Quantitative Verification on Product Graphs of Small Treewidth

Krishnendu Chatterjee
IST Austria, Austria

Rasmus Ibsen-Jensen
University of Liverpool, United Kingdom

Andreas Pavlogiannis
Aarhus University, Denmark

Abstract

Product graphs arise naturally in formal verification and program analysis. For example, the analysis of two concurrent threads requires the product of two component control-flow graphs, and for language inclusion of deterministic automata the product of two automata is constructed. In many cases, the component graphs have constant treewidth, e.g., when the input contains control-flow graphs of programs. We consider the algorithmic analysis of products of two constant-treewidth graphs with respect to three classic specification languages, namely, (a) algebraic properties, (b) mean-payoff properties, and (c) initial credit for energy properties.

Our main contributions are as follows. Consider a graph G that is the product of two constant-treewidth graphs of size n each. First, given an idempotent semiring, we present an algorithm that computes the semiring transitive closure of G in time $\tilde{O}(n^4)$. Since the output has size $\Theta(n^4)$, our algorithm is *optimal* (up to polylog factors). Second, given a mean-payoff objective, we present an $O(n^3)$-time algorithm for deciding whether the value of a starting state is non-negative, improving the previously known $O(n^4)$ bound. Third, given an initial credit for energy objective, we present an $O(n^5)$-time algorithm for computing the minimum initial credit for all nodes of G, improving the previously known $O(n^8)$ bound. At the heart of our approach lies an algorithm for the efficient construction of strongly-balanced tree decompositions of constant-treewidth graphs. Given a constant-treewidth graph G' of n nodes and a positive integer λ, our algorithm constructs a binary tree decomposition of G' of width $O(\lambda)$ with the property that the size of each subtree decreases geometrically with rate $(1/2 + 2^{-\lambda})$.

2012 ACM Subject Classification Theory of computation → Logic and verification; Theory of computation → Verification by model checking; Theory of computation → Graph algorithms analysis

Keywords and phrases graph algorithms, algebraic paths, mean-payoff, initial credit for energy

Digital Object Identifier 10.4230/LIPIcs.CVIT.2016.23

1 Introduction

Product graphs. Graphs are at the heart of formal analysis of reactive systems and programs. The nodes of the graph represent states of the system, edges represent transitions, and paths of the graph represent behaviors of the system. One graph problem that repeatedly arises in many applications is the analysis of product graphs (i.e., the synchronous product of two graphs). For example, in the analysis of two concurrent threads, the resulting graph for analysis is the product of two component control-flow graphs. Similarly, in language intersection or language inclusion between deterministic automata, the product of two automata is considered.

Specification languages. The analysis of programs and reactive systems is performed w.r.t. desired properties that are described as specification languages. We consider three classic specification languages: (i) algebraic properties w.r.t. a semiring, (ii) mean-payoff properties, and (iii) initial credit for energy properties. In algebraic properties for system

analysis, (a) each transition is associated with a weight from a semiring; (b) the value of a path is the semiring product operation of the weights along the transitions of the path; and (c) path aggregation is performed using the semiring sum operation across the values of paths. In mean-payoff properties for system analysis, (a) each transition of the system is associated with an integer-valued weight; (b) the value of an infinite path is the long-run average of the weights of the transitions of the path; and (c) an optimal path is selected that has the minimum mean-payoff value among all paths. In initial credit for energy for system analysis, (a) each transition of the system is associated with an integer-valued weight; (b) the value of an infinite path is the smallest weight of all of its prefixes, and (c) for each node, an optimal path is selected that starts in that node and such that it has the largest value among all paths originating in that node.

Constant-treewidth graphs. One key structural property of graphs that appears in several contexts is that of *constant-treewidth*. Treewidth is a classic measure of closeness of a graph to a tree [52]. Besides its mathematical elegance, constant-treewidth graphs are of practical relevance in formal verification and program analysis. For example, (i) the control-flow graphs of typical programming languages (such as goto-free Algol, Pascal, and C programs) have constant treewidth [53], which has been exploited for fast static analyses [21, 15, 18], and in practice even control-flow graphs of Java programs have constant treewidth [37]; (ii) the analysis of constant-treewidth graphs in logic plays a crucial role, such as the celebrated result of Courcelle for MSO [26] and its subsequent extensions [2, 30, 8]; as well as for logics such as modal mu-calculus [48].

Significance of the problems. In this work we consider the algorithmic analysis of the product of two constant-treewidth graphs with respect to algebraic properties and mean-payoff properties. We discuss the significance of the problems we consider. First, as mentioned above, products of two constant-treewidth graphs arise naturally (a) in the analysis of concurrent programs, and (b) in model checking an implementation against a high-level specification, a task typically expressed as language inclusion. We now discuss the relevance of the specification languages we consider here.

- *Semiring properties.* Semiring (or algebraic) properties have been widely used as specification formalisms as weighted automata [28], or properties of programs [51, 1]. Semirings also form the basis of dataflow analysis of concurrent programs [36, 45, 32, 24, 41, 27, 19], where the underlying analysis is based on an algebraic "meet-over-all-paths" formulation. Finally, semirings also arise in concurrent Kleene algebras for the analysis of concurrent programs [38, 39, 42].
- *Mean-payoff properties.* Mean-payoff is a classic quantitative property in performance analysis [33, 49, 3]. It has applications in (a) automata theoretic formalisms [17, 16]; (b) weighted logic formalisms [9, 12, 29]; (c) synthesis of reactive systems [5, 14]; and (d) quantitative interprocedural analysis [22]; to name a few applications in verification and program analysis.
- *Initial credit for energy properties.* Initial credit is a useful quantitative property for expressing energy constraints [11, 10, 13]. The goal is to determine for each state of the system an initial energy supply so that the system can exhibit infinite behavior without running out of energy, and has numerous applications in planning [31, 40].

In many cases in verification, instead of having a graph with constant treewidth, the input is a graph G that is the product of two constant treewidth graphs G_1, G_2. For example, this holds when we analyze control-flow graphs of two threads running in parallel, or when we test for language inclusion and the system and specification automaton have constant-treewidth. Note that G does not have constant treewidth: a simple example would be grid graphs - they

are trivial to get this way, but have tree-width equal to the minimum of height and width. Due to this fact, existing algorithms for constant-treewidth graphs give sub-optimal solutions. The question is thus whether better algorithms are possible given the fact that, although G does not have constant treewidth, its two components G_1 and G_2 do have constant treewidth. We address this challenge in this work.

Our contributions. Our main contributions are as follows.

1. *Mean-payoff properties.* Given a product G of two constant treewidth graphs G_1, G_2 of size n each and a mean-payoff objective, we present an $O(n^3)$-time algorithm for deciding whether the value of a starting state is non-negative. Note that constant-treewidth graphs have $O(n)$ edges. Existing algorithms (such as Karp's algorithm) solve this problem in $O(n \cdot m)$ time on a graph of n nodes and m edges, which results to $O(n^4)$ complexity on G (since G has n^2 nodes and edges). Hence our algorithm yields a factor-n improvement compared to existing approaches.
2. *Semiring properties.* Given an idempotent semiring and a product G of two constant-treewidth graphs G_1, G_2 of size n each, we present an algorithm that computes the semiring transitive closure of G in time $\tilde{O}(n^4)$. On the other hand, applying the classic cubic-time transitive closure algorithm on G yields an $O(n^6)$ bound. Although this naive bound has been improved to $O(n^{4+\epsilon})$, for fixed $\epsilon > 0$ [19], our work yields a further polynomial improvement to $\tilde{O}(n^4)$. Since G has n^2 nodes, the output has size $\Theta(n^4)$ and thus our algorithm is *optimal* (up to polylog factors).
3. *Initial credit for energy properties.* Given a product G of two constant treewidth graphs G_1, G_2 of size n each and an energy objective, we present an $O(n^5)$-time algorithm for computing the minimum initial credit for each node of G. The best existing solution is due to [20] which has quartic complexity in the size of the graph, and thus implies an $O(n^8)$ time bound on G (since G has n^2 nodes). Hence our algorithm yields a factor-n^3 improvement.
4. At the heart of our approach lies a new notion of (α, β) tree decompositions, as well as an efficient algorithm for their construction for constant-treewidth graphs. Given a constant-treewidth graph G of n nodes and an integer $\lambda \geq 2$, our algorithm constructs in $O(\lambda^2 \cdot n \cdot \log n)$ time a binary tree decomposition of G of width $O(\lambda)$ that has the following property: for each bag B of the tree decomposition at level i, the number of nodes of G contained in bags of the subtree rooted at B is at most $n \cdot (1/2 + 2^{-\lambda})^i$. Hence, for increasing values of λ, the number of contained nodes gets exponentially close to the optimal value of $n \cdot 2^{-i}$ (since the tree decomposition is binary). Note that for $\lambda = O(1)$, we obtain a tree decomposition with logarithmic depth and increased width by a constant factor. We complement this result with a lower bound stating that tree decompositions of logarithmic depth must, in general, incur a constant-factor increase in the width.

   Due to space restrictions, some proofs are relegated to the appendix.

Comparison to related existing work. The notion of balanced tree decompositions has long existed in the literature. The classic work of [50] presents the first algorithm to construct a balanced tree decomposition in time $O(n \cdot \log n)$, by finding balanced separators in the graph. Various works present parallel algorithms for constructing balanced tree decompositions in (poly-)logarithmic parallel time with $O(n)$ processors [46, 7]. The work of [30] constructs balanced tree decompositions in Logspace. More recently, the work of [35] constructs approximate balanced tree decompositions in time $O(f(t) \cdot n \cdot \log n))$, where $f(t)$ is a polynomial function of the treewidth t.

In all these cases, the balancing guarantee is that the tree decomposition has depth $\leq c \cdot \log n$, for some constant c (logarithms are on base 2). For binary tree decompositions,

these algorithms yield $c \geq 2$. Crucially, the complexity of the algorithms for our results (1)-(3) depends on c, and $c \geq 2$ is prohibitively large. E.g., using the existing algorithms for balanced tree decompositions yields a bound for the semiring properties (result (2)) that is at least n^6, as opposed to our $\tilde{O}(n^4)$ bound.

Although [19] also considers a notion of strong balancing, the balancing factor achieved there for level i is, on average, $(1/2 + P(\lambda^{-1}))^i$, where P is a sub-linear function. Hence, compared to that algorithm, our new algorithm yields an exponential improvement in the balancing factor (i.e., $(1/2 + 2^{-\lambda})^i$). This improvement is necessary to arrive at our results. Moreover, our new techniques are quite different from previous ones, and might be of independent interest.

2 Preliminaries

In this section we set up our main notation and introduce our new notion of (α, β) tree decompositions. In the next sections we will show how to construct such decompositions efficiently, as well as how they can be used for developing algorithmic improvements on product graphs.

Graphs. We consider directed graphs $G = (V, E)$ where V is a set of n nodes and $E \subseteq V \times V$ is an edge relation. Two nodes $u, v \in V$ are called *neighbors* if $(u, v) \in E$. Given a set $X \subseteq V$, we denote by $G \restriction X$ the subgraph $(X, E \cap (X \times X))$ of G induced by the set of nodes X. A path $P : u \rightsquigarrow v$ is a sequence of nodes $(x_1, \ldots, x_k)$ such that $u = x_1$, $v = x_k$, and for all $1 \leq i \leq k-1$ we have $(x_i, x_{i+1}) \in E$. The path P is *acyclic* if every node appears at most once in P. The length of P is $k-1$, and a single node is by itself a 0-length path. Given a path P we use the notation $u \in P$ to say that a node u appears in P, and $A \cap P$ to refer to the set of nodes that appear in both P and a set A.

Trees. A (rooted) tree $T = (I, F)$ is an undirected graph (the edge relation F is symmetric) without self loops and with a distinguished node r, which is the root of T, such that there is a unique acyclic path $P_u^v : u \rightsquigarrow v$ for each pair of nodes u, v. The *size* of T is $|I|$. Given a tree T with root r, the *level* $\mathsf{Lv}(u)$ of a node u is the length of the path P_u^r from u to the root r. Every node in P_u^r is an *ancestor* of u, and if v is an ancestor of u, then u is a *descendant* of v (u is both an ancestor and a descendant of itself). We call v a *strict ancestor* (resp., *strict descendant*) of u if $v \neq u$ and v is an ancestor (resp., descendant) of u. For a pair of nodes $u, v \in I$, the *lowest common ancestor (LCA)* of u and v is the common ancestor of u and v with the largest level. Given a node $v \neq r$, the *parent* u of v is the unique ancestor of v in level $\mathsf{Lv}(v) - 1$, and v is a *child* of u. We denote by $\mathsf{Parent}(v)$ the parent of node $v \neq r$. A *leaf* of T is a node with no children. For a node $u \in I$, we denote by $T(u)$ the subtree of T rooted in u (i.e., the tree consisting of all descendants of u). A node is k-ary if it has at most k children, and a tree is k-ary if every node is k-ary. The *depth* of T is $\max_u \mathsf{Lv}(u)$.

