}$ is again an equivalence relation and the number of equivalence classes of $\equiv_{\Pi, \mathcal{G}}$ is at most the number of equivalence classes of $\equiv_\Pi$ times the number of equivalence classes of $\equiv_{\mathcal{G}}$. Thus, if Π has FII and $\mathcal{G}$ is CMSO₂-definable, the number of equivalence classes of $\equiv_{\Pi, \mathcal{G}}$ that are subsets of $\mathcal{F}_I$ for a finite $I \subseteq \mathbb{Z}_{\geq 1}$ is finite.

Now, for every equivalence class of $\equiv_{\Pi, \mathcal{G}}$, we want to fix a “good” representative, with which we can replace all subgraphs that are in the same equivalence class. Consider an equivalence class $\mathcal{C}$ of $\equiv_{\Pi, \mathcal{G}}$. A graph $G \in \mathcal{C}$ is called a *progressive representative* of $\mathcal{C}$ if for any graph $G' \in \mathcal{C}$ it holds that $\Delta_\Pi(G', G) \leq 0$, or equivalently, $\Delta_\Pi(G, G') \geq 0$. For the equivalence relation $\equiv_\Pi$ it is known that such a progressive representative always exists for every equivalence class, regardless of whether Π has FII [\[BFL⁺16, FLSZ19\]](#).

Lemma 8.5 ([\[FLSZ19, Lemma 16.11\]](#)). *Let Π be a parameterized graph problem. Then, each equivalence class of $\equiv_\Pi$ has a progressive representative.*

We now have to prove the same thing for the equivalence relation $\equiv_{\Pi, \mathcal{G}}$. Since the proof is almost the same as for [\[FLSZ19, Lemma 16.11\]](#), we move it to [Section A](#).

Lemma 8.6. *Let $\mathcal{G}$ be a graph class and let Π be a parameterized graph problem. Then, each equivalence class of $\equiv_{\Pi, \mathcal{G}}$ has a progressive representative.*

We remark that there might be more than one progressive representative for each equivalence class, however, we can assume that we fix one of them. For this, recall that by definition, all graphs within the same equivalence class of $\equiv_\Pi$ (and thus, $\equiv_{\Pi, \mathcal{G}}$) have the same label set. Thus, every equivalence class of $\equiv_\Pi$ is a subset of $\mathcal{F}_I$ for some finite $I \subseteq \mathbb{Z}_{\geq 1}$, where $\mathcal{F}_I$ is the class of all boundaried graphs with label set I . So, for a fixed $I \subseteq \mathbb{Z}_{\geq 1}$, we can find one progressive representative for each equivalence class that is a subset of $\mathcal{F}_I$.

That is, for each $I \subseteq \mathbb{Z}_{\geq 1}$, we define $\mathcal{S}_I$ to be a set containing exactly one smallest (by number of vertices) progressive representative of each equivalence class of $\equiv_{\Pi, \mathcal{G}}$ that is a subset of $\mathcal{F}_I$. We also define $\mathcal{S}_{\subseteq I} := \bigcup_{I' \subseteq I} \mathcal{S}_{I'}$ and $c_{\Pi, \mathcal{G}}(t) := \max\{|V(G)| : G \in \mathcal{S}_{\subseteq [t]}\}$. Also, let $\varphi_{\Pi, \mathcal{G}, I} : \mathcal{F}_{\subseteq I} \rightarrow \mathcal{S}_{\subseteq I}$ be the function that maps every boundaried graph $G \in \mathcal{F}_{\subseteq I}$ to the unique smallest progressive representative $\varphi_{\Pi, \mathcal{G}, I}(G) \in \mathcal{S}_{\subseteq I}$ of G 's equivalence class under $\equiv_{\Pi, \mathcal{G}}$.

Note that the value of $c_{\Pi, \mathcal{G}}(t)$ only depends on Π , $\mathcal{G}$, and t , and that $c_{\Pi, \mathcal{G}}(t) \geq t$. Recall that, by definition of $\varphi_{\Pi, \mathcal{G}, [t]}$ and Δ_Π , it holds for every graph $G \in \mathcal{F}_{\subseteq [t]}$ and every pair $(F, k) \in \mathcal{F} \times \mathbb{Z}$ that $(F \oplus_u G, k) \in \Pi$ if and only if $(F \oplus_u \varphi_{\Pi, \mathcal{G}, [t]}(G), k + \Delta_\Pi(G, \varphi_{\Pi, \mathcal{G}, [t]}(G))) \in \Pi$.

Before constructing the protrusion replacement automaton, we need to take care of one last technicality. Note that the replacement function $\varphi_{\Pi, \mathcal{G}, [t]}$ replaces boundaried graphs that, by definition, have a fixed labeling. In particular, the replacement is different for different boundaried graphs with the same underlying non-boundaried graph and boundary set, but different labelings. This is a problem since we cannot maintain a global labeling for our dynamic

protrusion decomposition, but need to recompute labelings on the fly. To do this, we now show that first changing the labeling and then replacing is equivalent to first replacing and then changing the labeling.

Lemma 8.7. *Let Π be a parameterized graph problem and let t be an integer. Let $X = (G_X, B, \Lambda)$ be a t -boundaried graph, and let $\Lambda': B \rightarrow [t]$ be another labeling. Let $X' = (G_X, B, \Lambda')$, $Y := \varphi_{\Pi, \mathcal{G}, [t]}(X) = (G'_X, B, \Lambda)$, and $Y' = (G'_X, B, \Lambda')$. Then, we have that $X' \equiv_{\Pi} Y'$ and $\Delta_{\Pi}(X', Y') = \Delta_{\Pi}(X, Y)$.*

Proof. Let $(F', k) \in \mathcal{F} \times \mathbb{Z}$ be a fixed, but arbitrary pair. We show that $(F' \oplus_u X', k) \in \Pi$ if and only if $(F' \oplus_u Y', k + \Delta_{\Pi}(X, Y)) \in \Pi$. Note that for every such boundaried graph $F' \in \mathcal{F}$, there is a graph $\tilde{F}' \in \mathcal{F}$ with $\Lambda(\tilde{F}') = \Lambda(X) (= \Lambda(Y))$ such that $\tilde{F}' \oplus_u X' = F' \oplus_u X'$ and $\tilde{F}' \oplus_u Y' = F' \oplus_u Y'$. This boundaried graph $\tilde{F}'$ can be obtained from F' by removing boundary vertices with labels in $\Lambda(F') \setminus \Lambda(X')$ from the boundary of F' and inserting isolated vertices for every label in $\Lambda(X) \setminus \Lambda(F')$. Thus, without loss of generality, we can assume that $\Lambda(F') = \Lambda(X') (= \Lambda(Y'))$, and moreover, $F' = (G_F, B, \Lambda')$ for some graph G_F . We consider the boundaried graph $F = (G_F, B, \Lambda)$ and observe that $F \oplus_u X = F' \oplus_u X'$ and $F \oplus_u Y = F' \oplus_u Y'$. Thus, we have the following relations:

$$\begin{aligned} (F' \oplus_u X', k) &\in \Pi \\ \Leftrightarrow (F \oplus_u X, k) &\in \Pi & (F' \oplus_u X' = F \oplus_u X) \\ \Leftrightarrow (F \oplus_u Y, k + \Delta_{\Pi}(X, Y)) &\in \Pi & (Y = \varphi_{\Pi, \mathcal{G}, [t]}(X)) \\ \Leftrightarrow (F' \oplus_u Y', k + \Delta_{\Pi}(X, Y)) &\in \Pi & (F \oplus_u Y = F' \oplus_u Y'). \end{aligned}$$

It follows that $X' \equiv_{\Pi} Y'$ and $\Delta_{\Pi}(X', Y') = \Delta_{\Pi}(X, Y)$. $\square$

Our goal is now to provide a tree decomposition automaton that, given a graph G together with an annotated tree decomposition $\mathcal{T} = (T, \text{bag}, \text{edges})$, for every node $x \in V(T)$ computes the replacement $\varphi_{\Pi, \mathcal{G}, [t]}(X)$, where $X = (G_x, \text{bag}(x), \Lambda)$ and Λ is some labeling, which we fix in the following.

We assume without loss of generality that the set of vertices $V(G)$ of our graph is totally ordered by a relation $\prec$ that can be computed in $\mathcal{O}(1)$ time, e.g., by indexing them by integers. Let $B \subseteq V(G)$ be a set of size $|B| \leq t$. An injective labeling $\Lambda: B \rightarrow [t]$ is called *in-order* if for any two vertices $u, v \in B$, $u \prec v$ implies that $\Lambda(u) < \Lambda(v)$, and $\max_{v \in B} \Lambda(v) = |B|$. Note that there is exactly one injective in-order labeling for every such set $B \subseteq V(G)$, which we denote by Λ_B^* . Then, for a node $x \in V(T)$, we say that $(G_x, \text{bag}(x), \Lambda_{\text{bag}(x)}^*)$ is the boundaried graph that *corresponds to x* . We are now ready to construct the protrusion replacement automaton that, given a graph G together with a tree decomposition $\mathcal{T}$ of width $t - 1$, for every node $x \in V(T)$ with corresponding boundaried graph $X = (G_x, \text{bag}(x), \Lambda_{\text{bag}(x)}^*)$ computes the progressive representative $\varphi_{\Pi, \mathcal{G}, [t]}(X)$ and the transposition constant $\Delta_{\Pi}(X, \varphi(X))$.

Lemma 8.8. *Let $\mathcal{G}$ be a CMSO₂-definable graph class, Π a parameterized graph problem that has FII, and $t \in \mathbb{Z}_{\geq 1}$. There exists a tree decomposition automaton $\mathcal{A}$ of width $t - 1$ such that the following holds: Let G be a graph together with an annotated tree decomposition $\mathcal{T} = (T, \text{bag}, \text{edges})$ of width $t - 1$. Then, given the run $\rho_{\mathcal{A}}^T(x)$ of $\mathcal{A}$ on a node $x \in V(T)$, we can determine the progressive representative $\varphi_{\Pi, \mathcal{G}, [t]}(X)$ together with the transposition constant $\Delta_{\Pi}(X, \varphi_{\Pi, \mathcal{G}, [t]}(X))$ in $\mathcal{O}(1)$ time, where $X = (G_x, \text{bag}(x), \Lambda_B^*)$ is the boundaried graph that corresponds to x . The evaluation time τ of $\mathcal{A}$ is a constant depending on $\mathcal{G}$, Π and t .*

Proof. We start by arguing that it suffices to construct an automaton $\mathcal{A} = (Q, \iota, \emptyset, \delta)$ that assumes that the annotated tree decomposition is nice. To remove this assumption, we construct a modified transition map δ' as follows. Consider a node $x \in V(T)$ with two children y_1 and y_2

(if x has one child, this is handled analogously). We conceptually replace the edge xy_i by a path that successively introduces and forgets one node at a time to transform $\text{bag}(y_i)$ into $\text{bag}(x)$ for both $i \in \{1, 2\}$. This makes it “locally nice” and we can use the transition map δ on these paths: Given the state of the children, $\rho_{\mathcal{A}}^{\mathcal{T}}(y_1)$ and $\rho_{\mathcal{A}}^{\mathcal{T}}(y_2)$, we apply δ along the nodes of the path, until the resulting run $\rho_{\mathcal{A}}^{\mathcal{T}}(x)$ is obtained. Since the length of each path is at most $|\text{bag}(x)| \leq t$, the evaluation time τ still only depends on $\mathcal{G}$, Π , and t .

We will now describe a dynamic programming scheme that, given a graph G with an annotated nice tree decomposition $\mathcal{T} = (T, \text{bag}, \text{edges})$ of width $t - 1$, for every node $x \in V(T)$ with corresponding boundaried graph $X = (G_x, \text{bag}(x), \Lambda_{\text{bag}(x)}^*)$, determines the tuple $(\varphi_{\Pi, \mathcal{G}, [t]}(X), \Delta_{\Pi}(G_x, \varphi_{\Pi, \mathcal{G}, [t]}(X)))$. In the following, we drop the subscripts and set $\varphi(G) := \varphi_{\Pi, \mathcal{G}, [t]}(G)$ and $\Delta(G) := \Delta_{\Pi}(G, \varphi(G))$ for every boundaried graph $G \in \mathcal{F}_{\subseteq[t]}$. The dynamic programming algorithm can be implemented as a tree decomposition automaton $\mathcal{A} = (Q, \emptyset, \iota, \delta)$, whose states correspond to pairs $(G, \Delta) \in \mathcal{S}_{\subseteq[t]} \times \mathbb{Z}$, such that for every node $x \in V(T)$ the run $\rho_{\mathcal{A}}(x)$ corresponds to the pair $(\varphi(X), \Delta(X))$. Specifically, we start with the base case of the dynamic program, i.e., the leaf nodes of the tree decomposition (corresponding to the initial mapping ι in the automaton). Then, for the three cases of a non-leaf node x in a nice tree decomposition – forget, introduce, and join – we describe how we can determine the pair $(\varphi(X), \Delta(X))$ from the state(s) of the child(ren) (corresponding to the transition mapping δ of $\mathcal{A}$). For this, we can assume that for every boundaried graph $G \in \mathcal{F}_{\subseteq[t]}$ on at most $2 \cdot c_{\Pi, \mathcal{G}}(t)$ vertices, we can determine $\varphi(G)$ together with the value $\Delta(G)$ by hardcoding this mapping in our algorithm’s source code, similar to the protrusion replacement algorithm from Fomin, Lokshantanov, Saurabh, and Zehavi (see Lemma 16.18 in [FLSZ19]).