Connected components of trees. Let $T = (I, F)$ be a tree. A *connected component* $\mathcal{C} \subseteq I$ of T is such that for every pair of nodes $u, v \in \mathcal{C}$, the unique acyclic path P_u^v in T visits only nodes in $\mathcal{C}$. The *border* of a non-empty $\mathcal{C}$ is the set $\mathsf{Border}(\mathcal{C}) = \{u \in I \setminus \mathcal{C} : \exists v \in \mathcal{C} \text{ s.t. } (u, v) \in F\}$, i.e., it is the set of nodes of T that are adjacent to $\mathcal{C}$. For technical convenience, in this work we also make use of empty connected components of T. Empty connected components are also associated with a border, which is two endpoints of an edge of T. Hence we have $|F|$ many different empty connected components.

Balancing separators. Consider a connected component $\mathcal{C}$ of a tree. A *balancing separator* of $\mathcal{C}$ is a node $u \in \mathcal{C}$ such that removing u splits $\mathcal{C}$ into (at most) 3 connected components $\{\mathcal{C}_i\}_{1 \leq i \leq 3}$, with $|\mathcal{C}_i| \leq |\mathcal{C}|/2$ for each $1 \leq i \leq 3$. The following lemma is well-known (e.g., [25]).

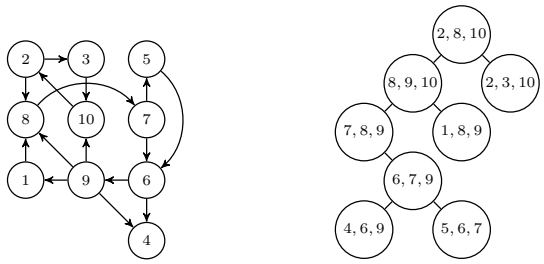

Figure 1 A graph with treewidth 2 (left) and a corresponding tree-decomposition (right).

▶ **Lemma 1.** *Consider a binary tree T and a connected component $\mathcal{C}$ of T. A balancing separator of $\mathcal{C}$ can be computed in $O(|\mathcal{C}|)$ time.*

Tree decompositions. A *tree decomposition* of a graph $G = (V, E)$ is a pair $(\mathcal{B}, T)$ where $T = (I, F)$ is a tree and $\mathcal{B} = \{B_i : i \in I\}$ is a family of subsets of V such that the following conditions hold.

1. $\bigcup_{i \in I} B_i = V$
2. For all $(u, v) \in E$ there exists $i \in I$ with $u, v \in B_i$.
3. For all $i, j, k \in I$, if $k \in P_i^j$ in T then $B_i \cap B_j \subseteq B_k$.

The sets B_i are called *bags* of the tree decomposition. Given a node $u \in V$, we denote by i_u the *root node* of u in T, which is the smallest-level node in T such that $u \in B_{i_u}$, and call B_{i_u} the *root bag* of u. Conditions Item 1 and Item 3 of tree decompositions guarantee that every node has a unique root bag. The *width* of the tree decomposition is $\max_{i \in I} |B_i| - 1$, i.e., it is the size of the largest bag of $\mathcal{B}$ minus 1. The *treewidth* of G is the smallest width of all tree decompositions of G. Given a node $i \in I$, we denote by $N_{\mathcal{B}}^T(i)$ the set of nodes of G that appear in bags of the subtree $T(i)$ (i.e., $u \in N_{\mathcal{B}}^T(i)$ iff i has a descendant j such that $u \in B_j$), and by $\mathcal{Y}_{\mathcal{B}}^T(i)$ the set of nodes of G that appear only in bags of the subtree $T(i)$ (i.e., $u \in \mathcal{Y}_{\mathcal{B}}^T(i)$ iff for the root bag B_j of u, j is a descendant of i). Observe that $N_{\mathcal{B}}^T(i) \subseteq \mathcal{Y}_{\mathcal{B}}^T(i) \cup B_i$. We assume w.l.o.g. that $\mathcal{Y}_{\mathcal{B}}^T(i) \neq \emptyset$ for every $i \in I$, as otherwise the subtree $T(i)$ can be removed from T and obtain a valid tree decomposition of G. For simplicity of exposition, we associate properties of the tree T with the tree decomposition $(\mathcal{B}, T)$, e.g., the depth of the tree decomposition is the depth of T, and we say that the tree decomposition is balanced if T is balanced. The following lemma states a well-known separator property of tree decompositions, which is a key property behind many efficient algorithms on low-treewidth graphs.

▶ **Lemma 2** (Lemma 3,[6]). *Consider a graph $G = (V, E)$, a tree-decomposition $(\mathcal{B}, T = (I, F))$ of G and a node $i \in I$. Let $\{\mathcal{C}_j\}_j$ be the connected components of T created by removing i from T. Consider two integers j_1, j_2 such that $j_1 \neq j_2$, and two nodes $i_1 \in \mathcal{C}_{j_1}$ and $i_2 \in \mathcal{C}_{j_2}$. For any two nodes $u \in B_{i_1}$ and $v \in B_{i_2}$, every path $P : u \rightsquigarrow v$ in G contains a node that appears in B_i.*

Approximate, balanced, and (α, β) tree decompositions. Consider a graph G of n nodes and treewidth t, and let $(\mathcal{B}, T = (I, F))$ be a tree decomposition of G. We refer to $(\mathcal{B}, T)$ as *α-approximate*, for some integer $\alpha \geq 1$, if the width of $(\mathcal{B}, T)$ is $\leq \alpha \cdot (t + 1) - 1$. We refer to $(\mathcal{B}, T)$ as *β-balanced*, for some $0 < \beta < 1$, if for every node $i \in I$ we have that $\mathcal{Y}_{\mathcal{B}}^T(i) \leq n \cdot \beta^{\mathsf{Lv}(i)}$. If $(\mathcal{B}, T)$ is both α-approximate and β-balanced, it is called a (α, β) tree decomposition. Intuitively, a (α, β) tree decomposition approximates the treewidth of G to a factor α, and for every $i \in I$, the number of nodes of G contained in bags of the subtree $T(i)$

decreases geometrically with $\mathsf{Lv}(i)$ by a factor β. Note that if β is constant (i.e., independent of G) then T has depth $O(\log n)$ (hence it is balanced).

3 Construction of (α, β) tree decompositions

In this section we present our algorithm for constructing (α, β) tree decompositions, where α and β depend on some integer input $\lambda \geq 2$. In particular, we establish the following theorem.

▶ **Theorem 3.** *Consider a graph G of n nodes and treewidth t, and any integer $\lambda \geq 2$. Let $\mathcal{T}(G)$ be the time required to construct a tree decomposition of G with $\leq n$ bags and width t. A (α, β) tree decomposition of G can be constructed in $O(\mathcal{T}(G) + \lambda^2 \cdot n \cdot \log n)$ time, where $\alpha = 11 \cdot \lambda + 32$ and $\beta = 1/2 + 2^{-\lambda}$.*

Hence, for larger values of λ, the constructed tree decomposition is more strongly balanced, which also incurs a factor increase in its width.

The motivation behind Theorem 3 is as follows. The properties we consider later for product graphs (i.e., semiring, mean-payoff and initial credit for energy properties) are solved by existing algorithms that operate on a tree decomposition of the input graph. In high level, these algorithms iterate over every bag in the input tree decomposition and perform a polynomial-time computation in it. Because we deal with tree decompositions of product graphs, as we go down the tree decomposition, the number of bags in each level increases geometrically. By using Theorem 3, we ensure that the size of the bags reduces geometrically by an appropriate factor, which in turn ensures that the total time spent for all bags in each level of the tree decomposition stays bounded. The constant λ is chosen in each case to ensure this effect.

The high-level intuition behind Theorem 3 is as follows. We start with a tree decomposition of G, obtained using standard algorithms. The tree decomposition is then split recursively. Each recursive step operates on a part of the tree that consists of connected components, and splits this part into two sub-parts, in such a way that the overall number of nodes is balanced, meaning that the two parts are of approximately equal size. Since these parts shrink by a factor of $1/2$ in every step, after λ steps, the balance is only off by at most a factor of $2^{-\lambda}$. The key challenge is to perform the aforementioned splits in such a way that (i) the two constructed parts are approximately balanced, and (ii) the nodes appearing in each such part are separated from the rest of the nodes via a few "border" nodes.

We complement Theorem 3 by showing that, generally, balanced tree decompositions are approximate.

▶ **Theorem 4.** *For any n and $t = o(n/\log n)$, there exists a graph G_t^n that has treewidth at most $2 \cdot t - 1$, but any tree decomposition of G_t^n with depth $O(\log n)$ has width at least $3 \cdot t - 1$.*

In Section 3.1 we develop two operations on tree components that will be used later on. In Section 3.2 we develop our main algorithm which uses the operations of Section 3.1 to construct the (α, β) tree decomposition of Theorem 3. Finally, in Section 3.3 we prove the lower-bound of Theorem 4.

3.1 Operations on tree components

In this section we define set-components of trees and two operations on such set-components that will be used later for constructing (α, β) tree decompositions. In words, a set-component of a tree is simply a set of pairwise-disjoint connected components of the tree. The operations

that we introduce here take as input a set-component. Each operation splits that set-component into multiple sub-components so that certain properties are met. In what follows, we fix a tree $T = (I, F)$.

Set-components of trees. A *set-component* of T is a set $\mathcal{X} = \{\mathcal{C}_i\}_i$ of connected components of T such that for each $i \neq j$ we have (i) $\mathcal{C}_i \cap \mathcal{C}_j = \emptyset$ and (ii) $\mathcal{C}_i \cap \mathsf{Border}(\mathcal{C}_j) = \emptyset$. The *size* of $\mathcal{X}$ is $\mathsf{size}(\mathcal{X}) = \sum_i |\mathcal{C}_i|$, i.e., it is the total size of all of its connected components. The *border* of the set-component $\mathcal{X}$ is defined as $\mathsf{Border}(\mathcal{X}) = \bigcup_i \mathsf{Border}(\mathcal{C}_i)$, i.e., the border of $\mathcal{X}$ is the union of the borders of its connected components. Given two set-components $\mathcal{X}$ and $\mathcal{X}'$, we define $\mathcal{X} \sqsubseteq \mathcal{X}' = \bigcup_{\mathcal{C}_i \in \mathcal{X}} \mathcal{C}_i \subseteq \bigcup_{\mathcal{C}'_i \in \mathcal{X}'} \mathcal{C}'_i$ and $\mathcal{X} \sqcap \mathcal{X}' = (\bigcup_{\mathcal{C}_i \in \mathcal{X}} \mathcal{C}_i) \cap (\bigcup_{\mathcal{C}'_i \in \mathcal{X}'} \mathcal{C}'_i)$.

3.1.1 The operation BorderSplit

Intuitively, a component $\mathcal{X}$ is connected to the rest of the tree via the nodes of its border $\mathsf{Border}(\mathcal{X})$. Our balancing algorithm later needs that this border never gets too large. To achieve this, we define the operation BorderSplit. In particular, consider a set-component $\mathcal{X} = \{\mathcal{C}_i\}_{1 \leq i \leq k}$ of T. We define the operation BorderSplit on $\mathcal{X}$ which returns (at most) three set-components$\{\mathcal{X}_i\}_{1 \leq i \leq 3}$ with $\mathcal{X}_i \sqsubseteq I$ for each $1 \leq i \leq 3$. In words, BorderSplit splits $\mathcal{X}$ into three set-components $\mathcal{X}_i$, possibly by removing a node $u \in \mathcal{X}$, such that $|\mathsf{Border}(\mathcal{X}_i)| \leq |\mathsf{Border}(\mathcal{X})|/2 + 1$ for each $1 \leq i \leq 3$. Formally, BorderSplit operates as follows (recall that the input tree T is binary). If $|\mathsf{Border}(\mathcal{X})| < 3$, then we simply return $\{\mathcal{X}\}$. Otherwise, we perform the following steps. First, we construct the LCA tree $T' = (I', F')$ of $\mathsf{Border}(\mathcal{X})$, defined as follows.

1. I' is the smallest subset of I such that (i) $\mathsf{Border}(\mathcal{X}) \subseteq I'$ and (ii) for every two nodes $u, v \in I'$, we have $w \in I'$, where w is the LCA of u and v. Hence, I' is the LCA-closure of $\mathsf{Border}(\mathcal{X})$.
2. For every pair of distinct nodes $u, v \in I'$, we have that $(u, v) \in F'$ iff v is the largest-level strict ancestor of u in T that appears in I'.

Then, we find a node u of T' such that the removal of u splits T' into three connected components $\{A_i\}_i$ with $|A_i \cap \mathsf{Border}(\mathcal{X})| \leq \lfloor |\mathsf{Border}(\mathcal{X})|/2 \rfloor$. This is easily done by a bottom-up pass of T'. For each $1 \leq i \leq 3$, we construct a set-component $\mathcal{X}_i$, as follows.

1. For every connected component $\mathcal{C}_j \in \mathcal{X}$ with $u \notin \mathcal{C}_j$, we make $\mathcal{C}_j \in \mathcal{X}_i$ where i is such that $\mathsf{Border}(\mathcal{C}_j) \subseteq A_i$.
2. If there exists a connected component $\mathcal{C}_j \in \mathcal{X}$ with $u \in \mathcal{C}_j$, we split $\mathcal{C}_j$ into three connected components $\{\mathcal{C}^i_j\}_i$ by removing u from $\mathcal{C}_j$, such that $\mathsf{Border}(\mathcal{C}^i_j) \setminus \{u\} \subseteq A_i$. In addition, we take $\mathcal{C}^i_j \in \mathcal{X}_i$.

Finally, we return the set $\{\mathcal{X}_i\}_i$. The following lemma states the properties of BorderSplit.

▶ **Lemma 5.** *Consider the operation* BorderSplit *on the set-component* $\mathcal{X} = \{\mathcal{C}_i\}_{1 \leq i \leq k}$. *Let* $z = |\mathsf{size}(\mathcal{X})|$, $m = |\mathsf{Border}(\mathcal{X})|$, *and* $\{\mathcal{X}_i\}_{1 \leq i \leq 3}$ *be the returned component set. The following assertions hold.*

1. *We have* $|\bigcup_i \mathsf{Border}(\mathcal{X}_i)| \leq m + 1$.
2. *For each* $1 \leq i \leq 3$, *we have* $|\mathsf{Border}(\mathcal{X}_i)| \leq \lfloor m/2 \rfloor + 1$.
3. *After one* $O(n)$*-time preprocessing of* T, *every call to* BorderSplit *requires* $O(z + m^2)$ *time.*

3.1.2 The operation Split

Consider a set-component $\mathcal{X} = \{\mathcal{C}_i\}_{1 \leq i \leq k}$ of T and some $\lambda \geq 2$. Let $z = \mathsf{size}(\mathcal{X})$ and $m = |\mathsf{Border}(\mathcal{X})|$. We define the operation Split on $\mathcal{X}$ which returns two set-components $\{\mathcal{X}_i\}_{1 \leq i \leq 2}$ such that the following properties hold.

1. $\mathcal{X}_i \sqsubseteq \mathcal{X}$ and $\mathcal{X}_1 \sqcap \mathcal{X}_2 = \emptyset$.

2. $\mathsf{size}(\mathcal{X}_i) \leq z \cdot (1/2 + 2^{-\lambda})$, i.e., the size of each set-component is approximately half the size of $\mathcal{X}$, and this approximation is controlled by λ, and
3. $|\mathsf{Border}(\mathcal{X}_i)| \leq 9/10 \cdot m + \lambda + 3$, i.e., the size of the border of each set-component is about a fraction $9/10$ of the size of the border of $\mathcal{X}$.

Split initializes the two set-components as empty, i.e., $\mathcal{X}_1 = \mathcal{X}_2 = \emptyset$, and will be inserting connected components in each $\mathcal{X}_i$ such that, in the end, the stated properties hold. The algorithm operates in two steps, where the input to the second step is a set-component $\mathcal{X}^*$ constructed in the first step. In the following, we describe the two steps. We say that a set-component $\mathcal{X}' \sqsubseteq \mathcal{X}$ is *border-balancing* if either $m < 10$, or $\mathsf{size}(\mathcal{X}') \leq z/2$ and $|\mathsf{Border}(\mathcal{X}')| \geq m/10 - 1$. Hence if $m < 10$ we take every sub-component of $\mathcal{X}$ to be border-balancing, otherwise a border-balancing sub-component must be at most of half the size of $\mathcal{X}$ and have about at least $1/10$-th of the size of the border of $\mathcal{X}$. Given two set-components $\mathcal{X}'$ and $\mathcal{X}''$, we say that we *add* $\mathcal{X}'$ to $\mathcal{X}''$ meaning that we update $\mathcal{X}''$ to $\mathcal{X}' \cup \mathcal{X}''$.

Step 1: making $\mathcal{X}_1$, $\mathcal{X}_2$ border-balancing. If $m < 10$, then we proceed with the second step with $\mathcal{X}^* = \mathcal{X}$. Otherwise, we apply the operation BorderSplit on $\mathcal{X}$ and obtain the set-components $\{\mathcal{X}'_i\}_{1 \leq i \leq 3}$. We assume w.l.o.g. that $\mathsf{size}(\mathcal{X}'_1) \geq \mathsf{size}(\mathcal{X}'_2) \geq \mathsf{size}(\mathcal{X}'_3)$. If there are two border-balancing set-components, $\mathcal{X}_a$ and $\mathcal{X}_b$, among $\mathcal{X}'_1, \mathcal{X}'_2, \mathcal{X}'_3$, we add $\mathcal{X}_a$ to $\mathcal{X}_1$ and $\mathcal{X}_b$ to $\mathcal{X}_2$, and proceed to the second step with $\mathcal{X}^*$ being the unique set-component in $\{\mathcal{X}'_1, \mathcal{X}'_2, \mathcal{X}'_3\} \setminus \{\mathcal{X}_a, \mathcal{X}_b\}$. Otherwise, we apply the operation BorderSplit on $\mathcal{X}'_1$ and obtain the set-components $\{\mathcal{X}''_i\}_{1 \leq i \leq 3}$. We assume w.l.o.g. that $\mathsf{size}(\mathcal{X}''_1) \geq \mathsf{size}(\mathcal{X}''_2) \geq \mathsf{size}(\mathcal{X}''_3)$. In the set $A = \{\mathcal{X}''_1, \mathcal{X}''_2, \mathcal{X}''_3, \mathcal{X}'_2, \mathcal{X}'_3\}$ there are at least two border-balancing set-components, $\mathcal{X}_a$ and $\mathcal{X}_b$. We add $\mathcal{X}_a$ to $\mathcal{X}_1$ and $\mathcal{X}_b$ to $\mathcal{X}_2$. Finally, we let $\mathcal{X}^*$ be the set-component in $A \setminus \{\mathcal{X}_a, \mathcal{X}_b\}$ with the largest size. We add one of the set-components in $A \setminus \{\mathcal{X}^*, \mathcal{X}_a, \mathcal{X}_b\}$ to the smaller (in size) of $\mathcal{X}_1$ and $\mathcal{X}_2$, and repeat with the other set-component of $A \setminus \{\mathcal{X}^*, \mathcal{X}_a, \mathcal{X}_b\}$.

Step 2: balancing $\mathcal{X}_1$, $\mathcal{X}_2$ based on size. Let $\mathcal{C}$ be a largest connected component of $\mathcal{X}^*$. First, we add every connected component of $\mathcal{X}^*$ except $\mathcal{C}$ to the smaller (in size) of $\mathcal{X}_1$ and $\mathcal{X}_2$, in order (i.e., once we have added one connected component to $\mathcal{X}_1$ or $\mathcal{X}_2$, we take into account the new size of $\mathcal{X}_1$ and $\mathcal{X}_2$ for choosing where to add the next connected component). The remaining of the second step is recursive for λ levels. The j-th recursive call operates on a connected component $\mathcal{C}^j$, where $\mathcal{C}^0 = \mathcal{C}$. Given $\mathcal{C}^j$, we use Lemma 1 to identify a balancing separator $u \in \mathcal{C}^j$, such that the removal of u splits $\mathcal{C}^j$ into three connected components $\{\mathcal{C}^j_i\}_i$. We assume w.l.o.g. that $|\mathcal{C}^j_1| \geq |\mathcal{C}^j_2| \geq |\mathcal{C}^j_3|$. We add each of $\mathcal{C}^j_2$ and $\mathcal{C}^j_3$ to the smallest (in size) of $\mathcal{X}_1$ and $\mathcal{X}_2$, in order, and proceed to the next recursive call with $\mathcal{C}^{j+1}$ be $\mathcal{C}^j_1$. Finally, at the end of the recursion, we add $\mathcal{C}^\lambda$ (from the last recursive call) to the smallest of $\mathcal{X}_1$ or $\mathcal{X}_2$.

The following lemma states the properties of Split, and relies on Lemma 5.

▶ **Lemma 6.** *Consider the operation* Split *on a set-component $\mathcal{X} = \{\mathcal{C}_i\}_{1 \leq i \leq k}$ Let $z = \mathsf{size}(\mathcal{X})$ and $m = |\mathsf{Border}(\mathcal{X})|$, and $\{\mathcal{X}_i\}_{1 \leq i \leq 2}$ be the returned component set. The following assertions hold.*

1. *For each $1 \leq i \leq 2$, we have $\mathsf{size}(\mathcal{X}_i) \leq z \cdot (1/2 + 2^{-\lambda})$.*
2. *For each $1 \leq i \leq 2$, we have $|\mathsf{Border}(\mathcal{X}_i)| \leq 9/10 \cdot m + \lambda + 3$.*
3. *We have $|\mathsf{Border}(\mathcal{X}_1) \cup \mathsf{Border}(\mathcal{X}_2)| \leq |\mathsf{Border}(\mathcal{X})| + \lambda + 2$*
4. *After $O(n)$-time preprocessing of T,* Split *requires $O(z + m^2)$ time.*

3.2 Construction of (α, β) Tree Decomposition

Here we prove the main result of this section, which concerns the construction of an (α, β) tree-decomposition of a graph G, where $\alpha = 11 \cdot \lambda + 32$ and $\beta = 1/2 + 2^{-\lambda}$, for any integer $\lambda \geq 2$. Our construction proceeds in two steps. In Section 3.2.1 we present an intermediate

step that takes as input a tree T and constructs a tree decomposition of T that has width α and is β-approximate. In Section 3.2.2 we use this construction towards Theorem 3.

3.2.1 Construction of tree decompositions of trees

Here we present algorithm Balance, which takes as input a tree T and some integer $\lambda \geq 2$, and constructs a tree decomposition of T that has width α and is β-approximate.

Step 1: constructing a component tree. Consider a tree $T = (I, F)$. A *component tree* of T is a pair $(\mathcal{V}, \mathcal{R})$ where $\mathcal{R} = (\mathcal{J}, \mathcal{D})$ is a rooted tree and $\mathcal{V} = \{\mathcal{X}_i : i \in \mathcal{J}\}$ is a set of components of T. In the first step, Balance constructs a component tree $(\mathcal{V}, \mathcal{R})$ recursively, as follows.

1. The root of $\mathcal{R}$ is the component $\{I\}$.
2. Given a node i of $\mathcal{R}$, if $\mathsf{size}(\mathcal{X}_i) > 0$, we use the operation Split on the component $\mathcal{X}_i$ to obtain two components $\mathcal{X}_{j_1}, \mathcal{X}_{j_2}$. We insert these components in the set $\mathcal{V}$, make j_1, j_2 children of i in $\mathcal{R}$, and proceed recursively with each of j_1, j_2.

Step 2: turning the component tree to a tree decomposition. At the end of the first step, we have constructed a component tree $(\mathcal{V}, \mathcal{R})$ where every leaf of $\mathcal{R}$ is an empty set-component. In the second step, we construct a tree decomposition $(\mathcal{B}, \mathcal{T} = (\mathcal{I}, \mathcal{F}))$, such that $\mathcal{I} = \mathcal{J} \setminus L$, where L is the set of leaves of $\mathcal{R}$, and $\mathcal{T} = \mathcal{R} \restriction \mathcal{I}$. In words, $\mathcal{T}$ is identical to $\mathcal{R}$ without the leaves of the latter. For each $i \in \mathcal{I}$, we have $B_i = \bigcup_j \mathsf{Border}(\mathcal{X}_j)$, where j ranges over the children of i in $\mathcal{R}$. Note that i ranges over non-leaves of $\mathcal{R}$, and thus each B_i is well-defined. The following remark states that given a node i of the induced tree decomposition, the corresponding set-component $\mathcal{X}_i$ of the component tree contains exactly the nodes that appear in the bags of the subtree $\mathcal{T}(i)$.

▶ Remark 7. Let $(\mathcal{B}, \mathcal{T} = (\mathcal{I}, \mathcal{F}))$ be the tree decomposition of the component tree $(\mathcal{V}, \mathcal{R} = (\mathcal{J}, \mathcal{D}))$ constructed in the second step. For every $i \in \mathcal{I}$, we have $\mathcal{Y}^{\mathcal{T}}_{\mathcal{B}}(i) = \mathcal{X}_i$.