First, recall that for a node $x \in V(T)$, the graph G_x is defined as $G_x = (V_x, E_x)$, where $V_x = \bigcup_{y \in \text{desc}(x)} \text{bag}(y)$ and $E_x = \bigcup_{y \in \text{desc}(x)} \text{edges}(y)$. Further recall that $\text{edges}(x)$ contains exactly the edges of G , for which x is the shallowest node that contains both endpoints, so $\text{edges}(x) = \emptyset$, unless the parent of x is a forget node. In particular, for a node x with children y and z (where $y = z$ if x has only one child), the question whether $E_x \supsetneq E_y \cup E_z$ depends solely on the parent of x . For the sake of simplification, we want $E_x = E_y \cup E_z$ for every forget, introduce, and join node x with children y and z (again, $y = z$ if x is a forget or introduce node). To achieve this, we introduce a new type of node, called *edges node*, where every edges node x has exactly one child y with $\text{bag}(x) = \text{bag}(y)$. From now on, we assume that the only child of every forget node is an edges node. Then, we have $\text{edges}(x) \neq \emptyset$ only if x is an edges node. This assumption does not change our requirements on $\mathcal{T}$, since processing an edges node before every forget node can also be implemented as part of the forget node procedure. We now describe how we process these edges nodes before taking care of the standard nodes: leaf, forget, introduce, and join.

Edges node: Let $x \in V(T)$ be an edges node with child y . Note that $\text{bag}(x) = \text{bag}(y)$, so let $X = (G_x, \text{bag}(x), \Lambda_{\text{bag}(x)}^*)$ and $Y = (G_y, \text{bag}(x), \Lambda_{\text{bag}(x)}^*)$ be the boundaried graph that corresponds to x and y . Let further $G_B = (\text{bag}(x), \text{edges}(x))$, and $B = (G_B, \text{bag}(x), \Lambda_{\text{bag}(x)}^*)$. Then, $X = B \oplus_b Y$. We now show that $\varphi(X) = \varphi(B \oplus_b \varphi(Y))$ and $\Delta(X) = \Delta(Y) + \Delta(B \oplus_b \varphi(Y))$. From our dynamic program, we know the pair $(\varphi(Y), \Delta(Y))$. Note that $|V(B \oplus_b \varphi(Y))| = |V(\varphi(Y))| \leq c_{\Pi, \mathcal{G}}(t)$, so we can explicitly construct the boundaried graph $B \oplus_b \varphi(Y)$ and then look up the progressive representative $\varphi(B \oplus_b \varphi(Y))$ in our hardcoded mapping.

Claim 8.9. $\varphi(X) = \varphi(B \oplus_b \varphi(Y))$ and $\Delta(X) = \Delta(Y) + \Delta(B \oplus_b \varphi(Y))$.

Proof of the claim. Let $(F, k) \in \mathcal{F} \times \mathbb{Z}$ be an arbitrary pair. We apply the definitions of φ and Δ and use the associativity of $\oplus_b$ to obtain the following:

$$\begin{aligned}
(F \oplus_b X, k) &\in \Pi \\
\Leftrightarrow (F \oplus_b (B \oplus_b Y), k) &\in \Pi & (X = B \oplus_b Y) \\
\Leftrightarrow ((F \oplus_b B) \oplus_b Y, k) &\in \Pi & (\text{associativity}) \\
\Leftrightarrow ((F \oplus_b B) \oplus_b \varphi(Y), k + \Delta(Y)) &\in \Pi & (F \oplus_b B \in \mathcal{F}, \text{ definitions of } \varphi, \Delta).
\end{aligned}$$

It follows that $X \equiv_{\Pi} B \oplus_b \varphi(Y)$, so $\varphi(X) = \varphi(B \oplus_b \varphi(Y))$, and further that $\Delta(X, B \oplus_b \varphi(Y)) = \Delta(Y)$. Applying the transitivity of Δ_{Π} , which we showed in [Lemma 8.3](#), we obtain that $\Delta(X) = \Delta_{\Pi}(X, \varphi(X)) = \Delta_{\Pi}(X, B \oplus_b \varphi(Y)) + \Delta_{\Pi}(B \oplus_b \varphi(Y), \varphi(X)) = \Delta(Y) + \Delta(B \oplus_b \varphi(Y))$. $\triangleleft$

From now on we can assume for every non-edges node x with children y and z (where $y = z$ if x has only one child), we have $\text{edges}(x) = \emptyset$, and thus $E(G_x) = E(G_y) \cup E(G_z)$.

Leaf node: Let $x \in V(T)$ be a leaf node and let $X = (G_x, \text{bag}(x), \Lambda)$ be the corresponding boundaried graph. Then, $|V(G_x)| \leq t \leq c_{\Pi}(t)$, and thus we can find the pair $(\varphi(X), \Delta(X))$ by looking it up in our hardcoded mapping.

Forget node: Let $x \in V(T)$ be a forget node with child y and let v be the forgotten vertex. Let $X = (G_x, \text{bag}(x), \Lambda_{\text{bag}(x)}^*)$ and $Y = (G_y, \text{bag}(y), \Lambda_{\text{bag}(y)}^*)$ be the boundaried graphs that correspond to x and y , respectively. Note that we have $\text{bag}(y) = \text{bag}(x) \cup \{v\}$. We define the injective labeling $\Lambda^*: \text{bag}(y) \rightarrow [t]$ as the extension of $\Lambda_{\text{bag}(x)}^*$ to $\text{bag}(y)$, i.e.,

$$\Lambda^*(u) = \begin{cases} \Lambda_{\text{bag}(x)}^*(u) & \text{if } u \in \text{bag}(x) \\ \max_{u' \in \text{bag}(x)} \Lambda_{\text{bag}(x)}^*(u') + 1 & \text{if } u = v \end{cases}.$$

Let then $Y' = (G_y, \text{bag}(y), \Lambda^*)$, $\tilde{Y} = \varphi(Y) = (G'_y, \text{bag}(y), \Lambda_{\text{bag}(y)}^*)$, $\tilde{Y}' = (G'_y, \text{bag}(y), \Lambda^*)$, and $\tilde{Y}'' = (G'_y, \text{bag}(x), \Lambda_{\text{bag}(x)}^*)$. Note that by [Lemma 8.7](#), we have $Y' \equiv_{\Pi} \tilde{Y}'$ and $\Delta_{\Pi}(Y', \tilde{Y}') = \Delta_{\Pi}(Y, \tilde{Y}) = \Delta(Y)$.

From our dynamic program, we know the boundaried graph $\tilde{Y} = \varphi(Y)$ together with the integer value $\Delta(Y)$. In order to show that we can determine $\varphi(X)$ and $\Delta(X)$ directly from $(\varphi(Y), \Delta(Y))$, we first show that $\varphi(X) = \varphi(\tilde{Y}'')$ and $\Delta(X) = \Delta(Y) + \Delta(\tilde{Y}'')$. Then, since $G'_y \in \mathcal{S}_{\subseteq[t]}$, and thus, $|V(G'_y)| \leq c_{\Pi, \mathcal{G}}(t)$, we can look up the replacement $\varphi(\tilde{Y}'') = \varphi(X)$ together with the transposition constant $\Delta(\tilde{Y}'')$ in our hardcoded mapping. Adding the known value $\Delta(Y)$ to this transposition constant, we obtain $\Delta(X)$.

Claim 8.10. $\varphi(X) = \varphi(\tilde{Y}'')$ and $\Delta(X) = \Delta(Y) + \Delta(\tilde{Y}'')$.

Proof of the claim. Let $(F, k) \in \mathcal{F} \times \mathbb{Z}$ be a fixed, but arbitrary pair. We show that $(F \oplus_u X, k) \in \Pi$ if and only if $(F \oplus_u \tilde{Y}'', k + \Delta(Y)) \in \Pi$. If $\Lambda(F) \neq \Lambda(X) (= \Lambda(\tilde{Y}''))$, we are done, so we can assume that $F = (G_F, \text{bag}(x), \Lambda_{\text{bag}(x)}^*)$ for some graph G_F . We consider the boundaried graph $F^* = (G_F \cup \{v\}, \text{bag}(y), \Lambda^*)$ and observe that $F \oplus_u X = F^* \oplus_u Y'$. For this, note that due to the edges nodes, we have $G_x = G_y$, and all edges that are incident to v , both in $F \oplus_u X$ and $F^* \oplus_u Y'$, are in $E(G_x) = E(G_y)$. Similarly, we also have $F \oplus_u \tilde{Y}'' = F^* \oplus_u \tilde{Y}'$. It follows that

$$\begin{aligned}
(F \oplus_u X, k) &\in \Pi \\
\Leftrightarrow (F^* \oplus_u Y', k) &\in \Pi & (F \oplus_u X = F^* \oplus_u Y') \\
\Leftrightarrow (F^* \oplus_u \tilde{Y}', k + \Delta(Y)) &\in \Pi & (Y' \equiv_{\Pi} \tilde{Y}' \text{ and } \Delta_{\Pi}(Y', \tilde{Y}') = \Delta(Y)) \\
\Leftrightarrow (F \oplus_u \tilde{Y}'', k + \Delta(Y)) &\in \Pi & (F^* \oplus_u \tilde{Y}' = F \oplus_u \tilde{Y}'').
\end{aligned}$$

It follows that $X \equiv_{\Pi} \tilde{Y}''$, so $\varphi(X) = \varphi(\tilde{Y}'')$, and $\Delta_{\Pi}(X, \tilde{Y}'') = \Delta(Y)$. Due to the transitivity of Δ_{Π} (see [Lemma 8.3](#)), this implies that $\Delta(X) = \Delta_{\Pi}(X, \varphi(X)) = \Delta_{\Pi}(X, \tilde{Y}'') + \Delta_{\Pi}(\tilde{Y}'', \varphi(X)) = \Delta(Y) + \Delta(\tilde{Y}'')$.

◁

Introduce node: Let $x \in V(T)$ be an introduce node with child y and let v be the introduced vertex. Let $X = (G_x, \text{bag}(x), \Lambda_{\text{bag}(x)}^*)$ and $Y = (G_y, \text{bag}(y), \Lambda_{\text{bag}(y)}^*)$ be the boundaried graphs that correspond to x and y , respectively. Note that we have $\text{bag}(y) = \text{bag}(x) \setminus \{v\}$. Conversely to the forget node, we define the injective labeling $\Lambda^*: \text{bag}(y) \rightarrow [t]$ as the restriction of $\Lambda_{\text{bag}(x)}^*$ to $\text{bag}(y)$, i.e., $\Lambda^* = \Lambda_{\text{bag}(x)}^* \upharpoonright_{\text{bag}(y)}$. Let $Y' = (G_y, \text{bag}(y), \Lambda^*)$, $\tilde{Y} = \varphi(Y) = (G'_y, \text{bag}(y), \Lambda_{\text{bag}(y)}^*)$, $\tilde{Y}' = (G'_y, \text{bag}(y), \Lambda^*)$, and $\tilde{Y}'' = (G'_y \cup \{v\}, \text{bag}(x), \Lambda_{\text{bag}(x)}^*)$. Note that by [Lemma 8.7](#), we have $Y' \equiv_{\Pi} \tilde{Y}'$ and $\Delta_{\Pi}(Y', \tilde{Y}') = \Delta_{\Pi}(Y, \tilde{Y}) = \Delta(Y)$.

Again, we show that $\varphi(X) = \varphi(\tilde{Y}'')$ and $\Delta(X) = \Delta(Y) + \Delta(\tilde{Y}'')$. Knowing the pair $(\varphi(Y), \Delta(Y))$, and in particular the graph G'_y , where $|V(G'_y \cup \{v\})| \leq c_{\Pi, \mathcal{G}}(t) + 1$, we can look up the replacement $\varphi(\tilde{Y}'')$ together with the transposition constant $\Delta(\tilde{Y}'')$.

Claim 8.11. $\varphi(X) = \varphi(\tilde{Y}'')$ and $\Delta(X) = \Delta(Y) + \Delta(\tilde{Y}'')$.

Proof of the claim. Let $(F, k) \in \mathcal{F} \times \mathbb{Z}$ be a fixed, but arbitrary pair. We show that $(F \oplus_u X, k) \in \Pi$ if and only if $(F \oplus_u \tilde{Y}'', k + \Delta(Y)) \in \Pi$. If $\Lambda(F) \neq \Lambda(X) (= \Lambda(\tilde{Y}''))$, we are done, so we can assume that $F = (G_F, \text{bag}(x), \Lambda_{\text{bag}(x)}^*)$ for some graph G_F . We consider the boundaried graph $F^* = (G_F, \text{bag}(y), \Lambda^*)$ and observe that $F \oplus_u X = F^* \oplus_u Y'$. For this, note that there are no edges incident to v in G_x , since v is the introduced vertex and $\text{edges}(x) = \emptyset$. Similarly, since v is isolated in $\tilde{Y}''$, we also have that $F \oplus_u \tilde{Y}'' = F^* \oplus_u \tilde{Y}'$. It follows that

$$\begin{aligned} (F \oplus_u X, k) &\in \Pi \\ \Leftrightarrow (F^* \oplus_u Y', k) &\in \Pi & (F \oplus_u X = F^* \oplus_u Y') \\ \Leftrightarrow (F^* \oplus_u \tilde{Y}', k + \Delta(Y)) &\in \Pi & (Y' \equiv_{\Pi} \tilde{Y}' \text{ and } \Delta_{\Pi}(Y', \tilde{Y}') = \Delta(Y)) \\ \Leftrightarrow (F \oplus_u \tilde{Y}'', k + \Delta(Y)) &\in \Pi & (F^* \oplus_u \tilde{Y}' = F \oplus_u \tilde{Y}''). \end{aligned}$$

It follows that $X \equiv_{\Pi} \tilde{Y}''$, so $\varphi(X) = \varphi(\tilde{Y}'')$, and further $\Delta_{\Pi}(X, \tilde{Y}'') = \Delta(Y)$. Due to the transitivity of Δ_{Π} (see [Lemma 8.3](#)), this implies that $\Delta(X) = \Delta_{\Pi}(X, \varphi(X)) = \Delta_{\Pi}(X, \tilde{Y}'') + \Delta_{\Pi}(\tilde{Y}'', \varphi(X)) = \Delta(Y) + \Delta(\tilde{Y}'')$.