The following lemmas establish the correctness and complexity of Balance.

▶ **Lemma 8.** *$(\mathcal{B}, \mathcal{T} = (\mathcal{I}, \mathcal{F}))$ is a tree decomposition of T that has width $\leq \alpha$ and is β-balanced, for $\alpha = 11 \cdot \lambda + 32$ and $\beta = 1/2 + 2^{-\lambda}$.*

▶ **Lemma 9.** Balance *runs in $O(\lambda^2 \cdot |I| \cdot \log |I|)$ time.*

3.2.2 Construction of (α, β) tree decompositions

Finally, we present an algorithm for constructing (α, β) tree decompositions of arbitrary graphs. The input is a graph and a tree decomposition of the graph, and our algorithm constructs a new tree decomposition with the desired properties.

Construction of an (α, β) tree decomposition. Consider a graph $G = (V, E)$ with treewidth t, and some integer $\lambda \geq 2$. We construct a (α, β) tree decomposition of G, where $\alpha = 11 \cdot \lambda + 32$ and $\beta = 1/2 + 2^{-\lambda}$ in three steps.

1. Let $(\mathcal{B} = \{B\}_i, T = (I, F))$ be a tree decomposition of G with $|I| \leq n$ and width $\leq t$. We assume w.l.o.g. that, in $(\mathcal{B}, T)$, every node of G has a unique root bag, as we can always replace a bag which is the root of $k > 1$ nodes with a sequence of k bags, each being the root of a single node. In addition, we can remove any bag that is not the root bag of any node, and thus the size of the tree decomposition is exactly n.
2. We use Balance to construct a tree decomposition $(\mathcal{B}' = \{B'_i\}_i, T' = (I', F'))$ of T.
3. We construct the tree decomposition $(\overline{\mathcal{B}} = \{\overline{B}_i\}_i, T')$ with $\overline{B}_i = \bigcup_{j \in B'_i} B_j$ for each $i \in I'$.

We conclude Theorem 3 by arguing that the construction produces a (α, β) tree decomposition of G.

Proof of Theorem 3. Consider the tuple $(\mathcal{B}' = \{B'_i\}_i, T' = (I', F'))$ constructed in Step 2. By Lemma 8, $(\mathcal{B}', T')$ is a tree decomposition of T, and has width $\leq \alpha$ and is β-balanced. It follows that $(\overline{\mathcal{B}}, T')$ is a α-approximate tree decomposition of G, and it remains to argue that it is also β-balanced. For a node $u \in V$, let $A_u = \{i \in I : u \in B_i\}$ be the connected component of T in which u appears and we have $|A_u| \geq 1$. In particular, this holds for the (unique) root node i_u of u in T. Observe that for every node $j \in I'$, if $u \in \mathcal{Y}_{\overline{\mathcal{B}}}^{T'}(j)$, then $A_u \subseteq \mathcal{Y}_{\mathcal{B}'}^{T'}(j)$. Hence $|\mathcal{Y}_{\overline{\mathcal{B}}}^{T'}(j)| \leq |\mathcal{Y}_{\mathcal{B}'}^{T'}(j)|$. By Lemma 8, we have that $(\mathcal{B}', T')$ is β-balanced, and thus $|\mathcal{Y}_{\mathcal{B}'}^{T'}(j)| \leq |I| \cdot \beta^{\mathsf{Lv}(j)}$. Since $|I| = n$, we conclude that $|\mathcal{Y}_{\overline{\mathcal{B}}}^{T'}(j)| \leq n \cdot \beta^{\mathsf{Lv}(j)}$, as required.

We now turn our attention to the running time. The algorithm requires $\mathcal{T}(G)$ time in Step 1 for obtaining the initial tree decomposition $(\mathcal{B}, T)$ plus $O(n \cdot t)$ time for ensuring the properties of $(\mathcal{B}, T)$. By Lemma 9, the algorithm requires $O(\lambda^2 \cdot n \cdot \log n)$ in Step 2. Finally, the algorithm requires $O(\alpha \cdot t \cdot n) = O(\lambda \cdot t \cdot n)$ time in Step 3. ◀

3.3 A lower bound on the width of balanced tree decompositions

Here we present a family $\{G_t^n \mid n \geq 3 \cdot t \text{ and } n \equiv 0 \pmod t\}$ of graphs, where G_t^n has n nodes and treewidth $2 \cdot t - 1$, and any tree decomposition of G_t^n with width t' and depth h is such that either $h \geq n/(2 \cdot t')$ or $t' \geq 3 \cdot t - 1$. For $t = o(n/\log n)$, only in the latter case can the tree decomposition have depth $O(\log n)$ (i.e., be balanced), and hence the width must increase by a constant factor.

The graph G_t^n. The graph G_t^n is defined as follows. Let $n' = n/t$. For each $i \in \{1, \ldots, n'\}$, let V_i be a set of t nodes, such that $V_i \cap V_j = \emptyset$ for $i \neq j$ and $V = \bigcup_i V_i$. Also, let $V_0 = V_{n'+1} = \emptyset$. For each $i \in \{1, \ldots, n'\}$, each node in V_i has an edge to each other node in $V_{i-1} \cup V_i \cup V_{i+1}$ (see Figure 2). We start with a technical lemma that will help us later. Recall that $N_{\mathcal{B}}^T(i)$ denotes the set of nodes contained in bags of the subtree $T(i)$.

▶ **Lemma 10.** *For any t and n consider the graph G_t^n and a tree decomposition $(\mathcal{B}, T = (I, F))$ of G_t^n with width t' and depth h. Either $h \geq n/(2 \cdot t')$ or there exists integers i, i_1, i_2, such that $i_1 - i_2 \geq 2$ and $V_{i_1} \cup V_{i_2} \subseteq B_i$.*

Proof. Without loss of generality, we assume that for every $i \in I$, we have $N_{\mathcal{B}}^T(j) \setminus B_i \neq \emptyset$, where j ranges over the children of i in T. This is valid since, otherwise, we can simply remove the subtree $T(j)$ and still have a tree decomposition of G_t^n with the same or lower depth and width. Similarly, for every $i \in I$ that is not the root of T, we assume that $V \setminus N_{\mathcal{B}}^T(i) \neq \emptyset$, otherwise $T(i)$ is a tree decomposition of G_t^n with the same or lower depth and width.

We distinguish the following cases. Either there exists an $i \in I$ with children $j_1, \ldots, j_k$ where $k \geq 2$ (or $k \geq 3$, if i is the root of T) or not. If not, T forms a line with length at least n/t', hence the depth of T is at least $n/(2 \cdot t')$ (regardless of how T is rooted).

Otherwise, pick three nodes v_1, v_2, v_3 such that $v_1 \in N_{\mathcal{B}}^T(j_1) \setminus B_i$, $v_2 \in N_{\mathcal{B}}^T(j_2) \setminus B_i$ and $v_3 \in V \setminus N_{\mathcal{B}}^T(i)$ (or $v_3 \in N_{\mathcal{B}}^T(j_3) \setminus B_i$ in case i is the root of T). Let i', j', k' be such that $v_1 \in V_{i'}$, $v_2 \in V_{j'}$ and $v_3 \in V_{k'}$. We assume w.l.o.g. that $i' \leq j' \leq k'$. We have that $j' - i' \geq 2$ (resp. $k' - j' \geq 2$), since otherwise there is an edge between v_1 and v_2 (resp. v_2 and v_3) and hence a path between them that does not intersect with nodes in B_i, contradicting Lemma 2. Also, there exist integers i_1 and i_2 such that $i' < i_2 < j' < i_1 < k'$ (hence, $i_1 - i_2 \geq 2$) and such that $V_{i_2} \cup V_{i_1} \subseteq B_i$, since otherwise, there is at least one node in V_{i_2},

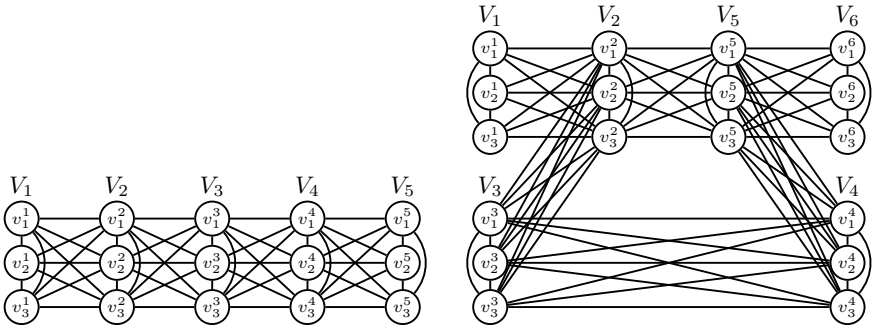

Figure 2 The graphs G_3^{15} (left) and $G(2,5,3,18)$ (right).

for $i' < i_2 < j'$ (resp. in $j' < i_1 < k'$) which is not in B_i and thus there is a path $P : v_1 \rightsquigarrow v_2$ (resp. $P : v_2 \rightsquigarrow v_3$) that does not contain nodes in B_i, again contradicting Lemma 2.

The desired result follows. ◀

We now show that the treewidth of G_t^n is at most $2 \cdot t - 1$.

▶ **Lemma 11.** *For any t and n, the treewidth of G_t^n is $\leq 2 \cdot t - 1$.*

Proof. Construct a tree $T = (I, F)$, where $I = \{1, \ldots, n' - 1\}$ and $F = \{(i, i+1)\}_{i \in I \setminus \{n'-1\}}$. Construct the set of bags $\mathcal{B} = \{B_i = V_i \cup V_{i+1}\}_{i \in I}$. It is easy to see that $(\mathcal{B}, T)$ is a tree decomposition of G_t^n. ◀

The graph $G(i,j,t,n)$. Given numbers i and j and the graph G_t^n, for some t and n, let $G(i,j,t,n)$ be the graph similar to G_t^n, except that it also has an edge between each pair of nodes in $V_i \times V_j$ (see Figure 2). Next, we show that the treewidth of $G(i,j,t,n)$ is a factor larger than the one of G_t^n.

▶ **Lemma 12.** *For any i, j, t, n, where $j - i \geq 2$, the treewidth of $G(i,j,t,n)$ is $\geq 3 \cdot t - 1$.*

Proof. We construct a minor $G'(i,j,t,n)$ of $G(i,j,t,n)$ by contracting every edge (v_k^i, v_k^j) for $1 \leq k \leq t$. Observe that the clique $K_{3 \cdot t}$ is a minor of $G'(i,j,t,n)$. Since $K_{3 \cdot t}$ has treewidth $3 \cdot t - 1$ and treewidth is monotonic under graph minors [6, Lemma 16], it follows that the treewidth of $G(i,j,t,n)$ is a least $3 \cdot t - 1$. ◀

We next show that any tree decomposition of G_t^n has either large depth or large width.

▶ **Lemma 13.** *For any n and t, consider a tree decomposition $(\mathcal{B}, T = (I, F))$ of G_t^n of width t' and depth h. Either $h \geq n/(2 \cdot t')$ or $t' \geq 3 \cdot t - 1$.*

Proof. If $h \geq n/(2 \cdot t')$, we are done. Otherwise, by Lemma 10, there exist integers i, i_1, i_2 such that $i_1 - i_2 \geq 2$ and $V_{i_1} \cup V_{i_2} \subseteq B_i$. Then, $(\mathcal{B}, T)$ is also a tree decomposition of $G(i_2, i_1, t, n)$, since each edge between V_{i_1} and V_{i_2} has both endpoints in B_i. By Lemma 12, the width of $(\mathcal{B}, T)$ is at least $3 \cdot t - 1$. ◀

We conclude with the proof of Theorem 4.

Proof of Theorem 4. The bound on t implies that any tree decomposition of width $t' < 3 \cdot t - 1$ and depth h, such that $h \geq n/(2 \cdot t')$ cannot be balanced. The statement then follows directly from Lemma 11 and Lemma 13. ◀

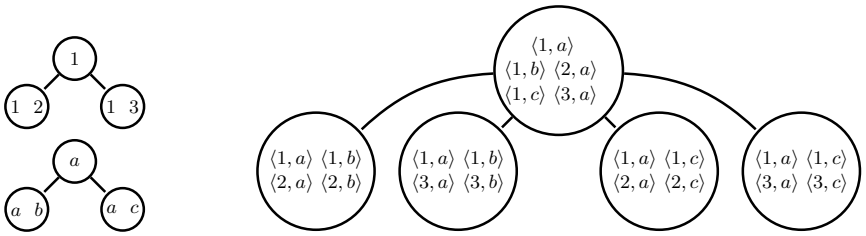

Figure 3 The product tree-decomposition given tree decompositions of the two constituent graphs.