◁

Join node: Let $x \in V(T)$ be a join node with children y and z and let $X = (G_x, \text{bag}(x), \Lambda_x)$, $Y = (G_y, \text{bag}(y), \Lambda_y)$, and $Z = (G_z, \text{bag}(z), \Lambda_z)$ be the corresponding boundaried graphs. Then, $X = Y \oplus_b Z$. Again, from our dynamic program, we know the pairs $(\varphi(Y), \Delta(Y))$ and $(\varphi(Z), \Delta(Z))$, and we need to determine $(\varphi(X), \Delta(X))$. For this, we show that $\varphi(X) = \varphi(\varphi(Y) \oplus_b \varphi(Z))$ and $\Delta(X) = \Delta(Y) + \Delta(Z) + \Delta(\varphi(Y) \oplus_b \varphi(Z))$. Note that $\varphi(Y)$ and $\varphi(Z)$ have at most $c_{\Pi, \mathcal{G}}(t)$ vertices each, and thus $\varphi(Y) \oplus_b \varphi(Z)$ has at most $2 \cdot c_{\Pi, \mathcal{G}}(t)$ vertices. Hence, we can look up the graph $\varphi(\varphi(Y) \oplus_b \varphi(Z))$ together with the value $\Delta(\varphi(Y) \oplus_b \varphi(Z))$ in our hardcoded mapping, and thus determine $(\varphi(X), \Delta(X))$ from the state of the two children y and z of x .

Claim 8.12. $\varphi(X) = \varphi(\varphi(Y) \oplus_b \varphi(Z))$ and $\Delta(X) = \Delta(Y) + \Delta(Z) + \Delta(\varphi(Y) \oplus_b \varphi(Z))$.

Proof of the claim. Let $(F, k) \in \mathcal{F} \times \mathbb{Z}$ be an arbitrary pair. Similarly to the edges node, we apply the definitions of φ and Δ and use the associativity and commutativity of $\oplus_b$ to obtain the following equivalences.

$$\begin{aligned}
& (F \oplus_b X, k) \in \Pi \\
& \Leftrightarrow (F \oplus_b (Y \oplus_b Z), k) \in \Pi & (X = Y \oplus_b Z) \\
& \Leftrightarrow ((F \oplus_b Y) \oplus_b Z, k) \in \Pi & (\text{associativity}) \\
& \Leftrightarrow ((F \oplus_b Y) \oplus_b \varphi(Z), k + \Delta(Z)) \in \Pi & (F \oplus_b Y \in \mathcal{F}, \text{definitions of } \varphi, \Delta) \\
& \Leftrightarrow ((F \oplus_b \varphi(Z)) \oplus_b Y, k + \Delta(Z)) \in \Pi & (\text{associativity, commutativity}) \\
& \Leftrightarrow ((F \oplus_b \varphi(Z)) \oplus_b \varphi(Y), k + \Delta(Z) + \Delta(Y)) \in \Pi & (F \oplus_b \varphi(Z) \in \mathcal{F}, \text{definitions of } \varphi, \Delta) \\
& \Leftrightarrow (F \oplus_b (\varphi(Y) \oplus_b \varphi(Z)), k + \Delta(Y) + \Delta(Z)) \in \Pi & (\text{associativity, commutativity})
\end{aligned}$$

It follows that X and $\varphi(Y) \oplus_b \varphi(Z)$ are equivalent under $\equiv_\Pi$, so $\varphi(X) = \varphi(\varphi(Y) \oplus_b \varphi(Z))$. Moreover, it follows that $\Delta_\Pi(X, \varphi(Y) \oplus_b \varphi(Z)) = \Delta(Y) + \Delta(Z)$. Together with [Lemma 8.3](#), we have $\Delta(X) = \Delta_\Pi(X, \varphi(X)) = \Delta_\Pi(X, \varphi(Y) \oplus_b \varphi(Z)) + \Delta_\Pi(\varphi(Y) \oplus_b \varphi(Z), \varphi(X)) = \Delta(Y) + \Delta(Z) + \Delta(\varphi(Y) \oplus_b \varphi(Z))$. $\triangleleft$

This completes the proof. $\square$

8.2 The dynamic kernelization data structure

In this section, we finally describe our dynamic kernelization data structure by combining our data structure from [Theorem 7.5](#) with our protrusion replacement automaton from [Lemma 8.8](#). First, we describe how the protrusion replacement automaton can be used to obtain a kernel in the static case. Then, we show that we can maintain this kernel in the dynamic case by using our dynamic protrusion decomposition data structure from [Theorem 7.5](#).

Let $\mathcal{G}$ be a CMSO₂-definable graph class and Π a parameterized graph problem. Let $G \in \mathcal{G}$ be a graph together with a normal (p, t) -protrusion decomposition $\mathcal{T} = (T, \text{bag})$ with root r . Let $\text{chd}(r) = \{c_1, \dots, c_q\}$ be the set of children of r . For $1 \leq i \leq q$, let $G_i = (G_{c_i}, \text{bag}(c_i), \Lambda_{\text{bag}(c_i)}^*)$ be the corresponding boundaried graph. We observe that $G = G[\text{bag}(r)] \cup \bigcup_{i=1}^q G_{c_i}$. By the definition of protrusion decompositions, we have $V(G_i) \cap V(G_j) \subseteq \text{bag}(r)$ for distinct i and j . Moreover, due to the `edges` function, there is no edge $uv \in E(G_i)$ with $u, v \in \text{bag}(r)$ (for $1 \leq i \leq q$). Thus, every edge of G is attributed to exactly one G_i or to $G[\text{bag}(r)]$ and when replacing one boundaried subgraph G_i , the other boundaried subgraphs G_j for $j \neq i$ are unaffected. We further remark that since $\mathcal{T}$ is normal, we have $\text{bag}(r) \subseteq \bigcup_{i=1}^q \text{bag}(c_i)$.

We define the graph $K = K(G, \mathcal{T})$ as the graph obtained by replacing each G_i with its progressive representative $\varphi_{\Pi, \mathcal{G}, [t+1]}(G_i)$ for every $1 \leq i \leq q$. That is, we replace the boundaried subgraphs G_i one by one, producing a sequence $G = G^{(0)}, G^{(1)}, \dots, G^{(q)} = K$ of graphs, where $G^{(i)}$ is the graph obtained from $G^{(i-1)}$ by replacing G_i with $\varphi_{\Pi, \mathcal{G}, [t+1]}(G_i)$. In particular, if we assume $G^{(i-1)} = H_{i-1} \oplus_u G_i$ for some boundaried graph H_{i-1} , then $G^{(i)} = H_{i-1} \oplus_u \varphi_{\Pi, \mathcal{G}, [t+1]}(G_i)$. As mentioned before, this replacement does not affect the other subgraphs G_j , so in particular, we have $G_{c_j} \subseteq G^{(i)}$ for $0 \leq i < j \leq q$, and this sequence of replacements is well-defined. We further remark that the graph K is independent of the order $c_1, \dots, c_q$ of the root-children. We additionally define $\Delta = \Delta(G, \mathcal{T}) = \sum_{i=1}^q \Delta_\Pi(G_i, \varphi_{\Pi, \mathcal{G}, [t+1]}(G_i))$. Now, we first show that (K, Δ) is indeed a kernel, $K \in \mathcal{G}$, $\Delta \leq 0$, and $|V(K)| \leq \mathcal{O}(p)$. Afterwards, we show how we can maintain $K(G, \mathcal{T})$ and $\Delta(G, \mathcal{T})$ efficiently for a dynamic graph G with the dynamic protrusion decomposition $\mathcal{T}$ from [Theorem 7.5](#).

Lemma 8.13. *Let $\mathcal{G}$ be a CMSO₂-definable graph class and Π be a parameterized graph problem. Let $G \in \mathcal{G}$ be a graph together with a normal (p, t) -protrusion decomposition $\mathcal{T} = (T, \text{bag})$. Let $K = K(G, \mathcal{T})$ and $\Delta = \Delta(G, \mathcal{T})$ be as defined above. Then, (K, Δ) is a kernel for Π on G , $K \in \mathcal{G}$, $\Delta \leq 0$, and $|V(K)| \leq \mathcal{O}(p)$. The hidden factors in the $\mathcal{O}$ -notation depend only on Π , $\mathcal{G}$, and t .*

Proof. We start by showing that $|V(K)| \leq \mathcal{O}(p)$ and $\Delta \leq 0$. For this, let r be the root of the (p, t) -protrusion decomposition. It holds that $|\text{bag}(r)| \leq p$ and r has at most $q \leq p$ children. In K , each of these children corresponds to a boundaried graph $\varphi_{\Pi, \mathcal{G}, [t+1]}(G_i)$ for $1 \leq i \leq q$ on at most $c_{\Pi}(t+1)$ vertices. Thus, we have $|V(K)| \leq p + q \cdot c_{\Pi}(t+1) \leq \mathcal{O}(p)$, where the hidden factors in the $\mathcal{O}$ -notation depend only on Π , $\mathcal{G}$, and t . Moreover, by the definition of progressive representatives, we have $\Delta_{\Pi}(G_i, \varphi_{\Pi, \mathcal{G}, [t+1]}(G_i)) \leq 0$ for every $1 \leq i \leq q$, and thus $\Delta \leq 0$.

We now continue with showing that $K \in \mathcal{G}$.

Claim 8.14. $K \in \mathcal{G}$.

Proof of the claim. By the definition of $\equiv_{\mathcal{G}}$, and thus $\varphi_{\Pi, \mathcal{G}, [t+1]}$, we have for every boundaried graph $F \in \mathcal{F}$ and every $1 \leq i \leq q$, $F \oplus_u G_i \in \mathcal{G}$ if and only if $F \oplus_u \varphi_{\Pi, \mathcal{G}, [t+1]}(G_i) \in \mathcal{G}$. In particular, we have $G^{(i-1)} = H_{i-1} \oplus_u G_i \in \mathcal{G}$ if and only if $H_{i-1} \oplus_u \varphi_{\Pi, \mathcal{G}, [t+1]}(G_i) = G^{(i)} \in \mathcal{G}$ for every $1 \leq i \leq q$. Thus, we have

$$G = G^{(0)} \in \mathcal{G} \Leftrightarrow G^{(1)} \in \mathcal{G} \Leftrightarrow G^{(2)} \in \mathcal{G} \Leftrightarrow \dots \Leftrightarrow G^{(q)} = K \in \mathcal{G}.$$

Since by assumption $G \in \mathcal{G}$, it follows that $K \in \mathcal{G}$. $\triangleleft$

Finally, we show that (K, Δ) is a kernel for Π on G .

Claim 8.15. For every $k \in \mathbb{Z}$, we have $(G, k) \in \Pi$ if and only if $(K, k + \Delta) \in \Pi$.

Proof of the claim. By definition of $\varphi_{\Pi, \mathcal{G}, [t+1]}$ and Δ_{Π} , we have for every boundaried graph $F \in \mathcal{F}$, every integer $k \in \mathbb{Z}$, and every $1 \leq i \leq q$, $(F \oplus_u G_i, k) \in \Pi$ if and only if $(F \oplus_u \varphi_{\Pi, \mathcal{G}, [t+1]}(G_i), k + \Delta_{\Pi}(G_i, \varphi_{\Pi, \mathcal{G}, [t+1]}(G_i))) \in \Pi$. Thus, it follows that for every integer k and every $1 \leq i \leq q$, $(G^{(i-1)}, k) = (H_{i-1} \oplus_u G_i, k) \in \Pi$ if and only if $(H_{i-1} \oplus_u \varphi_{\Pi, \mathcal{G}, [t+1]}(G_i), k + \Delta_{\Pi}(G_i, \varphi_{\Pi, \mathcal{G}, [t+1]}(G_i))) = (G^{(i)}, k + \Delta_{\Pi}(G_i, \varphi_{\Pi, \mathcal{G}, [t+1]}(G_i))) \in \Pi$. In total, we have for every integer k ,

$$\begin{aligned} (G, k) \in \Pi &\Leftrightarrow (G^{(0)}, k) \in \Pi \Leftrightarrow (G^{(1)}, k + \Delta_{\Pi}(G_1, \varphi_{\Pi, \mathcal{G}, [t+1]}(G_1))) \in \Pi \\ &\Leftrightarrow (G^{(2)}, k + \Delta_{\Pi}(G_1, \varphi_{\Pi, \mathcal{G}, [t+1]}(G_1)) + \Delta_{\Pi}(G_2, \varphi_{\Pi, \mathcal{G}, [t+1]}(G_2))) \in \Pi \\ &\Leftrightarrow \dots \Leftrightarrow (G^{(q)}, k + \sum_{i=1}^q \Delta_{\Pi}(G_i, \varphi_{\Pi, \mathcal{G}, [t+1]}(G_i))) \in \Pi \Leftrightarrow (K, \Delta) \in \Pi. \end{aligned}$$

$\triangleleft$

This completes the proof. $\square$

Finally, we are ready to construct the kernelization data structure by combining our data structure from [Theorem 7.5](#) with our protrusion replacement automaton from [Lemma 8.8](#) to maintain the kernel (K, Δ) for Π and $\mathcal{G}$ on a dynamic graph G . We first give a kernelization algorithm with the size of the smallest treewidth- η -modulator as parameter. Then, we use the treewidth-boundedness to lift the parameter to the solution size.

Lemma 8.16. Let $\mathcal{G}$ be a CMSO₂-definable graph class that excludes a topological minor, Π a parameterized graph problem that has FII, and η an integer.

There exists a data structure that, for a dynamic graph $G \in \mathcal{G}$, maintains a kernel (K, Δ) for Π on G so that $K \in \mathcal{G}$, $\Delta \leq 0$, and $|K| \leq \mathcal{O}(\text{tw-mod}_{\eta}(G))$. The data structure supports the following operations:

- **Init(G):** Initialize the data structure with a graph $G \in \mathcal{G}$. Outputs (K, Δ) . Runs in $\mathcal{O}(|G| \log |G|)$ amortized time.
- **AddVertex(v):** Given a new vertex $v \notin V(G)$, add v to G .

- **DeleteVertex(v):** Given an isolated vertex $v \in V(G)$, remove v from G .
- **AddEdge(e):** Given a new edge $e \in \binom{V(G)}{2} \setminus E(G)$, add e to G .
- **DeleteEdge(e):** Given an edge $e \in E(G)$, remove e from G .

Each update takes $\mathcal{O}(\log |G|)$ amortized time. For each operation, the changes to K can be described as a sequence of (hyper)graph operations of size $\mathcal{O}(1)$, which is returned. The change to Δ is also returned (and can be arbitrary). The data structure works under the promise that $G \in \mathcal{G}$ at all times.

Proof. Let H be a graph so that all graphs in $\mathcal{G}$ are H -topological-minor-free. We use the data structure of [Theorem 7.5](#) initialized with the parameters H and η . It maintains an annotated normal $(\mathcal{O}_{H,\eta}(\text{tw-mod}_\eta(G)), \mathcal{O}_{H,\eta}(1))$ -protrusion decomposition $\mathcal{T} = (T, \text{bag}, \text{edges})$ of G . Let $\gamma = \mathcal{O}_{H,\eta}(1)$ be a constant that depends on H and η so that the second parameter of the protrusion decomposition $\mathcal{T}$, i.e., the width of the protrusions, is upper bounded by γ . In addition, if r is the root of $\mathcal{T}$, the hypergraph torso(r) is maintained, and after every update operation to G , the changes to torso(r) can be described as a sequence of basic hypergraph operations of size $\mathcal{O}_{H,\eta}(1)$, which is returned. Moreover, the set of all edges that have been removed from edges(r) or inserted into edges(r) is also returned and has size $\mathcal{O}_{H,\eta}(1)$, as well.

Our goal is to maintain the kernel (K, Δ) , where $K = K(G, \mathcal{T}) \in \mathcal{G}$ and $\Delta = \Delta(G, \mathcal{T})$. By [Lemma 8.13](#) it is indeed a kernel, has $\Delta \leq 0$, $K \in \mathcal{G}$, and $|V(K)| \leq \mathcal{O}(\text{tw-mod}_\eta(G))$. Since $K \in \mathcal{G}$, K excludes H as a topological-minor and thus is sparse (see [Lemma 4.9](#)), so also $|K| \leq \mathcal{O}(\text{tw-mod}_\eta(G))$.

We now describe how we maintain $K(G, \mathcal{T})$ and $\Delta(G, \mathcal{T})$. By [Lemma 8.8](#), there exists a tree decomposition automaton $\mathcal{A}$ of width γ that, given a node $x \in V(T) \setminus \{r\}$ determines the pair $(\varphi_{\Pi, \mathcal{G}, [\gamma+1]}(X), \Delta(X, \varphi_{\Pi, \mathcal{G}, [\gamma+1]}(X)))$, where $X = (G_x, \text{bag}(x), \Lambda_{\text{bag}(x)}^*)$ is the boundaried graph that corresponds to x . By [Theorem 7.5](#), we can maintain the run $\rho_{\mathcal{A}}^{\mathcal{T}}(x)$ for every non-root node $x \in V(T)$ at the cost of an additional $\tau(\gamma)$ factor on the running times, where $\tau(\gamma)$ is the evaluation time of $\mathcal{A}$ that depends only on Π , $\mathcal{G}$ and γ . For the sake of convenience, we drop the subscripts in the following and set $\varphi(X) := \varphi_{\Pi, \mathcal{G}, [\gamma+1]}(X)$ and $\Delta(X) := \Delta(X, \varphi_{\Pi, \mathcal{G}, [\gamma+1]}(X))$ for every boundaried graph X .

Then, in particular, for every $c_i \in \text{chd}(r)$, we can in constant time determine the progressive representative $\varphi(G_i)$ together with the transposition constant $\Delta(G_i)$ by querying the run of the automaton $\mathcal{A}$. Then, whenever there is a “change” in the run $\rho_{\mathcal{A}}^{\mathcal{T}}(c_i)$ of $\mathcal{A}$ on a root-child c_i , we update the replacement $\varphi(G_i)$ in K . To keep track of which vertices and edges in K belong to which (boundaried) subgraph G_i , we additionally maintain a balanced binary search tree $\mathcal{S}$ that contains every root-child $c_i \in \text{chd}(r)$ together with a list of all edges in $E(\varphi(G_i))$ and all vertices in $\text{int}(\varphi(G_i))$.

We now describe in detail the operations of our data structure and how the kernel (K, Δ) is maintained. First, let us note that we can implement the initialization operation by first initializing the data structure of [Theorem 7.5](#) with an empty graph, and then inserting edges and vertices one at the time, because if G is H -topological-minor-free, then also every subgraph of G is H -topological-minor-free. We do not need to worry about the fact that intermittently the graph may be outside of $\mathcal{G}$, because we output the kernel only at the end of all of these operations, when $G \in \mathcal{G}$ is guaranteed.

For the other operations (vertex/edge insertion/deletion), we update the protrusion decomposition $\mathcal{T}$ with the respective operation from [Theorem 7.5](#) in $\mathcal{O}_{H,\eta}(\log |G|)$ time. Then, we remove every edge that has been removed from edges(r) from K . Given the set of size $\mathcal{O}_{H,\eta}(1)$ of all such edges, this can be done in $\mathcal{O}_{H,\eta}(1)$ time via $\mathcal{O}_{H,\eta}(1)$ basic graph operations to K . Then, we consider the root-children $c_i \in \text{chd}(r)$. By [Theorem 7.5](#), every change of the run $\rho_{\mathcal{A}}^{\mathcal{T}}(c_i)$ at a root-child c_i corresponds to a change in torso(r). Thus, we do not need to reconsider the

root-children $c_i \in \text{chd}(r)$, where the corresponding hyperedge e_{c_i} is in $\text{torso}(r)$ before and after the update.

Let G and G' denote the graph, $\mathcal{T} = (T, \text{bag}, \text{edges})$ and $\mathcal{T}' = (T', \text{bag}', \text{edges}')$ the protrusion decomposition and R and R' the hypergraph $\text{torso}(r)$ before and after the update, respectively. Then, by [Theorem 7.5](#), the changes from R to R' can be described as a sequence of operations $\mathcal{C}$ of size $\mathcal{O}_{H,\eta}(1)$ which is returned by the data structure. Suppose, this sequence $\mathcal{C}$ produces the following sequence of hypergraphs: $R = R^{(0)}, R^{(1)}, \dots, R^{(\zeta)} = R'$, where $\zeta = |\mathcal{C}| \leq \mathcal{O}_{H,\eta}(1)$. We update the kernel (K, Δ) step-by-step along $\mathcal{C}$ using $\mathcal{O}(1)$ basic graph operations for each operation in $\mathcal{C}$. In the end, these sequences of graph operations are concatenated to obtain the sequence of operations between the graph K before and after the update, which is then returned. In the following, we describe in detail how the kernel is updated for the different types of basic hypergraph operations in $\mathcal{C}$.

First, for every **AddVertex** operation in $\mathcal{C}$, we also add the new vertex v to K . Clearly, this requires only one basic graph operation, namely **AddVertex**(v), and can be done in $\mathcal{O}(1)$ time.

We then consider the **AddHyperedge** operations in $\mathcal{C}$. Due to the minimality of $\mathcal{C}$, if a hyperedge e is added during $\mathcal{C}$, then e cannot be deleted later within $\mathcal{C}$, so $e \in E(R')$. Thus, the inserted hyperedge e corresponds to a root-child $c_i \in V(T') \setminus V(T)$. To update the kernel (K, Δ) we glue the graph $\varphi(G'_i)$ to K and the integer $\Delta(G'_i)$ to Δ . Both $\varphi(G'_i)$ and $\Delta(G'_i)$ can be retrieved from the run of our automaton $\mathcal{A}$ in constant time. To add $\varphi(G'_i)$ to K , we iterate over $\varphi(G'_i)$ and apply the **AddVertex**(v) and **AddEdge**(e) operation for each $v \in \text{int}(\varphi(G'_i))$ and each edge $e \in E(\varphi(G'_i))$ to K . We also insert the same vertices and edges into a list and add this list to $\mathcal{S}$ with key c_i . Since the size of $\varphi(G'_i)$ depends only on $\mathcal{G}$, Π , and $\gamma = \gamma(H, \eta)$, this can be done in time $\mathcal{O}(\log |G|)$ and the created sequence of operations to K is in $\mathcal{O}(1)$, where the hidden factors in the $\mathcal{O}$ -notation depend only on $\mathcal{G}$, Π , and η .

Next, we consider the **DeleteHyperedge** operations in $\mathcal{C}$. Again, due to the minimality of $\mathcal{C}$, if a hyperedge e is deleted during $\mathcal{C}$, then e was not inserted before during the same sequence $\mathcal{C}$, so $e \in E(R)$. Thus, the hyperedge e corresponds to a root-child $c_i \in V(T) \setminus V(T')$. To update (K, Δ) , we remove $\varphi(G_i)$ from K and subtract $\Delta(G_i)$ from Δ . To do this, we need the balanced binary search tree $\mathcal{S}$. We first look up the entry for c_i in $\mathcal{S}$ and obtain a list of all edges in $\varphi(G_i)$ and all vertices in $\text{int}(\varphi(G_i))$. We remove every such vertex and edge from K by using the basic graph operations **DeleteEdge**(e) and **DeleteVertex**(v). Finally, we delete the entry with key c_i from $\mathcal{S}$. Again, the size of $\varphi(G_i)$ depends only on $\mathcal{G}$, Π , and γ . So, this can be done in time $\mathcal{O}(\log |G|)$ and the created sequence of operations to K is in $\mathcal{O}(1)$, where the hidden factors in the $\mathcal{O}$ -notation depend only on $\mathcal{G}$, Π , and η .

Then, for every **DeleteVertex** operation in $\mathcal{C}$, we delete the same vertex v from K , again using only one basic graph operation, namely **DeleteVertex**(v), which takes $\mathcal{O}(1)$ time. To do this, we need to argue that v is an isolated vertex in the current graph K . For this, note that due to the minimality of $\mathcal{C}$, every edge in the current K belongs to at least one $\varphi(G_i)$ or $\varphi(G'_i)$, where $c_i \in V(T) \cup V(T')$. Furthermore, each such $\varphi(G_i)$ or $\varphi(G'_i)$ corresponds to a hyperedge $e_t \in E(R) \cup E(R')$. Now, suppose that the current **DeleteVertex** operation is the j^{th} operation in $\mathcal{C}$, i.e., it transforms $R^{(j-1)}$ into $R^{(j)}$. Then, v is an isolated vertex in $R^{(j-1)}$, i.e., there is no hyperedge $e_t \in E(R^{(j-1)})$ with $v \in V(e_t)$, so every such hyperedge has been deleted in one of the previous steps together with the corresponding subgraph $\varphi(G_i)$ from K . Thus, v is isolated in K and can be removed via the **DeleteVertex** basic graph operation.

Finally, we need to add all the edges that have been added to $\text{edges}(r)$ to K . This can again be done in $\mathcal{O}_{H,\eta}(1)$ time via $\mathcal{O}_{H,\eta}(1)$ basic graph operations given the list of all such edges. $\square$

For linearly treewidth-bounding problems, we observe that we can instead parameterize by $\text{OPT}_{\Pi}(G)$.

Lemma 8.17. *Consider the setting of Lemma 8.16. If Π is linearly treewidth-bounding on $\mathcal{G}$, then the same lemma holds, but without taking η as a parameter and instead guaranteeing that $|K| \leq \mathcal{O}(\text{OPT}_\Pi(G))$.*

Proof. Because Π is linearly treewidth-bounding, there exists a constant η so that $\text{tw-mod}_\eta(G) \leq \mathcal{O}(\text{OPT}_\Pi(G))$. Therefore, by choosing the η in Lemma 8.16 to be this constant, we get that $|K| \leq \mathcal{O}(\text{OPT}_\Pi(G))$. $\square$

We then observe that Lemmas 8.16 and 8.17 form a (more precise) version of Theorem 1.3 (note that Δ should be negated), and thus this completes the proof of Theorem 1.3.

9 Conclusions

We gave a dynamic algorithm for maintaining an approximately optimal protrusion decomposition of a dynamic topological-minor-free graph, and applied it for dynamic kernelization. Let us discuss the applications of our results, future work, and open problems.

Direct applications. We list some concrete problems to which our dynamic kernelization meta-theorem Theorem 1.3 applies directly.