4 Applications to verification on product graphs

Here we discuss three applications of (α, β) tree decompositions in the analysis of product graphs of two constant-treewidth graphs, when the specification language is either w.r.t. semiring, mean-payoff or initial credit for energy properties. Product graphs arise frequently in verification, for example, when analyzing the behavior of two concurrent threads, or when the verification task is expressed as language inclusion wrt two automata. In such cases, the control-flow graphs or the automata are given explicitly, and the analysis proceeds by constructing and reasoning on the product graph.

We first establish a lemma which concerns the construction of a tree decomposition of the product graph with some desired properties. Afterwards we present algorithmic improvements for the analysis of such product graphs, w.r.t. mean-payoff, semiring and energy properties.

Product graphs. Given two graphs $G_i = (V_i, E_i)$, for $1 \leq i \leq 2$, the product graph of G_1 and G_2 is defined as the graph $G = (V, E)$ where $V = V_1 \times V_2$ and E is such that $(\langle u_1, u_2\rangle, \langle v_1, v_2\rangle) \in E$ iff $(u_i, v_i) \in E_i$ for each $1 \leq i \leq 2$. Let $(\mathcal{B}_i = \{B_i^j\}_j, T_i)$ be a tree decomposition of the graph G_i, for each $1 \leq i \leq 2$. We construct a tree decomposition $(\mathcal{B}, T)$ of G as follows (see Figure 3).

1. Let j_i be the root of T_i. Recall that $N_{\mathcal{B}_i}^{T_i}(j_i)$ is the set of nodes of G_i appearing in the bags of $T_i(j_i)$. We create a node j in T, and construct the bag $B_j = \left(B_1^{j_1} \times N_{\mathcal{B}_2}^{T_2}(j_2)\right) \cup \left(N_{\mathcal{B}_1}^{T_1}(j_1) \times B_2^{j_2}\right)$.
2. If for some $1 \leq i \leq 2$, the node j_i does not have a child (i.e., j_i is also a leaf in T_i), the process terminates. Otherwise, for every $1 \leq i \leq 2$ and child j_i' of j_i, we repeat recursively for the trees $T_i(j_i')$, and make every node j' constructed in the next recursive step a child of j in T.

It is not hard to verify that $(\mathcal{B}, T)$ is a tree decomposition of G, and T is a quaternary tree. By letting each $(\mathcal{B}_i, T_i)$ be a (α, β) tree decomposition for the graph G_i, we obtain the following lemma.

▶ **Lemma 14.** *Let $G = (V, E)$ be a product graph of two constant-treewidth graphs $G_i = (V_i, E_i)$, for $1 \leq i \leq 2$, with n nodes each, and consider any integer $\lambda \geq 2$. A quaternary tree decomposition $(\mathcal{B} = \{B_j\}_j, T = (I, F))$ of G can be constructed in $O((\lambda \cdot n)^2)$ time, such that for every $j \in I$ we have $|B_j| = O(\lambda \cdot n \cdot \beta^{\mathsf{Lv}(j)})$, where $\mathsf{Lv}(j)$ is the level of node j in T.*

Proof. For each $1 \leq i \leq 2$, we use Theorem 3 to obtain a (α, β) tree decomposition $(\mathcal{B}_i = \{B_i^j\}_j, T_i)$ for the graph G_i, where $\alpha = 11 \cdot \lambda + 32$ and $\beta = 1/2 \cdot (1 + 2^{-\lambda})$. Given each $(\mathcal{B}_i, T_i)$, the algorithm constructs the quaternary tree decomposition $(\mathcal{B}, T)$ of G. The bound $|B_j| = O(\lambda \cdot n \cdot \beta^{\mathsf{Lv}(j)})$ follows easily from Theorem 3 and the fact that each graph has treewidth t.

We now turn our attention to the time complexity, and show that the construction of $(\mathcal{B}, T)$

requires $O((\lambda \cdot n)^2)$ time. It is easy to see that the construction requires time proportional to the total sizes of all bags in $\mathcal{B}$, hence it suffices to argue that $\sum_j |B_j| = O((\lambda \cdot n)^2)$. We have

$$\sum_j |B_j| \leq 2 \cdot \sum_{j_1} \sum_{j_2} \left|B_1^{j_1}\right| \cdot \left|B_2^{j_2}\right| \leq \sum_{j_1} \left(|B_1^{j_1}| \cdot \sum_{j_2} |B_2^{j_2}| \right) \tag{1}$$

$$\leq \sum_{j_1} \left(\alpha \cdot (t+1) \cdot \sum_{j_2} \alpha \cdot (t+1) \right) = O((\lambda \cdot n)^2) \tag{2}$$

since $t = O(1)$. ◀

Note that the root bag of the product tree decomposition has $\Theta(n)$ nodes, and thus the width of the constructed tree decomposition is $\Theta(n)$ (as bags below the root decrease in size). We remark that this is unavoidable in general – e.g., two straight-line graphs G_1 and G_2 of n nodes each yield a product graph G that is a grid and thus has treewidth $n-1$. The crucial property of our product tree decomposition is that, although it has width $\Theta(n)$, most bags have smaller size (as stated in Lemma 14).

4.1 Mean-payoff properties

In the minimum mean payoff problem, we are given a weighted graph $G = (V, E)$, a weight function $\mathsf{wt}\colon E \to \mathbb{Z}$, and a starting node $u \in V$. The decision problem is to compute whether

$$\inf_{\substack{u=x_1,x_2,\ldots \\ (x_i,x_{i+1})\in E}} \lim_{k\to\infty} \inf \frac{\sum_{i=1}^{k-1} \mathsf{wt}(x_i, x_{i+1})}{k} \geq 0$$

In words, we are interested in deciding whether the smallest weight-average among all infinite paths that originate in u is non-negative. Here we focus on the case where G is the product of two constant treewidth graphs $G_i = (V_i, E_i)$, for $1 \leq i \leq 2$, with n nodes each.

Solution on tree decompositions. The problem reduces to determining whether G contains a negative cycle w.r.t. wt that is reachable from u. We outline an existing algorithm for detecting negative cycles on G given a tree decomposition $(\mathcal{B} = \{B_i\}_i, T = (I, F))$ of G. We refer to [23] for details. We use a single data structure, which is a map $\mathsf{D} : \bigcup_i (B_i \times B_i) \to \mathbb{Q} \cup \{\infty\}$. Initially, $\mathsf{D}(u, v) = \mathsf{wt}(u, v)$ for each $(u, v) \in E$, and $\mathsf{D}(u, v) = \infty$ if $(u, v) \notin E$. Given a set of nodes $X \subseteq V$, we denote by D_X the restriction of D to the set X. We traverse T bottom-up and for each encountered node i, we compute all-pairs distances in the weighted graph $G^i = (B_i, B_i \times B_i)$ w.r.t. the weight function D_{B_i}, using the Floyd-Warshall algorithm. If a negative cycle is detected, we report that G has a negative cycle w.r.t. wt and stop. Otherwise, for every pair of nodes $u, v \in B_i$, we update the entry $\mathsf{D}(u, v)$ with the distance $d(u, v)$ in G_i, and proceed with the next node of T.

Algorithm for product graphs. Now consider that $G = (V, E)$ is the product of two constant-treewidth graphs $G_i = (V_i, E_i)$ with n nodes each. We choose $\lambda = 3$ and use Lemma 14 to construct a tree decomposition $(\mathcal{B}, T)$ of G for $\alpha = 11 \cdot \lambda + 32 = O(1)$ and $\beta = 1/2 + 2^{-\lambda} = 5/8$. Afterwards, we use the solution on tree decompositions for $(\mathcal{B}, T)$. We have the following theorem.

▶ **Theorem 15.** *Let $G = (V, E)$ be the product graph of two constant treewidth graphs $G_i = (V_i, E_i)$, for $1 \leq i \leq 2$, with n nodes each. Let $\mathsf{wt}\colon E \to \mathbb{Z}$ be a weight function on G. The decision problem of minimum mean payoff on (G, wt) can be solved in $O(n^3)$ time.*

Proof. The correctness follows directly from [23], and here we focus on the complexity. Since the tree decomposition of the product graph is quaternary, the i-th level has 4^i nodes. Since

Floyd-Warshall has cubic complexity, the complexity of the algorithm for the whole level i of the tree decomposition is bounded, up to constant factors, by the following expression.

$$4^i \cdot \left(\beta^i \cdot n\right)^3 = n^3 \cdot \left(4 \cdot \left(\frac{5}{8}\right)^3\right)^i = n^3 \cdot A^i \tag{3}$$

where $A = 5^3/2^7 < 1$. It follows that the cost decreases geometrically along the levels of T and thus the running time is dominated by the first level, which runs in time $O(n^3)$. The desired result follows. ◀

The best current complexity bound for the problem is $O(n^4)$, achieved by Karp's classic algorithm [43] (recall that G has n^2 nodes). Hence Theorem 15 yields an improvement by a factor n, asymptotically.

4.2 Semiring properties

The problem of all-pairs semiring distances is defined w.r.t. a graph $G = (V, E)$ and a weight function $\mathsf{wt} : E \to \Sigma$, where Σ is the domain of an algebraic structure $(\Sigma, \oplus, \otimes, \overline{0}, \overline{1})$ with two associative operators $\oplus$ and $\otimes$ with neutral elements $\overline{0}$ and $\overline{1}$, respectively, such that (i) $\otimes$ distributes over $\oplus$, (ii) $\oplus$ is idempotent and (iii) $\overline{0}$ absorbs in $\otimes$. The task is to determine for very pair of nodes $u, v \in V$ the semiring distance from u to v, defined as

$$d(u, v) = \bigoplus_{P=(u_1,\ldots,u_k):u \rightsquigarrow v} \bigotimes_{1 \le i < k} \mathsf{wt}(u_i, u_{i+1})$$

Here we focus on the case where G is the product of two constant treewidth graphs $G_i = (V_i, E_i)$, for $1 \le i \le 2$, with n nodes each. There are various classic algorithms for the problem, e.g. Lehmann's [47], Floyd's [34], Warshall's [54] and Kleene's [44], all of which have cubic complexity, assuming constant-time semiring operations. Since G has n^2 nodes, these algorithms take $O(n^6)$ time on G. Here we use strongly-balanced tree decompositions to obtain a solution in $\tilde{O}(n^4)$ time.

Solution on tree decompositions. We now outline an existing algorithm for computing the semiring transitive closure of G, given a tree decomposition $(\mathcal{B} = \{B_i\}_i, T = (I, F))$ of G. The algorithm is similar in spirit to the one in [23] for solving all-pairs distances. We use a single data structure, which is a map $\mathsf{D} : V \times V \to \Sigma$. Given a set of nodes $X \subseteq V$, we denote by D_X the projection of D to the set X. Initially, $\mathsf{D}(u, v) = \mathsf{wt}(u, v)$ for each $(u, v) \in E$, and $\mathsf{D}(u, v) = \overline{0}$ if $(u, v) \notin E$. The algorithm consists of two parts.

1. We traverse T twice, first bottom-up and then top-down. For each encountered node i, we use the Floyd-Warshall algorithm to solve the algebraic path problem on the graph $G^i = (B_i, B_i \times B_i)$ w.r.t. the weight function D_{B_i}. For every pair of nodes $u, v \in B_i$, we update the entry $\mathsf{D}(u, v)$ with the semiring distance $d(u, v)$ in G^i.
2. For every node u, we perform a DFS in T starting from i_u. Given a current node i, for every node $v \in B_i$, we set $\mathsf{D}(u, v) = \bigoplus_{x \in B_i} (\mathsf{D}(u, x) \otimes \mathsf{D}(x, v))$. Finally, we return the map D, which contains the all-pairs semiring distances in G, and thus the solution to the algebraic path problem.

Algorithm for product graphs. Now consider that $G = (V, E)$ is the product of two constant-treewidth graphs $G_i = (V_i, E_i)$ with n nodes each. We choose $\lambda = c + \log\log n + 1$, for some suitable constant c, and use Lemma 14 to construct a tree decomposition $(\mathcal{B}, T)$ of G for $\alpha = 11 \cdot \lambda + 32 = O(\log\log n)$ and $\beta = 1/2 + 2^{-\lambda}$. Afterwards, we use the solution on tree decompositions for $(\mathcal{B}, T)$. We have the following theorem.