First, let us consider the problems where Theorem 1.3 applies for any topological-minor-free graph class $\mathcal{G}$. In this setting, most of the natural problems that are known to satisfy the required conditions are vertex-deletion problems to graph classes $\mathcal{Q}$, for which it is known that graphs in $\mathcal{G} \cap \mathcal{Q}$ have bounded treewidth. Such problems are trivially linearly treewidth-bounding, so Theorem 1.3 applies whenever they have FII. Examples with FII include VERTEX COVER, FEEDBACK VERTEX SET, more generally TREewidth- η -DELETION for any η , even more generally $\mathcal{F}$ -MINOR-FREE-DELETION for any finite set of connected graphs $\mathcal{F}$ that contains a planar graph, CLUSTER VERTEX DELETION, CHORDAL VERTEX DELETION, INTERVAL VERTEX DELETION, PROPER INTERVAL VERTEX DELETION, COGRAPH VERTEX DELETION, CONNECTED VERTEX COVER, CONNECTED COGRAPH VERTEX DELETION, and CONNECTED CLUSTER VERTEX DELETION [BFL⁺16, KLP⁺16]. Another problem which is not strictly speaking a vertex-deletion problem, but for which Theorem 1.3 applies in this setting is EDGE DOMINATING SET [KLP⁺16].

Let us then consider problems where Theorem 1.3 applies for any minor-free graph class. This of course includes everything mentioned above, but additionally includes “minor-bidimensional” problems that are linearly treewidth-bounding by non-trivial arguments [FLS18, FLST20]. Such problems include the problem $\mathcal{F}$ -MINOR-PACKING for any finite set of connected graphs $\mathcal{F}$ that contains a planar graph, which asks to pack a maximum number of vertex-disjoint minor-models of graphs from $\mathcal{F}$. Concretely, this contains for example the problem CYCLE PACKING.

We then consider applications of Theorem 1.3 to apex-minor-free graph classes, which include planar graphs and graphs of bounded genus. We cover all the problems above, but also “contraction-bidimensional” problems [FLS18, FLST20]. Such problems include DOMINATING SET, INDUCED MATCHING, r -DOMINATING SET for any r , CONNECTED DOMINATING SET, and r -SCATTERED SET for any r [FLST20].

Potential indirect and future applications. In Theorem 1.3, we applied the data structure of Theorem 1.2 to obtain dynamic linear kernels for problems that directly satisfy certain conditions. However, protrusion decompositions have been applied in kernelization even more broadly. As one example, Bodlaender et al. [BFL⁺16] obtain polynomial (but superlinear) kernels on graphs of bounded genus for problems that do not have FII but are instead expressible in CMSO₂ in a certain way, for example, INDEPENDENT DOMINATING SET. We conjecture that our data structure could be modified to obtain dynamic versions of these results, but with additional $\mathcal{O}(\text{poly}(\text{OPT}(G)))$ factors in the update time and the number of changes to the kernel.

Another problem for which [Theorem 1.3](#) does not apply but for which [\[BFL⁺16\]](#) obtain a linear kernel on graphs of bounded genus is TRIANGLE PACKING, or more generally, the problem of packing connected subgraphs from any fixed family of subgraphs. It has FII, but in order to relate it to treewidth, one must first apply a preprocessing routine that deletes all vertices not contained in any triangle. It would be interesting to see if this preprocessing routine could be implemented dynamically, so that [Theorem 1.3](#) could be applied also to this problem.

Besides kernelization, protrusion decompositions have been applied to design parameterized and approximation algorithms, for example, in [\[FLMS12\]](#) (see [Section 1](#) for more references). Such algorithms typically employ complex techniques in addition to protrusion replacement, and therefore it is not clear at all whether the data structure of [Theorem 1.2](#) (or its potential extensions) could help to design dynamic versions of them. As a benchmark problem in this direction, we ask whether FEEDBACK VERTEX SET (on general graphs) admits a dynamic constant-factor approximation algorithm, for example, with sublinear (amortized) update time.

Potential improvements and future work. As already discussed in the introduction, the lower bound of [\[PD06\]](#) implies that the logarithmic update time of the data structure of [Theorem 1.2](#) is tight, and furthermore it is tight even for some cases of [Theorem 1.3](#), such as CYCLE PACKING on planar graphs. It would be interesting to know if the logarithmic update time for dynamic kernelization is tight even for problems with more local flavor, such as DOMINATING SET on planar graphs.

While the logarithmic update time seems natural for problems associated with treewidth-modulators, perhaps it could be improved to constant update time for problems associated with treedepth-modulators, since graphs of bounded treedepth admit dynamic algorithms with constant update time [\[DKT14, CCD⁺21\]](#).

The fact that the update time of [Theorem 1.2](#) is amortized instead of worst-case comes from the techniques of [\[Kor25a\]](#), and we believe that if the result of [\[Kor25a\]](#) would be improved to worst-case instead of amortized, then [Theorem 1.2](#) most likely also could be improved.

The data structure of [Theorem 1.2](#) is restricted to topological-minor-free graph classes. This is a somewhat natural barrier because of the following [Proposition 9.1](#), whose proof is presented in [Section A.3](#). A graph class $\mathcal{G}$ is *subgraph-closed* if for all $G \in \mathcal{G}$, every subgraph of G is also contained in $\mathcal{G}$.

Proposition 9.1. *Let $\mathcal{G}$ be a graph class that is subgraph-closed. If $\mathcal{G}$ does not exclude a topological minor, then there is no function $f(\eta)$ and a constant $c < 2$ so that for every graph $G \in \mathcal{G}$ and every η , G admits an $(f(\eta) \cdot \text{tw-mod}_\eta(G)^c, f(\eta))$ -protrusion decomposition.*

[Proposition 9.1](#) tells that topological-minor-free graph classes are the most general subgraph-closed classes that admit a linear relation between treewidth-modulators and protrusion decompositions. However, one could still imagine a version of [Theorem 1.2](#) that directly maintains an approximately-optimal protrusion decomposition, regardless of the size of the minimum treewidth-modulator. Such a version of [Theorem 1.2](#) could be useful for dynamic versions of the applications of protrusion decompositions beyond topological-minor-free graph classes, e.g. [\[FLMS12\]](#).

Currently, the dynamic kernelization algorithms of [Theorem 1.3](#) provide no method of lifting a solution in the kernel K into a solution in the original graph. It would be interesting if such a method could be implemented, for example, lifting a dominating set of K of size $\text{OPT}(K)$ into a dominating set of G of size $\text{OPT}(G)$ in time $\mathcal{O}(\text{OPT}(G) \cdot \log n)$.

References

- [ACJ⁺24] Shinwoo An, Kyungjin Cho, Leo Jang, Byeonghyeon Jung, Yudam Lee, Eunjin Oh, Donghun Shin, Hyeonjun Shin, and Chanho Song. Dynamic parameterized

- problems on unit disk graphs. In *Proceedings of the 35th International Symposium on Algorithms and Computation, ISAAC 2024*, volume 322 of *LIPIcs*, pages 6:1–6:15. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2024. [3](#), [6](#)
- [AF93] Karl Abrahamson and Michael Fellows. Finite automata, bounded treewidth, and well-quasiordering. *Contemporary Mathematics*, 147:539–563, 1993. [55](#)
- [AFN04] Jochen Alber, Michael R. Fellows, and Rolf Niedermeier. Polynomial-time data reduction for dominating set. *J. ACM*, 51(3):363–384, 2004. [2](#)
- [ALS91] Stefan Arnborg, Jens Lagergren, and Detlef Seese. Easy problems for tree-decomposable graphs. *J. Algorithms*, 12(2):308–340, 1991. [55](#)
- [AMV20] Josh Alman, Matthias Mnich, and Virginia Vassilevska Williams. Dynamic parameterized problems and algorithms. *ACM Trans. Algorithms*, 16(4):45:1–45:46, 2020. [3](#), [6](#)
- [Bak94] Brenda S. Baker. Approximation algorithms for NP-complete problems on planar graphs. *J. ACM*, 41(1):153–180, 1994. [5](#)
- [BDFH09] Hans L. Bodlaender, Rodney G. Downey, Michael R. Fellows, and Danny Hermelin. On problems without polynomial kernels. *J. Comput. Syst. Sci.*, 75(8):423–434, 2009. [1](#)
- [BFL⁺09] Hans L. Bodlaender, Fedor V. Fomin, Daniel Lokshtanov, Eelko Penninkx, Saket Saurabh, and Dimitrios M. Thilikos. (Meta) kernelization. In *Proceedings of the 50th Annual IEEE Symposium on Foundations of Computer Science, FOCS 2009*, pages 629–638. IEEE, 2009. [2](#)
- [BFL⁺16] Hans L. Bodlaender, Fedor V. Fomin, Daniel Lokshtanov, Eelko Penninkx, Saket Saurabh, and Dimitrios M. Thilikos. (Meta) kernelization. *J. ACM*, 63(5):44:1–44:69, 2016. [2](#), [3](#), [5](#), [12](#), [53](#), [55](#), [64](#), [65](#)
- [BHI18] Sayan Bhattacharya, Monika Henzinger, and Giuseppe F. Italiano. Deterministic fully dynamic data structures for vertex cover and matching. *SIAM J. Comput.*, 47(3):859–887, 2018. [5](#)
- [BHRT22] Max Bannach, Zacharias Heinrich, Rüdiger Reischuk, and Till Tantau. Dynamic kernels for hitting sets and set packing. *Algorithmica*, 84(11):3459–3488, 2022. [3](#), [6](#)
- [BK96] Hans L. Bodlaender and Ton Kloks. Efficient and constructive algorithms for the pathwidth and treewidth of graphs. *J. Algorithms*, 21(2):358–402, 1996. [11](#), [48](#), [74](#)
- [BKSW24] Sayan Bhattacharya, Peter Kiss, Thatchaphol Saranurak, and David Wajc. Dynamic matching with better-than-2 approximation in polylogarithmic update time. *J. ACM*, 71(5):33:1–33:32, 2024. [5](#)
- [Bod93] Hans L. Bodlaender. Dynamic algorithms for graphs with treewidth 2. In *Proceedings of the 19th International Workshop on Graph-Theoretic Concepts in Computer Science, WG '93*, volume 790 of *LNCS*, pages 112–124. Springer, 1993. [6](#)
- [BP08] Hans L. Bodlaender and Eelko Penninkx. A linear kernel for planar feedback vertex set. In *Proceedings of the Third International Workshop on Parameterized and Exact Computation, IWPEC 2008*, volume 5018 of *LNCS*, pages 160–171. Springer, 2008. [2](#)

- [BPT92] Richard B. Borie, R. Gary Parker, and Craig A. Tovey. Automatic generation of linear-time algorithms from predicate calculus descriptions of problems on recursively constructed graph families. *Algorithmica*, 7(5&6):555–581, 1992. [55](#)
- [BPT08] Hans L. Bodlaender, Eelko Penninkx, and Richard B. Tan. A linear kernel for the k -disjoint cycle problem on planar graphs. In *Proceedings of the 19th International Symposium on Algorithms and Computation, ISAAC 2008*, volume 5369 of *LNCS*, pages 306–317. Springer, 2008. [2](#)
- [BST23] Julien Baste, Ignasi Sau, and Dimitrios M. Thilikos. Hitting minors on bounded treewidth graphs. IV. An optimal algorithm. *SIAM J. Comput.*, 52(4):865–912, 2023. [3](#)
- [CCD⁺21] Jiehua Chen, Wojciech Czerwinski, Yann Disser, Andreas Emil Feldmann, Danny Hermelin, Wojciech Nadara, Marcin Pilipczuk, Michal Pilipczuk, Manuel Sorge, Bartłomiej Wróblewski, and Anna Zych-Pawlewicz. Efficient fully dynamic elimination forests with applications to detecting long paths and cycles. In *Proceedings of the 2021 ACM-SIAM Symposium on Discrete Algorithms, SODA 2021*, pages 796–809. SIAM, 2021. [6](#), [65](#)
- [CCDF97] Liming Cai, Jianer Chen, Rodney G. Downey, and Michael R. Fellows. Advice classes of parameterized tractability. *Ann. Pure Appl. Log.*, 84(1):119–138, 1997. [1](#)
- [CCHM15] Rajesh Hemant Chitnis, Graham Cormode, Mohammad Taghi Hajiaghayi, and Morteza Monemizadeh. Parameterized streaming: Maximal matching and vertex cover. In *Proceedings of the Twenty-Sixth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2015*, pages 1234–1251. SIAM, 2015. [3](#)
- [CFKX07] Jianer Chen, Henning Fernau, Iyad A. Kanj, and Ge Xia. Parametric duality and kernelization: Lower bounds and upper bounds on kernel size. *SIAM J. Comput.*, 37(4):1077–1106, 2007. [2](#)
- [Cou90] Bruno Courcelle. The monadic second-order logic of graphs. I. Recognizable sets of finite graphs. *Inf. Comput.*, 85(1):12–75, 1990. [20](#), [55](#)
- [Cou92] Bruno Courcelle. The monadic second-order logic of graphs III: Tree-decompositions, minor and complexity issues. *RAIRO Theor. Informatics Appl.*, 26:257–286, 1992. [20](#)
- [Cou97] Bruno Courcelle. The expression of graph properties and graph transformations in monadic second-order logic. In Grzegorz Rozenberg, editor, *Handbook of Graph Grammars and Computing by Graph Transformations, Volume 1: Foundations*, pages 313–400. World Scientific, 1997. [20](#)
- [CRST18] Dimitris Chatzidimitriou, Jean-Florent Raymond, Ignasi Sau, and Dimitrios M. Thilikos. An $O(\log OPT)$ -approximation for covering and packing minor models of θ_r . *Algorithmica*, 80(4):1330–1356, 2018. [3](#)
- [DF95] Rodney G. Downey and Michael R. Fellows. Fixed-parameter tractability and completeness I: Basic results. *SIAM J. Comput.*, 24(4):873–921, 1995. [2](#)
- [DF99] R. G. Downey and M. R. Fellows. *Parameterized Complexity*. Monographs in Computer Science. Springer, New York, NY, 1999. [1](#), [55](#)