▶ **Theorem 16.** *Let $G = (V, E)$ be the product graph of two constant treewidth graphs $G_i = (V_i, E_i)$, for $1 \leq i \leq 2$, with n nodes each. Let $\mathsf{wt} : E \to \Sigma$ be a weight function, where Σ is the domain of an idempotent semiring $(\Sigma, \oplus, \otimes, \overline{0}, \overline{1})$. The semiring transitive closure of (G, wt) can be solved in $\tilde{O}(n^4)$ time.*

Proof. The correctness follows directly from [23], and here we focus on the complexity. Let $m = n \cdot \log \log n$. In Step 1, the algorithm spends cubic time in each bag, and by an argument similar to the proof of Theorem 15, we have that the running time of the first step is $O(m^3) = \tilde{O}(n^3)$.

We now proceed with the analysis of Step 2, which dominates the complexity. Observe that for each node u, the algorithm spends quadratic time in each bag of the product tree decomposition. Since the tree decomposition of the product graph is quaternary, the i-th level has 4^i nodes. Hence, the complexity of this step for the whole level i is bounded, up to constant factors, by the following expression

$$4^i \cdot \left(\beta^i \cdot m\right)^2 = m^2 \cdot \left(4 \cdot \left(\frac{1 + 2^{-\lambda+1}}{2}\right)^2\right)^i = m^2 \cdot \left(1 + \frac{1}{c' \cdot \log n}\right)^i \tag{4}$$

where $c' = 2^c$. Observe that the complexity increases with the level i, and there are at most $c'' \cdot \log n$ levels, for some constant c''. We let $c' = c''$ and thus $c = \log(c'')$, and hence the complexity in the last level is bounded by

$$m^2 \cdot \left(1 + \frac{1}{c'' \cdot \log n}\right)^{c'' \cdot \log n} = O(m^2) \tag{5}$$

since $(1 + 1/x)^x \leq e$ for $x > 0$. Summing up over all $O(\log n)$ levels, Step 2 of the algorithm spends $O(m^2 \cdot \log n)$ time per node u and thus $O(n^2 \cdot m^2 \cdot \log n) = \tilde{O}(n^4)$ time for all nodes. ◀

Note that since the output has size $\Theta(n^4)$, the algorithm is optimal (up to polylog factors).

4.3 Initial credit for energy properties

In the minimum initial credit for energy problem, we are given a weighted graph $G = (V, E)$ and a weight function $\mathsf{wt}\colon E \to \mathbb{Z}$. The task is to compute for every node $u \in V$ the smallest energy value $\mathsf{E}(u) \in \mathbb{N} \cup \{\infty\}$ with the following property: there exists an infinite path $\mathcal{P} = (u_1, u_2, \dots)$ with $u_1 = u$ such that for every i we have $\mathsf{E}(u) + \sum_{j<i} \mathsf{wt}(u_j, u_{j+1}) \geq 0$. Conceptually, $\mathsf{E}(u)$ is the smallest "charge" we need to supply the system, so that starting from u it can exhibit infinite behavior without running out of energy. Conventionally, we let $\mathsf{E}(u) = \infty$ if no finite value exists.

Solution on tree decompositions. We outline an existing algorithm for the problem on G, given a tree decomposition $(\mathcal{B} = \{B_i\}_i, T = (I, F))$ of G. We first sketch the solution for arbitrary graphs G, and then explain how the solution is adapted to constant-treewidth graphs. We refer to [20] for details. The problem reduces to detecting non-positive cycles on weighted graphs of the form $(G^i = (V^i, E^i), \mathsf{wt}^i)_i$, where initially

1. $V^1 = V \cup \{s\}$, for some fresh node $s \notin V$ (intuitively, s acts as a sink in which all nodes x with $\mathsf{E}(x) = 0$ are collapsed),
2. $E^1 = E \cup (\{s\} \times V)$, and
3. $\mathsf{wt}^1(u, v) = -\mathsf{wt}(u, v)$ if $u \neq s$ and $\mathsf{wt}^1(u, v) = 0$ otherwise.

Given some $i \geq 1$, we detect a non-positive cycle on (G^i, wt^i), which determines a node x in that cycle for which $\mathsf{E}(x) = 0$. Then, we construct the weighted graph $(G^{i+1}, \mathsf{wt}^{i+1})$ where

1. $V^{i+1} = V^i \setminus \{x\}$,
2. $E^{i+1} = (E^i \setminus (V^i \times \{x\})) \cup \{(u,s) : (u,x) \in E^i\}$, and
3. $\mathsf{wt}^{i+1}(u,v) = \mathsf{wt}^i(u,v)$ if $v \neq s$ else $\mathsf{wt}^{i+1}(u,v) = \min(z, \mathsf{wt}^i(u,x))$, where $z = \mathsf{wt}^i(u,s)$ if $(u,s) \in E^i$ and $z = \infty$ otherwise.

Finally, if (G^i, wt^i) has no non-positive cycle, we compute the distance $d(u,s)$ from every $u \in V^i \setminus \{s\}$ to s in (G^i, wt^i) and assign $\mathsf{E}(u) = d(u,s)$ (at this point $d(u,s) > 0$ for each u).

We now consider the tree decomposition $(\mathcal{B} = \{B_i\}_i, T = (I,F))$ of G. The above solution is adapted as follows. First we construct the family of bags $\mathcal{B}' = \{\{s\} \cup B_i\}_i$, i.e., we insert the fresh node s in all bags. Observe that $(\mathcal{B}', T)$ is a valid tree decomposition of all G^i. Then, every step of non-positive-cycle detection, as well as the last step of computing the distances $d(u,s)$ is performed by a single bottom-up pass of T. In every encountered node i, an all-pairs distance computation is performed in the graph induced by B'_i, using the Floyd-Warshall algorithm.

Algorithm for product graphs. Now consider that $G = (V,E)$ is the product of two constant-treewidth graphs $G_i = (V_i, E_i)$ with n nodes each. We choose $\lambda = 3$ and use Lemma 14 to construct a tree decomposition $(\mathcal{B}, T)$ of G for $\alpha = 11 \cdot \lambda + 32 = O(1)$ and $\beta = 1/2 + 2^{-\lambda} = 5/8$. Afterwards, we use the solution on tree decompositions for $(\mathcal{B}, T)$. We have the following theorem.

▶ **Theorem 17.** *Let $G = (V,E)$ be the product graph of two constant treewidth graphs $G_i = (V_i, E_i)$, for $1 \leq i \leq 2$, with n nodes each. Let $\mathsf{wt}\colon E \to \mathbb{Z}$ be a weight function on G. The minimum initial credit for energy problem on (G, wt) can be solved in $O(n^5)$ time.*

Proof. The correctness follows directly from [20], and here we focus on the complexity. Similarly to Theorem 15, every bottom-up traversal of the tree decomposition $(\mathcal{B}', T)$ requires $O(n^3)$ time. Since every time we construct G^{i+1} from G^i we remove one node, we have $i \leq n^2$, i.e., there will be at most n^2 iterations of non-positive-cycle detection. Hence we make at most $n^2 + 1$ bottom-up traversals of $(\mathcal{B}', T)$, for a total running time of $O(n^5)$. The desired result follows. ◀

The best existing solution for the problem is due to [20], which has running time $O((n^2)^4) = O(n^8)$. Hence, Theorem 17 yields an improvement by a factor n^3, asymptotically.

5 Conclusion

Product graphs have numerous applications in verification, such as in the analysis of concurrent systems, as well as in language inclusion problems, which arise frequently in model checking. In this work, we have studied product graphs of two components w.r.t. three classic specification languages that arise in verification, namely semiring, mean-payoff, and initial credit for energy properties. We have studied these problems under the consideration that the components are specified as low-treewidth graphs, a property that is met by control-flow graphs of programs and has also found applications in logic, most notably due to the celebrated theorem of Courcelle for MSO. Our results show that these problems admit faster solutions than existing approaches, and in the case of semiring properties, our algorithm is optimal. At the heart of our new algorithms lies the newly introduced concept of (α, β) tree decompositions, which have a strong balancing property while suffering a small factor increase in their width. Moreover, we have shown that for balanced tree decompositions, such a factor increase in the width is generally unavoidable. Finally, we have developed an algorithm for constructing (α, β) tree decompositions efficiently for low-treewidth graphs.

References

1 Luca Aceto, A. Ingólfsdóttir, Mohammad Reza Mousavi, and M. A. Reniers. Algebraic properties for free! *Bulletin of the European Association for Theoretical Computer Science*, 99:81–103, 2009.

2 Stefan Arnborg, Jens Lagergren, and Detlef Seese. Easy problems for tree-decomposable graphs. *J. Algorithms*, 12(2):308–340, April 1991.

3 C. Baier and J-P. Katoen. *Principles of Model Checking*. MIT Press, 2008.

4 Omer Berkman and Uzi Vishkin. Finding level-ancestors in trees. *Journal of Computer and System Sciences*, 48(2), 1994.

5 Roderick Bloem, Krishnendu Chatterjee, Thomas A. Henzinger, and Barbara Jobstmann. Better quality in synthesis through quantitative objectives. In *Computer Aided Verification*, pages 140–156, 2009.

6 Hans L. Bodlaender. A partial k-arboretum of graphs with bounded treewidth. *Theoretical Computer Science*, 209(1-2):1 – 45, 1998.

7 Hans L. Bodlaender and Torben Hagerup. Parallel algorithms with optimal speedup for bounded treewidth. In Zoltán Fülöp and Ferenc Gécseg, editors, *Automata, Languages and Programming*, pages 268–279, Berlin, Heidelberg, 1995. Springer Berlin Heidelberg.

8 Mikołaj Bojańczyk and Michał Pilipczuk. Definability equals recognizability for graphs of bounded treewidth. In *Proceedings of the 31st Annual ACM/IEEE Symposium on Logic in Computer Science*, LICS '16, pages 407–416, New York, NY, USA, 2016. ACM.

9 Udi Boker, Krishnendu Chatterjee, Thomas A. Henzinger, and Orna Kupferman. Temporal specifications with accumulative values. *ACM TOCL*, 15(4):27:1–27:25, 2014.

10 Patricia Bouyer, Uli Fahrenberg, Kim G. Larsen, and Nicolas Markey. Timed automata with observers under energy constraints. In *Proceedings of the 13th ACM International Conference on Hybrid Systems: Computation and Control*, HSCC '10, pages 61–70, New York, NY, USA, 2010. ACM.

11 Patricia Bouyer, Uli Fahrenberg, Kim G. Larsen, Nicolas Markey, and Jiří Srba. Infinite runs in weighted timed automata with energy constraints. In *Formal Modeling and Analysis of Timed Systems*, volume 5215 of *Lecture Notes in Computer Science*, pages 33–47. Springer Berlin Heidelberg, 2008.

12 Patricia Bouyer, Nicolas Markey, and Raj Mohan Matteplackel. Averaging in LTL. In *CONCUR 2014*, pages 266–280, 2014.

13 Patricia Bouyer, Nicolas Markey, Mickael Randour, Kim G. Larsen, and Simon Laursen. Average-energy games. *Acta Informatica*, 55(2):91–127, Mar 2018.

14 Pavol Cerný, Krishnendu Chatterjee, Thomas A. Henzinger, Arjun Radhakrishna, and Rohit Singh. Quantitative synthesis for concurrent programs. In *Computer Aided Verification*, pages 243–259, 2011.

15 Krishnendu Chatterjee, Bhavya Choudhary, and Andreas Pavlogiannis. Optimal dyck reachability for data-dependence and alias analysis. *PACMPL*, 2(POPL):30:1–30:30, 2018.

16 Krishnendu Chatterjee, Laurent Doyen, and Thomas A. Henzinger. Expressiveness and closure properties for quantitative languages. *LMCS*, 6(3), 2010.

17 Krishnendu Chatterjee, Laurent Doyen, and Thomas A. Henzinger. Quantitative languages. *ACM TOCL*, 11(4):23, 2010.

18 Krishnendu Chatterjee, Amir Kafshdar Goharshady, Rasmus Ibsen-Jensen, and Andreas Pavlogiannis. Optimal and perfectly parallel algorithms for on-demand data-flow analysis. In Peter Müller, editor, *ETAPS (ESOP)*, volume 12075 of *Lecture Notes in Computer Science*, pages 112–140. Springer, 2020.

19 Krishnendu Chatterjee, Rasmus Ibsen-Jensen, Amir Kafshdar Goharshady, and Andreas Pavlogiannis. Algorithms for algebraic path properties in concurrent systems of constant treewidth components. *ACM Trans. Program. Lang. Syst.*, 40(3):9:1–9:43, 2018.

20 Krishnendu Chatterjee, Rasmus Ibsen-Jensen, and Andreas Pavlogiannis. Faster algorithms for quantitative verification in constant treewidth graphs. In Daniel Kroening and Corina S. Păsăreanu, editors, *Computer Aided Verification*, pages 140–157, Cham, 2015. Springer International Publishing.

21 Krishnendu Chatterjee, Rasmus Ibsen-Jensen, Andreas Pavlogiannis, and Prateesh Goyal. Faster algorithms for algebraic path properties in recursive state machines with constant treewidth. In *Proceedings of the 42Nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '15, pages 97–109, New York, NY, USA, 2015. ACM.

22 Krishnendu Chatterjee, Andreas Pavlogiannis, and Yaron Velner. Quantitative interprocedural analysis. In *Principles of Programming Languages, POPL 2015*, pages 539–551, 2015.

23 Shiva Chaudhuri and Christos D. Zaroliagis. Shortest Paths in Digraphs of Small Treewidth. Part I: Sequential Algorithms. *Algorithmica*, 27:212–226, 1995.

24 Ravi Chugh, Jan W. Voung, Ranjit Jhala, and Sorin Lerner. Dataflow analysis for concurrent programs using datarace detection. In *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI, 2008.

25 Fan RK Chung. *Separator theorems and their applications*. Universität Bonn. Institut für Ökonometrie und Operations Research, 1988.

26 Bruno Courcelle. The monadic second-order logic of graphs. i. recognizable sets of finite graphs. *Information and Computation*, 85, 1990.

27 Arnab De, Deepak D'Souza, and Rupesh Nasre. Dataflow analysis for datarace-free programs. In *Proceedings of the 20th European Conference on Programming Languages and Systems: Part of the Joint European Conferences on Theory and Practice of Software*, ESOP'11/ETAPS'11, pages 196–215. Springer-Verlag, 2011.

28 Manfred Droste, Werner Kuich, and Heiko Vogler. *Handbook of Weighted Automata*. Springer, 1st edition, 2009.

29 Manfred Droste and Ingmar Meinecke. Weighted automata and weighted MSO logics for average and long-time behaviors. *Inf. Comput.*, 220:44–59, 2012.

30 Michael Elberfeld, Andreas Jakoby, and Till Tantau. Logspace versions of the theorems of Bodlaender and Courcelle. In *Proceedings of the 2010 IEEE 51st Annual Symposium on Foundations of Computer Science*, FOCS '10, page 143–152, USA, 2010. IEEE Computer Society.

31 Uli Fahrenberg, Line Juhl, Kim G. Larsen, and Jiří Srba. Energy games in multiweighted automata. In *Proceedings of the 8th International Conference on Theoretical Aspects of Computing*, ICTAC'11, pages 95–115, Berlin, Heidelberg, 2011. Springer-Verlag.

32 Azadeh Farzan and P. Madhusudan. Causal dataflow analysis for concurrent programs. In *Proceedings of the 13th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, TACAS, 2007.

33 J. Filar and K. Vrieze. *Competitive Markov Decision Processes*. Springer-Verlag, 1997.

34 Robert W. Floyd. Algorithm 97: Shortest path. *Communications of the ACM*, 5(6):345, 1962.

35 Fedor V. Fomin, Daniel Lokshtanov, MichałPilipczuk, Saket Saurabh, and Marcin Wrochna. Fully polynomial-time parameterized computations for graphs and matrices of low treewidth. In *Proceedings of the Twenty-Eighth Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '17, pages 1419–1432, Philadelphia, PA, USA, 2017. Society for Industrial and Applied Mathematics.

36 Dirk Grunwald and Harini Srinivasan. Data flow equations for explicitly parallel programs. In *Proceedings of the Fourth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPOPP, 1993.

37 Jens Gustedt, OleA. Maehle, and JanArne Telle. The treewidth of java programs. In *Algorithm Engineering and Experiments*, volume 2409 of *Lecture Notes in Computer Science*, pages 86–97. Springer Berlin Heidelberg, 2002.

38 C. A. Hoare, Bernhard Möller, Georg Struth, and Ian Wehrman. Concurrent kleene algebra. In *Proceedings of the 20th International Conference on Concurrency Theory*, CONCUR 2009, pages 399–414, Berlin, Heidelberg, 2009. Springer-Verlag.

39 Peter Jipsen. Concurrent kleene algebra with tests. In Peter Höfner, Peter Jipsen, Wolfram Kahl, and Martin Eric Müller, editors, *Relational and Algebraic Methods in Computer Science*, pages 37–48, Cham, 2014. Springer International Publishing.

40 Line Juhl, Kim Guldstrand Larsen, and Jean-François Raskin. Theories of programming and formal methods. chapter Optimal Bounds for Multiweighted and Parametrised Energy Games, pages 244–255. Springer-Verlag, Berlin, Heidelberg, 2013.

41 Vineet Kahlon, Nishant Sinha, Erik Kruus, and Yun Zhang. Static data race detection for concurrent programs with asynchronous calls. In *Proceedings of the the 7th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering*, ESEC/FSE '09, pages 13–22, 2009.

42 Tobias Kappé, Paul Brunet, Alexandra Silva, and Fabio Zanasi. Concurrent kleene algebra: Free model and completeness. In Amal Ahmed, editor, *Programming Languages and Systems*, pages 856–882, Cham, 2018. Springer International Publishing.

43 Richard M. Karp. A characterization of the minimum cycle mean in a digraph. *Discrete Mathematics*, 1978.

44 S. C. Kleene. Representation of events in nerve nets and finite automata. *Automata Studies*, 1956.

45 Jens Knoop, Bernhard Steffen, and Jürgen Vollmer. Parallelism for free: Efficient and optimal bitvector analyses for parallel programs. *ACM Trans. Program. Lang. Syst.*, 1996.

46 J. Lagergren. Efficient parallel algorithms for tree-decomposition and related problems. In *Proceedings [1990] 31st Annual Symposium on Foundations of Computer Science*, pages 173–182 vol.1, 1990.

47 Daniel J. Lehmann. Algebraic structures for transitive closure. *Theoretical Computer Science*, 1977.

48 Jan Obdrzálek. Fast mu-calculus model checking when tree-width is bounded. In *CAV*, 2003.

49 M.L. Puterman. *Markov Decision Processes*. John Wiley and Sons, 1994.

50 Bruce A. Reed. Finding approximate separators and computing tree width quickly. In *Proceedings of the Twenty-fourth Annual ACM Symposium on Theory of Computing*, STOC '92, 1992.

51 Thomas Reps, Susan Horwitz, and Mooly Sagiv. Precise interprocedural dataflow analysis via graph reachability. In *POPL*, New York, NY, USA, 1995. ACM.

52 Neil Robertson and P.D Seymour. Graph minors. iii. planar tree-width. *Journal of Combinatorial Theory, Series B*, 36(1):49 – 64, 1984.

53 Mikkel Thorup. All Structured Programs Have Small Tree Width and Good Register Allocation. *Information and Computation*, 142(2):159 – 181, 1998.

54 Stephen Warshall. A Theorem on Boolean Matrices. *J. ACM*, 9(1):11–12, January 1962.

A Proofs of Section 3

Here we present the proofs of Section 3. The structure of this section follows the structure of Section 3.

A.1 Proofs of Section 3.1

Here we prove Lemma 5 and Lemma 6 which state the correctness and complexity of the two operations on components BorderSplit and Split, respectively.

Proof of BorderSplit**.** Here we prove Lemma 5 which concerns the correctness and complexity of the operation BorderSplit.

▶ **Lemma 5.** *Consider the operation* BorderSplit *on the set-component* $\mathcal{X} = \{\mathcal{C}_i\}_{1\leq i\leq k}$*. Let* $z = |\mathsf{size}(\mathcal{X})|$*,* $m = |\mathsf{Border}(\mathcal{X})|$*, and* $\{\mathcal{X}_i\}_{1\leq i\leq 3}$ *be the returned component set. The following assertions hold.*

1. *We have* $|\bigcup_i \mathsf{Border}(\mathcal{X}_i)| \leq m + 1$*.*
2. *For each* $1 \leq i \leq 3$*, we have* $|\mathsf{Border}(\mathcal{X}_i)| \leq \lfloor m/2 \rfloor + 1$*.*
3. *After one* $O(n)$*-time preprocessing of* T*, every call to* BorderSplit *requires* $O(z + m^2)$ *time.*

Proof. We prove each assertion separately.

1. This item holds trivially, since we remove at most one node u from $\mathcal{X}$.
2. If $m \geq 3$, by construction, we have $|A_i \cap \mathsf{Border}(\mathcal{X})| \leq \lfloor m/2 \rfloor$ for each i. In addition, $\mathsf{Border}(\mathcal{X}_i) \subseteq (A_i \cap \mathsf{Border}(\mathcal{X})) \cup \{u\}$ and thus $|\mathsf{Border}(\mathcal{X}_i)| \leq \lfloor m/2 \rfloor + 1$. Finally, if $|\mathsf{Border}(\mathcal{X})| < 3$ then the algorithm simply returns $\{\mathcal{X}\}$ and thus $|\mathsf{Border}(\mathcal{X}_i)| = m \leq \lfloor m/2 \rfloor + 1$.
3. Since T is binary, we have $m = O(z)$, thus obtaining the set $\mathsf{Border}(\mathcal{X})$ requires $O(z)$ time. Note that the LCA tree $\mathcal{T}_{\mathsf{Border}(\mathcal{C}_j)}$ contains $O(m)$ nodes, as it is binary and it has $O(m)$ leaves. It is known that T can be preprocessed in $O(n)$ time, after which LCA queries can be answered in $O(1)$ time [4]. It follows easily that $\mathcal{T}_{\mathsf{Border}(\mathcal{C})}$ can be constructed in $O(m^2)$ time, by performing $O(m^2)$ LCA queries. Determining the desired node u can easily be done in $O(m)$ time. Finally, constructing each connected component $\mathcal{C}_i$ can be easily done in $O(m)$ time.

The desired result follows. ◀

Proof of Split**.** Here we prove Lemma 6 which concerns the correctness and complexity of the operation Split.

Towards the proof of Lemma 6, we will present some simple lemmas. In the end, we will combine these lemmas in the proof of Lemma 6. Recall that Split operates on a component $\mathcal{X}$, and given a positive integer λ. We also let $z = \mathsf{size}(\mathcal{X})$ and $m = |\mathsf{Border}(\mathcal{X})|$. We start with a simple lemma that will be used frequently.

▶ **Lemma 18.** *Consider integers* $x_1, x_2, x_{3,x}4$*. If* $x_1 \geq x_2$ *and* $x_3 \geq x_4$ *and* $x_1 + x_2 \geq x_3 + x_4$*, then* $x_4 \leq x_1$*.*

Proof. The proof is trivial:

$$x_4 \leq \frac{x_3 + x_4}{2} \leq \frac{x_1 + x_2}{2} \leq x_1$$

◀

The following lemma states the properties of the first step of Split.

▶ **Lemma 19.** *At the end of first step, either $m < 10$ and $\mathcal{C}^* = \mathcal{X}$ and $\mathcal{X}_1 = \mathcal{X}_2 = \emptyset$, or each $\mathcal{X}_i$ is border-balancing.*

Proof. The claim clearly holds if $m < 10$. Now consider that $m \geq 10$, and the algorithm performs an operation BorderSplit on $\mathcal{X}$ to obtain the components $\{\mathcal{X}'_i\}_{1 \leq i \leq 3}$. If there are two border-balancing components among all $\mathcal{X}'_i$ then the claim holds by construction.

Now assume that there are no two such components, and hence the algorithm proceeds with performing the operation BorderSplit on the component $\mathcal{X}'_1$. We first argue that in the set $A = \{\mathcal{X}'_2, \mathcal{X}'_3, \mathcal{X}''_2, \mathcal{X}''_3\}$ there are at least two border-balancing components, $\mathcal{X}_a$ and $\mathcal{X}_b$. Since $\mathcal{X}'_1$ is the largest component among $\mathcal{X}'_i$, we have that $\mathsf{size}(\mathcal{X}'_3) \leq \mathsf{size}(\mathcal{X}'_2) \leq z/2$. Also, by Lemma 5, we have $|\mathsf{Border}(\mathcal{X}'_1)| \leq m/2 + 1$, thus at least one of $\mathcal{X}'_2, \mathcal{X}'_3$ has border with size at least

$$\frac{m - (m/2 + 1)}{2} = \frac{m}{4} - \frac{1}{2} \geq \frac{m}{10} - 1 \tag{6}$$

Hence at least one of $\mathcal{X}'_2, \mathcal{X}'_3$ is border-balancing. We take that component to be $\mathcal{X}_a$.