- [DGvtH⁺17] Konrad Kazimierz Dabrowski, Petr A. Golovach, Pim van 't Hof, Daniël Paulusma, and Dimitrios M. Thilikos. Editing to a planar graph of given degrees. *J. Comput. Syst. Sci.*, 85:168–182, 2017. [3](#)
- [DKT14] Zdenek Dvorák, Martin Kupec, and Vojtech Tuma. A dynamic data structure for MSO properties in graphs with bounded tree-depth. In *Proceedings of the 22nd Annual European Symposium on Algorithms, ESA 2014*, volume 8737 of *LNCS*, pages 334–345. Springer, 2014. [6](#), [65](#)
- [DT13] Zdenek Dvorák and Vojtech Tuma. A dynamic data structure for counting subgraphs in sparse graphs. In *Proceedings of the 13th International Symposium on Algorithms and Data Structures, WADS 2013*, volume 8037 of *LNCS*, pages 304–315. Springer, 2013. [6](#)
- [DvM14] Holger Dell and Dieter van Melkebeek. Satisfiability allows no nontrivial sparsification unless the polynomial-time hierarchy collapses. *J. ACM*, 61(4):23:1–23:27, 2014. [1](#)
- [Epp00] David Eppstein. Diameter and treewidth in minor-closed graph families. *Algorithmica*, 27(3):275–291, 2000. [13](#)
- [FLM⁺15] Fedor V. Fomin, Daniel Lokshtanov, Neeldhara Misra, M. S. Ramanujan, and Saket Saurabh. Solving d -sat via backdoors to small treewidth. In *Proceedings of the Twenty-Sixth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2015*, pages 630–641. SIAM, 2015. [3](#), [4](#)
- [FLM⁺16] Fedor V. Fomin, Daniel Lokshtanov, Neeldhara Misra, Geevarghese Philip, and Saket Saurabh. Hitting forbidden minors: Approximation and kernelization. *SIAM J. Discret. Math.*, 30(1):383–410, 2016. [3](#)
- [FLMS12] Fedor V. Fomin, Daniel Lokshtanov, Neeldhara Misra, and Saket Saurabh. Planar F -deletion: Approximation, kernelization and optimal FPT algorithms. In *Proceedings of the 53rd Annual IEEE Symposium on Foundations of Computer Science, FOCS 2012*, pages 470–479. IEEE, 2012. [3](#), [65](#)
- [FLS18] Fedor V. Fomin, Daniel Lokshtanov, and Saket Saurabh. Excluded grid minors and efficient polynomial-time approximation schemes. *J. ACM*, 65(2):10:1–10:44, 2018. [64](#)
- [FLST12] Fedor V. Fomin, Daniel Lokshtanov, Saket Saurabh, and Dimitrios M. Thilikos. Linear kernels for (connected) dominating set on H -minor-free graphs. In *Proceedings of the Twenty-Third Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2012*, pages 82–93. SIAM, 2012. [3](#)
- [FLST18] Fedor V. Fomin, Daniel Lokshtanov, Saket Saurabh, and Dimitrios M. Thilikos. Kernels for (connected) dominating set on graphs with excluded topological minors. *ACM Trans. Algorithms*, 14(1):6:1–6:31, 2018. [3](#)
- [FLST20] Fedor V. Fomin, Daniel Lokshtanov, Saket Saurabh, and Dimitrios M. Thilikos. Bidimensionality and kernels. *SIAM J. Comput.*, 49(6):1397–1422, 2020. [2](#), [3](#), [5](#), [22](#), [64](#)
- [FLSZ19] Fedor V. Fomin, Daniel Lokshtanov, Saket Saurabh, and Meirav Zehavi. *Kernelization: Theory of Parameterized Preprocessing*. Cambridge University Press, 2019. [1](#), [2](#), [3](#), [4](#), [19](#), [20](#), [55](#), [57](#)

- [FS11] Lance Fortnow and Rahul Santhanam. Infeasibility of instance compression and succinct PCPs for NP. *J. Comput. Syst. Sci.*, 77(1):91–106, 2011. [1](#)
- [FT04] Fedor V. Fomin and Dimitrios M. Thilikos. Fast parameterized algorithms for graphs on surfaces: Linear kernel and exponential speed-up. In *Proceedings of 31st International Colloquium on Automata, Languages and Programming, ICALP 2004*, volume 3142 of *LNCS*, pages 581–592. Springer, 2004. [2](#)
- [GHO⁺17] Jakub Gajarský, Petr Hlinený, Jan Obdržálek, Sebastian Ordyniak, Felix Reidl, Peter Rossmanith, Fernando Sánchez Villamil, and Somnath Sikdar. Kernelization using structural parameters on sparse graph classes. *J. Comput. Syst. Sci.*, 84:219–242, 2017. [3](#)
- [GN07] Jiong Guo and Rolf Niedermeier. Linear problem kernels for NP-hard problems on planar graphs. In *Proceedings of the 34th International Colloquium on Automata, Languages and Programming, ICALP 2007*, volume 4596 of *LNCS*, pages 375–386. Springer, 2007. [2](#)
- [GNW06] Jiong Guo, Rolf Niedermeier, and Sebastian Wernicke. Fixed-parameter tractability results for full-degree spanning tree and its dual. In *Proceedings of the Second International Workshop on Parameterized and Exact Computation, IWPEC 2006*, volume 4169 of *LNCS*, pages 203–214. Springer, 2006. [2](#)
- [GPR⁺21] Archontia C. Giannopoulou, Michal Pilipczuk, Jean-Florent Raymond, Dimitrios M. Thilikos, and Marcin Wrochna. Linear kernels for edge deletion problems to immersion-closed graph classes. *SIAM J. Discret. Math.*, 35(1):105–151, 2021. [3](#)
- [GPST15] Valentin Garnero, Christophe Paul, Ignasi Sau, and Dimitrios M. Thilikos. Explicit linear kernels via dynamic programming. *SIAM J. Discret. Math.*, 29(4):1864–1894, 2015. [5](#)
- [GPST19] Valentin Garnero, Christophe Paul, Ignasi Sau, and Dimitrios M. Thilikos. Explicit linear kernels for packing problems. *Algorithmica*, 81(4):1615–1656, 2019. [5](#)
- [GRZ25] Harmender Gahlawat, Abhishek Rathod, and Meirav Zehavi. (Almost-)optimal FPT algorithm and kernel for T-cycle on planar graphs. In *Proceedings of the 52nd International Colloquium on Automata, Languages, and Programming, ICALP 2025*, volume 334 of *LIPICs*, pages 82:1–82:18. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2025. [3](#)
- [GST23] Petr A. Golovach, Giannos Stamoulis, and Dimitrios M. Thilikos. Hitting topological minor models in planar graphs is fixed parameter tractable. *ACM Trans. Algorithms*, 19(3):23:1–23:29, 2023. [3](#)
- [IO14] Yoichi Iwata and Keigo Oka. Fast dynamic graph algorithms for parameterized problems. In *Proceedings of the 14th Scandinavian Symposium and Workshops on Algorithm Theory, SWAT 2014*, volume 8503 of *LNCS*, pages 241–252. Springer, 2014. [3](#), [6](#)
- [JPS⁺14] Gwénaél Joret, Christophe Paul, Ignasi Sau, Saket Saurabh, and Stéphan Thomassé. Hitting and harvesting pumpkins. *SIAM Journal on Discrete Mathematics*, 28(3):1363–1390, 2014. [3](#)

- [JW25] Bart M. P. Jansen and Michal Włodarczyk. Lossy planarization: A constant-factor approximate kernelization for planar vertex deletion. *SIAM J. Comput.*, 54(1):1–91, 2025. [3](#)
- [KLP⁺16] Eun Jung Kim, Alexander Langer, Christophe Paul, Felix Reidl, Peter Rossmanith, Ignasi Sau, and Somnath Sikdar. Linear kernels and single-exponential algorithms via protrusion decompositions. *ACM Trans. Algorithms*, 12(2):21:1–21:41, 2016. [3](#), [5](#), [10](#), [11](#), [22](#), [64](#)
- [KMN⁺23] Tuukka Korhonen, Konrad Majewski, Wojciech Nadara, Michal Pilipczuk, and Marek Sokolowski. Dynamic treewidth. In *Proceedings of the 64th IEEE Annual Symposium on Foundations of Computer Science, FOCS 2023*, pages 1734–1744. IEEE, 2023. [6](#), [48](#), [73](#), [74](#)
- [KNPS24] Tuukka Korhonen, Wojciech Nadara, Michal Pilipczuk, and Marek Sokolowski. Fully dynamic approximation schemes on planar and apex-minor-free graphs. In *Proceedings of the 2024 ACM-SIAM Symposium on Discrete Algorithms, SODA 2024*, pages 296–313. SIAM, 2024. [5](#)
- [Kor25a] Tuukka Korhonen. Dynamic treewidth in logarithmic time. *CoRR*, abs/2504.02790, 2025. To appear in FOCS 2025. [4](#), [5](#), [6](#), [7](#), [8](#), [9](#), [10](#), [11](#), [13](#), [17](#), [37](#), [38](#), [41](#), [42](#), [43](#), [50](#), [65](#), [71](#), [73](#)
- [Kor25b] Tuukka Korhonen. Linear-time algorithms for k-edge-connected components, k-lean tree decompositions, and more. In *Proceedings of the 57th Annual ACM Symposium on Theory of Computing, STOC 2025*, pages 111–119. ACM, 2025. [7](#), [10](#), [14](#), [31](#), [50](#)
- [KPP15] Eun Jung Kim, Christophe Paul, and Geevarghese Philip. A single-exponential FPT algorithm for the K_4 -minor cover problem. *J. Comput. Syst. Sci.*, 81(1):186–207, 2015. [3](#)
- [KPSX11] Iyad A. Kanj, Michael J. Pelsmajer, Marcus Schaefer, and Ge Xia. On the induced matching problem. *J. Comput. Syst. Sci.*, 77(6):1058–1070, 2011. [2](#)
- [KST18] Eun Jung Kim, Maria J. Serna, and Dimitrios M. Thilikos. Data-compression for parametrized counting problems on sparse graphs. In *Proceedings of the 29th International Symposium on Algorithms and Computation, ISAAC 2018*, volume 123 of *LIPIcs*, pages 20:1–20:13. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2018. [3](#), [4](#), [12](#), [22](#)
- [LMS11] Daniel Lokshtanov, Matthias Mnich, and Saket Saurabh. A linear kernel for planar connected dominating set. *Theor. Comput. Sci.*, 412(23):2536–2543, 2011. [2](#)
- [LSS⁺25] Daniel Lokshtanov, Ramanujan Maadapuzhi Sridharan, Saket Saurabh, Roohani Sharma, and Meirav Zehavi. Wannabe bounded treewidth graphs admit a polynomial kernel for directed feedback vertex set. *ACM Trans. Comput. Theory*, 17(1):2:1–2:28, 2025. [3](#)
- [Mad67] W. Mader. Homomorphieeigenschaften und mittlere kantendichte von graphen. *Mathematische Annalen*, 174(4):265–268, 1967. [28](#)
- [MPS25] Konrad Majewski, Michal Pilipczuk, and Marek Sokolowski. Maintaining cms_2 properties on dynamic structures with bounded feedback vertex number. *ACM Trans. Comput. Theory*, 17(2):8:1–8:72, 2025. [6](#)

- [MPZ24] Konrad Majewski, Michal Pilipczuk, and Anna Zych-Pawlewicz. Parameterized dynamic data structure for split completion. In *Proceedings of the 32nd Annual European Symposium on Algorithms, ESA 2024*, volume 308 of *LIPIcs*, pages 87:1–87:17. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2024. 6
- [MS07] Hannes Moser and Somnath Sikdar. The parameterized complexity of the induced matching problem in planar graphs. In *Proceedings of the First Annual International Workshop on Frontiers in Algorithmics, FAW 2007*, volume 4613 of *LNCS*, pages 325–336. Springer, 2007. 2
- [OPR⁺23] Jędrzej Olkowski, Michal Pilipczuk, Mateusz Rychlicki, Karol Węgrzycki, and Anna Zych-Pawlewicz. Dynamic data structures for parameterized string problems. In *Proceedings of the 40th International Symposium on Theoretical Aspects of Computer Science, STACS 2023*, volume 254 of *LIPIcs*, pages 50:1–50:22. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2023. 6
- [PD06] Mihai Pătraşcu and Erik D. Demaine. Logarithmic lower bounds in the cell-probe model. *SIAM J. Comput.*, 35(4):932–963, 2006. 4, 5, 65
- [RS91] Neil Robertson and Paul D. Seymour. Graph minors. X. Obstructions to tree-decomposition. *J. Comb. Theory B*, 52(2):153–190, 1991. 7, 14
- [RS95] Neil Robertson and Paul D. Seymour. Graph minors. XIII. The disjoint paths problem. *J. Comb. Theory B*, 63(1):65–110, 1995. 16
- [Sol16] Shay Solomon. Fully dynamic maximal matching in constant update time. In *Proceedings of the 57th Annual Symposium on Foundations of Computer Science, FOCS 2016*, pages 325–334. IEEE, 2016. 5
- [Tho10] Stéphan Thomassé. A $4k^2$ kernel for feedback vertex set. *ACM Trans. Algorithms*, 6(2):32:1–32:8, 2010. 1

A Missing proofs

A.1 Proofs missing from Section 3

We prove Lemma 3.2 using the following lemma from [Kor25a]. We denote by $e_{\perp}$ the hyperedge with $V(e_{\perp}) = \emptyset$. For a hypergraph G with a hyperedge $e \in E(G)$, an e -rooted superbranch decomposition of G , where the root is the leaf $\mathcal{L}^{-1}(e)$.