Second, we argue that at least one of $\mathcal{X}''_2, \mathcal{X}''_3$ is border-balancing. A similar argument to the previous case shows that $\mathsf{size}(\mathcal{X}''_3) \leq \mathsf{size}(\mathcal{X}''_2) \leq z/2$. Let $m' = |\mathsf{Border}(\mathcal{X}'_1)|$. Since one of $\mathcal{X}'_1, \mathcal{X}'_2$ is not border-balancing and by Lemma 5 the other has border of size at most $m/2 + 1$, we have that

$$m' \geq m - \left(\frac{m}{2} + 1\right) - \left(\frac{m}{10} - 1\right) = \frac{4}{10} \cdot m \tag{7}$$

A similar argument as before shows that at least one of $\mathcal{C}''_2, \mathcal{C}''_3$ has border with size at least

$$\frac{m' - (m'/2 + 1)}{2} = \frac{m'}{4} - \frac{1}{2} \geq \frac{m}{10} - 1/2 \tag{8}$$

and thus it is border-balancing. We take that component to be $\mathcal{X}_b$.

It remains to argue that $\mathsf{size}(\mathcal{X}_i) \leq z/2$ for each $1 \leq i \leq 2$. Since each $\mathcal{X}_a$ and $\mathcal{X}_b$ are border-balancing, the claim clearly holds after we have added $\mathcal{X}_a$ to $\mathcal{X}_1$ and $\mathcal{X}_b$ to $\mathcal{X}_2$. Assume w.l.o.g. that $\mathsf{size}(\mathcal{X}_1) \leq \mathsf{size}(\mathcal{X}_2)$, and let $x_1 = m/2 - \mathsf{size}(\mathcal{X}_1)$ and $x_2 = m/2 - \mathsf{size}(\mathcal{X}_2)$, thus $x_1 \geq x_2$. Let $x_3 = \mathsf{size}(\mathcal{X}^*)$, and x_4 be the size of any other component in $A \setminus \{\mathcal{X}^*, \mathcal{X}_a, \mathcal{X}_b\}$, and by definition $x_3 \geq x_4$. Observe that $x_1 + x_2 \geq x_3 + x_4$ since $x_1 + x_2$ is at least as large as $|A \setminus \{\mathcal{X}_a, \mathcal{X}_b\}|$. It follows by Lemma 18 that we can add the component with size x_4 to $\mathcal{X}_1$ while ensuring that the size of $\mathcal{X}_1$ stays at most $z/2$ after this operation. Similarly for adding the second component in the smaller of $\mathcal{X}_1, \mathcal{X}_2$. It follows that at the end of the second step, each $\mathcal{X}_1, \mathcal{X}_2$ is border-balancing, as desired. ◀

We now turn our attention to the second step of Split. The following lemma is straightforward.

▶ **Lemma 20.** *For all j we have that $\mathsf{size}(\mathcal{C}^{j+1}) \leq \mathsf{size}(\mathcal{C}^j)/2$ and thus $\mathsf{size}(\mathcal{C}^j) \leq z \cdot 2^{-j}$.*

Proof. The lemma follows since $\mathcal{C}^{j+1} = \mathcal{C}^j_1$, and due to Lemma 1 we have $|\mathcal{C}^j_1| \leq |\mathcal{C}^j|/2$. ◀

We now show that throughout the recursion of the second step, each of $\mathcal{X}_1, \mathcal{X}_2$ has size at most $z/2$.

▶ **Lemma 21.** *Until (but not including) the very end of the second step, we have $\mathsf{size}(\mathcal{X}_1), \mathsf{size}(\mathcal{X}_2) \leq z/2$.*

Proof. The statement holds at the beginning of the second step since by Lemma 19 each of $\mathcal{X}_1, \mathcal{X}_2$ is border-balancing. The statement also holds at the beginning of the recursion, since $\mathcal{C}$ is the largest connected component of $\mathcal{X}^*$, by Lemma 18 similarly to the proof of Lemma 19. Now consider the recursive step j. The statement follows by the induction hypothesis and the fact that $\mathcal{C}_1^j$ is the largest connected component among $\{\mathcal{C}_i^j\}_{1\leq i\leq 3}$, as above. ◀

Finally, we conclude with the proof of Lemma 6.

Proof of Lemma 6. We prove each item separately.

1. Lemma 20 and Lemma 21 together ensure that $\mathsf{size}(\mathcal{X}_i) \leq z\cdot(1/2+2^{-\lambda})$ for each $1 \leq i \leq 2$ (because we add $\mathcal{C}^\lambda$ to either $\mathcal{X}_1$ or $\mathcal{X}_2$ at the very end of the second step).
2. First assume that $m \geq 10$. By Lemma 19, we have that each of $\mathcal{X}_1, \mathcal{X}_2$ is border-balancing, thus $\mathsf{Border}(\mathcal{X}_i) \geq m/10 - 1$ for each $1 \leq i \leq 2$. Also, for each $1 \leq i \leq 2$, we have $|\mathsf{Border}(\mathcal{X}_i) \cap (\mathsf{Border}(\mathcal{X}_{3-i}) \cup \mathsf{Border}(\mathcal{X}^*)| \leq 2$. This follows from the fact that every time we apply the operation BorderSplit, we create three components and the intersection between the borders of any pair of components is either empty, or it is the singleton $\{u\}$, where the node u is defined in BorderSplit. The inequality then holds as the components $\mathcal{X}_1$, $\mathcal{X}_2$ and $\mathcal{X}^*$ are created by applying BorderSplit at most twice. Let Q_i and Q'_i denote the border of component $\mathcal{X}_i$ at the beginning of the second step, and before the recursive process of the second step, hence the previous inequality can be written as $|Q_i \cap (Q_{3-i} \cup \mathsf{Border}(\mathcal{X}^*))| \leq 2$. Thus, we have $|Q_i \setminus (Q_{3-i} \cup \mathsf{Border}(\mathcal{X}^*))| \geq m/10 - 3$. In addition, we have $Q'_i \subseteq Q_i \cup \mathsf{Border}(\mathcal{X}^*)$, and thus $Q'_i \leq 9/10 \cdot m + 3$. Finally, in each recursive call of the second step we create at most one new border node, which is the balancing separator of the connected component $\mathcal{C}^j$. Hence after λ recursive calls, we have created at most λ new border nodes. It follows that at the end of the second step, we have $|\mathsf{Border}(\mathcal{X}_i)| \leq 9/10 \cdot m + \lambda + 3$.
   Now assume that $m < 10$. Then clearly $\mathsf{Border}(\mathcal{X}_i) \leq m + \lambda \leq 9/10 \cdot m + \lambda + 3$.
3. This part is trivial, since Split removes at most $\lambda + 2$ nodes when creating the components $\mathcal{X}_1$ and $\mathcal{X}_2$.
4. The first step takes $O(z + m^2)$ time, by Lemma 5. Each recursive call of the second step takes $O(|\mathcal{X}^j|)$ time, by Lemma 1. By Lemma 20 the size of $\mathcal{X}^j$ halves in each call, hence the time for the second step is $O(|\mathcal{X}^*|) = O(z)$.

◀

A.2 Proofs of Section 3.2

Here we prove Lemma 8 and Lemma 9 which concern the correctness and complexity of Balance. We start with an auxiliary lemma, which states a bound on the size of the border of each component of the component tree constructed in the first step of Balance.

▶ **Lemma 22.** *Consider the component tree $(\mathcal{V}, \mathcal{R} = (\mathcal{J}, \mathcal{D}))$ constructed by* Balance. *For every $i \in \mathcal{J}$, we have $|\mathsf{Border}(\mathcal{X}_i)| \leq 10 \cdot (\lambda + 3)$.*

Proof. The claim clearly holds for i being the root of $\mathcal{R}$, since in that case $|\mathsf{Border}(\mathcal{X}_i)| = 0$. Now we will argue that the claim holds for any $i \in \mathcal{J}$, assuming that it holds for the parent j of i in $\mathcal{R}$.

Indeed, let $m = |\mathsf{Border}(\mathcal{X}_j)|$ and $m' = |\mathsf{Border}(\mathcal{X}_i)|$. By Lemma 6 we have that

$$m' \leq \frac{9}{10} \cdot m + \lambda + 3 \leq \frac{9}{10} \cdot 10 \cdot (\lambda + 3) + \lambda + 3 = 10 \cdot (\lambda + 3) \tag{9}$$

The desired result follows. ◀

We now turn our attention to Lemma 8 which concerns the correctness of Balance.

▶ **Lemma 8.** *$(\mathcal{B}, \mathcal{T} = (\mathcal{I}, \mathcal{F}))$ is a tree decomposition of T that has width $\leq \alpha$ and is β-balanced, for $\alpha = 11 \cdot \lambda + 32$ and $\beta = 1/2 + 2^{-\lambda}$.*

Proof. We first argue that $(\mathcal{B}, \mathcal{T})$ is a tree decomposition of T, and then that it has width $\leq \alpha$ and is β-balanced.

First, consider any edge $(u, v) \in F$, and assume w.l.o.g. that u is the parent of v. Observe that T has the (empty) connected component $\mathcal{C}$ with $\mathsf{Border}(\mathcal{C}) = \{u, v\}$. It follows easily that there exists a leaf i of $\mathcal{R}$ such that $\mathcal{C} \in \mathcal{X}_i$ and thus $\{u, v\} \subseteq \mathsf{Border}(\mathcal{X}_i)$. By construction, $u, v \in B_j$, where j is the parent of i in $\mathcal{R}$. Hence, $(\mathcal{B}, \mathcal{R})$ satisfies condition 2 of tree decompositions. Since T is connected, it follows that condition 1 is also satisfied.

We now turn our attention into showing that every node u appears in a contiguous subtree of $(\mathcal{B}, \mathcal{R})$, which will prove that $(\mathcal{B}, \mathcal{T})$ satisfies condition 3 of tree decompositions. First, observe that there exists a lowest-level node $i \in \mathcal{I}$ such that $u \in \mathcal{X}_i$, and in fact $u \in B_i$, as $u \in \mathsf{Border}(\mathcal{X}_j)$ for some child j of i. In addition, note that for every bag B_j with $u \in B_j$, we have that j is a descendant of i. Finally, consider any strict descendant j of i in $\mathcal{R}$ such that $u \notin B_j$. It follows that (i) $u \notin \mathcal{X}_j$ (by our choice of i) and (ii) $u \notin \mathsf{Border}(\mathcal{X}_j)$ (since $u \notin B_j$). It is straightforward to see that $u \notin B_{j'}$ for any descendant j' of j in $\mathcal{R}$, which concludes the condition 3.

We now turn our attention in showing that $(\mathcal{B}, \mathcal{T})$ has width $\leq \alpha$ and is β-balanced. We first argue about the width. Consider the component tree $(\mathcal{V}, \mathcal{R} = (\mathcal{J}, \mathcal{D}))$ constructed by Balance in the first step. Consider any node $i \in \mathcal{J}$ that is not a leaf, and let j_1, j_2 be the children of i in $\mathcal{R}$. By Lemma 22, we have $|\mathsf{Border}(\mathcal{X}_i)| \leq 10 \cdot (\lambda + 3)$. By Lemma 6, we have

$$|\mathsf{Border}(\mathcal{X}_{j_1}) \cup \mathsf{Border}(\mathcal{X}_{j_2})| \leq |\mathsf{Border}(\mathcal{X}_i)| + \lambda + 2 \leq 11 \cdot \lambda + 32$$

By construction, we have $B_i = \mathsf{Border}(\mathcal{X}_{j_1}) \cup \mathsf{Border}(\mathcal{X}_{j_2})$, and thus $|B_i| \leq 11 \cdot \lambda + 32$.

Finally, we show that $(\mathcal{B}, \mathcal{T})$ is β-balanced. By Lemma 6, for every $i \in \mathcal{I}$ and j child of i in $\mathcal{R}$, we have $\mathsf{size}(\mathcal{X}_j) \leq \mathsf{size}(\mathcal{X}_i) \cdot (1/2 + 2^{-\lambda})$, and since the size of the root component of $\mathcal{R}$ is $|I|$, we have $\mathsf{size}(\mathcal{X}_j) \leq |I| \cdot (1/2 + 2^{-\lambda})^{\mathsf{Lv}(j)}$. By Remark 7, we have $\mathcal{Y}^{\mathcal{T}}_{\mathcal{B}}(j) = \mathsf{size}(\mathcal{X}_j) \leq |I| \cdot (1/2 + 2^{-\lambda})^{\mathsf{Lv}(j)}$, as required.

The desired result follows. ◀

Finally, we prove Lemma 9 which captures the running time of Balance.

Proof of Lemma 9. We start with the first step of Balance. Since $\lambda \geq 2$, by Lemma 6 the size of each component decreases by at least a constant factor with each recursive call, hence the first step is executed for $O(\log |I|)$ levels. By the same lemma, and since the border of each component has size $O(\lambda)$ (by Lemma 22), each level in this recursion runs in $O(\lambda^2 \cdot |I|)$. Hence the first step runs in $O(\lambda^2 \cdot |I| \cdot \log |I|)$ time. In the second step, the tree decomposition is easily constructed in $O(\alpha \cdot |I|) = O(\lambda \cdot |I|)$ time.

The desired result follows. ◀