Lemma A.1 ([Kor25a, Lemma 4.3]). *Let G be a hypergraph, $e_{\perp} \in E(G)$, and $\mathcal{T} = (T, \mathcal{L})$ an $e_{\perp}$ -rooted superbranch decomposition of G . Let also $t \in V_{\text{int}}(T)$ be a node with parent p , so that $\mathcal{L}[c]$ is well-linked for every child c of t . Let $e_p \in E(\text{torso}(t))$ be the hyperedge of $\text{torso}(t)$ that corresponds to p . Then, a set $A \subseteq E(\text{torso}(t)) \setminus \{e_p\}$ is well-linked in $\text{torso}(t)$ if and only if $A \triangleright \mathcal{T}$ is well-linked in G .*

Now, Lemma 3.2 is an easy generalization of Lemma A.1.

Lemma 3.2. *Let G be a hypergraph and $\mathcal{T} = (T, \mathcal{L})$ a rooted superbranch decomposition of G with root r . Let also $t \in V_{\text{int}}(T)$ be a node, so that $\mathcal{L}[c]$ is well-linked for every child c of t . Let $C \subseteq E(\text{torso}(t))$ be the set of hyperedges corresponding to the children of t . Then, a set $A \subseteq C$ is well-linked in $\text{torso}(t)$ if and only if $A \triangleright \mathcal{T}$ is well-linked in G .*

Proof. Let $G' = G \cup \{e_\perp\}$ and let $\mathcal{T}' = (T', \mathcal{L}')$ be the rooted superbranch decomposition of G' obtained from $\mathcal{T}$ by adding a node r' to T as a leaf-child of r , and setting $\mathcal{L}'(r') = e_\perp$. We first note that a set $B \subseteq E(G)$ is well-linked in G if and only if it is well-linked in G' . For a node $t \in V_{\text{int}}(T')$, we denote by $\text{torso}'(t)$ the torso of t in $\mathcal{T}'$. We then further observe that for every node $t \in V_{\text{int}}(T') \setminus \{r\} = V_{\text{int}}(T) \setminus \{r\}$, $\text{torso}(t) = \text{torso}'(t)$, while $\text{torso}'(r) = \text{torso}(r) \cup e_\perp$. Therefore, for every node $t \in V_{\text{int}}(T)$, a set $A \subseteq E(\text{torso}(t))$ is well-linked in $\text{torso}(t)$ if and only if it is well-linked in $\text{torso}'(t)$. Applying Lemma A.1, the lemma follows. $\square$

A.2 Proofs missing from Section 8

Lemma 8.6. *Let $\mathcal{G}$ be a graph class and let Π be a parameterized graph problem. Then, each equivalence class of $\equiv_{\Pi, \mathcal{G}}$ has a progressive representative.*

Proof. Let $\mathcal{C}$ be an equivalence class of $\equiv_{\Pi, \mathcal{G}}$. Without loss of generality (by flipping the $\in \Pi$ relation if needed), we assume that $(G, k) \notin \Pi$ for every graph G and $k < 0$. If $\mathcal{C}$ is a class of monotone graphs, i.e., for every boundaried graph $G \in \mathcal{C}$, every $F \in \mathcal{F}$, and every $k \in \mathbb{Z}$, $(F \oplus_u G, k) \notin \Pi$, then $\Delta_\Pi(G_1, G_2) = 0$ for any two graphs $G_1, G_2 \in \mathcal{C}$, and thus every graph in $\mathcal{C}$ is a progressive representative.

Otherwise, there exist $G_0 \in \mathcal{C}$, $F_0 \in \mathcal{F}$, and a (non-negative) integer k_0 such that $(F_0 \oplus_u G_0, k_0) \in \Pi$. Among all those triples, we choose (G_0, F_0, k_0) such that k_0 is minimized. We now show that G_0 is a progressive representative of $\mathcal{C}$. For this, let $G \in \mathcal{C}$ and suppose that $\Delta_\Pi(G_0, G) < 0$. For every $(F, k) \in \mathcal{F} \times \mathbb{Z}$, we have $(F \oplus_u G_0, k) \in \Pi$ if and only if $(F \oplus_u G, k + \Delta_\Pi(G_0, G)) \in \Pi$. Thus, since $(F_0 \oplus_u G_0, k_0) \in \Pi$, we have $(F_0 \oplus_u G, k_0 + \Delta_\Pi(G_0, G)) \in \Pi$, which is a contradiction to the minimality of (G_0, F_0, k_0) . It follows that $\Delta_\Pi(G_0, G) \geq 0$, and since G was chosen arbitrarily from $\mathcal{C}$, G_0 is a progressive representative of $\mathcal{C}$. $\square$

A.3 Proofs missing from Section 9

We re-state Proposition 9.1 and prove it.

Proposition 9.1. *Let $\mathcal{G}$ be a graph class that is subgraph-closed. If $\mathcal{G}$ does not exclude a topological minor, then there is no function $f(\eta)$ and a constant $c < 2$ so that for every graph $G \in \mathcal{G}$ and every η , G admits an $(f(\eta) \cdot \text{tw-mod}_\eta(G)^c, f(\eta))$ -protrusion decomposition.*

Proof. Let $\mathcal{G}$ be a graph class that is subgraph-closed and does not exclude a topological minor. For integers $r, t \geq 1$, a graph is an $(\geq r)$ -subdivided t -clique if it is obtained from the t -clique by replacing each edge by a path of $\geq r$ edges. We call the t “non-subdivision” vertices of such a graph the *junction vertices*. We observe that because graphs in $\mathcal{G}$ contain all graphs as topological minors, they particularly contain an (≥ 2) -subdivided t -clique as a topological minor for every $t \geq 1$, which by subgraph-closedness implies that $\mathcal{G}$ contains an (≥ 2) -subdivided t -clique for every $t \geq 1$.

For the sake of contradiction, suppose that there is a function $f(\eta)$ and a constant $c < 2$, so that for every graph $G \in \mathcal{G}$ and every η , G admits an $(f(\eta) \cdot \text{tw-mod}_\eta(G)^c, f(\eta))$ -protrusion decomposition. Let $G \in \mathcal{G}$ be a (≥ 2) -subdivided t -clique for a large enough t that we will choose later. By the assumption, it admits an $(f(1) \cdot t^c, f(1))$ -protrusion-decomposition (T, bag) , rooted at a node $r \in V(T)$. For a child c of r , denote by $\mathcal{T}_c = (T_c, \text{bag}_c)$ the tree decomposition rooted at c , and by G_c the subgraph of G induced by the bags of $\mathcal{T}_c$, and note that $\text{tw}(G_c) \leq f(1) - 1$.

A *subdivision path* of G is a maximal path in G containing no junction vertices. In particular, G has exactly $\binom{t}{2}$ subdivision paths, which are pairwise disjoint.

Claim A.2. *For each c , G_c contains vertices from at most $\binom{4 \cdot f(1)}{2} + 2 \cdot f(1)$ subdivision paths.*

Proof of the claim. Let P be a subdivision path of G . We say that G_c *partially contains* P if some vertices of P are in G_c and some are not, and *fully contains* P if all vertices of P are in G_c . We observe that if G_c partially contains P , then at least one vertex of P must be in $\mathbf{bag}(c)$, and therefore G_c can partially contain at most $f(1)$ subdivision paths.

Suppose that G_c fully contains more than $\binom{4 \cdot f(1)}{2} + f(1)$ subdivision paths. It follows that $G_c \setminus \mathbf{bag}(c)$ fully contains more than $\binom{4 \cdot f(1)}{2}$ subdivision paths, which implies that G_c contains more than $4 \cdot f(1)$ junction vertices, which implies that $G_c \setminus \mathbf{bag}(c)$ contains more than $3 \cdot f(1)$ junction vertices. Denote these junction vertices by J .

Each subdivision path between a pair of junction vertices in J must be partially or fully contained in G_c , so there is a subset $J' \subseteq J$ of size $|J'| > f(1)$ so that all subdivision paths between vertices in J' are fully contained in G_c . However, then G_c contains an $(f(1) + 1)$ -clique as a (topological) minor, implying that $\text{tw}(G_c) \geq f(1)$, which is a contradiction. $\triangleleft$

Now, the graphs G_c over all c can contain vertices from at most $f(1) \cdot t^c \cdot \left(\binom{4 \cdot f(1)}{2} + 2 \cdot f(1) \right)$ subdivision paths, and the root-bag $\mathbf{bag}(r)$ can contain vertices from at most $f(1) \cdot t^c$ subdivision paths. By choosing t large enough depending on $f(1)$ and c , we get that

$$\binom{t}{2} > f(1) \cdot t^c \cdot \left(\binom{4 \cdot f(1)}{2} + 2 \cdot f(1) + 1 \right),$$

so there would be a subdivision path that appears nowhere in the protrusion decomposition $(T, \mathbf{bag})$, which is a contradiction. $\square$

B Tree decomposition automata

In this section, following [KM^N+23, Kor25a], we formally define tree decomposition automata. We assume that the vertices of all graphs that we process come from a countable, totally ordered universe Ω (e.g., $\Omega = \mathbb{N}$)

Definition B.1 ([KM^N+23]). A *(deterministic) tree decomposition automaton of width ℓ* is a tuple $\mathcal{A} = (Q, F, \iota, \delta)$, where

- Q is a (possibly infinite) set of *states*, which we assume to contain the “null state” $\perp$,
- $F \subseteq Q$ is a set of *accepting states*,
- ι is an *initial mapping* that assigns to each boundaried graph G with at most $\ell + 1$ vertices a state $\iota(G) \in Q$,
- $\delta: 2^\Omega \times 2^\Omega \times 2^\Omega \times 2^{\binom{\Omega}{2}} \times Q \times Q \rightarrow Q$ is a *transition mapping* that describes the transitions.

Definition B.2 ([KM^N+23]). Let $\mathcal{A} = (Q, F, \iota, \delta)$ be a tree decomposition automaton and $\mathcal{T} = (T, \mathbf{bag}, \text{edges})$ be an annotated tree decomposition of a graph G . The *run* $\rho_{\mathcal{A}}^{\mathcal{T}}: V(T) \rightarrow Q$ of $\mathcal{A}$ on $\mathcal{T}$ is the unique labeling satisfying the following:

- For every leaf l of T : $\rho_{\mathcal{A}}^{\mathcal{T}}(l) = \iota(G_l)$.
- For every non-leaf node $x \in V(T)$ with one child y :

$$\rho_{\mathcal{A}}^{\mathcal{T}}(x) = \delta(\mathbf{bag}(x), \mathbf{bag}(y), \emptyset, \text{edges}(x), \rho_{\mathcal{A}}^{\mathcal{T}}(y), \perp).$$

- For every non-leaf node $x \in V(T)$ with two children y and z :

$$\rho_{\mathcal{A}}^{\mathcal{T}}(x) = \delta(\mathbf{bag}(x), \mathbf{bag}(y), \mathbf{bag}(z), \text{edges}(x), \rho_{\mathcal{A}}^{\mathcal{T}}(y), \rho_{\mathcal{A}}^{\mathcal{T}}(z)).$$

A tree decomposition automaton $\mathcal{A}$ *accepts* a tree decomposition $(T, \text{bag}, \text{edges})$ with root r if $\rho_{\mathcal{A}}^{\mathcal{T}}(r) \in F$.

Observation B.3. *Let $\mathcal{A}$ be a tree decomposition automaton and $\mathcal{T} = (T, \text{bag}, \text{edges})$ be an annotated tree decomposition of a graph G . For a given node $t \in V(T)$ with parent p , the run $\rho_{\mathcal{A}}^{\mathcal{T}}(t)$ depends only on $\mathcal{T} \upharpoonright_{V(T_t)}$ and $\text{adh}(tp)$.*

We say that an automaton has *evaluation time* τ if the functions ι and δ can be evaluated in time τ , and, given a state $q \in Q$, it can be decided whether $q \in F$ in time τ as well.

In [KMN⁺23], a treewidth automaton was introduced that, given a tree decomposition of width ℓ of a graph G together with an integer $k \leq \ell$, decides whether $\text{tw}(G) \leq k$. The algorithm is based on the linear-time dynamic programming algorithm by Bodlaender and Kloks [BK96], and thus called *Bodlaender-Kloks automaton*.

Lemma B.4 (Lemma A.3 in [KMN⁺23]). *For every pair of integers $k \leq \ell$, there is a tree decomposition automaton $\mathcal{BK}_{k,\ell}$ of width ℓ with the following property: For any graph G and its annotated tree decomposition $(T, \text{bag}, \text{edges})$ of width at most ℓ , $\mathcal{BK}_{k,\ell}$ accepts $(T, \text{bag}, \text{edges})$ if and only if the treewidth of G is at most k . The state space of $\mathcal{BK}_{k,\ell}$ is of size $\mathcal{O}_{k,\ell}(1)$ and can be computed in time $\mathcal{O}_{k,\ell}(1)$. The evaluation time of $\mathcal{BK}_{k,\ell}$ is $\mathcal{O}_{k,\ell}(1)$ as well.*

We now use this Bodlaender-Kloks automaton to define an internal treewidth automaton, that is, an automaton that, given a boundaried tree decomposition of width ℓ of a boundaried graph G and an integer k , decides whether $\text{itw}(G) \leq k$. Recall that $\text{itw}(G) = \text{tw}(G \setminus \text{bd}(G))$. The high-level idea of our automaton is to keep a run of the Bodlaender-Kloks automaton for $G \setminus B$ for every set $B \subseteq V(G)$ that could be the boundary of G .

Lemma 7.1. *For every pair of integers $k \leq \ell$, there is a tree decomposition automaton $\mathcal{IBK}_{k,\ell}$ of width ℓ with the following property: For any boundaried graph G and its annotated boundaried tree decomposition $\mathcal{T} = (T, \text{bag}, \text{edges})$ of width at most ℓ , $\mathcal{IBK}_{k,\ell}$ accepts $\mathcal{T}$ if and only if the internal treewidth $\text{itw}(G)$ of G is at most k . The state space of $\mathcal{IBK}_{k,\ell}$ is of size $\mathcal{O}_{k,\ell}(1)$ and can be computed in time $\mathcal{O}_{k,\ell}(1)$. The evaluation time of $\mathcal{IBK}_{k,\ell}$ is $\mathcal{O}_{k,\ell}(1)$ as well.*

Proof. Let $\mathcal{BK}_{k,\ell} = (Q, F, \iota, \delta)$ be the automaton from Lemma B.4. In the following, we define a tree decomposition automaton $\mathcal{IBK}_{k,\ell}$ that, given a graph G with its annotated tree decomposition $\mathcal{T} = (T, \text{bag}, \text{edges})$ of width at most ℓ , decides whether $\text{itw}(G) \leq k$.

For this, we introduce the following notation. Let G be a graph and $\mathcal{T} = (T, \text{bag}, \text{edges})$ be its annotated tree decomposition. For a set $S \subseteq V(G)$, we denote by $E_S := \{uv \in E(G) \mid u \in S\}$ the set of edges with at least one endpoint in S . Further, given a set $S \subseteq V(G)$, we define the annotated tree decomposition $\mathcal{T}_S = (T, \text{bag}_S, \text{edges}_S)$ with $\text{bag}_S(x) = \text{bag}(x) \setminus S$ and $\text{edges}_S(x) = \text{edges}(x) \setminus E_S$ for every node $x \in V(T)$. Note that $\mathcal{T}_S$ is an annotated tree decomposition of the graph $G - S$.

Our goal is to define the automaton $\mathcal{IBK}_{k,\ell} = (Q^{2^{\ell+1}}, Q^{2^{\ell+1}-1} \times F, \iota^*, \delta^*)$ in such a way that for any node $x \in V(T)$, the run $\rho_{\mathcal{IBK}_{k,\ell}}^{\mathcal{T}}(x)$ is a list that contains for every set $S \subseteq \text{bag}(x)$ the run $\rho_{\mathcal{BK}_{k,\ell}}^{\mathcal{T}_S}(x)$ of the automaton $\mathcal{BK}_{k,\ell}$ on the tree decomposition $\mathcal{T}_S$ for the same node x . Specifically, when defining $\mathcal{IBK}_{k,\ell}$, we want to keep the following invariant for every graph G and every annotated tree decomposition $\mathcal{T} = (T, \text{bag}, \text{edges})$ of G .

Invariant B.5. *For every node $x \in V(T)$, $\rho_{\mathcal{IBK}_{k,\ell}}^{\mathcal{T}}(x)$ is a list that contains for every set $S \subseteq \text{bag}(x)$, the run $\rho_{\mathcal{BK}_{k,\ell}}^{\mathcal{T}_S}(x)$.*

So, formally, a state of $\mathcal{IBK}_{k,\ell}$ is a list of (at most) $2^{\ell+1}$ states of $\mathcal{BK}_{k,\ell}$, each corresponding to a set $S \subseteq \Omega$ of size at most $\ell + 1$. Given a node $x \in V(T)$, the internal treewidth of the boundaried graph G_x , i.e., the treewidth of $G_x \setminus \text{bag}(x)$, is at most k if and only if $\rho_{\mathcal{BK}_{k,\ell}}^{\mathcal{T}_{\text{bag}(x)}}(x) \in F$,

so our automaton $\mathcal{IBK}_{k,\ell}$ accepts exactly in this case. Technically, we realize this by defining an order $\prec$ on the sets $S \subseteq \Omega$ so that $|S_1| \leq |S_2|$ implies $S_1 \prec S_2$ for all sets $S_1, S_2 \in \Omega$, and sorting our “state-lists” accordingly (and filling up with $\perp$ -entries from the left). Then, for every node $x \in V(T)$, the run $\rho_{\mathcal{IBK}_{k,\ell}}^{\mathcal{T}_{\text{bag}(x)}}(x)$ corresponding to the set $S = \text{bag}(x)$ will be the last entry in the list $\rho_{\mathcal{IBK}_{k,\ell}}^{\mathcal{T}}(x)$, so taking $Q^{2^{\ell+1}-1} \times F$ as the set of accepting states of $\mathcal{IBK}_{k,\ell}$ does the trick.

Now, let us define the initial and transition mapping ι^* and δ^* of $\mathcal{IBK}_{k,\ell}$. We start with the initial mapping ι^* . For a graph G with at most $\ell + 1$ vertices, we define $\iota^*(G)$ to be a list containing $\iota(G - S)$ for every set $S \subseteq V(G)$ (ordered with respect to $\prec$).

Claim B.6. *Every leaf node $x \in V(T)$ satisfies [Invariant B.5](#).*

Proof of the claim. Let $x \in V(T)$ be a leaf node. Then, $\rho_{\mathcal{IBK}_{k,\ell}}^{\mathcal{T}}(x) = \iota^*(G_x)$. By the definition of ι^* , $\iota^*(G_x)$ is a list containing for every subset $S \subseteq V(G_x) = \text{bag}(x)$ the state $\iota(G_x - S) \in Q$. Since x is also a leaf node of $\mathcal{T}_S$ and by [Definition B.2](#), we have $\iota(G_x - S) = \rho_{\mathcal{BK}_{k,\ell}}^{\mathcal{T}_S}(x)$, so [Invariant B.5](#) is satisfied. $\triangleleft$

Next, we define the transition mapping δ^* . Let (X, Y, Z, J, q', q'') be a 6-tuple, where X, Y , and Z are sets of size at most $\ell + 1$, $J \subseteq \binom{\Omega}{2} \setminus \binom{X}{2}$, and $q', q'' \in Q^{2^{\ell+1}}$. Let further q' (and q'' if $q'' \neq \perp$) be a list that contains for every set $S' \subseteq Y$ (or $S' \subseteq Z$) a state $q'_{S'} \in Q$ (or $q''_{S'} \in Q$). Then, we define $\delta^*(X, Y, Z, J, q', q'')$ to be a list that contains for every set $S \subseteq B$ the state $\delta(X \setminus S, Y \setminus S, Z \setminus S, J \setminus E_S, q'_{S \cap Y}, q''_{S \cap Z})$. We will later discuss how we can determine this list within our running time. But first, let us prove that this definition satisfies [Invariant B.5](#).

Claim B.7. *Every node $x \in V(T)$ satisfies [Invariant B.5](#).*

Proof of the claim. We show this claim by induction on the nodes of T . By [Claim B.6](#), every leaf node of T satisfies [Invariant B.5](#). So, let $x \in V(T)$ be a non-leaf node. We distinguish the two cases whether x has one or two children.

First, assume that x has only one child y . By [Definition B.2](#), we have

$$\rho_{\mathcal{IBK}_{k,\ell}}^{\mathcal{T}}(x) = \delta^*(\text{bag}(x), \text{bag}(y), \emptyset, \text{edges}(x), \rho_{\mathcal{IBK}_{k,\ell}}^{\mathcal{T}}(y), \perp).$$

By induction, [Invariant B.5](#) is satisfied for y , i.e., $\rho_{\mathcal{IBK}_{k,\ell}}^{\mathcal{T}}(y)$ is a list that contains for every $S \subseteq \text{bag}(y)$, the state $\rho_{\mathcal{BK}_{k,\ell}}^{\mathcal{T}_S}(y)$. Thus, δ^* is defined on this 6-tuple and we can apply it to obtain $\rho_{\mathcal{IBK}_{k,\ell}}^{\mathcal{T}}(x)$, which is a list that contains for every $S \subseteq \text{bag}(x)$ the state

$$\delta(\text{bag}(x) \setminus S, \text{bag}(y) \setminus S, \emptyset, \text{edges}(x) \setminus E_S, \rho_{\mathcal{BK}_{k,\ell}}^{\mathcal{T}_{S \cap \text{bag}(y)}}(y), \perp)$$

If there is a vertex $v \in S \setminus \text{bag}(y)$, then $v \notin V_y$, i.e., v is not in the bag of a node in the subtree of $\mathcal{T}$ (or $\mathcal{T}_S$ or $\mathcal{T}_{S \cap \text{bag}(y)}$) rooted at y . It follows that $\mathcal{T}_S \upharpoonright_{V_y} = \mathcal{T}_{S \cap \text{bag}(y)} \upharpoonright_{V_y}$, so by [Observation B.3](#) $\rho_{\mathcal{BK}_{k,\ell}}^{\mathcal{T}_{S \cap \text{bag}(y)}}(y) = \rho_{\mathcal{BK}_{k,\ell}}^{\mathcal{T}_S}(y)$. Therefore, we have

$$\begin{aligned} & \delta(\text{bag}(x) \setminus S, \text{bag}(y) \setminus S, \emptyset, \text{edges}(x) \setminus E_S, \rho_{\mathcal{BK}_{k,\ell}}^{\mathcal{T}_{S \cap \text{bag}(y)}}(y), \perp) \\ &= \delta(\text{bag}_S(x), \text{bag}_S(y), \emptyset, \text{edges}_S(x), \rho_{\mathcal{BK}_{k,\ell}}^{\mathcal{T}_S}(y), \perp) \\ &= \rho_{\mathcal{BK}_{k,\ell}}^{\mathcal{T}_S}(x), \end{aligned}$$

so for every $S \subseteq \text{bag}(x)$, the list $\rho_{\mathcal{IBK}_{k,\ell}}^{\mathcal{T}}(x)$ contains $\rho_{\mathcal{BK}_{k,\ell}}^{\mathcal{T}_S}(x)$ and [Invariant B.5](#) is satisfied.

Now, assume that x has two children y, z . By [Definition B.2](#), we have

$$\rho_{\mathcal{IBK}_{k,\ell}}^{\mathcal{T}}(x) = \delta^*(\text{bag}(x), \text{bag}(y), \text{bag}(z), \text{edges}(x), \rho_{\mathcal{IBK}_{k,\ell}}^{\mathcal{T}}(y), \rho_{\mathcal{IBK}_{k,\ell}}^{\mathcal{T}}(z)).$$

By induction, [Invariant B.5](#) is satisfied for y and z , i.e., $\rho_{\mathcal{IBK}_{k,\ell}}^{\mathcal{T}}(y)$ and $\rho_{\mathcal{IBK}_{k,\ell}}^{\mathcal{T}}(z)$ are both lists that contain for every $S \subseteq \text{bag}(y)$ and $S \subseteq \text{bag}(z)$ the state $\rho_{\mathcal{BK}_{k,\ell}}^{\mathcal{T}_S}(y)$ and $\rho_{\mathcal{BK}_{k,\ell}}^{\mathcal{T}_S}(z)$, respectively. Thus, δ^* is again defined on this input and we can apply it to obtain $\rho_{\mathcal{IBK}_{k,\ell}}^{\mathcal{T}}(x)$, which is a list that contains for every $S \subseteq \text{bag}(x)$ the state

$$\begin{aligned} & \delta(\text{bag}(x) \setminus S, \text{bag}(y) \setminus S, \text{bag}(z) \setminus S, \text{edges}(x) \setminus E_S, \rho_{\mathcal{BK}_{k,\ell}}^{\mathcal{T}_{S \cap \text{bag}(y)}}(y), \rho_{\mathcal{BK}_{k,\ell}}^{\mathcal{T}_{S \cap \text{bag}(z)}}(z)) \\ &= \delta(\text{bag}_S(x), \text{bag}_S(y), \text{bag}_S(z), \text{edges}_S(x), \rho_{\mathcal{BK}_{k,\ell}}^{\mathcal{T}_S}(y), \rho_{\mathcal{BK}_{k,\ell}}^{\mathcal{T}_S}(z)) \\ &= \rho_{\mathcal{BK}_{k,\ell}}^{\mathcal{T}_S}(x), \end{aligned}$$

where we used the same arguments as before to show that $\rho_{\mathcal{BK}_{k,\ell}}^{\mathcal{T}_{S \cap \text{bag}(y)}}(y) = \rho_{\mathcal{BK}_{k,\ell}}^{\mathcal{T}_S}(y)$ and $\rho_{\mathcal{BK}_{k,\ell}}^{\mathcal{T}_{S \cap \text{bag}(z)}}(z) = \rho_{\mathcal{BK}_{k,\ell}}^{\mathcal{T}_S}(z)$. Again, this shows that [Invariant B.5](#) is satisfied, and thus finishes the proof of the claim. $\triangleleft$

This completes the description of the tree decomposition automaton $\mathcal{IBK}_{k,\ell}$ and, as argued before, proves the correctness of the automaton. Note that the states of $\mathcal{IBK}_{k,\ell}$ are lists of length $2^{\ell+1}$ consisting of states from $\mathcal{BK}_{k,\ell}$. Since the state space of $\mathcal{BK}_{k,\ell}$ is of size $\mathcal{O}_{k,\ell}(1)$ and can be computed in time $\mathcal{O}_{k,\ell}(1)$, the the same holds for $\mathcal{IBK}_{k,\ell}$. Moreover, the evaluation of δ^* requires $2^{\ell+1}$ evaluations of δ , where determining the input for each of these evaluations is possible in time $\mathcal{O}(\ell^2)$. To decide whether a state of $\mathcal{IBK}_{k,\ell}$ is accepting, we only need to look at the last state in the list and let $\mathcal{BK}_{k,\ell}$ decide, which takes time $\mathcal{O}_{k,\ell}(1)$. Thus, the evaluation time of $\mathcal{IBK}_{k,\ell}$ is $\mathcal{O}_{k,\ell}(1)$ as well. $\square$